[AMD Public Use]



AMD64 Technology

AMD64 Architecture Programmer's Manual

Volume 2: System Programming

Publication No.	Revision	Date
24593	3.46	October 2026

Advanced Micro Devices 

[AMD Public Use]

© 2013 – 2026 Advanced Micro Devices Inc. All rights reserved.

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD products are as set forth in a signed agreement between the parties or in AMD Standard Terms and Conditions of Sale. Any unauthorized copying, alteration, distribution, transmission, performance, display or other use of this material is prohibited.

Trademarks

AMD, the AMD arrow logo, and combinations thereof, AMD Virtualization and 3DNow! are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

MMX is a trademark and Pentium is a registered trademark of Intel Corporation.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

[AMD Public Use]

Contents

Contents . . . . . **iii**

Figures . . . . . **xxi**

Tables . . . . . **xxxi**

Revision History . . . . . **xxxvi**

Preface . . . . . **l**

About This Book l
Audience l
Organization l
Conventions and Definitions li
Notational Conventions lii
Definitions liii
Registers lix
Endian Order lxii
Related Documents lxii

1 System-Programming Overview **1**

1.1 Memory Model 1
1.1.1 Memory Addressing 2
1.1.2 Memory Organization 3
1.1.3 Canonical Address Form 4
1.2 Memory Management 5
1.2.1 Segmentation 5
1.2.2 Paging 7
1.2.3 Mixing Segmentation and Paging 8
1.2.4 Real Addressing 10
1.3 Operating Modes 11
1.3.1 Long Mode 12
1.3.2 64-Bit Mode 13
1.3.3 Compatibility Mode 13
1.3.4 Legacy Modes 14
1.3.5 System Management Mode (SMM) 15
1.4 System Registers 15
1.5 System-Data Structures 17
1.6 Interrupts 19
1.7 Additional System-Programming Facilities 20
1.7.1 Hardware Multitasking 20
1.7.2 Machine Check 20
1.7.3 Software Debugging 21
1.7.4 Performance Monitoring 21

2 x86 and AMD64 Architecture Differences **23**

2.1 Operating Modes 23
2.1.1 Long Mode 23

[AMD Public Use]

2.1.2 Legacy Mode . . . 23
2.1.3 System-Management Mode . . . 24
2.2 Memory Model . . . 24
2.2.1 Memory Addressing . . . 24
2.2.2 Page Translation . . . 25
2.2.3 Segmentation . . . 26
2.3 Protection Checks . . . 27
2.4 Registers . . . 28
2.4.1 General-Purpose Registers . . . 28
2.4.2 YMM/XMM Registers . . . 28
2.4.3 Flags Register . . . 28
2.4.4 Instruction Pointer . . . 28
2.4.5 Stack Pointer . . . 29
2.4.6 Control Registers . . . 29
2.4.7 Debug Registers . . . 29
2.4.8 Extended Feature Register (EFER) . . . 29
2.4.9 Memory Type Range Registers (MTRRs) . . . 29
2.4.10 Other Model-Specific Registers (MSRs) . . . 29
2.5 Instruction Set . . . 30
2.5.1 REX Prefixes . . . 30
2.5.2 Segment-Override Prefixes in 64-Bit Mode . . . 30
2.5.3 Operands and Results . . . 30
2.5.4 Address Calculations . . . 31
2.5.5 Instructions that Reference RSP . . . 31
2.5.6 Branches . . . 32
2.5.7 NOP Instruction . . . 34
2.5.8 Single-Byte INC and DEC Instructions . . . 34
2.5.9 MOVSXD Instruction . . . 34
2.5.10 Invalid Instructions . . . 35
2.5.11 Reassigned Opcodes . . . 36
2.5.12 FXSAVE and FXRSTOR Instructions . . . 36
2.6 Interrupts and Exceptions . . . 37
2.6.1 Interrupt Descriptor Table . . . 37
2.6.2 Stack Frame Pushes . . . 37
2.6.3 Stack Switching . . . 37
2.6.4 IRET Instruction . . . 38
2.6.5 Task-Priority Register (CR8) . . . 38
2.6.6 New Exception Conditions . . . 38
2.7 Hardware Task Switching . . . 38
2.8 Long-Mode vs. Legacy-Mode Differences . . . 39

3 System Resources . . . 41

3.1 System-Control Registers . . . 41
3.1.1 CR0 Register . . . 42
3.1.2 CR2 and CR3 Registers . . . 45
3.1.3 CR4 Register . . . 47
3.1.4 Additional Control Registers in 64-Bit-Mode . . . 52
3.1.5 CR8 (Task Priority Register, TPR) . . . 52

[AMD Public Use]

3.1.6 RFLAGS Register . . . 52
3.1.7 Extended Feature Enable Register (EFER) . . . 56
3.1.8 Extended Control Registers (XCRn) . . . 60
3.2 Model-Specific Registers (MSRs) . . . 60
3.2.1 System Configuration Register (SYSCFG) . . . 61
3.2.2 System-Linkage Registers . . . 63
3.2.3 Memory-Typing Registers . . . 64
3.2.4 Debug-Extension Registers . . . 64
3.2.5 Performance-Monitoring Registers . . . 65
3.2.6 Machine-Check Registers . . . 65
3.2.7 Shadow Stack Registers . . . 66
3.2.8 Extended State Save Registers . . . 67
3.2.9 Speculation Control Registers . . . 67
3.2.10 Hardware Configuration Register (HWCR) . . . 72
3.3 Processor Feature Identification . . . 73

4 Segmented Virtual Memory . . . 75

4.1 Real Mode Segmentation . . . 75
4.2 Virtual-8086 Mode Segmentation . . . 76
4.3 Protected Mode Segmented-Memory Models . . . 76
4.3.1 Multi-Segmented Model . . . 76
4.3.2 Flat-Memory Model . . . 77
4.3.3 Segmentation in 64-Bit Mode . . . 77
4.4 Segmentation Data Structures and Registers . . . 77
4.5 Segment Selectors and Registers . . . 79
4.5.1 Segment Selectors . . . 79
4.5.2 Segment Registers . . . 81
4.5.3 Segment Registers in 64-Bit Mode . . . 82
4.6 Descriptor Tables . . . 84
4.6.1 Global Descriptor Table . . . 84
4.6.2 Global Descriptor-Table Register . . . 85
4.6.3 Local Descriptor Table . . . 86
4.6.4 Local Descriptor-Table Register . . . 87
4.6.5 Interrupt Descriptor Table . . . 89
4.6.6 Interrupt Descriptor-Table Register . . . 90
4.7 Legacy Segment Descriptors . . . 90
4.7.1 Descriptor Format . . . 90
4.7.2 Code-Segment Descriptors . . . 93
4.7.3 Data-Segment Descriptors . . . 94
4.7.4 System Descriptors . . . 96
4.7.5 Gate Descriptors . . . 97
4.8 Long-Mode Segment Descriptors . . . 99
4.8.1 Code-Segment Descriptors . . . 99
4.8.2 Data-Segment Descriptors . . . 100
4.8.3 System Descriptors . . . 101
4.8.4 Gate Descriptors . . . 103
4.8.5 Long Mode Descriptor Summary . . . 105
4.9 Segment-Protection Overview . . . 106

[AMD Public Use]

4.9.1 Privilege-Level Concept . . . 107
4.9.2 Privilege-Level Types . . . 107
4.10 Data-Access Privilege Checks . . . 108
4.10.1 Accessing Data Segments . . . 108
4.10.2 Accessing Stack Segments . . . 109
4.11 Control-Transfer Privilege Checks . . . 111
4.11.1 Direct Control Transfers . . . 111
4.11.2 Control Transfers Through Call Gates . . . 115
4.11.3 Return Control Transfers . . . 122
4.12 Limit Checks . . . 123
4.12.1 Determining Limit Violations . . . 123
4.12.2 Data Limit Checks in 64-bit Mode . . . 125
4.13 Type Checks . . . 125
4.13.1 Type Checks in Legacy and Compatibility Modes . . . 125
4.13.2 Long Mode Type Check Differences . . . 126

5 Page Translation and Protection . . . 128

5.1 Page Translation Overview . . . 129
5.1.1 Page-Translation Options . . . 131
5.1.2 Page-Translation Enable (PG) Bit . . . 131
5.1.3 Physical-Address Extensions (PAE) Bit . . . 131
5.1.4 Page-Size Extensions (PSE) Bit . . . 132
5.1.5 Page-Directory Page Size (PS) Bit . . . 132
5.2 Legacy-Mode Page Translation . . . 133
5.2.1 CR3 Register . . . 133
5.2.2 Non-PAE Paging . . . 134
5.2.3 PAE Paging . . . 137
5.3 Long-Mode Page Translation . . . 141
5.3.1 Canonical Address Form . . . 142
5.3.2 CR3 . . . 142
5.3.3 4-Kbyte Page Translation . . . 143
5.3.4 2-Mbyte Page Translation . . . 147
5.3.5 1-Gbyte Page Translation . . . 151
5.4 Page-Translation-Table Entry Fields . . . 155
5.4.1 Field Definitions . . . 155
5.4.2 Notes on Accessed and Dirty Bits . . . 159
5.5 Translation-Lookaside Buffer (TLB) . . . 159
5.5.1 Process Context Identifier . . . 160
5.5.2 Global Pages . . . 160
5.5.3 TLB Management . . . 161
5.6 Page-Protection Checks . . . 163
5.6.1 User/Supervisor (U/S) Bit . . . 164
5.6.2 Read/Write (R/W) Bit . . . 165
5.6.3 No Execute (NX) Bit . . . 165
5.6.4 Write Protect (CR0.WP) Bit . . . 165
5.6.5 Supervisor-Mode Execution Prevention (CR4.SMEP) Bit . . . 166
5.6.6 Supervisor-Mode Access Prevention(CR4.SMAP) Bit . . . 166
5.6.7 Memory Protection Keys (MPK) Bit . . . 166

[AMD Public Use]

5.7 Shadow Stack Protection . . . 167
5.7.1 Shadow Stack Accesses . . . 167
5.7.2 Shadow Stack Pages . . . 168
5.7.3 Shadow Stack Protection Checks . . . 168
5.8 Protection Across Paging Hierarchy . . . 169
5.8.1 Access to User Pages when CR0.WP=1 . . . 170
5.9 Effects of Segment Protection . . . 170
5.10 Upper Address Ignore . . . 170
5.10.1 Detecting and Enabling Upper Address Ignore . . . 170
5.10.2 Upper Address Ignore Operation . . . 171
5.10.3 Address Tag Storage . . . 171
5.10.4 Debug Breakpoint Behavior with Upper Address Ignore . . . 171

6 System Instructions . . . 172

6.1 Fast System Call and Return . . . 175
6.1.1 SYSCALL and SYSRET . . . 176
6.1.2 SYSENTER and SYSEXIT (Legacy Mode Only) . . . 178
6.1.3 SWAPGS Instruction . . . 178
6.2 System Status and Control . . . 179
6.2.1 Processor Feature Identification (CPUID) . . . 179
6.2.2 Accessing Control Registers . . . 179
6.2.3 Accessing the RFLAGS Register . . . 180
6.2.4 Accessing Debug Registers . . . 180
6.2.5 Accessing Model-Specific Registers . . . 180
6.3 Segment Register and Descriptor Register Access . . . 181
6.3.1 Accessing Segment Registers . . . 181
6.3.2 Accessing Segment Register Hidden State . . . 181
6.3.3 Accessing Descriptor-Table Registers . . . 181
6.4 Protection Checking . . . 182
6.4.1 Checking Access Rights . . . 182
6.4.2 Checking Segment Limits . . . 182
6.4.3 Checking Read/Write Rights . . . 182
6.4.4 Adjusting Access Rights . . . 183
6.5 Processor Halt . . . 183
6.6 Cache and TLB Management . . . 183
6.6.1 Cache Management . . . 183
6.6.2 TLB Invalidation . . . 184
6.7 Shadow Stack Management . . . 185

7 Memory System . . . 186

7.1 Single-Processor Memory Access Ordering . . . 189
7.1.1 Read Ordering . . . 189
7.1.2 Write Ordering . . . 190
7.1.3 Read/Write Barriers . . . 191
7.2 Multiprocessor Memory Access Ordering . . . 191
7.3 Memory Coherency and Protocol . . . 194
7.3.1 Special Coherency Considerations . . . 196
7.3.2 Access Atomicity . . . 197

[AMD Public Use]

7.3.3 Cacheable Locks and Bus Locks . . . 197
7.4 Memory Types . . . 198
7.4.1 Instruction Fetching from Uncacheable Memory . . . 201
7.4.2 Memory Barrier Interaction with Memory Types . . . 201
7.4.3 Floating Point Instructions and Memory Types . . . 202
7.5 Buffering and Combining Memory Writes . . . 204
7.5.1 Write Buffering . . . 204
7.5.2 Write Combining . . . 205
7.6 Memory Caches . . . 206
7.6.1 Cache Organization and Operation . . . 206
7.6.2 Cache Control Mechanisms . . . 209
7.6.3 Cache and Memory Management Instructions . . . 212
7.6.4 Serializing Instructions . . . 213
7.6.5 Cache and Processor Topology . . . 214
7.6.6 L3 Cache Range Reservation . . . 215
7.7 Memory-Type Range Registers . . . 219
7.7.1 MTRR Type Fields . . . 219
7.7.2 MTRRs . . . 220
7.7.3 Using MTRRs . . . 226
7.7.4 MTRRs and Page Cache Controls . . . 227
7.7.5 MTRRs in Multi-Processing Environments . . . 229
7.8 Page-Attribute Table Mechanism . . . 229
7.8.1 PAT Register . . . 229
7.8.2 PAT Indexing . . . 230
7.8.3 Identifying PAT Support . . . 231
7.8.4 PAT Accesses . . . 231
7.8.5 Combined Effect of MTRRs and PAT . . . 232
7.8.6 PATs in Multi-Processing Environments . . . 233
7.8.7 Changing Memory Type . . . 233
7.9 Memory-Mapped I/O . . . 234
7.9.1 Extended Fixed-Range MTRR Type-Field Encodings . . . 234
7.9.2 IORRs . . . 236
7.9.3 IORR Overlapping . . . 238
7.9.4 Top of Memory . . . 238
7.10 Secure Memory Encryption . . . 240
7.10.1 Determining Support for Secure Memory Encryption . . . 240
7.10.2 Enabling Memory Encryption Extensions . . . 240
7.10.3 Supported Operating Modes . . . 241
7.10.4 Page Table Support . . . 241
7.10.5 I/O Accesses . . . 242
7.10.6 Restrictions . . . 242
7.10.7 SMM Interaction . . . 242
7.10.8 Encrypt-in-Place . . . 242
7.10.9 Secure Multi-Key Memory Encryption . . . 243

8 Exceptions and Interrupts . . . 244

8.1 General Characteristics . . . 244
8.1.1 Precision . . . 244

[AMD Public Use]

8.1.2 Instruction Restart . . . 245
8.1.3 Types of Exceptions . . . 245
8.1.4 Masking External Interrupts . . . 246
8.1.5 Masking Floating-Point and Media Instructions . . . 246
8.1.6 Disabling Exceptions . . . 247
8.2 Vectors . . . 247
8.2.1 #DE—Divide-by-Zero-Error Exception (Vector 0) . . . 250
8.2.2 #DB—Debug Exception (Vector 1) . . . 250
8.2.3 NMI—Non-Maskable-Interrupt Exception (Vector 2) . . . 251
8.2.4 #BP—Breakpoint Exception (Vector 3) . . . 251
8.2.5 #OF—Overflow Exception (Vector 4) . . . 252
8.2.6 #BR—Bound-Range Exception (Vector 5) . . . 252
8.2.7 #UD—Invalid-Opcode Exception (Vector 6) . . . 252
8.2.8 #NM—Device-Not-Available Exception (Vector 7) . . . 253
8.2.9 #DF—Double-Fault Exception (Vector 8) . . . 253
8.2.10 Coprocessor-Segment-Overrun Exception (Vector 9) . . . 254
8.2.11 #TS—Invalid-TSS Exception (Vector 10) . . . 255
8.2.12 #NP—Segment-Not-Present Exception (Vector 11) . . . 256
8.2.13 #SS—Stack Exception (Vector 12) . . . 256
8.2.14 #GP—General-Protection Exception (Vector 13) . . . 257
8.2.15 #PF—Page-Fault Exception (Vector 14) . . . 258
8.2.16 #MF—x87 Floating-Point Exception-Pending (Vector 16) . . . 259
8.2.17 #AC—Alignment-Check Exception (Vector 17) . . . 260
8.2.18 #MC—Machine-Check Exception (Vector 18) . . . 261
8.2.19 #XF—SIMD Floating-Point Exception (Vector 19) . . . 261
8.2.20 #CP—Control-Protection Exception (Vector 21) . . . 262
8.2.21 #HV—Hypervisor Injection Exception (Vector 28) . . . 263
8.2.22 #VC—VMM Communication Exception (Vector 29) . . . 263
8.2.23 #SX—Security Exception (Vector 30) . . . 263
8.2.24 User-Defined Interrupts (Vectors 32–255) . . . 263
8.3 Exceptions During a Task Switch . . . 264
8.4 Error Codes . . . 264
8.4.1 Selector-Error Code . . . 264
8.4.2 Page-Fault Error Code . . . 265
8.4.3 Control-Protection Error Code . . . 265
8.5 Priorities . . . 266
8.5.1 Floating-Point Exception Priorities . . . 267
8.5.2 External Interrupt Priorities . . . 269
8.6 Real-Mode Interrupt Control Transfers . . . 270
8.7 Legacy Protected-Mode Interrupt Control Transfers . . . 272
8.7.1 Locating the Interrupt Handler . . . 273
8.7.2 Interrupt To Same Privilege . . . 274
8.7.3 Interrupt To Higher Privilege . . . 275
8.7.4 Privilege Checks . . . 276
8.7.5 Returning From Interrupt Procedures . . . 279
8.7.6 Shadow Stack Support for Interrupts and Exceptions . . . 279
8.8 Virtual-8086 Mode Interrupt Control Transfers . . . 280

[AMD Public Use]

8.8.1 Protected-Mode Handler Control Transfer . . . 281
8.8.2 Virtual-8086 Handler Control Transfer . . . 282
8.9 Long-Mode Interrupt Control Transfers . . . 283
8.9.1 Interrupt Gates and Trap Gates . . . 283
8.9.2 Locating the Interrupt Handler . . . 283
8.9.3 Interrupt Stack Frame . . . 284
8.9.4 Interrupt-Stack Table . . . 287
8.9.5 Returning From Interrupt Procedures . . . 289
8.10 Virtual Interrupts . . . 290
8.10.1 Virtual-8086 Mode Extensions . . . 290
8.10.2 Protected Mode Virtual Interrupts . . . 293
8.10.3 Effect of Instructions that Modify EFLAGS.IF . . . 293

9 Machine Check Architecture . . . 298
9.1 Introduction . . . 298
9.1.1 Reliability, Availability, and Serviceability . . . 298
9.1.2 Error Detection, Logging, and Reporting . . . 299
9.1.3 Error Recovery . . . 301
9.2 Determining Machine-Check Architecture Support . . . 302
9.3 Machine Check Architecture MSRs . . . 303
9.3.1 Global Status and Control Registers . . . 303
9.3.2 Error-Reporting Register Banks . . . 308
9.3.3 Machine-Check Architecture Extension Registers . . . 316
9.4 Initializing the Machine-Check Mechanism . . . 321
9.5 Using MCA Features . . . 322
9.5.1 Determining the Scope of Detected Errors . . . 323
9.5.2 Handling Machine Check Exceptions . . . 323
9.5.3 Reporting Corrected Errors . . . 325

10 System-Management Mode . . . 326
10.1 SMM Differences . . . 326
10.2 SMM Resources . . . 327
10.2.1 SMRAM . . . 327
10.2.2 SMBASE Register . . . 328
10.2.3 SMRAM State-Save Area . . . 329
10.2.4 SMM-Revision Identifier . . . 333
10.2.5 SMRAM Protected Areas . . . 334
10.3 Using SMM . . . 336
10.3.1 System-Management Interrupt (SMI) . . . 336
10.3.2 SMM Operating-Environment . . . 336
10.3.3 Exceptions and Interrupts . . . 337
10.3.4 Invalidating the Caches . . . 338
10.3.5 Saving Additional Processor State . . . 339
10.3.6 Operating in Protected Mode and Long Mode . . . 339
10.3.7 Auto-Halt Restart . . . 339
10.3.8 I/O Instruction Restart . . . 340
10.3.9 SMM Page Configuration Lock . . . 341
10.4 Leaving SMM . . . 342

[AMD Public Use]

10.5 Multiprocessor Considerations . . . 343

11 SSE, MMX, and x87 Programming . . . 344

11.1 Overview of System-Software Considerations . . . 344
11.2 Determining Media and x87 Feature Support . . . 344
11.3 Enabling SSE Instructions . . . 346
11.3.1 Enabling Legacy SSE Instruction Execution . . . 346
11.3.2 Enabling Extended SSE Instruction Execution . . . 346
11.3.3 SIMD Floating-Point Exception Handling . . . 347
11.4 Media and x87 Processor State . . . 347
11.4.1 SSE Execution Unit State . . . 347
11.4.2 MMX Execution Unit State . . . 348
11.4.3 x87 Execution Unit State . . . 349
11.4.4 Saving Media and x87 Execution Unit State . . . 351
11.5 XSAVE/XRSTOR Instructions . . . 357
11.5.1 CPUID Enhancements . . . 357
11.5.2 XCR0 . . . 358
11.5.3 Extended Save Area . . . 359
11.5.4 Instruction Functions . . . 359
11.5.5 ZMM/YMM States and Supported Operating Modes . . . 359
11.5.6 Extended SSE Execution State Management . . . 360
11.5.7 Saving Processor State . . . 361
11.5.8 Restoring Processor State . . . 361
11.5.9 MXCSR State Management . . . 362
11.5.10 Mode-Specific XSAVE/XRSTOR State Management . . . 362

12 Task Management . . . 368

12.1 Hardware Multitasking Overview . . . 368
12.2 Task-Management Resources . . . 369
12.2.1 TSS Selector . . . 371
12.2.2 TSS Descriptor . . . 371
12.2.3 Task Register . . . 372
12.2.4 Legacy Task-State Segment . . . 374
12.2.5 64-Bit Task State Segment . . . 378
12.2.6 Task Gate Descriptor (Legacy Mode Only) . . . 381
12.3 Hardware Task-Management in Legacy Mode . . . 381
12.3.1 Task Memory-Mapping . . . 381
12.3.2 Switching Tasks . . . 382
12.3.3 Task Switches Using Task Gates . . . 387
12.3.4 Nesting Tasks . . . 389

13 Software Debug and Performance Resources . . . 392

13.1 Software-Debug Resources . . . 392
13.1.1 Debug Registers . . . 393
13.1.2 Setting Breakpoints . . . 404
13.1.3 Using Breakpoints . . . 406
13.1.4 Single Stepping . . . 410
13.1.5 Breakpoint Instruction (INT3) . . . 410

[AMD Public Use]

13.1.6 Control-Transfer Breakpoint Features . . . 411
13.1.7 Debug Breakpoint Address Masking . . . 412
13.1.8 Last Branch Record Stack . . . 412
13.2 Performance Monitoring Counters . . . 413
13.2.1 Performance Counter MSRs . . . 413
13.2.2 Detecting Hardware Support for Performance Counters . . . 423
13.2.3 Using Performance Counters . . . 424
13.2.4 Time-Stamp Counter . . . 425
13.3 Instruction-Based Sampling . . . 426
13.3.1 IBS Fetch Sampling . . . 426
13.3.2 IBS Fetch Sampling Registers . . . 427
13.3.3 IBS Execution Sampling . . . 434
13.3.4 IBS Execution Sampling Registers . . . 435
13.3.5 IBS Filtering . . . 444
13.3.6 IBS Buffering . . . 448
13.3.7 IBS Memory Profiler . . . 453
13.4 Lightweight Profiling . . . 460
13.4.1 Overview . . . 460
13.4.2 Events and Event Records . . . 464
13.4.3 Detecting LWP . . . 473
13.4.4 LWP Registers . . . 477
13.4.5 LWP Instructions . . . 479
13.4.6 LWP Control Block . . . 483
13.4.7 XSAVE/XRSTOR . . . 493
13.4.8 Implementation Notes . . . 497

14 Processor Initialization and Long Mode Activation . . . 502

14.1 Processor Initialization . . . 502
14.1.1 Built-In Self Test (BIST) . . . 502
14.1.2 Clock Multiplier Selection . . . 503
14.1.3 Processor Initialization State . . . 503
14.1.4 Multiple Processor Initialization . . . 505
14.1.5 Fetching the First Instruction . . . 505
14.2 Hardware Configuration . . . 506
14.2.1 Processor Implementation Information . . . 506
14.2.2 Enabling Internal Caches . . . 506
14.2.3 Initializing Media and x87 Processor State . . . 506
14.2.4 Model-Specific Initialization . . . 508
14.3 Initializing Real Mode . . . 509
14.4 Initializing Protected Mode . . . 509
14.5 Initializing Long Mode . . . 510
14.6 Enabling and Activating Long Mode . . . 511
14.6.1 Activating Long Mode . . . 512
14.6.2 Consistency Checks . . . 513
14.6.3 Updating System Descriptor Table References . . . 514
14.6.4 Relocating Page-Translation Tables . . . 514
14.7 Leaving Long Mode . . . 514
14.8 Long-Mode Initialization Example . . . 515

[AMD Public Use]

15 Secure Virtual Machine. 520
15.1 The Virtual Machine Monitor 520
15.2 SVM Hardware Overview 520
15.2.1 Virtualization Support 520
15.2.2 Guest Mode 521
15.2.3 External Access Protection 521
15.2.4 Interrupt Support 521
15.2.5 Restartable Instructions 521
15.2.6 Security Support. 522
15.3 SVM Processor and Platform Extensions 522
15.4 Enabling SVM. 522
15.5 VMRUN Instruction 523
15.5.1 Basic Operation 523
15.5.2 VMSAVE and VMLOAD Instructions 528
15.6 #VMEXIT 529
15.7 Intercept Operation 530
15.7.1 State Saved on Exit 530
15.7.2 Intercepts During IDT Interrupt Delivery 531
15.7.3 EXITINTINFO Pseudo-Code 533
15.8 Decode Assists 533
15.8.1 MOV CRx/DRx Intercepts 533
15.8.2 INTn Intercepts 534
15.8.3 INVLPG and INVLPGA Intercepts 534
15.8.4 Nested and Intercepted #PF 535
15.9 Instruction Intercepts. 535
15.10 IOIO Intercepts 538
15.10.1 I/O Permissions Map 538
15.10.2 IN and OUT Behavior 539
15.10.3 (REP) OUTS and INS 540
15.11 MSR Intercepts 540
15.12 Exception Intercepts 541
15.12.1 #DE (Divide By Zero) 541
15.12.2 #DB (Debug) 542
15.12.3 Vector 2 (Reserved) 542
15.12.4 #BP (Breakpoint) 542
15.12.5 #OF (Overflow) 542
15.12.6 #BR (Bound-Range) 542
15.12.7 #UD (Invalid Opcode) 542
15.12.8 #NM (Device-Not-Available) 542
15.12.9 #DF (Double Fault) 542
15.12.10 Vector 9 (Reserved) 543
15.12.11 #TS (Invalid TSS) 543
15.12.12 #NP (Segment Not Present) 543
15.12.13 #SS (Stack Fault) 543
15.12.14 #GP (General Protection) 543
15.12.15 #PF (Page Fault). 543
15.12.16 #MF (X87 Floating Point) 543

[AMD Public Use]

15.12.17 #AC (Alignment Check) . . . 543
15.12.18 #MC (Machine Check). . . . 544
15.12.19 #XF (SIMD Floating Point) . . . 544
15.12.20 #SX (Security Exception) . . . 544
15.12.21 #CP (Control Protection) . . . 544
15.13 Interrupt Intercepts . . . 544
15.13.1 INTR Intercept . . . 544
15.13.2 NMI Intercept. . . . 544
15.13.3 SMI Intercept . . . 545
15.13.4 INIT Intercept . . . 546
15.13.5 Virtual Interrupt Intercept . . . 546
15.14 Miscellaneous Intercepts . . . 546
15.14.1 Task Switch Intercept. . . . 547
15.14.2 Ferr_Freeze Intercept . . . 547
15.14.3 Shutdown Intercept . . . 547
15.14.4 Pause Intercept Filtering . . . 547
15.14.5 Bus Lock Threshold . . . 548
15.15 VMCB State Caching . . . 548
15.15.1 VMCB Clean Bits . . . 549
15.15.2 Guidelines for Clearing VMCB Clean Bits . . . 549
15.15.3 VMCB Clean Field . . . 550
15.16 TLB Control . . . 551
15.16.1 TLB Flush . . . 551
15.16.2 Invalidate Page, Alternate ASID . . . 552
15.17 Global Interrupt Flag, STGI and CLGI Instructions . . . 553
15.18 VMMCALL Instruction . . . 553
15.19 Paged Real Mode . . . 554
15.20 Event Injection . . . 554
15.21 Interrupt and Local APIC Support . . . 555
15.21.1 Physical (INTR) Interrupt Masking in EFLAGS . . . 555
15.21.2 Virtualizing APIC TPR . . . 556
15.21.3 TPR Access in 32-Bit Mode . . . 556
15.21.4 Injecting Virtual (INTR) Interrupts . . . 556
15.21.5 Interrupt Shadows . . . 557
15.21.6 Virtual Interrupt Intercept . . . 557
15.21.7 Interrupt Masking in Local APIC . . . 558
15.21.8 INIT Support . . . 558
15.21.9 NMI Support . . . 559
15.21.10NMI Virtualization . . . 559
15.22 SMM Support . . . 560
15.22.1 Sources of SMI . . . 560
15.22.2 Response to SMI . . . 561
15.22.3 Containerizing Platform SMM . . . 561
15.23 Last Branch Record Virtualization . . . 562
15.23.1 Hardware Acceleration for LBR Virtualization . . . 563
15.23.2 LBR Virtualization CPUID Feature Detection . . . 563
15.24 External Access Protection . . . 564

[AMD Public Use]

15.24.1 Device IDs and Protection Domains . . . 564
15.24.2 Device Exclusion Vector (DEV) . . . 564
15.24.3 Access Checking . . . 565
15.24.4 DEV Capability Block . . . 566
15.24.5 DEV Register Access Mechanism . . . 567
15.24.6 DEV Control and Status Registers . . . 568
15.24.7 Unauthorized Access Logging . . . 570
15.24.8 Secure Initialization Support . . . 570
15.25 Nested Paging . . . 570
15.25.1 Nested Paging Overview . . . 570
15.25.2 Replicated State . . . 572
15.25.3 Enabling Nested Paging . . . 573
15.25.4 Nested Paging and VMRUN/#VMEXIT . . . 573
15.25.5 Nested Table Walk . . . 573
15.25.6 Guest Page Table Protection . . . 574
15.25.7 Nested Page Faults and Fault Ordering . . . 575
15.25.8 Combining Nested and Guest Attributes . . . 576
15.25.9 Combining Memory Types, MTRRs . . . 577
15.25.10 Page Splintering . . . 579
15.25.11 Legacy PAE Mode . . . 579
15.25.12 A20 Masking . . . 579
15.25.13 Detecting Nested Paging Support . . . 579
15.25.14 Guest Mode Execute Trap Extension . . . 579
15.25.15 Supervisor Shadow Stacks . . . 580
15.25.16 Page Modification Logging . . . 581
15.26 Security . . . 582
15.27 Secure Startup with SKINIT . . . 582
15.27.1 Secure Loader . . . 582
15.27.2 Secure Loader Image . . . 582
15.27.3 Secure Loader Block . . . 583
15.27.4 Trusted Platform Module . . . 584
15.27.5 System Interface, Memory Controller and I/O Hub Logic . . . 585
15.27.6 SKINIT Operation . . . 585
15.27.7 SL Abort . . . 586
15.27.8 Secure Multiprocessor Initialization . . . 586
15.28 Security Exception (#SX) . . . 588
15.29 Advanced Virtual Interrupt Controller . . . 588
15.29.1 Introduction . . . 588
15.29.2 Local APIC Register Virtualization . . . 589
15.29.3 AVIC Backing Page . . . 590
15.29.4 VMCB Changes for AVIC . . . 594
15.29.5 AVIC Memory Data Structures . . . 596
15.29.6 Interrupt Delivery . . . 601
15.29.7 AVIC CPUID Feature Flags . . . 603
15.29.8 New Processor Mechanisms . . . 603
15.29.9 AVIC Exit Codes . . . 604
15.29.10 x2AVIC . . . 607

[AMD Public Use]

15.30 SVM Related MSRs . . . 608
15.30.1 VM_CR MSR (C001_0114h) . . . 608
15.30.2 IGNNE MSR (C001_0115h) . . . 609
15.30.3 SMM_CTL MSR (C001_0116h) . . . 609
15.30.4 VM_HSAVE_PA MSR (C001_0117h) . . . 610
15.30.5 TSC Ratio MSR (C000_0104h) . . . 610
15.31 SVM-Lock . . . 611
15.31.1 SVM_KEY MSR (C001_0118h) . . . 611
15.32 SMM-Lock . . . 612
15.32.1 SmmLock Bit — HWCR[0] . . . 612
15.32.2 SMM_KEY MSR (C001_0119h) . . . 612
15.33 Nested Virtualization . . . 612
15.33.1 VMSAVE and VMLOAD Virtualization . . . 613
15.33.2 Virtual GIF . . . 613
15.34 Secure Encrypted Virtualization . . . 613
15.34.1 Determining Support for SEV . . . 614
15.34.2 Key Management . . . 614
15.34.3 Enabling SEV . . . 614
15.34.4 Supported Operating Modes . . . 615
15.34.5 SEV Encryption Behavior . . . 615
15.34.6 Page Table Support . . . 615
15.34.7 Restrictions . . . 616
15.34.8 SEV Interaction with SME . . . 616
15.34.9 Page Flush MSR . . . 618
15.34.10 SEV_STATUS MSR . . . 618
15.34.11 Virtual Transparent Encryption (VTE) . . . 619
15.35 Encrypted State (SEV-ES) . . . 620
15.35.1 Determining Support for SEV-ES . . . 620
15.35.2 Enabling SEV-ES . . . 620
15.35.3 SEV-ES Overview . . . 620
15.35.4 Types of Exits . . . 621
15.35.5 #VC Exception . . . 622
15.35.6 VMGEXIT . . . 625
15.35.7 GHCB . . . 625
15.35.8 VMRUN . . . 625
15.35.9 Automatic Exits . . . 626
15.35.10 Control Register Write Traps . . . 627
15.35.11 Interaction with SMI and #MC . . . 627
15.36 Secure Nested Paging (SEV-SNP) . . . 627
15.36.1 Determining Support for SEV-SNP . . . 628
15.36.2 Enabling SEV-SNP . . . 628
15.36.3 Reverse Map Table . . . 628
15.36.4 Initializing the RMP . . . 630
15.36.5 Hypervisor RMP Management . . . 630
15.36.6 Page Validation . . . 631
15.36.7 Virtual Machine Privilege Levels . . . 632
15.36.8 Virtual Top-of-Memory . . . 633

[AMD Public Use]

15.36.9 Reflect #VC 634
15.36.10RMP and VMPL Access Checks 634
15.36.11Large Page Management 637
15.36.12Running SNP-Active Virtual Machines 637
15.36.13Debug Registers 639
15.36.14Memory Types 639
15.36.15TLB management 639
15.36.16Interrupt Injection Restrictions 640
15.36.17Side-Channel Protection 641
15.36.18Secure TSC 645
15.36.19SEV-SNP Instruction Virtualization 645
15.36.20Allowed SEV Features 646
15.36.21Secure AVIC 646
15.36.22Segmented RMP 649
15.36.23Guest Intercept Control 652
15.36.24RMPOPT 652
15.37 Speculation Control Virtualization 654
15.37.1 SPEC_CTRL Virtualization 654
15.37.2 ERAPS Virtualization 654
15.38 Instruction-Based Sampling Virtualization 654
15.39 Performance Monitoring Counter Virtualization 656
15.39.1 Guest PMC Event Filtering 656
15.40 Guest Alternate Address Translation 657
15.40.1 GAAT Paging 657
15.40.2 GAAT CPUID 658
15.40.3 GAAT Enablement 658

16 Advanced Programmable Interrupt Controller (APIC) 659

16.1 Sources of Interrupts to the Local APIC 660
16.2 Interrupt Control 660
16.3 Local APIC 661
16.3.1 Local APIC Enable 661
16.3.2 APIC Registers 662
16.3.3 Local APIC ID 664
16.3.4 APIC Version Register 664
16.3.5 Extended APIC Feature Register 665
16.3.6 Extended APIC Control Register 665
16.4 Local Interrupts 666
16.4.1 APIC Timer Interrupt 668
16.4.2 Local Interrupts LINT0 and LINT1 670
16.4.3 Performance Monitor Counter Interrupts 670
16.4.4 Thermal Sensor Interrupts 671
16.4.5 Extended Interrupts 671
16.4.6 APIC Error Interrupts 671
16.4.7 Spurious Interrupts 672
16.5 Interprocessor Interrupts (IPI) 673
16.6 Local APIC Handling of Interrupts 677
16.6.1 Receiving System and IPI Interrupts 677

[AMD Public Use]

16.6.2 Lowest Priority Messages and Arbitration . . . 678
16.6.3 Accepting System and IPI Interrupts . . . 679
16.6.4 Selecting and Handling Interrupts . . . 682
16.7 SVM Support for Interrupts and the Local APIC . . . 684
16.7.1 Specific End of Interrupt Register . . . 684
16.7.2 Interrupt Enable Register . . . 685
16.8 x2APIC Mode . . . 686
16.8.1 x2APIC Terminology . . . 686
16.9 Detecting and Enabling x2APIC Mode . . . 686
16.9.1 Enabling x2APIC Mode . . . 688
16.10 x2APIC Initialization . . . 689
16.11 Accessing x2APIC Register . . . 689
16.11.1 x2APIC Register Address Space . . . 689
16.11.2 WRMSR / RDMSR serialization for x2APIC Register . . . 691
16.11.3 Reserved Bit Checking in x2APIC Mode . . . 691
16.12 x2APIC_ID . . . 691
16.13 x2APIC Interrupt Command Register (ICR) Operations . . . 692
16.14 Logical Destination Register . . . 693
16.15 Self_IPI Register . . . 694

17 Hardware Performance Monitoring and Control . . . 696

17.1 P-State Control . . . 696
17.2 Core Performance Boost . . . 698
17.3 Determining Processor Effective Frequency . . . 699
17.3.1 Actual Performance Frequency Clock Count (APERF) . . . 700
17.3.2 Maximum Performance Frequency Clock Count (MPERF) . . . 700
17.3.3 APERF Read-only (AperfReadOnly) . . . 701
17.3.4 MPERF Read-only (MperfReadOnly) . . . 701
17.4 Processor Feedback Interface . . . 701
17.5 Processor Core Power Reporting . . . 701
17.5.1 Processor Facilities . . . 702
17.5.2 Software Algorithm . . . 702
17.6 Collaborative Processor Performance Control . . . 703
17.6.1 Detecting Support for CPPC . . . 703
17.6.2 CPPC Model Specific Registers . . . 703
17.6.3 Enabling CPPC . . . 704
17.6.4 CPPC Performance Levels and Capabilities . . . 704
17.6.5 Using CPPC to Manage Processor Performance . . . 706
17.6.6 CPPC Feedback Mechanisms . . . 709

18 Shadow Stacks . . . 710

18.1 Shadow Stack Overview . . . 710
18.1.1 Detecting and Enabling Shadow Stack Support . . . 710
18.2 The Shadow Stack Pointer . . . 711
18.3 Shadow Stack Operation for CALL (near) and RET (near) . . . 711
18.4 Shadow Stack Operation for Far Transfers . . . 711
18.5 Far Transfer to the Same Privilege Level . . . 712
18.6 Far Transfer to Different Privilege Level . . . 712

[AMD Public Use]

18.6.1 Shadow Stack Switching . . . 712
18.6.2 Handling CS, LIP and SSP on Privilege Transistions . . . 714
18.6.3 Supervisor Shadow Stack Token . . . 714
18.6.4 Shadow Stack Token Validation for Inter-privilege CALL (far) and Interrupts/Exceptions . . . 715
18.6.5 Shadow Stack Token Validation for Inter-privilege RET and IRET . . . 716
18.7 Shadow Stack Operation for SYSCALL and SYSRET . . . 716
18.8 Shadow Stack Operation for Task Switches . . . 717
18.9 Restricting Speculative Execution of RET targets . . . 718
18.10 Shadow Stack Switching Using RSTORSSP . . . 718
18.11 Shadow Stack Management Instructions . . . 721
18.12 Shadow Stack MSRs . . . 722
18.13 XSAVES/XRSTORS . . . 723

19 Platform Quality of Service (PQOS) Extension . . . 724
19.1 PQOS Extension Overview . . . 724
19.1.1 Detecting Support for PQOS . . . 725
19.1.2 Identifying PQM and PQE Capabilities . . . 726
19.2 PQOS Monitoring (PQM) Overview . . . 726
19.2.1 Detecting PQM Monitoring Resources and Capabilities . . . 726
19.3 L3 Cache Monitoring . . . 728
19.3.1 PQM MSRs . . . 728
19.3.2 Monitoring L3 Occupancy . . . 734
19.3.3 Monitoring L3 Memory Bandwidth (MBM) . . . 735
19.4 PQOS Enforcement (PQE) . . . 742
19.4.1 Identifying support for PQE . . . 743
19.4.2 L3 Cache Allocation Enforcement (CAT) . . . 743
19.4.3 Code and Data Prioritization (CDP) . . . 745
19.4.4 AMD Bandwidth Enforcement . . . 747
19.4.5 L3 External Bandwidth Enforcement (L3BE) . . . 747
19.4.6 L3 Slow Memory Bandwidth Enforcement (L3SMBE) . . . 749
19.4.7 L3 Smart Data Cache Injection Allocation Enforcement (SDCIAE) . . . 750
19.4.8 Global Bandwidth Enforcement (GLBE) . . . 751
19.4.9 Global Slow Bandwidth Enforcement (GLSBE) . . . 754

Appendix A MSR Cross-Reference . . . 757
A.1 MSR Cross-Reference by MSR Address . . . 757
A.2 System-Software MSRs . . . 767
A.3 Memory-Typing MSRs . . . 768
A.4 Machine-Check MSRs . . . 770
A.5 Software-Debug MSRs . . . 771
A.6 Performance-Monitoring MSRs . . . 772
A.7 Secure Virtual Machine MSRs . . . 774
A.8 System Management Mode MSRs . . . 776
A.9 CPUID Name MSR Cross-Reference . . . 776
A.10 Shadow Stack MSRs . . . 777
A.11 Speculation Control MSRs . . . 777
A.12 Memory Cache MSRs . . . 777

[AMD Public Use]

A.13 Quality of Service MSRs 778
A.14 Collaborative Processor Performance Control MSRs 778
A.15 Instruction-Based Sampling MSRs 779

Appendix B VMCB Layout 780
B.1 Guest MSR Intercepts 797

Appendix C SVM Intercept Exit Codes 801

Appendix D SMM Containerization 804
D.1 SMM Containerization Pseudocode 804

Appendix E OS-Visible Workarounds 810
E.1 Erratum Process Overview 812

Index 813

[AMD Public Use]

Figures

Figure 1-1. Segmented-Memory Model 6
Figure 1-2. Flat Memory Model 7
Figure 1-3. Paged Memory Model 8
Figure 1-4. 64-Bit Flat, Paged-Memory Model 9
Figure 1-5. Real-Address Memory Model 10
Figure 1-6. Operating Modes of the AMD64 Architecture 12
Figure 1-7. System Registers 16
Figure 1-8. System-Data Structures 18
Figure 3-1. Control Register 0 (CR0) 43
Figure 3-2. Control Register 2 (CR2)—Legacy-Mode 46
Figure 3-3. Control Register 2 (CR2)—Long Mode 46
Figure 3-4. Control Register 3 (CR3)—Legacy-Mode Non-PAE Paging 46
Figure 3-5. Control Register 3 (CR3)—Legacy-Mode PAE Paging 46
Figure 3-6. Control Register 3 (CR3)—Long Mode 47
Figure 3-7. CR4 Register 48
Figure 3-8. RFLAGS Register 53
Figure 3-9. Extended Feature Enable Register (EFER) 57
Figure 3-10. AMD64 Architecture Model-Specific Registers 61
Figure 3-11. System-Configuration Register (SYSCFG) 62
Figure 3-12. XSS Register 67
Figure 3-13. SPEC_CTRL Register 68
Figure 3-14. PRED_CMD Register 70
Figure 4-1. Segmentation Data Structures 78
Figure 4-2. Segment and Descriptor-Table Registers 79
Figure 4-3. Segment Selector 79
Figure 4-4. Segment-Register Format 81
Figure 4-5. FS and GS Segment-Register Format—64-Bit Mode 83
Figure 4-6. Global and Local Descriptor-Table Access 85
Figure 4-7. GDTR and IDTR Format—Legacy Modes 85
Figure 4-8. GDTR and IDTR Format—Long Mode 86
Figure 4-9. Relationship between the LDT and GDT 87

[AMD Public Use]

Figure 4-10. LDTR Format—Legacy Mode . . . 88
Figure 4-11. LDTR Format—Long Mode . . . 88
Figure 4-12. Indexing an IDT . . . 90
Figure 4-13. Generic Segment Descriptor—Legacy Mode . . . 91
Figure 4-14. Code-Segment Descriptor—Legacy Mode . . . 93
Figure 4-15. Data-Segment Descriptor—Legacy Mode . . . 94
Figure 4-16. LDT and TSS Descriptor—Legacy/Compatibility Modes . . . 97
Figure 4-17. Call-Gate Descriptor—Legacy Mode . . . 98
Figure 4-18. Interrupt-Gate and Trap-Gate Descriptors—Legacy Mode . . . 98
Figure 4-19. Task-Gate Descriptor—Legacy Mode . . . 98
Figure 4-20. Code-Segment Descriptor—Long Mode . . . 99
Figure 4-21. Data-Segment Descriptor—Long Mode . . . 100
Figure 4-22. System-Segment Descriptor—64-Bit Mode . . . 102
Figure 4-23. Call-Gate Descriptor—Long Mode . . . 103
Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode . . . 104
Figure 4-25. Privilege-Level Relationships . . . 107
Figure 4-26. Data-Access Privilege-Check Examples . . . 109
Figure 4-27. Stack-Access Privilege-Check Examples . . . 110
Figure 4-28. Nonconforming Code-Segment Privilege-Check Examples . . . 113
Figure 4-29. Conforming Code-Segment Privilege-Check Examples . . . 114
Figure 4-30. Legacy-Mode Call-Gate Transfer Mechanism . . . 115
Figure 4-31. Long-Mode Call-Gate Access Mechanism . . . 116
Figure 4-32. Privilege-Check Examples for Call Gates . . . 118
Figure 4-33. Legacy-Mode 32-Bit Stack Switch, with Parameters . . . 120
Figure 4-34. 32-Bit Stack Switch, No Parameters—Legacy Mode . . . 120
Figure 4-35. Stack Switch—Long Mode . . . 121
Figure 5-1. Virtual to Physical Address Translation—Long Mode . . . 130
Figure 5-2. Control Register 3 (CR3)—Non-PAE Paging Legacy-Mode . . . 134
Figure 5-3. Control Register 3 (CR3)—PAE Paging Legacy-Mode . . . 134
Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode . . . 135
Figure 5-5. 4-Kbyte PDE—Non-PAE Paging Legacy-Mode . . . 135
Figure 5-6. 4-Kbyte PTE—Non-PAE Paging Legacy-Mode . . . 136
Figure 5-7. 4-Mbyte Page Translation—Non-PAE Paging Legacy-Mode . . . 137

[AMD Public Use]

Figure 5-8. 4-Mbyte PDE—Non-PAE Paging Legacy-Mode . . . 137
Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode . . . 138
Figure 5-10. 4-Kbyte PDPE—PAE Paging Legacy-Mode . . . 139
Figure 5-11. 4-Kbyte PDE—PAE Paging Legacy-Mode . . . 139
Figure 5-12. 4-Kbyte PTE—PAE Paging Legacy-Mode . . . 139
Figure 5-13. 2-Mbyte PAE Page Translation—Legacy Mode . . . 140
Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode . . . 141
Figure 5-15. 2-Mbyte PDE—PAE Paging Legacy-Mode . . . 141
Figure 5-16. Control Register 3 (CR3)—Long Mode . . . 142
Figure 5-17. 4-Kbyte Page Translation—Long Mode 4-Level Paging . . . 144
Figure 5-18. 4-Kbyte Page Translation—Long Mode 5-Level Paging . . . 145
Figure 5-19. 4-Kbyte PML5E—Long Mode . . . 146
Figure 5-20. 4-Kbyte PML4E—Long Mode . . . 146
Figure 5-21. 4-Kbyte PDPE—Long Mode . . . 146
Figure 5-22. 4-Kbyte PDE—Long Mode . . . 146
Figure 5-23. 4-Kbyte PTE—Long Mode . . . 147
Figure 5-24. 2-Mbyte Page Translation—Long Mode 4-Level Paging . . . 148
Figure 5-25. 2-Mbyte Page Translation—Long Mode 5-Level Paging . . . 149
Figure 5-26. 2-Mbyte PML5E—Long Mode . . . 150
Figure 5-27. 2-Mbyte PML4E—Long Mode . . . 150
Figure 5-28. 2-Mbyte PDPE—Long Mode . . . 150
Figure 5-29. 2-Mbyte PDE—Long Mode . . . 151
Figure 5-30. 1-Gbyte Page Translation—Long Mode 4-Level Paging . . . 152
Figure 5-31. 1-Gbyte Page Translation—Long Mode 5-Level Paging . . . 153
Figure 5-32. 1-Gbyte PML5E—Long Mode . . . 154
Figure 5-33. 1-Gbyte PML4E—Long Mode . . . 154
Figure 5-34. 1-Gbyte PDPE—Long Mode . . . 155
Figure 5-35. PKRU Register . . . 167
Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs . . . 177
Figure 6-2. SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP MSRs . . . 178
Figure 6-3. FLUSH_CMD Register . . . 184
Figure 7-1. Processor and Memory System . . . 187
Figure 7-2. MOESDIF State Transitions . . . 195

[AMD Public Use]

Figure 7-3. Cache Organization Example 207
Figure 7-4. L3 Range Reserve Base Address Register (L3RangeReserveBaseAddr) 216
Figure 7-5. L3 Range Reserve Maximum Address Register (L3RangeReserveMaxAddr) 216
Figure 7-6. L3 Range Reserve Way Mask Register (L3RangeReserveWayMask) 217
Figure 7-7. MTRR Mapping of Physical Memory 221
Figure 7-8. Fixed-Range MTRR 222
Figure 7-9. MTRRphysBase*n* Register 224
Figure 7-10. MTRRphysMask*n* Register 224
Figure 7-11. MTRRdefType Register Format 226
Figure 7-12. MTRR Capability Register Format 227
Figure 7-13. PAT Register 229
Figure 7-14. Extended MTRR Type-Field Format (Fixed-Range MTRRs) 235
Figure 7-15. IORRBase*n* Register 237
Figure 7-16. IORRMask*n* Register 238
Figure 7-17. Memory Organization Using Top-of-Memory Registers 239
Figure 7-18. Top-of-Memory Registers (TOP_MEM, TOP_MEM2) 239
Figure 7-19. Encrypted Memory Accesses 241
Figure 8-1. Control Register 2 (CR2) 259
Figure 8-2. Selector Error Code 264
Figure 8-3. Page-Fault Error Code 265
Figure 8-4. Control-Protection Error Code 266
Figure 8-5. Task Priority Register (CR8) 269
Figure 8-6. Real-Mode Interrupt Control Transfer 271
Figure 8-7. Stack After Interrupt in Real Mode 272
Figure 8-8. Protected-Mode Interrupt Control Transfer 274
Figure 8-9. Stack After Interrupt to Same Privilege Level 275
Figure 8-10. Stack After Interrupt to Higher Privilege 276
Figure 8-11. Privilege-Check Examples for Interrupts 278
Figure 8-12. Stack After Virtual-8086 Mode Interrupt to Protected Mode 282
Figure 8-13. Long-Mode Interrupt Control Transfer 284
Figure 8-14. Long-Mode Stack After Interrupt—Same Privilege 286
Figure 8-15. Long-Mode Stack After Interrupt—Higher Privilege 287
Figure 8-16. Long-Mode IST Mechanism 288

[AMD Public Use]

Figure 9-1. MCG_CAP Register . . . 304
Figure 9-2. MCG_STATUS Register . . . 305
Figure 9-3. MCG_CTL Register . . . 306
Figure 9-4. MCA_INTR_CFG Register . . . 306
Figure 9-5. CPU Watchdog Timer Register Format . . . 307
Figure 9-6. MCi_CTL Register . . . 310
Figure 9-7. MC*i*_STATUS Register . . . 311
Figure 9-8. MCi_MISC1 Addressing . . . 314
Figure 9-9. Miscellaneous Information Register (Thresholding Register Format) . . . 315
Figure 9-10. Address Mapping of Registers in MCAX Bank . . . 317
Figure 9-11. MCA_IPID Register . . . 319
Figure 9-12. MCA_DESTAT Register . . . 321
Figure 10-1. Default SMRAM Memory Map . . . 328
Figure 10-2. SMBASE Register . . . 328
Figure 10-3. SMM-Revision Identifier . . . 334
Figure 10-4. SMM_ADDR Register Format . . . 335
Figure 10-5. SMM_MASK Register Format . . . 335
Figure 10-6. I/O Instruction Restart Dword . . . 341
Figure 11-1. SSE Execution Unit State . . . 348
Figure 11-2. MMX Execution Unit State . . . 349
Figure 11-3. x87 Execution Unit State . . . 351
Figure 11-4. FSAVE/FNSAVE Image (32-Bit, Protected Mode) . . . 353
Figure 11-5. FSAVE/FNSAVE Image (32-Bit, Real/Virtual-8086 Modes) . . . 354
Figure 11-6. FSAVE/FNSAVE Image (16-Bit, Protected Mode) . . . 355
Figure 11-7. FSAVE/FNSAVE Image (16-Bit, Real/Virtual-8086 Modes) . . . 356
Figure 11-8. XCR0 Format . . . 358
Figure 11-9. FXSAVE and FXRSTOR Image (64-bit Mode) . . . 363
Figure 11-10. FXSAVE and FXRSTOR Image (Non-64-bit Mode) . . . 364
Figure 12-1. Task-Management Resources . . . 370
Figure 12-2. Task-Segment Selector . . . 371
Figure 12-3. TR Format, Legacy Mode . . . 372
Figure 12-4. TR Format, Long Mode . . . 373
Figure 12-5. Relationship between the TSS and GDT . . . 373

[AMD Public Use]

Figure 12-6. Legacy 32-bit TSS . . . 375
Figure 12-7. I/O-Permission Bitmap Example . . . 378
Figure 12-8. Long Mode TSS Format . . . 380
Figure 12-9. Task-Gate Descriptor, Legacy Mode Only . . . 381
Figure 12-10. Privilege-Check Examples for Task Gates . . . 389
Figure 13-1. Address-Breakpoint Registers (DR0–DR3) . . . 394
Figure 13-2. Debug-Status Register (DR6) . . . 395
Figure 13-3. Debug-Control Register (DR7) . . . 397
Figure 13-4. Debug-Control MSR (DebugCtl) . . . 400
Figure 13-5. Control-Transfer Recording MSRs . . . 401
Figure 13-6. Debug-Extension-Control MSR (DebugExtnCtl) . . . 402
Figure 13-7. LastBranchStackFromIp MSR . . . 402
Figure 13-8. LastBranchStackToIp MSR . . . 403
Figure 13-9. LastBranchStackSelect MSR . . . 404
Figure 13-10. Performance Counter Format . . . 414
Figure 13-11. Core Performance Event-Select Register (PerfEvtSel*n*) . . . 416
Figure 13-12. Performance Counter Global Control (PerfCntrGlobalCtl) . . . 419
Figure 13-13. Performance Counter Global Status Register (PerfCntrGlobalStatus) . . . 419
Figure 13-14. Performance Counter Global Status Set Register (PerfCntrGlobalStatusSet) . . . 420
Figure 13-15. Performance Counter Global Status Clear Register (PerfCntrGlobalStatusClr) . . . 421
Figure 13-16. Northbridge Performance Event-Select Register (NB_PerfEvtSel*n*) . . . 422
Figure 13-17. L2 Cache Performance Event-Select Register (L2I_PerfEvtSel*n*) . . . 423
Figure 13-18. Time-Stamp Counter (TSC) . . . 425
Figure 13-19. IBS Fetch Control Register (IbsFetchCtl) . . . 428
Figure 13-20. IBS Fetch Linear Address Register (IbsFetchLinAd) . . . 430
Figure 13-21. IBS Fetch Physical Address Register (IbsFetchPhysAd) . . . 431
Figure 13-22. IBS Fetch Extended Control Register (IbsFetchExtdCtl) . . . 431
Figure 13-23. IBS Fetch Control Register 2 (IbsFetchCtl2) . . . 432
Figure 13-24. IBS Control Register (IbsCtl) . . . 433
Figure 13-25. IBS Execution Control Register (IbsOpCtl) . . . 436
Figure 13-26. IBS Op Linear Address Register (IbsOpRip) . . . 437
Figure 13-27. IBS Op Data 1 Register (IbsOpData1) . . . 438
Figure 13-28. IBS Op Data 3 Register (IbsOpData3) . . . 440

[AMD Public Use]

Figure 13-29. IBS Data Cache Linear Address Register (IbsDcLinAd) 442
Figure 13-30. IBS Data Cache Physical Address Register (IbsDcPhysAd) 443
Figure 13-31. IBS Branch Target Address Register (IbsBrTarget) 443
Figure 13-32. IBS Execution Control Register 2 (IbsOpCtl2) 444
Figure 13-33. Generic Event Record 465
Figure 13-34. Programmed Value Sample Event Record 466
Figure 13-35. Instructions Retired Event Record 467
Figure 13-36. Branch Retired Event Record 468
Figure 13-37. DCache Miss Event Record 470
Figure 13-38. CPU Clocks not Halted Event Record 471
Figure 13-39. CPU Reference Clocks not Halted Event Record 472
Figure 13-40. Programmed Event Record 473
Figure 13-41. LWP_CFG—Lightweight Profiling Features MSR 478
Figure 13-42. LWPCB—Lightweight Profiling Control Block 485
Figure 13-43. LWPCB Flags 489
Figure 13-44. LWPCB Filters 490
Figure 13-45. XSAVE Area for LWP 494
Figure 15-1. EXITINTINFO 532
Figure 15-2. EXITINFO1 for IOIO Intercept 539
Figure 15-3. EXITINFO1 for SMI Intercept 546
Figure 15-4. VMCB Clean Field 550
Figure 15-5. EVENTINJ Field in the VMCB 555
Figure 15-6. Host Bridge DMA Checking. 566
Figure 15-7. Format of DEV_OP Register (in PCI Config Space) 567
Figure 15-8. Format of DEV_CAP Register (in PCI Config Space) 568
Figure 15-9. Format of DEV_BASE_HI[n] Registers. 569
Figure 15-10. Format of DEV_BASE_LO[n] Registers 569
Figure 15-11. Format of DEV_MAP[n] Registers 569
Figure 15-12. Address Translation with Single-Level Paging 571
Figure 15-13. Address Translation with Nested Paging 572
Figure 15-14. SLB Example Layout 584
Figure 15-15. vAPIC Backing Page Access 590
Figure 15-16. Virtual APIC Task Priority Register Synchronization 594

[AMD Public Use]

Figure 15-17. Physical APIC ID Table Entry 597
Figure 15-18. Physical APIC Table in Memory 598
Figure 15-19. Logical APIC ID Table Entry 599
Figure 15-20. Logical APIC ID Table Format, Flat Mode 600
Figure 15-21. Logical APIC ID Table Format, Cluster Mode. 601
Figure 15-22. Doorbell Register, MSR C001_011Bh 604
Figure 15-23. EXITINFO1 for AVIC_INCOMPLETE_IPI 605
Figure 15-24. EXITINFO2 for AVIC_INCOMPLETE_IPI 605
Figure 15-25. EXITINFO1 for AVIC_NOACCEL. 607
Figure 15-26. EXITINFO2 for AVIC_NOACCEL. 607
Figure 15-27. VM_CR MSR (C001_0114h). 608
Figure 15-28. SMM_CTL MSR (C001_0116h) 609
Figure 15-29. TSC Ratio MSR (C000_0104h) 611
Figure 15-30. Guest Data Request. 616
Figure 15-31. EXAMPLE #VC FLOW. 624
Figure 15-32. Secure AVIC Control MSR 648
Figure 15-33. Segmented RMP Configuration Register 650
Figure 15-34. RMP Segment Table Entry 651
Figure 15-35. RMPOPT_BASE MSR. 653
Figure 16-1. Block Diagram of a Typical APIC Implementation 659
Figure 16-2. APIC Base Address Register (MSR 0000_001Bh). 662
Figure 16-3. APIC ID Register (APIC Offset 20h). 664
Figure 16-4. APIC Version Register (APIC Offset 30h). 664
Figure 16-5. Extended APIC Feature Register (APIC Offset 400h) 665
Figure 16-6. Extended APIC Control Register (APIC Offset 410h). 666
Figure 16-7. General Local Vector Table Register Format. 667
Figure 16-8. APIC Timer Local Vector Table Register (APIC Offset 320h) 668
Figure 16-9. Timer Current Count Register (APIC Offset 390h) 669
Figure 16-10. Timer Initial Count Register (APIC Offset 380h) 669
Figure 16-11. Divide Configuration Register (APIC Offset 3E0h). 669
Figure 16-12. Local Interrupt 0/1 (LINT0/1) Local Vector Table Register (APIC Offset 350h/360h) 670
Figure 16-13. Performance Monitor Counter Local Vector Table Register

[AMD Public Use]

(APIC Offset 340h) . . . 670
Figure 16-14. Thermal Sensor Local Vector Table Register (APIC Offset 330h) . . . 671
Figure 16-15. APIC Error Local Vector Table Register (APIC Offset 370h) . . . 671
Figure 16-16. APIC Error Status Register (APIC Offset 280h) . . . 672
Figure 16-17. Spurious Interrupt Register (APIC Offset F0h) . . . 673
Figure 16-18. Interrupt Command Register (APIC Offset 300h–310h) . . . 674
Figure 16-19. Remote Read Register (APIC Offset C0h) . . . 676
Figure 16-20. Logical Destination Register (APIC Offset D0h) . . . 677
Figure 16-21. Destination Format Register (APIC Offset E0h) . . . 678
Figure 16-22. Arbitration Priority Register (APIC Offset 90h) . . . 679
Figure 16-23. Interrupt Request Register (APIC Offset 200h–270h) . . . 680
Figure 16-24. In Service Register (APIC Offset 100h–170h) . . . 681
Figure 16-25. Trigger Mode Register (APIC Offset 180h–1F0h) . . . 681
Figure 16-26. Task Priority Register (APIC Offset 80h) . . . 682
Figure 16-27. Processor Priority Register (APIC Offset A0h) . . . 683
Figure 16-28. End of Interrupt (APIC Offset B0h) . . . 684
Figure 16-29. Specific End of Interrupt (APIC Offset 420h) . . . 685
Figure 16-30. Interrupt Enable Register (APIC Offset 480h–4F0h) . . . 685
Figure 16-31. APIC Base Address Register (MSR 01Bh) support for x2APIC . . . 687
Figure 16-32. APIC State Transitions . . . 688
Figure 16-33. x2APIC_ID Register (MSR 802h) . . . 692
Figure 16-34. Interrupt Command Register (MSR 830h) . . . 693
Figure 16-35. Logical Destination (MSR 80Dh) . . . 694
Figure 16-36. Self_IPI Register (MSR 83Fh) . . . 695
Figure 17-1. P-State Current Limit Register (MSR C001_0061h) . . . 697
Figure 17-2. P-State Control Register (MSR C001_0062h) . . . 697
Figure 17-3. P-State Status Register (MSR C001_0063h) . . . 698
Figure 17-4. Core Performance Boost (MSR C001_0015h) . . . 699
Figure 17-5. Actual Performance Frequency Count (MSR0000_00E8h) . . . 700
Figure 17-6. Max Performance Frequency Count (MSR0000_00E7h) . . . 700
Figure 17-7. APREF Read Only (MSR C000_00E8h) . . . 701
Figure 17-8. MPERF Read Only (MSR C000_00E7h) . . . 701
Figure 17-9. CPPC_ENABLE . . . 704

[AMD Public Use]

Figure 17-10. CPPC_CAPABILITY_1 . . . 705
Figure 17-11. CPPC_CAPABILITY_2 . . . 706
Figure 17-12. CPPC_REQUEST . . . 707
Figure 17-13. CPPC_REQUEST_2 . . . 708
Figure 17-14. CPPC_STATUS . . . 709
Figure 18-1. Interrupt Shadow Stack Table (ISST) . . . 713
Figure 18-2. Shadow Stacks and Supervisor Shadow Stack Tokens . . . 715
Figure 18-3. Supervisor Shadow Stack Token . . . 715
Figure 18-4. Shadow Stack Restore Token . . . 718
Figure 18-5. Previous SSP Token . . . 719
Figure 18-6. RSTORSSP and SAVEPREVSSP Operation . . . 720
Figure 19-1. PQR_ASSOC . . . 729
Figure 19-2. QM_EVTSEL . . . 730
Figure 19-3. QM_CTR . . . 731
Figure 19-4. PQR_PLZA_ASSOC . . . 732
Figure 19-5. QOS_EVT_CFG_n . . . 738
Figure 19-6. L3_QOS_ABMC_CFG . . . 739
Figure 19-7. L3_QOS_ABMC_DSC . . . 740
Figure 19-8. L3_QOS_EXT_CFG . . . 741
Figure 19-9. L3_MASK_n . . . 744
Figure 19-10. L3_QOS_CFG1 . . . 746
Figure 19-11. L3QOS_BW_CONTROL_n . . . 748
Figure 19-12. L3QOS_SMBW_CONTROL_n . . . 750
Figure 19-13. GLBE Control Domains within the System . . . 751
Figure 19-14. L3QOS_GL_BW_CONTROL_n . . . 752
Figure 19-15. L3QOS_GL_SLOWBW_CONTROL_n . . . 755

[AMD Public Use]

Tables

Table 1-1. Operating Modes . . . 11
Table 2-1. Instructions That Reference RSP . . . 31
Table 2-2. 64-Bit Mode Near Branches, Default 64-Bit Operand Size . . . 32
Table 2-3. Invalid Instructions in 64-Bit Mode . . . 35
Table 2-4. Invalid Instructions in Long Mode . . . 36
Table 2-5. Opcodes Reassigned in 64-Bit Mode . . . 36
Table 2-6. Differences Between Long Mode and Legacy Mode . . . 39
Table 4-1. Segment Registers . . . 81
Table 4-2. Descriptor Types . . . 92
Table 4-3. Code-Segment Descriptor Types . . . 94
Table 4-4. Data-Segment Descriptor Types . . . 95
Table 4-5. System-Segment Descriptor Types (S=0)—Legacy Mode . . . 96
Table 4-6. System-Segment Descriptor Types—Long Mode . . . 101
Table 4-7. Descriptor-Entry Field Changes in Long Mode . . . 105
Table 4-8. Segment Limit Checks in 64-Bit Mode . . . 125
Table 5-1. Supported Paging Alternatives (CR0.PG=1) . . . 131
Table 5-2. Physical-Page Protection, CR0.WP=0 . . . 169
Table 5-3. Effect of CR0.WP=1 on Supervisor Page Access . . . 170
Table 6-1. System Management Instructions . . . 172
Table 7-1. Atomic Read-Modify-Write Operation . . . 198
Table 7-2. Memory Access by Memory Type . . . 200
Table 7-3. Caching Policy by Memory Type . . . 201
Table 7-4. Memory Access Ordering Rules . . . 203
Table 7-5. AMD64 Architecture Cache-Operating Modes . . . 210
Table 7-6. MTRR Type Field Encodings . . . 220
Table 7-7. Fixed-Range MTRR Address Ranges . . . 222
Table 7-8. Combined MTRR and Page-Level Memory Type with Unmodified PAT MSR . . . 228
Table 7-9. PAT Type Encodings . . . 230
Table 7-10. PAT-Register PA-Field Indexing . . . 231
Table 7-11. Combined Effect of MTRR and PAT Memory Types . . . 232
Table 7-12. Serialization Requirements for Changing Memory Types . . . 234
Table 7-13. Extended Fixed-Range MTRR Type Encodings . . . 236
Table 8-1. Interrupt Vector Source and Cause . . . 248
Table 8-2. Interrupt Vector Classification . . . 249

[AMD Public Use]

Table 8-3. Double-Fault Exception Conditions . . . 254
Table 8-4. Invalid-TSS Exception Conditions . . . 255
Table 8-5. Stack Exception Error Codes . . . 257
Table 8-6. General-Protection Exception Conditions . . . 257
Table 8-7. Data-Type Alignment . . . 260
Table 8-8. Control-Protection Error Codes . . . 266
Table 8-9. Simultaneous Interrupt Priorities . . . 266
Table 8-10. Simultaneous Floating-Point Exception Priorities . . . 268
Table 8-11. Virtual-8086 Mode Interrupt Mechanisms . . . 281
Table 8-12. Effect of Instructions that Modify the IF Bit . . . 294
Table 9-1. CPU Watchdog Timer Time Base . . . 307
Table 9-2. CPU Watchdog Timer Count Select . . . 307
Table 9-3. Error Logging Priorities . . . 309
Table 9-4. Error Scope . . . 323
Table 10-1. AMD64 Architecture SMM State-Save Area . . . 329
Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture) . . . 332
Table 10-3. SMM Register Initialization . . . 336
Table 11-5. SSE Subsets – CPUID Feature Identifiers . . . 345
Table 11-6. Extended Save Area Format . . . 359
Table 11-7. XRSTOR Hardware-Specified Initial Values . . . 362
Table 11-8. Deriving FSAVE Tag Field from FXSAVE Tag Field . . . 367
Table 12-1. Effects of Task Nesting . . . 390
Table 13-1. Breakpoint-Setting Examples . . . 405
Table 13-2. Breakpoint Location by Condition . . . 407
Table 13-3. Types of Branch Records . . . 413
Table 13-4. Host/Guest Only Bits . . . 417
Table 13-5. Count Control Using CNT_MASK and INV . . . 417
Table 13-6. Operating-System Mode and User Mode Bits . . . 418
Table 13-7. Supported IBS Filtering Criteria . . . 447
Table 13-8. Fetch IBS Record . . . 452
Table 13-9. Execution IBS Record . . . 452
Table 13-10. Timestamp Counter (TSC) Record . . . 453
Table 13-11. IBS Execution Sampling and Memory Profiler Register Equivalents . . . 453
Table 13-12. EventId Values . . . 465
Table 13-13. Lightweight Profiling CPUID Values . . . 475
Table 13-14. LWPCB—Lightweight Profiling Control Block Fields . . . 486

[AMD Public Use]

Table 13-15. LWPCB Filters Fields . . . 491
Table 13-16. XSAVE Area for LWP Fields . . . 495
Table 14-1. Initial Processor State . . . 503
Table 14-2. Initial State of Segment-Register Attributes . . . 505
Table 14-3. x87 Floating-Point State Initialization . . . 507
Table 14-4. Processor Operating Modes . . . 512
Table 14-5. Long-Mode Consistency Checks . . . 513
Table 15-1. Guest Exception or Interrupt Types . . . 532
Table 15-2. EXITINFO1 for MOV CR*x* . . . *534*
Table 15-3. EXITINFO1 for MOV DR*x* . . . *534*
Table 15-4. EXITINFO1 for INTn . . . 534
Table 15-5. EXITINFO1 for INVLPG . . . 534
Table 15-6. Guest Instruction Bytes . . . 535
Table 15-7. Instruction Intercepts . . . 536
Table 15-8. MSR Ranges Covered by MSRPM . . . 540
Table 15-9. TLB Control Byte Encodings . . . 552
Table 15-10. Effect of the GIF on Interrupt Handling . . . 553
Table 15-11. Guest Exception or Interrupt Types . . . 555
Table 15-12. INIT Handling in Different Operating Modes . . . 559
Table 15-13. NMI Handling in Different Operating Modes . . . 559
Table 15-14. SMI Handling in Different Operating Modes . . . 561
Table 15-15. DEV Capability Block, Overall Layout . . . 566
Table 15-16. DEV Capability Header (DEV_HDR) (in PCI Config Space) . . . 567
Table 15-17. Encoding of Function Field in DEV_OP Register . . . 567
Table 15-18. DEV_CR Control Register . . . 568
Table 15-19. Combining Guest and Host PAT Types . . . 578
Table 15-20. Combining PAT and MTRR Types . . . 578
Table 15-21. GMET Page Configuration . . . 580
Table 15-22. Guest vAPIC Register Access Behavior . . . 591
Table 15-23. Physical APIC ID Table Entry Fields . . . 597
Table 15-24. Logical APIC ID Table Entry Fields . . . 599
Table 15-25. EXITINFO1 Fields for AVIC_INCOMPLETE_IPI . . . 605
Table 15-26. EXITINFO2 Fields for AVIC_INCOMPLETE_IPI . . . 605
Table 15-27. ID Field—IPI Delivery Failure Cause . . . 606
Table 15-28. EXITINFO1 Fields for AVIC_NOACCEL . . . 607
Table 15-29. EXITINFO2 Fields for AVIC_NOACCEL . . . 607

[AMD Public Use]

Table 15-30. Encryption Control . . . 617
Table 15-31. SEV/SME Interaction . . . 618
Table 15-32. SEV_STATUS MSR Fields . . . 619
Table 15-33. AE Exitcodes. . . . 621
Table 15-34. Fields of an RMP Entry . . . 629
Table 15-35. RMP Page Assignment Settings . . . 631
Table 15-36. VMPL Permission Mask Definition . . . 632
Table 15-37. VMPL Misconfiguration Conditions . . . 633
Table 15-38. RMP Memory Access Checks . . . 635
Table 15-39. PVALIDATE/RMPADJUST Page Size Mismatch Combinations. . . . 637
Table 15-40. VMRUN Page Checks . . . 638
Table 15-41. Non-Coherent Memory Type Conversion . . . 639
Table 16-1. Interrupt Sources for Local APIC . . . 660
Table 16-2. APIC Registers . . . 663
Table 16-3. Divide Values . . . 670
Table 16-4. Valid ICR Field Combinations . . . 676
Table 16-5. Local APIC Operating Modes. . . . 687
Table 16-6. x2APIC Register . . . 690
Table 17-1. CPPC-related MSRs . . . 703
Table 18-1. Shadow Stack Operations for Far Transfers . . . 712
Table 18-2. Shadow Stack Management Instructions . . . 721
Table 19-1. CPUID Fn000_0007_EBX_x0 Extended Feature Identifiers. . . . 725
Table 19-2. PQOS Versions . . . 725
Table 19-3. CPUID Fn0000_000F_x0 PQM Capabilities . . . 726
Table 19-4. CPUID Fn0000_000F_EDX_x1 PQM Event Identifiers . . . 727
Table 19-5. Fn8000_0020_EBX_x0 PQM Extended Feature Identifiers . . . 727
Table 19-6. QM_EVTSEL.EvtID Encodings. . . . 731
Table 19-7. Counter Size when Fn0000_000F_EAX_x1[CounterSize] = 0 . . . 732
Table 19-8. Transaction Types and EvtIDs . . . 736
Table 19-9. BMEC Events and Corresponding Attributes. . . . 737
Table 19-10. CPUID Fn0000_00010_EDX_x0 PQE Resources . . . 743
Table 19-11. CAT vs. CDP L3 Mask Register Usage . . . 746
Table A-1. MSRs of the AMD64 Architecture . . . 757
Table A-2. System-Software MSR Cross-Reference . . . 767
Table A-3. Memory-Typing MSR Cross-Reference . . . 768
Table A-4. Machine-Check MSR Cross-Reference . . . 770

[AMD Public Use]

Table A-5. Software-Debug MSR Cross-Reference . . . 772
Table A-6. Performance-Monitoring MSR Cross-Reference . . . 773
Table A-7. Secure Virtual Machine MSR Cross-Reference . . . 775
Table A-8. System Management Mode MSR Cross-Reference . . . 776
Table A-9. CPUID Namestring MSR Cross Reference . . . 776
Table A-10. Shadow Stack MSR Cross Reference . . . 777
Table A-11. Speculation Control MSRs . . . 777
Table A-12. Memory Cache MSRs . . . 777
Table A-13. Quality of Service MSRs . . . 778
Table A-14. Collaborative Processor Performance Control MSRs . . . 778
Table A-15. Instruction-Based Sampling MSR Cross-Reference . . . 779
Table B-1. VMCB Layout, Control Area . . . 780
Table B-2. VMCB Layout, State Save Area . . . 786
Table B-3. Swap Types . . . 791
Table B-4. VMSA Layout, State Save Area for SEV-ES . . . 791
Table B-5. INTERCEPT_MSR_VEC0 Layout . . . 797
Table B-6. INTERCEPT_MSR_VEC1 Layout . . . 798
Table B-7. INTERCEPT_MSR_VEC2 Layout . . . 799
Table B-8. INTERCEPT_MSR_VEC3 Layout . . . 800
Table C-1. SVM Intercept Codes . . . 801

Revision History

Date	Revision	Description
October 2026	3.46	Added new features: • Guest Alternate Address Translation (GAAT) in Section 15.40 ”Guest Alternate Address Translation.” • Paging-Write in Section 15.25.6.2 ”Paging-Write.” • Verify Paging-Write in Section 15.25.6.3 ”Verify Paging-Write.” • Guest PMC Event Filtering in Section 15.39.1 ”Guest PMC Event Filtering.” • Not-Dirty RMP bit definition in Section 15.36.3 ”Reverse Map Table.” • RMPOPT in Section 15.36.24 ”RMPOPT.” • Enhanced SMT Protection (ESMTP) in Section 15.36.17.6 ”Enhanced SMT Protection.” • New HWCR bits in Section 3.2.10 ”Hardware Configuration Register (HWCR).” DowngradeFp512ToFp256 (HWCR bit 34) ReportSameFpWidth (HWCR bit 40) • VIRT_RMPREAD MSR in Section 15.36.19 ”SEV-SNP Instruction Virtualization.” • New Platform Quality of Service features Global Bandwidth Enforcement (GLBE) in Section 19.4.8 ”Global Bandwidth Enforcement (GLBE).” Global Slow Bandwidth Enforcement (GLSBE) in Section 19.4.9 ”Global Slow Bandwidth Enforcement (GLSBE).” Privilege Level Zero Association (PLZA) in Section 19.3.1.4 ”PQR_PLZA_ASSOC.” Corrected and clarified: • Updated RSM shutdown conditions and CPL recomputation in Section 10.4 ”Leaving SMM,” including CR4.CET and CR4.PCIDE conditions. • Updates and clarifications in Section 13.3.6 ”IBS Buffering.” • Added IBS Memory Profiler condition to Section 15.36.17 ”Side-Channel Protection.” • Added discovery of Northbridge performance counters in Section 13.2.1 ”Performance Counter MSRs.” • Widened CtrID and BwSrc fields for L3_QOS_ABMC_CFG and L3_QOS_ABMC_DSC MSRs. • Added V_NMI and V_NMI_MASK to the VMCB clean-bit list in Section 15.15.3 ”VMCB Clean Field.” • Various MSR table additions and corrections in Appendix A. • Added new VMCB control area fields and SVM intercept exit codes in Appendix B and Appendix C.

[AMD Public Use]

Date	Revision	Description
July 2026	3.45	Added new features: • EFER.EnhancedTlbi in Section 3.1.7 ”Extended Feature Enable Register (EFER).” • MCA Query Registers in Section 9.3.1.6 ”MCA Query Registers.” • Physical Address Valid in MCA status registers in Section 9.3.2 ”Error-Reporting Register Banks.” • IBS Buffering in Section 13.3.6 ”IBS Buffering.” • IBS Memory Profiler in Section 13.3.7 ”IBS Memory Profiler.” • IBS MSRs (IbsFetchExtdCtl, IbsOpCtl2, and IbsFetchCtl2) in Section 13.3 ”Instruction-Based Sampling.”. • Updated IBS filter criteria and pseudo-code in Section 13.3 ”Instruction-Based Sampling.” • CPPC Performance Priority (CPPC_REQUEST_2[FloorPerf] and CPPC_CAPABILITY_1[FloorPerfCnt]) in Section 17.6 ”Collaborative Processor Performance Control.” • Shadow stack 32-byte region containment check in Section 18.6.4 ”Shadow Stack Token Validation for Inter-privilege CALL (far) and Interrupts/Exceptions.” • Guest MSR Intercepts for FRED MSRs in Appendix B.1 • APIC Extended LVT expansion from 4 to 8 entries. • Page Modification Logging in Section 15.25.16 ”Page Modification Logging.” • VMPL permission misconfiguration in Section 15.36.7 ”Virtual Machine Privilege Levels.” Corrected and clarified: • Corrected text from XSAVE/XRSTOR to XSAVES/XRSTORS in Section 18.13 ”XSAVES/XRSTORS.” • Clarified Section 15.33.1 ”VMSAVE and VMLOAD Virtualization.” • Changed Section 15.35.10 ”Control Register Write Traps” to explicitly list supported write traps. • Added a sentence to Section 15.14.3 ”Shutdown Intercept” about the VMCB control area validity after a shutdown intercept. Clarified EFER.TCE description in Section 3.1.7 ”Extended Feature Enable Register (EFER).”
March 2026	3.44	Corrected Figure 15-32 in Section 15.36.21.3, ”Secure AVIC Control MSR,” on page 648.

[AMD Public Use]

Date	Revision	Description
June 2025	3.43	Added new features: • AVX512 in Section 11, ”SSE, MMX, and x87 Programming,” on page 344 and throughout the document. • Segmented RMP in Section 15.36.22 on page 649. • Guest Intercept Control in Section 15.36.23 on page 652. • Support for up to 4096 vCPUs in x2AVIC mode in Section 15.29, ”Advanced Virtual Interrupt Controller,” on page 588. • Selective Branch Prediction Barrier (SBPB) in Section 3.2.9, ”Speculation Control Registers,” on page 67. • Enhanced Return Address Predictor Security (ERAPS) in Section 3.2.9, ”Speculation Control Registers,” on page 67 and Section 15.37.2, ”ERAPS Virtualization,” on page 654. • PREC_RET in PerfEvtSeln MSR in Section 13.2.1, ”Performance Counter MSRs,” on page 413. • 4 Kbyte coalesced IBS Fetch L1 TLB Page Size in Section 13.3.2, ”IBS Fetch Sampling Registers,” on page 427. Corrected: • #CP Error Codes in Section 8.4.3, ”Control-Protection Error Code,” on page 265. • CET_U and CET_S component identification in Section 18.13, ”XSAVES/XRSTORS,” on page 723.
March 2024	3.42	Added new features: • Replaced MOESI with MOESDIF protocol in Section 7.3, ”Memory Coherency and Protocol,” on page 194. • Indirect Branch Prediction Barrier on Entry under in Section 15.36.17, ”Side-Channel Protection,” on page 641. • Idle HLT intercept in Section 15.9, ”Instruction Intercepts,” on page 535. • Allowed SEV Features in Section 15.36.20 on page 646. • Secure AVIC in Section 15.36.21 on page 646. • Performance Monitoring Counter Virtualization in Section 15.39 on page 656.

[AMD Public Use]

Date	Revision	Description
June 2023	3.41	Added the following corrections/clarifications: Section 2.5.2, Added that an exception exists when Upper Address Ignore (UAI) is enabled. Section 15.30.4: Removed statement that "software must not attempt to read or write the host save-state area directly." Added AE Exitcode A5h, VMEXIT_BUSLOCK, to Table 15-35 Section 15.36.13: Removed sentence: "Note that the DR6 and DR7 registers are always swapped as type 'A' state for any SEV guest." Added new MSRs to Table A-1. Added new features: Bus Lock: Section 7.3.3, section 8.2.2, Figure 13.2, Figure 13.4, section 13.1.3.6, section 15.14.5, Tables B-1 and C-1. Collaborative Processor Performance Control (CPPC): Section 17.6 and Section A.14. Platform Quality of Service: Chapter 19 and Section A.13.
January 2023	3.40	Added the following corrections/clarifications: Replaced figure 5-17 with correct drawing Added paragraph at end of section 7.3.2 Corrected description for the "Valid" field in the LastBranchStackToIp MSR documentation in section 13.1.1.9 Corrected heading text for first and second columns of table 13-3 Corrected heading level of section 13.3.5, IBS Filtering Removed a duplicate paragraph from section 15.34.10

[AMD Public Use]

Date	Revision	Description
November 2022	3.39	Added corrections/clarifications to: Write combining memory type in section 7.4 Added new features: 5-level paging in section 5.3 Automatic IBRS in section 3.1.7 Section 7.10.9 Secure Multi-Key Memory Encryption CPUID disable for non-privileged software in section 3.2.10 Section 7.6.6 L3 Cache Range Reservation Last Branch Record Stack in section 13.1.1.8 Core Performance Global Control Register in section 13.2.1 Core Performance Counter Status Registers in section 13.2.1 Section 13.3.4.1 IBS Filtering Section 15.21.10 NMI Virtualization Read Only Guest Page Tables in section 15.25.5 Section 15.29.10 x2AVIC VMGEXIT Parameter in section 15.35.6 VMPL Supervisor Shadow Stack in section 15.36.7 Virtual TOM MSR in section 15.36.8 SMT Protection for SEV-SNP VMs in section 15.36.17 Section 15.36.19 SEV-SNP Instruction Virtualization Section 15.38 Instruction-Based Sampling Virtualization
November 2021	3.38	Added corrections/clarifications to: Address generation in Section 4.5.3. PCID behavior in Section 5.5. Machine Check Exception in Section 8.2.18. SEV Status MSR in Section 15.34.10. SEV_SNP in Sections 15.36.7, .10, .17. VMCB and VMSA contents in Appendix B. SVM Intercepts in Appendix C. Added new features: Chapter 5: Upper Address Ignore. Chapter 9, Section 3.2.6 and Appendix A: Machine Check Architecture Extension. Section 15.36: VMSA Register Protection, SEV-SNP VMSA Register Protection and Secure TSC.

[AMD Public Use]

Date	Revision	Description
March 2021	3.37	Added SPEC_CTRL and PRED_CMD support. Added x2APIC support. Added SMM Page Configuration Lock. Corrected the AMD64 Architecture SMM State-Save Area table. Add new section: APERF Read-only (AperfReadOnly). Updated the MSRs of the AMD64 Architecture table. Updated the Machine-Check MSR Cross-Reference table. Updated the Performance-Monitoring MSR Cross-Reference table. Updated the Secure Virtual Machine MSR Cross-Reference table. Added new section: Speculation Control MSRs.
August 2020	3.36	Added Shadow Stack support. Chapter 3: Section 3.1: Added content. Section 3.1.3: Added bit 23 and new content. Section 3.2.7: Added Shadow Stack Registers as new section 3.2.7. Chapter 5: Section 5.6: Updated content. Chapter 6: Added content to Table 6-1. Added section 6.7. Chapter 7: Table 7-3. Updated. Chapter 8: Table 8-1 and Table 8-2: Added content. Section 8.4: Updated content. Section 8.4.2: Added content. Added #CP as new section 8.2.20. Table 8-8: Added content. Added new Shadow Stack section 8.7.6. Section 8.9: Added bullet. Added new 8.9.4.1 section. Chapter 10: Updated Table 10-1. Section 10.4: Added bullet. Chapter 11: Section 11.5.2 and Figure 11-8: Updated table and figure. Section 11.5.8: Updated Table 11-3. Chapter 12: Section 12.2.2: Updated. Section 12.2.4: Updated Figure 12-6. Section 12.3.2: Updated content and added bullets. Chapter 15: Section 15.5.1: Updates. Section 15.15.3: Updated Figure 15-4. Section 15.25.6: Added bullet. Section 15.29.1. Updates. Section 15.29.4.1: Added content. Section 15.29.8.2: Added content. Added Chapter 18. Appendix A: Table A-1: Added content. Added new section A.10. Appendix B: Table B-1, Table B-2, and Table 4: Added content.
May 2020	3.35	Sections 5.6.1 and 5.6.6: Minor updates. Figure 5-16: Updated figure.

[AMD Public Use]

Date	Revision	Description
April 2020	3.34	Section 3.1.3: Updated register information. Added PCIDE and PKE registers. Updated (TCE) content. Section 5.3.2: Added Process Context Identifier register information and register figure. Section 5.3.3: Updated figure. Section 5.3.4: Updated figure. Section 5.3.5: Updated figure. Section 5.4.1: Added (MPK) register information. Section 5.5.1: Inserted Process Context Identifiers as Section 5.5.1. Section 5.5.3: Added bullets to Implicit Invalidations list. Section 5.6: Updated content. Section 8.2.15: Added bullet. Section 8.4.2: Updated register figure and added PK register information. Section 11.5.2: Updated register figure and table. Section 14.1.3: Updated table. Section 15.9: Updated table. Appendix B: Updated table. Appendix C: Updated table. Section 1.1.2: Clarification on address size support. Section 3.2.1: New feature enable bits in SYSCFG MSR. Section 7.6.5: Updated terminology. Section 7.10.6: Clarification to encrypted memory operation. Section 8.1.4: Clarification to IRET and NMI behavior. Tables 8-1 and 8-2: Added #HV exception. Inserted new 8.2.20 section for #HV exception. Section 8.4.2: Changes for SEV-SNP extension. Table 8-8: Added SEV-related exceptions. Figure 10-6: Updated I/O Restart DWORD. Section 15 and 15.1: General updates. Section 15.5.2: Relocated VMLOAD/VMSAVE documentation. Section 15.2.4 and 15.2.6: Updated content. Section 15.6: Added content. Table 15-7: Added content. Section 15.25.6: Clarification. Section 15.25.13: General clarifications. Section 15.33.1 and 15.33.2: General clarifications. Section 15.34.3, 15.34.7, and 15.34.10: Clarifications, and additions for SEV-SNP. Table 15-35: Added content. Section 15.35.8: Corrected terminology. Section 15.36: Added SEV-SNP extension documentation.

[AMD Public Use]

Date	Revision	Description
April 2020	3.33	Table A-1 and Table A-7: Added SEV-SNP related MSRs. Appendix B: Updates for SEV-SNP extension. Table C-1: Added exit code for SEV-SNP extension.
October 2019	3.32	Added UMIP, XSS, GMET, VTE, MCOMMIT, and RDPRU.
July 2019	3.31	Added CLWB and WBNOINVD details. Clarified FP error pointer save/restore behavior. Corrected description of APIC Software Enable functionality. Clarified canonical address checking behavior. Clarified fault generation. October 2026n for instructions that cross page or segment boundaries.
September 2018	3.30	Modified Section 7.4 Modified Section 7.6.4 Modified Section 8.5.2 Modified Section 9.2 Corrected Figure 9-4 Corrected Table 9-1 Modified Section 9.3.2 Corrected Figure 9-6 Corrected Table 9-4 Modified Section 14.2.3 Modified Section 14.4 Modified Section 15.6 Modified Section 15.7 Modified Section 15.34.9 Modified Section 15.34.10 Modified Section 15.35.2 Corrected Table B-4 in Appendix B
December 2017	3.29	Modified Sections 7.10.1 and 7.10.4. Modified Sections 15.34.1, 15.34.7. Added new Section 15.34.10. Modified Section 15.35.10. Modified Appendix A, Table A-7.

[AMD Public Use]

Date	Revision	Description
March 2017	3.28	Modified CR4 Register, Section 3.1.3. Removed UD2 in Table 6-1. Added new bullet in Section 7.1.1. Modified Note in Table 7-1. Added new Section 7.4.1. Clarified Self Modifying Code in Section 7.6.1. Added UD0 and UD1 instructions in Section 8.2.7. Added Instructions Retired Performance counter in Section 13.1.1. Modified Table in Section 15.34.9.
December 2016	3.27	Added Resume Flag (RF) Bit in Section 3.1.6, ”RFLAGS Register,” on page 52. Added Tom2ForceMemTypeWB in Section 3.2.1, ”System Configuration Register (SYSCFG),” on page 61. Clarified SYSCALL and SYSRET in Section 6.1.1, ”SYSCALL and SYSRET,” on page 176. Added Section 7.3.2, ”Access Atomicity,” on page 197. Updated Note b in Table 7-12 on page 234. Modified Table 8-1, “Interrupt Vector Source and Cause,” on page 248. Modified Table 8-2, “Interrupt Vector Classification,” on page 249. Added Section 8.2.22, ”#VC—VMM Communication Exception (Vector 29),” on page 263. Added a Note in Chapter 10, "System-Management Mode," on page 326. Added Section 10.5, ”Multiprocessor Considerations,” on page 343. Updated CPUID 8000_001F[EAX] and added CPUID 8000_001F[EDX] in Section 15.34.1, ”Determining Support for SEV,” on page 614. Added new Section 15.35, ”Encrypted State (SEV-ES),” on page 620. Clarified TSC Ratio MSR in Section 15.30.5 ”TSC Ratio MSR (C000_0104h)” on page 610. Modified Appendix B, ”VMCB Layout” on page 780. Added Table B-3, “Swap Types,” on page 791. Added Codes 8Fh, 90h-9Fh, and 403h in Table C-1, “SVM Intercept Codes,” on page 801.

[AMD Public Use]

Date	Revision	Description
April 2016	3.26	Clarification on loading a null selector into FS or GS added in Section 4.5.3, ”Segment Registers in 64-Bit Mode,” on page 82 Translation table diagrams corrected for definition of bit 8 in Section 5.5, ”Translation-Lookaside Buffer (TLB),” on page 159 CR0.CD implementation-dependent behavior noted in Section 7.6.2, ”Cache Control Mechanisms,” on page 209 Added clarification on IST usage in Section 8.9.4, ”Interrupt-Stack Table,” on page 287. Added new Section 7.10, ”Secure Memory Encryption,” on page 240. Added guideline for secure AP startup in Section 15.27.8, ”Secure Multiprocessor Initialization,” on page 586 Added TLB maintenance requirement for multiprocessor VM's in Section 15.29.4, ”VMCB Changes for AVIC,” on page 594. Added new Section 15.34, ”Secure Encrypted Virtualization,” on page 613
June 2015	3.25	Added new section 15.33 Nested Virtualization for coverage of VMSAVE and VMLOAD Virtualization and Virtual GIF. Various minor edits.
October 2013	3.24	Added description of Supervisor-Mode Execution Prevention. See Section 5.6.5 ”Supervisor-Mode Execution Prevention (CR4.SMEP) Bit” on page 166. Indicated the deprecation of the Processor Feedback Interface. See Section 17.4, ”Processor Feedback Interface,” on page 701. Added Section 17.5, ”Processor Core Power Reporting,” on page 701.
May 2013	3.23	Clarified guidelines for implementing cross-modifying code in the sub-section ”Cross-Modifying Code” on page 208. Added AVIC description. See Section 15.29, ”Advanced Virtual Interrupt Controller,” on page 588. Added L2I PMC architecture definition. See Section 13.2, ”Performance Monitoring Counters,” on page 413.
September 2012	3.22	Clarified processor behavior on write of EFER[LMA] bit in Section 3.1.7 ”Extended Feature Enable Register (EFER)” on page 56. Clarified difference between cold reset and warm reset in Section 9.3, ”Machine Check Architecture MSRs,” on page 303. Added information on FFXSR feature bit to Table 11-5 on page 345. Clarified SMM code responsibility to manage VMCB clean bits. See Section 15.15.2, ”Guidelines for Clearing VMCB Clean Bits,” on page 549. Added a note to Table 15-9 on page 552 to indicate that all encodings of TLB_CONTROL not defined are reserved. Corrected information concerning the assignment of logical APIC IDs in Section 16.6.1, ”Receiving System and IPI Interrupts,” on page 677.

[AMD Public Use]

Date	Revision	Description
March 2012	3.21	Added definition of processor feedback interface—frequency sensitivity monitor (See Section 17.4, ”Processor Feedback Interface,” on page 701) Added Instruction-Based Sampling in a new section of Chapter 13 (See Section 13.3, ”Instruction-Based Sampling,” on page 426.) Reworked Introduction and first section of Chapter 9, "Machine Check Architecture," on page 298 and added deferred error handling. Added description of CR4[FSGSBASE] bit. (See Section 3.1.3, ”CR4 Register,” on page 47.) Added references to the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions in discussion of FS and GS segment descriptors. (See ”FS and GS Registers in 64-Bit Mode” on page 82) Added Section 6.3.2, ”Accessing Segment Register Hidden State,” on page 181.
December 2011	3.20	Clarified description of the Cache Disable (CD) memory type in Section 7.4 ”Memory Types” on page 198. Added caveat: an overflow of either APERF or MPERF can invalidate the effective frequency calculation. See Section 17.3, ”Determining Processor Effective Frequency,” on page 699. Other minor editorial changes.
September 2011	3.19	Added XSAVEOPT to discussions on XSAVE. Corrections to discussion on multiprocessor memory access ordering in Chapter 7. Added discussion of extended core and northbridge performance counters and feature indicators to Chapter 13. Added Lightweight Profiling (LWP) to Chapter 13. Added Global Timestamp Counter, Continuous Mode to LWP description Clarification: Function of pin A20M# is only defined in real mode. Statement added to Section 1.2.4, ”Real Addressing,” on page 10. Eliminated hardware P-state references
May 2011	3.18	Added information for OSXSAVE and XSAVE features. Added Cache Topology, Pause Filter Threshold, and XSETBV information. Updated TSC ratio information. Corrected description of FXSAVE/FXRSTOR exception behavior when CR0.EM=1
June 2010	3.17	Replaced missing figures in Chapter 8, "Exceptions and Interrupts," on page 244.

[AMD Public Use]

Date	Revision	Description
June 2010	3.16	Updated information on performance monitoring counters in ”Performance-Monitoring Counter Enable (PCE)” on page 50 and 6.2.5, ”Accessing Model-Specific Registers” on page 180. Revised Table 4-1, ”Segment Registers” on page 81. Add flush by ASID information to section 15.16, ”TLB Control” on page 551. Added information on VMCB clean field to Chapter15, ”Secure Virtual Machine” on page 520 and Appendix B, ”VMCB Layout” on page 780. Added section 15.10, ”IOIO Intercepts” on page 538. Added section 15.30.5, ”TSC Ratio MSR (C000_0104h)” on page 610. Added section 17.2, ”Core Performance Boost” on page 698.
November 2009	3.15	Added section 7.5, ”Buffering and Combining Memory Writes” on page 204 Added MFENCE to list of ”Serializing Instructions” on page 213. Updated section 7.6.1, ”Cache Organization and Operation” on page 206. Updated Table 7-4, “Memory Access Ordering Rules,” on page 203 and notes. Updated 7.4, ”Memory Types” on page 198. Clarified 5.5.3, ”TLB Management” on page 161. Added ”Invalidation of Table Entry Upgrades.” on page 162. Updated ”Speculative Caching of Address Translations” on page 162. Update ”Handling of D-Bit Updates” on page 163. Revised and updated section 7.2, ”Multiprocessor Memory Access Ordering” on page 191 ff. Added information on long mode segment-limit checks in ”Extended Feature Enable Register (EFER)” on page 57table on page 57 and ”Long Mode Segment Limit Enable (LMSLE) Bit” on page 58 on page 58. Added discussion of ”Data Limit Checks in 64-bit Mode” on page 125on page 125. Updated Table 6-1, “System Management Instructions,” on page 172. Updated ”Canonicalization and Consistency Checks” on page 526on page 526. Added information about the next sequential instruction pointer (nRIP) in 15.7.1, ”State Saved on Exit” on page 530. Updated priority definition of PAUSE instruction intercept in Table 15-7, “Instruction Intercepts,” on page 536. Added nRIP field to Table B-1, “VMCB Layout, Control Area,” on page 780. Clarified information on ICEBP event injection, on page 554. Deleted erroneous statement concerning the operation of the General Local Vector Table register Mask bit in section 16.4. Clarified the description of the Interrupt Command Register Delivery Status bit in section ”Interprocessor Interrupts (IPI)” on page 673on page 673.

[AMD Public Use]

Date	Revision	Description
September 2007	3.14	Added information on ”Speculative Caching of Address Translations,” ”Caching of Upper Level Translation Table Entries,” ”Use of Cached Entries When Reporting a Page Fault Exception,” ”Use of Cached Entries When Reporting a Page Fault Exception,” ”Handling of D-Bit Updates,” ”Invalidation of Cached Upper-level Entries by INVLPG” on page 163 and ”Handling of PDPT Entries in PAE Mode” on page 163to section 5.5.3, ”TLB Management” on page 161. Added 15.21.7, ”Interrupt Masking in Local APIC” on page 558. Added 16.3.6, ”Extended APIC Control Register” on page 665; clarified the use of the ICR DS bit in 16.5, ”Interprocessor Interrupts (IPI)” on page 673. Added minor clarifications and corrected typographical and formatting errors.
July 2007	3.13	Added 5.3.5, ”1-Gbyte Page Translation” on page 151. Added 7.2, ”Multiprocessor Memory Access Ordering” on page 191 Added divide-by-zero exception to Table 8-9, “Simultaneous Interrupt Priorities,” on page 266. Added information on ”CPU Watchdog Timer Register” on page 306and ”Machine-Check Miscellaneous-Error Information Register 0 (MCi_MISC0)” on page 313to Chapter 9. Added SSE4A support to Chapter 11, ”SSE, MMX, and x87 Programming” on page 344. Added Monitor and MWAIT intercept information to section 15.9, ”Instruction Intercepts” on page 535 and reorganized intercept information; clarified 15.16.1, ”TLB Flush” on page 551. Added Monitor and MWAIT intercepts to tables B-1, ”VMCB Layout, Control Area” on page 780 and C-1, ”SVM Intercept Codes” on page 801. Added Chapter 16, ”Advanced Programmable Interrupt Controller (APIC)” on page 659, Chapter 17, ”OS-Visible Workaround Information” on page 515, Chapter 17, ”Hardware Performance Monitoring and Control” on page 696. Added Table A-7, “Secure Virtual Machine MSR Cross-Reference,” on page 775. Added minor clarifications and corrected typographical and formatting errors.
September 2006	3.12	Added numerous minor clarifications.
December 2005	3.11	Added Chapter 15, Secure Virtual Machine. Incorporated numerous factual corrections and updates.
February 2005	3.10	Corrected Table 8-6, “General-Protection Exception Conditions,” on page 257. Added SSE3 information. Clarified and corrected information on the CPUID instruction and feature identification. Added information on the RDTSCP instruction. Clarified information about MTRRs and PATs in multiprocessing systems.
September 2003	3.09	Corrected numerous minor typographical errors.

[AMD Public Use]

Date	Revision	Description
April 2003	3.08	Clarified terms in section on FXSAVE/FXSTOR. Corrected several minor errors of omission. Documentation of CR0.NW bit has been corrected. Several register diagrams and figure labels have been corrected. Description of shared cache lines has been clarified in 7.3, ”Memory Coherency and Protocol” on page 194.
September 2002	3.07	Made numerous small grammatical changes and factual clarifications. Added Revision History.

[AMD Public Use]

Preface

About This Book

This book is part of a multi-volume work entitled the *AMD64 Architecture Programmer's Manual*. This table lists each volume and its order number.

Title	Order No.
Volume 1: Application Programming	24592
Volume 2: System Programming	24593
Volume 3: General-Purpose and System Instructions	24594
Volume 4: 128-Bit and 256-Bit Media Instructions	26568
Volume 5: 64-Bit Media and x87 Floating-Point Instructions	26569

Audience

This volume (Volume 2) is intended for programmers writing operating systems, loaders, linkers, device drivers, or system utilities. It assumes an understanding of AMD64 architecture application-level programming as described in Volume 1.

This volume describes the AMD64 architecture's resources and functions that are managed by system software, including operating-mode control, memory management, interrupts and exceptions, task and state-change management, system-management mode (including power management), multi-processor support, debugging, and processor initialization.

Application-programming topics are described in Volume 1. Details about each instruction are described in Volumes 3, 4, and 5.

Organization

This volume begins with an overview of system programming and differences between the x86 and AMD64 architectures. This is followed by chapters that describe the following details of system programming:

- *System Resources*—The system registers and processor ID (CPUID) functions.
- *Segmented Virtual Memory*—The segmented-memory models supported by the architecture and their associated data structures and protection checks.
- *Page Translation and Protection*—The page-translation functions supported by the architecture and their associated data structures and protection checks.

[AMD Public Use]

- *System Instructions*—The instructions used to manage system functions.
- *Memory System*—The memory-system hierarchy and its resources and protocols, including memory-characterization, caching, and buffering functions.
- *Exceptions and Interrupts*—Details about the types and causes of exceptions and interrupts, and the methods of transferring control during these events.
- *Machine-Check Mechanism*—The resources and functions that support detection and handling of machine-check errors.
- *System-Management Mode*—The resources and functions that support system-management mode (SMM), including power-management functions.
- *SSE, MMX, and x87 Programming*—The resources and functions that support use (by application software) and state-saving (by the operation system) of the 256-bit media, 128-bit media, 64-bit media, and x87 floating-point instructions.
- *Multiple-Processor Management*—The features of the instruction set and the system resources and functions that support multiprocessing environments.
- *Debug and Performance Resources*—The system resources and functions that support software debugging and performance monitoring.
- *Legacy Task Management*—Support for the legacy hardware multitasking functions, including register resources and data structures.
- *Processor Initialization and Long-Mode Activation*—The methods by which system software initializes and changes operating modes.
- *Mixing Code Across Operating Modes*—Things to remember when running programs in different operating modes.
- *Secure Virtual Machine*—The system resources that support machine virtualization.
- *Advanced Programmable Interrupt Controller (APIC) operation.*

There are appendices describing details of model-specific registers (MSRs) and machine-check implementations. Definitions assumed throughout this volume are listed below. The index at the end of this volume cross-references topics within the volume. For other topics relating to the AMD64 architecture, see the tables of contents and indexes of the other volumes.

Conventions and Definitions

The section which follows, **Notational Conventions,** describes notational conventions used in this volume. The next section, **Definitions,** lists a number of terms used in this volume along with their technical definitions. Some of these definitions assume knowledge of the legacy x86 architecture. See “Related Documents” on page lxii for further information about the legacy x86 architecture. Finally, the **Registers** section lists the registers which are a part of the system programming model.

[AMD Public Use]

Notational Conventions

#GP(0)

An instruction exception—in this example, a general-protection exception with error code of 0.

1011b

A binary value—in this example, a 4-bit value.

F0EA_0B02h

A hexadecimal value. Underscore characters may be inserted to improve readability.

128

Numbers without an alpha suffix are decimal unless the context indicates otherwise.

7:4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first. Commas may be inserted to indicate gaps.

CPUID Fn*XXXX_XXXX_RRR*[*FieldName*]

Support for optional features or the value of an implementation-specific parameter of a processor can be discovered by executing the CPUID instruction on that processor. To obtain this value, software must execute the CPUID instruction with the function code *XXXX_XXXX*h in EAX and then examine the field *FieldName* returned in register *RRR*. If the "_*RRR*" notation is followed by "_x*YYY*", register ECX must be set to the value *YYY*h before executing CPUID. When *FieldName* is not given, the entire contents of register *RRR* contains the desired value. When determining optional feature support, if the bit identified by *FieldName* is set to a one, the feature is supported on that processor.

CR0–CR4

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR0[PE], CR0.PE

Notation for referring to a field within a register—in this case, the PE field of the CR0 register.

CR0[PE] = 1, CR0.PE = 1

The PE field of the CR0 register is set (contains the value 1).

EFER[LME] = 0, EFER.LME = 0

The LME field of the EFER register is cleared (contains a value of 0).

DS:SI

A far pointer or logical address. The real address or segment descriptor specified by the segment register (DS in this example) is combined with the offset contained in the second register (SI in this example) to form a real or virtual address.

[AMD Public Use]

RFLAGS[13:12]

A field within a register identified by its bit range. In this example, corresponding to the IOPL field.

Definitions

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See *legacy mode* and *compatibility mode*.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode*.

64-bit mode

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

absolute

Said of a displacement that references the base of a code segment rather than an instruction pointer. Contrast with *relative*.

ASID

Address space identifier.

byte

Eight bits.

clear

To write a bit value of 0. Compare *set*.

compatibility mode

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction's result to software-visible storage, such as a register (including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

direct

Referencing a memory location whose address is included in the instruction's syntax as an immediate operand. The address may be an absolute or relative address. Compare *indirect*.

[AMD Public Use]

dirty data

Data held in the processor's caches or internal buffers that is more recent than the copy held in main memory.

displacement

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

doubleword

Two words, or four bytes, or 32 bits.

double quadword

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

effective address size

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

effective operand size

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

exception

An abnormal condition that occurs as the result of executing an instruction. The processor's response to an exception depends on the type of the exception. For all exceptions except 128-bit media SIMD floating-point exceptions and x87 floating-point exceptions, control is transferred to the handler (or service routine) for that exception, as defined by the exception's vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

flush

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

GDT

Global descriptor table.

GIF

Global interrupt flag.

GPA

Guest physical address. In a virtualized environment, the page tables maintained by the guest operating system provide the translation from the linear (virtual) address to the guest physical

[AMD Public Use]

address. Nested page tables define the translation of the GPA to the host physical address (HPA). See *SPA* and *HPA*.

HPA

Host physical address. The address space owned by the virtual machine monitor. In a virtualized environment, nested page translation tables controlled by the VMM provide the translation from the guest physical address to the host physical address. See *GPA*.

IDT

Interrupt descriptor table.

IGN

Ignored. Value written is ignored by hardware. Value returned on a read is indeterminate. See *reserved*.

indirect

Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. Compare *direct*.

IRB

The virtual-8086 mode interrupt-redirection bitmap.

IST

The long-mode interrupt-stack table.

IVT

The real-address mode interrupt vector table.

LDT

Local descriptor table.

legacy x86

The legacy x86 architecture. See “Related Documents” on page lxii for descriptions of the legacy x86 architecture.

legacy mode

An operating mode of the AMD64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Legacy mode has three submodes, *real mode*, *protected mode*, and *virtual-8086 mode*.

LIP

Linear Instruction Pointer. LIP = (CS.base + rIP).

[AMD Public Use]

long mode

An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

lsb

Least-significant bit.

LSB

Least-significant byte.

main memory

Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask

(1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ

Must be zero. If software attempts to set an MBZ bit to 1 in a system register, a general-protection exception (#GP) occurs; if in a translation table entry, a reserved-bit page fault exception (#PF) will occur if the hardware attempts to use the entry for address translation. See *reserved*.

memory

Unless otherwise specified, *main memory*.

ModRM

A byte following an instruction opcode that specifies address calculation based on mode (Mod), register (R), and memory (M) variables.

moffset

A 16, 32, or 64-bit offset that specifies a memory operand directly, without using a ModRM or SIB byte.

msb

Most-significant bit.

MSB

Most-significant byte.

octword

Same as *double quadword.*

offset

Same as *displacement*.

[AMD Public Use]

overflow

The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.

PAE

Physical-address extensions.

physical memory

Actual memory, consisting of *main memory* and cache.

probe

A check for an address in a processor's caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

procedure stack

A portion of a stack segment in memory that is used to link procedures. Also known as a *program stack.*

program stack

See *procedure stack.*

protected mode

A submode of *legacy mode*.

quadword

Four words, or eight bytes, or 64 bits.

RAZ

Value returned on a read is always zero (0) regardless of what was previously written. See *reserved*.

RA1

Value returned on a read is always one (1) regardless of what was previously written. See *reserved*.

real-address mode

See *real mode*.

real mode

A short name for *real-address mode,* a submode of *legacy mode*.

relative

Referencing with a displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with *absolute*.

[AMD Public Use]

reserved

Fields marked as reserved may be used at some future time.

To preserve compatibility with future processors, reserved fields require special handling when read or written by software. Software must not depend on the state of a reserved field (unless qualified as RAZ), nor upon the ability of such fields to return a previously written state.

If a field is marked reserved without qualification, software must not change the state of that field; it must reload that field with the same value returned from a prior read.

Reserved fields may be qualified as IGN, MBZ, RAZ, or SBZ (see definitions).

REX

An instruction prefix that specifies a 64-bit operand size and provides access to additional registers.

RIP-relative addressing

Addressing relative to the 64-bit RIP instruction pointer.

SBZ

Should be zero. An attempt by software to set an SBZ bit to 1 results in undefined behavior. See *reserved*.

shadow stack

A shadow stack is a separate, protected stack that is conceptually parallel to the procedure stack and used only by the shadow stack feature.

set

To write a bit value of 1. Compare *clear*.

SIB

A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).

SPA

System physical address. The address directly used to address system memory. Under SVM, also known as the host physical address. See *HPA*.

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

SVM

Secure virtual machine. AMD's virtualization architecture. SVM is defined in Chapter 15 on page 520.

[AMD Public Use]

System software

Privileged software that owns and manages the hardware resources of a system after initialization by *system firmware* and controls access to these resources. In a non-virtualized environment, system software is provided by the operating system. In a virtualized environment, system software is largely equivalent to the virtual machine monitor (VMM), also commonly known as the *hypervisor*.

TOP

The x87 top-of-stack pointer.

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called *elements*, that are packed into a single data object. Most of the SSE and 64-bit media instructions use vectors as operands.

(2) An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare *exception*.

virtual-8086 mode

A submode of *legacy mode*.

VMCB

Virtual machine control block.

VMM

Virtual machine monitor.

word

Two bytes, or 16 bits.

x86

See *legacy x86*.

Registers

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

AH–DH

The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL*.

[AMD Public Use]

AL–DL
: The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH*.

AL–r15B
: The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

BP
: Base pointer register.

CR*n*
: Control register number *n*.

CS
: Code segment register.

eAX–eSP
: The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP*.

EFER
: Extended features enable register.

eFLAGS
: 16-bit or 32-bit flags register. Compare *rFLAGS*.

EFLAGS
: 32-bit (extended) flags register.

eIP
: 16-bit or 32-bit instruction-pointer register. Compare *rIP*.

EIP
: 32-bit (extended) instruction-pointer register.

FLAGS
: 16-bit flags register.

GDTR
: Global descriptor table register.

GPRs
: General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

[AMD Public Use]

IDTR
: Interrupt descriptor table register.

IP
: 16-bit instruction-pointer register.

LDTR
: Local descriptor table register.

MSR
: Model-specific register.

r8–r15
: The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP
: The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

RAX
: 64-bit version of the EAX register.

RBP
: 64-bit version of the EBP register.

RBX
: 64-bit version of the EBX register.

RCX
: 64-bit version of the ECX register.

RDI
: 64-bit version of the EDI register.

RDX
: 64-bit version of the EDX register.

rFLAGS
: 16-bit, 32-bit, or 64-bit flags register. Compare *RFLAGS*.

RFLAGS
: 64-bit flags register. Compare *rFLAGS*.

[AMD Public Use]

rIP
: 16-bit, 32-bit, or 64-bit instruction-pointer register. Compare *RIP*.

RIP
: 64-bit instruction-pointer register.

RSI
: 64-bit version of the ESI register.

RSP
: 64-bit version of the ESP register.

SP
: Stack pointer register.

SS
: Stack segment register.

SSP
: Shadow-stack pointer register.

TPR
: Task priority register (CR8), a new register introduced in the AMD64 architecture to speed interrupt management.

TR
: Task register.

ZMM/YMM/XMM
: Set of 8, 16, or 32 registers, 128, 256, or 512 bits wide, depending on features enabled and mode of the CPU.

Endian Order

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

Related Documents

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[AMD Public Use]

- AMD, *BIOS and Kernel Developer's Guide* (BKDG) for particular hardware implementations of older families of the AMD64 architecture.
- AMD, *Processor Programming Reference (PPR)* for particular hardware implementations of newer families of the AMD64 architecture.
- AMD, *AMD I/O Virtualization Technology (IOMMU) Specification*, Revision 2.2 or later; order number 48882.
- AMD, *Software Optimization Guide for AMD Family 15h Processors,* order number 47414.
- Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.
- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts,* Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals,* Addison-Wesley, New York, 1994.
- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.
- Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC,* Addison-Wesley, New York, 1994.
- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.
- Hal Katircioglu, *Inside the 486, Pentium, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.

[AMD Public Use]

- IBM Corporation, *486SLC Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *486SLC2 Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightning 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book,* Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc.*, Nx586™ Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc.*, Nx686™ Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.
- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium® III*, www.x86.org/articles/sse_pt1/ simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible,* Microsoft Press, Redmond, WA, 1993.
- *PharLap 386|ASM Reference Manual*, Pharlap, Cambridge MA, 1993.
- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Jeffrey P. Royer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.
- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.
- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.

[AMD Public Use]

1 System-Programming Overview

This entire volume is intended for system-software developers—programmers writing operating systems, loaders, linkers, device drivers, or utilities that require access to system resources. These system resources are generally available only to software running at the highest-privilege level (CPL=0), also referred to as *privileged software*. Privilege levels and their interactions are fully described in “Segment-Protection Overview” on page 106.

This chapter introduces the basic features and capabilities of the AMD64 architecture that are available to system-software developers. The concepts include:

- The supported address forms and how memory is organized.
- How memory-management hardware makes use of the various address forms to access memory.
- The processor operating modes, and how the memory-management hardware supports each of those modes.
- The system-control registers used to manage system resources.
- The interrupt and exception mechanism, and how it is used to interrupt program execution and to report errors.
- Additional, miscellaneous features available to system software, including support for hardware multitasking, reporting machine-check exceptions, debugging software problems, and optimizing software performance.

Many of the legacy features and capabilities are enhanced by the AMD64 architecture to support 64-bit operating systems and applications, while providing backward-compatibility with existing software.

1.1 Memory Model

The AMD64 architecture memory model is designed to allow system software to manage application software and associated data in a secure fashion. The memory model is backward-compatible with the legacy memory model. Hardware-translation mechanisms are provided to map addresses between virtual-memory space and physical-memory space. The translation mechanisms allow system software to relocate applications and data transparently, either anywhere in physical-memory space, or in areas on the system hard drive managed by the operating system.

In long mode, the AMD64 architecture implements a flat-memory model. In legacy mode, the architecture implements all legacy memory models.

[AMD Public Use]

1.1.1 Memory Addressing

The AMD64 architecture supports address relocation. To do this, several types of addresses are needed to completely describe memory organization. Specifically, four types of addresses are defined by the AMD64 architecture:

- Logical addresses
- Effective addresses, or segment offsets, which are a portion of the logical address.
- Linear (virtual) addresses
- Physical addresses

Logical Addresses. A *logical address* is a reference into a segmented-address space. It is comprised of the segment selector and the effective address. Notationally, a logical address is represented as

```
Logical Address = Segment Selector : Offset
```

The segment selector specifies an entry in either the global or local descriptor table. The specified descriptor-table entry describes the segment location in virtual-address space, its size, and other characteristics. The effective address is used as an offset into the segment specified by the selector.

Logical addresses are often referred to as *far pointers*. Far pointers are used in software addressing when the segment reference must be explicit (i.e., a reference to a segment outside the current segment).

Effective Addresses. The offset into a memory segment is referred to as an effective address (see "Segmentation" on page 5 for a description of segmented memory). Effective addresses are formed by adding together elements comprising a base value, a scaled-index value, and a displacement value. The effective-address computation is represented by the equation

```
Effective Address = Base + (Scale x Index) + Displacement
```

The elements of an effective-address computation are defined as follows:

- *Base*—A value stored in any general-purpose register.
- *Scale*—A positive value of 1, 2, 4, or 8.
- *Index*—A two's-complement value stored in any general-purpose register.
- *Displacement*—An 8-bit, 16-bit, or 32-bit two's-complement value encoded as part of the instruction.

Effective addresses are often referred to as *near pointers*. A near pointer is used when the segment selector is known implicitly or when the flat-memory model is used.

Long mode defines a 64-bit effective-address length. If a processor implementation does not support the full 64-bit virtual-address space, the effective address must be in *canonical form* (see "Canonical Address Form" on page 4).

[AMD Public Use]

Linear (Virtual) Addresses. The segment-selector portion of a logical address specifies a segment-descriptor entry in either the global or local descriptor table. The specified segment-descriptor entry contains the segment-base address, which is the starting location of the segment in linear-address space. A *linear address* is formed by adding the segment-base address to the effective address (segment offset), which creates a reference to any byte location within the supported linear-address space. Linear addresses are often referred to as *virtual addresses*, and both terms are used interchangeably throughout this document.

```
Linear Address = Segment Base Address + Effective Address
```

When the flat-memory model is used—as in 64-bit mode—a segment-base address is treated as 0. In this case, the linear address is identical to the effective address. In long mode, linear addresses must be in canonical address form, as described in “Canonical Address Form” on page 4.

Physical Addresses. A *physical address* is a reference into the physical-address space, typically main memory. Physical addresses are translated from virtual addresses using page-translation mechanisms. See “Paging” on page 7 for information on how the paging mechanism is used for virtual-address to physical-address translation. When the paging mechanism is not enabled, the virtual (linear) address is used as the physical address.

1.1.2 Memory Organization

The AMD64 architecture organizes memory into *virtual memory* and *physical memory*. Virtual-memory and physical-memory spaces can be (and usually are) different in size. Generally, the virtual-address space is much larger than physical-address memory. System software relocates applications and data between physical memory and the system hard disk to make it appear that much more memory is available than really exists. System software then uses the hardware memory-management mechanisms to map the larger virtual-address space into the smaller physical-address space.

Virtual Memory. Software uses virtual addresses to access locations within the virtual-memory space. System software is responsible for managing the relocation of applications and data in virtual-memory space using segment-memory management. System software is also responsible for mapping virtual memory to physical memory through the use of page translation. The AMD64 architecture supports different virtual-memory sizes using the following address-translation modes:

- *Protected Mode*—This mode supports 4 gigabytes of virtual-address space using 32-bit virtual addresses.
- *Long Mode*—This mode supports 16 exabytes of virtual-address space using 64-bit virtual addresses. A given implementation may however support less than this, as reported by the CPUID feature identification facility.

[AMD Public Use]

Physical Memory. Physical addresses are used to directly access main memory. For a particular computer system, the size of the *available* physical-address space is equal to the amount of main memory installed in the system. The maximum amount of physical memory accessible depends on the processor implementation and on the address-translation mode. The AMD64 architecture supports varying physical-memory sizes using the following address-translation modes:

- *Real-Address Mode*—This mode, also called *real mode*, supports 1 megabyte of physical-address space using 20-bit physical addresses. This address-translation mode is described in "Real Addressing" on page 10. Real mode is available only from legacy mode (see "Legacy Modes" on page 14).
- *Legacy Protected Mode*—This mode supports several different address-space sizes, depending on the translation mechanism used and whether extensions to those mechanisms are enabled.

  Legacy protected mode supports 4 gigabytes of physical-address space using 32-bit physical addresses. Both segment translation (see "Segmentation" on page 5) and page translation (see "Paging" on page 7) can be used to access the physical address space, when the processor is running in legacy protected mode.

  When the physical-address size extensions are enabled (see "Physical-Address Extensions (PAE) Bit" on page 131), the page-translation mechanism can be extended to support 52-bit physical addresses. 52-bit physical addresses allow up to 4 petabytes of physical-address space to be supported. (Currently, the AMD64 architecture supports 40-bit addresses in this mode, allowing up to 1 terabyte of physical-address space to be supported.
- *Long Mode*—This mode is unique to the AMD64 architecture. This mode supports up to 4 petabytes of physical-address space using 52-bit physical addresses. Long mode requires the use of page-translation and the physical-address size extensions (PAE).

1.1.3 Canonical Address Form

Long mode defines 64 bits of virtual-address space, but processor implementations can support less. Although some processor implementations do not use all 64 bits of the virtual address, they all check bits 63 through the most-significant implemented bit to see if those bits are all zeros or all ones. An address that complies with this property is in *canonical address form*. In most cases, a virtual-memory reference that is not in canonical form (in either the linear or effective form of the address) causes a general-protection exception (#GP) to occur. However, implied stack references where the stack address is not in canonical form causes a stack exception (#SS) to occur. Implied stack references include all push and pop instructions, and any instruction using RSP or RBP as a base register.

By checking canonical-address form, the AMD64 architecture prevents software from exploiting unused high bits of pointers for other purposes. Software complying with canonical-address form on a specific processor implementation can run unchanged on long-mode implementations supporting larger virtual-address spaces.

[AMD Public Use]

1.2 Memory Management

Memory management consists of the methods by which addresses generated by software are translated by segmentation and/or paging into addresses in physical memory. Memory management is not visible to application software. It is handled by the system software and processor hardware.

1.2.1 Segmentation

Segmentation was originally created as a method by which system software could isolate software processes (tasks), and the data used by those processes, from one another in an effort to increase the reliability of systems running multiple processes simultaneously.

The AMD64 architecture is designed to support all forms of legacy segmentation. However, most modern system software does not use the segmentation features available in the legacy x86 architecture. Instead, system software typically handles program and data isolation using page-level protection. For this reason, the AMD64 architecture dispenses with multiple segments in 64-bit mode and, instead, uses a flat-memory model. The elimination of segmentation allows new 64-bit system software to be coded more simply, and it supports more efficient management of multi-processing than is possible in the legacy x86 architecture.

Segmentation is, however, used in compatibility mode and legacy mode. Here, segmentation is a form of base memory-addressing that allows software and data to be relocated in virtual-address space off of an arbitrary base address. Software and data can be relocated in virtual-address space using one or more variable-sized *memory segments.* The legacy x86 architecture provides several methods of restricting access to segments from other segments so that software and data can be protected from interfering with each other.

In compatibility and legacy modes, up to 16,383 unique segments can be defined. The base-address value, segment size (called a *limit*), protection, and other attributes for each segment are contained in a data structure called a *segment descriptor*. Collections of segment descriptors are held in *descriptor tables*. Specific segment descriptors are referenced or selected from the descriptor table using a *segment selector register*. Six segment-selector registers are available, providing access to as many as six segments at a time.

Figure 1-1 on page 6 shows an example of segmented memory. Segmentation is described in Chapter 4, “Segmented Virtual Memory.”

[AMD Public Use]

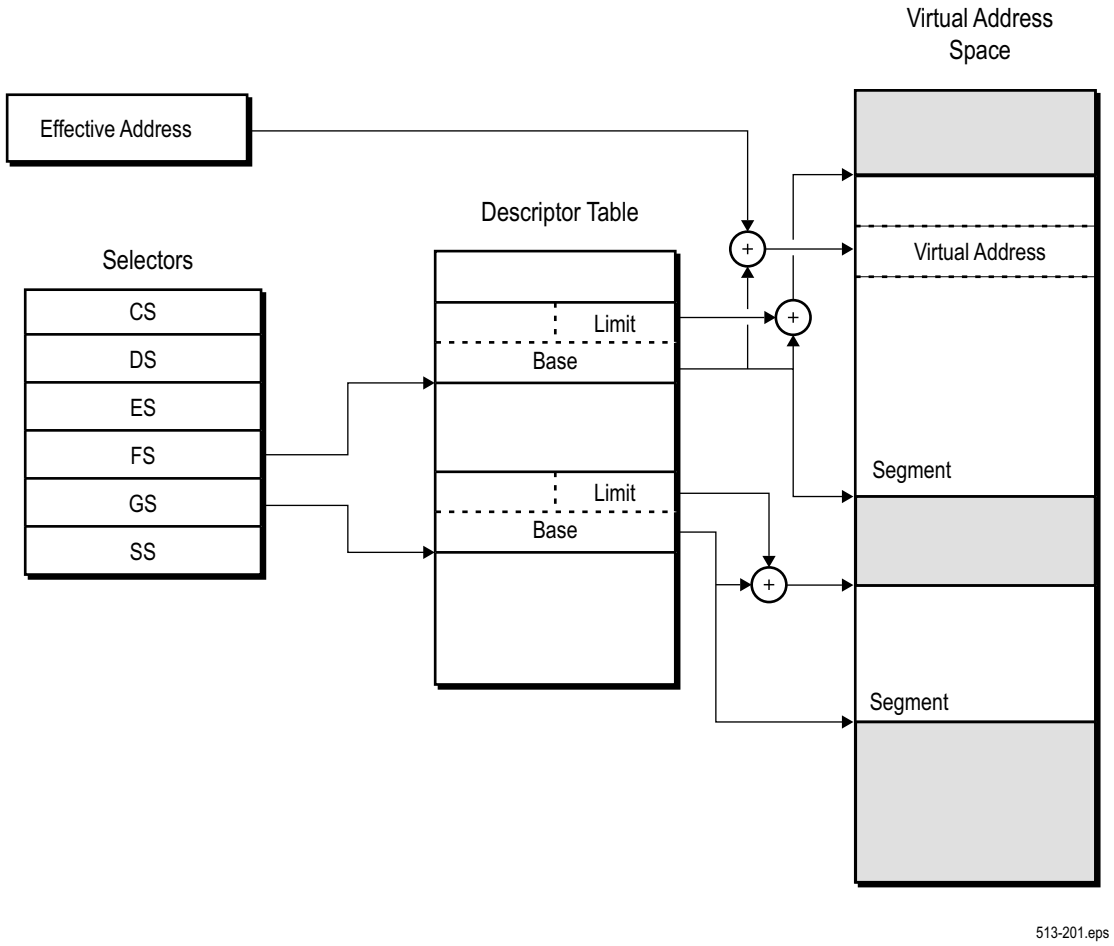


Figure 1-1. Segmented-Memory Model

Flat Segmentation. One special case of segmented memory is the flat-memory model. In the legacy flat-memory model, all segment-base addresses have a value of 0, and the segment limits are fixed at 4 Gbytes. Segmentation cannot be disabled but use of the flat-memory model effectively disables segment translation. The result is a virtual address that equals the effective address. Figure 1-2 on page 7 shows an example of the flat-memory model.

Software running in 64-bit mode automatically uses the flat-memory model. In 64-bit mode, the segment base is treated as if it were 0, and the segment limit is ignored. This allows an effective addresses to access the full virtual-address space supported by the processor.

[AMD Public Use]

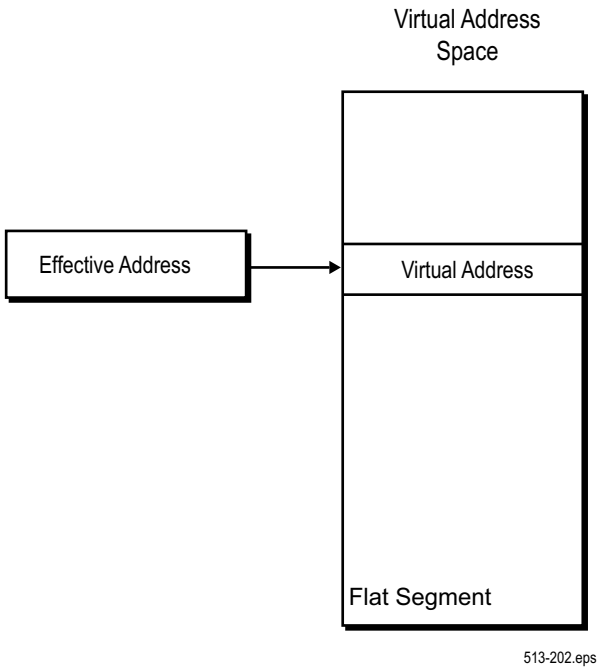


Figure 1-2. Flat Memory Model

1.2.2 Paging

Paging allows software and data to be relocated in physical-address space using fixed-size blocks called *physical pages*. The legacy x86 architecture supports three different physical-page sizes of 4 Kbytes, 2 Mbytes, and 4 Mbytes. As with segment translation, access to physical pages by lesser-privileged software can be restricted.

Page translation uses a hierarchical data structure called a page-translation table to translate virtual pages into physical-pages. The number of levels in the translation-table hierarchy can be as few as one or as many as four, depending on the physical-page size and processor operating mode. Translation tables are aligned on 4-Kbyte boundaries. Physical pages must be aligned on 4-Kbyte, 2-Mbyte, or 4-Mbyte boundaries, depending on the physical-page size.

Each table in the translation hierarchy is indexed by a portion of the virtual-address bits. The entry referenced by the table index contains a pointer to the base address of the next-lower-level table in the translation hierarchy. In the case of the lowest-level table, its entry points to the physical-page base address. The physical page is then indexed by the least-significant bits of the virtual address to yield the physical address.

Figure 1-3 on page 8 shows an example of paged memory with three levels in the translation-table hierarchy. Paging is described in Chapter 5, “Page Translation and Protection.”

[AMD Public Use]

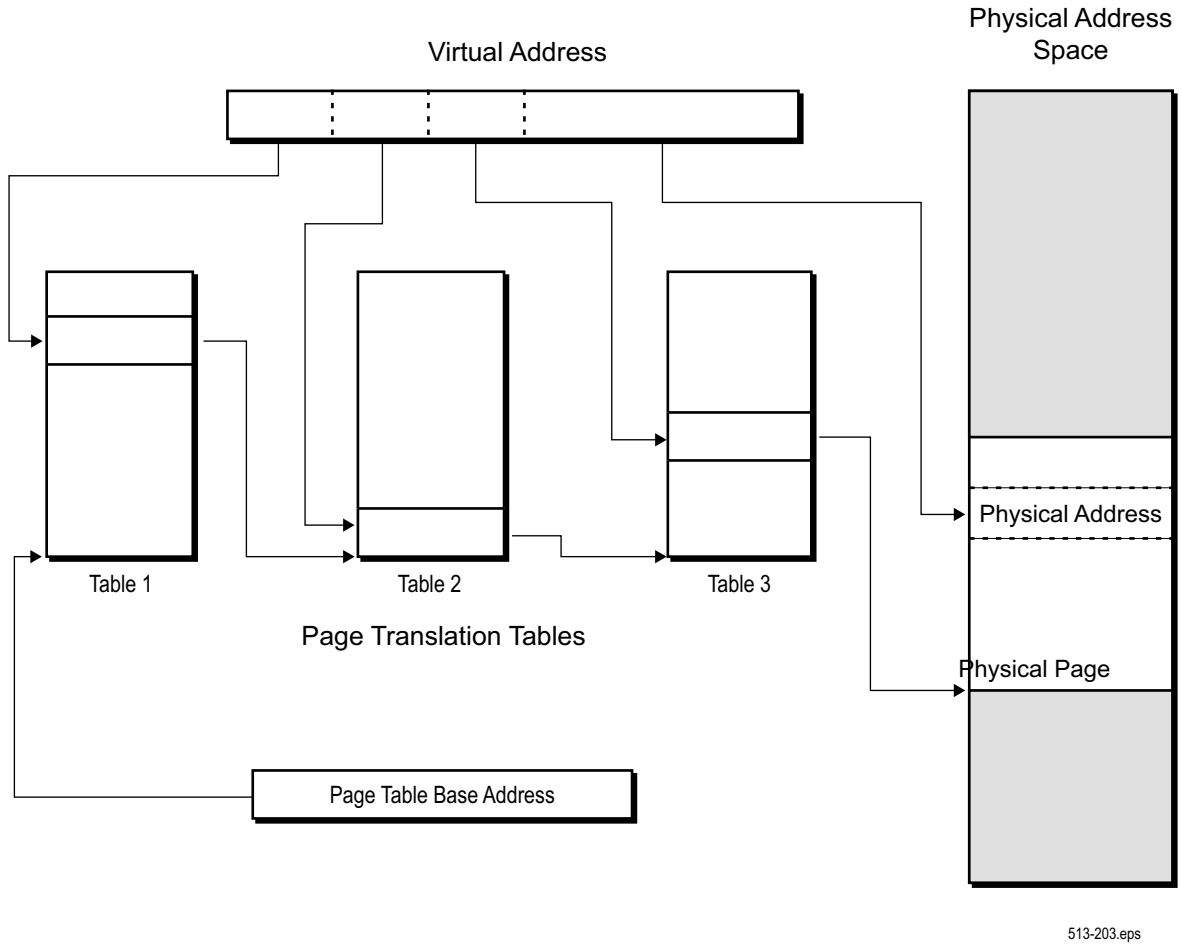


Figure 1-3. Paged Memory Model

Software running in long mode is required to have page translation enabled.

1.2.3 Mixing Segmentation and Paging

Memory-management software can combine the use of segmented memory and paged memory. Because segmentation cannot be disabled, paged-memory management requires some minimum initialization of the segmentation resources. Paging can be completely disabled, so segmented-memory management does not require initialization of the paging resources.

Segments can range in size from a single byte to 4 Gbytes in length. It is therefore possible to map multiple segments to a single physical page and to map multiple physical pages to a single segment. Alignment between segment and physical-page boundaries is not required, but memory-management software is simplified when segment and physical-page boundaries are aligned.

[AMD Public Use]

The simplest, most efficient method of memory management is the flat-memory model. In the flat-memory model, all segment base addresses have a value of 0 and the segment limits are fixed at 4 Gbytes. The segmentation mechanism is still used each time a memory reference is made, but because virtual addresses are identical to effective addresses in this model, the segmentation mechanism is effectively ignored. Translation of virtual (or effective) addresses to physical addresses takes place using the paging mechanism only.

Because 64-bit mode disables segmentation, it uses a flat, paged-memory model for memory management. The 4 Gbyte segment limit is ignored in 64-bit mode. Figure 1-4 shows an example of this model.

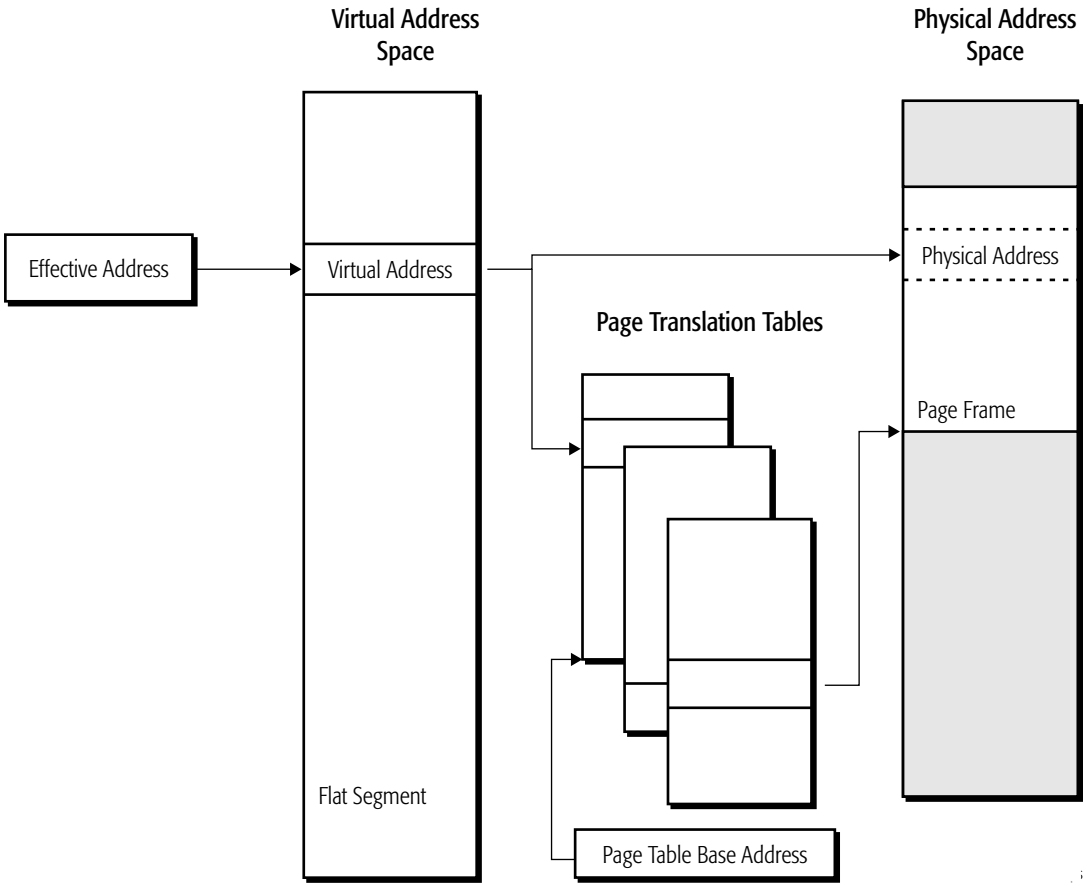


Figure 1-4. 64-Bit Flat, Paged-Memory Model

[AMD Public Use]

1.2.4 Real Addressing

Real addressing is a legacy-mode form of address translation used in real mode. This simplified form of address translation is backward compatible with 8086-processor effective-to-physical address translation. In this mode, 16-bit effective addresses are mapped to 20-bit physical addresses, providing a 1-Mbyte physical-address space.

Segment selectors are used in real-address translation, but not as an index into a descriptor table. Instead, the 16-bit segment-selector value is shifted left by 4 bits to form a 20-bit segment-base address. The 16-bit effective address is added to this 20-bit segment base address to yield a 20-bit physical address. If the sum of the segment base and effective address carries over into bit 20, that bit can be optionally truncated to mimic the 20-bit address wrapping of the 8086 processor by using the A20M# input signal to mask the A20 address bit.

A20 address bit masking should only be used in real mode (see next section for information on real mode). Use in other modes may result in address translation errors.

Real-address translation supports a 1-Mbyte physical-address space using up to 64K segments aligned on 16-byte boundaries. Each segment is exactly 64 Kbytes long. Figure 1-5 shows an example of real-address translation.

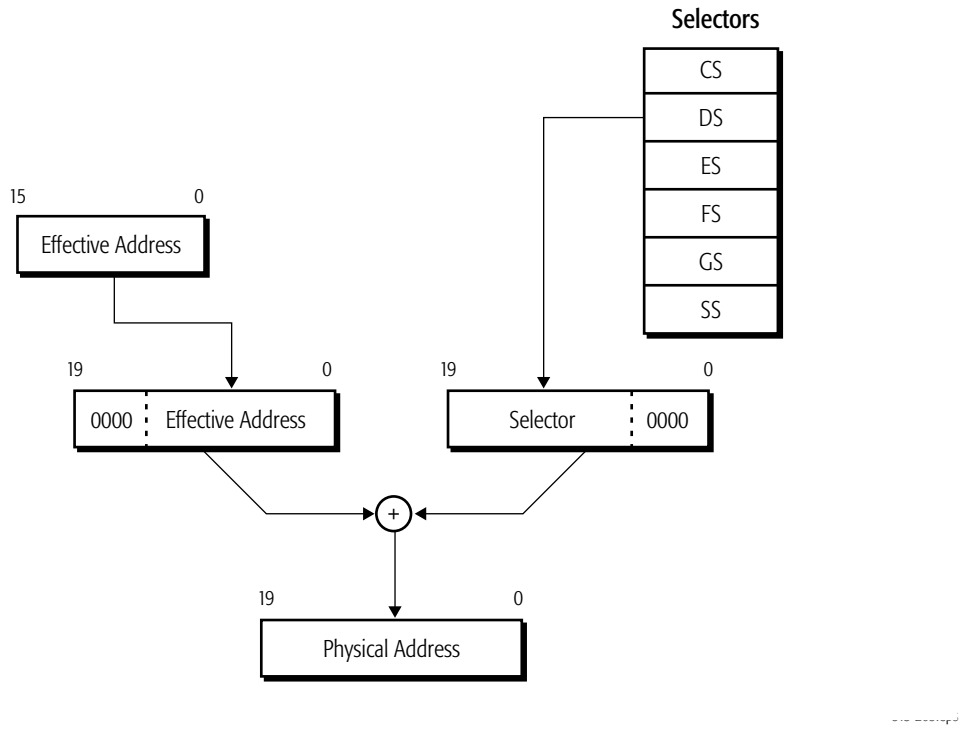


Figure 1-5. Real-Address Memory Model

[AMD Public Use]

1.3 Operating Modes

The legacy x86 architecture provides four operating modes or environments that support varying forms of memory management, virtual-memory and physical-memory sizes, and protection:

- Real Mode.
- Protected Mode.
- Virtual-8086 Mode.
- System Management Mode.

The AMD64 architecture supports all these legacy modes, and it adds a new operating mode called *long mode*. Table 1-1 shows the differences between long mode and legacy mode. Software can move between all supported operating modes as shown in Figure 1-6 on page 12. Each operating mode is described in the following sections.

Table 1-1. Operating Modes

Mode		System Software Required	Application Recompile Required	Defaults[1]		Register Extensions[2]	Maximum GPR Width (bits)
				Address Size (bits)	Operand Size (bits)		
Long Mode3	64-Bit Mode	New 64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16	16		
Legacy Mode	Protected Mode	Legacy 32-bit OS	no	32	32	no	32
				16	16		
	Virtual-8086 Mode			16	16		32
	Real Mode	Legacy 16-bit OS					

Note:

1. *Defaults can be overridden in most modes using an instruction prefix or system control bit.*
2. *Register extensions include access to the upper eight general-purpose and YMM/XMM registers, uniform access to lower 8 bits of all GPRs, and access to the upper 32 bits of the GPRs.*
3. *Long mode supports only x86 protected mode. It does not support x86 real mode or virtual-8086 mode.*

[AMD Public Use]

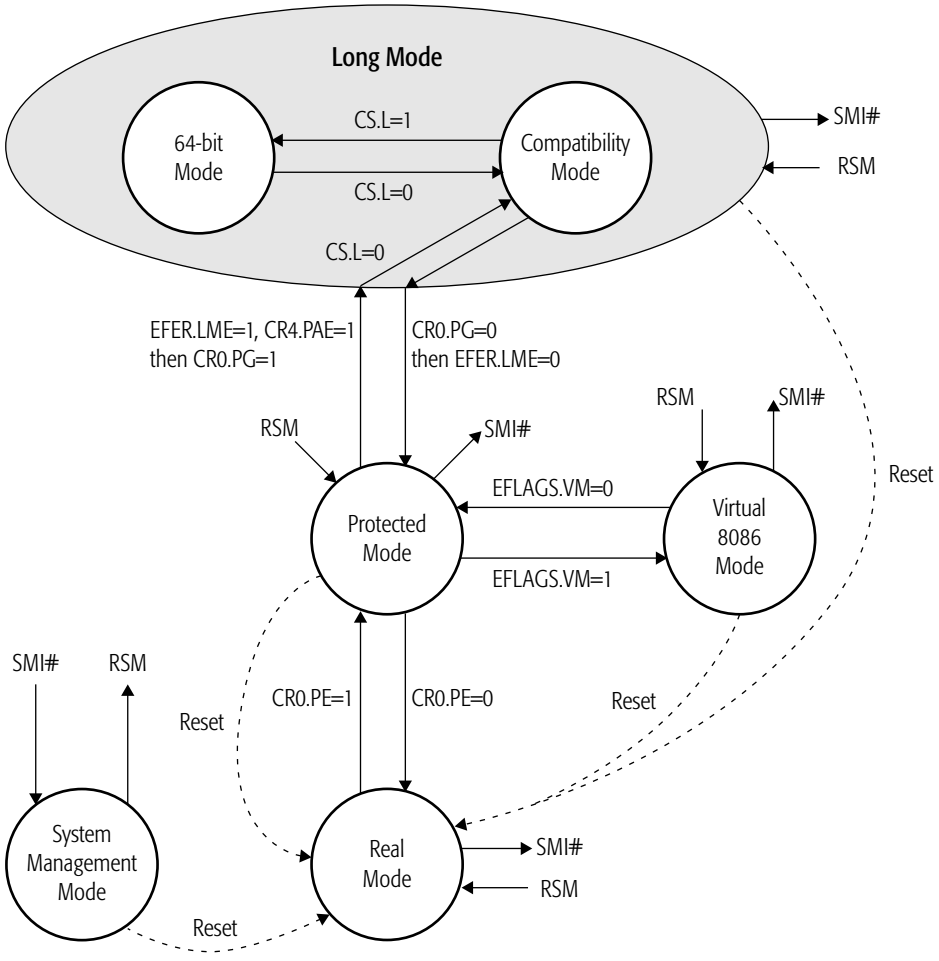


Figure 1-6. Operating Modes of the AMD64 Architecture

1.3.1 Long Mode

Long mode consists of two submodes: *64-bit mode* and *compatibility mode*. 64-bit mode supports several new features, including the ability to address 64-bit virtual-address space. Compatibility mode provides binary compatibility with existing 16-bit and 32-bit applications when running on 64-bit system software.

Throughout this document, references to *long mode* refer collectively to both *64-bit mode* and *compatibility mode*. If a function is specific to either 64-bit mode or compatibility mode, then those specific names are used instead of the name *long mode*.

Before enabling and activating long mode, system software must first enable protected mode. The process of enabling and activating long mode is described in Chapter 14, “Processor Initialization and

[AMD Public Use]

Long Mode Activation.” Long mode features are described throughout this document, where applicable.

1.3.2 64-Bit Mode

64-bit mode, a submode of long mode, provides support for 64-bit system software and applications by adding the following features:

- 64-bit virtual addresses (processor implementations can have fewer).
- Access to General Purpose Register bits 63:32
- Access to additional registers through the REX, VEX, and XOP instruction prefixes:
  - eight additional GPRs (R8–R15)
  - eight additional Streaming SIMD Extension (SSE) registers (YMM/XMM8–15)
- 64-bit instruction pointer (RIP).
- New RIP-relative data-addressing mode.
- Flat-segment address space with single code, data, and stack space.

The mode is enabled by the system software on an individual code-segment basis. Although code segments are used to enable and disable 64-bit mode, the legacy segmentation mechanism is largely disabled. Page translation is required for memory management purposes. Because 64-bit mode supports a 64-bit virtual-address space, it requires 64-bit system software and development tools.

In 64-bit mode, the default address size is 64 bits, and the default operand size is 32 bits. The defaults can be overridden on an instruction-by-instruction basis using instruction prefixes. A new REX prefix is introduced for specifying a 64-bit operand size and the new registers.

1.3.3 Compatibility Mode

Compatibility mode, a submode of long mode, allows system software to implement binary compatibility with existing 16-bit and 32-bit x86 applications. It allows these applications to run, without recompilation, under 64-bit system software in long mode, as shown in Table 1-1 on page 11.

In compatibility mode, applications can only access the first 4 Gbytes of virtual-address space. Standard x86 instruction prefixes toggle between 16-bit and 32-bit address and operand sizes.

Compatibility mode, like 64-bit mode, is enabled by system software on an individual code-segment basis. Unlike 64-bit mode, however, segmentation functions the same as in the legacy-x86 architecture, using 16-bit or 32-bit protected-mode semantics. From an application viewpoint, compatibility mode looks like a legacy protected-mode environment. From a system-software viewpoint, the long-mode mechanisms are used for address translation, interrupt and exception handling, and system data-structures.

[AMD Public Use]

1.3.4 Legacy Modes

L*egacy mode* consists of three submodes: real mode, protected mode, and virtual-8086 mode. Protected mode can be either paged or unpaged. L*egacy mode* preserves binary compatibility not only with existing x86 16-bit and 32-bit applications but also with existing x86 16-bit and 32-bit system software.

Real Mode. In this mode, also called real-address mode, the processor supports a physical-memory space of 1 Mbyte and operand sizes of 16 bits (default) or 32 bits (with instruction prefixes). Interrupt handling and address generation are nearly identical to the 80286 processor's real mode. Paging is not supported. All software runs at privilege level 0.

Real mode is entered after reset or processor power-up. The mode is not supported when the processor is operating in long mode because long mode requires that paged protected mode be enabled.

Protected Mode. In this mode, the processor supports virtual-memory and physical-memory spaces of 4 Gbytes and operand sizes of 16 or 32 bits. All segment translation, segment protection, and hardware multitasking functions are available. System software can use segmentation to relocate effective addresses in virtual-address space. If paging is not enabled, virtual addresses are equal to physical addresses. Paging can be optionally enabled to allow translation of virtual addresses to physical addresses and to use the page-based memory-protection mechanisms.

In protected mode, software runs at privilege levels 0, 1, 2, or 3. Typically, application software runs at privilege level 3, the system software runs at privilege levels 0 and 1, and privilege level 2 is available to system software for other uses. The 16-bit version of this mode was first introduced in the 80286 processor.

Virtual-8086 Mode. Virtual-8086 mode allows system software to run 16-bit real-mode software on a virtualized-8086 processor. In this mode, software written for the 8086, 8088, 80186, or 80188 processor can run as a privilege-level-3 task under protected mode. The processor supports a virtual-memory space of 1 Mbytes and operand sizes of 16 bits (default) or 32 bits (with instruction prefixes), and it uses real-mode address translation.

Virtual-8086 mode is enabled by setting the virtual-machine bit in the EFLAGS register (EFLAGS.VM). EFLAGS.VM can only be set or cleared when the EFLAGS register is loaded from the TSS as a result of a task switch, or by executing an IRET instruction from privileged software. The POPF instruction cannot be used to set or clear the EFLAGS.VM bit.

Virtual-8086 mode is not supported when the processor is operating in long mode. When long mode is enabled, any attempt to enable virtual-8086 mode is silently ignored.

[AMD Public Use]

1.3.5 System Management Mode (SMM)

System management mode (SMM) is an operating mode designed for system-control activities that are typically transparent to conventional system software. Power management is one popular use for system management mode. SMM is primarily targeted for use by platform firmware and specialized low-level device drivers. The code and data for SMM are stored in the SMM memory area, which is isolated from main memory by the SMM output signal.

SMM is entered by way of a system management interrupt (SMI). Upon recognizing an SMI, the processor enters SMM and switches to a separate address space where the SMM handler is located and executes. In SMM, the processor supports real-mode addressing with 4 Gbyte segment limits and default operand, address, and stack sizes of 16 bits (prefixes can be used to override these defaults).

1.4 System Registers

Figure 1-7 on page 16 shows the system registers defined for the AMD64 architecture. System software uses these registers to, among other things, manage the processor operating environment, define system resource characteristics, and to monitor software execution. With the exception of the RFLAGS register, system registers can be read and written only from privileged software.

Except for the descriptor-table registers and task register, the AMD64 architecture defines all system registers to be 64 bits wide. The descriptor table and task registers are defined by the AMD64 architecture to include 64-bit base-address fields, in addition to their other fields.

As shown in Figure 1-7 on page 16, the system registers include:

- *Control Registers*—These registers are used to control system operation and some system features. See “System-Control Registers” on page 41 for details.
- *System-Flags Register*—The RFLAGS register contains system-status flags and masks. It is also used to enable virtual-8086 mode and to control application access to I/O devices and interrupts. See “RFLAGS Register” on page 52 for details.
- *Descriptor-Table Registers*—These registers contain the location and size of descriptor tables stored in memory. Descriptor tables hold segmentation data structures used in protected mode. See “Descriptor Tables” on page 84 for details.
- *Task Register*—The task register contains the location and size in memory of the task-state segment. The hardware-multitasking mechanism uses the task-state segment to hold state information for a given task. The TSS also holds other data, such as the inner-level stack pointers used when changing to a higher privilege level. See “Task Register” on page 372 for details.
- *Debug Registers*—Debug registers are used to control the software-debug mechanism, and to report information back to a debug utility or application. See “Debug Registers” on page 393 for details.

[AMD Public Use]

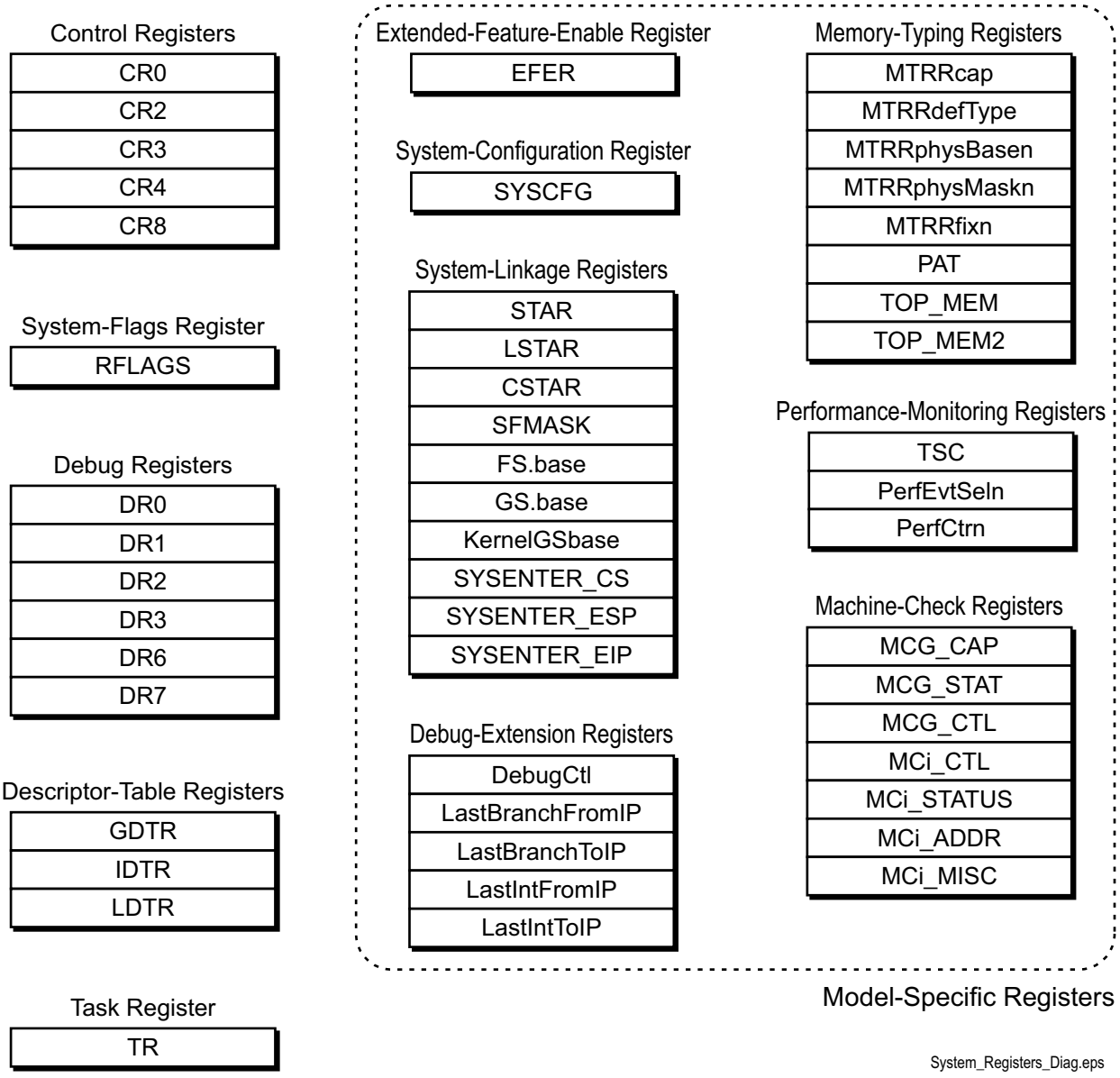


Figure 1-7. System Registers

Also defined as system registers are a number of *model-specific registers* included in the AMD64 architectural definition, and shown in Figure 1-7:

- *Extended-Feature-Enable Register*—The EFER register is used to enable and report status on special features not controlled by the CR*n* control registers. In particular, EFER is used to control activation of long mode. See “Extended Feature Enable Register (EFER)” on page 56 for more information.

[AMD Public Use]

- *System-Configuration Register*—The SYSCFG register is used to enable and configure system-bus features. See “System Configuration Register (SYSCFG)” on page 61 for more information.
- *System-Linkage Registers*—These registers are used by system-linkage instructions to specify operating-system entry points, stack locations, and pointers into system-data structures. See “Fast System Call and Return” on page 175 for details.
- *Memory-Typing Registers*—Memory-typing registers can be used to characterize (type) system memory. Typing memory gives system software control over how instructions and data are cached, and how memory reads and writes are ordered. See “MTRRs” on page 220 for details.
- *Debug-Extension Registers*—These registers control additional software-debug reporting features. See “Debug Registers” on page 393 for details.
- *Performance-Monitoring Registers*—Performance-monitoring registers are used to count processor and system events, or the duration of events. See “Performance Monitoring Counters” on page 413 for more information.
- *Machine-Check Registers*—The machine-check registers control the response of the processor to non-recoverable failures. They are also used to report information on such failures back to system utilities designed to respond to such failures. See “Machine Check Architecture MSRs” on page 303 for more information.

1.5 System-Data Structures

Figure 1-8 on page 18 shows the system-data structures defined for the AMD64 architecture. System-data structures are created and maintained by system software for use by the processor when running in protected mode. A processor running in protected mode uses these data structures to manage memory and protection, and to store program-state information when an interrupt or task switch occurs.

[AMD Public Use]

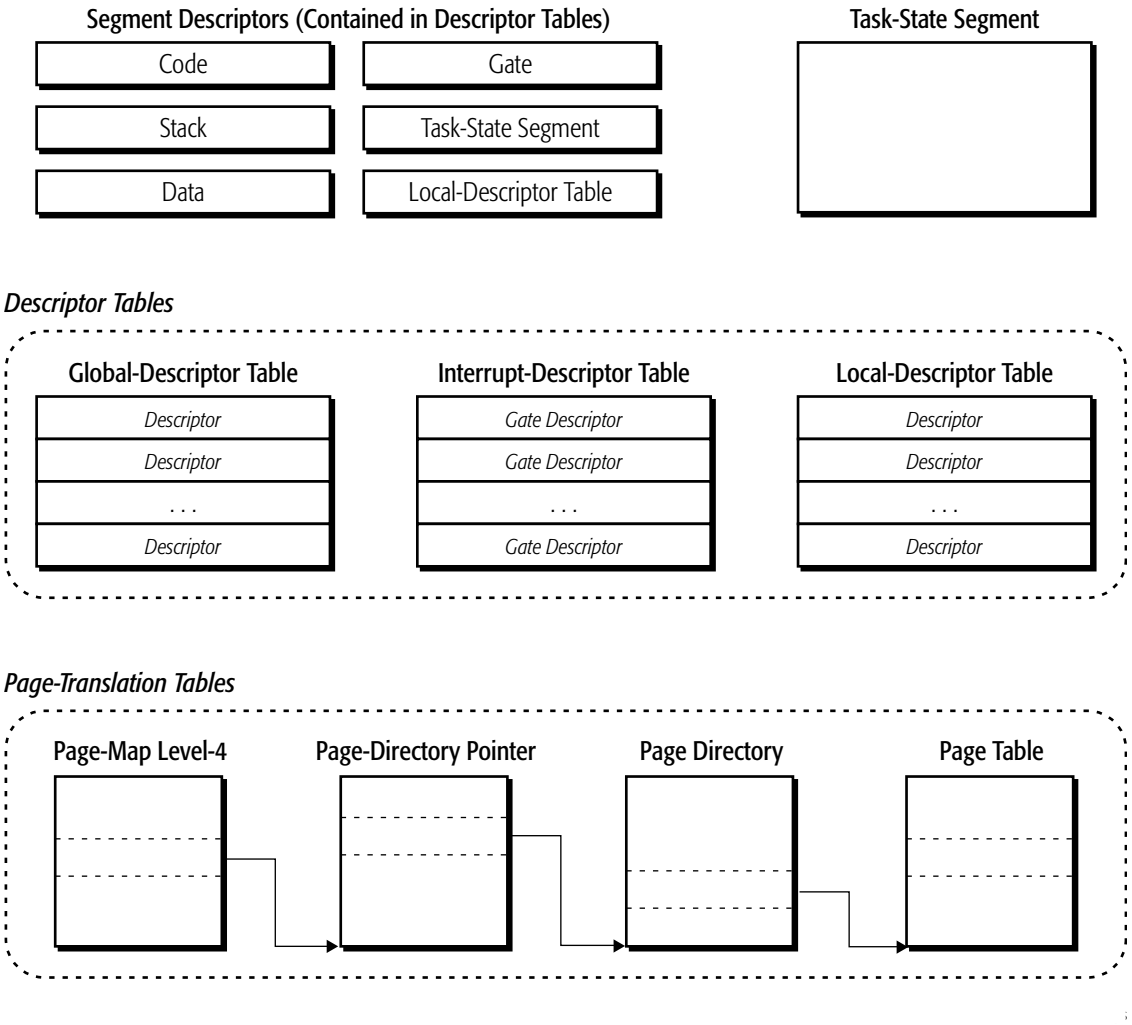


Figure 1-8. System-Data Structures

As shown in Figure 1-8, the system-data structures include:

- *Descriptors*—A descriptor provides information about a segment to the processor, such as its location, size and privilege level. A special type of descriptor, called a *gate*, is used to provide a code selector and entry point for a software routine. Any number of descriptors can be defined, but system software must at a minimum create a descriptor for the currently executing code segment and stack segment. See “Legacy Segment Descriptors” on page 90, and “Long-Mode Segment Descriptors” on page 99 for complete information on descriptors.
- *Descriptor Tables*—As the name implies, descriptor tables hold descriptors. The global-descriptor table holds descriptors available to all programs, while a local-descriptor table holds descriptors used by a single program. The interrupt-descriptor table holds only gate descriptors used by

[AMD Public Use]

interrupt handlers. System software must initialize the global-descriptor and interrupt-descriptor tables, while use of the local-descriptor table is optional. See “Descriptor Tables” on page 84 for more information.

- *Task-State Segment*—The task-state segment is a special segment for holding processor-state information for a specific program, or task. It also contains the stack pointers used when switching to more-privileged programs. The hardware multitasking mechanism uses the state information in the segment when suspending and resuming a task. Calls and interrupts that switch stacks cause the stack pointers to be read from the task-state segment. System software must create at least one task-state segment, even if hardware multitasking is not used. See “Legacy Task-State Segment” on page 374, and “64-Bit Task State Segment” on page 378 for details.
- *Page-Translation Tables*—Use of page translation is optional in protected mode, but it is required in long mode. A four-level page-translation data structure is provided to allow long-mode operating systems to translate a 64-bit virtual-address space into a 52-bit physical-address space. Legacy protected mode can use two- or three-level page-translation data structures. See “Page Translation Overview” on page 129 for more information on page translation.

1.6 Interrupts

The AMD64 architecture provides a mechanism for the processor to automatically suspend (interrupt) software execution and transfer control to an interrupt handler when an interrupt or exception occurs. An interrupt handler is privileged software designed to identify and respond to the cause of an interrupt or exception, and return control back to the interrupted software. *Interrupts* can be caused when system hardware signals an interrupt condition using one of the external-interrupt signals on the processor. Interrupts can also be caused by software that executes an interrupt instruction. *Exceptions* occur when the processor detects an abnormal condition as a result of executing an instruction. The term “interrupts” as used throughout this volume includes both interrupts and exceptions when the distinction is unnecessary.

System software not only sets up the interrupt handlers, but it must also create and initialize the data structures the processor uses to execute an interrupt handler when an interrupt occurs. The data structures include the code-segment descriptors for the interrupt-handler software and any data-segment descriptors for data and stack accesses. Interrupt-gate descriptors must also be supplied. Interrupt gates point to interrupt-handler code-segment descriptors, and the entry point in an interrupt handler. Interrupt gates are stored in the interrupt-descriptor table. The code-segment and data-segment descriptors are stored in the global-descriptor table and, optionally, the local-descriptor table.

When an interrupt occurs, the processor uses the interrupt vector to find the appropriate interrupt gate in the interrupt-descriptor table. The gate points to the interrupt-handler code segment and entry point, and the processor transfers control to that location. Before invoking the interrupt handler, the processor saves information required to return to the interrupted program. For details on how the processor transfers control to interrupt handlers, see “Legacy Protected-Mode Interrupt Control Transfers” on page 272, and “Long-Mode Interrupt Control Transfers” on page 283.

[AMD Public Use]

Refer to “Vectors” on page 247 for a description of each interrupt, and a description of the interrupt mechanism and exception.

1.7 Additional System-Programming Facilities

1.7.1 Hardware Multitasking

A task is any program that the processor can execute, suspend, and later resume executing at the point of suspension. During the time a task is suspended, other tasks are allowed to execute. Each task has its own execution space, consisting of a code segment, data segments, and a stack segment for each privilege level. Tasks can also have their own virtual-memory environment managed by the page-translation mechanism. The state information defining this execution space is stored in the task-state segment (TSS) maintained for each task.

Support for hardware multitasking is provided by implementations of the AMD64 architecture when software is running in legacy mode. Hardware multitasking provides automated mechanisms for switching tasks, saving the execution state of the suspended task, and restoring the execution state of the resumed task. When hardware multitasking is used to switch tasks, the processor takes the following actions:

- The processor automatically suspends execution of the task, allowing any executing instructions to complete and save their results.
- The execution state of a task is saved in the task TSS.
- The execution state of a new task is loaded into the processor from its TSS.
- The processor begins executing the new task at the location specified in the new task TSS.

Use of hardware-multitasking features is optional in legacy mode. Generally, modern operating systems do not use the hardware-multitasking features, and instead perform task management entirely in software. Long mode does not support hardware multitasking at all.

Whether hardware multitasking is used or not, system software must create and initialize at least one task-state segment data-structure. This requirement holds for both long-mode and legacy-mode software. The single task-state segment holds critical pieces of the task execution environment and is referenced during certain control transfers.

Detailed information on hardware multitasking is available in Chapter 12, “Task Management,” along with a full description of the requirements that must be met in initializing a task-state segment when hardware multitasking is not used.

1.7.2 Machine Check

Implementations of the AMD64 architecture support the machine-check exception. This exception is useful in system applications with stringent requirements for reliability, availability, and serviceability. The exception allows specialized system-software utilities to report hardware errors that are generally

[AMD Public Use]

severe and non-recoverable. Providing the capability to report such errors can allow complex system problems to be pinpointed rapidly.

The machine-check exception is described in Chapter 9, “Machine Check Architecture.” Much of the error-reporting capabilities is implementation dependent. For more information, developers of machine-check error-reporting software should refer to the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* or applicable to your product.

1.7.3 Software Debugging

A software-debugging mechanism is provided in hardware to help software developers quickly isolate programming errors. This capability can be used to debug system software and application software alike. Only privileged software can access the debugging facilities. Generally, software-debug support is provided by a privileged application program rather than by the operating system itself.

The facilities supported by the AMD64 architecture allow debugging software to perform the following:

- Set breakpoints on specific instructions within a program.
- Set breakpoints on an instruction-address match.
- Set breakpoints on a data-address match.
- Set breakpoints on specific I/O-port addresses.
- Set breakpoints to occur on task switches when hardware multitasking is used.
- Single step an application instruction-by-instruction.
- Single step only branches and interrupts.
- Record a history of branches and interrupts taken by a program.

The debugging facilities are fully described in “Software-Debug Resources” on page 392. Some processors provide additional, implementation-specific debug support. For more information, refer to the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

1.7.4 Performance Monitoring

For many software developers, the ability to identify and eliminate performance bottlenecks from a program is nearly as important as quickly isolating programming errors. Implementations of the AMD64 architecture provide hardware performance-monitoring resources that can be used by special software applications to identify such bottlenecks. Non-privileged software can access the performance monitoring facilities, but only if privileged software grants that access.

The performance-monitoring facilities allow the counting of events, or the duration of events. Performance-analysis software can use the data to calculate the frequency of certain events, or the time spent performing specific activities. That information can be used to suggest areas for improvement and the types of optimizations that are helpful.

[AMD Public Use]

The performance-monitoring facilities are fully described in “Performance Monitoring Counters” on page 413. The specific events that can be monitored are generally implementation specific. For more information, refer to the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

[AMD Public Use]

2 x86 and AMD64 Architecture Differences

The AMD64 architecture is designed to provide full binary compatibility with all previous AMD implementations of the x86 architecture. This chapter summarizes the new features and architectural enhancements introduced by the AMD64 architecture, and compares those features and enhancements with previous AMD x86 processors. Most of the new capabilities introduced by the AMD64 architecture are available only in long mode (64-bit mode, compatibility mode, or both). However, some of the new capabilities are also available in legacy mode, and are mentioned where appropriate.

The material throughout this chapter assumes the reader has a solid understanding of the x86 architecture. For those who are unfamiliar with the x86 architecture, please read the remainder of this volume before reading this chapter.

2.1 Operating Modes

See “Operating Modes” on page 11 for a complete description of the operating modes supported by the AMD64 architecture.

2.1.1 Long Mode

The AMD64 architecture introduces long mode and its two sub-modes: 64-bit mode and compatibility mode.

64-Bit Mode. 64-bit mode provides full support for 64-bit system software and applications. The new features introduced in support of 64-bit mode are summarized throughout this chapter. To use 64-bit mode, a 64-bit operating system and tool chain are required.

Compatibility Mode. Compatibility mode allows 64-bit operating systems to implement binary compatibility with existing 16-bit and 32-bit x86 applications. It allows these applications to run, without recompilation, under control of a 64-bit operating system in long mode. The architectural enhancements introduced by the AMD64 architecture that support compatibility mode are summarized throughout this chapter.

Unsupported Modes. Long mode does not support the following two operating modes:

- *Virtual-8086 Mode*—The virtual-8086 mode bit (EFLAGS.VM) is ignored when the processor is running in long mode. When long mode is enabled, any attempt to enable virtual-8086 mode is silently ignored. System software must leave long mode in order to use virtual-8086 mode.
- *Real Mode*—Real mode is not supported when the processor is operating in long mode because long mode requires that protected mode be enabled.

2.1.2 Legacy Mode

The AMD64 architecture supports a pure x86 legacy mode, which preserves binary compatibility not only with existing 16-bit and 32-bit applications but also with existing 16-bit and 32-bit operating

[AMD Public Use]

systems. L*egacy mode* supports real mode, protected mode, and virtual-8086 mode. A reset always places the processor in legacy mode (real mode), and the processor continues to run in legacy mode until system software activates long mode. New features added by the AMD64 architecture that are supported in legacy mode are summarized in this chapter.

2.1.3 System-Management Mode

The AMD64 architecture supports system-management mode (SMM). SMM can be entered from both long mode and legacy mode, and SMM can return directly to either mode. The following differences exist between the support of SMM in the AMD64 architecture and the SMM support found in previous processor generations:

- The SMRAM state-save area format is changed to hold the 64-bit processor state. This state-save area format is used regardless of whether SMM is entered from long mode or legacy mode.
- The auto-halt restart and I/O-instruction restart entries in the SMRAM state-save area are one byte instead of two bytes.
- The initial processor state upon entering SMM is expanded to reflect the 64-bit nature of the processor.
- New conditions exist that can cause a processor shutdown while exiting SMM.
- SMRAM caching considerations are modified because the legacy FLUSH# external signal (writeback, if modified, and invalidate) is not supported on implementations of the AMD64 architecture.

See Chapter 10, “System-Management Mode,” for more information on the SMM differences.

2.2 Memory Model

The AMD64 architecture provides enhancements to the legacy memory model to support very large physical-memory and virtual-memory spaces while in long mode. Some of this expanded support for physical memory is available in legacy mode.

2.2.1 Memory Addressing

Virtual-Memory Addressing. Virtual-memory support is expanded to 64 address bits in long mode. This allows up to 16 exabytes of virtual-address space to be accessed. The virtual-address space supported in legacy mode is unchanged.

Physical-Memory Addressing. Physical-memory support is expanded to 52 address bits in long mode and legacy mode. This allows up to 4 petabytes of physical memory to be accessed. The expanded physical-memory support is achieved by using paging and the page-size extensions.

Note that given processor may implement less than the architecturally-defined physical address size of 52 bits.

[AMD Public Use]

Effective Addressing. The effective-address length is expanded to 64 bits in long mode. An effective-address calculation uses 64-bit base and index registers, and sign-extends 8-bit and 32-bit displacements to 64 bits. In legacy mode, effective addresses remain 32 bits long.

2.2.2 Page Translation

The AMD64 architecture defines an expanded page-translation mechanism supporting translation of a 64-bit virtual address to a 52-bit physical address. See “Long-Mode Page Translation” on page 141 for detailed information on the enhancements to page translation in the AMD64 architecture. The enhancements are summarized below.

Physical-Address Extensions (PAE). The AMD64 architecture requires physical-address extensions to be enabled (CR4.PAE=1) before long mode is entered. When PAE is enabled, all paging data-structures are 64 bits, allowing references into the full 52-bit physical-address space supported by the architecture.

Page-Size Extensions (PSE). Page-size extensions (CR4.PSE) are ignored in long mode. Long mode does not support the 4-Mbyte page size enabled by page-size extensions. Long mode does, however, support 4-Kbyte and 2-Mbyte page sizes.

Paging Data Structures. The AMD64 architecture extends the page-translation data structures in support of long mode. The extensions are:

- *Page-map level-5 (PML5)*—Long mode with 5-Level paging support defines a new page translation data structure, the PML5 table. The PML5 table is at the top of the page translation hierarchy and references PML4 tables.
- *Page-map level-4 (PML4)*—Long mode defines a new page-translation data structure, the PML4 table. Without 5-Level paging, the PML4 table is at the top of the page-translation hierarchy and references PDP tables.
- *Page-directory pointer (PDP)*—The PDP tables in long mode are expanded from 4 entries to 512 entries each.
- *Page-directory pointer entry (PDPE)*—Previously undefined fields within the legacy-mode PDPE are defined by the AMD64 architecture.

CR3 Register. The CR3 register is expanded to 64 bits for use in long-mode page translation. When long mode is active, the CR3 register references the base address of either the PML5 or PML4 table. The PML5 table is referenced when 5-Level paging is active (CR4.LA57=1). Otherwise, the PML4 table is referenced. In legacy mode, the upper 32 bits of CR3 are masked by the processor to support legacy page translation. CR3 references the PDP base-address when physical-address extensions are enabled, or the page-directory table base-address when physical-address extensions are disabled.

Legacy-Mode Enhancements. Legacy-mode software can take advantage of the enhancements made to the physical-address extension (PAE) support and page-size extension (PSE) support. The four-level page translation mechanism introduced by long mode is not available to legacy-mode software.

[AMD Public Use]

- *PAE*—When physical-address extensions are enabled (CR4.PAE=1), the AMD64 architecture allows legacy-mode software to load up to 52-bit (maximum size) physical addresses into the PDE and PTE. Note that addresses are expanded to the maximum physical address size supported by the implementation.
- *PSE*—The use of page-size extensions allows legacy mode software to define 4-Mbyte pages using the 32-bit page-translation tables. When page-size extensions are enabled (CR4.PSE=1), the AMD64 architecture enhances the 4-Mbyte PDE to support 40 physical-address bits.

See "Legacy-Mode Page Translation" on page 133 for more information on these enhancements.

2.2.3 Segmentation

In long mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode:

- In compatibility mode, segmentation functions just as it does in legacy mode, using legacy 16-bit or 32-bit protected mode semantics.
- 64-bit mode requires a flat-memory model for creating a flat 64-bit virtual-address space. Much of the segmentation capability present in legacy mode and compatibility mode is disabled when the processor is running in 64-bit mode.

The differences in the segmentation model as defined by the AMD64 architecture are summarized in the following sections. See Chapter 4, "Segmented Virtual Memory," for a thorough description of these differences.

Descriptor-Table Registers. In long mode, the base-address portion of the descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded to 64 bits. The full 64-bit base address can only be loaded by software when the processor is running in 64-bit mode (using the LGDT, LIDT, LLDT, and LTR instructions, respectively). However, the full 64-bit base address is *used* by a processor running in compatibility mode (in addition to 64-bit mode) when making a reference into a descriptor table.

A processor running in legacy mode can only load the low 32 bits of the base address, and the high 32 bits are ignored when references are made to the descriptor tables.

Code-Segment Descriptors. The AMD64 architecture defines a new code-segment descriptor attribute, L (long). In compatibility mode, the processor treats code-segment descriptors as it does in legacy mode, with the exception that the processor recognizes the L attribute. If a code descriptor with L=1 is loaded in compatibility mode, the processor leaves compatibility mode and enters 64-bit mode. In legacy mode, the L attribute is reserved.

The following differences exist for code-segment descriptors in 64-bit mode only:

- The CS base-address field is ignored by the processor.
- The CS limit field is ignored by the processor.
- Only the L (long), D (default size), and DPL (descriptor-privilege level) fields are used by the processor in 64-bit mode. All remaining attributes are ignored.

[AMD Public Use]

Data-Segment Descriptors. The following differences exist for data-segment descriptors in 64-bit mode only:

- The DS, ES, and SS descriptor base-address fields are ignored by the processor.
- The FS and GS descriptor base-address fields are expanded to 64 bits and used in effective-address calculations. The 64 bits of base address are mapped to model-specific registers (MSRs), and can only be loaded using the WRMSR instruction.
- The limit fields and attribute fields of all data-segment descriptors (DS, ES, FS, GS, and SS) are ignored by the processor.

In compatibility mode, the processor treats data-segment descriptors as it does in legacy mode. Compatibility mode ignores the high 32 bits of base address in the FS and GS segment descriptors when calculating an effective address.

System-Segment Descriptors. In 64-bit mode only, The LDT and TSS system-segment descriptor formats are expanded by 64 bits, allowing them to hold 64-bit base addresses. LLDT and LTR instructions can be used to load these descriptors into the LDTR and TR registers, respectively, from 64-bit mode.

In compatibility mode and legacy mode, the *formats* of the LDT and TSS system-segment descriptors are unchanged. Also, unlike code-segment and data-segment descriptors, system-segment descriptor limits *are checked* by the processor in long mode.

Some legacy mode LDT and TSS type-field encodings are illegal in long mode (both compatibility mode and 64-bit mode), and others are redefined to new types. See “System Descriptors” on page 101 for additional information.

Gate Descriptors. The following differences exist between gate descriptors in long mode (both compatibility mode and 64-bit mode) and in legacy mode:

- In long mode, all 32-bit gate descriptors are redefined as 64-bit gate descriptors, and are expanded to hold 64-bit offsets. The length of a gate descriptor in long mode is therefore 128 bits (16 bytes), versus the 64 bits (8 bytes) in legacy mode.
- Some type-field encodings are illegal in long mode, and others are redefined to new types. See “Gate Descriptors” on page 103 for additional information.
- The interrupt-gate and trap-gate descriptors define a new field, called the interrupt-stack table (IST) field.

2.3 Protection Checks

The AMD64 architecture makes the following changes to the protection mechanism in long mode:

- The page-protection-check mechanism is expanded in long mode to include the U/S and R/W protection bits stored in the PML5, PML4 and PDP entries.
- Several system-segment types and gate-descriptor types that are legal in legacy mode are illegal in long mode (compatibility mode and 64-bit mode) and fail type checks when used in long mode.

[AMD Public Use]

- Segment-limit checks are disabled in 64-bit mode for the CS, DS, ES, FS, GS, and SS segments. Segment-limit checks remain enabled for the LDT, GDT, IDT and TSS system segments.

  All segment-limit checks are performed in compatibility mode.
- Code and data segments used in 64-bit mode are treated as both readable and writable.

See “Page-Protection Checks” on page 163 and “Segment-Protection Overview” on page 106 for detailed information on the protection-check changes.

2.4 Registers

The AMD64 architecture adds additional registers to the architecture, and in many cases expands the size of existing registers to 64 bits. The 80-bit floating-point stack registers and their overlaid 64-bit MMX™ registers are not modified by the AMD64 architecture.

2.4.1 General-Purpose Registers

In 64-bit mode, the general-purpose registers (GPRs) are 64 bits wide, and eight additional GPRs are available. The GPRs are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and the new R8–R15 registers. To access the full 64-bit operand size, or the new R8–R15 registers, an instruction must include a new REX instruction-prefix byte (see “REX Prefixes” on page 30 for a summary of this prefix).

In compatibility and legacy modes, the GPRs consist only of the eight legacy 32-bit registers. All legacy rules apply for determining operand size.

2.4.2 YMM/XMM Registers

In 64-bit mode, eight additional YMM/XMM registers are available, YMM/XMM8–15. A REX instruction prefix is used to access these registers. In compatibility and legacy modes, only registers YMM/XMM0–7 are accessible.

2.4.3 Flags Register

The flags register is expanded to 64 bits, and is called RFLAGS. All 64 bits can be accessed in 64-bit mode, but the upper 32 bits are reserved and always read back as zeros. Compatibility mode and legacy mode can read and write only the lower-32 bits of RFLAGS (the legacy EFLAGS).

2.4.4 Instruction Pointer

In long mode, the instruction pointer is extended to 64 bits, to support 64-bit code offsets. This 64-bit instruction pointer is called RIP.

[AMD Public Use]

2.4.5 Stack Pointer

In 64-bit mode, the size of the stack pointer, RSP, is always 64 bits. The stack size is not controlled by a bit in the SS descriptor, as it is in compatibility or legacy mode, nor can it be overridden by an instruction prefix. Address-size overrides are ignored for implicit stack references.

2.4.6 Control Registers

The AMD64 architecture defines several enhancements to the control registers (CR*n*). In long mode, all control registers are expanded to 64 bits, although the entire 64 bits can be read and written only from 64-bit mode. A new control register, the task-priority register (CR8 or TPR) is added, and can be read and written from 64-bit mode. Last, the function of the page-enable bit (CR0.PG) is expanded. When long mode is enabled, the PG bit is used to activate and deactivate long mode.

2.4.7 Debug Registers

In long mode, all debug registers are expanded to 64 bits, although the entire 64 bits can be read and written only from 64-bit mode. Expanded register encodings for the decode registers allow up to eight new registers to be defined (DR8–DR15), although presently those registers are not supported by the AMD64 architecture.

2.4.8 Extended Feature Register (EFER)

The EFER is expanded by the AMD64 architecture to include a long-mode-enable bit (LME), and a long-mode-active bit (LMA). These new bits can be accessed from legacy mode and long mode.

2.4.9 Memory Type Range Registers (MTRRs)

The legacy MTRRs are architecturally defined as 64 bits, and can accommodate the maximum 52-bit physical address allowed by the AMD64 architecture. From both long mode and legacy mode, implementations of the AMD64 architecture reference the entire 52-bit physical-address value stored in the MTRRs. Long mode and legacy mode system software can update all 64 bits of the MTRRs to manage the expanded physical-address space.

2.4.10 Other Model-Specific Registers (MSRs)

Several other MSRs have fields holding physical addresses. Examples include the APIC-base register and top-of-memory register. Generally, any model-specific register that contains a physical address is defined architecturally to be 64 bits wide, and can accommodate the maximum physical-address size defined by the AMD64 architecture. When physical addresses are read from MSRs by the processor, the entire value is read regardless of the operating mode. In legacy implementations, the high-order MSR bits are reserved, and software must write those values with zeros. In legacy mode on AMD64 architecture implementations, software can read and write all supported high-order MSR bits.

[AMD Public Use]

2.5 Instruction Set

2.5.1 REX Prefixes

REX prefixes are used in 64-bit mode to:

- Specify the new GPRs and YMM/XMM registers.
- Specify a 64-bit operand size.
- Specify additional control registers. One additional control register, CR8, is defined in 64-bit mode.
- Specify additional debug registers (although none are currently defined).

Not all instructions require a REX prefix. The prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Default 64-Bit Operand Size. In 64-bit mode, two groups of instructions have a default operand size of 64 bits and thus do not need a REX prefix for this operand size:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP. See “Instructions that Reference RSP” on page 31 for additional information.

2.5.2 Segment-Override Prefixes in 64-Bit Mode

In 64-bit mode, the DS, ES, SS, and CS segment-override prefixes have no effect. These four prefixes are no longer treated as segment-override prefixes in the context of multiple-prefix rules. Instead, they are treated as null prefixes. An exception exists when Upper Address Ignore (UAI) is enabled. When UAI is enabled, these prefixes may control if the canonical check for a memory pointer is modified by UAI.

The FS and GS segment-override prefixes are treated as segment-override prefixes in 64-bit mode. Use of the FS and GS prefixes cause their respective segment bases to be added to the effective address calculation. See “FS and GS Registers in 64-Bit Mode” on page 82 for additional information on using these segment registers.

2.5.3 Operands and Results

The AMD64 architecture provides support for using 64-bit operands and generating 64-bit results when operating in 64-bit mode.

Operand-Size Overrides. In 64-bit mode, the default operand size is 32 bits. A REX prefix can be used to specify a 64-bit operand size. Software uses a legacy operand-size (66h) prefix to toggle to 16-bit operand size. The REX prefix takes precedence over the legacy operand-size prefix.

Zero Extension of Results. In 64-bit mode, when performing 32-bit operations with a GPR destination, the processor zero-extends the 32-bit result into the full 64-bit destination. Both 8-bit and

[AMD Public Use]

16-bit operations on GPRs preserve all unwritten upper bits of the destination GPR. This is consistent with legacy 16-bit and 32-bit semantics for partial-width results.

2.5.4 Address Calculations

The AMD64 architecture modifies aspects of effective-address calculation to support 64-bit mode. These changes are summarized in the following sections. See “Memory Addressing” in Volume 1 for details.

Address-Size Overrides. In 64-bit mode, the default-address size is 64 bits. The address size can be overridden to 32 bits by using the address-size prefix (67h). 16-bit addresses are not supported in 64-bit mode. In compatibility mode and legacy mode, address-size overrides function the same as in x86 legacy architecture.

Displacements and Immediates. Generally, displacement and immediate values in 64-bit mode are not extended to 64 bits. They are still limited to 32 bits and are sign extended during effective-address calculations. In 64-bit mode, however, support is provided for some 64-bit displacement and immediate forms of the MOV instruction.

Zero Extending 16-Bit and 32-Bit Addresses. All 16-bit and 32-bit address calculations are zero-extended in long mode to form 64-bit addresses. Address calculations are first truncated to the effective-address size of the current mode (64-bit mode or compatibility mode), as overridden by any address-size prefix. The result is then zero-extended to the full 64-bit address width.

RIP-Relative Addressing. A new addressing form, RIP-relative (instruction-pointer relative) addressing, is implemented in 64-bit mode. The effective address is formed by adding the displacement to the 64-bit RIP of the next instruction.

2.5.5 Instructions that Reference RSP

With the exception of far branches, all instructions that implicitly reference the 64-bit stack pointer, RSP, default to a 64-bit operand size in 64-bit mode (see Table 2-1 for a listing). Pushes and pops of 32-bit stack values are not possible in 64-bit mode with these instructions, but they can be overridden to 16 bits.

Table 2-1. Instructions That Reference RSP

Mnemonic	Opcode (hex)	Description
ENTER	C8	Create Procedure Stack Frame
LEAVE	C9	Delete Procedure Stack Frame
POP reg/mem	8F/0	Pop Stack (register or memory)
POP reg	58-5F	Pop Stack (register)
POP FS	0F A1	Pop Stack into FS Segment Register
POP GS	0F A9	Pop Stack into GS Segment Register

[AMD Public Use]

Table 2-1. Instructions That Reference RSP (continued)

Mnemonic	Opcode (hex)	Description
POPF, POPFD, POPFQ	9D	Pop to rFLAGS Word, Doubleword, or Quadword
PUSH imm32	68	Push onto Stack (sign-extended doubleword)
PUSH imm8	6A	Push onto Stack (sign-extended byte)
PUSH reg/mem	FF/6	Push onto Stack (register or memory)
PUSH reg	50-57	Push onto Stack (register)
PUSH FS	0F A0	Push FS Segment Register onto Stack
PUSH GS	0F A8	Push GS Segment Register onto Stack
PUSHF, PUSHFD, PUSHFQ	9C	Push rFLAGS Word, Doubleword, or Quadword onto Stack

2.5.6 Branches

The AMD64 architecture expands two branching mechanisms to accommodate branches in the full 64-bit virtual-address space:

- In 64-bit mode, near-branch semantics are redefined.
- In both 64-bit and compatibility modes, a 64-bit call-gate descriptor is defined for far calls.

In addition, enhancements are made to the legacy SYSCALL and SYSRET instructions.

Near Branches. In 64-bit mode, the operand size for all near branches defaults to 64 bits (see Table 2-2 for a listing). Therefore, these instructions update the full 64-bit RIP without the need for a REX operand-size prefix. The following aspects of near branches default to 64 bits:

- Truncation of the instruction pointer.
- Size of a stack pop or stack push, resulting from a CALL or RET.
- Size of a stack-pointer increment or decrement, resulting from a CALL or RET.
- Size of operand fetched by indirect-branch operand size.

The operand size for near branches can be overridden to 16 bits in 64-bit mode.

Table 2-2. 64-Bit Mode Near Branches, Default 64-Bit Operand Size

Mnemonic	Opcode (hex)	Description
CALL	E8, FF/2	Call Procedure Near
Jcc	many	Jump Conditional Near
JMP	E9, EB, FF/4	Jump Near
LOOP	E2	Loop

[AMD Public Use]

Table 2-2. 64-Bit Mode (continued)Near Branches, Default 64-Bit Operand

Mnemonic	Opcode (hex)	Description
LOOPcc	E0, E1	Loop Conditional
JrCXZ	E3	Jump Near if ECX/RCX Zero
RET	C3, C2	Return From Call (near)

The address size of near branches is not forced in 64-bit mode. Such addresses are 64 bits by default, but they can be overridden to 32 bits by a prefix.

The size of the displacement field for relative branches is still limited to 32 bits.

Far Branches Through Long-Mode Call Gates. Long mode redefines the 32-bit call-gate descriptor type as a 64-bit call-gate descriptor and expands the call-gate descriptor size to hold a 64-bit offset. The long-mode call-gate descriptor allows far branches to reference any location in the supported virtual-address space. In long mode, the call-gate mechanism is changed as follows:

- In long mode, CALL and JMP instructions that reference call-gates must reference 64-bit call gates.
- A 64-bit call-gate descriptor must reference a 64-bit code-segment.
- When a control transfer is made through a 64-bit call gate, the 64-bit target address is read from the 64-bit call-gate descriptor. The base address in the target code-segment descriptor is ignored.

Stack Switching. Automatic stack switching is also modified when a control transfer occurs through a call gate in long mode:

- The target-stack pointer read from the TSS is a 64-bit RSP value.
- The SS register is loaded with a null selector. Setting the new SS selector to null allows nested control transfers in 64-bit mode to be handled properly. The SS.RPL value is updated to remain consistent with the newly loaded CPL value.
- The size of pushes onto the new stack is modified to accommodate the 64-bit RIP and RSP values.
- Automatic parameter copying is not supported in long mode.

Far Returns. In long mode, far returns can load a null SS selector from the stack under the following conditions:

- The target operating mode is 64-bit mode.
- The target CPL<3.

Allowing RET to load SS with a null selector under these conditions makes it possible for the processor to unnest far CALLs (and interrupts) in long mode.

Task Gates. Control transfers through task gates are not supported in long mode.

[AMD Public Use]

Branches to 64-Bit Offsets. Because immediate values are generally limited to 32 bits, the only way a full 64-bit absolute RIP can be specified in 64-bit mode is with an indirect branch. For this reason, direct forms of far branches are eliminated from the instruction set in 64-bit mode.

SYSCALL and SYSRET Instructions. The AMD64 architecture expands the function of the legacy SYSCALL and SYSRET instructions in long mode. In addition, two new STAR registers, LSTAR and CSTAR, are provided to hold the 64-bit target RIP for the instructions when they are executed in long mode. The legacy STAR register is not expanded in long mode. See "SYSCALL and SYSRET" on page 176 for additional information.

SWAPGS Instruction. The AMD64 architecture provides the SWAPGS instruction as a fast method for system software to load a pointer to system data-structures. SWAPGS is valid only in 64-bit mode. An undefined-opcode exception (#UD) occurs if software attempts to execute SWAPGS in legacy mode or compatibility mode. See "SWAPGS Instruction" on page 178 for additional information.

SYSENTER and SYSEXIT Instructions. The SYSENTER and SYSEXIT instructions are invalid in long mode, and result in an invalid opcode exception (#UD) if software attempts to use them. Software should use the SYSCALL and SYSRET instructions when running in long mode. See "SYSENTER and SYSEXIT (Legacy Mode Only)" on page 178 for additional information.

2.5.7 NOP Instruction

The legacy x86 architecture commonly uses opcode 90h as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h specially in order to preserve this NOP definition. This is necessary because opcode 90h is actually the XCHG EAX, EAX instruction in the legacy architecture. Without special handling in 64-bit mode, the instruction would not be a true no-operation. Therefore, in 64-bit mode the processor treats opcode 90h (the legacy XCHG EAX, EAX instruction) as a true NOP, regardless of a REX operand-size prefix.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction. Unless a 64-bit operand size is specified using a REX prefix byte, using the two-byte form of XCHG to exchange a register with itself does not result in a no-operation, because the default operation size is 32 bits in 64-bit mode.

2.5.8 Single-Byte INC and DEC Instructions

In 64-bit mode, the legacy encodings for the 16 single-byte INC and DEC instructions (one for each of the eight GPRs) are used to encode the REX prefix values. The functionality of these INC and DEC instructions is still available, however, using the ModRM forms of those instructions (opcodes FF /0 and FF /1). See "Single-Byte INC and DEC Instructions in 64-Bit Mode" in Volume 3 for additional information.

2.5.9 MOVSXD Instruction

MOVSXD is a new instruction in 64-bit mode (the legacy ARPL instruction opcode, 63h, is reassigned as the MOVSXD opcode). It reads a fixed-size 32-bit source operand from a register or memory and (if a REX prefix is used with the instruction) sign-extends the value to 64 bits. MOVSXD

[AMD Public Use]

is analogous to the MOVSX instruction, which sign-extends a byte to a word or a word to a doubleword, depending on the effective operand size. See the instruction reference page for the MOVSXD instruction in Volume 3 for additional information.

2.5.10 Invalid Instructions

Table 2-3 lists instructions that are illegal in 64-bit mode. Table 2-4 on page 36 lists instructions that are invalid in long mode (both compatibility mode and 64-bit mode). Attempted use of these instructions causes an invalid-opcode exception (#UD) to occur.

Table 2-3. Invalid Instructions in 64-Bit Mode

Mnemonic	Opcode (hex)	Description
AAA	37	ASCII Adjust After Addition
AAD	D5	ASCII Adjust Before Division
AAM	D4	ASCII Adjust After Multiply
AAS	3F	ASCII Adjust After Subtraction
BOUND	62	Check Array Bounds
CALL (far)	9A	Procedure Call Far (absolute)
DAA	27	Decimal Adjust after Addition
DAS	2F	Decimal Adjust after Subtraction
INTO	CE	Interrupt to Overflow Vector
JMP (far)	EA	Jump Far (absolute)
LDS	C5	Load DS Segment Register
LES	C4	Load ES Segment Register
POP DS	1F	Pop Stack into DS Segment
POP ES	07	Pop Stack into ES Segment
POP SS	17	Pop Stack into SS Segment
POPA, POPAD	61	Pop All to GPR Words or Doublewords
PUSH CS	0E	Push CS Segment Selector onto Stack
PUSH DS	1E	Push DS Segment Selector onto Stack
PUSH ES	06	Push ES Segment Selector onto Stack
PUSH SS	16	Push SS Segment Selector onto Stack
PUSHA, PUSHAD	60	Push All GPR Words or Doublewords onto Stack
Redundant Grp1 (undocumented)	82	Redundant encoding of group1 Eb,Ib opcodes
SALC (undocumented)	D6	Set AL According to CF

[AMD Public Use]

Table 2-4. Invalid Instructions in Long Mode

Mnemonic	Opcode (hex)	Description
SYSENTER	0F 34	System Call
SYSEXIT	0F 35	System Return

2.5.11 Reassigned Opcodes

Table 2-5 below lists opcodes that are assigned functions in 64-bit mode that differ from their legacy functions.

Table 2-5. Opcodes Reassigned in 64-Bit Mode

Opcode (hex)	Compatibility and Legacy Modes	64-Bit Mode
63	ARPL—Adjust Requestor Privilege Level	MOVSXD—Move Doubleword with Sign Extension
40–4F	DEC—Decrement by 1 INC—Increment by 1	REX Prefix
***Note**: Two-byte versions of DEC and INC are still available in 64-bit mode.*		

2.5.12 FXSAVE and FXRSTOR Instructions

The FXSAVE and FXRSTOR instructions are used to save and restore the entire 128-bit media (XMM), 64-bit media, and x87 instruction-set environment during a context switch. The AMD64 architecture modifies the memory format used by these instructions in order to save and restore the full 64-bit instruction and data pointers, as well as the XMM8–15 registers. Selection of the 32-bit legacy format or the expanded 64-bit format is accomplished by using the corresponding operand size with the FXSAVE and FXRSTOR instructions. When 64-bit software executes an FXSAVE and FXRSTOR with a 32-bit operand size (no operand-size override) the 32-bit legacy format is used. When 64-bit software executes an FXSAVE and FXRSTOR with a 64-bit operand size, the 64-bit format is used.

For more information on the save area formats, see Section 11.4.4, “Saving Media and x87 Execution Unit State,” on page 351

If the fast FXSAVE/FXRSTOR (FFXSR) feature is enabled in EFER, FXSAVE and FXRSTOR do not save or restore the XMM0–15 registers when executed in 64-bit mode at CPL 0. The x87 environment and MXCSR are saved whether fast FXSAVE/FXRSTOR is enabled or not. The fast FXSAVE/FXRSTOR feature has no effect on FXSAVE/FXRSTOR in non 64-bit mode or when CPL > 0.

[AMD Public Use]

Software can use the CPUID instruction to determine whether the fast FXSAVE/FXRSTOR feature is available (CPUID Fn8000_0001h_EDX[FFXSR]). For information on using the CPUID instruction to obtain processor feature information, see Section 3.3, “Processor Feature Identification,” on page 73.

2.6 Interrupts and Exceptions

When a processor is running in long mode, an interrupt or exception causes the processor to enter 64-bit mode. All long-mode interrupt handlers must be implemented as 64-bit code. The AMD64 architecture expands the legacy interrupt-processing and exception-processing mechanism to support handling of interrupts by 64-bit operating systems and applications. The changes are summarized in the following sections. See “Long-Mode Interrupt Control Transfers” on page 283 for detailed information on these changes.

2.6.1 Interrupt Descriptor Table

The long-mode interrupt-descriptor table (IDT) must contain 64-bit mode interrupt-gate or trap-gate descriptors for all interrupts or exceptions that can occur while the processor is running in long mode. Task gates cannot be used in the long-mode IDT, because control transfers through task gates are not supported in long mode. In long mode, the IDT index is formed by scaling the interrupt vector by 16. In legacy protected mode, the IDT is indexed by scaling the interrupt vector by eight.

2.6.2 Stack Frame Pushes

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-frame pushes, and SS:eSP is pushed only on a CPL change. In long mode, the size of interrupt stack-frame pushes is fixed at eight bytes, because interrupts are handled in 64-bit mode. Long mode interrupts also cause SS:RSP to be pushed unconditionally, rather than pushing only on a CPL change.

2.6.3 Stack Switching

Legacy mode provides a mechanism to automatically switch stack frames in response to an interrupt. In long mode, a slightly modified version of the legacy stack-switching mechanism is implemented, and an alternative stack-switching mechanism—called the interrupt stack table (IST)—is supported.

Long-Mode Stack Switches. When stacks are switched as part of a long-mode privilege-level change resulting from an interrupt, the following occurs:

- The target-stack pointer read from the TSS is a 64-bit RSP value.
- The SS register is loaded with a null selector. Setting the new SS selector to null allows nested control transfers in 64-bit mode to be handled properly. The SS.RPL value is cleared to 0.
- The old SS and RSP are saved on the new stack.

Interrupt Stack Table. In long mode, a new interrupt stack table (IST) mechanism is available as an alternative to the modified legacy stack-switching mechanism. The IST mechanism unconditionally switches stacks when it is enabled. It can be enabled for individual interrupt vectors using a field in the

[AMD Public Use]

IDT entry. This allows mixing interrupt vectors that use the modified legacy mechanism with vectors that use the IST mechanism. The IST pointers are stored in the long-mode TSS. The IST mechanism is only available when long mode is enabled.

2.6.4 IRET Instruction

In compatibility mode, IRET pops SS:eSP off the stack only if there is a CPL change. This allows legacy applications to run properly in compatibility mode when using the IRET instruction.

In 64-bit mode, IRET unconditionally pops SS:eSP off of the interrupt stack frame, even if the CPL does not change. This is done because the original interrupt always pushes SS:RSP. Because interrupt stack-frame pushes are always eight bytes in long mode, an IRET from a long-mode interrupt handler (64-bit code) must pop eight-byte items off the stack. This is accomplished by preceding the IRET with a 64-bit REX operand-size prefix.

In long mode, an IRET can load a null SS selector from the stack under the following conditions:

- The target operating mode is 64-bit mode.
- The target CPL< 3.

Allowing IRET to load SS with a null selector under these conditions makes it possible for the processor to unnest interrupts (and far CALLs) in long mode.

2.6.5 Task-Priority Register (CR8)

The AMD64 architecture allows software to define up to 15 external interrupt-priority classes. Priority classes are numbered from 1 to 15, with priority-class 1 being the lowest and priority-class 15 the highest.

A new control register (CR8) is introduced by the AMD64 architecture for managing priority classes. This register, also called the *task-priority register* (TPR), uses the four low-order bits for specifying a task priority. How external interrupts are organized into these priority classes is implementation dependent. See “External Interrupt Priorities” on page 269 for information on this feature.

2.6.6 New Exception Conditions

The AMD64 architecture defines a number of new conditions that can cause an exception to occur when the processor is running in long mode. Many of the conditions occur when software attempts to use an address that is not in canonical form. See “Vectors” on page 247 for information on the new exception conditions that can occur in long mode.

2.7 Hardware Task Switching

The legacy hardware task-switch mechanism is disabled when the processor is running in long mode. However, long mode requires system software to create data structures for a single task—the long-mode task.

[AMD Public Use]

- *TSS Descriptors*—A new TSS-descriptor type, the 64-bit TSS type, is defined for use in long mode. It is the only valid TSS type that can be used in long mode, and it must be loaded into the TR by executing the LTR instruction in *64-bit mode*. See “TSS Descriptor” on page 371 for additional information.
- *Task Gates*—Because the legacy task-switch mechanism is not supported in long mode, *software cannot use task gates in long mode*. Any attempt to transfer control to another task through a task gate causes a general-protection exception (#GP) to occur.
- *Task-State Segment*—A 64-bit task state segment (TSS) is defined for use in long mode. This new TSS format contains 64-bit stack pointers (RSP) for privilege levels 0–2, interrupt-stack-table (IST) pointers, and the I/O-map base address. See “64-Bit Task State Segment” on page 378 for additional information.

2.8 Long-Mode vs. Legacy-Mode Differences

Table 2-6 on page 39 summarizes several major system-programming differences between 64-bit mode and legacy protected mode. The third column indicates whether the difference also applies to compatibility mode. “Differences Between Long Mode and Legacy Mode” in Volume 3 summarizes the application-programming model differences.

Table 2-6. Differences Between Long Mode and Legacy Mode

Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
x86 Modes	Real and virtual-8086 modes not supported	Yes
Task Switching	Task switching not supported	Yes
Addressing	64-bit virtual addresses	No
	4-level paging structures	Yes
	PAE must always be enabled	
Loaded Segment (Usage during memory reference)	CS, DS, ES, SS segment bases are ignored	No
	CS, DS, ES, FS, GS, SS segment limits are ignored	
	DS, ES, FS, GS attribute are ignored	
	CS, DS, ES, SS Segment prefixes are ignored	
Exception and Interrupt Handling	All pushes are 8 bytes	Yes
	IDT entries are expanded to 16 bytes	
	SS is not changed for stack switch	
	SS:RSP is pushed unconditionally	

[AMD Public Use]

Table 2-6. Differences Between Long Mode and Legacy Mode (continued)

Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
Call Gates	All pushes are 8 bytes	Yes
	16-bit call gates are illegal	
	32-bit call gate type is redefined as 64-bit call gate and is expanded to 16 bytes	
	SS is not changed for stack switch	
System-Descriptor Registers	GDT, IDT, LDT, TR base registers expanded to 64 bits	Yes
System-Descriptor Table Entries and Pseudo-Descriptors	LGDT and LIDT use expanded 10-byte pseudo-descriptors	No
	LLDT and LTR use expanded 16-byte table entries	

[AMD Public Use]

3 System Resources

The operating system manages the software-execution environment and general system operation through the use of system resources. These resources consist of system registers (control registers and model-specific registers) and system-data structures (memory-management and protection tables). The system-control registers are described in detail in this chapter; many of the features they control are described elsewhere in this volume. The model-specific registers supported by the AMD64 architecture are introduced in this chapter.

Because of their complexity, system-data structures are described in separate chapters. Refer to the following chapters for detailed information on these data structures:

- Descriptors and descriptor tables are described in Section 4.4 “Segmentation Data Structures and Registers,” on page 77.
- Page-translation tables are described in Section 5.2 “Legacy-Mode Page Translation,” on page 133 and Section 5.3 “Long-Mode Page Translation,” on page 141.
- The task-state segment is described in Section 12.2.4 “Legacy Task-State Segment,” on page 374 and Section 12.2.5 “64-Bit Task State Segment,” on page 378.

Not all processor implementations are required to support all possible features. The last section in this chapter addresses processor-feature identification. System software uses the capabilities described in that section to determine which features are supported so that the appropriate service routines are loaded.

3.1 System-Control Registers

The registers that control the AMD64 architecture operating environment include:

- *CR0*—Provides operating-mode controls and some processor-feature controls.
- *CR2*—This register is used by the page-translation mechanism. It is loaded by the processor with the page-fault virtual address when a page-fault exception occurs.
- *CR3*—This register is also used by the page-translation mechanism. It contains the base address of the highest-level page-translation table, and also contains cache controls for the specified table.
- *CR4*—This register contains additional controls for various operating-mode features.
- *CR8*—This new register, accessible in 64-bit mode using the REX prefix, is introduced by the AMD64 architecture. CR8 is used to prioritize external interrupts and is referred to as the *task-priority register* (TPR).
- *RFLAGS*—This register contains processor-status and processor-control fields. The status and control fields are used primarily in the management of virtual-8086 mode, hardware multitasking, and interrupts.

[AMD Public Use]

- *EFER*—This model-specific register contains status and controls for additional features not managed by the CR0 and CR4 registers. Included in this register are the long-mode enable and activation controls introduced by the AMD64 architecture.

Control registers CR1, CR5–CR7, and CR9–CR15 are reserved.

In legacy mode, all control registers and RFLAGS are 32 bits. The EFER register is 64 bits in all modes. The AMD64 architecture expands all 32-bit system-control registers to 64 bits. In 64-bit mode, the MOV CR*n* instructions read or write all 64 bits of these registers (operand-size prefixes are ignored). In compatibility and legacy modes, control-register writes fill the low 32 bits with data and the high 32 bits with zeros, and control-register reads return only the low 32 bits.

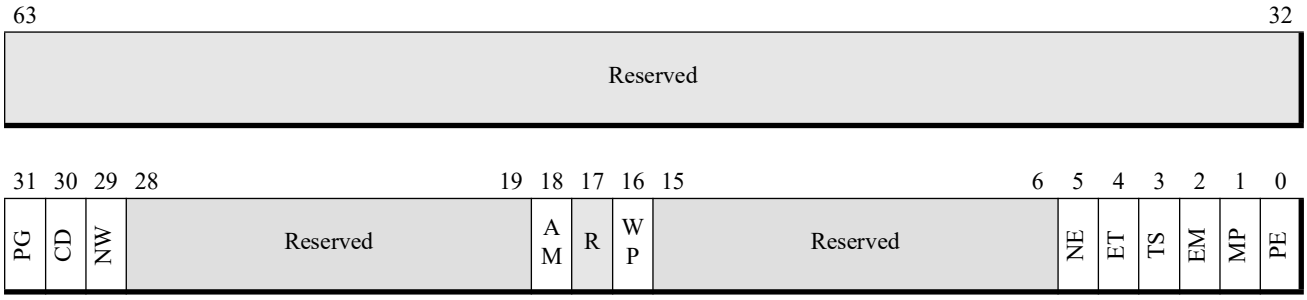
In 64-bit mode, the high 32 bits of CR0 and CR4 are reserved and must be written with zeros. Writing a 1 to any of the high 32 bits results in a general-protection exception, #GP(0). All 64 bits of CR2 are writable. However, the MOV CR*n* instructions *do not* check that addresses written to CR2 are within the virtual-address limitations of the processor implementation.

All CR3 bits are writable, except for unimplemented physical address bits, which must be cleared to 0.

The upper 32 bits of RFLAGS are always read as zero by the processor. Attempts to load the upper 32 bits of RFLAGS with anything other than zero are ignored by the processor.

3.1.1 CR0 Register

The CR0 register is shown in Figure 3-1 on page 43. The legacy CR0 register is identical to the low 32 bits of this register (CR0 bits 31:0).



[AMD Public Use]

Bits	Mnemonic	Description	Access type
63:32	Reserved		MBZ
31	PG	Paging	R/W
30	CD	Cache Disable	R/W
29	NW	Not Writethrough	R/W
28:19	Reserved	do not change	
18	AM	Alignment Mask	R/W
17	Reserved	do not change	
16	WP	Write Protect	R/W[1]
15:6	Reserved	do not change	
5	NE	Numeric Error	R/W
4	ET	Extension Type	R
3	TS	Task Switched	R/W
2	EM	Emulation	R/W
1	MP	Monitor Coprocessor	R/W
0	PE	Protection Enabled	R/W

1. Conditionally -- see description below.

Figure 3-1. Control Register 0 (CR0)

The functions of the CR0 control bits are (unless otherwise noted, all bits are read/write):

Protected-Mode Enable (PE) Bit. Bit 0. Software enables protected mode by setting PE to 1, and disables protected mode by clearing PE to 0. When the processor is running in protected mode, segment-protection mechanisms are enabled.

See Section 4.9 “Segment-Protection Overview,” on page 106 for information on the segment-protection mechanisms.

Monitor Coprocessor (MP) Bit. Bit 1. Software uses the MP bit with the task-switched control bit (CR0.TS) to control whether execution of the WAIT/FWAIT instruction causes a device-not-available exception (#NM) to occur, as follows:

- If both the monitor-coprocessor and task-switched bits are set (CR0.MP=1 *and* CR0.TS=1), then executing the WAIT/FWAIT instruction causes a device-not-available exception (#NM).
- If either the monitor-coprocessor or task-switched bits are clear (CR0.MP=0 *or* CR0.TS=0), then executing the WAIT/FWAIT instruction proceeds normally.

Software typically should set MP to 1 if the processor implementation supports x87 instructions. This allows the CR0.TS bit to completely control when the x87-instruction context is saved as a result of a task switch.

Emulate Coprocessor (EM) Bit. Bit 2. Software forces all x87 instructions to cause a device-not-available exception (#NM) by setting EM to 1. Likewise, setting EM to 1 forces an invalid-opcode exception (#UD) when an attempt is made to execute any of the 64-bit or 128-bit media instructions except the FXSAVE and FXRSTOR instructions. Attempting to execute these instructions when EM is

[AMD Public Use]

set results in an #NM exception instead.The exception handlers can emulate these instruction types if desired. Setting the EM bit to 1 does not cause an #NM exception when the WAIT/FWAIT instruction is executed.

Task Switched (TS) Bit. Bit 3. When an attempt is made to execute an x87 or media instruction while TS=1, a device-not-available exception (#NM) occurs. Software can use this mechanism—sometimes referred to as "lazy context-switching"—to save the unit contexts before executing the next instruction of those types. As a result, the x87 and media instruction-unit contexts are saved only when necessary as a result of a task switch.

When a hardware task switch occurs, TS is automatically set to 1. System software that implements software task-switching rather than using the hardware task-switch mechanism can still use the TS bit to control x87 and media instruction-unit context saves. In this case, the task-management software uses a MOV CR0 instruction to explicitly set the TS bit to 1 during a task switch. Software can clear the TS bit by either executing the CLTS instruction or by writing to the CR0 register directly. Long-mode system software can use this approach even though the hardware task-switch mechanism is not supported in long mode.

The CR0.MP bit controls whether the WAIT/FWAIT instruction causes an #NM exception when TS=1.

Extension Type (ET) Bit. Bit 4, read-only. In some early x86 processors, software set ET to 1 to indicate support of the 387DX math-coprocessor instruction set. This bit is now reserved and forced to 1 by the processor. Software cannot clear this bit to 0.

Numeric Error (NE) Bit. Bit 5. Clearing the NE bit to 0 disables internal control of x87 floating-point exceptions and enables external control. When NE is cleared to 0, the IGNNE# input signal controls whether x87 floating-point exceptions are ignored:

- When IGNNE# is 1, x87 floating-point exceptions are ignored.
- When IGNNE# is 0, x87 floating-point exceptions are reported by setting the FERR# input signal to 1. External logic can use the FERR# signal as an external interrupt.

When NE is set to 1, internal control over x87 floating-point exception reporting is enabled and the external reporting mechanism is disabled. It is recommended that software set NE to 1. This enables optimal performance in handling x87 floating-point exceptions.

Write Protect (WP) Bit. Bit 16. Read-only pages are protected from supervisor-level writes when the WP bit is set to 1. When WP is cleared to 0, supervisor software can write into read-only pages.

See Section 5.6 "Page-Protection Checks," on page 163 for information on the page-protection mechanism. If the shadow stack feature has been enabled (CR4.CET=1), attempting to clear WP to 0 causes a general-protection exception (#GP).

Alignment Mask (AM) Bit. Bit 18. Software enables automatic alignment checking by setting the AM bit to 1 when RFLAGS.AC=1. Alignment checking can be disabled by clearing either AM or

[AMD Public Use]

RFLAGS.AC to 0. When automatic alignment checking is enabled and CPL=3, a memory reference to an unaligned operand causes an alignment-check exception (#AC).

Not Writethrough (NW) Bit. Bit 29. Ignored. This bit can be set to 1 or cleared to 0, but its value is ignored. The NW bit exists only for legacy purposes.

Cache Disable (CD) Bit. Bit 30. When CD is cleared to 0, the internal caches are enabled. When CD is set to 1, no new data or instructions are brought into the internal caches. However, the processor still accesses the internal caches when CD = 1 under the following situations:

- Reads that hit in an internal cache cause the data to be read from the internal cache that reported the hit.
- Writes that hit in an internal cache cause the cache line that reported the hit to be written back to memory and invalidated in the cache.

Cache misses do not affect the internal caches when CD = 1. Software can prevent cache access by setting CD to 1 and invalidating the caches.

Setting CD to 1 also causes the processor to ignore the page-level cache-control bits (PWT and PCD) when paging is enabled. These bits are located in the page-translation tables and CR3 register. See Section “Page-Level Writethrough (PWT) Bit,” on page 157 and Section “Page-Level Cache Disable (PCD) Bit,” on page 157 for information on page-level cache control.

See Section 7.6 “Memory Caches,” on page 206 for information on the internal caches.

Paging Enable (PG) Bit. Bit 31. Software enables page translation by setting PG to 1, and disables page translation by clearing PG to 0. Page translation cannot be enabled unless the processor is in protected mode (CR0.PE=1). If software attempts to set PG to 1 when PE is cleared to 0, the processor causes a general-protection exception (#GP).

See Section 5.1 “Page Translation Overview,” on page 129 for information on the page-translation mechanism.

Reserved Bits. Bits 28:19, 17, 15:6, and 63:32. When writing the CR0 register, software should set the values of reserved bits to the values found during the previous CR0 read. No attempt should be made to change reserved bits, and software should never rely on the values of reserved bits. In long mode, bits 63:32 are reserved and must be written with zero, otherwise a #GP occurs.

3.1.2 CR2 and CR3 Registers

The CR2 (page-fault linear address) register, shown in Figure 3-2 on page 46 and Figure 3-3 on page 46, and the CR3 (page-translation-table base address) register, shown in Figure 3-4 and Figure 3-5 on page 46, and Figure 3-6 on page 47, are used only by the page-translation mechanism.

[AMD Public Use]

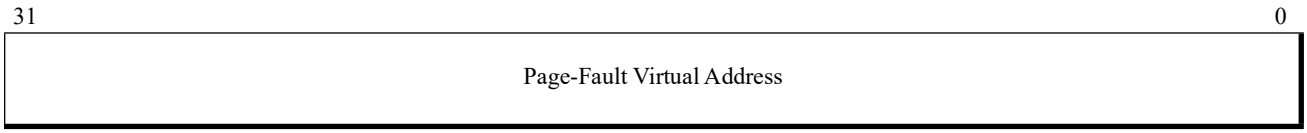


Figure 3-2. Control Register 2 (CR2)—Legacy-Mode

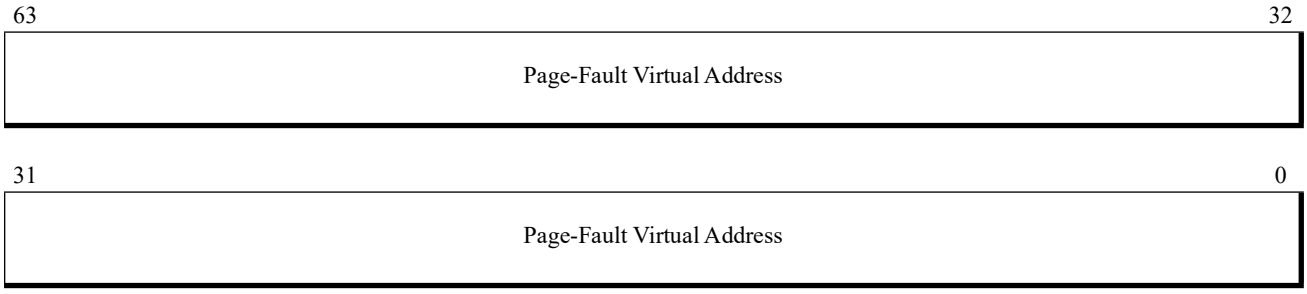


Figure 3-3. Control Register 2 (CR2)—Long Mode

See Section “CR2 Register,” on page 258 for a description of the CR2 register.

The CR3 register is used to point to the base address of the highest-level page-translation table.

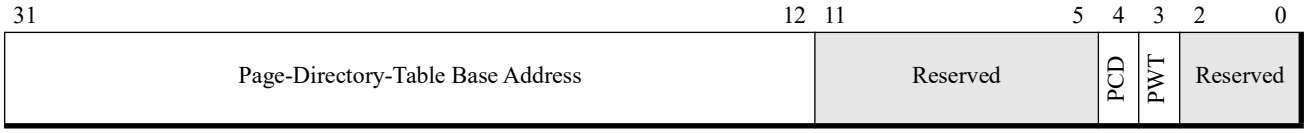


Figure 3-4. Control Register 3 (CR3)—Legacy-Mode Non-PAE Paging

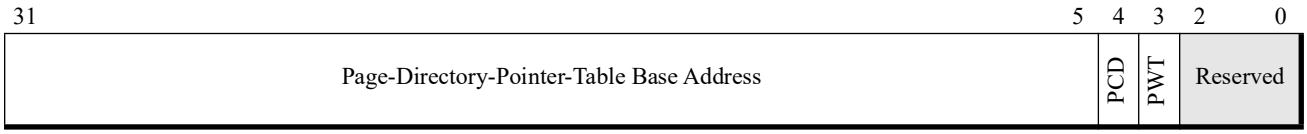


Figure 3-5. Control Register 3 (CR3)—Legacy-Mode PAE Paging

[AMD Public Use]

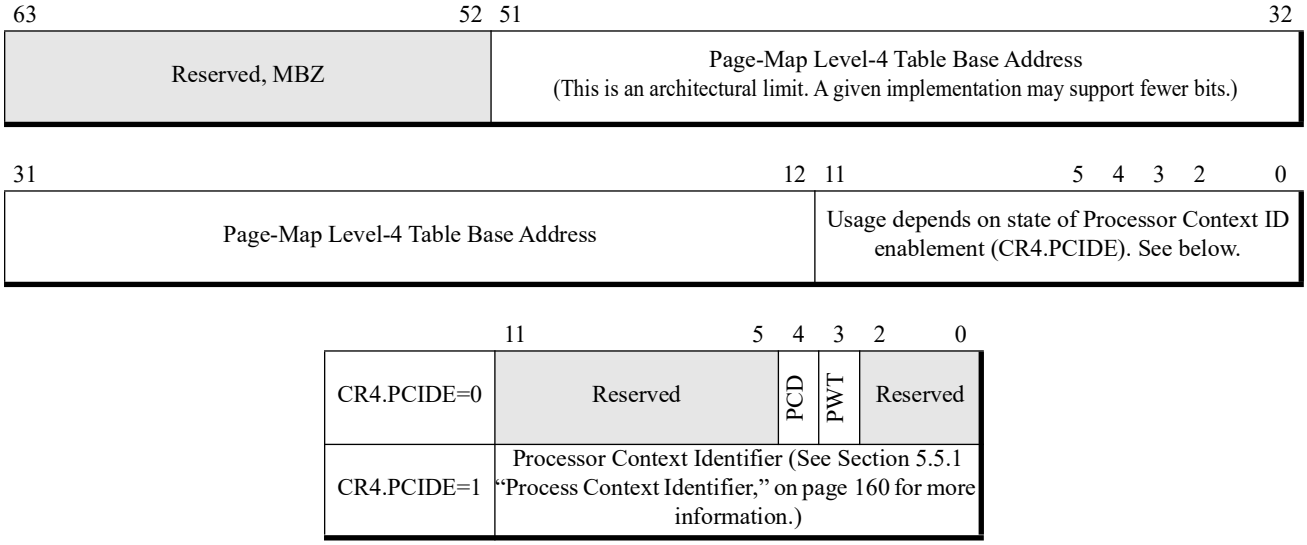


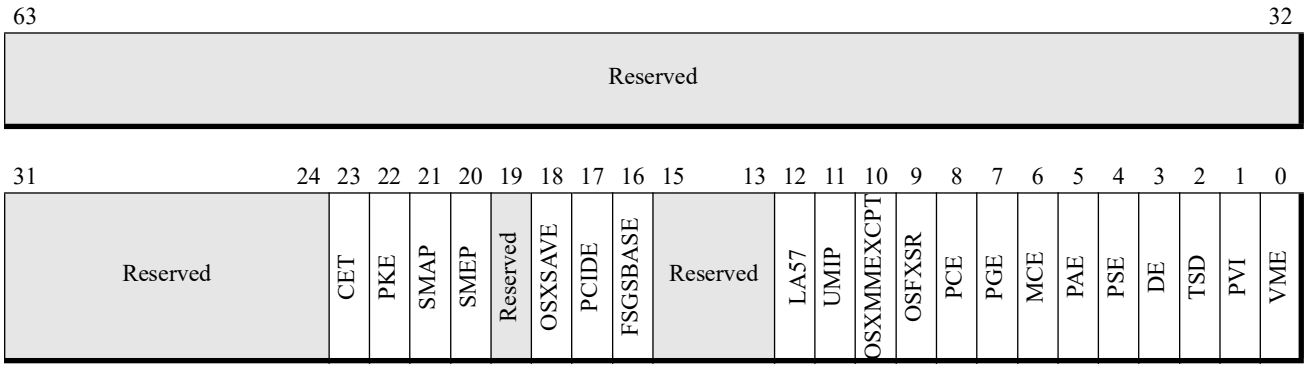
Figure 3-6. Control Register 3 (CR3)—Long Mode

The legacy CR3 register is described in Section 5.2.1 “CR3 Register,” on page 133, and the long-mode CR3 register is described in Section 5.3.2 “CR3,” on page 142.

3.1.3 CR4 Register

The CR4 register is shown in Figure 3-7. In legacy mode, the CR4 register is identical to the low 32 bits of the register (CR4 bits 31:0). The features controlled by the bits in the CR4 register are model-specific extensions. Except for the performance-counter extensions (PCE) feature, software can use the CPUID instruction to verify that each feature is supported before using that feature. See Section 3.3 “Processor Feature Identification,” on page 73 for information on using the CPUID instruction.

[AMD Public Use]



Bits	Mnemonic	Description	Access Type
63:24	Reserved		MBZ
23	CET	Control-flow Enforcement Technology	R/W
22	PKE	Protection Key Enable	R/W
21	SMAP	Supervisor Mode Access Protection	R/W
20	SMEP	Supervisor Mode Execution Prevention	R/W
19	Reserved		MBZ
18	OSXSAVE	XSAVE and Processor Extended States Enable Bit	R/W
17	PCIDE	Process Context Identifier Enable	R/W
16	FSGSBASE	Enable RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions	R/W
15:13	Reserved		MBZ
12	LA57	5-Level Paging Enable	R/W
11	UMIP	User Mode Instruction Prevention	R/W
10	OSXMMEXCPT	Operating System Unmasked Exception Support	R/W
9	OSFXSR	Operating System FXSAVE/FXRSTOR Support	R/W
8	PCE	Performance-Monitoring Counter Enable	R/W
7	PGE	Page-Global Enable	R/W
6	MCE	Machine Check Enable	R/W
5	PAE	Physical-Address Extension	R/W
4	PSE	Page Size Extensions	R/W
3	DE	Debugging Extensions	R/W
2	TSD	Time Stamp Disable	R/W
1	PVI	Protected-Mode Virtual Interrupts	R/W
0	VME	Virtual-8086 Mode Extensions	R/W

Figure 3-7. CR4 Register

The function of the CR4 control bits are (all bits are read/write):

Virtual-8086 Mode Extensions (VME). Bit 0. Setting VME to 1 enables hardware-supported performance enhancements for software running in virtual-8086 mode. Clearing VME to 0 disables this support. The enhancements enabled when VME=1 include:

[AMD Public Use]

- Virtualized, maskable, external-interrupt control and notification using the VIF and VIP bits in the RFLAGS register. Virtualizing affects the operation of several instructions that manipulate the RFLAGS.IF bit.
- Selective intercept of software interrupts (INT*n* instructions) using the interrupt-redirection bitmap in the TSS.

Protected-Mode Virtual Interrupts (PVI). Bit 1. Setting PVI to 1 enables support for protected-mode virtual interrupts. Clearing PVI to 0 disables this support. When PVI=1, hardware support of two bits in the RFLAGS register, VIF and VIP, is enabled.

Only the STI and CLI instructions are affected by enabling PVI. Unlike the case when CR0.VME=1, the interrupt-redirection bitmap in the TSS cannot be used for selective INT*n* interception.

PVI enhancements are also supported in long mode. See Section 8.10 “Virtual Interrupts,” on page 290 for more information on using PVI.

Time-Stamp Disable (TSD). Bit 2. The TSD bit allows software to control the privilege level at which the time-stamp counter can be read. When TSD is cleared to 0, software running at any privilege level can read the time-stamp counter using the RDTSC or RDTSCP instructions. When TSD is set to 1, only software running at privilege-level 0 can execute the RDTSC or RDTSCP instructions.

Debugging Extensions (DE). Bit 3. Setting the DE bit to 1 enables the I/O breakpoint capability and enforces treatment of the DR4 and DR5 registers as reserved. Software that accesses DR4 or DR5 when DE=1 causes a invalid opcode exception (#UD).

When the DE bit is cleared to 0, I/O breakpoint capabilities are disabled. Software references to the DR4 and DR5 registers are aliased to the DR6 and DR7 registers, respectively.

Page-Size Extensions (PSE). Bit 4. Setting PSE to 1 enables the use of 4-Mbyte physical pages. With PSE=1, the physical-page size is selected between 4 Kbytes and 4 Mbytes using the page-directory entry page-size field (PS). Clearing PSE to 0 disables the use of 4-Mbyte physical pages and restricts all physical pages to 4 Kbytes.

The PSE bit has no effect when physical-address extensions are enabled (CR4.PAE=1). Because long mode requires CR4.PAE=1, the PSE bit is ignored when the processor is running in long mode.

See Section “4-Mbyte Page Translation,” on page 136 for more information on 4-Mbyte page translation.

Physical-Address Extension (PAE). Bit 5. Setting PAE to 1 enables the use of physical-address extensions and 2-Mbyte physical pages. Clearing PAE to 0 disables these features.

With PAE=1, the page-translation data structures are expanded from 32 bits to 64 bits, allowing the translation of up to 52-bit physical addresses. Also, the physical-page size is selectable between 4 Kbytes and 2 Mbytes using the page-directory-entry page-size field (PS). Long mode requires PAE to be enabled in order to use the 64-bit page-translation data structures to translate 64-bit virtual addresses to 52-bit physical addresses.

[AMD Public Use]

See Section 5.2.3 “PAE Paging,” on page 137 for more information on physical-address extensions.

Machine-Check Enable (MCE). Bit 6. Setting MCE to 1 enables the machine-check exception mechanism. Clearing this bit to 0 disables the mechanism. When enabled, a machine-check exception (#MC) occurs when an uncorrectable machine-check error is encountered.

Regardless of whether machine-check exceptions are enabled, the processor records enabled-errors when they occur. Error-reporting is performed by the machine-check error-reporting register banks. Each bank includes a control register for enabling error reporting and a status register for capturing errors. Correctable machine-check errors are also reported, but they do not cause a machine-check exception.

See Chapter 9, “Machine Check Architecture,” for a description of the machine-check mechanism, the registers used, and the types of errors captured by the mechanism.

Page-Global Enable (PGE). Bit 7. When page translation is enabled, system-software performance can often be improved by making some page translations *global* to all tasks and procedures. Setting PGE to 1 enables the global-page mechanism. Clearing this bit to 0 disables the mechanism.

When PGE is enabled, system software can set the global-page (G) bit in the lowest level of the page-translation hierarchy to 1, indicating that the page translation is global. Page translations marked as global are not invalidated in the TLB when the page-translation-table base address (CR3) is updated. When the G bit is cleared, the page translation is not global. All supported physical-page sizes also support the global-page mechanism. See Section 5.5.2 “Global Pages,” on page 160 for information on using the global-page mechanism.

Performance-Monitoring Counter Enable (PCE). Bit 8. Setting PCE to 1 allows software running at any privilege level to use the RDPMC instruction. Software uses the RDPMC instruction to read the performance-monitoring counter MSRs, *PerfCtr*n*. Clearing PCE to 0 allows only the most-privileged software (CPL=0) to use the RDPMC instruction.

FXSAVE/FXRSTOR Support (OSFXSR). Bit 9. System software must set the OSFXSR bit to 1 to enable use of the legacy SSE instructions. When this bit is set to 1, it also indicates that system software uses the FXSAVE and FXRSTOR instructions to save and restore the processor state for the x87, 64-bit media, and 128-bit media instructions.

Clearing the OSFXSR bit to 0 indicates that legacy SSE instructions cannot be used. Attempts to use those instructions while this bit is clear result in an invalid-opcode exception (#UD). Software can continue to use the FXSAVE/FXRSTOR instructions for saving and restoring the processor state for the x87 and 64-bit media instructions.

Unmasked Exception Support (OSXMMEXCPT). Bit 10. System software must set the OSXMMEXCPT bit to 1 when it supports the SIMD floating-point exception (#XF) for handling of unmasked 256-bit and 128-bit media floating-point errors. Clearing the OSXMMEXCPT bit to 0 indicates the #XF handler is not supported. When OSXMMEXCPT=0, unmasked 128-bit media floating-point exceptions cause an invalid-opcode exception (#UD). See “SIMD Floating-Point Exception Causes” in Volume 1 for more information on unmasked SSE floating-point exceptions.

[AMD Public Use]

User Mode Instruction Prevention (UMIP). Bit 11. Setting UMIP to 1 enables a security mode to restrict certain instructions executing at CPL>0 so that they do not reveal information about structures that are controlled by the processor when it is at CPL=0. When UMIP is enabled, execution of SGDT, SIDT, SLDT, SMSW and STR instructions become available only at CPL=0 and any attempt to execute them with CPL>0 results in a #GP fault with error code 0. See Section 6 “System Instructions,” on page 172 for more information about UMIP.

5-Level Paging Enable (LA57) . Bit 12. Setting this bit to 1 while EFER[LMA]=1 enables 5-Level paging. 5-Level paging allows for the translation of up to 57 virtual address bits. See Section Section 5.3 “Long-Mode Page Translation,” on page 141 for a description of 4-Level and 5-Level paging.

FSGSBASE. Bit 16. System software must set this bit to 1 to enable the execution of the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions when supported. When enabled, these instructions allow software running in 64-bit mode at any privilege level to read and write the FS.base and GS.base hidden segment register state. See the discussion of segment registers in 64-bit mode in Section 4.5.3 “Segment Registers in 64-Bit Mode,” on page 82. Also see descriptions of the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions in Volume 3.

Processor Context Identifier Enable (PCIDE). Bit 17. Enable support for Process Context Identifiers (PCIDs). System software must set this bit to 1 to enable execution of the INVPCID instruction when supported. Can only be set in long mode (EFER.LMA = 1). See Chapter 5.5.1, “Process Context Identifier,” for more information on Process Context Identifiers.

XSAVE and Extended States (OSXSAVE). Bit 18. After verifying hardware support for the extended processor state management instructions, operating system software sets this bit to indicate support for the XGETBV, XSAVE and XRSTOR instructions.

Setting this bit also:

- allows the execution of the XGETBV and XSETBV instructions, and
- enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), along with other processor extended states enabled in XCR0.

After initializing the XSAVE/XRSTOR save area, XSAVEOPT (if supported) may be used to save x87 FPU and other enabled extended processor state. For more information on XSAVEOPT, see individual instruction listing in Chapter 2 of Volume 4.

Note that legacy SSE instruction execution must be enabled prior to enabling extended processor state management.

Supervisor Mode Execution Prevention (SMEP). Bit 20. Setting this bit enables the supervisor mode execution prevention feature, if supported. This feature prevents the execution of instructions that reside in pages accessible by user-mode software when the processor is in supervisor-mode. See Section 5.6 “Page-Protection Checks,” on page 163 for more information.

[AMD Public Use]

Supervisor Mode Access Prevention (SMAP). Bit 21. Setting this bit enables the supervisor mode access prevention feature, if supported. This feature prevents certain data accesses to pages accessible by user-mode software when the processor is in supervisor mode. See Section 5.6.6 “Supervisor-Mode Access Prevention(CR4.SMAP) Bit,” on page 166 for more information.

Protection Key Enable (PKE). Bit 22. Enable support for memory Protection Keys. Also enables support for the RDPKRU and WRPKRU instructions. A MOV to CR4 that changes CR4.PKE from 0 to 1 causes all cached entries in the TLB for the logical processor to be invalidated. (See Section 5.6.7 “Memory Protection Keys (MPK) Bit,” on page 166 for more information on memory protection keys.)

Control-flow Enforcement Technology (CET). Bit 23. Setting this bit enables the shadow stack feature. This feature ensures that return addresses read from the stack by RET and IRET instructions originated from a CALL instruction or similar control transfer.

See Section 18 “Shadow Stacks,” on page 710 for more information. Before setting this bit, CR0.WP must be set to 1, otherwise a #GP fault is generated.

CR1 and CR5–CR7 Registers. Control registers CR1, CR5–CR7, and CR9–CR15 are reserved. Attempts by software to use these registers result in an undefined-opcode exception (#UD).

3.1.4 Additional Control Registers in 64-Bit-Mode

In 64-bit mode, additional encodings are available to address up to eight additional control registers. The REX.R bit, in a REX prefix, is used to modify the ModRM *reg* field when that field encodes a control register, as shown in “REX Prefixes” in Volume 3. These additional encodings enable the processor to address CR8–CR15.

One additional control register, CR8, is defined in 64-bit mode for all hardware implementations, as described in “CR8 (Task Priority Register, TPR),” below. Access to the CR9–CR15 registers is implementation-dependent. Any attempt to access an unimplemented register results in an invalid-opcode exception (#UD).

3.1.5 CR8 (Task Priority Register, TPR)

The AMD64 architecture introduces a new control register, CR8, defined as the task priority register (TPR). The register is accessible in 64-bit mode using the REX prefix. See Section 8.5.2 “External Interrupt Priorities,” on page 269 for a description of the TPR and how system software can use the TPR for controlling external interrupts.

3.1.6 RFLAGS Register

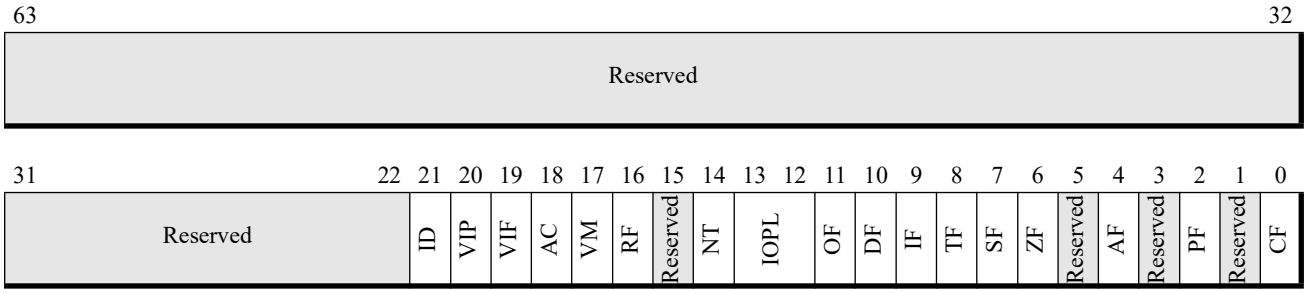
The RFLAGS register contains two different types of information:

- *Control bits* provide system-software controls and directional information for string operations. Some of these bits can have privilege-level restrictions.

[AMD Public Use]

- *Status bits* provide information resulting from logical and arithmetic operations. These are written by the processor and can be read by software running at any privilege level.

Figure 3-8 on page 53 shows the format of the RFLAGS register. The legacy EFLAGS register is identical to the low 32 bits of the register shown in Figure 3-8 (RFLAGS bits 31:0). The term *rFLAGS* is used to refer to the 16-bit, 32-bit, or 64-bit flags register, depending on context.



Bits	Mnemonic	Description	Access type
63:22	Reserved		RAZ
21	ID	ID Flag	R/W
20	VIP	Virtual Interrupt Pending	R/W
19	VIF	Virtual Interrupt Flag	R/W
18	AC	Alignment Check	R/W
17	VM	Virtual-8086 Mode	R/W
16	RF	Resume Flag	R/W
15	Reserved		RAZ
14	NT	Nested Task	R/W
13:12	IOPL	I/O Privilege Level	R/W
11	OF	Overflow Flag	R/W
10	DF	Direction Flag	R/W
9	IF	Interrupt Flag	R/W
8	TF	Trap Flag	R/W
7	SF	Sign Flag	R/W
6	ZF	Zero Flag	R/W
5	Reserved		RAZ
4	AF	Auxiliary Flag	R/W
3	Reserved		RAZ
2	PF	Parity Flag	R/W
1	Reserved		RA1
0	CF	Carry Flag	R/W

Figure 3-8. RFLAGS Register

The functions of the RFLAGS control and status bits used by application software are described in “Flags Register” in Volume 1. The functions of RFLAGS system bits are (unless otherwise noted, all bits are read/write):

[AMD Public Use]

Trap Flag (TF) Bit. Bit 8. Software sets the TF bit to 1 to enable single-step mode during software debug. Clearing this bit to 0 disables single-step mode.

When single-step mode is enabled at the start of an instruction's execution, a debug exception (#DB) occurs immediately after the instruction completes execution. Single stepping is automatically disabled (TF is set to 0) when the #DB exception occurs or when any exception or interrupt occurs.

See Section 13.1.4 “Single Stepping,” on page 410 for information on using the single-step mode during debugging.

Interrupt Flag (IF) Bit. Bit 9. Software sets the IF bit to 1 to enable maskable interrupts. Clearing this bit to 0 causes the processor to ignore maskable interrupts. The state of the IF bit does not affect the response of a processor to non-maskable interrupts, software-interrupt instructions, or exceptions.

The ability to modify the IF bit depends on several factors:

- The current privilege-level (CPL)
- The I/O privilege level (RFLAGS.IOPL)
- Whether or not virtual-8086 mode extensions are enabled (CR4.VME=1)
- Whether or not protected-mode virtual interrupts are enabled (CR4.PVI=1)

See Section 8.1.4 “Masking External Interrupts,” on page 246 for information on interrupt masking. See Section 6.2.3 “Accessing the RFLAGS Register,” on page 180 for information on the specific instructions used to modify the IF bit.

I/O Privilege Level Field (IOPL) Field. Bits 13:12. The IOPL field specifies the privilege level required to execute I/O address-space instructions (i.e., instructions that address the I/O space rather than memory-mapped I/O, such as IN, OUT, INS, OUTS, etc.). For software to execute these instructions, the current privilege-level (CPL) must be equal to or higher than (lower numerical value than) the privilege specified by IOPL (CPL <= IOPL). If the CPL is lower than (higher numerical value than) that specified by the IOPL (CPL > IOPL), the processor causes a general-protection exception (#GP) when software attempts to execute an I/O instruction. See “Protected-Mode I/O” in Volume 1 for information on how IOPL controls access to address-space I/O.

Virtual-8086 mode uses IOPL to control virtual interrupts and the IF bit when virtual-8086 mode extensions are enabled (CR4.VME=1). The protected-mode virtual-interrupt mechanism (PVI) also uses IOPL to control virtual interrupts and the IF bit when PVI is enabled (CR4.PVI=1). See Section 8.10 “Virtual Interrupts,” on page 290 for information on how IOPL is used by the virtual interrupt mechanism.

Nested Task (NT) Bit. Bit 14, IRET reads the NT bit to determine whether the current task is nested within another task. When NT is set to 1, the current task is nested within another task. When NT is cleared to 0, the current task is at the top level (not nested).

The processor sets the NT bit during a task switch resulting from a CALL, interrupt, or exception through a task gate. When an IRET is executed from legacy mode while the NT bit is set, a task switch occurs. See Section 12.3.3 “Task Switches Using Task Gates,” on page 387 for information on

[AMD Public Use]

switching tasks using task gates, and Section 12.3.4 “Nesting Tasks,” on page 389 for information on task nesting.

Resume Flag (RF) Bit. Bit 16. The RF bit, when set to 1, temporarily disables instruction breakpoint reporting to prevent repeated debug exceptions (#DB) from occurring. This allows an instruction which had been inhibited by an instruction-breakpoint debug exception to be restarted by the debug exception handler.

The processor clears the RF bit after every instruction is successfully executed, except when the

instruction is:

- An IRET that sets the RF bit.
- JMP, CALL, or INT*n* through a task gate.

In both of the above cases, RF is not cleared to 0 until the *next* instruction successfully executes.

When an exception occurs (or when a string instruction is interrupted), the processor normally sets RF=1 in the RFLAGS image saved on the interrupt stack. However, when a #DB exception occurs as a result of an instruction breakpoint, the processor clears the RF bit to 0 in the interrupt-stack RFLAGS image.

For instruction restart to work properly following an instruction breakpoint, the #DB exception handler must set RF to 1 in the interrupt-stack RFLAGS image. When an IRET is later executed to return to the instruction that caused the instruction-breakpoint #DB exception, the set RF bit (RF=1) is loaded from the interrupt-stack RFLAGS image. RF is not cleared by the processor until the instruction causing the #DB exception successfully executes.

Virtual-8086 Mode (VM) Bit. Bit 17. Software sets the VM bit to 1 to enable virtual-8086 mode. Software clears the VM bit to 0 to disable virtual-8086 mode. System software can only change this bit using a task switch or an IRET. It cannot modify the bit using the POPFD instruction.

Alignment Check (AC) Bit. Bit 18. Software enables automatic alignment checking by setting the AC bit to 1 when CR0.AM=1. Alignment checking can be disabled by clearing either AC or CR0.AM to 0. When automatic alignment checking is enabled and the current privilege-level (CPL) is 3 (least privileged), a memory reference to an unaligned operand causes an alignment-check exception (#AC).

When the supervisor mode access prevention feature is enabled (CR4.SMAP=1), certain supervisor-mode data accesses to pages accessible by user mode are allowed only if RFLAGS.AC=1. See Section 5.6.6 “Supervisor-Mode Access Prevention(CR4.SMAP) Bit,” on page 166 for more information.

Virtual Interrupt (VIF) Bit. Bit 19. The VIF bit is a virtual image of the RFLAGS.IF bit. It is enabled when either virtual-8086 mode extensions are enabled (CR4.VME=1) or protected-mode virtual interrupts are enabled (CR4.PVI=1), and the RFLAGS.IOPL field is less than 3. When enabled, instructions that ordinarily would modify the IF bit actually modify the VIF bit with no effect on the RFLAGS.IF bit.

[AMD Public Use]

System software that supports virtual-8086 mode should enable the VIF bit using CR4.VME. This allows 8086 software to execute instructions that can set and clear the RFLAGS.IF bit without causing an exception. With VIF enabled in virtual-8086 mode, those instructions set and clear the VIF bit instead, giving the appearance to the 8086 software that it is modifying the RFLAGS.IF bit. System software reads the VIF bit to determine whether or not to take the action desired by the 8086 software (enabling or disabling interrupts by setting or clearing the RFLAGS.IF bit).

In long mode, the use of the VIF bit is supported when CR4.PVI=1. See Section 8.10 “Virtual Interrupts,” on page 290 for more information on virtual interrupts.

Virtual Interrupt Pending (VIP) Bit. Bit 20. The VIP bit is provided as an extension to both virtual-8086 mode and protected mode. It is used by system software to indicate that an external, maskable interrupt is pending (awaiting) execution by either a virtual-8086 mode or protected-mode interrupt-service routine. Software must enable virtual-8086 mode extensions (CR4.VME=1) or protected-mode virtual interrupts (CR4.PVI=1) before using VIP.

VIP is normally set to 1 by a protected-mode interrupt-service routine that was entered from virtual-8086 mode as a result of an external, maskable interrupt. Before returning to the virtual-8086 mode application, the service routine sets VIP to 1 if EFLAGS.VIF=1. When the virtual-8086 mode application attempts to enable interrupts by clearing EFLAGS.VIF to 0 while VIP=1, a general-protection exception (#GP) occurs. The #GP service routine can then decide whether to allow the virtual-8086 mode service routine to handle the pending external, maskable interrupt. (EFLAGS is specifically referred to in this case because virtual-8086 mode is supported only from legacy mode.)

In long mode, the use of the VIP bit is supported when CR4.PVI=1. See Section 8.10 “Virtual Interrupts,” on page 290 for more information on virtual-8086 mode interrupts and the VIP bit.

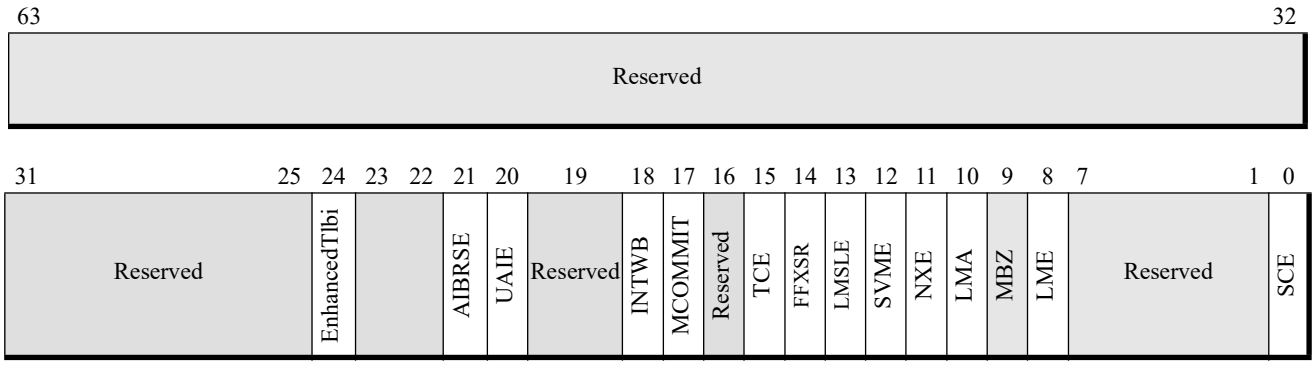
Processor Feature Identification (ID) Bit. Bit 21. The ability of software to modify this bit indicates that the processor implementation supports the CPUID instruction. See Section 3.3 “Processor Feature Identification,” on page 73 for more information on the CPUID instruction.

3.1.7 Extended Feature Enable Register (EFER)

The extended-feature-enable register (EFER) contains control bits that enable additional processor features not controlled by the legacy control registers. The EFER is a model-specific register (MSR) with an address of C000_0080h (see Section 3.2 “Model-Specific Registers (MSRs),” on page 60 for more information on MSRs). It can be read and written only by privileged software. Figure 3-9 on page 57 shows the format of the EFER register.

Note: *Attempting to set an enable bit for a feature that is not supported by the processor will result in a #GP(0) exception.*

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:22	Reserved		MBZ
24	EnhancedTlbi	Enhanced TLB Invalidate Enable	R/W
23:22	Reserved		MBZ
21	AIBRSE	Automatic IBRS Enable	R/W
20	UAIE	Upper Address Ignore Enable	R/W
19	Reserved		MBZ
18	INTWB	Interruptible WBINVD/WBNOINVD enable	R/W
17	MCOMMIT	Enable MCOMMIT instruction	R/W
16	Reserved		MBZ
15	TCE	Translation Cache Extension	R/W
14	FFXSR	Fast FXSAVE/FXRSTOR	R/W
13	LMSLE	Long Mode Segment Limit Enable	R/W
12	SVME	Secure Virtual Machine Enable	R/W
11	NXE	No-Execute Enable	R/W
10	LMA	Long Mode Active	R/W
9	Reserved		MBZ
8	LME	Long Mode Enable	R/W
7:1	Reserved		RAZ
0	SCE	System Call Extensions	R/W

Figure 3-9. Extended Feature Enable Register (EFER)

The defined EFER bits shown in Figure 3-9 above are described below:

System-Call Extension (SCE) Bit. Bit 0. Setting this bit to 1 enables the SYSCALL and SYSRET instructions. Application software can use these instructions for low-latency system calls and returns in a non-segmented (flat) address space. See Section 6.1 “Fast System Call and Return,” on page 175 for additional information.

Long Mode Enable (LME) Bit. Bit 8. Setting this bit to 1 enables the processor to activate long mode. Long mode is not activated until software enables paging some time later. When paging is enabled after LME is set to 1, the processor sets the EFER.LMA bit to 1, indicating that long mode is not only enabled but also active. See Chapter 14, “Processor Initialization and Long Mode Activation,” for more information on activating long mode.

[AMD Public Use]

Long Mode Active (LMA) Bit. Bit 10. This bit indicates that long mode is active. The processor sets LMA to 1 when both long mode and paging have been enabled by system software. See Chapter 14, "Processor Initialization and Long Mode Activation," for more information on activating long mode.

When LMA=1, the processor is running either in compatibility mode or 64-bit mode, depending on the value of the L bit in a code-segment descriptor, as shown in Figure 1-6 on page 12.

When LMA=0, the processor is running in legacy mode. In this mode, the processor behaves like a standard 32-bit x86 processor, with none of the new 64-bit features enabled. When writing the EFER register the value of this bit must be preserved. Software must read the EFER register to determine the value of LMA, change any other bits as required and then write the EFER register. An attempt to write a value that differs from the state determined by hardware results in a #GP fault.

No-Execute Enable (NXE) Bit. Bit 11. Setting this bit to 1 enables the no-execute page-protection feature. The feature is disabled when this bit is cleared to 0. See Section "No Execute (NX) Bit," on page 158 for more information.

Before setting NXE, system software should verify the processor supports the feature by examining the feature flag CPUID Fn8000_0001_EDX[NX]. See Section 3.3 "Processor Feature Identification," on page 73 for information on using the CPUID instruction.

Secure Virtual Machine Enable (SVME) Bit. Bit 12. Enables the SVM extensions. When this bit is zero, the SVM instructions cause #UD exceptions. EFER.SVME defaults to a reset value of zero. The effect of turning off EFER.SVME while a guest is running is undefined; therefore, the VMM should always prevent guests from writing EFER. An attempt by an SEV-ES guest to write 0 to EFER.SVME is ignored. SVM extensions can be disabled by setting VM_CR.SVME_DISABLE. For more information, see descriptions of LOCK and SMVE_DISABLE bits in Section 15.30.1 "VM_CR MSR (C001_0114h)," on page 608.

Long Mode Segment Limit Enable (LMSLE) Bit. Bit 13. Setting this bit to 1 enables certain limit checks in 64-bit mode. This feature has been deprecated and is not supported by all processor implementations. If CPUID Fn8000_0008_EBX[EferLmlseUnsupported](bit 20)=1, 64-bit mode segment limit checking is not supported and attempting to set EFER.LMSLE =1 causes a #GP exception. See Section 4.12.2 "Data Limit Checks in 64-bit Mode," on page 125, for more information on these limit checks.

Fast FXSAVE/FXRSTOR (FFXSR) Bit. Bit 14. Setting this bit to 1 enables the FXSAVE and FXRSTOR instructions to execute faster in 64-bit mode at CPL 0. This is accomplished by not saving or restoring the XMM registers (XMM0-XMM15). The FFXSR bit has no effect when the FXSAVE/FXRSTOR instructions are executed in non 64-bit mode, or when CPL is > 0. The FFXSR bit does not affect the save/restore of the legacy x87 floating-point state, or the save/restore of MXCSR.

Before setting FFXSR, system software should verify whether this feature is supported by examining the feature flag CPUID Fn8000_0001_EDX[FFXSR]. See Section 3.3 "Processor Feature Identification," on page 73 for information on using the CPUID instruction.

[AMD Public Use]

Translation Cache Extension (TCE) Bit. Bit 15. Setting this bit to 1 changes how the INVLPG, INVLPGB, and INVPCID instructions operate on TLB entries. When this bit is 0, these instructions remove the target PTE from the TLB as well as *all* upper-level table entries that are cached in the TLB, whether or not they are associated with the target PTE. When this bit is set, these instructions will remove the target PTE and may only remove those upper-level entries that lead to the target PTE in the page table hierarchy, leaving unrelated upper-level entries intact. This may provide a performance benefit.

Page table management software must be written in a way that takes this behavior into account. Software that was written for a processor that does not cache upper-level table entries may result in stale entries being incorrectly used for translations when TCE is enabled. Software that is compatible with TCE mode will operate in either mode.

For software using INVLPGB to broadcast TLB invalidations, the invalidations are controlled by the EFER.TCE value on the processor executing the INVLPGB instruction.

Before setting TCE, system software should verify that this feature is supported by examining the feature flag CPUID Fn8000_0001_ECX[TCE]. See Section 3.3 “Processor Feature Identification,” on page 73 for information on using the CPUID instruction.

Mcommit Enable (MCOMMIT) Bit. Bit 17. Setting this bit to 1 enables the MCOMMIT instruction. When clear, attempting to execute MCOMMIT causes a #UD exception.

Interruptible Wbinvd (INTWB) Bit. Bit 18. Setting this bit to 1 allows the WBINVD and WBNOINVD instructions to be interruptible. See WBINVD and WBNOINVD in Volume 3.

Upper Address Ignore Enable (UAIE) Bit. Bit 20. Setting this bit to 1 excludes bits 63:57 of an address from the canonical check for some memory references. Section 5.10 “Upper Address Ignore,” on page 170.

Automatic IBRS Enable (AIBRSE) Bit. Bit 21. Setting this bit to 1 enables Automatic IBRS (Indirect Branch Restricted Speculation). When Automatic IBRS is enabled, the following processes have IBRS protection even when SPEC_CTRL[IBRS] is not set:

- Processes running at CPL=0
- Processes running as host when Secure Nested Paging (SEV-SNP) is enabled

When Automatic IBRS is enabled and Enhanced Return Address Predictor Security is not supported (CPUID Fn8000_0021_EAX[ERAPS] (bit 24) = 0), the return address predictor is cleared on VMEXIT. More information about IBRS protection can be found in Section 3.2.9 “Speculation Control Registers,” on page 67.

Enhanced TLB Invalidate Enable (EnhancedTlbi) Bit. Bit 24. Setting this bit to 1 enables Enhanced TLB Invalidate for the INVLPGB and TLBSYNC instructions. With this bit set, the INVLPGB and TLBSYNC instructions consider additional input arguments which allow INVLPGB to invalidate translations in the IOMMU and allow TLBSYNC to sync with the IOMMU. Support for Enhanced TLB Invalidate is indicated by CPUID Fn8000_000A_ECX [EnhancedTlbi] (bit 5) = 1.

[AMD Public Use]

3.1.8 Extended Control Registers (XCRn)

Extended Control Registers (XCR*n*) form a new register space that is available for managing processor architectural features and capabilities. Currently only XCR0 (XFEATURE_ENABLED_MASK) is defined. All other XCR registers are reserved. For more details on the Extended Control Registers, see “Extended Control Registers” in APM Volume 4, Chapter 1.

3.2 Model-Specific Registers (MSRs)

Processor implementations provide model-specific registers (MSRs) for software control over the unique features supported by that implementation. Software reads and writes MSRs using the privileged RDMSR and WRMSR instructions. Implementations of the AMD64 architecture can contain a mixture of two basic MSR types:

- *Legacy MSRs*. The AMD family of processors often share model-specific features with other x86 processor implementations. Where possible, AMD implementations use the same MSRs for the same functions. For example, the memory-typing and debug-extension MSRs are implemented on many AMD and non-AMD processors.
- *AMD model-specific MSRs*. There are many MSRs common to the AMD family of processors but not to legacy x86 processors. Where possible, AMD implementations use the same AMD-specific MSRs for the same functions.

Every model-specific register, as the name implies, is not necessarily implemented by all members of the AMD family of processors. Appendix A, “MSR Cross-Reference,” lists MSR-address ranges currently used by various AMD and other x86 processors.

The AMD64 architecture includes a number of features that are controlled using MSRs. Those MSRs are shown in Figure 3-10. The EFER register—described in Section 3.1.7 “Extended Feature Enable Register (EFER),” on page 56—is also an MSR.

[AMD Public Use]

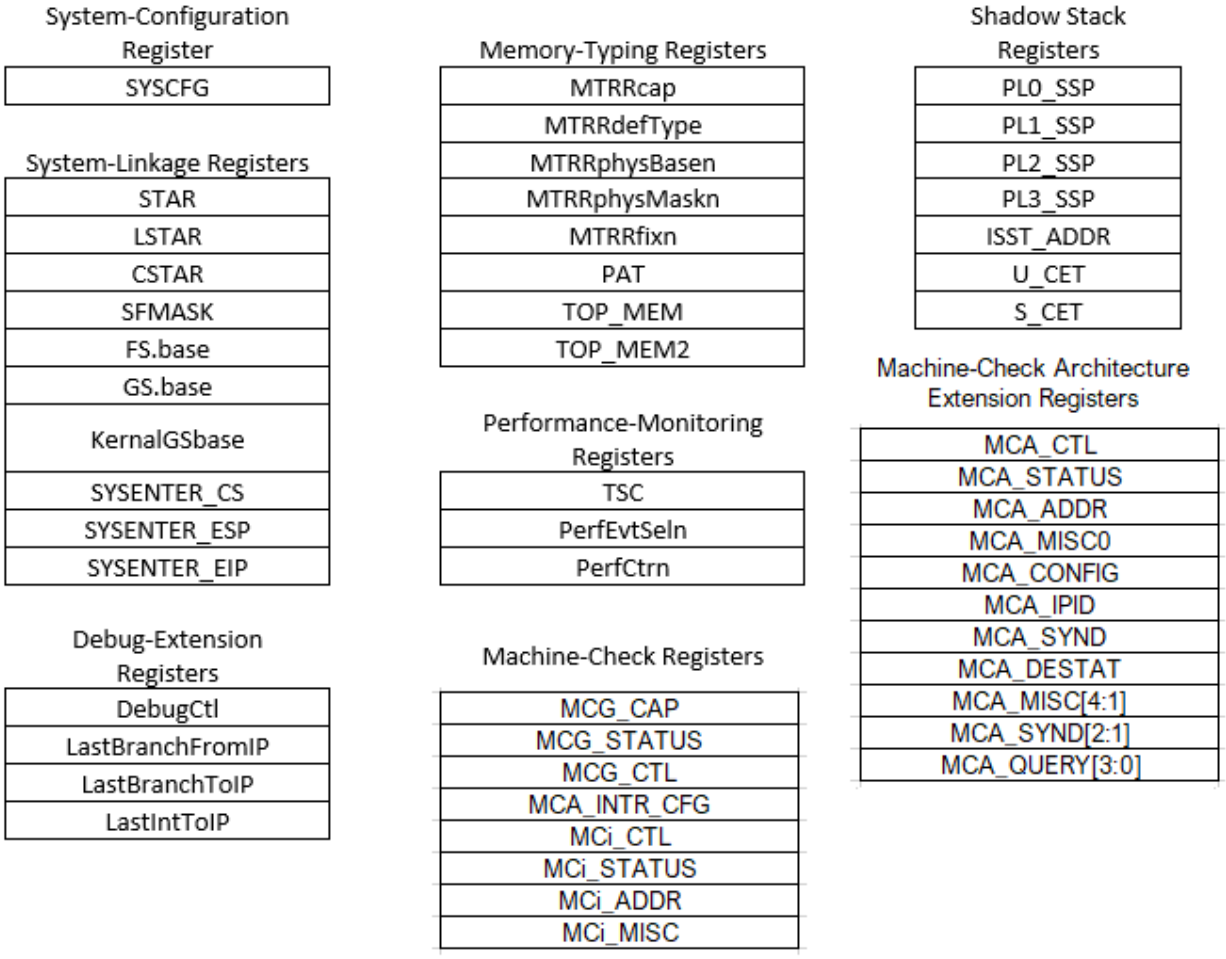


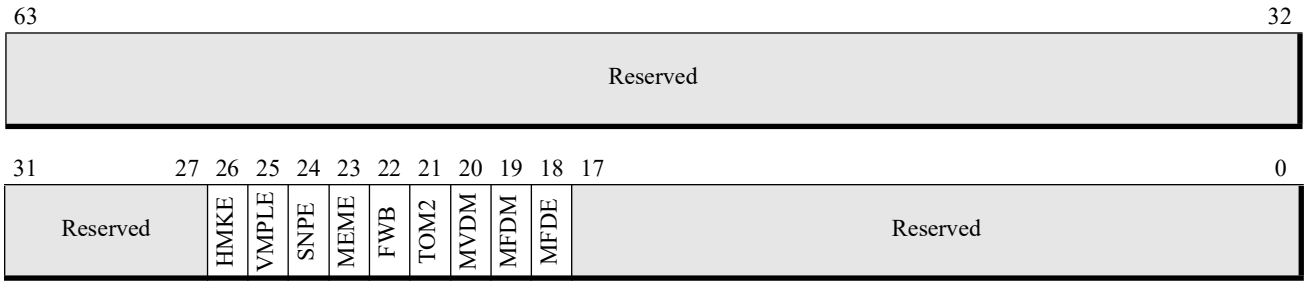
Figure 3-10. AMD64 Architecture Model-Specific Registers

The following sections briefly describe the MSRs in the AMD64 architecture.

3.2.1 System Configuration Register (SYSCFG)

The system-configuration register (SYSCFG) contains control bits for enabling and configuring system bus features. SYSCFG is a model-specific register (MSR) with an address of C001_0010h. Figure 3-11 on page 62 shows the format of the SYSCFG register. Some features are implementation specific, and are described in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product. Implementation-specific features are not shown in Figure 3-11.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:27	Reserved		MBZ
26	HMKE	HostMultiKeyMemEncrModeEn	R/W
25	VMPLE	VMPLEn	R/W
24	SNPE	SecureNestedPagingEn	R/W
23	MEME	MemEncryptionModeEn	R/W
22	FWB	Tom2ForceMemTypeWB	R/W
21	TOM2	MtrrTom2En	R/W
20	MVDM	MtrrVarDramEn	R/W
19	MFDM	MtrrFixDramModEn	R/W
18	MFDE	MtrrFixDramEn	R/W
17:0	Reserved		MBZ

Figure 3-11. System-Configuration Register (SYSCFG)

The function of the SYSCFG bits are (all bits are read/write unless otherwise noted):

MtrrFixDramEn Bit. Bit 18. Setting this bit to 1 enables use of the RdMem and WrMem attributes in the fixed-range MTRR registers. When cleared, these attributes are disabled. The RdMem and WrMem attributes allow system software to define fixed-range IORRs using the fixed-range MTRRs. See Section 7.9.1 “Extended Fixed-Range MTRR Type-Field Encodings,” on page 234 for information on using this feature.

MtrrFixDramModEn Bit. Bit 19. Setting this bit to 1 allows software to read and write the RdMem and WrMem bits. When cleared, writes do not modify the RdMem and WrMem bits, and reads return 0. See Section 7.9.1 “Extended Fixed-Range MTRR Type-Field Encodings,” on page 234 for information on using this feature.

MtrrVarDramEn Bit. Bit 20. Setting this bit to 1 enables the TOP_MEM register and the variable-range IORRs. These registers are disabled when the bit is cleared to 0. See Section 7.9.2 “IORRs,” on page 236 and Section 7.9.4 “Top of Memory,” on page 238 for information on using these features.

MtrrTom2En Bit. Bit 21. Setting this bit to 1 enables the TOP_MEM2 register. The register is disabled when this bit is cleared to 0. See Section 7.9.4 “Top of Memory,” on page 238 for information on using this feature.

[AMD Public Use]

Tom2ForceMemTypeWB. Bit 22. Setting this bit to 1 forces the default memory type for memory between 4GB and the address specified by TOP_MEM2 to be write back instead of the memory type defined by MTRRdefType[Type]. For this bit to have any effect, MTRRdefType[E] must be 1. MTRR variable-range settings and PAT can be used to override this memory type.

MemEncryptionModeEn. Bit 23. Setting this bit to 1 enables the SME (Section 7.10 “Secure Memory Encryption,” on page 240) and SEV (Section 15.34 “Secure Encrypted Virtualization,” on page 613) memory encryption features. When cleared, these features are disabled. Once this bit or HostMultiKeyMemEncrModeEn is set to 1, it cannot be changed.

SecureNestedPagingEn. Bit 24. Setting this bit to 1 enables SEV-SNP (Section 15.36 “Secure Nested Paging (SEV-SNP),” on page 627). When cleared, this feature is disabled. Once this bit is set to 1, it cannot be changed. This bit can only be set if MemEncryptionModeEn is already set or is simultaneously also set to 1. After SecureNestedPagingEn is set to 1, certain MSRs may no longer be written. See Section 15.36.2 “Enabling SEV-SNP,” on page 628 for details.

VMPLEn. Bit 25. Setting this bit to 1 enables the VMPL feature (Section 15.36.7 “Virtual Machine Privilege Levels,” on page 632). Software should set this bit to 1 when SecureNestedPagingEn is being set to 1. Once SecureNestedPagingEn is set to 1, VMPLEn cannot be changed.

HostMultiKeyMemEncrModeEn. Bit 26. Setting this bit to 1 enables the SME-MK memory encryption feature (See Section 7.10 “Secure Memory Encryption,” on page 240). When cleared, this feature is disabled. Once this bit or MemEncryptionModeEn is set to 1, it cannot be changed.

3.2.2 System-Linkage Registers

System-linkage MSRs are used by system software to allow fast control transfers between applications and the operating system. The functions of these registers are:

STAR, LSTAR, CSTAR, and SFMASK Registers. These registers are used to provide mode-dependent linkage information for the SYSCALL and SYSRET instructions. STAR is used in legacy modes, LSTAR in 64-bit mode, and CSTAR in compatibility mode. SFMASK is used by the SYSCALL instruction for RFLAGS in long mode.

FS.base and GS.base Registers. These registers allow 64-bit base-address values to be specified for the FS and GS segments, for use in 64-bit mode. See Section “FS and GS Registers in 64-Bit Mode,” on page 82 for a description of the special treatment the FS and GS segments receive.

KernelGSbase Register. This register is used by the SWAPGS instruction. This instruction exchanges the value located in KernelGSbase with the value located in GS.base.

SYSENTER*x* Registers. The SYSENTER_CS, SYSENTER_ESP, and SYSENTER_EIP registers are used to provide linkage information for the SYSENTER and SYSEXIT instructions. These instructions are only used in legacy mode.

The system-linkage instructions and their use of MSRs are described in Section 6.1 “Fast System Call and Return,” on page 175.

[AMD Public Use]

3.2.3 Memory-Typing Registers

Memory-typing MSRs are used to characterize, or type, memory. Memory typing allows software to control the cacheability of memory, and determine how accesses to memory are ordered. The memory-typing registers perform the following functions:

MTRRcap Register. This register contains information describing the level of MTRR support provided by the processor.

MTRRdefType Register. This register establishes the default memory type to be used for physical memory that is not specifically characterized using the fixed-range and variable-range MTRRs.

MTRRphysBasen and MTRRphysMaskn Registers. These registers form a register pair that can be used to characterize any address range within the physical-memory space, including all of physical memory. Up to eight address ranges of varying sizes can be characterized using these registers.

MTRRfixn Registers. These registers are used to characterize fixed-size memory ranges in the first 1 Mbytes of physical-memory space.

PAT Register. This register allows memory-type characterization based on the virtual (linear) address. It is an extension to the PCD and PWT memory types supported by the legacy paging mechanism. The PAT mechanism provides the same memory-typing capabilities as the MTRRs, but with the added flexibility provided by the paging mechanism.

TOP_MEM and TOP_MEM2 Registers. These top-of-memory registers allow system software to specify physical addresses ranges as memory-mapped I/O locations.

Refer to Section 7.7 “Memory-Type Range Registers,” on page 219 for more information on using these registers.

3.2.4 Debug-Extension Registers

The debug-extension MSRs provide software-debug capability not available in the legacy debug registers (DR0–DR7). These MSRs allow single stepping and recording of control transfers to take place. The debug-extension registers perform the following functions:

DebugCtl Register. This MSR register provides control over control-transfer recording and single stepping, and external-breakpoint reporting and trace messages.

LastBranch*x* and LastInt*x* Registers. The four registers, LastBranchToIP, LastBranchFromIP, LastIntToIP, and LastIntFromIP, are all used to record the source and target of control transfers when branch recording is enabled.

Refer to Section 13.1.6 “Control-Transfer Breakpoint Features,” on page 411 for more information on using these debug registers.

[AMD Public Use]

3.2.5 Performance-Monitoring Registers

The time-stamp counter and performance-monitoring registers are useful in identifying performance bottlenecks. The number of performance counters can vary based on the implementation. These registers perform the following functions:

TSC Register. This register is used to count processor-clock cycles. It can be read using the RDMSR instruction, or it can be read using the either of the *read time-stamp counter* instructions, RDTSC or RDTSCP. System software can make RDTSC or RDTSCP available for use by non-privileged software by clearing the time-stamp disable bit (CR4.TSD) to 0.

***PerfEvtSel*n* Registers.** These registers are used to specify the events counted by the corresponding performance counter, and to control other aspects of its operation.

***PerfCtr*n* Registers.** These registers are performance counters that hold a count of processor, northbridge, or L2 cache events or the duration of events, under the control of the corresponding *PerfEvtSel*n* register. Each *PerfCtr*n* register can be read using the RDMSR instruction, or they can be read using the *read performance-monitor counter* instruction, RDPMC. System software can make RDPMC available for use by non-privileged software by setting the performance-monitor counter enable bit (CR4.PCE) to 1.

Refer to Section 13.2.3 “Using Performance Counters,” on page 424 for more information on using these registers.

3.2.6 Machine-Check Registers

The machine-check registers control the detection and reporting of hardware machine-check errors. The types of errors that can be reported include cache-access errors, load-data and store-data errors, bus-parity errors, and ECC errors. Three types of machine-check MSRs are shown in Figure 3-10 on page 61.

The first type is global machine-check registers, which perform the following functions:

MCG_CAP Register. This register identifies the machine-check capabilities supported by the processor.

MCG_CTL Register. This register provides global control over machine-check-error reporting.

MCG_STATUS Register. This register reports global status on detected machine-check errors.

The second type is error-reporting register banks, which report on machine-check errors associated with a specific processor unit (or group of processor units). There can be different numbers of register banks for each processor implementation, and each bank is numbered from 0 to *i*. The registers in each bank perform the following functions:

MC*i*_CTL Registers. These registers control error-reporting.

MC*i*_STATUS Registers. These registers report machine-check errors.

[AMD Public Use]

MC*i*_ADDR Registers. These registers report the machine-check error address.

MC*i*_MISC Registers. These registers report miscellaneous-error information.

The third type is MCA Extension (MCAX) register banks, which report on machine-check errors associated with a specific processor unit (or group of processor units). There can be different numbers of register banks for each processor implementation, and each bank is numbered from 0 to i. Legacy MCA supports up to 32 banks. MCAX bank 0 will alias to Legacy bank 0 registers. Similarly, MCAX bank n will alias to Legacy bank n registers. The registers in each bank perform the following functions:

MCA_CTL Register. This register is an alias to MCi_CTL for banks 0 to 31. For banks 32 and above, this register controls error-reporting.

MCA_STATUS Register. This register is an alias to MCi_Status.

MCA_ADDR Register. This register is an alias to MCi_ADDR.

MCA_MISC0 Register. This register is an alias to MCi_MISC0.

MCA_CONFIG Register. This register holds configuration information for the MCA bank.

MCA_IPID Register. This register holds information which identifies the specific MCA bank.

MCA_SYND Register. This register stores information associated with the error in MCA_STATUS or MCA_DESTAT.

MCA_DESTAT Register. This register reports deferred machine check errors.

MCA_DEADDR Register. This register provides the address associated with the deferred machine check error.

MCA_MISC[4:1] Registers. Extended miscellaneous error-information registers.

MCA_SYND[2:1] Registers. This register contains information associated with the error in MCA_STATUS or MCA_DESTAT.

MCA_QUERY Registers. These registers report MCA bank error status information.

Refer to Section 9.5 “Using MCA Features,” on page 322 for more information on using these registers.

3.2.7 Shadow Stack Registers

These registers are defined if the shadow stack feature is supported as indicated by CPUID Fn0000_0007_0 ECX[CET_SS] (bit 7) = 1.

[AMD Public Use]

PL0_SSP, PL1_SSP, PL2_SSP Registers. These registers specify the linear address to be loaded into SSP on the next transition to CPLn, where n=0, 1, 2. The linear address must be in canonical format and aligned to 4 bytes.

PL3_SSP Register. The user mode SSP is saved to and restored from this register. The linear address must be in canonical format and aligned to 4 bytes.

ISST_ADDR Register. This register specifies the linear address of the Interrupt Shadow Stack Table (ISST). The linear address must be in canonical format.

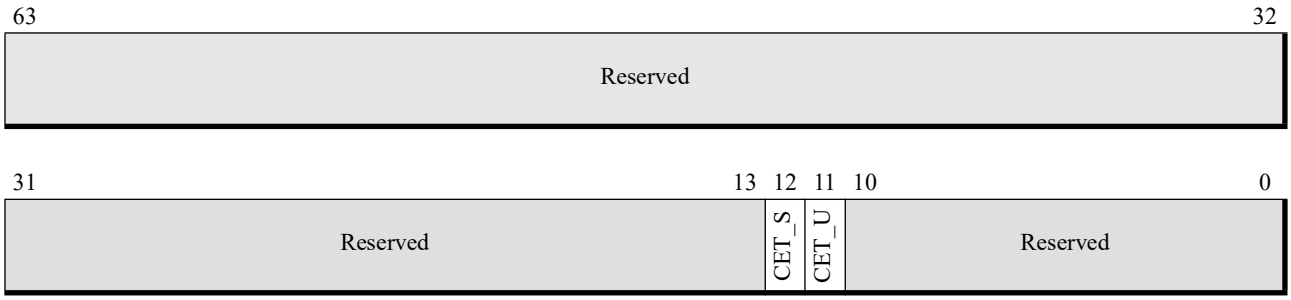
U_CET Register. This register specifies the user mode shadow stack controls.

S_CET Register. This register specifies the supervisor mode shadow stack controls.

3.2.8 Extended State Save Registers

XSS Register. This register contains a bitmap of supervisor-level state components. System software sets bits in the XSS register bitmap to enable management of corresponding state component by the XSAVES/XRSTORS instructions. XSS register support is indicated by CPUID Fn0000_000D_EAX[XSAVES]_x1 = 1.

The XSS bitmap is defined as follows:



Bits	Mnemonic	Description	Access type
63:13	Reserved		MBZ
12	CET_S	Enables the CET_U state component.	R/W
11	CET_U	Enables the CET_S state component.	R/W
10:0	Reserved		MBZ

Figure 3-12. XSS Register

3.2.9 Speculation Control Registers

Modern processors implement hardware techniques such as branch prediction, speculative execution and out-of-order processing to significantly improve performance. If the processor incorrectly predicts or speculates on an outcome, this is detected and any speculative results are discarded. The processor then supplies the architecturally correct, in-order response to the program's instructions. Even though

[AMD Public Use]

the speculative results are discarded, microarchitectural side effects may remain which can be detected by software, and which in some cases may lead to side-channel vulnerabilities.

The two speculation control MSRs, SPEC_CTRL (MSR 048h) and PRED_CMD (MSR 049h), enable hardware features that are designed to limit certain types of speculation.

SPEC_CTRL (MSR 048h)

SPEC_CTRL is a read-write register. Attempts to write a 1 into any reserved bit cause a #GP(0) fault. Unlike most MSRs, a WRMSR to SPEC_CTRL does not serialize memory operations. However, a write to this register is dispatch serializing and prevents execution of younger instructions until the WRMSR has completed. The format of SPEC_CTRL is shown in Figure 3-12.



Bits	Mnemonic	Description	Access type
63:8	Reserved		MBZ
7	PSFD	Predicted Store Forward Disable.	R/W
6:3	Reserved		MBZ
2	SSBD	Speculative Store Bypass Disable.	R/W
1	STIBP	Single-Thread Indirect Branch Prediction Mode.	R/W
0	IBRS	Indirect Branch Restricted Speculation.	R/W

Figure 3-13. SPEC_CTRL Register

The SPEC_CTRL bits defined in Figure 3-13 and are described below:

Indirect Branch Restricted Speculation (IBRS). Bit 0. Setting this bit to 1 prevents indirect branches that occurred in a less privileged prediction mode before this bit was set from influencing the predictions of future indirect branches in a more privileged prediction mode that occur after this bit is set. A lesser privileged prediction mode is defined as CPL 3 or Guest mode, and a more privileged prediction mode is defined as CPL 0-2 or Host mode. Support for IBRS is indicated by CPUID Fn8000_0008_EBX[IBRS] (bit 14) = 1.

After setting IBRS to 1, if software subsequently

- clears IBRS to 0: The processor may allow older indirect branches that occurred when IBRS was previously 0 to influence future indirect branch predictions.
- writes another 1 to IBRS: The processor starts a new window where older indirect branches do not influence future indirect branch predictions.

[AMD Public Use]

Only indirect branches that occurred prior to setting IBRS are prevented from influencing future indirect branches. Therefore, if IBRS were already set on a transition to a more privileged mode from a lesser privileged mode, software at the more privileged mode must write a 1 to IBRS if it requires indirect branch predictions in the new mode to not be influenced by those from the previous mode.

On processors with a shared indirect branch predictor, setting IBRS also prevents indirect branch predictions of one thread from influencing the predictions of its sibling threads, as if SPEC_CTRL[STIBP] was also set. For more information on STIBP, see Single Thread Indirect Branch Predictor below.

Some processors, identified by CPUID Fn8000_0008_EBX[IbrsSameMode] (bit 19) = 1, provide additional speculation limits. For these processors, when IBRS is set, indirect branch predictions are not influenced by any prior indirect branches, regardless of mode (CPL and guest/host) and regardless of whether the prior indirect branches occurred before or after the setting of IBRS. This is referred to as Same Mode IBRS.

Some processors, identified by CPUID Fn8000_0021_EAX[AutomaticIBRS] (bit 8) = 1, support Automatic IBRS. Refer to the description of EFER[AIBRSE] on page 59 for more details.

Although return instructions can be considered a type of indirect branch, IBRS does not affect them. See “Return Address Predictor” on page 71 for more information on IBRS-style indirect branch speculation limits.

Single Thread Indirect Branch Prediction mode (STIBP). Bit 1. Setting this bit to 1 prevents indirect branch predictions of one thread from influencing the predictions of any sibling threads, on processors where branch prediction resources are shared. RET (return) instructions are not influenced by sibling threads. Therefore, setting STIBP is not required to prevent one thread’s RET predictions from influencing the predictions of a sibling thread. Support for STIBP is indicated by CPUID Fn8000_0008_EBX[STIBP] (bit 15) = 1.

Note that STIBP mode is automatically enabled when SPEC_CTRL[IBRS] is set, regardless of value of SPEC_CTRL[STIBP].

Speculative Store Bypass Disable (SSBD). Bit 2. Setting this bit to 1 prevents load-type instructions from speculatively bypassing older store instructions whose final address have not yet been resolved. Support for SSBD is indicated by CPUID Fn8000_0008_EBX[SSBD] (bit 24) = 1.

As specified in Section 7.1.1 “Read Ordering,” on page 189, loads from memory marked with the proper memory type can read memory out-of-order, speculatively and before older stores have completed. This means it is possible for a load to read and pass forward, in a speculative manner, previous values of the memory location. The processor has logic to correct this occurrence and provide the proper in-order load response to the program. However, this mis-speculation may have resulted in microarchitectural side effects. When SSBD is set to 1, the processor can return speculative load data only if there are no older stores with unknown addresses.

Some legacy processors implement SSBD in a different MSR. On these processors, indicated by CPUID Fn8000_0008[SsbdVirtSpecCtrl] (bit 25) = 1, SSBD is enabled by setting

[AMD Public Use]

VIRT_SPEC_CTRL (MSR C001_011F) bit 2. On processors that support both SPEC_CTRL and VIRT_SPEC_CTRL, if SSBD is enabled in either MSR, the processor prevents loads from speculating around older stores. However, it is preferred that software uses SPEC_CTRL[SSBD] in this case.

On some processor models, setting SSBD is not needed to prevent speculative loads from bypassing older stores. This is indicated by CPUID Fn8000_0008_EBX[SsbdNotRequired] (bit 26) = 1.

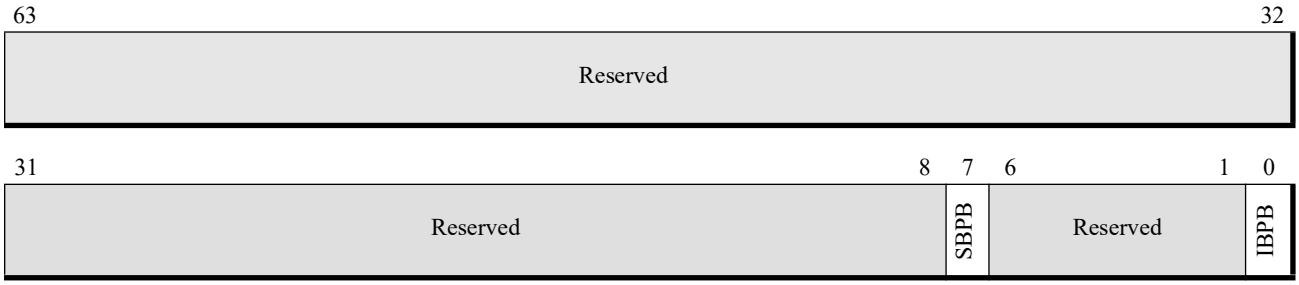
Predicted Store Forward Disable (PSFD). Bit 7. Setting this bit disables Predictive Store Forwarding (PSF). Support for PSFD is indicated by CPUID Fn8000_0008_EBX[PSFD] (bit 28) = 1.

As specified in Section 7.1.1 “Read Ordering,” on page 189, write data for cacheable memory types can be forwarded to read instructions before that data is actually written to memory, via a mechanism called store-to-load forwarding. PSF expands on store-to-load forwarding via a mechanism that predicts the relationship between loads and stores based on past behavior, without waiting for their address calculations to complete. Like other forms of speculative store bypass, an incorrect prediction may result in microarchitectural side effects. PSF speculation can be disabled by setting PSFD.

The PSF feature is also disabled when SPEC_CTRL[SSBD] is set. However, SSBD disables both PSF and speculative store bypass, while PSFD only disables PSF. PSFD may be desirable for software which is concerned with the speculative behavior of PSF but desires a smaller performance impact than setting SSBD.

PRED_CMD (MSR 049h)

PRED_CMD is a write-only register. Attempts to read this register or to write a 1 into any reserved bit cause a #GP(0) fault. Unlike most MSRs, a WRMSR to PRED_CMD does not serialize memory operations. However, a write to this register is dispatch serializing and prevents execution of younger instructions until the WRMSR has completed. The format of the PRED_CMD register is shown in the following figure below.



Bits	Mnemonic	Description	Access type
63:8	Reserved		MBZ
7	SBPB	Selective Branch Prediction Barrier.	W
6:1	Reserved		MBZ
0	IBPB	Indirect Branch Prediction Barrier.	W

Figure 3-14. PRED_CMD Register

[AMD Public Use]

Indirect Branch Prediction Barrier (IBPB). Bit 0, write only. Setting this bit to 1 prevents the processor from using older indirect branch predictions to influence future indirect branches. This applies to JMP indirect and CALL indirect instructions. If CPUID Fn8000_0008_EBX[IBPB_RET] (bit 30) = 1, setting IBPB additionally clears the return address predictor. On processors that support IBPB branch type clearing, indicated by CPUID Fn8000_0021_EAX[IBPB_BRTYPE] (bit 28) = 1, setting IBPB also clears all branch type predictions from the branch predictor. IBPB is intended to be used by software when switching between contexts that do not trust each other. Examples of such contexts include switching from one user context to another, or from one guest to another. CPUID Fn8000_0008_EBX[IBPB] (bit 12) = 1 indicates support for IBPB.

Selective Branch Prediction Barrier (SBPB). Bit 7, write only. Setting this bit to 1 prevents the processor from using older indirect branch predictions to influence future indirect branches. This applies to JMP indirect and CALL indirect instructions. If CPUID Fn8000_0008_EBX[IBPB_RET] (bit 30) = 1, setting SBPB additionally clears the return address predictor. Setting SBPB is not guaranteed to clear older branch type predictions from the CPU branch predictor. If PRED_CMD MSR is written with both IBPB and SBPB bits set to 1, the processor performs an IBPB operation. Support for SBPB is indicated by CPUID Fn8000_0021_EAX[SBPB] (bit 27) = 1.

Return Address Predictor

If IBRS-style indirect branch speculation limits for RET instructions are required and the processor does not support Enhanced Return Address Prediction Security (ERAPS), software should clear return address predictor by executing 32 CALL instructions with a non-zero displacement. If SMEP is enabled, the kernel and user virtual address spaces do not overlap, and at least one unmapped 4K page is separating them, there is no need to clear return address predictor.

The ERAPS feature eliminates the need to execute CALL instructions to clear the return address predictor in most cases. On processors that support ERAPS, return addresses from CALL instructions executed in host mode are not used in guest mode, and vice versa. Additionally, the return address predictor is cleared in all cases when the TLB is implicitly invalidated (see Section 5.5.3 “TLB Management,” on page 161) and in the following cases:

- MOV CR3 instruction
- INVPCID other than single address invalidation (operation type 0)

ERAPS support is indicated by CPUID Fn8000_0021_EAX[ERAPS] (bit 24) = 1. ERAPS virtualization behavior is described in Section 15.37.2 “ERAPS Virtualization,” on page 654.

Additional CPUID Information

The following CPUID functions provide more information on the speculation control features to aid system software in optimizing processor performance:

- CPUID Function 8000_0008_EBX[16] (IBRS always on). When set, indicates that it is recommended that IBRS is only set once during boot and not changed. If IBRS is set on a processor supporting IBRS always on mode, indirect branches executed in a less privileged prediction mode will not influence branch predictions for indirect branches in a more privileged

[AMD Public Use]

prediction mode. This eliminates the need for WRMSR instructions to manage speculation effects at elevated-privilege entry and exit points.

- CPUID Function 8000_0008_EBX[17] (STIBP always on). When set, indicates that it is recommended that STIBP is only set once during boot and not changed. This eliminates the need for a WRMSR instruction at the necessary transition points.
- CPUID Function 8000_0008_EBX[18] (IBRS preferred). When set, indicates that it is recommended to use the IBRS feature instead of other software mitigations such as retpoline. This allows software to remove the software mitigation and utilize the more performant IBRS mechanism.

3.2.10 Hardware Configuration Register (HWCR)

The HWCR register contains control bits that affect the functionality of other features. Some HWCR bits are implementation specific, and are described in the BIOS and Kernel Developer's Guide (BKDG) or Processor Programming Reference Manual applicable to your product. Implementation specific HWCR bits are not listed below.

SmmLock. Bit 0. Enables SMM code lock. When set to 1, SMM code in the ASeg and TSeg memory ranges, and the SMM registers, become read-only and SMI interrupts are not intercepted in SVM. See Section 15.32 "SMM-Lock," on page 612 for information on using this feature.

CpbDis. Bit 25. Core performance boost disable. When set to 1, core performance boost is disabled. See Section 17.2 "Core Performance Boost," on page 698 for further details on this feature.

IRPerfEn. Bit 30. Setting this bit to 1 enables the instructions retired counter. See Section 13.2.1 "Performance Counter MSRs," on page 413 for information on using this feature.

SmmPgCfgLock. Bit 33. Setting this bit to 1 locks the paging configuration while in SMM. See Section 10.3.9 "SMM Page Configuration Lock," on page 341 for information on using this feature.

DowngradeFp512ToFp256. Bit 34. Setting this bit to 1 on processors with a SIMD datapath width of 512 bits or wider reduces the SIMD performance to that of an equivalent processor with a SIMD datapath width of 256 bits. Support for the DowngradeFp512ToFp256 feature is indicated by CPUID Fn8000_0021_EAX[FP512_DOWNGRADE] (bit 21) = 1.

CpuidUserDis. Bit 35. Setting this bit to 1 causes #GP(0) when the CPUID instruction is executed by non-privileged software (CPL is > 0) outside SMM. Support for the CPUID User Disable feature is indicated by CPUID Fn8000_0021_EAX[CpuidUserDis]=1.

ReportSameFpWidth. Bit 40. Setting this bit to 1 causes the processor to report the smallest FP/SIMD datapath width across all cores with the FP128, FP256 or FP512 bits in CPUID Fn8000_001A_EAX. When this bit is cleared, the reported FP/SIMD datapath width is that of the core executing the CPUID instruction. Support for ReportSameFpWidth is indicated by CPUID Fn8000_0021_ECX[SameFpWidthReporting] (bit 10) = 1.

[AMD Public Use]

3.3 Processor Feature Identification

The CPUID instruction provides information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this information. Software can utilize this information to optimize performance.

The CPUID instruction supports multiple functions, each providing specific information about the processor implementation, including the vendor, model number, revision (stepping), features, cache organization, and name. The multifunction approach allows the CPUID instruction to return a detailed picture of the processor implementation and its capabilities—more detailed information than could be returned by a single function. This flexibility also allows for the addition of new CPUID functions in future processor generations.

The desired function number is loaded into the EAX register before executing the CPUID instruction. CPUID functions are divided into two types:

- *Standard functions* return information about features common to all x86 implementations, including the earliest features offered in the x86 architecture, as well as information about the presence of features such as support for the AVX and FMA instruction subsets. Standard function numbers are in the range 0000_0000h–0000_FFFFh.
- *Extended functions* return information about AMD-specific features such as long mode and the presence of features such as support for the FMA4 and XOP instruction subsets. Extended function numbers are in the range 8000_0000h–8000_FFFFh.

Feature information is returned in the EAX, EBX, ECX, and EDX registers. Some functions accept a second input parameter passed to the instruction in the ECX register.

In this and the other three volumes of this *Programmer's Manual*, the notation *CPUID FnXXXX_XXXX_RRR[FieldName]*_x*YY* is used to represent the input parameters and return value that corresponds to a particular processor capability or feature.

In this notation, *XXXX_XXXX* represents the 32-bit value to be placed in the EAX register prior to executing the CPUID instruction. This value is the function number. *RRR* is either EAX, EBX, ECX, or EDX and represents the register to be examined after the execution of the instruction. If the contents of the entire 32-bit register provides the capability information, the notation *[FieldName]* is omitted, otherwise this provides the name of the field within the return value that represents the capability or feature.

When the field is a single bit, this is called a feature flag. Normally, if a feature flag bit is set, the corresponding processor feature is supported and if it is cleared, the feature is not supported. The optional input parameter passed to the CPUID instruction in the ECX register is represented by the notation _x*YY* appended after the return value notation. If a CPUID function does not accept this optional input parameter, this notation is omitted.

For more specific information on the CPUID instruction, see the instruction reference page in Volume 3. For a description of all feature flags related to instruction subset support, see Volume 3, Appendix D, "Instruction Subsets and CPUID Feature Flags." For a comprehensive list of all processor

[AMD Public Use]

capabilities and feature flags, see Volume 3, Appendix E, “Obtaining Processor Information Via the CPUID Instruction.”

[AMD Public Use]

4 Segmented Virtual Memory

The legacy x86 architecture supports a segment-translation mechanism that allows system software to relocate and isolate instructions and data anywhere in the virtual-memory space. A segment is a contiguous block of memory within the linear address space. The size and location of a segment within the linear address space is arbitrary. Instructions and data can be assigned to one or more memory segments, each with its own protection characteristics. The processor hardware enforces the rules dictating whether one segment can access another segment.

The segmentation mechanism provides ten segment registers, each of which defines a single segment. Six of these registers (CS, DS, ES, FS, GS, and SS) define user segments. User segments hold software, data, and the stack and can be used by both application software and system software. The remaining four segment registers (GDT, LDT, IDT, and TR) define system segments. System segments contain data structures initialized and used only by system software. Segment registers contain a *base address* pointing to the starting location of a segment, a *limit* defining the segment size, and *attributes* defining the segment-protection characteristics.

Although segmentation provides a great deal of flexibility in relocating and protecting software and data, it is often more efficient to handle memory isolation and relocation with a combination of software and hardware paging support. For this reason, most modern system software bypasses the segmentation features. However, segmentation cannot be completely disabled, and an understanding of the segmentation mechanism is important to implementing long-mode system software.

In long mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode:

- In compatibility mode, segmentation functions just as it does in legacy mode, using legacy 16-bit or 32-bit protected mode semantics.
- 64-bit mode, segmentation is disabled, creating a flat 64-bit virtual-address space. As will be seen, certain functions of some segment registers, particularly the system-segment registers, continue to be used in 64-bit mode.

4.1 Real Mode Segmentation

After reset or power-up, the processor always initially enters real mode. Protected modes are entered from real mode.

As noted in “Real Addressing” on page 10, real mode (real-address mode), provides a physical-memory space of 1 Mbyte. In this mode, a 20-bit physical address is determined by shifting a 16-bit segment selector to the left four bits and adding the 16-bit effective address.

Each 64K segment (CS, DS, ES, FS, GS, SS) is aligned on 16-byte boundaries. The *segment base* is the lowest address in a given segment, and is equal to the segment selector * 16. The POP and MOV instructions can be used to load a (possibly) new segment selector into one of the segment registers.

[AMD Public Use]

When this occurs, the selector is updated and the segment base is set to selector * 16. The segment limit and segment attributes are unchanged, but are normally 64K (the maximum allowable limit) and read/write data, respectively.

On FAR transfers, CS (code segment) selector is updated to the new value, and the CS segment base is set to selector * 16. The CS segment limit and attributes are unchanged, but are usually 64K and read/write, respectively.

If the interrupt descriptor table (IDT) is used to find the real mode IDT see “Real-Mode Interrupt Control Transfers” on page 270.

The GDT, LDT, and TSS (see below) are not used in real mode.

4.2 Virtual-8086 Mode Segmentation

Virtual-8086 mode supports 16-bit real mode programs running under protected mode (see below). It uses a simple form of memory segmentation, optional paging, and limited protection checking. Programs running in virtual-8086 mode can access up to 1MB of memory space.

As with real mode segmentation, each 64K segment (CS, DS, ES, FS, GS, SS) is aligned on 16-byte boundaries. The *segment base* is the lowest address in a given segment, and is equal to the segment selector * 16. The POP and MOV instructions work exactly as in real mode and can be used to load a (possibly) new segment selector into one of the segment registers. When this occurs, the selector is updated and the segment base is set to selector * 16. The segment limit and segment attributes are unchanged, but are normally 64K (the maximum allowable limit) and read/write data, respectively.

FAR transfers, with the exception of interrupts and exceptions, operate as in real mode. On FAR transfers, the CS (code segment) selector is updated to the new value, and the CS segment base is set to selector * 16. The CS segment limit and attributes are unchanged, but are usually 64K and read/write, respectively. Interrupts and exceptions switch the processor to protected mode. (See Chapter 8, “Exceptions and Interrupts” for more information.)

4.3 Protected Mode Segmented-Memory Models

System software can use the segmentation mechanism to support one of two basic segmented-memory models: a flat-memory model or a multi-segmented model. These segmentation models are supported in legacy mode and in compatibility mode. Each type of model is described in the following sections.

4.3.1 Multi-Segmented Model

In the multi-segmented memory model, each segment register can reference a unique base address with a unique segment size. Segments can be as small as a single byte or as large as 4 Gbytes. When page translation is used, multiple segments can be mapped to a single page and multiple pages can be mapped to a single segment. Figure 1-1 on page 6 shows an example of the multi-segmented model.

[AMD Public Use]

The multi-segmented memory model provides the greatest level of flexibility for system software using the segmentation mechanism.

Compatibility mode allows the multi-segmented model to be used in support of legacy software. However, in compatibility mode, the multi-segmented memory model is restricted to the first 4 Gbytes of virtual-memory space. Access to virtual memory above 4 Gbytes requires the use of 64-bit mode, which does not support segmentation.

4.3.2 Flat-Memory Model

The flat-memory model is the simplest form of segmentation to implement. Although segmentation cannot be disabled, the flat-memory model allows system software to bypass most of the segmentation mechanism. In the flat-memory model, all segment-base addresses have a value of 0 and the segment limits are fixed at 4 Gbytes. Clearing the segment-base value to 0 effectively disables segment translation, resulting in a single segment spanning the entire virtual-address space. All segment descriptors reference this single, flat segment. Figure 1-2 on page 7 shows an example of the flat-memory model.

4.3.3 Segmentation in 64-Bit Mode

In 64-bit mode, segmentation is disabled. The segment-base value is ignored and treated as 0 by the segmentation hardware. Likewise, segment limits and most attributes are ignored. There are a few exceptions. The CS-segment DPL, D, and L attributes are used (respectively) to establish the privilege level for a program, the default operand size, and whether the program is running in 64-bit mode or compatibility mode. The FS and GS segments can be used as additional base registers in address calculations, and those segments can have non-zero base-address values. This facilitates addressing thread-local data and certain system-software data structures. See “FS and GS Registers in 64-Bit Mode” on page 82 for details about the FS and GS segments in 64-bit mode. The system-segment registers are always used in 64-bit mode.

4.4 Segmentation Data Structures and Registers

Figure 4-1 on page 78 shows the following data structures used by the segmentation mechanism:

- *Segment Descriptors*—As the name implies, a segment descriptor *describes* a segment, including its location in virtual-address space, its size, protection characteristics, and other attributes.
- *Descriptor Tables*—Segment descriptors are stored in memory in one of three tables. The global-descriptor table (GDT) holds segment descriptors that can be shared among all tasks. Multiple local-descriptor tables (LDT) can be defined to hold descriptors that are used by specific tasks and are not shared globally. The interrupt-descriptor table (IDT) holds gate descriptors that are used to access the segments where interrupt handlers are located.
- *Task-State Segment*—A task-state segment (TSS) is a special type of system segment that contains task-state information and data structures for each task. For example, a TSS holds a copy of the GPRs and EFLAGS register when a task is suspended. A TSS also holds the pointers to privileged-

[AMD Public Use]

software stacks. The TSS and task-switch mechanism are described in Chapter 12, “Task Management.”

- *Segment Selectors*—Descriptors are selected for use from the descriptor tables using a segment selector. A segment selector contains an index into either the GDT or LDT. The IDT is indexed using an interrupt vector, as described in “Legacy Protected-Mode Interrupt Control Transfers” on page 272, and in “Long-Mode Interrupt Control Transfers” on page 283.

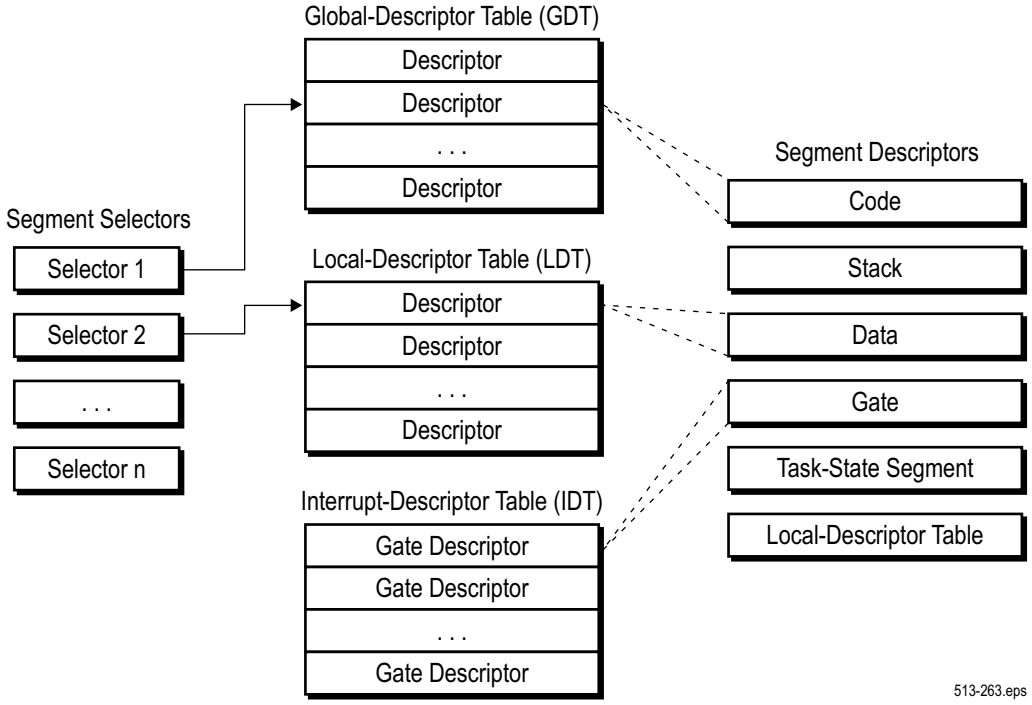


Figure 4-1. Segmentation Data Structures

Figure 4-2 on page 79 shows the registers used by the segmentation mechanism. The registers have the following relationship to the data structures:

- *Segment Registers*—The six segment registers (CS, DS, ES, FS, GS, and SS) are used to point to the user segments. A segment selector selects a descriptor when it is loaded into one of the segment registers. This causes the processor to automatically load the selected descriptor into a software-invisible portion of the segment register.
- *Descriptor-Table Registers*—The three descriptor-table registers (GDTR, LDTR, and IDTR) are used to point to the system segments. The descriptor-table registers identify the virtual-memory location and size of the descriptor tables.
- *Task Register (TR)*—Describes the location and limit of the current task state segment (TSS).

[AMD Public Use]

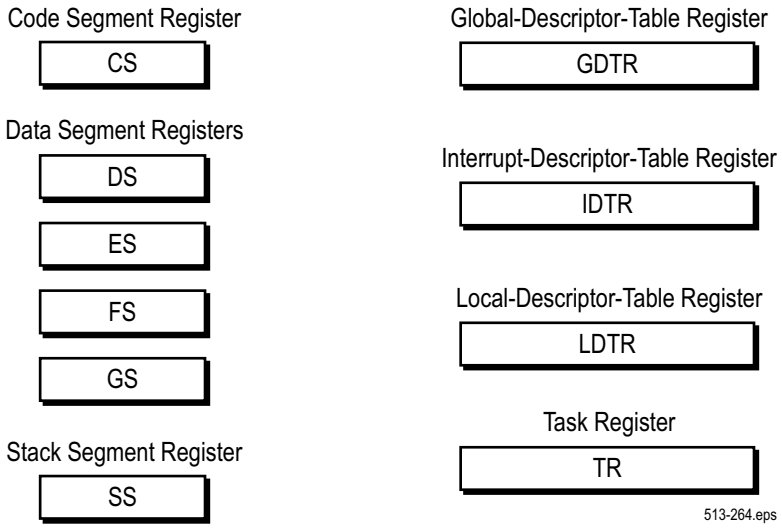


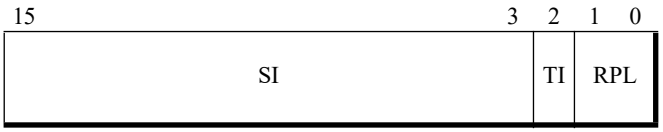
Figure 4-2. Segment and Descriptor-Table Registers

A fourth system-segment register, the TR, points to the TSS. The data structures and registers associated with task-state segments are described in “Task-Management Resources” on page 369.

4.5 Segment Selectors and Registers

4.5.1 Segment Selectors

Segment selectors are pointers to specific entries in the global and local descriptor tables. Figure 4-3 shows the segment selector format.



Bits	Mnemonic	Description
15:3	SI	Selector Index
2	TI	Table Indicator
1:0	RPL	Requestor Privilege Level

Figure 4-3. Segment Selector

[AMD Public Use]

The selector format consists of the following fields:

Selector Index Field. Bits 15:3. The selector-index field specifies an entry in the descriptor table. Descriptor-table entries are eight bytes long, so the selector index is scaled by 8 to form a byte offset into the descriptor table. The offset is then added to either the global or local descriptor-table base address (as indicated by the table-index bit) to form the descriptor-entry address in virtual-address space.

Some descriptor entries in long mode are 16 bytes long rather than 8 bytes (see "Legacy Segment Descriptors" on page 90 for more information on long-mode descriptor-table entries). These expanded descriptors consume two entries in the descriptor table. Long mode, however, continues to scale the selector index by eight to form the descriptor-table offset. It is the responsibility of system software to assign selectors such that they correctly point to the start of an expanded entry.

Table Indicator (TI) Bit. Bit 2. The TI bit indicates which table holds the descriptor referenced by the selector index. When TI=0 the GDT is used and when TI=1 the LDT is used. The descriptor-table base address is read from the appropriate descriptor-table register and added to the scaled selector index as described above.

Requestor Privilege-Level (RPL) Field. Bits 1:0. The RPL represents the privilege level (CPL) the processor is operating under at the time the selector is created.

RPL is used in segment privilege-checks to prevent software running at lesser privilege levels from accessing privileged data. See "Data-Access Privilege Checks" on page 108 and "Control-Transfer Privilege Checks" on page 111 for more information on segment privilege-checks.

Null Selector. Null selectors have a selector index of 0 and TI=0, corresponding to the first entry in the GDT. However, null selectors do not reference the first GDT entry but are instead used to invalidate unused segment registers. A general-protection exception (#GP) occurs if a reference is made to use a segment register containing a null selector in non-64-bit mode. By initializing unused segment registers with null selectors software can trap references to unused segments.

Null selectors can only be loaded into the DS, ES, FS and GS data-segment registers, and into the LDTR descriptor-table register. A #GP occurs if software attempts to load the CS register with a null selector or if software attempts to load the SS register with a null selector in non 64-bit mode or at CPL 3.

If CPUID Fn8000_0021_EAX[NullSelectorClearsBase] (bit 6) = 1, loading a segment register with a null selector clears the base address and limit of the segment register in all cases except a load of DS, ES, FS, or GS by an IRET, IRETD, IRETQ, or RETF instruction that changes the current privilege level, in which case these fields are left untouched. If CPUID Fn8000_0021_EAX[NullSelectorClearsBase] (bit 6) = 0, loading a segment register with a null selector makes the base address and limit of the segment register undefined. Because references to segment registers containing a null selector cause a #GP exception, the segment base and limit values have no effect. However, OS management of segment state may be simplified for processors supporting this clearing functionality.

[AMD Public Use]

4.5.2 Segment Registers

Six 16-bit segment registers are provided for referencing up to six segments at one time. All software tasks require segment selectors to be loaded in the CS and SS registers. Use of the DS, ES, FS, and GS segments is optional, but nearly all software accesses data and therefore requires a selector in the DS register. Table 4-1 on page 81 lists the supported segment registers and their functions.

Table 4-1. Segment Registers

Segment Register	Encoding	Segment Register Function
ES	/0	References optional data-segment descriptor entry
CS	/1	References code-segment descriptor entry
SS	/2	References stack segment descriptor entry
DS	/3	References default data-segment descriptor entry
FS	/4	References optional data-segment descriptor entry
GS	/5	References optional data-segment descriptor entry

The processor maintains a *hidden portion* of the segment register in addition to the selector value loaded by software. This hidden portion contains the values found in the descriptor-table entry referenced by the segment selector. The processor loads the descriptor-table entry into the hidden portion when the segment register is loaded. By keeping the corresponding descriptor-table entry in hardware, performance is optimized for the majority of memory references.

Figure 4-4 shows the format of the visible and hidden portions of the segment register. Except for the FS and GS segment base, software cannot directly read or write the hidden portion (shown as gray-shaded boxes in Figure 4-4).

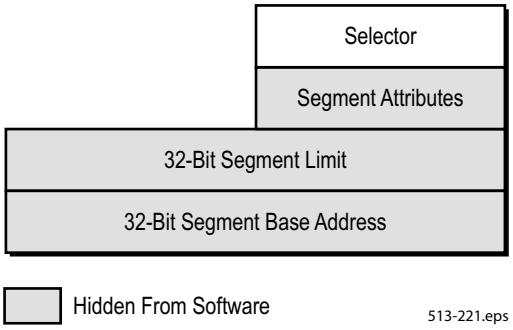


Figure 4-4. Segment-Register Format

CS Register. The CS register contains the segment selector referencing the current code-segment descriptor entry. All instruction fetches reference the CS descriptor. When a new selector is loaded into the CS register, the current-privilege level (CPL) of the processor is set to that of the CS-segment descriptor-privilege level (DPL).

[AMD Public Use]

Data-Segment Registers. The DS register contains the segment selector referencing the default data-segment descriptor entry. The SS register contains the stack-segment selector. The ES, FS, and GS registers are optionally loaded with segment selectors referencing other data segments. Data accesses default to referencing the DS descriptor except in the following two cases:

- The ES descriptor is referenced for string-instruction destinations.
- The SS descriptor is referenced for stack operations.

4.5.3 Segment Registers in 64-Bit Mode

CS Register in 64-Bit Mode. In 64-bit mode, most of the hidden portion of the CS register is ignored. Only the L (long), D (default operation size), and DPL (descriptor privilege-level) attributes are recognized by 64-bit mode. Address calculations assume a CS.base value of 0. CS references do not check the CS.limit value, but instead check that the effective address is in canonical form.

DS, ES, and SS Registers in 64-Bit Mode. In 64-bit mode, the contents of the ES, DS, and SS segment registers are ignored. All fields (base, limit, and attribute) in the hidden portion of the segment registers are ignored.

Address calculations in 64-bit mode that reference the ES, DS, or SS segments are treated as if the segment base is 0. Instead of performing limit checks, the processor checks that all virtual-address references are in canonical form.

Neither enabling and activating long mode nor switching between 64-bit and compatibility modes changes the contents of the visible or hidden portions of the segment registers. These registers remain unchanged during 64-bit mode execution unless explicit segment loads are performed.

FS and GS Registers in 64-Bit Mode. Unlike the CS, DS, ES, and SS segments, the FS and GS segment overrides can be used in 64-bit mode. When FS and GS segment overrides are used in 64-bit mode, their respective base addresses are used in the effective-address (EA) calculation. The complete EA calculation then becomes (FS or GS).base + base + (scale ∗ index) + displacement. The FS.base and GS.base values are also expanded to the full 64-bit virtual-address size, as shown in Figure 4-5. Any overflow in the 64-bit linear address calculation is ignored and the resulting address instead wraps around to the other end of the address space.

[AMD Public Use]

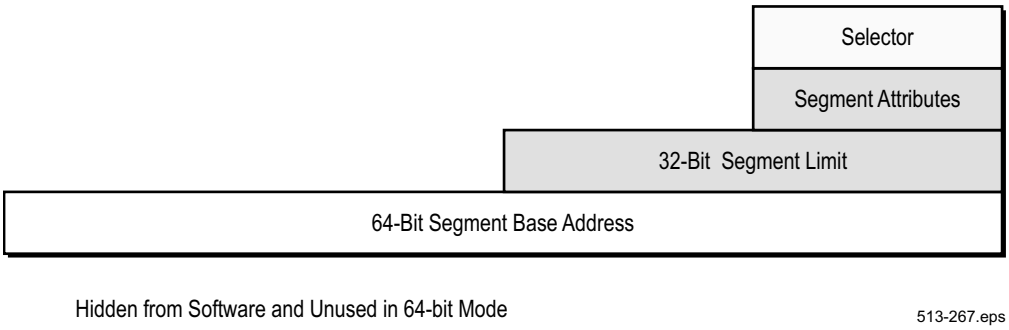


Figure 4-5. FS and GS Segment-Register Format—64-Bit Mode

In 64-bit mode, FS-segment and GS-segment overrides are not checked for limit or attributes. Instead, the processor checks that all virtual-address references are in canonical form.

Segment register-load instructions (MOV to Sreg and POP Sreg) load only a 32-bit base-address value into the hidden portion of the FS and GS segment registers. The base-address bits above the low 32 bits are cleared to 0 as a result of a segment-register load. When a null selector is loaded into FS or GS, the contents of the corresponding hidden descriptor register are not altered.

There are two methods to update the contents of the FS.base and GS.base hidden descriptor fields. The first is available exclusively to privileged software (CPL = 0). The FS.base and GS.base hidden descriptor-register fields are mapped to MSRs. Privileged software can load a 64-bit base address in canonical form into FS.base or GS.base using a single WRMSR instruction. The FS.base MSR address is C000_0100h while the GS.base MSR address is C000_0101h.

The second method of updating the FS and GS base fields is available to software running at any privilege level (when supported by the implementation and enabled by setting CR4[FSGSBASE]). The WRFSBASE and WRGSBASE instructions copy the contents of a GPR to the FS.base and GS.base fields respectively. When the operand size is 32 bits, the upper doubleword of the base is cleared. WRFSBASE and WRGSBASE are only supported in 64-bit mode.

The addresses written into the expanded FS.base and GS.base registers must be in canonical form. Any instruction that attempts to write a non-canonical address to these registers causes a general-protection exception (#GP) to occur.

When in compatibility mode, the FS and GS overrides operate as defined by the legacy x86 architecture regardless of the value loaded into the high 32 bits of the hidden descriptor-register base-address field. Compatibility mode ignores the high 32 bits when calculating an effective address.

[AMD Public Use]

4.6 Descriptor Tables

Descriptor tables are used by the segmentation mechanism when protected mode is enabled (CR0.PE=1). These tables hold descriptor entries that describe the location, size, and privilege attributes of a segment. All memory references in protected mode access a descriptor-table entry.

As previously mentioned, there are three types of descriptor tables supported by the x86 segmentation mechanism:

- Global descriptor table (GDT)
- Local descriptor table (LDT)
- Interrupt descriptor table (IDT)

Software establishes the location of a descriptor table in memory by initializing its corresponding descriptor-table register. The descriptor-table registers and the descriptor tables are described in the following sections.

4.6.1 Global Descriptor Table

Protected-mode system software must create a global descriptor table (GDT). The GDT contains code-segment and data-segment descriptor entries (user segments) for segments that can be shared by all tasks. In addition to the user segments, the GDT can also hold gate descriptors and other system-segment descriptors. System software can store the GDT anywhere in memory and should protect the segment containing the GDT from non-privileged software.

Segment selectors point to the GDT when the table-index (TI) bit in the selector is cleared to 0. The selector index portion of the segment selector references a specific entry in the GDT. Figure 4-6 on page 85 shows how the segment selector indexes into the GDT. One special form of a segment selector is the *null selector*. A null selector points to the first entry in the GDT (the selector index is 0 and TI=0). However, null selectors do not reference memory, so the first GDT entry cannot be used to describe a segment (see “Null Selector” on page 80 for information on using the null selector). The first usable GDT entry is referenced with a selector index of 1.

[AMD Public Use]

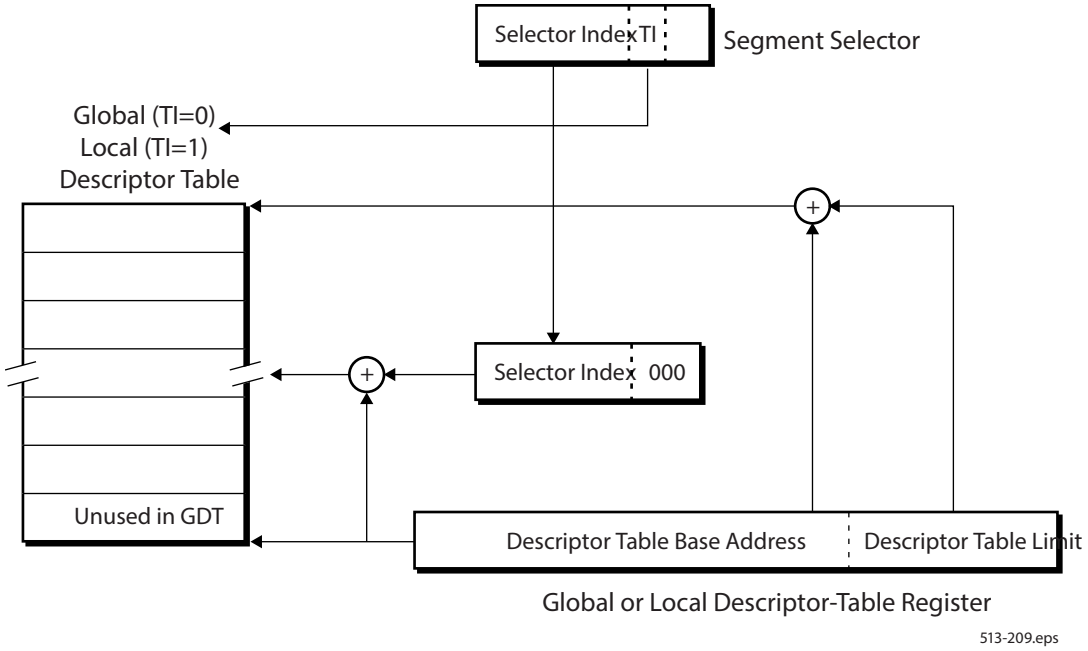


Figure 4-6. Global and Local Descriptor-Table Access

4.6.2 Global Descriptor-Table Register

The global descriptor-table register (GDTR) points to the location of the GDT in memory and defines its size. This register is loaded from memory using the LGDT instruction (see “LGDT and LIDT Instructions” on page 181). Figure 4-7 shows the format of the GDTR in legacy mode and compatibility mode.

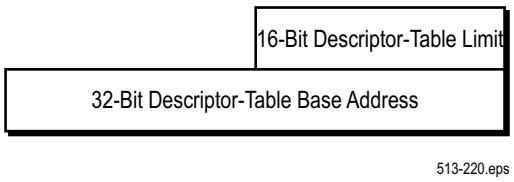


Figure 4-7. GDTR and IDTR Format—Legacy Modes

Figure 4-8 on page 86 shows the format of the GDTR in 64-bit mode.

[AMD Public Use]

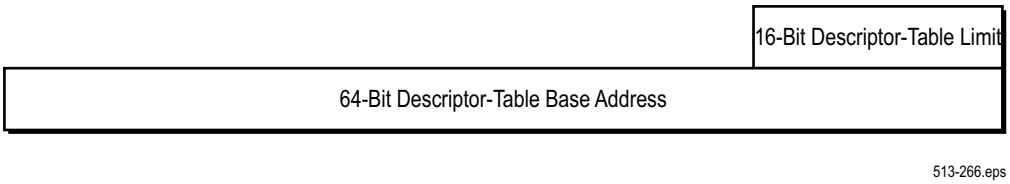


Figure 4-8. GDTR and IDTR Format—Long Mode

The GDTR contains two fields:

Limit. 2 bytes. These bits define the 16-bit limit, or size, of the GDT in bytes. The limit value is added to the base address to yield the ending byte address of the GDT. A general-protection exception (#GP) occurs if software attempts to access a descriptor beyond the GDT limit.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the GDTR and IDTR limit-field sizes are unchanged from the legacy sizes. The processor does check the limits in long mode during GDT and IDT accesses.

Base Address. 8 bytes. The base-address field holds the starting byte address of the GDT in virtual-memory space. The GDT can be located at any byte address in virtual memory, but system software should align the GDT on a quadword boundary to avoid the potential performance penalties associated with accessing unaligned data.

The AMD64 architecture increases the base-address field of the GDTR to 64 bits so that system software running in long mode can locate the GDT anywhere in the 64-bit virtual-address space. The processor ignores the high-order 4 bytes of base address when running in legacy mode.

4.6.3 Local Descriptor Table

Protected-mode system software can optionally create a local descriptor table (LDT) to hold segment descriptors belonging to a single task or even multiple tasks. The LDT typically contains code-segment and data-segment descriptors as well as gate descriptors referenced by the specified task. Like the GDT, system software can store the LDT anywhere in memory and should protect the segment containing the LDT from non-privileged software.

Segment selectors point to the LDT when the table-index bit (TI) in the selector is set to 1. The selector index portion of the segment selector references a specific entry in the LDT (see Figure 4-6 on page 85). Unlike the GDT, however, a selector index of 0 references the first entry in the LDT (when TI=1, the selector is not a null selector).

LDTs are described by system-segment descriptor entries located in the GDT, and a GDT can contain multiple LDT descriptors. The LDT system-segment descriptor defines the location, size, and privilege rights for the LDT. Figure 4-9 on page 87 shows the relationship between the LDT and GDT data structures.

[AMD Public Use]

Loading a null selector into the LDTR is useful if software does not use an LDT. This causes a #GP if an erroneous reference is made to the LDT.

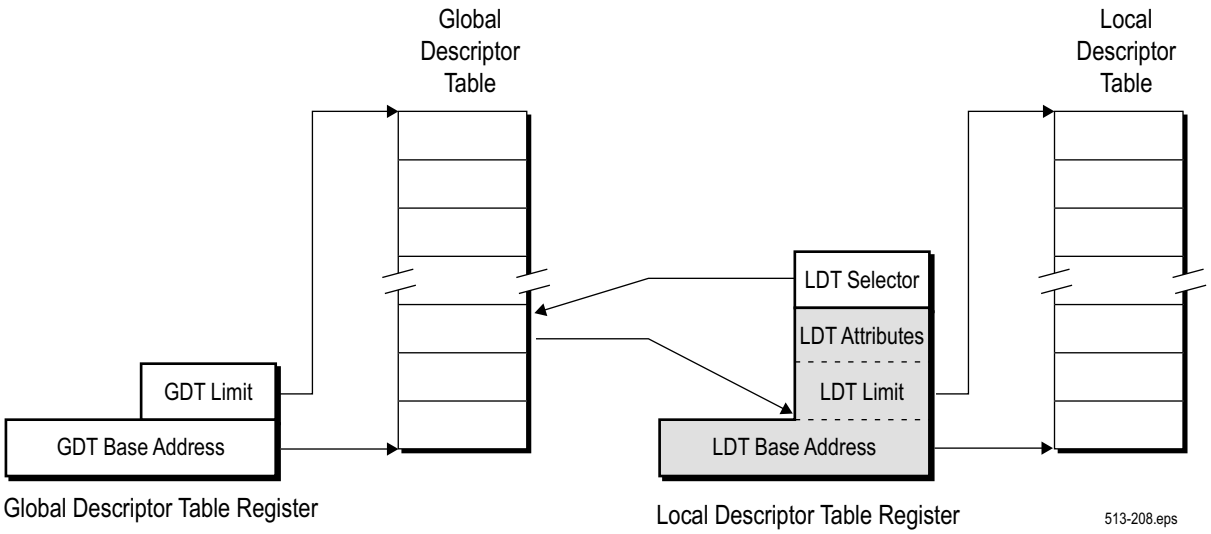


Figure 4-9. Relationship between the LDT and GDT

4.6.4 Local Descriptor-Table Register

The local descriptor-table register (LDTR) points to the location of the LDT in memory, defines its size, and specifies its attributes. The LDTR has two portions. A *visible* portion holds the LDT selector, and a *hidden* portion holds the LDT descriptor. When the LDT selector is loaded into the LDTR, the processor automatically loads the LDT descriptor from the GDT into the hidden portion of the LDTR. The LDTR is loaded in one of two ways:

- Using the LLDT instruction (see “LLDT and LTR Instructions” on page 182).
- Performing a task switch (see “Switching Tasks” on page 382).

Figure 4-10 on page 88 shows the format of the LDTR in legacy mode.

[AMD Public Use]

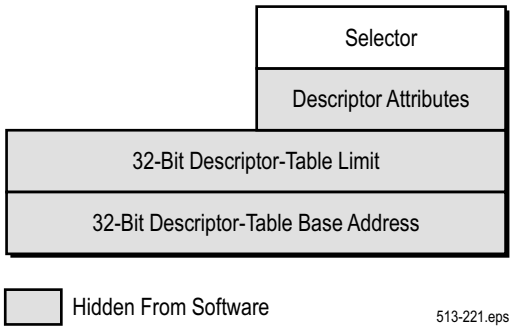


Figure 4-10. LDTR Format—Legacy Mode

Figure 4-11 shows the format of the LDTR in long mode (both compatibility mode and 64-bit mode).

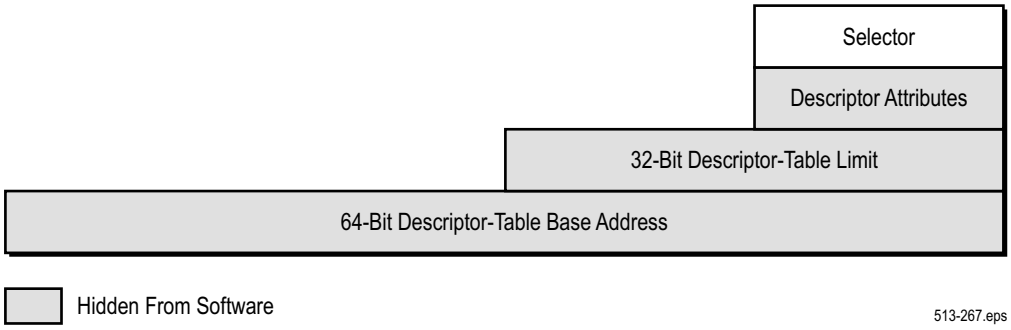


Figure 4-11. LDTR Format—Long Mode

The LDTR contains four fields:

LDT Selector. 2 bytes. These bits are loaded explicitly from the TSS during a task switch, or by using the LLDT instruction. The LDT selector must point to an LDT system-segment descriptor entry in the GDT. If it does not, a general-protection exception (#GP) occurs.

The following three fields are loaded automatically from the LDT descriptor in the GDT as a result of loading the LDT selector. The register fields are shown as shaded boxes in Figure 4-10 and Figure 4-11.

Base Address. The base-address field holds the starting byte address of the LDT in virtual-memory space. Like the GDT, the LDT can be located anywhere in system memory, but software should align the LDT on a quadword boundary to avoid performance penalties associated with accessing unaligned data.

[AMD Public Use]

The AMD64 architecture expands the base-address field of the LDTR to 64 bits so that system software running in long mode can locate an LDT anywhere in the 64-bit virtual-address space. The processor ignores the high-order 32 base-address bits when running in legacy mode. Because the LDTR is loaded from the GDT, the system-segment descriptor format (LDTs are system segments) has been expanded by the AMD64 architecture in support of 64-bit mode. See “Long Mode Descriptor Summary” on page 105 for more information on this expanded format. The high-order base-address bits are only loaded from 64-bit mode using the LLDT instruction (see “LLDT and LTR Instructions” on page 182 for more information on this instruction).

Limit. This field defines the limit, or size, of the LDT in bytes. The LDT limit as stored in the LDTR is 32 bits. When the LDT limit is loaded from the GDT descriptor entry, the 20-bit limit field in the descriptor is expanded to 32 bits and scaled based on the value of the descriptor granularity (G) bit. For details on the limit biasing and granularity, see “Granularity (G) Bit” on page 92.

If an attempt is made to access a descriptor beyond the LDT limit, a general-protection exception (#GP) occurs.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the LDTR limit-field size is unchanged from the legacy size. The processor does check the LDT limit in long mode during LDT accesses.

Attributes. This field holds the descriptor attributes, such as privilege rights, segment presence and segment granularity.

4.6.5 Interrupt Descriptor Table

The final type of descriptor table is the interrupt descriptor table (IDT). Multiple IDTs can be maintained by system software. System software selects a specific IDT by loading the interrupt descriptor table register (IDTR) with a pointer to the IDT. As with the GDT and LDT, system software can store the IDT anywhere in memory and should protect the segment containing the IDT from non-privileged software.

The IDT can contain only the following types of gate descriptors:

- Interrupt gates
- Trap gates
- Task gates.

The use of gate descriptors by the interrupt mechanism is described in Chapter 8, “Exceptions and Interrupts.” A general-protection exception (#GP) occurs if the IDT descriptor referenced by an interrupt or exception is not one of the types listed above.

IDT entries are selected using the interrupt vector number rather than a selector value. The interrupt vector number is scaled by the interrupt-descriptor entry size to form an offset into the IDT. The interrupt-descriptor entry size depends on the processor operating mode as follows:

- In long mode, interrupt descriptor-table entries are 16 bytes.

[AMD Public Use]

- In legacy mode, interrupt descriptor-table entries are eight bytes.

Figure 4-12 shows how the interrupt vector number indexes the IDT.

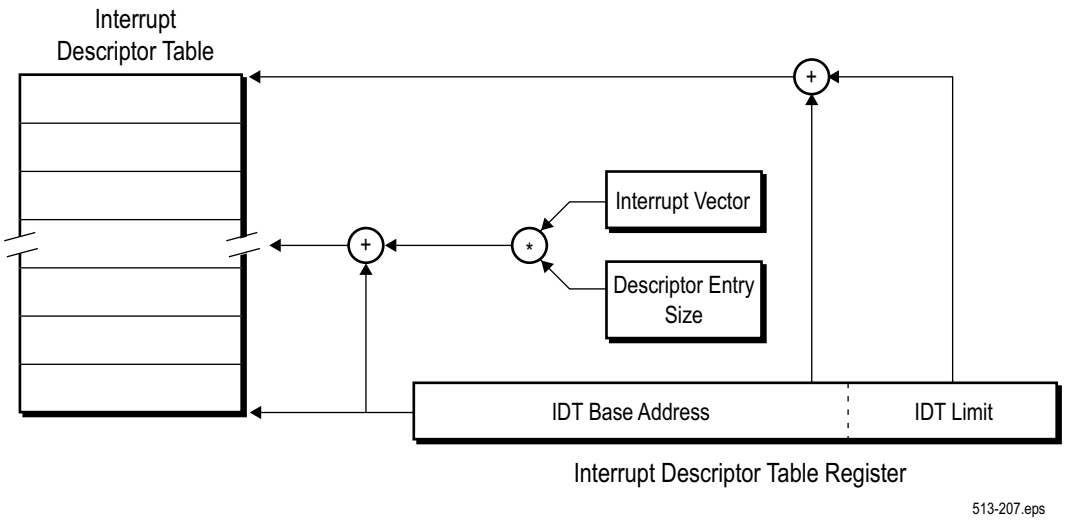


Figure 4-12. Indexing an IDT

4.6.6 Interrupt Descriptor-Table Register

The interrupt descriptor-table register (IDTR) points to the IDT in memory and defines its size. This register is loaded from memory using the LIDT instruction (see “LGDT and LIDT Instructions” on page 181). The format of the IDTR is identical to that of the GDTR in all modes. Figure 4-7 on page 85 shows the format of the IDTR in legacy mode. Figure 4-8 on page 86 shows the format of the IDTR in long mode.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the IDTR limit-field size is unchanged from the legacy size. The processor does check the IDT limit in long mode during IDT accesses.

4.7 Legacy Segment Descriptors

4.7.1 Descriptor Format

Segment descriptors define, protect, and isolate segments from each other. There are two basic types of descriptors, each of which are used to describe different segment (or gate) types:

- *User Segments*—These include code segments and data segments. Stack segments are a type of data segment.

[AMD Public Use]

- *System Segments*—System segments consist of LDT segments and task-state segments (TSS). Gate descriptors are another type of system-segment descriptor. Rather than describing segments, gate descriptors point to program entry points.

Figure 4-13 shows the generic format for user-segment and system-segment descriptors. User and system segments are differentiated using the S bit. S=1 indicates a user segment, and S=0 indicates a system segment. Gray shading indicates the field or bit is reserved. The format for a gate descriptor differs from the generic segment descriptor, and is described separately in “Gate Descriptors” on page 97.

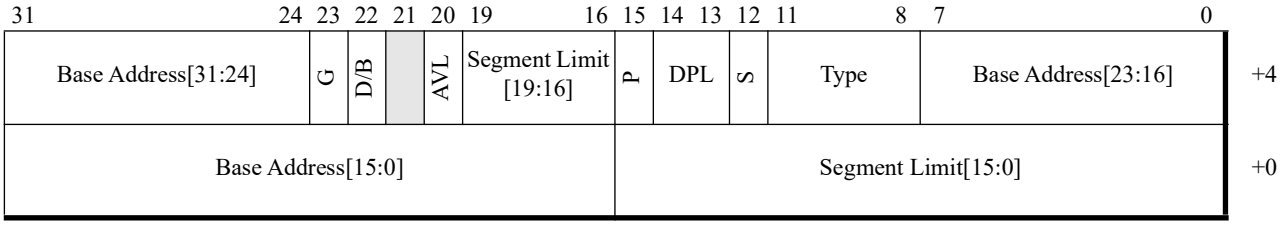


Figure 4-13. Generic Segment Descriptor—Legacy Mode

Figure 4-13 shows the fields in a generic, legacy-mode, 8-byte (two doubleword) segment descriptor. In this figure, the upper doubleword (located at byte offset +4) is shown on top and the lower doubleword (located at byte offset +0) is shown on the bottom. The fields are defined as follows:

Segment Limit. The 20-bit segment limit is formed by concatenating bits 19:16 of the upper doubleword with bits 15:0 of lower doubleword. The segment limit defines the segment size, in bytes. The granularity (G) bit controls how the segment-limit field is scaled (see “Granularity (G) Bit” on page 92). For data segments, the expand-down (E) bit determines whether the segment limit defines the lower or upper segment-boundary (see “Expand-Down (E) Bit” on page 95).

If software references a segment descriptor with an address beyond the segment limit, a general-protection exception (#GP) occurs. The #GP occurs if any part of the memory reference falls outside the segment limit. For example, a doubleword (4-byte) address reference causes a #GP if one or more bytes are located beyond the segment limit.

Base Address. The 32-bit base address is formed by concatenating bits 31:24 of the upper doubleword with bits 7:0 of the same doubleword and bits 15:0 of the lower doubleword. The segment-base address field locates the start of a segment in virtual-address space.

[AMD Public Use]

S Bit and Type Field. Bit 12 and bits 11:8 of the upper doubleword. The S and Type fields, together, specify the descriptor type and its access characteristics. Table 4-2 summarizes the descriptor types by S-field encoding and gives a cross reference to descriptions of the Type-field encodings.

Table 4-2. Descriptor Types

S Field	Descriptor Type	Type-Field Encoding
0 (System)	LDT	See Table 4-5 on page 96
	TSS	
	Gate	
1 (User)	Code	See Table 4-3 on page 94
	Data	See Table 4-4 on page 95

Descriptor Privilege-Level (DPL) Field. Bits 14:13 of the upper doubleword. The DPL field indicates the descriptor-privilege level of the segment. DPL can be set to any value from 0 to 3, with 0 specifying the most privilege and 3 the least privilege. See "Data-Access Privilege Checks" on page 108 and "Control-Transfer Privilege Checks" on page 111 for more information on how the DPL is used during segment privilege-checks.

Present (P) Bit. Bit 15 of the upper doubleword. The segment-present bit indicates that the segment referenced by the descriptor is loaded in memory. If a reference is made to a descriptor entry when P = 0, a segment-not-present exception (#NP) occurs. This bit is set and cleared by system software and is never altered by the processor.

Available To Software (AVL) Bit. Bit 20 of the upper doubleword. This field is available to software, which can write any value to it. The processor does not set or clear this field.

Default Operand Size (D/B) Bit. Bit 22 of the upper doubleword. The default operand-size bit is found in code-segment and data-segment descriptors but not in system-segment descriptors. Setting this bit to 1 indicates a 32-bit default operand size, and clearing it to 0 indicates a 16-bit default size. The effect this bit has on a segment depends on the segment-descriptor type. See "Code-Segment Default-Operand Size (D) Bit" on page 94 for a description of the D bit in code-segment descriptors. "Data-Segment Default Operand Size (D/B) Bit" on page 96 describes the D bit in data-segment descriptors, including stack segments, where the bit is referred to as the "B" bit.

Granularity (G) Bit. Bit 23 of the upper doubleword. The granularity bit specifies how the segment-limit field is scaled. Clearing the G bit to 0 indicates that the limit field is not scaled. In this case, the limit equals the number of bytes available in the segment. Setting the G bit to 1 indicates that the limit field is scaled by 4 Kbytes (4096 bytes). Here, the limit field equals the number of 4-Kbyte *blocks* available in the segment.

Setting a limit of 0 indicates a 1-byte segment limit when G = 0. Setting the same limit of 0 when G = 1 indicates a segment limit of 4095.

[AMD Public Use]

Reserved Bits. Generally, software should clear all reserved bits to 0, so they can be defined in future revisions to the AMD64 architecture.

4.7.2 Code-Segment Descriptors

Figure 4-14 shows the code-segment descriptor format (gray shading indicates the bit is reserved). All software tasks require that a segment selector, referencing a valid code-segment descriptor, is loaded into the CS register. Code segments establish the processor operating mode and execution privilege-level. The segments generally contain only instructions and are execute-only, or execute and read-only. Software cannot write into a segment whose selector references a code-segment descriptor.

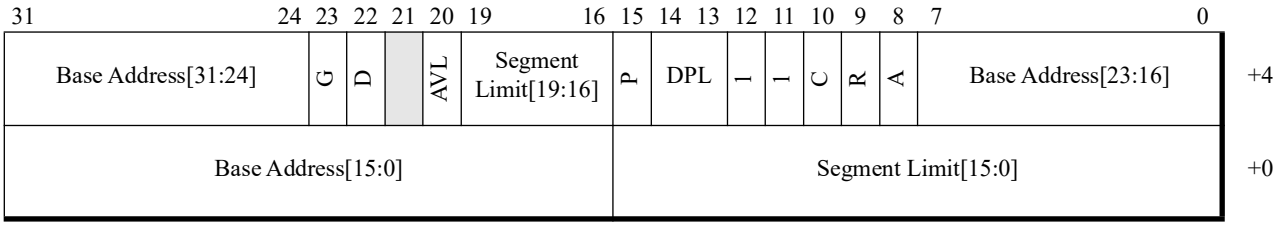


Figure 4-14. Code-Segment Descriptor—Legacy Mode

Code-segment descriptors have the S bit set to 1, identifying the segments as user segments. Type-field bit 11 differentiates code-segment descriptors (bit 11 set to 1) from data-segment descriptors (bit 11 cleared to 0). The remaining type-field bits (10:8) define the access characteristics for the code-segment, as follows:

Conforming (C) Bit. Bit 10 of the upper doubleword. Setting this bit to 1 identifies the code segment as *conforming*. When control is transferred to a higher-privilege conforming code-segment (C=1) from a lower-privilege code segment, the processor CPL does not change. Transfers to non-conforming code-segments (C = 0) with a higher privilege-level than the CPL can occur only through gate descriptors. See “Control-Transfer Privilege Checks” on page 111 for more information on conforming and non-conforming code-segments.

Readable (R) Bit. Bit 9 of the upper doubleword. Setting this bit to 1 indicates the code segment is both executable and readable as data. When this bit is cleared to 0, the code segment is executable, but attempts to read data from the code segment cause a general-protection exception (#GP) to occur.

Accessed (A) Bit. Bit 8 of the upper doubleword. The accessed bit is set to 1 by the processor when the descriptor is copied from the GDT or LDT into the CS register. This bit is only cleared by software.

Table 4-3 on page 94 summarizes the code-segment type-field encodings.

[AMD Public Use]

Table 4-3. Code-Segment Descriptor Types

Hex Value	Type Field				Description
	Bit 11 (Code/Data)	Bit 10	Bit 9	Bit 8	
		Conforming (C)	Readable (R)	Accessed (A)	
8	1	0	0	0	Execute-Only
9		0	0	1	Execute-Only — Accessed
A		0	1	0	Execute/Readable
B		0	1	1	Execute/Readable — Accessed
C		1	0	0	Conforming, Execute-Only
D		1	0	1	Conforming, Execute-Only — Accessed
E		1	1	0	Conforming, Execute/Readable
F		1	1	1	Conforming, Execute/Readable — Accessed

Code-Segment Default-Operand Size (D) Bit. Bit 22 of byte +4. In code-segment descriptors, the D bit selects the default operand size and address sizes. In legacy mode, when D=0 the default operand size and address size is 16 bits and when D=1 the default operand size and address size is 32 bits. Instruction prefixes can be used to override the operand size or address size, or both.

4.7.3 Data-Segment Descriptors

Figure 4-15 shows the data-segment descriptor format. Data segments contain non-executable information and can be accessed as read-only or read/write. They are referenced using the DS, ES, FS, GS, or SS data-segment registers. The DS data-segment register holds the segment selector for the default data segment. The ES, FS and GS data-segment registers hold segment selectors for additional data segments usable by the current software task.

The stack segment is a special form of data-segment register. It is referenced using the SS segment register and must be read/write. When loading the SS register, the processor requires that the selector reference a valid, writable data-segment descriptor.

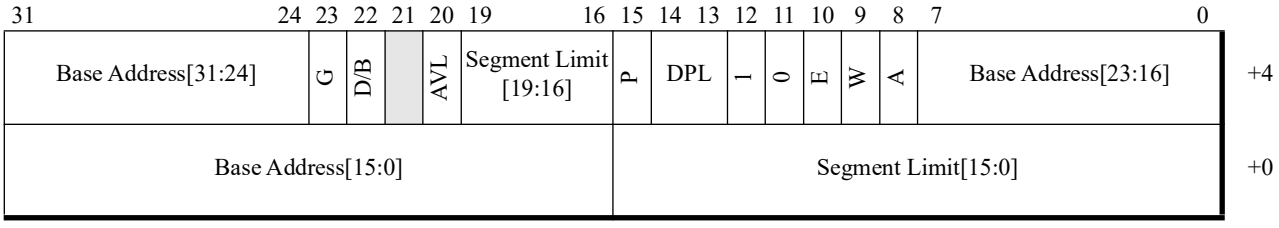


Figure 4-15. Data-Segment Descriptor—Legacy Mode

[AMD Public Use]

Data-segment descriptors have the S bit set to 1, identifying them as user segments. Type-field bit 11 differentiates data-segment descriptors (bit 11 cleared to 0) from code-segment descriptors (bit 11 set to 1). The remaining type-field bits (10:8) define the data-segment access characteristics, as follows:

Expand-Down (E) Bit. Bit 10 of the upper doubleword. Setting this bit to 1 identifies the data segment as *expand-down*. In expand-down segments, the segment limit defines the *lower* segment boundary while the base is the upper boundary. Valid segment offsets in expand-down segments lie in the byte range limit+1 to FFFFh or FFFF_FFFFh, depending on the value of the data segment default operand size (D/B) bit.

Expand-down segments are useful for stacks, which grow in the downward direction as elements are pushed onto the stack. The stack pointer, ESP, is *decremented* by an amount equal to the operand size as a result of executing a PUSH instruction.

Clearing the E bit to 0 identifies the data segment as expand-up. Valid segment offsets in expand-up segments lie in the byte range 0 to segment limit.

Writable (W) Bit. Bit 9 of the upper doubleword. Setting this bit to 1 identifies the data segment as read/write. When this bit is cleared to 0, the segment is read-only. A general-protection exception (#GP) occurs if software attempts to write into a data segment when W=0.

Accessed (A) Bit. Bit 8 of the upper doubleword. The accessed bit is set to 1 by the processor when the descriptor is copied from the GDT or LDT into one of the data-segment registers or the stack-segment register. This bit is only cleared by software.

Table 4-4 summarizes the data-segment type-field encodings.

Table 4-4. Data-Segment Descriptor Types

Hex Value	Type Field				Description
	Bit 11 (Code/Data)	Bit 10	Bit 9	Bit 8	
		Expand-Down (E)	Writable (W)	Accessed (A)	
0	0	0	0	0	Read-Only
1		0	0	1	Read-Only — Accessed
2		0	1	0	Read/Write
3		0	1	1	Read/Write — Accessed
4		1	0	0	Expand-down, Read-Only
5		1	0	1	Expand-down, Read-Only — Accessed
6		1	1	0	Expand-down, Read/Write
7		1	1	1	Expand-down, Read/Write — Accessed

[AMD Public Use]

Data-Segment Default Operand Size (D/B) Bit. Bit 22 of the upper doubleword. For expand-down data segments (E=1), setting D=1 sets the upper bound of the segment at 0_FFFF_FFFFh. Clearing D=0 sets the upper bound of the segment at 0_FFFFh.

In the case where a data segment is referenced by the stack selector (SS), the D bit is referred to as the B bit. For stack segments, the B bit sets the default stack size. Setting B=1 establishes a 32-bit stack referenced by the 32-bit ESP register. Clearing B=0 establishes a 16-bit stack referenced by the 16-bit SP register.

4.7.4 System Descriptors

There are two general types of system descriptors: system-segment descriptors and gate descriptors. System-segment descriptors are used to describe the LDT and TSS segments. Gate descriptors do not describe segments, but instead hold pointers to code-segment descriptors. Gate descriptors are used for protected-mode control transfers between less-privileged and more-privileged software.

System-segment descriptors have the S bit cleared to 0. The type field is used to differentiate the various LDT, TSS, and gate descriptors from one another. Table 4-5 summarizes the system-segment type-field encodings.

Table 4-5. System-Segment Descriptor Types (S=0)—Legacy Mode

Hex Value	Type Field (Bits 11:8)	Description
0	0000	Reserved (Illegal)
1	0001	Available 16-bit TSS
2	0010	LDT
3	0011	Busy 16-bit TSS
4	0100	16-bit Call Gate
5	0101	Task Gate
6	0110	16-bit Interrupt Gate
7	0111	16-bit Trap Gate
8	1000	Reserved (Illegal)
9	1001	Available 32-bit TSS
A	1010	Reserved (Illegal)
B	1011	Busy 32-bit TSS
C	1100	32-bit Call Gate
D	1101	Reserved (Illegal)
E	1110	32-bit Interrupt Gate
F	1111	32-bit Trap Gate

[AMD Public Use]

Figure 4-16 shows the legacy-mode system-segment descriptor format used for referencing LDT and TSS segments (gray shading indicates the bit is reserved). This format is also used in compatibility mode. The system-segments are used as follows:

- The LDT typically holds segment descriptors belonging to a single task (see “Local Descriptor Table” on page 86).
- The TSS is a data structure for holding processor-state information. Processor state is saved in a TSS when a task is suspended, and state is restored from the TSS when a task is restarted. System software must create at least one TSS referenced by the task register, TR. See “Legacy Task-State Segment” on page 374 for more information on the TSS.

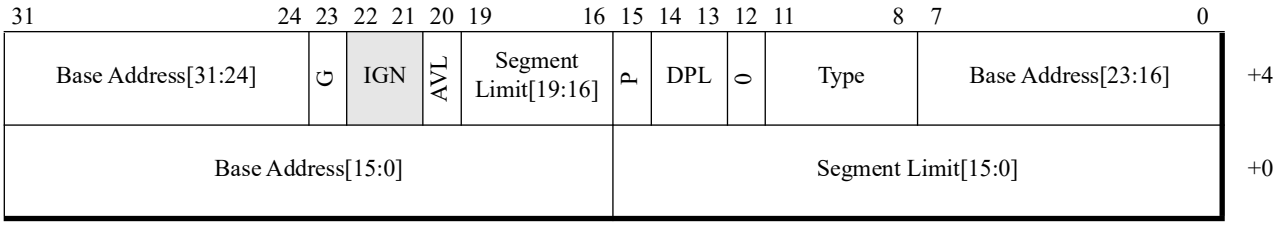


Figure 4-16. LDT and TSS Descriptor—Legacy/Compatibility Modes

4.7.5 Gate Descriptors

Gate descriptors hold pointers to code segments and are used to control access between code segments with different privilege levels. There are four types of gate descriptors:

- *Call Gates*—These gates (Figure 4-17 on page 98) are located in the GDT or LDT and are used to control access between code segments in the same task or in different tasks. See “Control Transfers Through Call Gates” on page 115 for information on how call gates are used to control access between code segments operating in the same task. The format of a call-gate descriptor is shown in Figure 4-17 on page 98.
- *Interrupt Gates* and *Trap Gates*—These gates (Figure 4-18 on page 98) are located in the IDT and are used to control access to interrupt-service routines. “Legacy Protected-Mode Interrupt Control Transfers” on page 272 contains information on using these gates for interrupt-control transfers. The format of interrupt-gate and trap-gate descriptors is shown in Figure 4-17 on page 98.
- *Task Gates*—These gates (Figure 4-19 on page 98) are used to control access between different tasks. They are also used to transfer control to interrupt-service routines if those routines are themselves a separate task. See “Task-Management Resources” on page 369 for more information on task gates and their use.

[AMD Public Use]

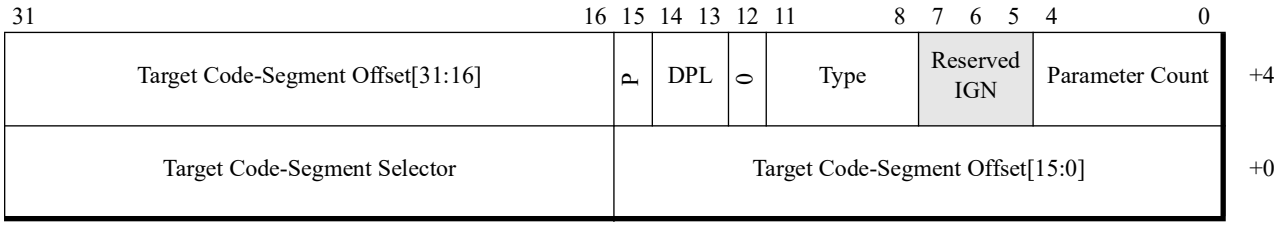


Figure 4-17. Call-Gate Descriptor—Legacy Mode

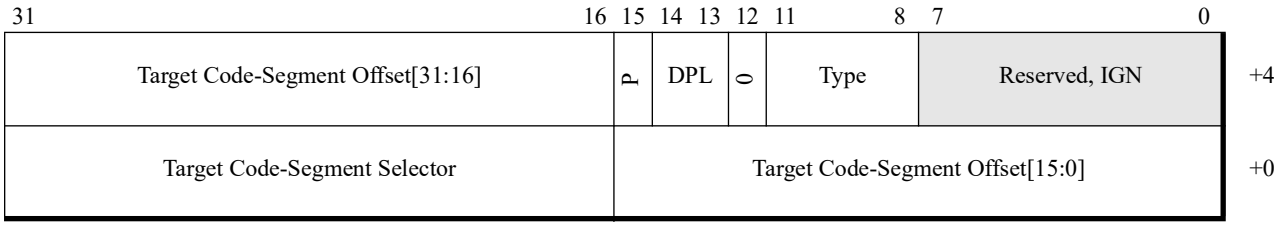


Figure 4-18. Interrupt-Gate and Trap-Gate Descriptors—Legacy Mode

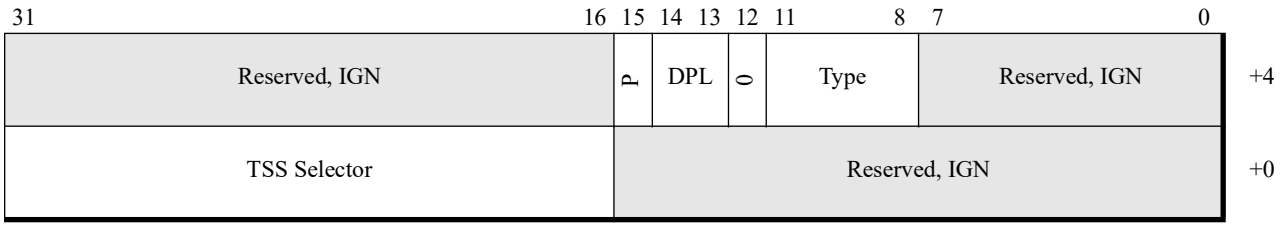


Figure 4-19. Task-Gate Descriptor—Legacy Mode

There are several differences between the gate-descriptor format and the system-segment descriptor format. These differences are described as follows, from least-significant to most-significant bit positions:

Target Code-Segment Offset. The 32-bit segment offset is formed by concatenating bits 31:16 of byte +4 with bits 15:0 of byte +0. The segment-offset field specifies the target-procedure entry point (offset) into the segment. This field is loaded into the EIP register as a result of a control transfer using the gate descriptor.

Target Code-Segment Selector. Bits 31:16 of byte +0. The segment-selector field identifies the target-procedure segment descriptor, located in either the GDT or LDT. The segment selector is loaded into the CS segment register as a result of a control transfer using the gate descriptor.

TSS Selector. Bits 31:16 of byte +0 (task gates only). This field identifies the target-task TSS descriptor, located in any of the three descriptor tables (GDT, LDT, and IDT).

[AMD Public Use]

Parameter Count (Call Gates Only). Bits 4:0 of byte +4. Legacy-mode call-gate descriptors contain a 5-bit *parameter-count* field. This field specifies the number of parameters to be copied from the currently-executing program stack to the target program stack during an automatic stack switch. Automatic stack switches are performed by the processor during a control transfer through a call gate to a greater privilege-level. The parameter size depends on the call-gate size as specified in the type field. 32-bit call gates copy 4-byte parameters, and 16-bit call gates copy 2-byte parameters. See “Stack Switching” on page 119 for more information on call-gate parameter copying.

4.8 Long-Mode Segment Descriptors

The interpretation of descriptor fields is changed in long mode, and in some cases the format is expanded. The changes depend on the operating mode (compatibility mode or 64-bit mode) and on the descriptor type. The following sections describe the changes.

4.8.1 Code-Segment Descriptors

Code segments continue to exist in long mode. Code segments and their associated descriptors and selectors are needed to establish the processor operating mode as well as execution privilege-level. The new L attribute specifies whether the processor is running in compatibility mode or 64-bit mode (see “Long (L) Attribute Bit” on page 100). Figure 4-20 shows the long-mode code-segment descriptor format. In compatibility mode, the code-segment descriptor is interpreted and behaves just as it does in legacy mode as described in “Code-Segment Descriptors” on page 93.

In Figure 4-20, gray shading indicates the code-segment descriptor fields that are *ignored in 64-bit mode* when the descriptor is used during a memory reference. However, the fields are loaded whenever the segment register is loaded in 64-bit mode.

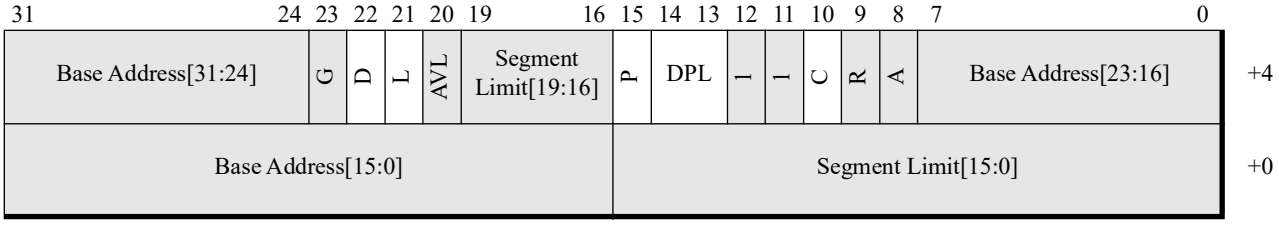


Figure 4-20. Code-Segment Descriptor—Long Mode

Fields Ignored in 64-Bit Mode. Segmentation is disabled in 64-bit mode, and code segments span all of virtual memory. In this mode, code-segment base addresses are ignored. For the purpose of virtual-address calculations, the base address is treated as if it has a value of zero.

Segment-limit checking is not performed, and both the segment-limit field and granularity (G) bit are ignored. Instead, the virtual address is checked to see if it is in canonical-address form.

The readable (R) and accessed (A) attributes in the type field are also ignored.

[AMD Public Use]

Long (L) Attribute Bit. Bit 21 of byte +4. Long mode introduces a new attribute, the *long* (L) bit, in code-segment descriptors. This bit specifies that the processor is running in 64-bit mode (L=1) or compatibility mode (L=0). When the processor is running in legacy mode, this bit is reserved.

Compatibility mode maintains binary compatibility with legacy 16-bit and 32-bit applications. Compatibility mode is selected on a code-segment basis, and it allows legacy applications to coexist under the same 64-bit system software along with 64-bit applications running in 64-bit mode. System software running in long mode can execute existing 16-bit and 32-bit applications by clearing the L bit of the code-segment descriptor to 0.

When L=0, the legacy meaning of the code-segment D bit (see “Code-Segment Default-Operand Size (D) Bit” on page 94)—and the address-size and operand-size prefixes—are observed. Segmentation is enabled when L=0. From an application viewpoint, the processor is in a legacy 16-bit or 32-bit operating environment (depending on the D bit), even though long mode is activated.

If the processor is running in 64-bit mode (L=1), the only valid setting of the D bit is 0. This setting produces a default operand size of 32 bits and a default address size of 64 bits. The combination L=1 and D=1 is reserved for future use.

“Instruction Prefixes” in Volume 3 describes the effect of the code-segment L and D bits on default operand and address sizes when long mode is activated. These default sizes can be overridden with operand size, address size, and REX prefixes.

4.8.2 Data-Segment Descriptors

Data segments continue to exist in long mode. Figure 4-21 shows the long-mode data-segment descriptor format. In compatibility mode, data-segment descriptors are interpreted and behave just as they do in legacy mode.

In Figure 4-21, gray shading indicates the fields that are *ignored in 64-bit mode* when the descriptor is used during a memory reference. However, the fields are loaded whenever the segment register is loaded in 64-bit mode.

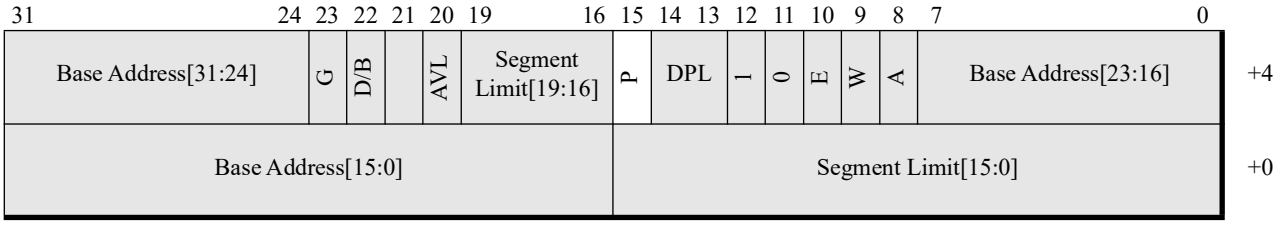


Figure 4-21. Data-Segment Descriptor—Long Mode

Fields Ignored in 64-Bit Mode. Segmentation is disabled in 64-bit mode. The interpretation of the segment-base address depends on the segment register used:

[AMD Public Use]

- In data-segment descriptors referenced by the DS, ES and SS segment registers, the base-address field is ignored. For the purpose of virtual-address calculations, the base address is treated as if it has a value of zero.
- Data segments referenced by the FS and GS segment registers receive special treatment in 64-bit mode. For these segments, the base address field is not ignored, and a non-zero value can be used in virtual-address calculations. A 64-bit segment-base address can be specified using model-specific registers. See “FS and GS Registers in 64-Bit Mode” on page 82 for more information.

Segment-limit checking is not performed on any data segments in 64-bit mode, and both the segment-limit field and granularity (G) bit are ignored. The D/B bit is unused in 64-bit mode.

The expand-down (E), writable (W), and accessed (A) type-field attributes are ignored.

A data-segment-descriptor DPL field is ignored in 64-bit mode, and segment-privilege checks are not performed on data segments. System software can use the page-protection mechanisms to isolate and protect data from unauthorized access.

4.8.3 System Descriptors

In long mode, the allowable system-descriptor types encoded by the type field are changed. Some descriptor types are modified, and others are illegal. The changes are summarized in Table 4-6. An attempt to use an illegal descriptor type causes a general-protection exception (#GP).

Table 4-6. System-Segment Descriptor Types—Long Mode

Hex Value	Type Field				Description
	Bit 11	Bit 10	Bit 9	Bit 8	
0	0	0	0	0	Reserved (Illegal)
1	0	0	0	1	
2	0	0	1	0	64-bit LDT[1]
3	0	0	1	1	Reserved (Illegal)
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	
8	1	0	0	0	
9	1	0	0	1	Available 64-bit TSS
A	1	0	1	0	Reserved (Illegal)
B	1	0	1	1	Busy 64-bit TSS
C	1	1	0	0	64-bit Call Gate

Note(s):
1. In 64-bit mode only. In compatibility mode, the type specifies a 32-bit LDT.

[AMD Public Use]

Table 4-6. System-Segment Descriptor Types—Long Mode (continued)

Hex Value	Type Field				Description
	Bit 11	Bit 10	Bit 9	Bit 8	
D	1	1	0	1	Reserved (Illegal)
E	1	1	1	0	64-bit Interrupt Gate
F	1	1	1	1	64-bit Trap Gate

Note(s):
1. In 64-bit mode only. In compatibility mode, the type specifies a 32-bit LDT.

In long mode, the modified system-segment descriptor types are:

- The 32-bit LDT (02h), which is redefined as the 64-bit LDT.
- The available 32-bit TSS (09h), which is redefined as the available 64-bit TSS.
- The busy 32-bit TSS (0Bh), which is redefined as the busy 64-bit TSS.

In 64-bit mode, the LDT and TSS system-segment descriptors are expanded by 64 bits, as shown in Figure 4-22. In this figure, gray shading indicates the fields that are *ignored in 64-bit mode*. Expanding the descriptors allows them to hold 64-bit base addresses, so their segments can be located anywhere in the virtual-address space. The expanded descriptor can be loaded into the corresponding descriptor-table register (LDTR or TR) only from 64-bit mode. In compatibility mode, the legacy system-segment descriptor format, shown in Figure 4-16 on page 97, is used. See “LLDT and LTR Instructions” on page 182 for more information.

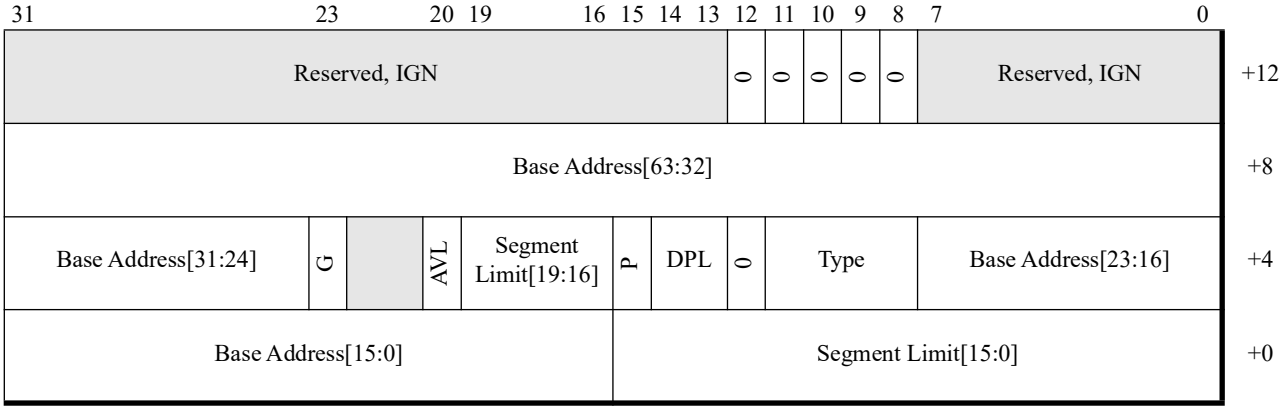


Figure 4-22. System-Segment Descriptor—64-Bit Mode

The 64-bit system-segment base address must be in canonical form. Otherwise, a general-protection exception occurs with a selector error-code, #GP(selector), when the system segment is loaded. System-segment limit values are checked by the processor in both 64-bit and compatibility modes, under the control of the granularity (G) bit.

Figure 4-22 shows that bits 12:8 of doubleword +12 must be cleared to 0. These bits correspond to the S and Type fields in a legacy descriptor. Clearing these bits to 0 corresponds to an illegal type in legacy

[AMD Public Use]

mode and causes a #GP if an attempt is made to access the upper half of a 64-bit mode system-segment descriptor as a legacy descriptor or as the lower half of a 64-bit mode system-segment descriptor.

4.8.4 Gate Descriptors

As shown in Table 4-6 on page 101, the allowable gate-descriptor types are changed in long mode. Some gate-descriptor types are modified and others are illegal. The modified gate-descriptor types in long mode are:

- The 32-bit call gate (0Ch), which is redefined as the 64-bit call gate.
- The 32-bit interrupt gate (0Eh), which is redefined as the 64-bit interrupt gate.
- The 32-bit trap gate (0Fh), which is redefined as the 64-bit trap gate.

In long mode, several gate-descriptor types are illegal. An attempt to use these gates causes a general-protection exception (#GP) to occur. The illegal gate types are:

- The 16-bit call gate (04h).
- The task gate (05h).
- The 16-bit interrupt gate (06h).
- The 16-bit trap gate (07h).

In long mode, gate descriptors are expanded by 64 bits, allowing them to hold 64-bit offsets. The 64-bit call-gate descriptor is shown in Figure 4-23 and the 64-bit interrupt gate and trap gate are shown in Figure 4-24 on page 104. In these figures, gray shading indicates the fields that are *ignored in long mode*. The interrupt and trap gates contain an additional field, the IST, that is not present in the call gate—see “IST Field (Interrupt and Trap Gates)” on page 104.

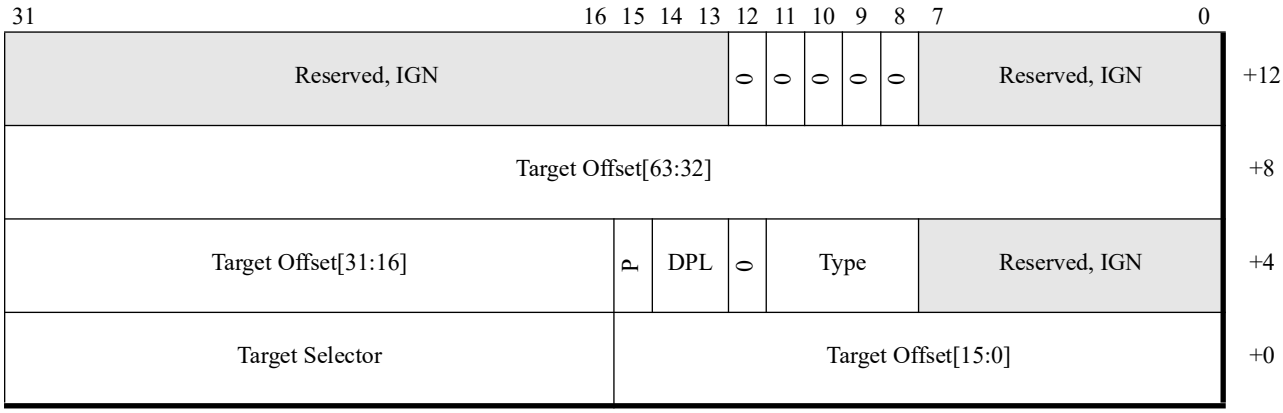


Figure 4-23. Call-Gate Descriptor—Long Mode

[AMD Public Use]

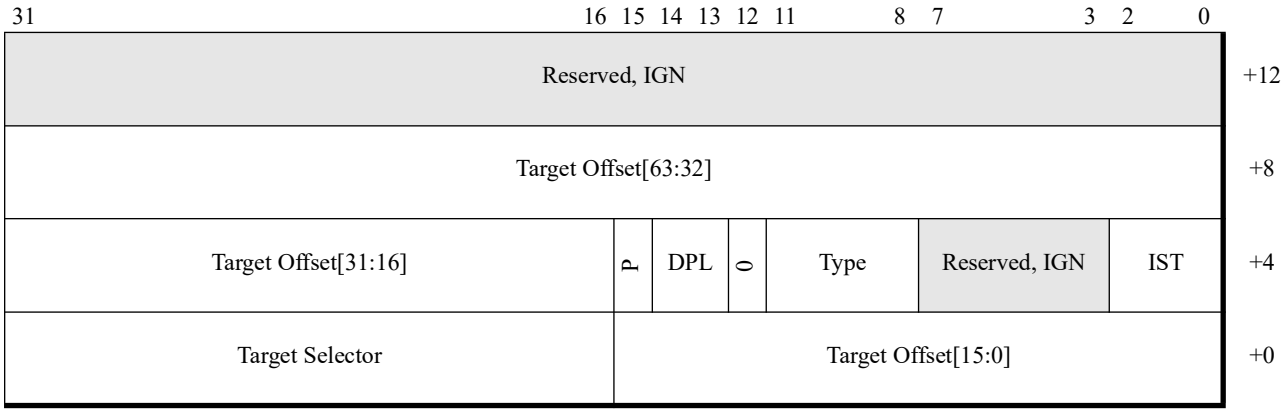


Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode

The target code segment referenced by a long-mode gate descriptor must be a 64-bit code segment (CS.L=1, CS.D=0). If the target is not a 64-bit code segment, a general-protection exception, #GP(error), occurs. The error code reported depends on the gate type:

- Call gates report the target code-segment selector as the error code.
- Interrupt and trap gates report the interrupt vector number as the error code.

A general-protection exception, #GP(0), occurs if software attempts to reference a long-mode gate descriptor with a target-segment offset that is not in canonical form.

It is possible for software to store legacy and long mode gate descriptors in the same descriptor table. Figure 4-23 on page 103 shows that bits 12:8 of byte +12 in a long-mode call gate must be cleared to 0. These bits correspond to the S and Type fields in a legacy call gate. Clearing these bits to 0 corresponds to an illegal type in legacy mode and causes a #GP if an attempt is made to access the upper half of a 64-bit mode call-gate descriptor as a legacy call-gate descriptor.

It is not necessary to clear these same bits in a long-mode interrupt gate or trap gate. In long mode, the interrupt-descriptor table (IDT) must contain 64-bit interrupt gates or trap gates. The processor automatically indexes the IDT by scaling the interrupt vector by 16. This makes it impossible to access the upper half of a long-mode interrupt gate, or trap gate, as a legacy gate when the processor is running in long mode.

IST Field (Interrupt and Trap Gates). Bits 2:0 of byte +4. Long-mode interrupt gate and trap gate descriptors contain a new, 3-bit interrupt-stack-table (IST) field not present in legacy gate descriptors. The IST field is used as an index into the IST portion of a long-mode TSS. If the IST field is not 0, the index references an IST pointer in the TSS, which the processor loads into the RSP register when an interrupt occurs. If the IST index is 0, the processor uses the legacy stack-switching mechanism (with some modifications) when an interrupt occurs. See “Interrupt-Stack Table” on page 287 for more information.

[AMD Public Use]

Count Field (Call Gates). The count field found in legacy call-gate descriptors is not supported in long-mode call gates. In long mode, the field is reserved and should be cleared to zero.

4.8.5 Long Mode Descriptor Summary

System descriptors and gate descriptors are expanded by 64 bits to handle 64-bit base addresses in long mode or 64-bit mode. The mode in which the expansion occurs depends on the purpose served by the descriptor, as follows:

- *Expansion Only In 64-Bit Mode*—The system descriptors and pseudo-descriptors that are loaded into the GDTR, IDTR, LDTR, and TR registers are expanded only in 64-bit mode. They are not expanded in compatibility mode.
- *Expansion In Long Mode*—Gate descriptors (call gates, interrupt gates, and trap gates) are expanded in long mode (both 64-bit mode and compatibility mode). Task gates and 16-bit gate descriptors are illegal in long mode.

The AMD64 architecture redefines several of the descriptor-entry fields in support of long mode. The specific change depends on whether the processor is in 64-bit mode or compatibility mode. Table 4-7 summarizes the changes in the descriptor entry field when the descriptor entry is loaded into a segment register (as opposed to when the segment register is subsequently used to access memory).

Table 4-7. Descriptor-Entry Field Changes in Long Mode

<table>
<tr><th rowspan="2">Descriptor Field</th><th rowspan="2">Descriptor Type</th><th colspan="2">Long Mode</th></tr>
<tr><th>Compatibility Mode</th><th>64-Bit Mode</th></tr>
<tr><td rowspan="3">Limit</td><td>Code</td><td rowspan="3">Same as legacy x86</td><td rowspan="3">Same as legacy x86</td></tr>
<tr><td>Data</td></tr>
<tr><td>System</td></tr>
<tr><td>Offset</td><td>Gate</td><td>Expanded to 64 bits</td><td>Expanded to 64 bits</td></tr>
<tr><td rowspan="3">Base</td><td>Code</td><td rowspan="3">Same as legacy x86</td><td rowspan="4">Same as legacy x86</td></tr>
<tr><td>Data</td></tr>
<tr><td>System</td></tr>
<tr><td>Selector</td><td>Gate</td><td>Same as legacy x86</td></tr>
<tr><td>IST[1]</td><td>Gate</td><td colspan="2">Interrupt and trap gates only. (New for long mode.)</td></tr>
<tr><td rowspan="4">S and Type</td><td>Code</td><td rowspan="2">Same as legacy x86</td><td rowspan="2">Same as legacy x86</td></tr>
<tr><td>Data</td></tr>
<tr><td>System</td><td colspan="2">Types 02h, 09h, and 0Bh redefined
Types 01h and 03h are illegal</td></tr>
<tr><td>Gate</td><td colspan="2">Types 0Ch, 0Eh, and 0Fh redefined
Types 04h–07h are illegal</td></tr>
<tr><td colspan="4">Note(s):
1. Not available (reserved) in legacy mode.</td></tr>
</table>

[AMD Public Use]

Table 4-7. Descriptor-Entry Field Changes in Long Mode (continued)

Descriptor Field	Descriptor Type	Long Mode	
		Compatibility Mode	64-Bit Mode
DPL	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
	Gate		
Present	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
	Gate		
Default Size	Code	Same as legacy x86	D=0 Indicates 64-bit address, 32-bit data D=1 Reserved
	Data		Same as legacy x86
Long[1]	Code	Specifies compatibility mode	Specifies 64-bit mode
Granularity	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
Available	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		

Note(s):
1. Not available (reserved) in legacy mode.

4.9 Segment-Protection Overview

The AMD64 architecture is designed to fully support the legacy segment-protection mechanism. The segment-protection mechanism provides system software with the ability to restrict program access into other software routines and data.

Segment-level protection remains enabled in compatibility mode. 64-bit mode eliminates most type checking, and limit checking is not performed, except on accesses to system-descriptor tables.

The preferred method of implementing memory protection in a long-mode operating system is to rely on the page-protection mechanism as described in “Page-Protection Checks” on page 163. System software still needs to create basic segment-protection data structures for 64-bit mode. These structures are simplified, however, by the use of the flat-memory model in 64-bit mode, and the limited segmentation checks performed when executing in 64-bit mode.

[AMD Public Use]

4.9.1 Privilege-Level Concept

Segment protection is used to isolate and protect programs and data from each other. The segment-protection mechanism supports four privilege levels in protected mode. The privilege levels are designated with a numerical value from 0 to 3, with 0 being the most privileged and 3 being the least privileged. System software typically assigns the privilege levels in the following manner:

- *Privilege-level 0 (most privilege)*—This level is used by critical system-software components that require direct access to, and control over, all processor and system resources. This can include platform firmware, memory-management functions, and interrupt handlers.
- *Privilege-levels 1 and 2 (moderate privilege)*—These levels are used by less-critical system-software services that can access and control a limited scope of processor and system resources. Software running at these privilege levels might include some device drivers and library routines. These software routines can call more-privileged system-software services to perform functions such as memory garbage-collection and file allocation.
- *Privilege-level 3 (least privilege)*—This level is used by application software. Software running at privilege-level 3 is normally prevented from directly accessing most processor and system resources. Instead, applications request access to the protected processor and system resources by calling more-privileged service routines to perform the accesses.

Figure 4-25 shows the relationship of the four privilege levels to each other.

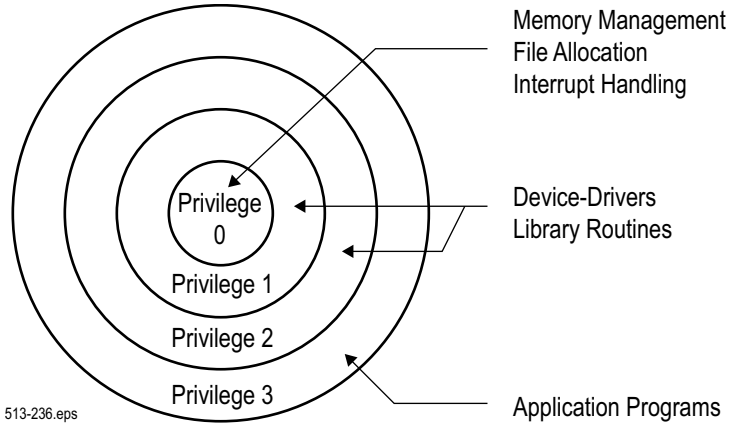


Figure 4-25. Privilege-Level Relationships

4.9.2 Privilege-Level Types

There are three types of privilege levels the processor uses to control access to segments. These are CPL, DPL, and RPL.

Current Privilege-Level. The current privilege-level (CPL) is the privilege level at which the processor is currently executing. The CPL is stored in an internal processor register that is invisible to

[AMD Public Use]

software. Software changes the CPL by performing a control transfer to a different code segment with a new privilege level.

Descriptor Privilege-Level. The descriptor privilege-level (DPL) is the privilege level that system software assigns to individual segments. The DPL is used in privilege checks to determine whether software can access the segment referenced by the descriptor. In the case of gate descriptors, the DPL determines whether software can access the descriptor reference by the gate. The DPL is stored in the segment (or gate) descriptor.

Requestor Privilege-Level. The requestor privilege-level (RPL) reflects the privilege level of the program that created the selector. The RPL can be used to let a called program know the privilege level of the program that initiated the call. The RPL is stored in the selector used to reference the segment (or gate) descriptor.

The following sections describe how the CPL, DPL, and RPL are used by the processor in performing privilege checks on data accesses and control transfers. Failure to pass a protection check generally causes an exception to occur.

4.10 Data-Access Privilege Checks

4.10.1 Accessing Data Segments

Before loading a data-segment register (DS, ES, FS, or GS) with a segment selector, the processor checks the privilege levels as follows to see if access is allowed:

1. The processor compares the CPL with the RPL in the data-segment selector and determines the effective privilege level for the data access. The processor sets the effective privilege level to the lowest privilege (numerically-higher value) indicated by the comparison.
2. The processor compares the effective privilege level with the DPL in the descriptor-table entry referenced by the segment selector. If the effective privilege level is greater than or equal to (numerically lower-than or equal-to) the DPL, then the processor loads the segment register with the data-segment selector. The processor automatically loads the corresponding descriptor-table entry into the hidden portion of the segment register.

   If the effective privilege level is lower than (numerically greater-than) the DPL, a general-protection exception (#GP) occurs and the segment register is not loaded.

Figure 4-26 on page 109 shows two examples of data-access privilege checks.

[AMD Public Use]

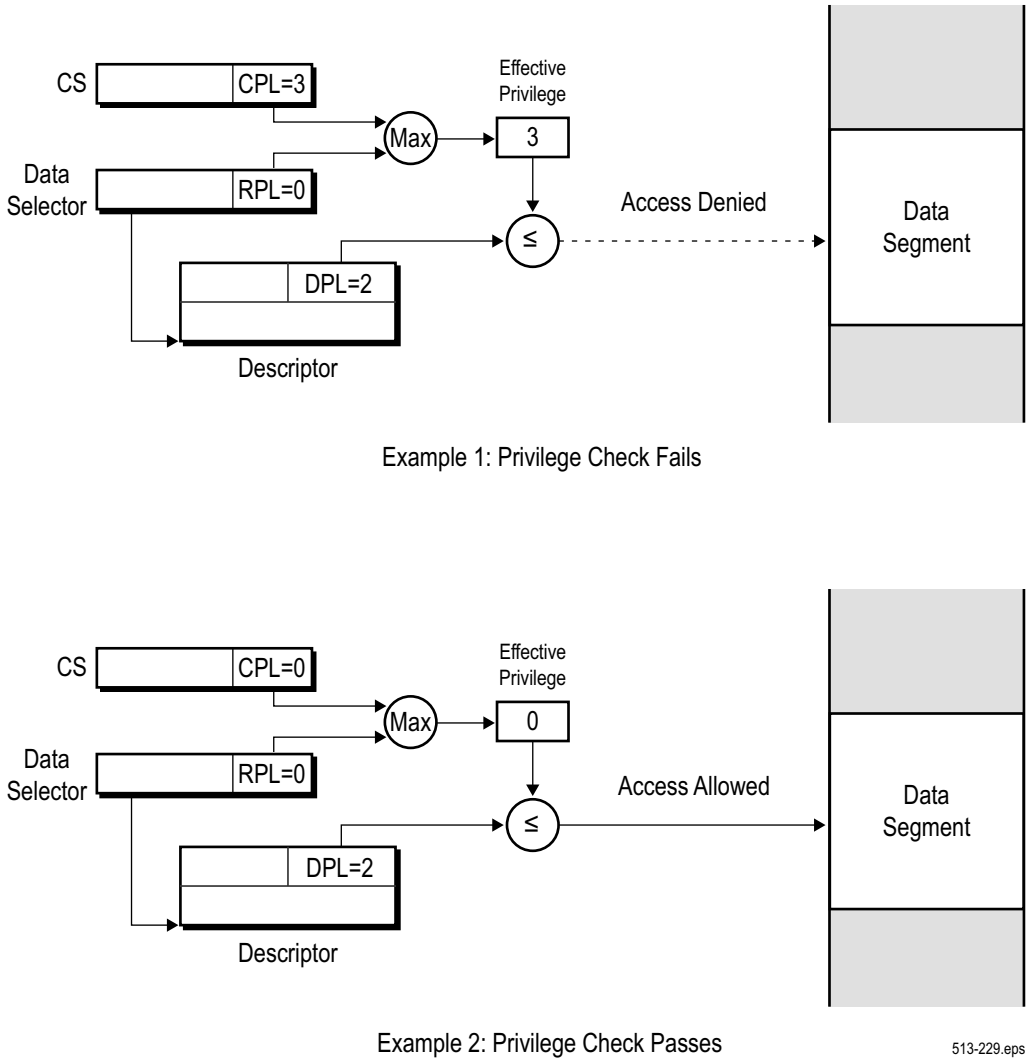


Figure 4-26. Data-Access Privilege-Check Examples

Example 1 in Figure 4-26 shows a failing data-access privilege check. The effective privilege level is 3 because CPL=3. This value is greater than the descriptor DPL, so access to the data segment is denied.

Example 2 in Figure 4-26 shows a passing data-access privilege check. Here, the effective privilege level is 0 because both the CPL and RPL have values of 0. This value is less than the descriptor DPL, so access to the data segment is allowed, and the data-segment register is successfully loaded.

4.10.2 Accessing Stack Segments

Before loading the stack segment register (SS) with a segment selector, the processor checks the privilege levels as follows to see if access is allowed:

[AMD Public Use]

1. The processor checks that the CPL and the stack-selector RPL are *equal*. If they are not equal, a general-protection exception (#GP) occurs and the SS register is not loaded.
2. The processor compares the CPL with the DPL in the descriptor-table entry referenced by the segment selector. The two values *must be equal*. If they are not equal, a #GP occurs and the SS register is not loaded.

Figure 4-27 shows two examples of stack-access privilege checks. In Example 1 the CPL, stack-selector RPL, and stack segment-descriptor DPL are all equal, so access to the stack segment using the SS register is allowed. In Example 2, the stack-selector RPL and stack segment-descriptor DPL are both equal. However, the CPL is not equal to the stack segment-descriptor DPL, and access to the stack segment through the SS register is denied.

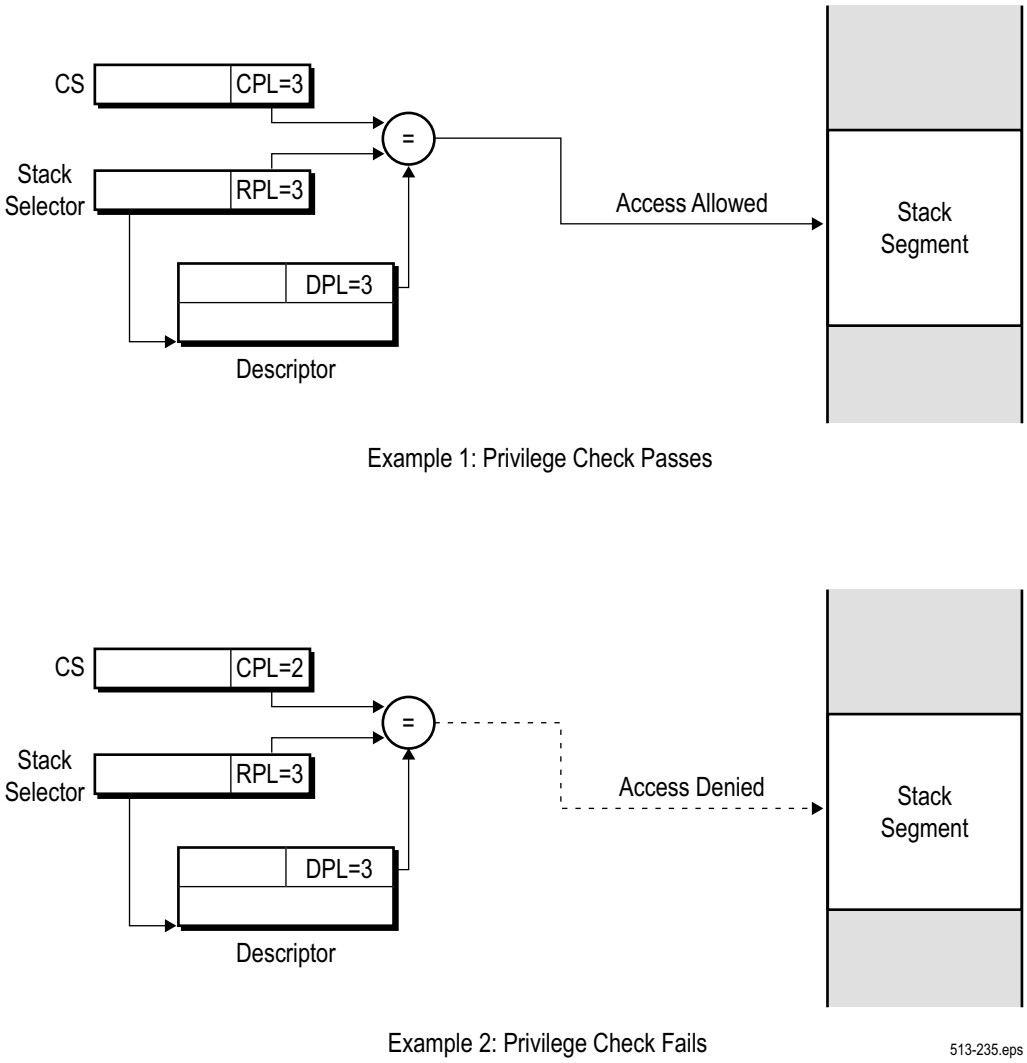


Figure 4-27. Stack-Access Privilege-Check Examples

[AMD Public Use]

4.11 Control-Transfer Privilege Checks

Control transfers between code segments (also called *far control transfers*) cause the processor to perform privilege checks to determine whether the source program is allowed to transfer control to the target program. If the privilege checks pass, access to the target code-segment is granted. When access is granted, the target code-segment selector is loaded into the CS register. The rIP register is updated with the target CS offset taken from either the far-pointer operand or the gate descriptor. Privilege checks are not performed during *near control transfers* because such transfers do not change segments.

The following mechanisms can be used by software to perform far control transfers:

- System-software control transfers using the *system-call* and *system-return* instructions. See "SYSCALL and SYSRET" on page 176 and "SYSENTER and SYSEXIT (Legacy Mode Only)" on page 178 for more information on these instructions. SYSCALL and SYSRET are the preferred method of performing control transfers in long mode. *SYSENTER and SYSEXIT are not supported in long mode.*
- Direct control transfers using CALL and JMP instructions. These are discussed in the next section, "Direct Control Transfers."
- Call-gate control transfers using CALL and JMP instructions. These are discussed in "Control Transfers Through Call Gates" on page 115.
- Return control transfers using the RET instruction. These are discussed in "Return Control Transfers" on page 122.
- Interrupts and exceptions, including the INT*n* and IRET instructions. These are discussed in Chapter 8, "Exceptions and Interrupts."
- Task switches initiated by CALL and JMP instructions. Task switches are discussed in Chapter 12, "Task Management." *The hardware task-switch mechanism is not supported in long mode.*

4.11.1 Direct Control Transfers

A *direct control transfer* occurs when software executes a far-CALL or a far-JMP instruction without using a call gate. The privilege checks and type of access allowed as a result of a direct control transfer depends on whether the target code segment is conforming or nonconforming. The code-segment-descriptor conforming (C) bit indicates whether or not the target code-segment is conforming (see "Conforming (C) Bit" on page 93 for more information on the conforming bit).

Privilege levels are not changed as a result of a direct control transfer. Program stacks are not automatically switched by the processor as they are with privilege-changing control transfers through call gates (see "Stack Switching" on page 119 for more information on automatic stack switching during privilege-changing control transfers).

Nonconforming Code Segments. Software can perform a direct control transfer to a nonconforming code segment only if the target code-segment descriptor DPL and the CPL are equal and the RPL is less than or equal to the CPL. Software must use a call gate to transfer control to a

[AMD Public Use]

more-privileged, nonconforming code segment (see “Control Transfers Through Call Gates” on page 115 for more information).

In far calls and jumps, the far pointer (CS:rIP) references the target code-segment descriptor. Before loading the CS register with a nonconforming code-segment selector, the processor checks as follows to see if access is allowed:

1. *DPL = CPL Check*—The processor compares the target code-segment descriptor DPL with the currently executing program CPL. If they are equal, the processor performs the next check. If they are not equal, a general-protection exception (#GP) occurs.
2. *RPL ≤ CPL Check*—The processor compares the target code-segment selector RPL with the currently executing program CPL. If the RPL is less than or equal to the CPL, access is allowed. If the RPL is greater than the CPL, a #GP exception occurs.

If access is allowed, the processor loads the CS and rIP registers with their new values and begins executing from the target location. The CPL is *not changed*—the target-CS selector RPL value is disregarded when the selector is loaded into the CS register.

Figure 4-28 on page 113 shows three examples of privilege checks performed as a result of a far control transfer to a nonconforming code-segment. In Example 1, access is allowed because CPL = DPL and RPL ≤ CPL. In Example 2, access is denied because CPL ≠ DPL. In Example 3, access is denied because RPL > CPL.

[AMD Public Use]

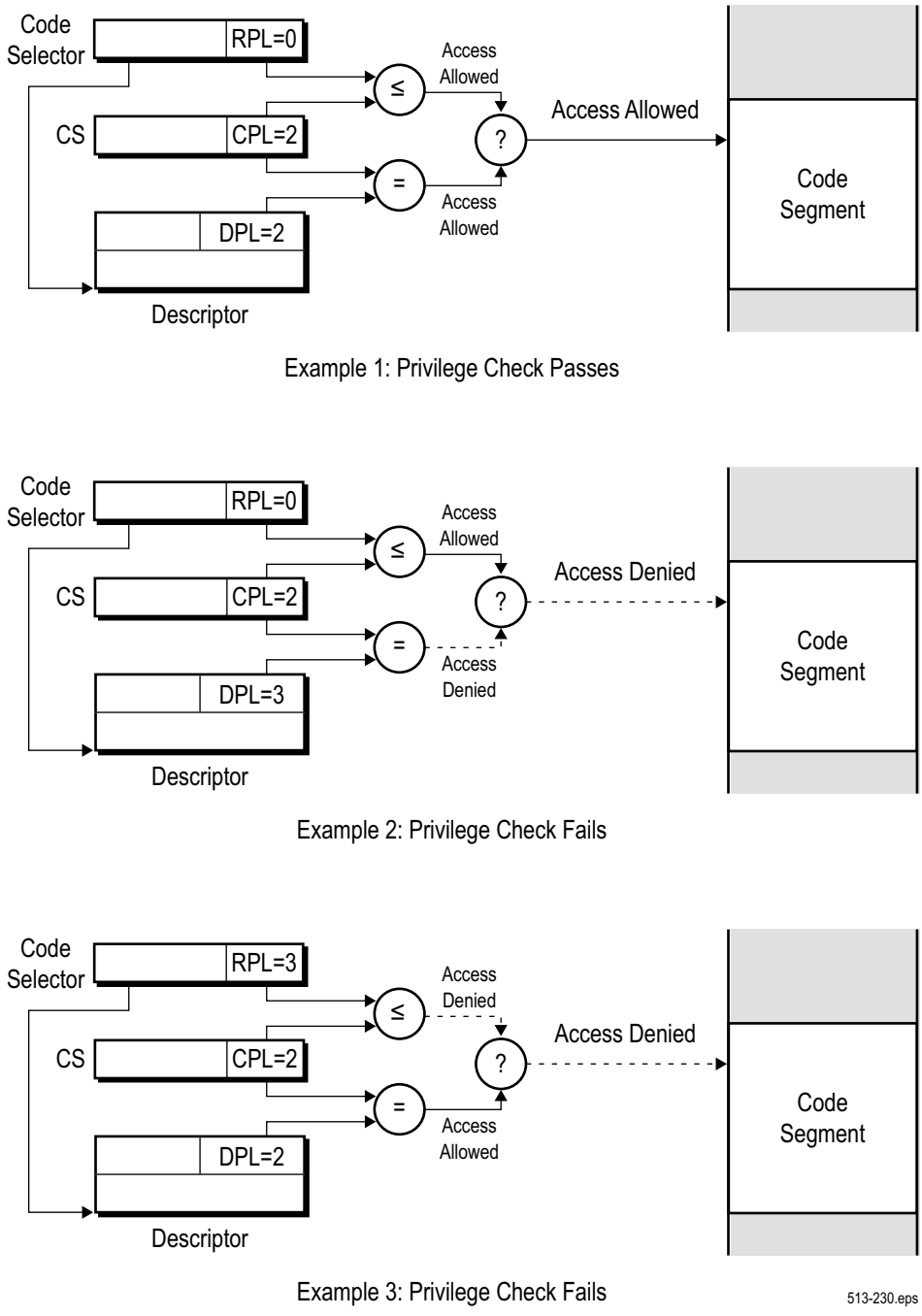


Figure 4-28. Nonconforming Code-Segment Privilege-Check Examples

Conforming Code Segments. On a direct control transfer to a conforming code segment, the target code-segment descriptor DPL can be lower than (at a greater privilege) the CPL. Before loading the

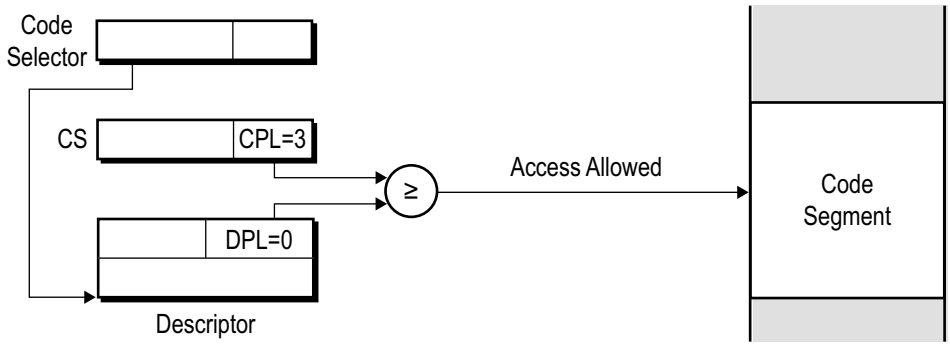
[AMD Public Use]

CS register with a conforming code-segment selector, the processor compares the target code-segment descriptor DPL with the currently-executing program CPL. If the DPL is less than or equal to the CPL, access is allowed. If the DPL is greater than the CPL, a #GP exception occurs.

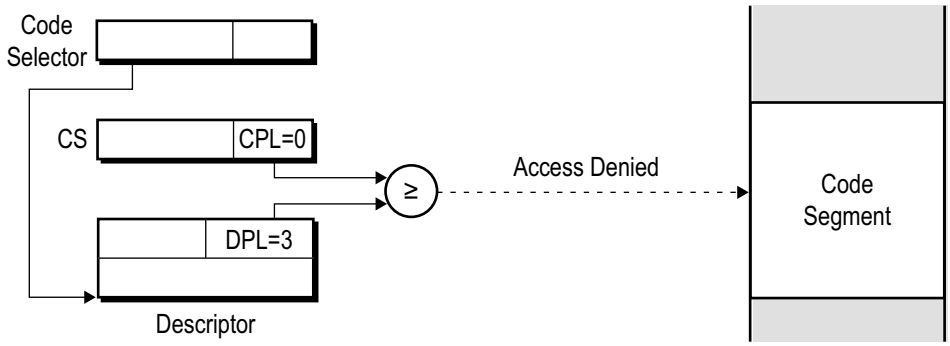
On an access to a conforming code segment, the RPL is ignored and not involved in the privilege check.

When access is allowed, the processor loads the CS and rIP registers with their new values and begins executing from the target location. The CPL is *not changed*—the target CS-descriptor DPL value is disregarded when the selector is loaded into the CS register. The target program runs at the same privilege as the program that called it.

Figure 4-29 shows two examples of privilege checks performed as a result of a direct control transfer to a conforming code segment. In Example 1, access is allowed because the CPL of 3 is greater than the DPL of 0. As the target code selector is loaded into the CS register, the old CPL value of 3 replaces the target-code selector RPL value, and the target program executes with CPL=3. In Example 2, access is denied because CPL < DPL.



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

513-231.eps

Figure 4-29. Conforming Code-Segment Privilege-Check Examples

[AMD Public Use]

4.11.2 Control Transfers Through Call Gates

Control transfers to more-privileged code segments are accomplished through the use of *call gates*. Call gates are a type of descriptor that contain pointers to code-segment descriptors and control access to those descriptors. System software uses call gates to establish protected entry points into system-service routines.

Transfer Mechanism. The pointer operand of a far-CALL or far-JMP instruction consists of two pieces: a code-segment selector (CS) and a code-segment offset (rIP). In a call-gate transfer, the CS selector points to a call-gate descriptor rather than a code-segment descriptor, and the rIP is ignored (but required by the instruction).

Figure 4-30 shows a call-gate control transfer in legacy mode. The call-gate descriptor contains segment-selector and segment-offset fields (see “Gate Descriptors” on page 97 for a detailed description of the call-gate format and fields). These two fields perform the same function as the pointer operand in a direct control-transfer instruction. The segment-selector field points to the target code-segment descriptor, and the segment-offset field is the instruction-pointer offset into the target code-segment. The code-segment base taken from the code-segment descriptor is added to the offset field in the call-gate descriptor to create the target virtual address (linear address).

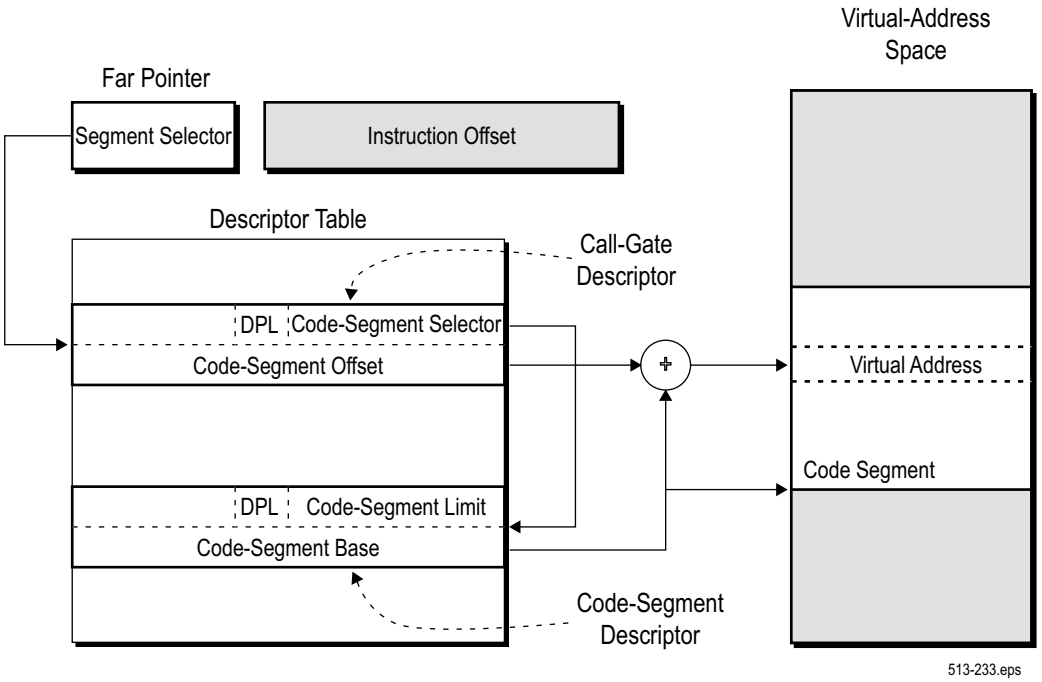


Figure 4-30. Legacy-Mode Call-Gate Transfer Mechanism

[AMD Public Use]

Figure 4-31 shows a call-gate control transfer in long mode. The long-mode call-gate descriptor format is expanded by 64 bits to hold a full 64-bit offset into the virtual-address space. Only long-mode call gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy-mode 32-bit call-gate types are redefined in long mode as 64-bit types, and 16-bit call-gate types are illegal.

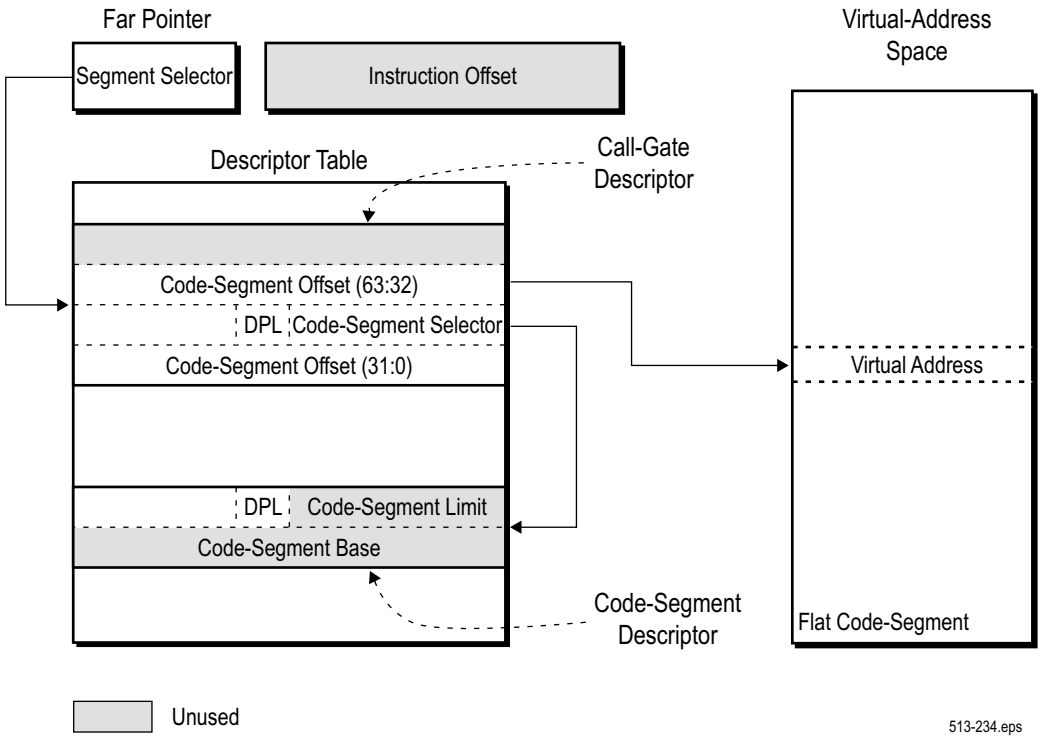


Figure 4-31. Long-Mode Call-Gate Access Mechanism

A long-mode call gate must reference a 64-bit code-segment descriptor. In 64-bit mode, the code-segment descriptor base-address and limit fields are ignored. The target virtual-address is the 64-bit offset field in the expanded call-gate descriptor.

Privilege Checks. Before loading the CS register with the code-segment selector located in the call gate, the processor performs three privilege checks. The following checks are performed when either conforming or nonconforming code segments are referenced:

1. The processor compares the CPL with the call-gate DPL from the call-gate descriptor (DPL_G). The CPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $CPL \leq DPL_G$.

[AMD Public Use]

2. The processor compares the RPL in the call-gate selector with DPL_G. The RPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $RPL \leq DPL_G$.
3. The processor compares the CPL with the target code-segment DPL from the code-segment descriptor (DPL_S). The type of comparison varies depending on the type of control transfer.
   - When a call—or a jump to a *conforming* code segment—is used to transfer control through a call gate, the CPL must be numerically *greater than or equal to* DPL_S for this check to pass. (This check prevents control transfers to less-privileged programs.) In other words, the following expression must be true: $CPL \geq DPL_S$.
   - When a JMP instruction is used to transfer control through a call gate to a *nonconforming* code segment, the CPL must be numerically *equal to* DPL_S for this check to pass. (JMP instructions cannot change CPL.) In other words, the following expression must be true: $CPL = DPL_S$.

Figure 4-32 on page 118 shows two examples of call-gate privilege checks. In Example 1, all privilege checks pass as follows:

- The call-gate DPL (DPL_G) is at the lowest privilege (3), specifying that software running at any privilege level (CPL) can access the gate.
- The selector referencing the call gate passes its privilege check because the RPL is numerically less than or equal to DPL_G.
- The target code segment is at the highest privilege level ($DPL_S = 0$). This means software running at any privilege level can access the target code segment through the call gate.

[AMD Public Use]

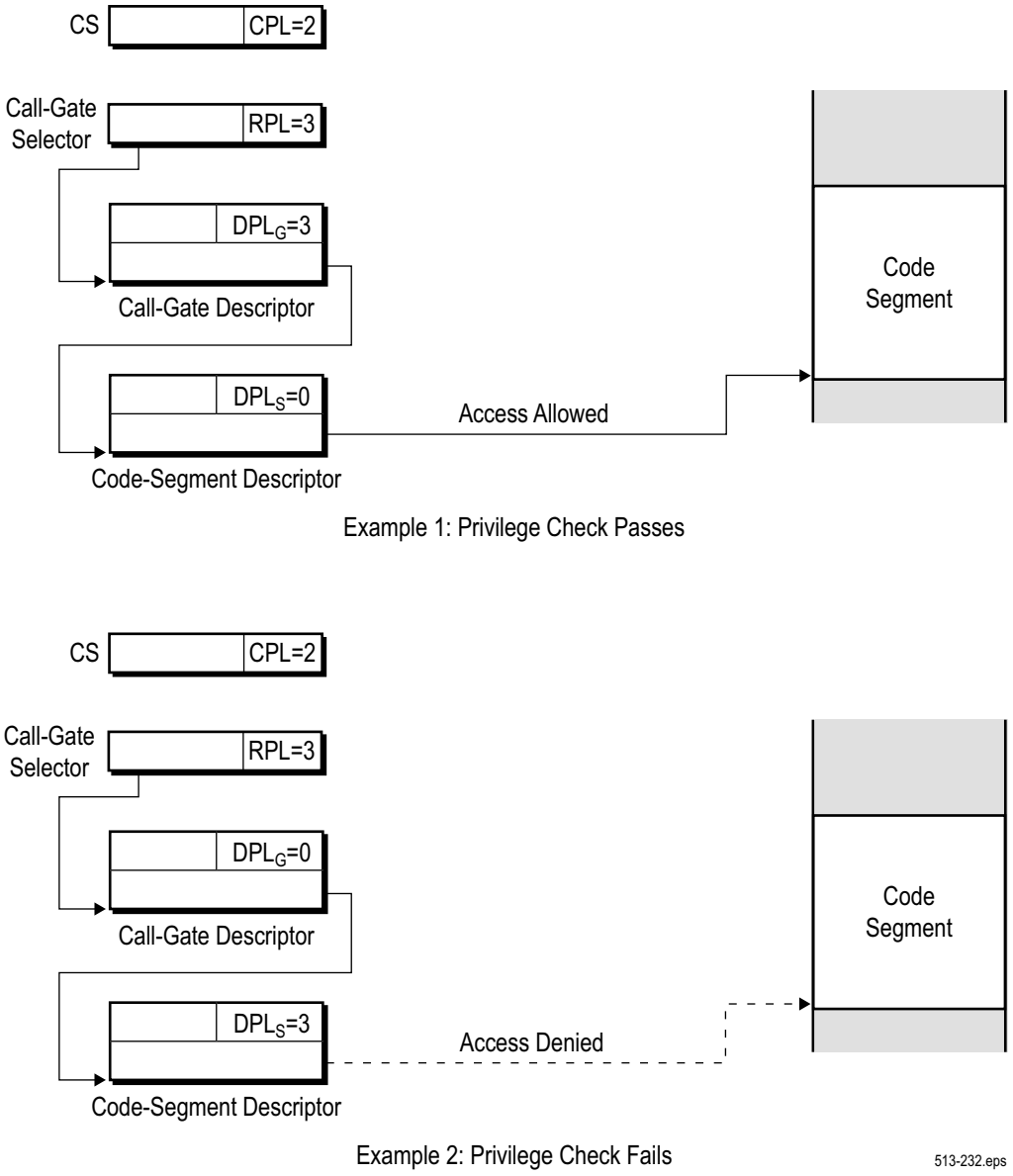


Figure 4-32. Privilege-Check Examples for Call Gates

In Example 2, all privilege checks fail as follows:

- The call-gate DPL (DPL_G) specifies that only software at privilege-level 0 can access the gate. The current program does not have enough privilege to access the call gate because its CPL is 2.
- The selector referencing the call-gate descriptor does not have enough privilege to complete the reference. Its RPL is numerically greater than DPL_G.

[AMD Public Use]

- The target code segment is at a lower privilege ($DPL_S = 3$) than the currently running software (CPL = 2). Transitions from more-privileged software to less-privileged software are not allowed, so this privilege check fails as well.

Although all three privilege checks failed in Example 2, failing only one check is sufficient to deny access into the target code segment.

Stack Switching. The processor performs an automatic stack switch when a control transfer causes a change in privilege levels to occur. Switching stacks isolates more-privileged software stacks from less-privileged software stacks and provides a mechanism for saving the return pointer back to the program that initiated the call.

When switching to more-privileged software, as is done when transferring control using a call gate, the processor uses the corresponding stack pointer (privilege-level 0, 1, or 2) stored in the task-state segment (TSS). The format of the stack pointer stored in the TSS depends on the system-software operating mode:

- Legacy-mode system software stores a 32-bit ESP value (stack offset) and 16-bit SS selector register value in the TSS for each of three privilege levels 0, 1, and 2.
- Long-mode system software stores a 64-bit RSP value in the TSS for privilege levels 0, 1, and 2. No SS register value is stored in the TSS because in long mode a call gate *must* reference a 64-bit code-segment descriptor. 64-bit mode does not use segmentation, and the stack pointer consists solely of the 64-bit RSP. Any value loaded in the SS register is ignored.

See “Task-Management Resources” on page 369 for more information on the legacy-mode and long-mode TSS formats.

Figure 4-33 on page 120 shows a 32-bit stack in legacy mode before and after the automatic stack switch. This particular example assumes that parameters are passed from the current program to the target program. The process followed by legacy mode in switching stacks and copying parameters is:

1. The target code-segment DPL is read by the processor and used as an index into the TSS for selecting the new stack pointer (SS:ESP). For example, if DPL=1 the processor selects the SS:ESP for privilege-level 1 from the TSS.
2. The SS and ESP registers are loaded with the new SS:ESP values read from the TSS.
3. The old values of the SS and ESP registers are pushed onto the stack pointed to by the new SS:ESP.
4. The 5-bit count field is read from the call-gate descriptor.
5. The number of parameters specified in the count field (up to 31) are copied from the old stack to the new stack. The size of the parameters copied by the processor depends on the call-gate size: 32-bit call gates copy 4-byte parameters and 16-bit call gates copy 2-byte parameters.
6. The return pointer is pushed onto the stack. The return pointer consists of the current CS-register value and the EIP of the instruction following the calling instruction.

[AMD Public Use]

7. The CS register is loaded from the segment-selector field in the call-gate descriptor, and the EIP is loaded from the offset field in the call-gate descriptor.
8. The target program begins executing with the instruction referenced by new CS:EIP.

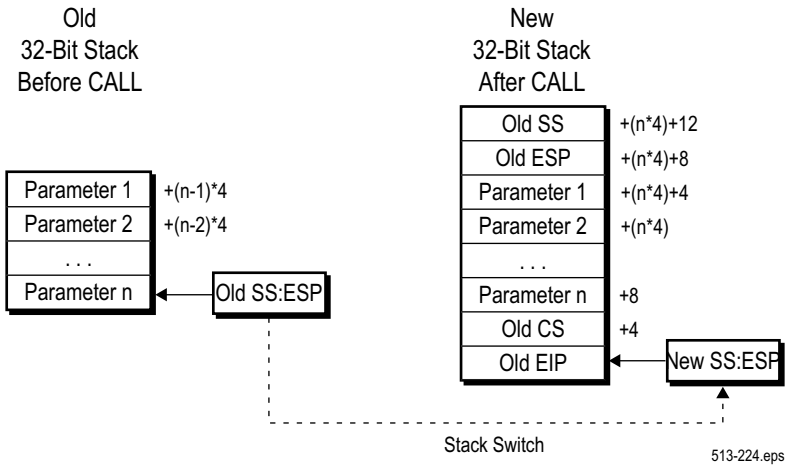


Figure 4-33. Legacy-Mode 32-Bit Stack Switch, with Parameters

Figure 4-34 shows a 32-bit stack in legacy mode before and after the automatic stack switch when no parameters are passed (count=0). Most software does not use the call-gate descriptor count-field to pass parameters. System software typically defines linkage mechanisms that do not rely on automatic parameter copying.

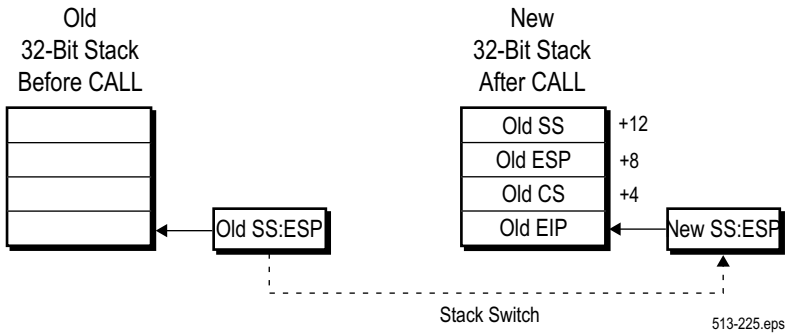


Figure 4-34. 32-Bit Stack Switch, No Parameters—Legacy Mode

Figure 4-35 on page 121 shows a long-mode stack switch. In long mode, all call gates *must* reference 64-bit code-segment descriptors, so a long-mode stack switch uses a 64-bit stack. The process of

[AMD Public Use]

switching stacks in long mode is similar to switching in legacy mode when no parameters are passed. The process is as follows:

1. The target code-segment DPL is read by the processor and used as an index into the 64-bit TSS for selecting the new stack pointer (RSP).
2. The RSP register is loaded with the new RSP value read from the TSS. The SS register is loaded with a null selector (SS=0). Setting the new SS selector to null allows proper handling of nested control transfers in 64-bit mode. See “Nested Returns to 64-Bit Mode Procedures” on page 123 for additional information.

   As in legacy mode, it is desirable to keep the stack-segment requestor privilege-level (SS.RPL) equal to the current privilege-level (CPL). When using a call gate to change privilege levels, the SS.RPL is updated to reflect the new CPL. The SS.RPL is restored from the return-target CS.RPL on the subsequent privilege-level-changing far return.
3. The old values of the SS and RSP registers are pushed onto the stack pointed to by the new RSP. The old SS value is popped on a subsequent far return. This allows system software to set up the SS selector for a compatibility-mode process by executing a RET (or IRET) that changes the privilege level.
4. The return pointer is pushed onto the stack. The return pointer consists of the current CS-register value and the RIP of the instruction following the calling instruction.
5. The CS register is loaded from the segment-selector field in the long-mode call-gate descriptor, and the RIP is loaded from the offset field in the long-mode call-gate descriptor.

The target program begins execution with the instruction referenced by the new RIP.

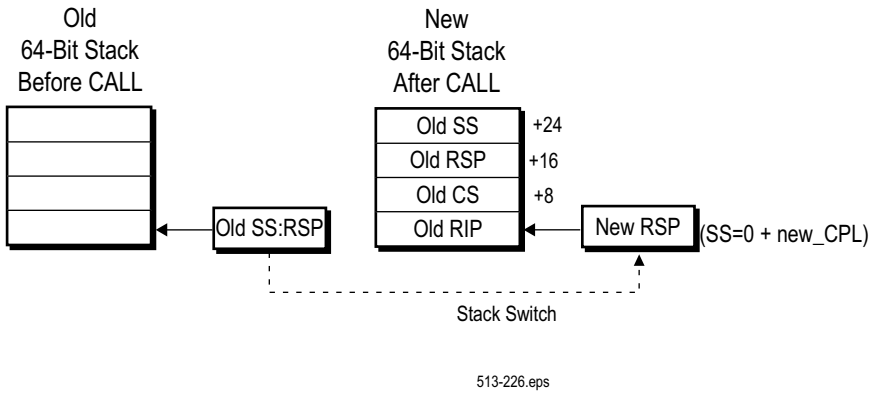


Figure 4-35. Stack Switch—Long Mode

All long-mode stack pushes resulting from a privilege-level-changing far call are eight-bytes wide and increment the RSP by eight. Long mode ignores the call-gate count field and does not support the automatic parameter-copy feature found in legacy mode. Software can access parameters on the old stack, if necessary, by referencing the old stack segment selector and stack pointer saved on the new process stack.

[AMD Public Use]

4.11.3 Return Control Transfers

Returns to calling programs can be performed by using the RET instruction. The following types of returns are possible:

- *Near Return*—Near returns perform control transfers within the same code segment, so the CS register is unchanged. The new offset is popped off the stack and into the rIP register. No privilege checks are performed.
- *Far Return, Same Privilege*—A far return transfers control from one code segment to another. When the original code segment is at the same privilege level as the target code segment, a far pointer (CS:rIP) is popped off the stack and the RPL of the new code segment (CS) is checked. If the requested privilege level (RPL) matches the current privilege level (CPL), then a return is made to the same privilege level. This prevents software from changing the CS value on the stack in an attempt to return to higher-privilege software.
- *Far Return, Less Privilege*—Far returns can change privilege levels, but only to a *lower*-privilege level. In this case a stack switch is performed between the current, higher-privilege program and the lower-privilege return program. The CS-register and rIP-register values are popped off the stack. The lower-privilege stack pointer is also popped off the stack and into the SS register and rSP register. The processor checks both the CS and SS privilege levels to ensure they are equal and at a lesser privilege than the current CS.

  In the case of nested returns to 64-bit mode, a null selector can be popped into the SS register. See “Nested Returns to 64-Bit Mode Procedures” on page 123.

  Far returns also check the privilege levels of the DS, ES, FS and GS selector registers. If any of these segment registers have a selector with a higher privilege than the return program, the segment register is loaded with the null selector.

Stack Switching. The stack switch performed by a far return to a lower-privilege level reverses the stack switch of a call gate to a higher-privilege level, except that parameters are never automatically copied as part of a return. The process followed by a far-return stack switch in long mode and legacy mode is:

1. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that a lower-privilege control transfer is occurring.
2. The return-program instruction pointer is popped off the current-program (higher privilege) stack and loaded into the CS and rIP registers.
3. The return instruction can include an immediate operand that specifies the number of additional bytes to be popped off of the stack. These bytes may correspond to the parameters pushed onto the stack previously by a call through a call gate containing a non-zero parameter-count field. If the return includes the immediate operand, then the stack pointer is adjusted upward by adding the specified number of bytes to the rSP.
4. The return-program stack pointer is popped off the current-program (higher privilege) stack and loaded into the SS and rSP registers. In the case of nested returns to 64-bit mode, a null selector can be popped into the SS register.

[AMD Public Use]

The operand size of a far return determines the size of stack pops when switching stacks. If a far return is used in 64-bit mode to return from a prior call through a long-mode call gate, the far return must use a 64-bit operand size. The 64-bit operand size allows the far return to properly read the stack established previously by the far call.

Nested Returns to 64-Bit Mode Procedures. In long mode, a far call that changes privilege levels causes the SS register to be loaded with a null selector (this is the same action taken by an interrupt in long mode). If the called procedure performs another far call to a higher-privileged procedure, or is interrupted, the null SS selector is pushed onto the stack frame, and another null selector is loaded into the SS register. Using a null selector in this way allows the processor to properly handle returns nested within 64-bit-mode procedures and interrupt handlers.

Normally, a RET that pops a null selector into the SS register causes a general-protection exception (#GP) to occur. However, in long mode, the null selector acts as a flag indicating the existence of nested interrupt handlers or other privileged software in 64-bit mode. Long mode allows RET to pop a null selector into SS from the stack under the following conditions:

- The target mode is 64-bit mode.
- The target CPL is less than 3.

In this case, the processor does not load an SS descriptor, and the null selector is loaded into SS without causing a #GP exception.

4.12 Limit Checks

Except in 64-bit mode, limit checks are performed by all instructions that reference memory. Limit checks detect attempts to access memory outside the current segment boundary, attempts at executing instructions outside the current code segment, and indexing outside the current descriptor table. If an instruction fails a limit check, either (1) a general-protection exception occurs for all other segment-limit violations or (2) a stack-fault exception occurs for stack-segment limit violations.

In 64-bit mode, segment limits are *not checked* during accesses to any segment referenced by the CS, DS, ES, FS, GS, and SS selector registers. Instead, the processor checks that the virtual addresses used to reference memory are in canonical-address form. In 64-bit mode, as with legacy mode and compatibility mode, descriptor-table limits *are checked*.

4.12.1 Determining Limit Violations

To determine segment-limit violations, the processor checks a virtual (linear) address to see if it falls outside the valid range of segment offsets determined by the segment-limit field in the descriptor. If any part of an operand or instruction falls outside the segment-offset range, a limit violation occurs. For example, a doubleword access, two bytes from an upper segment boundary, causes a segment violation because half of the doubleword is outside the segment.

[AMD Public Use]

Three bits from the descriptor entry are used to control how the segment-limit field is interpreted: the granularity (G) bit, the default operand-size (D) bit, and for data segments, the expand-down (E) bit. See “Legacy Segment Descriptors” on page 90 for a detailed description of each bit.

For all segments other than expand-down segments, the minimum segment-offset is 0. The maximum segment-offset depends on the value of the G bit:

- If G=0 (byte granularity), the maximum allowable segment-offset is equal to the value of the segment-limit field.
- If G=1 (4096-byte granularity), the segment-limit field is first scaled by 4096 (1000h). Then 4095 (0FFFh) is added to the scaled value to arrive at the maximum allowable segment-offset, as shown in the following equation:

  maximum segment-offset = (limit × 1000h) + 0FFFh

  For example, if the segment-limit field is 0100h, then the maximum allowable segment-offset is (0100h × 1000h) + 0FFFh = 10_1FFFh.

In both cases, the maximum segment-size is specified when the descriptor segment-limit field is 0F_FFFFh.

Expand-Down Segments. Expand-down data segments are supported in legacy mode and compatibility mode but not in 64-bit mode. With expand-down data segments, the maximum segment offset depends on the value of the D bit in the data-segment descriptor:

- If D=0 the maximum segment-offset is 0_FFFFh.
- If D=1 the maximum segment-offset is 0_FFFF_FFFFh.

The minimum allowable segment offset in expand-down segments depends on the value of the G bit:

- If G=0 (byte granularity), the minimum allowable segment offset is the segment-limit value plus 1.

  For example, if the segment-limit field is 0100h, then the minimum allowable segment-offset is 0101h.
- If G=1 (4096-byte granularity), the segment-limit value in the descriptor is first scaled by 4096 (1000h), and then 4095 (0FFFh) is added to the scaled value to arrive at a scaled segment-limit value. The minimum allowable segment-offset is this scaled segment-limit value plus 1, as shown in the following equation:

  minimum segment-offset = (limit × 1000) + 0FFFh + 1

  For example, if the segment-limit field is 0100h, then the minimum allowable segment-offset is (0100h × 1000h) + 0FFFh + 1 = 10_1000h.

For expand-down segments, the maximum segment size is specified when the segment-limit value is 0.

[AMD Public Use]

4.12.2 Data Limit Checks in 64-bit Mode

In 64-bit mode, data reads and writes are not normally checked for segment-limit violations. When EFER.LMSLE = 1, reads and writes in 64-bit mode at CPL > 0, using the DS, ES, FS, or SS segments, have a segment-limit check applied.

This limit-check uses the 32-bit segment-limit to find the maximum allowable address in the top 4GB of the 64-bit virtual (linear) address space.

Table 4-8. Segment Limit Checks in 64-Bit Mode

Memory Address	Effect of Limit Check
Linear Address ≤ (0FFFFFFFF_00000000h + 32-bit Limit)	Access OK.
Linear Address > (0FFFFFFFF_00000000h + 32-bit Limit)	Exception (#GP or #SS)

This segment-limit check does not apply to accesses through the GS segment, or to code reads. If the DS, ES, FS, or SS segment is null or expand-down, the effect of the limit check is undefined. Data segment limit checking in 64-bit mode is not supported by all processor implementations and has been deprecated. If CPUID Fn8000_0008_EBX[EferLmlseUnsupported](bit 20) = 1, 64-bit mode segment limit checking is not supported and attempting to enable this feature by setting EFER.LMSLE =1 will result in a #GP exception.

4.13 Type Checks

Type checks prevent software from using descriptors in invalid ways. Failing a type check results in an exception. Type checks are performed using five bits from the descriptor entry: the S bit and the 4-bit Type field. Together, these five bits are used to specify the descriptor type (code, data, segment, or gate) and its access characteristics. See “Legacy Segment Descriptors” on page 90 for a detailed description of the S bit and Type-field encodings. Type checks are performed by the processor in compatibility mode as well as legacy mode. Limited type checks are performed in 64-bit mode.

4.13.1 Type Checks in Legacy and Compatibility Modes

The type checks performed in legacy mode and compatibility mode are listed in the following sections.

Descriptor-Table Register Loads. Loads into the LDTR and TR descriptor-table registers are checked for the appropriate system-segment type. The LDTR can only be loaded with an LDT descriptor, and the TR only with a TSS descriptor. The checks are performed during any action that causes these registers to be loaded. This includes execution of the LLDT and LTR instructions and during task switches.

Segment Register Loads. The following restrictions are placed on the segment-descriptor types that can be loaded into the six user segment registers:

- Only code segments can be loaded into the CS register.
- Only writable data segments can be loaded into the SS register.

[AMD Public Use]

- Only the following segment types can be loaded into the DS, ES, FS, or GS registers:
  - Read-only or read/write data segments.
  - Readable code segments.

These checks are performed during any action that causes the segment registers to be loaded. This includes execution of the MOV segment-register instructions, control transfers, and task switches.

Control Transfers. Control transfers (branches and interrupts) place additional restrictions on the segment types that can be referenced during the transfer:

- The segment-descriptor type referenced by far CALLs and far JMPs must be one of the following:
  - A code segment
  - A call gate or a task gate
  - An available TSS (only allowed in legacy mode)
  - A task gate (only allowed in legacy mode)
- Only code-segment descriptors can be referenced by call-gate, interrupt-gate, and trap-gate descriptors.
- Only TSS descriptors can be referenced by task-gate descriptors.
- The link field (selector) in the TSS can only point to a TSS descriptor. This is checked during an IRET control transfer to a task.
- The far RET and far IRET instructions can only reference code-segment descriptors.
- The interrupt-descriptor table (IDT), which is referenced during interrupt control transfers, can only contain interrupt gates, trap gates, and task gates.

Segment Access. After a segment descriptor is successfully loaded into one of the segment registers, reads and writes into the segments are restricted in the following ways:

- Writes are not allowed into read-only data-segment types.
- Writes are not allowed into code-segment types (executable segments).
- Reads from code-segment types are not allowed if the readable (R) type bit is cleared to 0.

These checks are generally performed during execution of instructions that access memory.

4.13.2 Long Mode Type Check Differences

Compatibility Mode and 64-Bit Mode. The following type checks differ in long mode (64-bit mode and compatibility mode) as compared to legacy mode:

- *System Segments*—System-segment types are checked, but the following types that are valid in legacy mode are illegal in long mode:
  - 16-bit available TSS.
  - 16-bit busy TSS.

[AMD Public Use]

  - Type-field encoding of 00h in the upper half of a system-segment descriptor to indicate an illegal type and prevent access as a legacy descriptor.
- *Gates*—Gate-descriptor types are checked, but the following types that are valid in legacy mode are illegal in long mode:
  - 16-bit call gate.
  - 16-bit interrupt gate.
  - 16-bit trap gate.
  - Task gate.

64-Bit Mode. 64-bit mode disables segmentation, and most of the segment-descriptor fields are ignored. The following list identifies situations where type checks in 64-bit mode differ from those in compatibility mode and legacy mode:

- *Code Segments*—The readable (R) type bit is ignored in 64-bit mode. None of the legacy type-checks that prevent reads from or writes into code segments are performed in 64-bit mode.
- *Data Segments*—Data-segment type attributes are ignored in 64-bit mode. The writable (W) and expand-down (E) type bits are ignored. All data segments are treated as writable.

[AMD Public Use]

5 Page Translation and Protection

The x86 page-translation mechanism (or simply *paging mechanism*) enables system software to create separate address spaces for each process or application. These address spaces are known as *virtual-address* spaces. System software uses the paging mechanism to selectively map individual pages of physical memory into the virtual-address space using a set of hierarchical address-translation tables known collectively as *page tables*.

The paging mechanism and the page tables are used to provide each process with its own private region of physical memory for storing its code and data. Processes can be protected from each other by isolating them within the virtual-address space. A process cannot access physical memory that is not mapped into its virtual-address space by system software.

System software can use the paging mechanism to selectively map physical-memory pages into multiple virtual-address spaces. Mapping physical pages in this manner allows them to be shared by multiple processes and applications. The physical pages can be configured by the page tables to allow read-only access. This prevents applications from altering the pages and ensures their integrity for use by all applications.

Shared mapping is typically used to allow access of shared-library routines by multiple applications. A read-only copy of the library routine is mapped to each application virtual-address space, but only a single copy of the library routine is present in physical memory. This capability also allows a copy of the operating-system kernel and various device drivers to reside within the application address space. Applications are provided with efficient access to system services without requiring costly address-space switches.

The system-software portion of the address space necessarily includes system-only data areas that must be protected from accesses by applications. System software uses the page tables to protect this memory by designating the pages as *supervisor* pages. Such pages are only accessible by system software.

When the supervisor mode execution prevention (SMEP) feature is supported and enabled, attempted instruction fetches from user-mode accessible pages while in supervisor-mode triggers a page fault (#PF). This protects the integrity of system software by preventing the execution of instructions at a supervisor privilege level (CPL < 3) when these instructions could have been written or modified by user-mode code.

When the supervisor mode access prevention (SMAP) feature is supported and enabled and RFLAGS.AC=0, some attempted explicit data accesses from user-mode accessible pages while in supervisor-mode trigger a page fault (#PF). This protects the integrity of system software by preventing the use of data at a supervisor privilege level (CPL < 3) that could have been modified by user-mode code. Supervisor software that requires access to data which is marked as user-mode accessible may temporarily suppress SMAP checks by modifying RFLAGS.AC, such as with the CLAC/STAC instructions.

[AMD Public Use]

Finally, system software can use the paging mechanism to map multiple, large virtual-address spaces into a much smaller amount of physical memory. Each application can use the entire 32-bit or 64-bit virtual-address space. System software actively maps the most-frequently-used virtual-memory pages into the available pool of physical-memory pages. The least-frequently-used virtual-memory pages are swapped out to the hard drive. This process is known as *demand-paged virtual memory*.

5.1 Page Translation Overview

The legacy x86 architecture provides support for translating 32-bit virtual addresses into 32-bit physical addresses (larger physical addresses, such as 36-bit or 40-bit addresses, are supported as a special mode). The AMD64 architecture enhances this support to allow translation of 64-bit virtual addresses into 52-bit physical addresses, although processor implementations can support smaller virtual-address and physical-address spaces.

Virtual addresses are translated to physical addresses through hierarchical translation tables created and managed by system software. Each table contains a set of entries that point to the next-lower table in the translation hierarchy. A single table at one level of the hierarchy can have hundreds of entries, each of which points to a unique table at the next-lower hierarchical level. Each lower-level table can in turn have hundreds of entries pointing to tables further down the hierarchy. The lowest-level table in the hierarchy points to the translated physical page.

Figure 5-1 on page 130 shows an overview of the page-translation hierarchy used in long mode. Legacy mode paging uses a subset of this translation hierarchy (the page-map level-4 table does not exist in legacy mode and the PDP table may or may not be used, depending on which paging mode is enabled). The 5th level of translation, Page Map Level 5, only exists when 5-Level paging is enabled. As this figure shows, a virtual address is divided into fields, each of which is used as an offset into a translation table. The complete translation chain is made up of all table entries referenced by the virtual-address fields. The lowest-order virtual-address bits are used as the byte offset into the physical page.

[AMD Public Use]

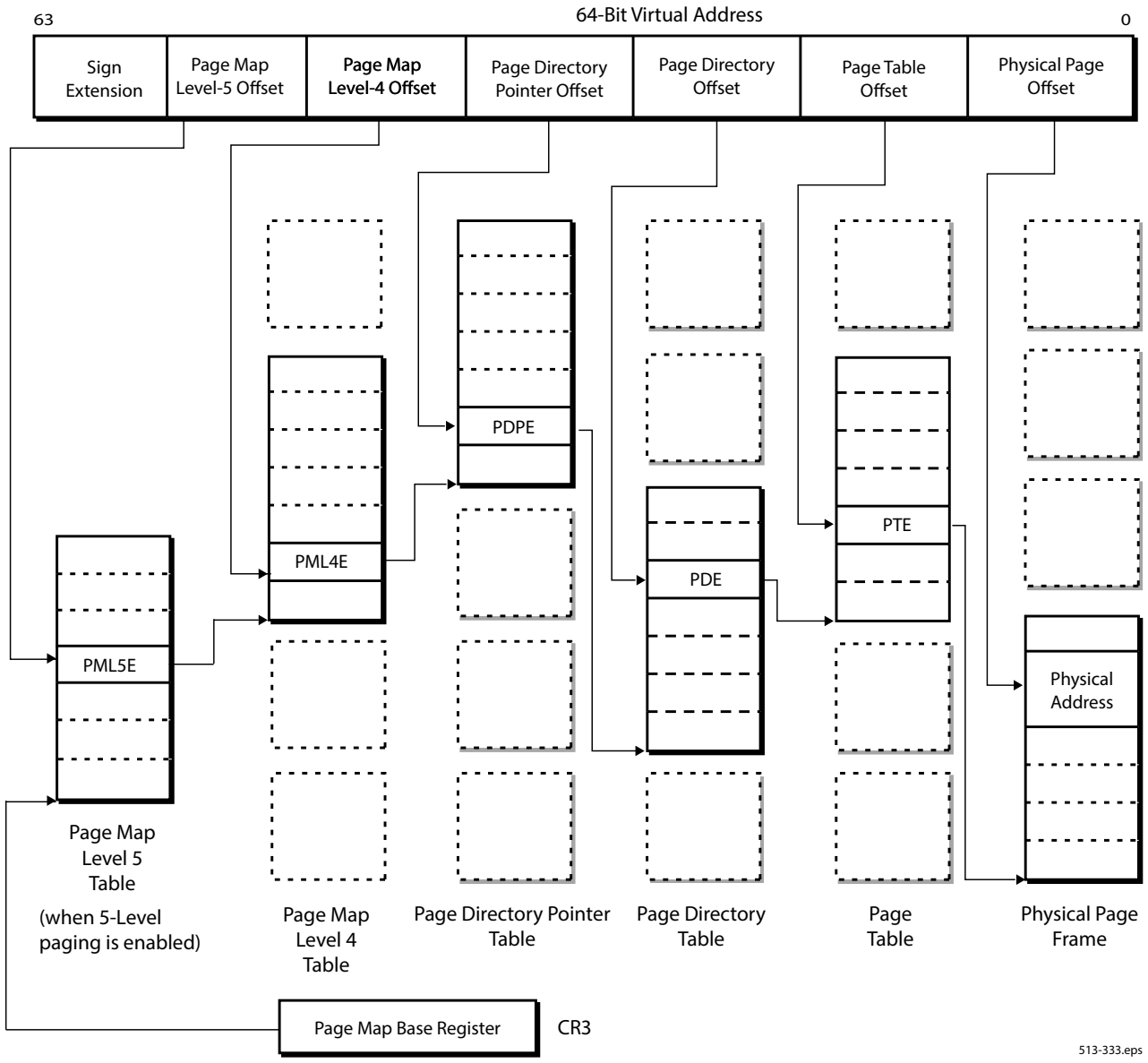


Figure 5-1. Virtual to Physical Address Translation—Long Mode

The following physical-page sizes are supported: 4 Kbytes, 2 Mbytes, 4 Mbytes, and 1 Gbytes. In long mode 4-Kbyte, 2-MByte, and 1-GByte sizes are available. In legacy mode 4-Kbyte, 2-MByte, and 4-MByte sizes are available.

Virtual addresses are 32 bits long, and physical addresses up to the supported physical-address size can be used. The AMD64 architecture enhances the legacy translation support by allowing virtual addresses of up to 64 bits long to be translated into physical addresses of up to 52 bits long.

[AMD Public Use]

Currently, the AMD64 architecture defines a mechanism for translating 48-bit virtual addresses to 52-bit physical addresses. The mechanism used to translate a full 64-bit virtual address is reserved and will be described in a future AMD64 architectural specification.

5.1.1 Page-Translation Options

The form of page-translation support available to software depends on which paging features are enabled. Four controls are available for selecting the various paging alternatives:

- Page-Translation Enable (CR0.PG)
- Physical-Address Extensions (CR4.PAE)
- Page-Size Extensions (CR4.PSE)
- Long-Mode Active (EFER.LMA)

Not all paging alternatives are available in all modes. Table 5-1 summarizes the paging support available in each mode.

Table 5-1. Supported Paging Alternatives (CR0.PG=1)

Mode	Physical-Address Extensions (CR4.PAE)	Page-Size Extensions (CR4.PSE)	Page-Directory Pointer Offset	Page-Directory Page Size	Resulting Physical-Page Size	Maximum Virtual Address	Maximum Physical Address
Long Mode (64-bit and compatibility modes)	Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	64-bit	52-bit
				PDE.PS=1	2 Mbyte		
			PDPE.PS=1	–	1 Gbyte		
Legacy Mode	Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	32-bit	52-bit
				PDE.PS=1	2 Mbyte		52-bit
	Disabled	Disabled		–	4 Kbyte		32-bit
		Enabled		PDE.PS=0	4 Kbyte		32-bit
				PDE.PS=1	4 Mbyte		40-bit

5.1.2 Page-Translation Enable (PG) Bit

Page translation is controlled by the PG bit in CR0 (bit 31). When CR0.PG is set to 1, page translation is enabled. When CR0.PG is cleared to 0, page translation is disabled.

The AMD64 architecture uses CR0.PG to activate and deactivate long mode when long mode is enabled. See “Enabling and Activating Long Mode” on page 511 for more information.

5.1.3 Physical-Address Extensions (PAE) Bit

Physical-address extensions are controlled by the PAE bit in CR4 (bit 5). When CR4.PAE is set to 1, physical-address extensions are enabled. When CR4.PAE is cleared to 0, physical-address extensions are disabled.

[AMD Public Use]

Setting CR4.PAE = 1 enables virtual addresses to be translated into physical addresses up to 52 bits long. This is accomplished by doubling the size of paging data-structure entries from 32 bits to 64 bits to accommodate the larger physical base-addresses for physical-pages.

PAE must be enabled before activating long mode. See “Enabling and Activating Long Mode” on page 511.

5.1.4 Page-Size Extensions (PSE) Bit

Page-size extensions are controlled by the PSE bit in CR4 (bit 4). Setting CR4.PSE to 1 allows operating-system software to use 4-Mbyte physical pages in the translation process. The 4-Mbyte physical pages can be mixed with standard 4-Kbyte physical pages or replace them entirely. The selection of physical-page size is made on a page-directory-entry basis. See “Page Size (PS) Bit” on page 157 for more information on physical-page size selection. When CR4.PSE is cleared to 0, page-size extensions are disabled.

The choice of 2 Mbyte or 4 Mbyte as the large physical-page size depends on the value of CR4.PSE and CR4.PAE, as follows:

- If physical-address extensions are enabled (CR4.PAE=1), the large physical-page size is 2 Mbytes, regardless of the value of CR4.PSE.
- If physical-address extensions are disabled (CR4.PAE=0) *and* CR4.PSE=1, the large physical-page size is 4 Mbytes.
- If both CR4.PAE=0 and CR4.PSE=0, the only available page size is 4 Kbytes.

The value of CR4.PSE is ignored when long mode is active. This is because physical-address extensions must be enabled in long mode, and the only available page sizes are 4 Kbytes and 2 Mbytes.

In legacy mode, physical addresses up to 40 bits long can be translated from 32-bit virtual addresses using 32-bit paging data-structure entries when 4-Mbyte physical-page sizes are selected. In this special case, CR4.PSE=1 and CR4.PAE=0. See “4-Mbyte Page Translation” on page 136 for a description of the 4-Mbyte PDE that supports 40-bit physical-address translation. The 40-bit physical-address capability is an AMD64 architecture enhancement over the similar capability available in the legacy x86 architecture.

5.1.5 Page-Directory Page Size (PS) Bit

The page directory offset entry (PDE) and page directory pointer offset entry (PDPE) are data structures used in page translation (see Figure 5-1 on page 130). The page-size (PS) bit in the PDE (bit 7, referred to as PDE.PS) selects between standard 4-Kbyte physical-page sizes and larger (2-Mbyte or 4-Mbyte) physical-page sizes. The page-size (also PS) bit in the PDPE (bit 7, referred to as PDPE.PS) selects between 2-Mbyte and 1-Gbyte physical-page sizes in long mode.

When PDE.PS is set to 1, large physical pages are used, and the PDE becomes the lowest level of the translation hierarchy. The size of the large page is determined by the values of CR4.PAE and

[AMD Public Use]

CR4.PSE, as shown in Figure 5-1 on page 131. When PDE.PS is cleared to 0, standard 4-Kbyte physical pages are used, and the PTE is the lowest level of the translation hierarchy.

When PDPE.PS is set to 1, 1-Gbyte physical pages are used, and the PDPE becomes the lowest level of the translation hierarchy. Neither the PDE nor PTE are used for 1-Gbyte paging.

5.2 Legacy-Mode Page Translation

Legacy mode supports two forms of translation:

- *Non-PAE Paging*—This is used when physical-address extensions are disabled (CR4.PAE=0). Entries in the page translation table are 32 bits and are used to translate 32-bit virtual addresses into physical addresses as large as 40 bits.
- *PAE Paging*—This is used when physical-address extensions are enabled (CR4.PAE=1). Entries in the page translation table are 64 bits and are used to translate 32-bit virtual addresses into physical addresses as large as 52 bits.

Legacy paging uses up to three levels of page-translation tables, depending on the paging form used and the physical-page size. Entries within each table are selected using virtual-address bit fields. The legacy page-translation tables are:

- *Page Table*—Each page-table entry (PTE) points to a physical page. If 4-Kbyte pages are used, the page table is the lowest level of the page-translation hierarchy. PTEs are not used when translating 2-Mbyte or 4-Mbyte pages.
- *Page Directory*—If 4-Kbyte pages are used, each page-directory entry (PDE) points to a page table. If 2-Mbyte or 4-Mbyte pages are used, a PDE is the lowest level of the page-translation hierarchy and points to a physical page. In non-PAE paging, the page directory is the highest level of the translation hierarchy.
- *Page-Directory Pointer*—Each page-directory pointer entry (PDPE) points to a page directory. Page-directory pointers are only used in PAE paging (CR4.PAE=1), and are the highest level in the legacy page-translation hierarchy.

The translation-table-entry formats and how they are used in the various forms of legacy page translation are described beginning on page 134.

5.2.1 CR3 Register

The CR3 register is used to point to the base address of the highest-level page-translation table. The base address is either the page-directory pointer table or the page directory table. The CR3 register format depends on the form of paging being used. Figure 5-2 on page 134 shows the CR3 format when non-PAE paging is used (CR4.PAE=0). Figure 5-3 shows the CR3 format when PAE paging is used (CR4.PAE=1).

[AMD Public Use]

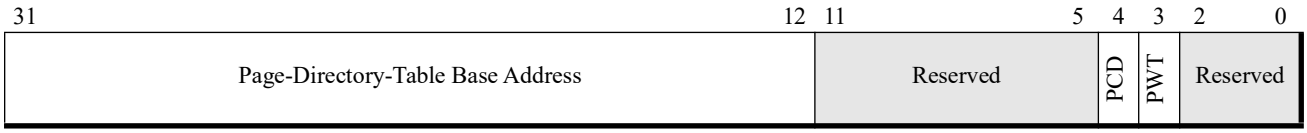


Figure 5-2. Control Register 3 (CR3)—Non-PAE Paging Legacy-Mode

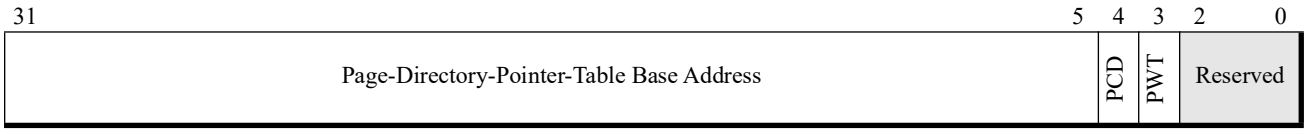


Figure 5-3. Control Register 3 (CR3)—PAE Paging Legacy-Mode

The CR3 register fields for legacy-mode paging are:

Table Base Address Field. This field points to the starting physical address of the highest-level page-translation table. The size of this field depends on the form of paging used:

- *Non-PAE Paging (CR4.PAE=0)*—This 20-bit field occupies bits 31:12, and points to the base address of the page-directory table. The page-directory table is aligned on a 4-Kbyte boundary, with the low-order 12 address bits 11:0 assumed to be 0. This yields a total base-address size of 32 bits.
- *PAE Paging (CR4.PAE=1)*—This field is 27 bits and occupies bits 31:5. The CR3 register points to the base address of the page-directory-pointer table. The page-directory-pointer table is aligned on a 32-byte boundary, with the low 5 address bits 4:0 assumed to be 0.

Page-Level Writethrough (PWT) Bit. Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

Page-Level Cache Disable (PCD) Bit. Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

Reserved Bits. Reserved fields should be cleared to 0 by software when writing CR3.

5.2.2 Non-PAE Paging

Non-PAE paging (CR4.PAE=0) supports 4-Kbyte and 4-Mbyte physical pages, as described in the following sections.

4-Kbyte Page Translation. 4-Kbyte physical-page translation is performed by dividing the 32-bit virtual address into three fields. Each of the upper two fields is used as an index into a two-level page-translation hierarchy. The virtual-address fields are used as follows, and are shown in Figure 5-4:

- Bits 31:22 index into the 1024-entry page-directory table.

[AMD Public Use]

- Bits 21:12 index into the 1024-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

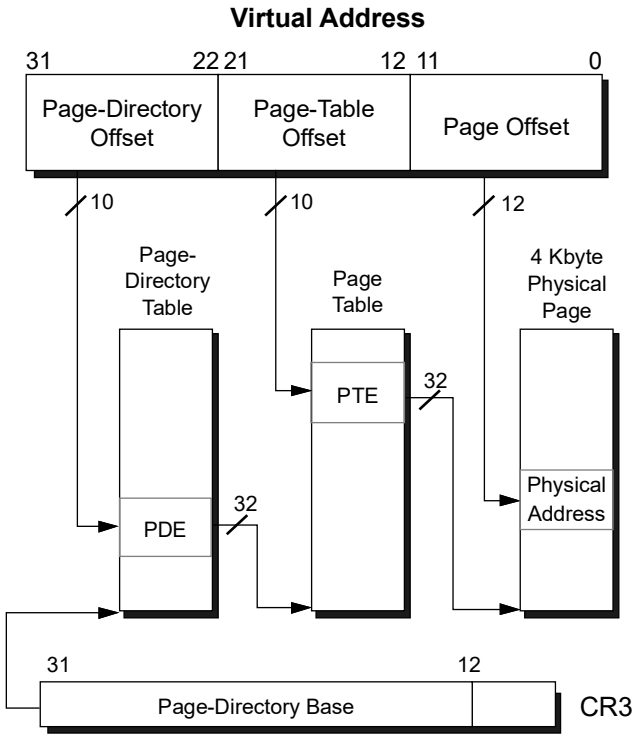


Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode

Figure 5-5 on page 135 shows the format of the PDE (page-directory entry), and Figure 5-6 on page 136 shows the format of the PTE (page-table entry). Each table occupies 4 Kbytes and can hold 1024 of the 32-bit table entries. The fields within these table entries are described in "Page-Translation-Table Entry Fields" on page 155.

Figure 5-5 shows bit 7 cleared to 0. This bit is the *page-size* bit (PS), and specifies a 4-Kbyte physical-page translation.

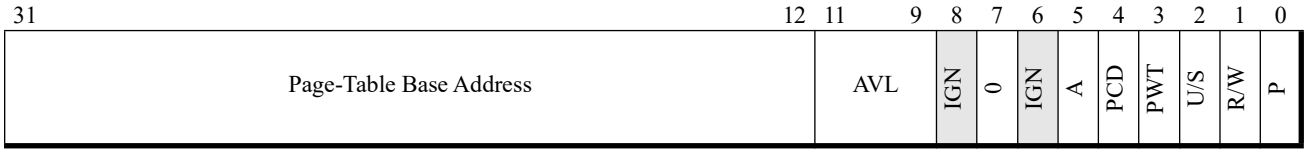


Figure 5-5. 4-Kbyte PDE—Non-PAE Paging Legacy-Mode

[AMD Public Use]

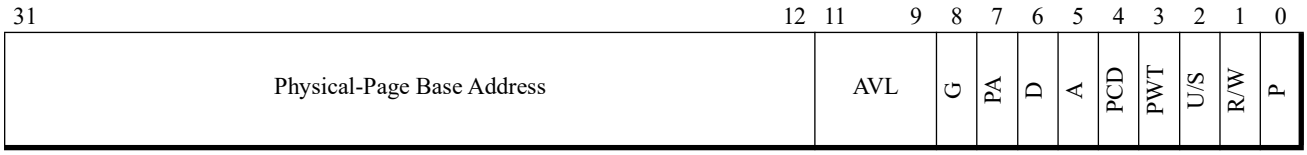


Figure 5-6. 4-Kbyte PTE—Non-PAE Paging Legacy-Mode

4-Mbyte Page Translation. 4-Mbyte page translation is only supported when page-size extensions are enabled (CR4.PSE=1) and physical-address extensions are disabled (CR4.PAE=0).

PSE defines a page-size bit in the 32-bit PDE format (PDE.PS). This bit is used by the processor during page translation to support both 4-Mbyte and 4-Kbyte pages. 4-Mbyte pages are selected when PDE.PS is set to 1, and the PDE points directly to a 4-Mbyte physical page. PTEs are not used in a 4-Mbyte page translation. If PDE.PS is cleared to 0, or if 4-Mbyte page translation is disabled, the PDE points to a PTE.

4-Mbyte page translation is performed by dividing the 32-bit virtual address into two fields. Each field is used as an index into a single-level page-translation hierarchy. The virtual-address fields are used as follows, and are shown in Figure 5-7 on page 137:

- Bits 31:22 index into the 1024-entry page-directory table.
- Bits 21:0 provide the byte offset into the physical page.

[AMD Public Use]

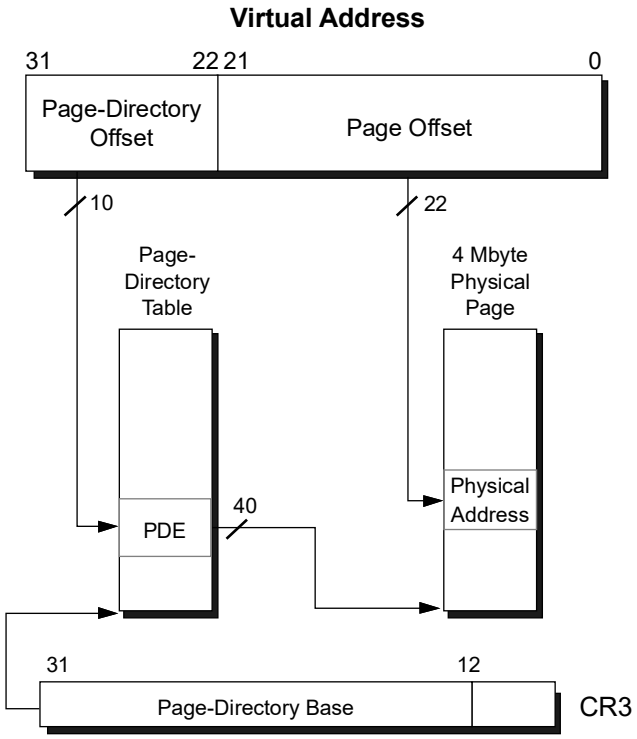


Figure 5-7. 4-Mbyte Page Translation—Non-PAE Paging Legacy-Mode

The AMD64 architecture modifies the legacy 32-bit PDE format in PSE mode to increase physical-address size support to 40 bits. This increase in address size is accomplished by using bits 20:13 to hold eight additional high-order physical-address bits. Bit 21 is reserved and must be cleared to 0.

Figure 5-8 shows the format of the PDE when PSE mode is enabled. The physical-page base-address bits are contained in a split field. The high-order, physical-page base-address bits 39:32 are located in PDE[20:13], and physical-page base-address bits 31:22 are located in PDE[31:22].

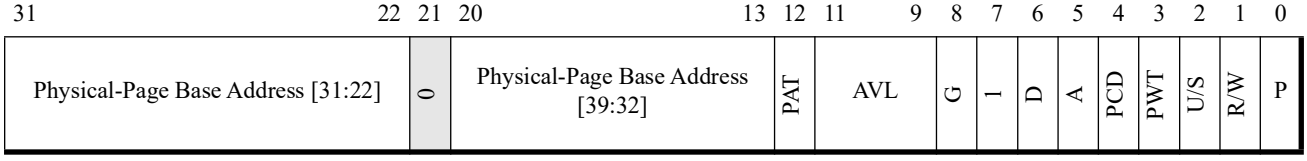


Figure 5-8. 4-Mbyte PDE—Non-PAE Paging Legacy-Mode

5.2.3 PAE Paging

PAE paging is used when physical-address extensions are enabled (CR4.PAE=1). PAE paging doubles the size of page-translation table entries to 64 bits so that the table entries can hold larger physical

[AMD Public Use]

addresses (up to 52 bits). The size of each table remains 4 Kbytes, which means each table can hold 512 of the 64-bit entries. PAE paging also introduces a third-level page-translation table, known as the page-directory-pointer table (PDPT).

The size of large pages in PAE-paging mode is 2 Mbytes rather than 4 Mbytes. PAE uses the page-directory page-size bit (PDE.PS) to allow selection between 4-Kbyte and 2-Mbyte page sizes. PAE automatically uses the page-size bit, so the value of CR4.PSE is ignored by PAE paging.

4-Kbyte Page Translation. With PAE paging, 4-Kbyte physical-page translation is performed by dividing the 32-bit virtual address into four fields, each of the upper three fields is used as an index into a 3-level page-translation hierarchy. The virtual-address fields are described as follows and are shown in Figure 5-9:

- Bits 31:30 index into a 4-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:12 index into the 512-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

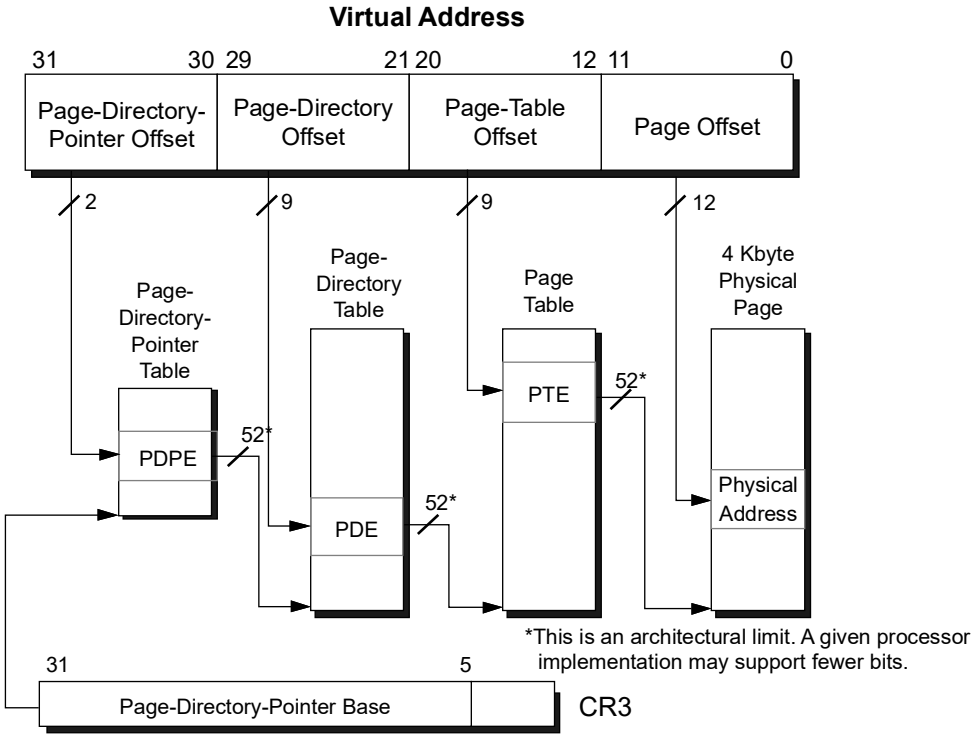


Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode

Figures 5-10 through 5-12 show the legacy-mode 4-Kbyte translation-table formats:

[AMD Public Use]

- Figure 5-10 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-11 shows the PDE (page-directory entry) format.
- Figure 5-12 shows the PTE (page-table entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 155.

Figure 5-11 shows the PDE.PS bit cleared to 0 (bit 7), specifying a 4-Kbyte physical-page translation.

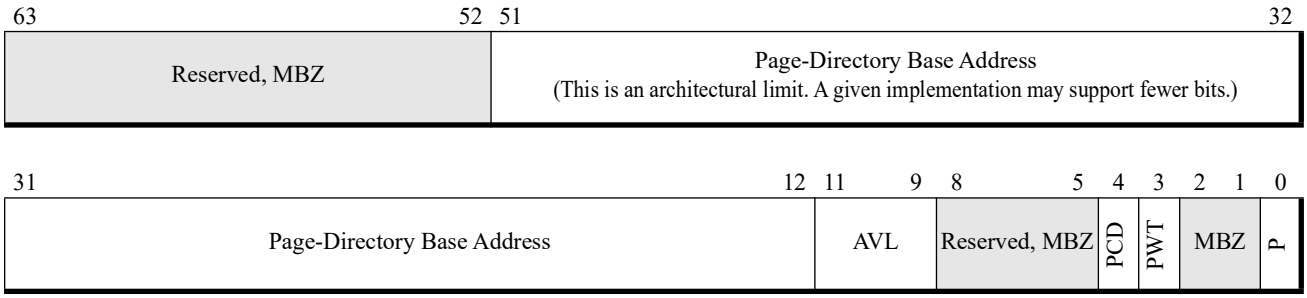


Figure 5-10. 4-Kbyte PDPE—PAE Paging Legacy-Mode

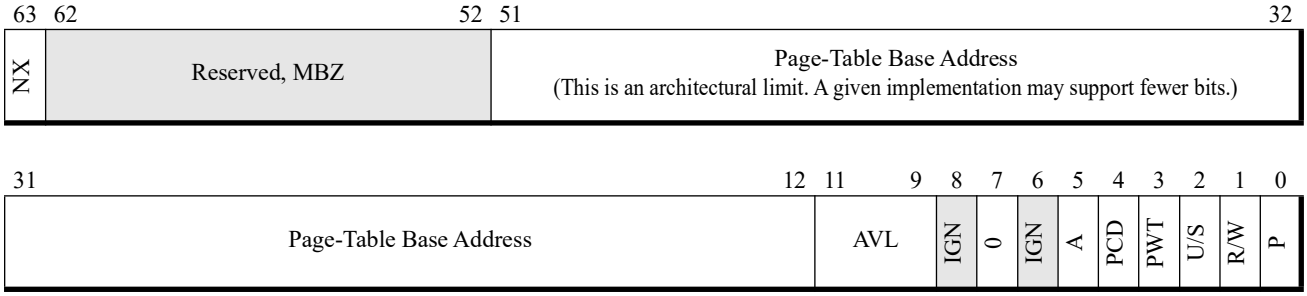


Figure 5-11. 4-Kbyte PDE—PAE Paging Legacy-Mode

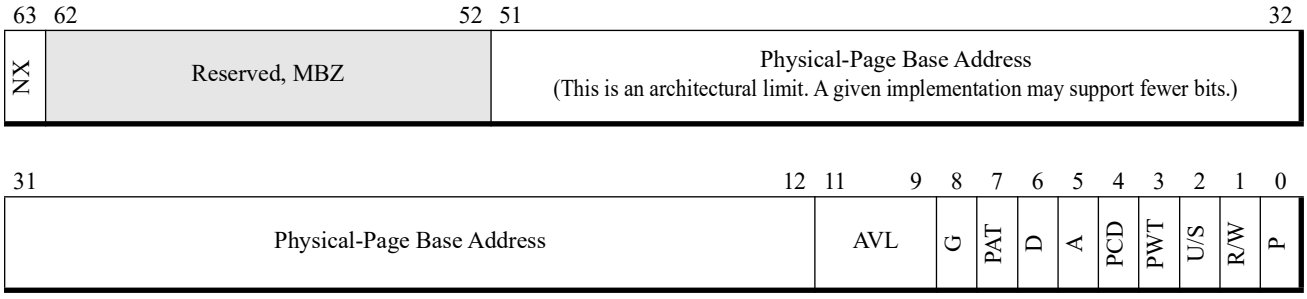


Figure 5-12. 4-Kbyte PTE—PAE Paging Legacy-Mode

[AMD Public Use]

2-Mbyte Page Translation. 2-Mbyte page translation is performed by dividing the 32-bit virtual address into three fields. Each field is used as an index into a 2-level page-translation hierarchy. The virtual-address fields are described as follows and are shown in Figure 5-13 on page 140:

- Bits 31:30 index into the 4-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:0 provide the byte offset into the physical page.

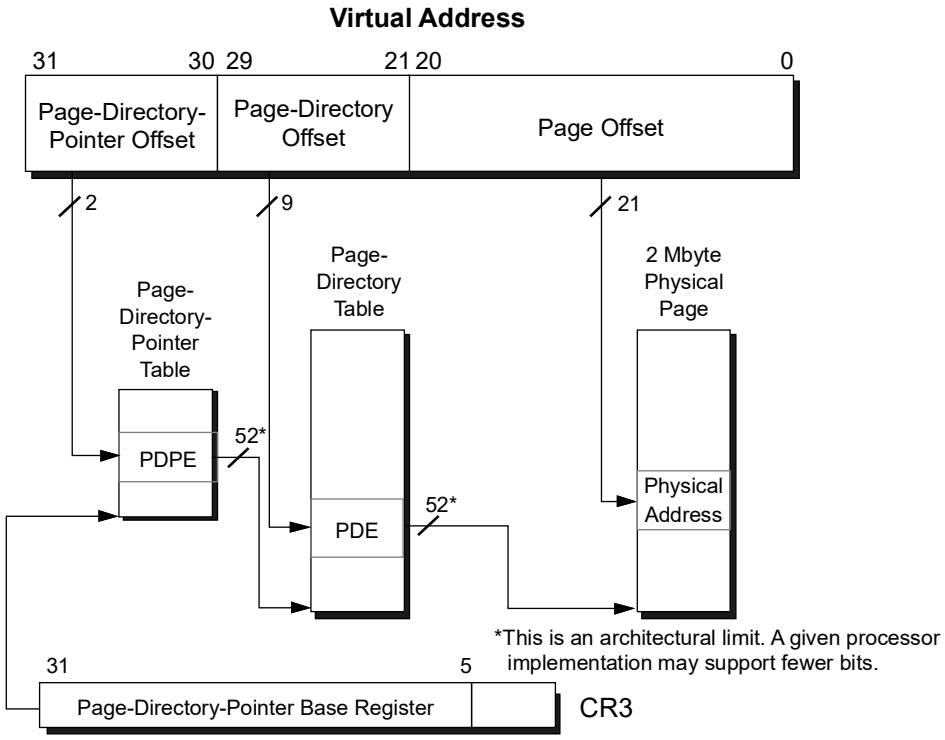


Figure 5-13. 2-Mbyte PAE Page Translation—Legacy Mode

Figure 5-14 shows the format of the PDPE (page-directory-pointer entry) and Figure 5-15 on page 141 shows the format of the PDE (page-directory entry). PTEs are not used in 2-Mbyte page translations.

Figure 5-15 on page 141 shows the PDE.PS bit set to 1 (bit 7), specifying a 2-Mbyte physical-page translation.

[AMD Public Use]

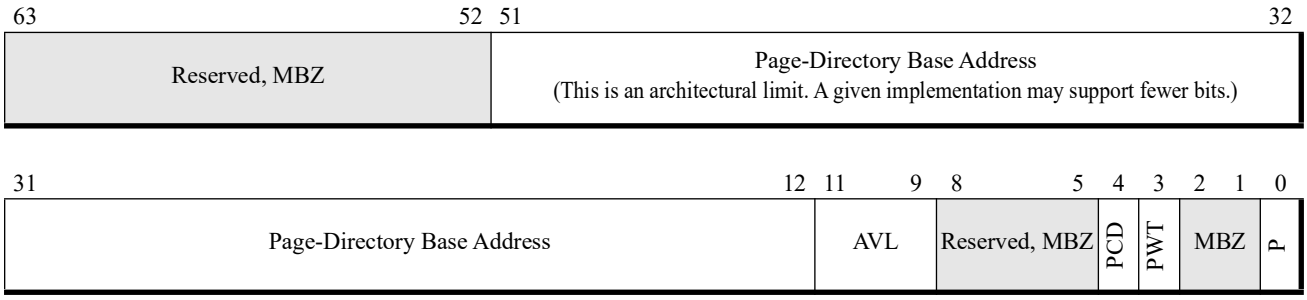


Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode

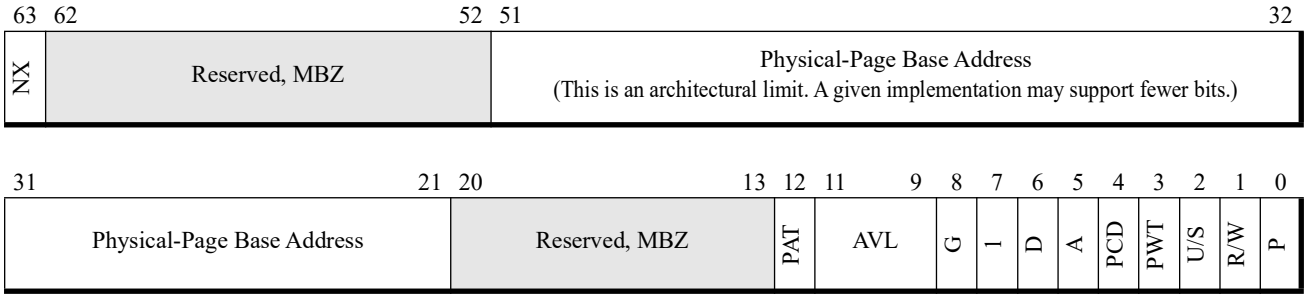


Figure 5-15. 2-Mbyte PDE—PAE Paging Legacy-Mode

5.3 Long-Mode Page Translation

Long-mode page translation requires the use of physical-address extensions (PAE). Before activating long mode, PAE must be enabled by setting CR4.PAE to 1. Activating long mode before enabling PAE causes a general-protection exception (#GP) to occur.

The PAE-paging data structures support mapping of 64-bit virtual addresses into 52-bit physical addresses. PAE expands the size of legacy page-directory entries (PDEs) and page-table entries (PTEs) from 32 bits to 64 bits, allowing physical-address sizes of greater than 32 bits.

The AMD64 architecture enhances the page-directory-pointer entry (PDPE) by defining previously reserved bits for access and protection control. A new translation table is added to PAE paging, called the page-map level-4 (PML4). The PML4 table precedes the PDP table in the page-translation hierarchy.

Because PAE is always enabled in long mode, the PS bit in the page directory entry (PDE.PS) selects between 4-Kbyte and 2-Mbyte page sizes, and the CR4.PSE bit is ignored. When 1-Gbyte pages are supported, the PDPE. PS bit selects the 1-Gbyte page size.

Some processors, identified by CPUID Fn0000_0007_ECX[LA57] (bit 16) = 1, support an additional translation table called page-map level-5 (PML5). The PML5 table precedes the PML4 table and allows translation of up to 57 bits of virtual address. 5-Level paging is enabled by setting

[AMD Public Use]

CR4[LA57]=1 when EFER[LMA]=1. CR4[LA57] is ignored when long mode is not active (EFER[LMA]=0).

5.3.1 Canonical Address Form

The AMD64 architecture requires implementations supporting fewer than the full 64-bit virtual address to ensure that those addresses are in canonical form. An address is in canonical form if the address bits from the most-significant implemented bit up to bit 63 are all ones or all zeros. If the addresses of all bytes in a virtual-memory reference are not in canonical form, the processor generates a general-protection exception (#GP) or a stack fault (#SS) as appropriate.

5.3.2 CR3

In long mode, the CR3 register is used to point to the PML4 base address when 5-Level paging is disabled (CR4[LA57]=0) or the PML5 base address when 5-Level paging is enabled (CR4[LA57]=1). CR3 is expanded to 64 bits in long mode, allowing the PML4 or PML5 table to be located anywhere in the 52-bit physical-address space. Figure 5-16 on page 142 shows the long-mode CR3 format.

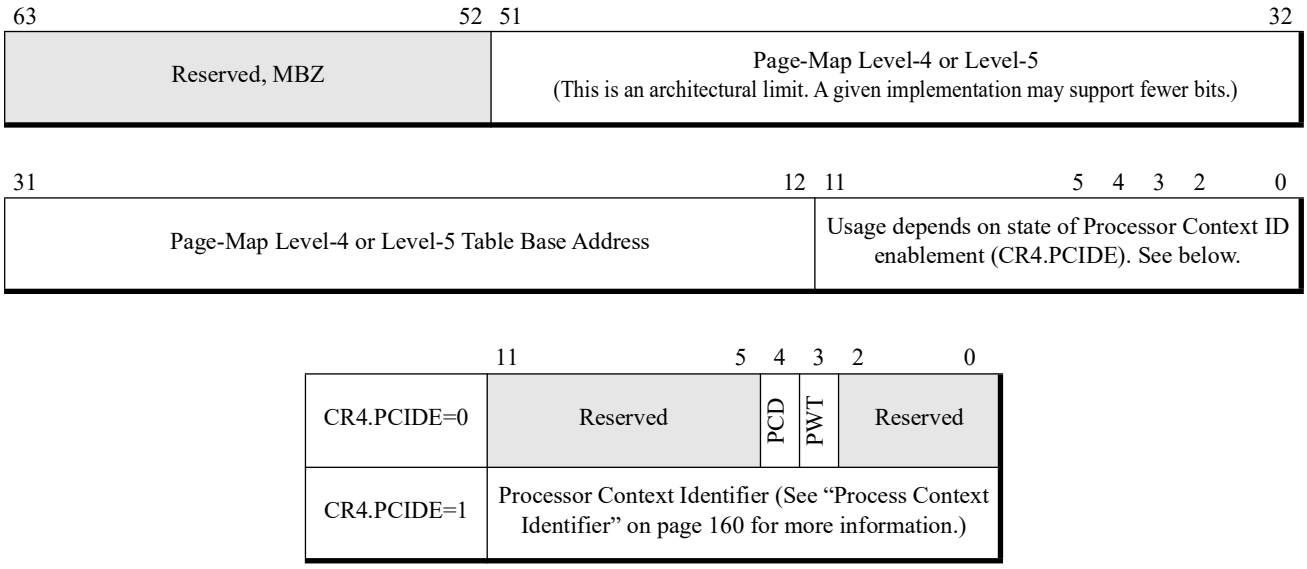


Figure 5-16. Control Register 3 (CR3)—Long Mode

Note: *When PCID is enabled, bit 63 is set to 1 to enable PCID for this linear space as discussed in section 5.5.1. It is no longer reserved.*

The CR3 register fields for long mode are:

Table Base Address Field. Bits 51:12. When 5-Level paging is enabled (CR4.LA57=1) this 40-bit field points to the PML5 base address. Otherwise it points to the PML4 base address. The PML4 or PML5 table is aligned on a 4-Kbyte boundary with the low-order 12 address bits (11:0) assumed to be 0. This yields a total base-address size of 52 bits. System software running on processor

[AMD Public Use]

implementations supporting less than the full 52-bit physical-address space must clear the unimplemented upper base-address bits to 0.

Page-Level Writethrough (PWT) Bit. Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

Page-Level Cache Disable (PCD) Bit. Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

Process Context Identifier. Bits 11:0. This 12-bit field determines the current Processor Context Identifier (PCID) when CR4.PCIDE=1.

Reserved Bits. Reserved fields should be cleared to 0 by software when writing CR3.

5.3.3 4-Kbyte Page Translation

In long mode, 4-Kbyte physical-page translation is performed by dividing the virtual address into up to seven fields. Up to four of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-17 on page 144 (4-Kbyte Page Translation—Long Mode 4-Level Paging) and Figure 5-18 on page 145 (4-Kbyte Page Translation—Long Mode 5-Level Paging):

- Bits 63:57 are a sign extension of bit 56, as required for canonical-address forms.
- Bits 56:48 index into the 512-entry page-map level-5 table. (Only when 5-Level paging is enabled).
- Bits 47:39 index into the 512-entry page-map level-4 table.
- Bits 38:30 index into the 512-entry page-directory pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:12 index into the 512-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

Note: *The sizes of the sign extension, PML5, and the PML4 fields depend on the number of virtual address bits supported by the implementation.*

[AMD Public Use]

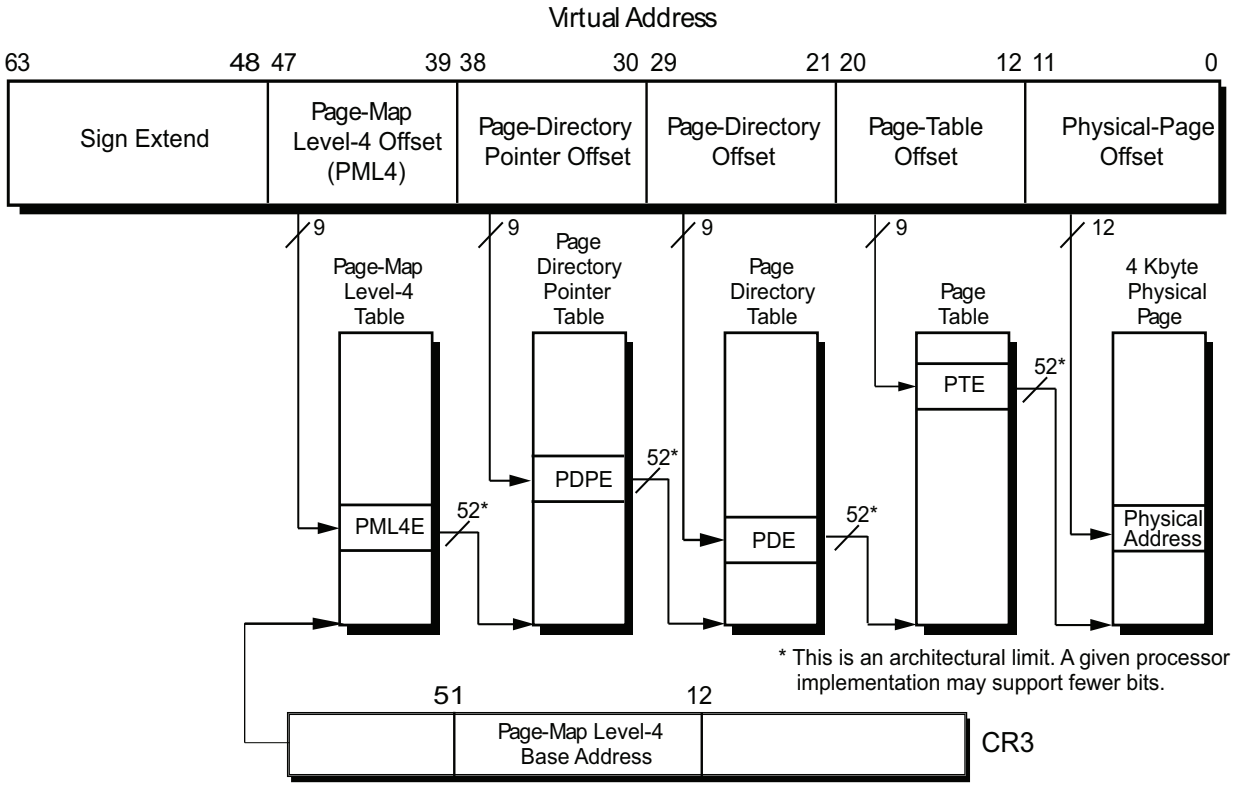


Figure 5-17. 4-Kbyte Page Translation—Long Mode 4-Level Paging

[AMD Public Use]

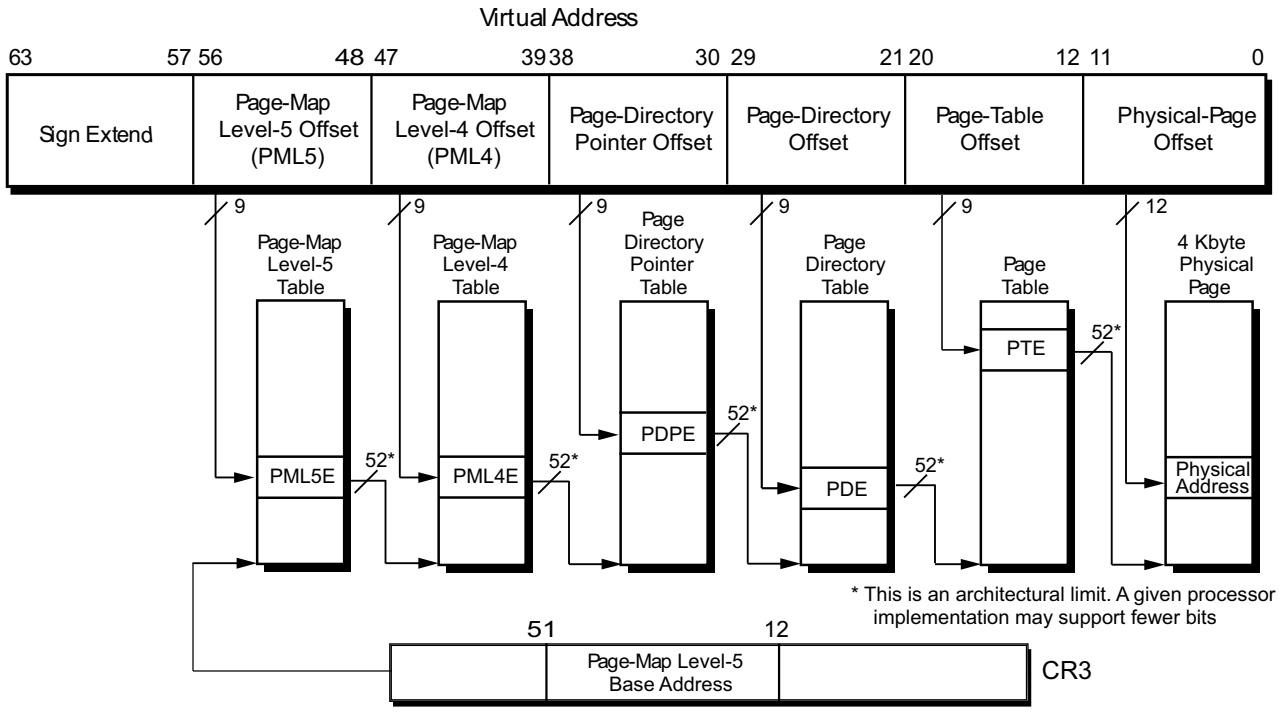


Figure 5-18. 4-Kbyte Page Translation—Long Mode 5-Level Paging

Figures 5-19 through 5-22 on page 146 and Figure 5-23 on page 147 show the long-mode 4-Kbyte translation-table formats:

- Figure 5-19 on page 146 shows the PML5E (page-map level-5 entry) format.
- Figure 5-20 on page 146 shows the PML4E (page-map level-4 entry) format.
- Figure 5-21 on page 146 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-22 on page 146 shows the PDE (page-directory entry) format.
- Figure 5-23 on page 147 shows the PTE (page-table entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 155.

Figure 5-22 on page 146 shows the PDE.PS bit (bit 7) cleared to 0, indicating a 4-Kbyte physical-page translation.

[AMD Public Use]

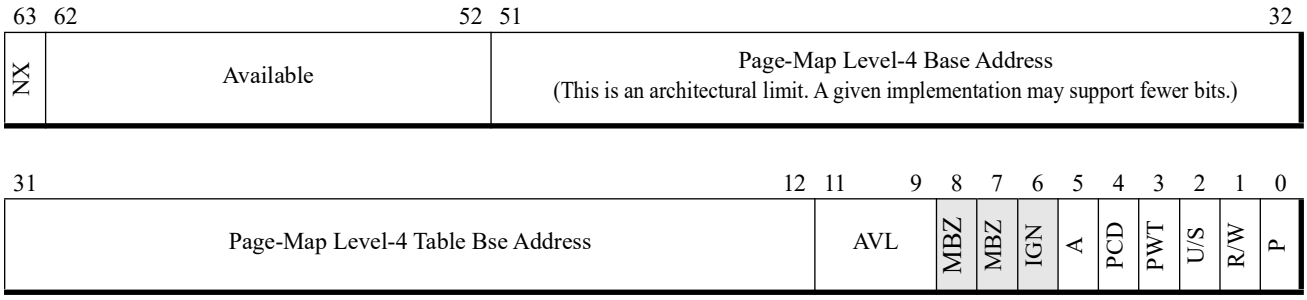


Figure 5-19. 4-Kbyte PML5E—Long Mode

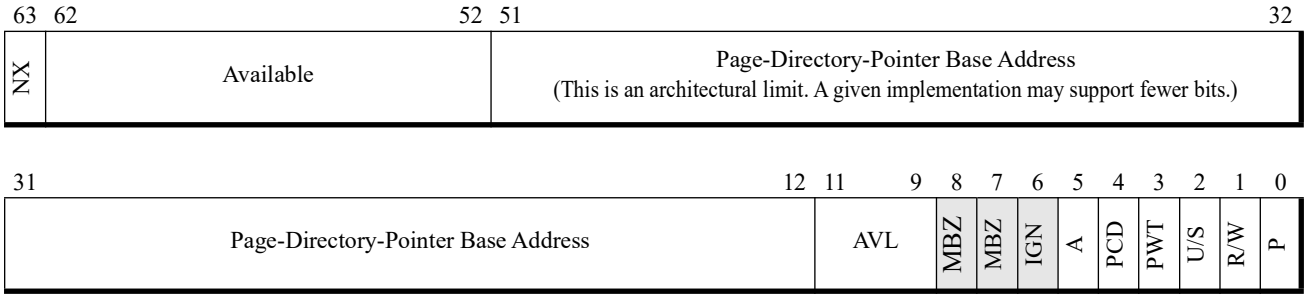


Figure 5-20. 4-Kbyte PML4E—Long Mode

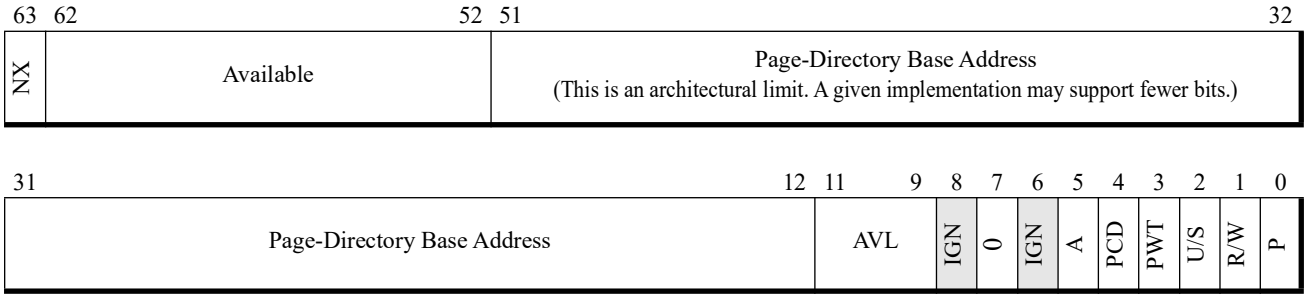


Figure 5-21. 4-Kbyte PDPE—Long Mode

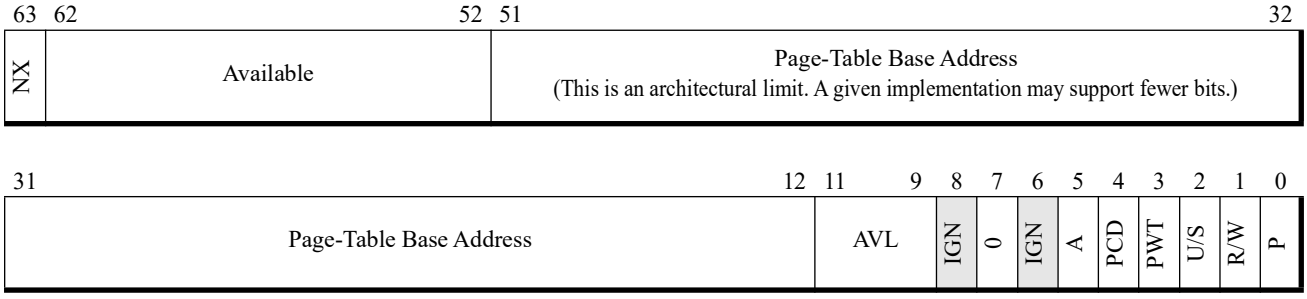


Figure 5-22. 4-Kbyte PDE—Long Mode

[AMD Public Use]

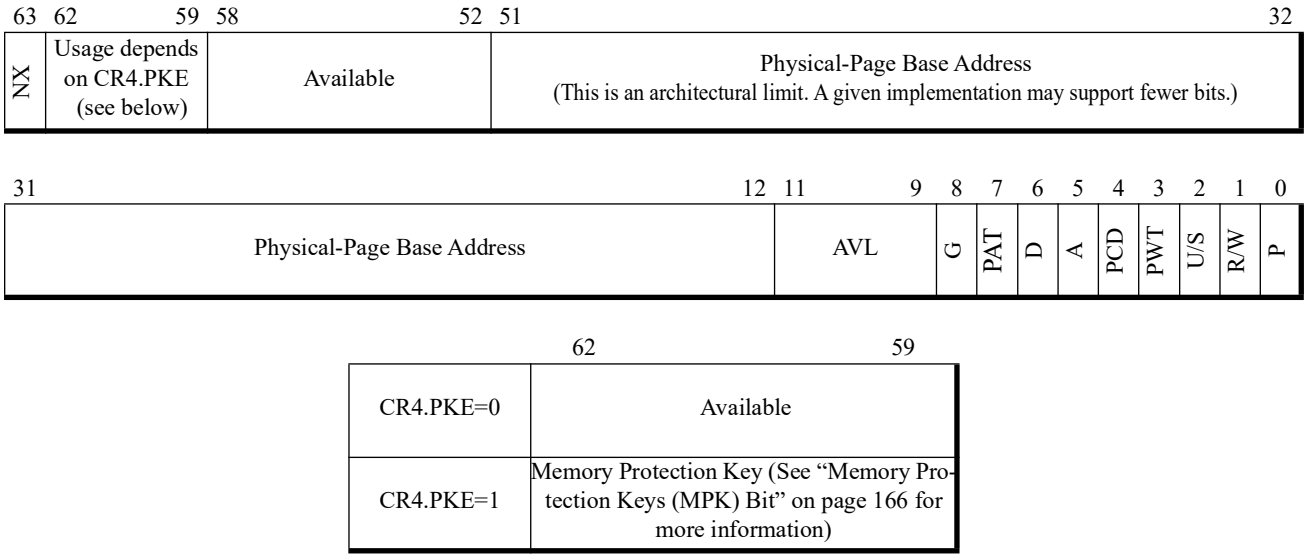


Figure 5-23. 4-Kbyte PTE—Long Mode

5.3.4 2-Mbyte Page Translation

In long mode, 2-Mbyte physical-page translation is performed by dividing the virtual address into up to six fields. Up to four of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-24 on page 148 (2-Mbyte Page Translation—Long Mode 4-Level Paging) and Figure 5-25 on page 149 (2-Mbyte Page Translation—Long Mode 5-Level Paging):

- Bits 63:57 are a sign extension of bit 56 as required for canonical address forms.
- Bits 56:48 index into the 512-entry page-map level-5 table (only when 5-Level paging is enabled).
- Bits 47:39 index into the 512-entry page-map level-4 table.
- Bits 38:30 index into the 512-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:0 provide the byte offset into the physical page.

[AMD Public Use]

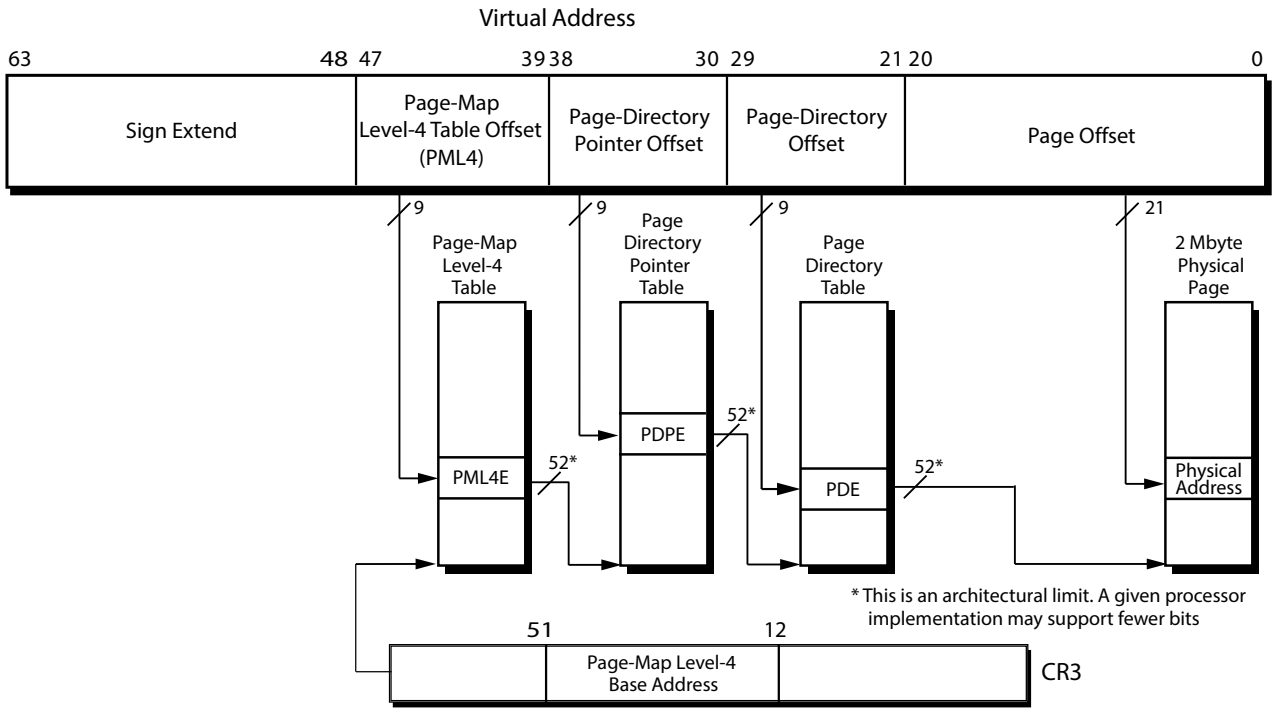


Figure 5-24. 2-Mbyte Page Translation—Long Mode 4-Level Paging

[AMD Public Use]

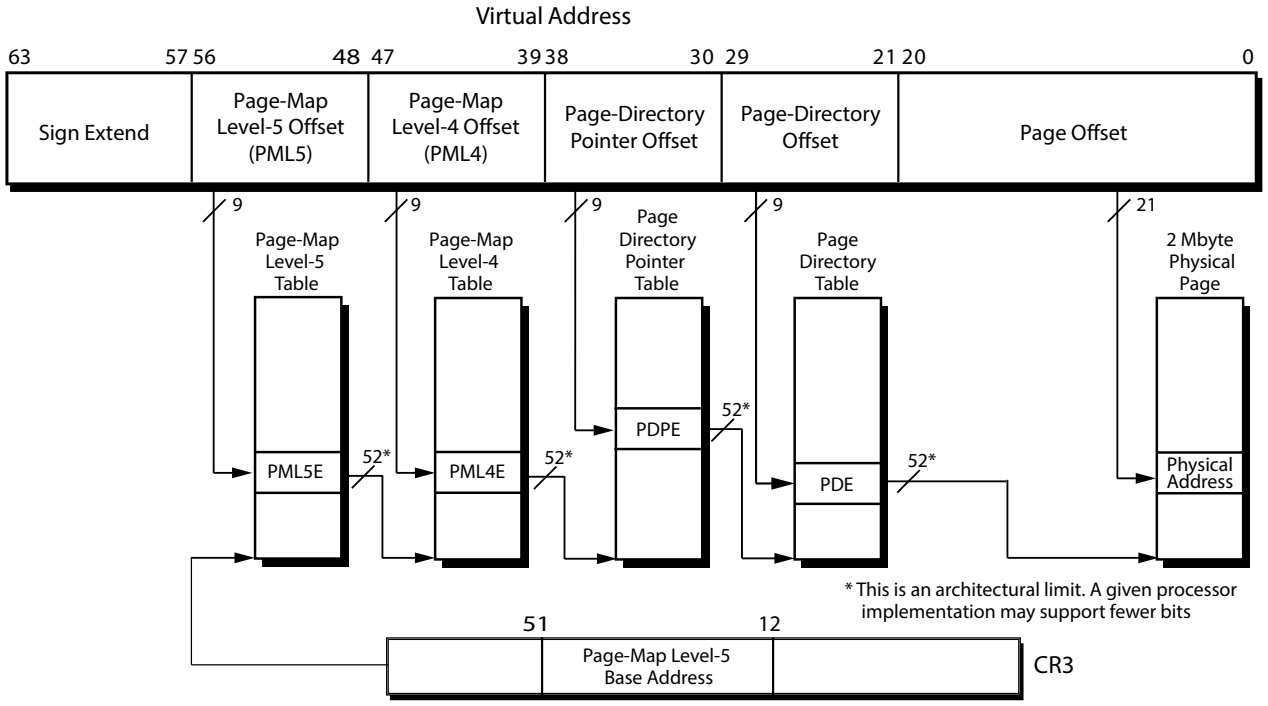


Figure 5-25. 2-Mbyte Page Translation—Long Mode 5-Level Paging

Figures 5-26 through 5-29 on page 151 show the long-mode 2-Mbyte translation-table formats (the PML4 and PDPT formats are identical to those used for 4-Kbyte page translations and are repeated here for clarity):

- Figure 5-26 on page 150 shows the PML5E (page-map level-5 entry) format.
- Figure 5-27 on page 150 shows the PML4E (page-map level-4 entry) format.
- Figure 5-28 on page 150 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-29 on page 151 shows the PDE (page-directory entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 155. PTEs are not used in 2-Mbyte page translations.

Figure 5-29 shows the PDE.PS bit (bit 7) set to 1, indicating a 2-Mbyte physical-page translation.

[AMD Public Use]

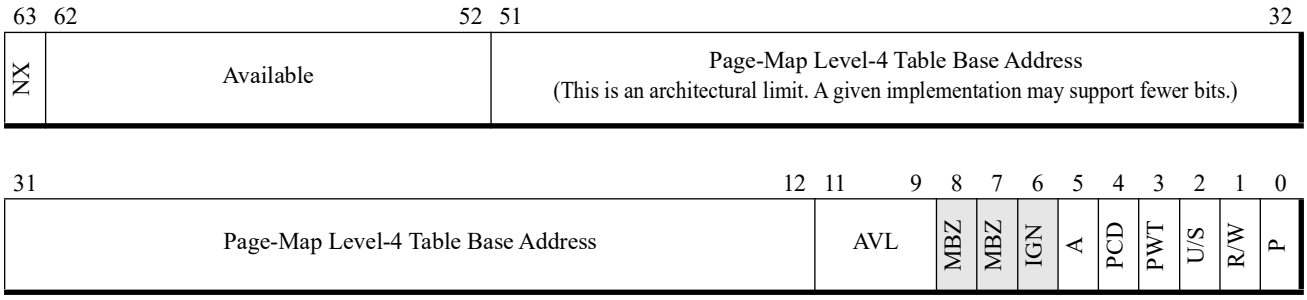


Figure 5-26. 2-Mbyte PML5E—Long Mode

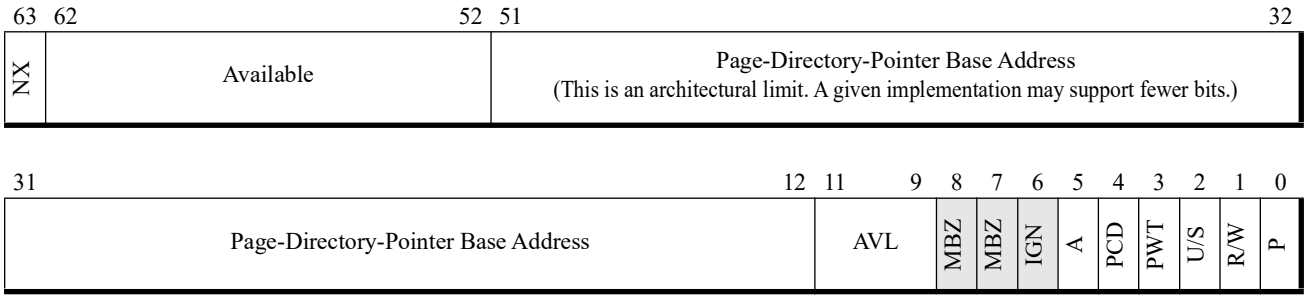


Figure 5-27. 2-Mbyte PML4E—Long Mode

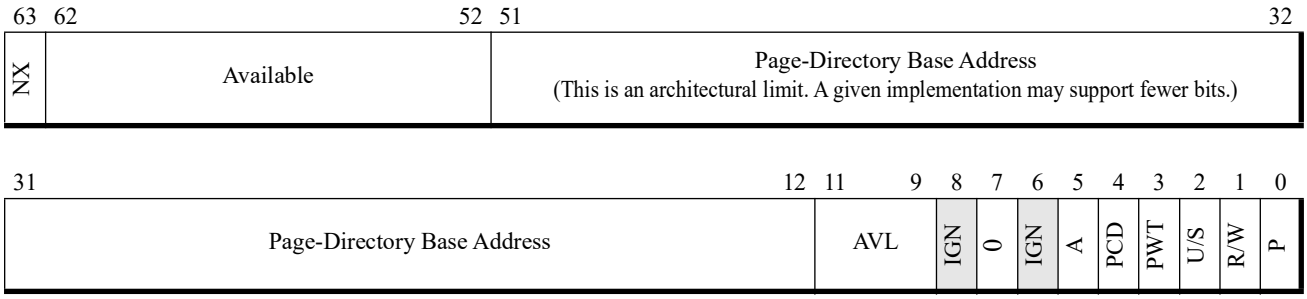


Figure 5-28. 2-Mbyte PDPE—Long Mode

[AMD Public Use]

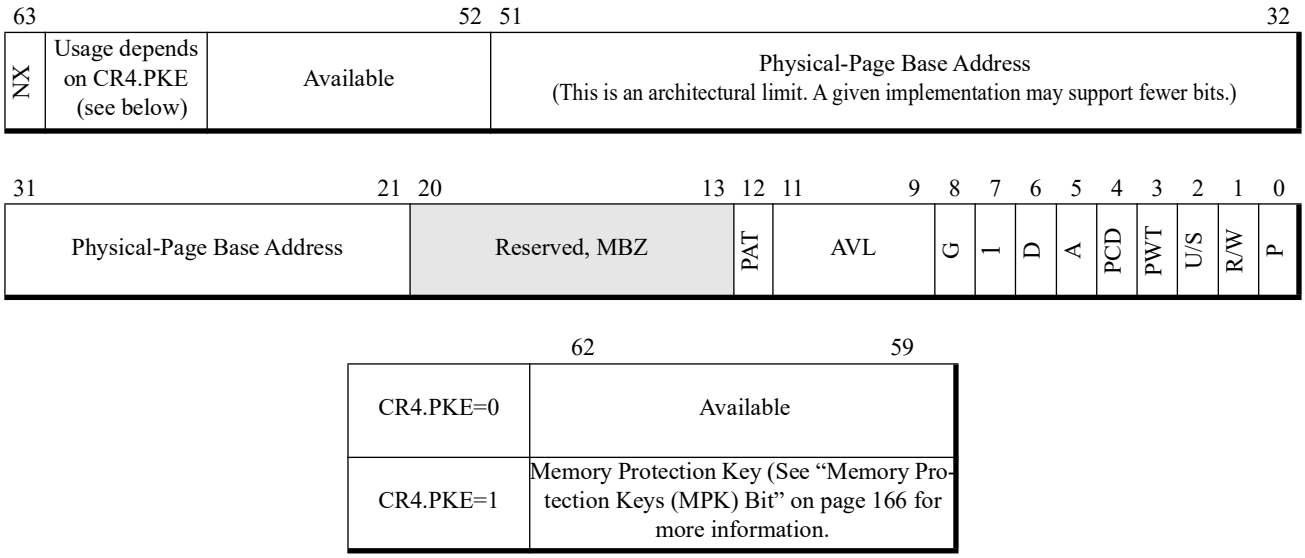


Figure 5-29. 2-Mbyte PDE—Long Mode

5.3.5 1-Gbyte Page Translation

In long mode, 1-Gbyte physical-page translation is performed by dividing the virtual address into up to five fields. Up to three of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-30 on page 152 (1-Gbyte Page Translation—Long Mode 4-Level Paging) and Figure 5-31 on page 153 (1-Gbyte Page Translation—Long Mode 5-Level Paging):

- Bits 63:57 are a sign extension of bit 56 as required for canonical address forms.
- Bits 56:48 index into the 512-entry page-map level-5 table. (Only when 5-Level paging is enabled)
- Bits 47:39 index into the 512-entry page-map level-4 table.
- Bits 38:30 index into the 512-entry page-directory-pointer table.
- Bits 29:0 provide the byte offset into the physical page.

[AMD Public Use]

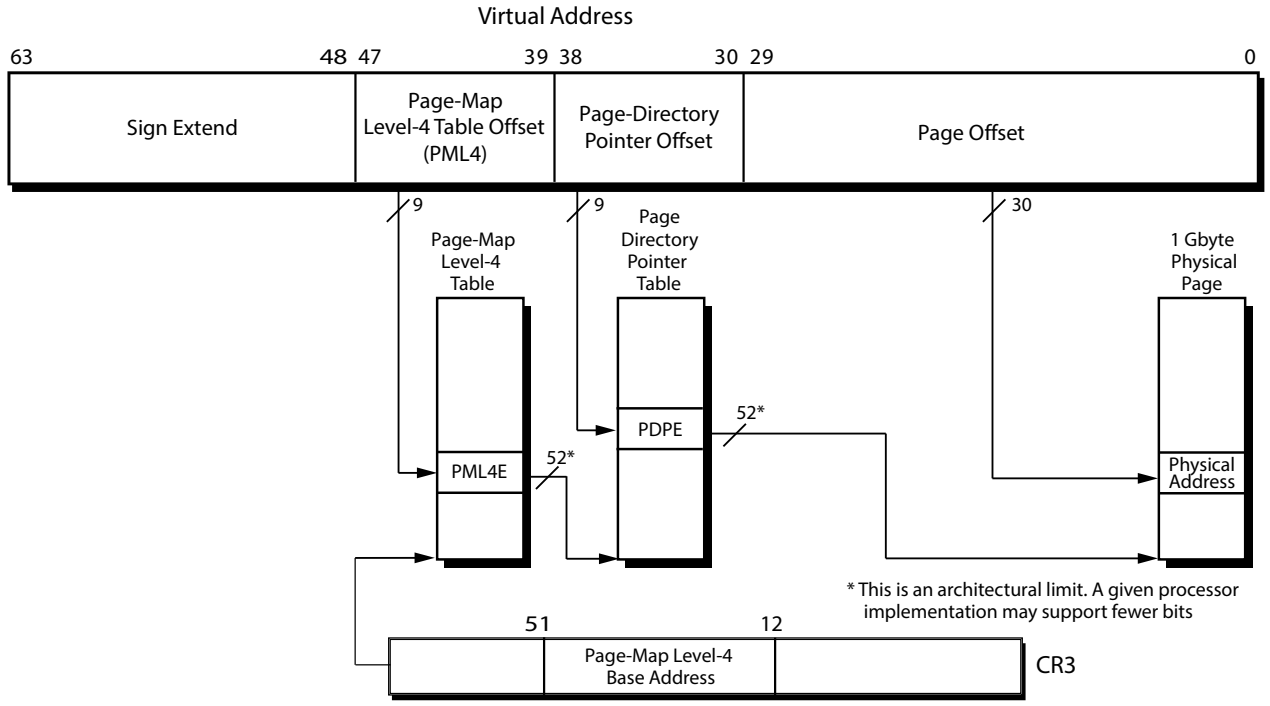


Figure 5-30. 1-Gbyte Page Translation—Long Mode 4-Level Paging

[AMD Public Use]

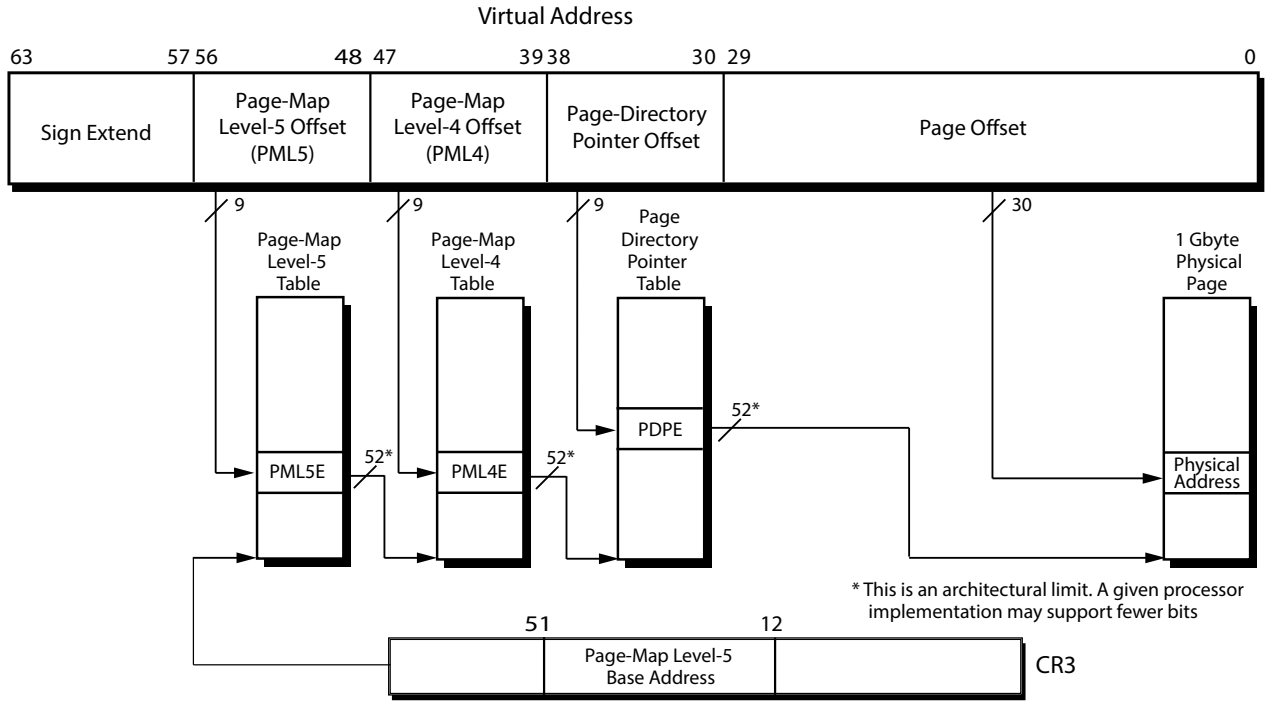


Figure 5-31. 1-Gbyte Page Translation—Long Mode 5-Level Paging

Figure 5-32 to Figure 5-34 show the long mode 1-Gbyte translation-table formats (the PML4 and PML5 formats are identical to the ones used for 4-Kbyte page translations and are repeated here for clarity):

- Figure 5-32 shows the PML5E (page-map level-5 entry) format.
- Figure 5-33 shows the PML4E (page-map level-4 entry) format.
- Figure 5-34 shows the PDPE (page-directory-pointer entry) format.

The fields within these table entries are described in Section 5.4 “Page-Translation-Table Entry Fields” on page 155 in the current volume. PTEs and PDEs are not used in 1-Gbyte page translations.

Figure 5-34 on page 155 shows the PDPE.PS bit (bit 7) set to 1, indicating a 1-Gbyte physical-page translation.

[AMD Public Use]

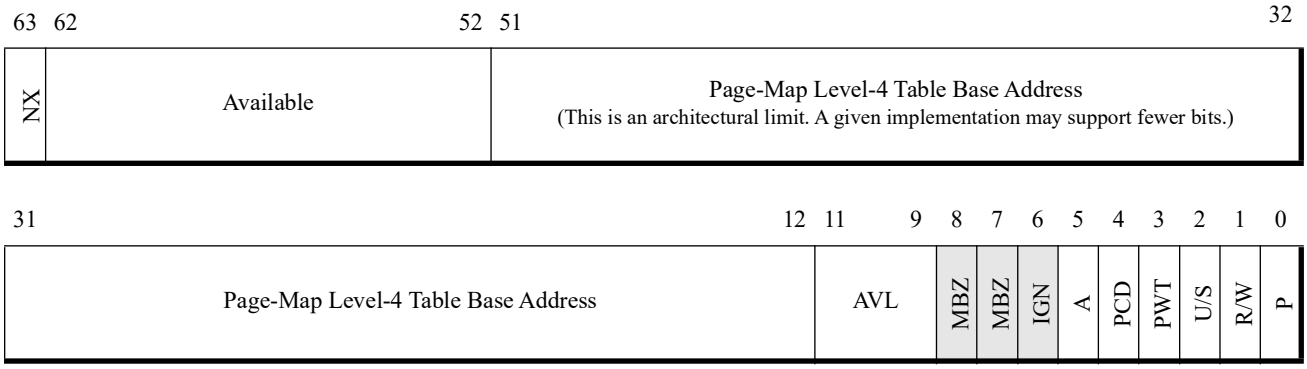


Figure 5-32. 1-Gbyte PML5E—Long Mode

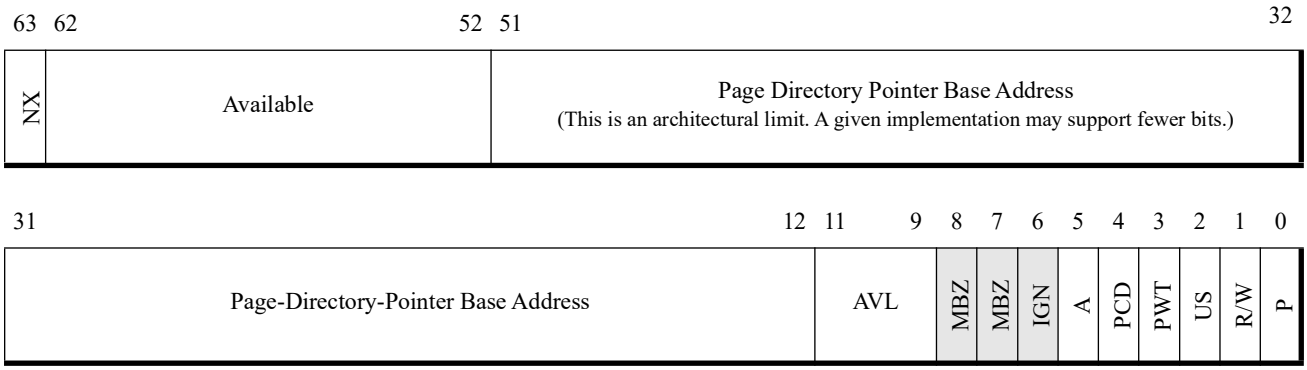


Figure 5-33. 1-Gbyte PML4E—Long Mode

[AMD Public Use]

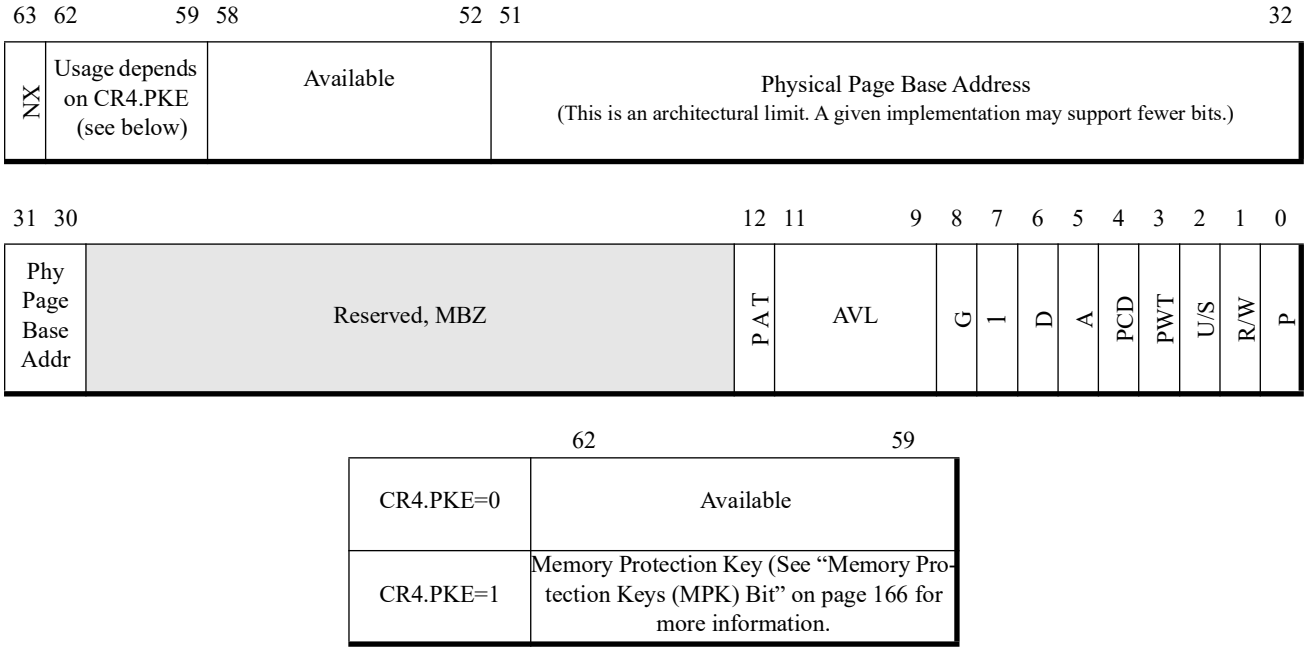


Figure 5-34. 1-Gbyte PDPE—Long Mode

1-Gbyte Paging Feature Identification. EDX bit 26 as returned by CPUID function 8000_0001h indicates 1-Gbyte page support. The EAX register as returned by CPUID function 8000_0019h reports the number of 1-Gbyte L1 TLB entries supported and EBX reports the number of 1-Gbyte L2 TLB entries. For more information using the CPUID instruction see Section 3.3 "Processor Feature Identification" on page 73.

5.4 Page-Translation-Table Entry Fields

The page-translation-table entries contain control and informational fields used in the management of the virtual-memory environment. Most fields are common across all translation table entries and modes and occupy the same bit locations. However, some fields are located in different bit positions depending on the page translation hierarchical level, and other fields have different sizes depending on which physical-page size, physical-address size, and operating mode are selected. Although these fields can differ in bit position or size, their meaning is consistent across all levels of the page translation hierarchy and in all operating modes.

5.4.1 Field Definitions

The following sections describe each field within the page-translation table entries.

Translation-Table Base Address Field. The translation-table base-address field points to the physical base address of the next-lower-level table in the page-translation hierarchy. Page data-structure tables are always aligned on 4-Kbyte boundaries, so only the address bits above bit 11 are

[AMD Public Use]

stored in the translation-table base-address field. Bits 11:0 are assumed to be 0. The size of the field depends on the mode:

- In non-PAE paging (CR4.PAE=0), this field specifies a 32-bit physical address.
- In PAE paging (CR4.PAE=1), this field specifies a 52-bit physical address.

52 bits correspond to the maximum physical-address size allowed by the AMD64 architecture. If a processor implementation supports fewer than the full 52-bit physical address, software must clear the unimplemented high-order translation-table base-address bits to 0. For example, if a processor implementation supports a 40-bit physical-address size, software must clear bits 51:40 when writing a translation-table base-address field in a page data-structure entry.

Physical-Page Base Address Field. The physical-page base-address field points to the base address of the translated physical page. This field is found only in the lowest level of the page-translation hierarchy. The size of the field depends on the mode:

- In non-PAE paging (CR4.PAE=0), this field specifies a 32-bit base address for a physical page.
- In PAE paging (CR4.PAE=1), this field specifies a 52-bit base address for a physical page.

Physical pages can be 4 Kbytes, 2 Mbytes, 4 Mbytes, or 1-Gbyte and they are always aligned on an address boundary corresponding to the physical-page length. For example, a 2-Mbyte physical page is always aligned on a 2-Mbyte address boundary. Because of this alignment, the low-order address bits are assumed to be 0, as follows:

- 4-Kbyte pages, bits 11:0 are assumed 0.
- 2-Mbyte pages, bits 20:0 are assumed 0.
- 4-Mbyte pages, bits 21:0 are assumed 0.
- 1-Gbyte pages, bits 29:0 are assumed 0.

Present (P) Bit. Bit 0. This bit indicates whether the page-translation table or physical page is loaded in physical memory. When the P bit is cleared to 0, the table or physical page is not loaded in physical memory. When the P bit is set to 1, the table or physical page is loaded in physical memory.

Software clears this bit to 0 to indicate a page table or physical page is not loaded in physical memory. A page-fault exception (#PF) occurs if an attempt is made to access a table or page when the P bit is 0. System software is responsible for loading the missing table or page into memory and setting the P bit to 1.

When the P bit is 0, indicating a not-present page, all remaining bits in the page data-structure entry are available to software.

Entries with P cleared to 0 are never cached in TLB nor will the processor set the Accessed or Dirty bit for the table entry.

Read/Write (R/W) Bit. Bit 1. This bit controls read/write access to all physical pages mapped by the table entry. For example, a page-map level-4 R/W bit controls read/write access to all 128M ($512 \times 512 \times 512$) physical pages it maps through the lower-level translation tables. When the R/W bit

[AMD Public Use]

is cleared to 0, access is restricted to read-only. When the R/W bit is set to 1, both read and write access is allowed. See “Page-Protection Checks” on page 163 for a description of the paging read/write protection mechanism.

User/Supervisor (U/S) Bit. Bit 2. This bit controls user (CPL 3) access to all physical pages mapped by the table entry. For example, a page-map level-4 U/S bit controls the access allowed to all 128M (512 × 512 × 512) physical pages it maps through the lower-level translation tables. When the U/S bit is cleared to 0, access is restricted to supervisor level (CPL 0, 1, 2). When the U/S bit is set to 1, both user and supervisor access is allowed. See “Page-Protection Checks” on page 163 for a description of the paging user/supervisor protection mechanism.

Page-Level Writethrough (PWT) Bit. Bit 3. This bit indicates whether the page-translation table or physical page to which this entry points has a writeback or writethrough caching policy. When the PWT bit is cleared to 0, the table or physical page has a writeback caching policy. When the PWT bit is set to 1, the table or physical page has a writethrough caching policy. See “Memory Caches” on page 206 for additional information on caching. If the processor supports the Page-Attribute Table Mechanism, this bit is used in constructing an index into the PAT register. See Section 7.8 “Page-Attribute Table Mechanism” on page 229.

Page-Level Cache Disable (PCD) Bit. Bit 4. This bit indicates whether the page-translation table or physical page to which this entry points is cacheable. When the PCD bit is cleared to 0, the table or physical page is cacheable. When the PCD bit is set to 1, the table or physical page is not cacheable. See “Memory Caches” on page 206 for additional information on caching. If the processor supports the Page-Attribute Table Mechanism, this bit is used in constructing an index into the PAT register. See Section 7.8 “Page-Attribute Table Mechanism” on page 229.

Accessed (A) Bit. Bit 5. This bit indicates whether the page-translation table or physical page to which this entry points has been accessed. The A bit is set to 1 by the processor the first time the table or physical page is either read from or written to. The A bit is never cleared by the processor. Instead, software must clear this bit to 0 when it needs to track the frequency of table or physical-page accesses.

Dirty (D) Bit. Bit 6. This bit is only present in the lowest level of the page-translation hierarchy. It indicates whether the physical page to which this entry points has been written. The D bit is set to 1 by the processor the first time there is a write to the physical page. The D bit is never cleared by the processor. Instead, software must clear this bit to 0 when it needs to track the frequency of physical-page writes.

Page Size (PS) Bit. Bit 7. This bit is present in page-directory entries and long-mode page-directory-pointer entries. When the PS bit is set in the page-directory-pointer entry (PDPE) or page-directory entry (PDE), that entry is the lowest level of the page-translation hierarchy. When the PS bit is cleared to 0 in all levels above PTE, the lowest level of the page-translation hierarchy is the page-table entry (PTE), and the physical-page size is 4 Kbytes. The physical-page size is determined as follows:

- If EFER.LMA=1 and PDPE.PS=1, the physical-page size is 1 Gbyte.
- If CR4.PAE=0 and PDE.PS=1, the physical-page size is 4 Mbytes.
- If CR4.PAE=1 and PDE.PS=1, the physical-page size is 2 Mbytes.

[AMD Public Use]

See Table 5-1 on page 131 for a description of the relationship between the PS bit, PAE, physical-page sizes, and page-translation hierarchy.

Global Page (G) Bit. Bit 8. This bit is only present in the lowest level of the page-translation hierarchy. It indicates the physical page is a global page. The TLB entry for a global page (G=1) is not invalidated when CR3 is loaded either explicitly by a MOV CRn instruction or implicitly during a task switch. Use of the G bit requires the page-global enable bit in CR4 to be set to 1 (CR4.PGE=1). See "Global Pages" on page 160 for more information on the global-page mechanism.

Restart (R) Bit. Bit 11. This bit indicates page-translation restart for GAAT paging. For normal paging, this bit is available to software.

Available to Software (AVL) Bit. These bits are not interpreted by the processor and are available for use by system software.

Page-Attribute Table (PAT) Bit. This bit is only present in the lowest level of the page-translation hierarchy, as follows:

- If the lowest level is a PTE (PDE.PS=0), PAT occupies bit 7.
- If the lowest level is a PDE (PDE.PS=1) or PDPE (PDPE.PS=1), PAT occupies bit 12.

The PAT bit is the high-order bit of a 3-bit index into the PAT register (Figure 7-13 on page 229). The other two bits involved in forming the index are the PCD and PWT bits. Not all processors support the PAT bit by implementing the PAT registers. See "Page-Attribute Table Mechanism" on page 229 for a description of the PAT mechanism and how it is used.

Memory Protection Key (MPK) Bits. Bits 62:59. When Memory Protection Keys are enabled (CR4.PKE=1), this 4-bit field selects the memory protection key for the physical page mapped by this entry. Ignored if memory protection keys are disabled (CR4.PKE=0). (See "Memory Protection Keys (MPK) Bit" on page 166 for a description of this mechanism.)

No Execute (NX) Bit. Bit 63. This bit is present in the translation-table entries defined for PAE paging, with the exception that the legacy-mode PDPE does not contain this bit. This bit is not supported by non-PAE paging.

The NX bit can only be set when the no-execute page-protection feature is enabled by setting EFER.NXE to 1 (see "Extended Feature Enable Register (EFER)" on page 56). If EFER.NXE=0, the NX bit is treated as reserved. In this case, a page-fault exception (#PF) occurs if the NX bit is not cleared to 0.

This bit controls the ability to execute code from all physical pages mapped by the table entry. For example, a page-map level-4 NX bit controls the ability to execute code from all 128M ($512 \times 512 \times 512$) physical pages it maps through the lower-level translation tables. When the NX bit is cleared to 0, code can be executed from the mapped physical pages. When the NX bit is set to 1, code cannot be executed from the mapped physical pages. See "No Execute (NX) Bit" on page 158 for a description of the no-execute page-protection mechanism.

[AMD Public Use]

Reserved Bits. Software should clear all reserved bits to 0. If the processor is in long mode, or if page-size and physical-address extensions are enabled in legacy mode, a page-fault exception (#PF) occurs if reserved bits are not cleared to 0.

5.4.2 Notes on Accessed and Dirty Bits

The processor never sets the Accessed bit or the Dirty bit for a not present page (P = 0). The ordering of Accessed and Dirty bit updates with respect to surrounding loads and stores is discussed below.

Accessed (A) Bit. The Accessed bit can be set for instructions that are speculatively executed by the processor.

For example, the Accessed bit may be set by instructions in a mispredicted branch path even though those instructions are never retired. Thus, software must not assume that the TLB entry has not been cached in the TLB, just because no instruction that accessed the page was successfully retired. Nevertheless, a table entry is never cached in the TLB without its Accessed bit being set at the same time.

The processor does not order Accessed bit updates with respect to loads done by other instructions.

Dirty (D) Bit. The Dirty bit is not updated speculatively. For instructions with multiple writes, the D bit may be set for any writes completed up to the point of a fault. In rare cases, the Dirty bit may be set even if a write was not actually performed, including MASKMOVQ with a mask of zero and certain x87 floating point instructions that cause an exception. Thus software can not assume that the page has actually been written even where PTE[D] is set to 1.

If PTE[D] is cleared to 0, software can rely on the fact that the page has not been written.

In general, Dirty bit updates are ordered with respect to other loads and stores, although not necessarily with respect to accesses to WC memory; in particular, they may not cause WC buffers to be flushed. However, to ensure compatibility with future processors, a serializing operation should be inserted before reading the D bit.

5.5 Translation-Lookaside Buffer (TLB)

When paging is enabled, every memory access has its virtual address automatically translated into a physical address using the page-translation hierarchy. *Translation-lookaside buffers* (TLBs), also known as *page-translation caches*, nearly eliminate the performance penalty associated with page translation. TLBs are special on-chip caches that hold the most-recently used virtual-to-physical address translations. Each memory reference (instruction and data) is checked by the TLB. If the translation is present in the TLB, it is immediately provided to the processor, thus avoiding external memory references for accessing page tables.

TLBs take advantage of the *principle of locality.* That is, if a memory address is referenced, it is likely that nearby memory addresses will be referenced in the near future. In the context of paging, the proximity of memory addresses required for locality can be broad—it is equal to the page size. Thus, it

[AMD Public Use]

is possible for a large number of addresses to be translated by a small number of page translations. This high degree of locality means that almost all translations are performed using the on-chip TLBs.

System software is responsible for managing the TLBs when updates are made to the linear-to-physical mapping of addresses. A change to any paging data-structure entry is not automatically reflected in the TLB, and hardware snooping of TLBs during memory-reference cycles is not performed. Software must invalidate the TLB entry of a modified translation-table entry so that the change is reflected in subsequent address translations. TLB invalidation is described in “TLB Management” on page 161. Only privileged software running at CPL=0 can manage the TLBs.

5.5.1 Process Context Identifier

The Process Context Identifier (PCID) feature allows a logical processor to cache TLB mappings concurrently for multiple virtual address spaces. When enabled (by setting CR4.PCIDE=1), the processor associates the current 12-bit PCID with each TLB mapping it creates. Only entries matching the current PCID are used when performing address translations. In this way, the processor may retain cached TLB mappings for multiple contexts.

The current PCID is the value in CR3[11:0]. When PCIDs are enabled the system software can store 12-bit Process Context Identifiers in CR3 for different address spaces. Subsequently, when system software switches address spaces (by writing the page table base pointer in CR3[62:12]), the processor may use TLB mappings previously stored for that address space and PCID, providing that bit 63 of the source operand is set to 1. If bit 63 is set to 0, the legacy behavior of a move to CR3 is maintained, invalidating TLB entries but only non-global entries for the specified PCID. Note that this bit is not stored in the CR3 register itself. PCID-tagged TLB contents may also be managed using the INVPCID instruction; see the INVPCID description in Volume 3 for details. A MOV to CR4 that clears CR4.PCIDE causes all cached entries in the TLB for the logical processor to be invalidated. When PCIDs are not enabled (CR4.PCIDE=0) the current PCID is always zero and all TLB mappings are associated with PCID=0.

Attempting to set CR4.PCIDE with a MOV to CR4 if EFER.LMA = 0 or CR3[11:0] <> 0 causes a #GP exception. Attempting to clear CR0.PG with a MOV to CR0 if CR4.PCIDE is set also causes a #GP exception. The presence of PCID functionality is indicated by CPUID Function 1, ECX[PCID]=1.

5.5.2 Global Pages

The processor invalidates the TLB whenever CR3 is loaded either explicitly or implicitly. After the TLB is invalidated, subsequent address references can consume many clock cycles until their translations are cached as new entries in the TLB. Invalidation of TLB entries for frequently-used or critical pages can be avoided by specifying the translations for those pages as *global.* TLB entries for global pages are not invalidated as a result of a CR3 load. Global pages are invalidated using the INVLPG instruction.

Global-page extensions are controlled by setting and clearing the PGE bit in CR4 (bit 7). When CR4.PGE is set to 1, global-page extensions are enabled. When CR4.PGE is cleared to 0, global-page

[AMD Public Use]

extensions are disabled. When CR4.PGE=1, setting the global (G) bit in the translation-table entry marks the page as global.

The INVLPG instruction ignores the G bit and can be used to invalidate individual global-page entries in the TLB. To invalidate all entries, including global-page entries, disable global-page extensions (CR4.PGE=0).

5.5.3 TLB Management

Generally, unless system software modifies the linear-to-physical address mapping, the processor manages the TLB transparently to software. This includes allocating entries and replacing old entries with new entries. In general, software changes made to paging-data structures are not automatically reflected in the TLB. In these situations, it is necessary for software to invalidate TLB entries so that these changes will be propagated to the page-translation mechanism.

TLB entries can be explicitly invalidated using operations intended for that purpose or implicitly invalidated as a result of another operation. TLB invalidation has no effect on the associated page-translation tables in memory.

Explicit Invalidations. Several mechanisms are provided to explicitly invalidate the TLB:

- The *Invalidate TLB Entry* instruction (INVLPG) can be used to invalidate a specific entry within the TLB. This instruction invalidates an entry regardless of whether it is marked as global or not.
- The *Invalidate TLB entry in a Specified ASID* instruction (INVLPGA) operates similarly, but operates only on entries associated with the specified ASID. See “Invalidate Page, Alternate ASID” on page 552.
- The *Invalidate TLB with Broadcast* instruction (INVLPGB) can be used to invalidate a specified range of TLB entries on the local processor and broadcast the invalidation operation to remote processors. See INVLPGB in Volume 3.
- The *Invalidate TLB entries in Specified PCID* instruction (INVPCID) can be used to invalidate TLB entries of the specified Processor Context ID. See INVPCID in Volume 3.
- Updates to the CR3 register cause the entire TLB to be invalidated *except* for global pages. The CR3 register can be updated with the MOV CR3 instruction. CR3 is also updated during a task switch, with the updated CR3 value read from the TSS of the new task. This behavior for CR3 is modified by the PCID extension; See “Process Context Identifier” on page 160 for details.
- The TLB_CONTROL field of a VMCB can request specific flushes of the TLB to occur when the VMRUN instruction is executed on that VMCB. See “TLB Flush” on page 551.

Implicit Invalidations. The following operations cause the entire TLB to be invalidated, including global pages:

- Modifying the CR0.PG bit (paging enable).
- Modifying the CR4.PAE bit (physical-address extensions), the CR4.PSE bit (page-size extensions), or the CR4.PGE bit (page-global enable).
- Entering SMM as a result of an SMI interrupt.

[AMD Public Use]

- Executing the RSM instruction to return from SMM.
- Updating a memory-type range register (MTRR) with the WRMSR instruction.
- External initialization of the processor.
- External masking of the A20 address bit (asserting the A20M# input signal).
- Writes to certain model-specific registers with the WRMSR instruction; see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product for more information.
- A MOV to CR4 that changes CR4.PKE from 0 to 1.
- A MOV to CR4 that changes CR4.PCIDE from 1 to 0.

Invalidation of Table Entry Upgrades. If a table entry is updated to remove a page access constraint, such as removing supervisor, read-only, and/or no-execute restrictions, an invalidation is not required because the hardware will automatically detect the changes. If a table entry is updated and does not remove a permission violation, it is unpredictable whether the old or updated entry will be used until an invalidation is performed.

Speculative Caching of Address Translations. For performance reasons, AMD64 processors may speculatively load valid address translations into the TLB on false execution paths. Such translations are not based on references that a program makes from an "architectural state" perspective, but which the processor may make in speculatively following an instruction path which turns out to be mispredicted. In general, the processor may create a TLB entry for any linear address for which valid entries exist in the page table structure currently pointed to by CR3. This may occur for both instruction fetches and data references. Such entries remain cached in the TLBs and may be used in subsequent translations. Loading a translation speculatively will set the Accessed bit, if not already set. A translation will *not* be loaded speculatively if the Dirty bit needs to be set.

Caching of Upper Level Translation Table Entries. Similarly, to improve the performance of table walks on TLB misses, AMD64 processors may save upper level translation table entries in special table walk caching structures which are kept coherent with the tables in memory via the same mechanisms as the TLBs—by means of the INVLPG instruction, moves to CR3, and modification of paging control bits in CR0 and CR4. Like address translations in the TLB, these upper level entries may also be cached speculatively and by false-path execution. These entries are never cached if their P (present) bits are set to 0.

Under certain circumstances, an upper-level table entry that cannot ultimately lead to a valid translation (because there are no valid entries in the lower level table to which it points) may also be cached. This can happen while executing down a false path, when an in-progress table walk gets cancelled by the branch mispredict before the low level table entry that would cause a fault is encountered. Said another way, the fact that a page table has no valid entries does not guarantee that upper level table entries won't be accessed and cached in the processor, as long as those upper level entries are marked as present. For this reason, it is not safe to modify an upper level entry, even if no valid lower-level entries exist, without first clearing its present bit, followed by an INVLPG instruction.

[AMD Public Use]

Use of Cached Entries When Reporting a Page Fault Exception. On current AMD64 processors, when any type of page fault exception is encountered by the MMU, any cached upper-level entries that lead to the faulting entry are flushed (along with the TLB entry, if already cached) and the table walk is repeated to confirm the page fault using the table entries in memory. This is done because a table entry is allowed to be upgraded (by marking it as present, or by removing its write, execute or supervisor restrictions) without explicitly maintaining TLB coherency. Such an upgrade will be found when the table is re-walked, which resolves the fault. If the fault is confirmed on the re-walk however, a page fault exception is reported, and upper level entries that may have been cached on the re-walk are flushed.

Handling of D-Bit Updates. When the processor needs to set the D bit in the PTE for a TLB entry that is already marked as writable at all cached TLB levels, the table walk that is performed to access the PTE in memory may use cached upper level table entries. This differs from the fault situation previously described, in which cached entries aren't used to confirm the fault during the table walk.

Invalidation of Cached Upper-level Entries by INVLPG. The effect of INVLPG on TLB caching of upper-level page table entries is controlled by EFER[TCE] on processors that support the translation cache extension feature. If EFER[TCE] is 0, or if the processor does not support the translation cache extension feature, an INVLPG will flush all upper-level page table entries in the TLB as well as the target PTE. If EFER[TCE] is 1, INVLPG will flush only those upper-level entries that lead to the target PTE, along with the target PTE itself. INVLPGA may flush all upper-level entries regardless of the state of TCE. For further details, see Section 3.1.7 "Extended Feature Enable Register (EFER)" on page 56.

Handling of PDPT Entries in PAE Mode. When 32-bit PAE mode is enabled on AMD64 processors (CR4.PAE is set to 1) a third level of the address translation table hierarchy, the page directory pointer table (PDPT), is enabled. This table contains four entries. On older AMD64 processors, in native mode, these four entries are unconditionally loaded into the table walk cache whenever CR3 is written with the PDPT base address, and remain locked in. At this point they are also checked for reserved bit violations, and if such violations are present a general-protection exception (#GP) occurs.

Under SVM, however, when the processor is in guest mode with PAE enabled, the guest PDPT entries are not cached or validated at this point, but instead are loaded and checked on demand in the normal course of address translation, just like page directory and page table entries. Any reserved bit violations are detected at the point of use, and result in a page-fault (#PF) exception rather than a general-protection (#GP) exception. The cached PDPT entries are subject to displacement from the table walk cache and reloading from the PDPT, hence software must assume that the PDPT entries may be read by the processor at any point while those tables are active. Modern AMD processors implement this same behavior in native mode as well, rather than pre-loading the PDPT entries.

5.6 Page-Protection Checks

The AMD64 architecture provides the following forms of page-level memory protection:

[AMD Public Use]

- Supervisor pages. This form of protection prevents non-privileged (user) code from accessing privileged (supervisor) code and data.
- Read-only pages. This form of protection prevents writes into read-only address spaces.
- Instruction fetch restrictions. Two forms of page-level memory protection prevent the processor from fetching instructions from pages that are either known to contain non-executable data or that are accessible by user-mode code.
- Memory protection keys. This form of protection allows an application to manage page-based data access protections from user mode.
- Shadow stack pages. The processor restricts the types of memory accesses that are allowed to read or write a shadow stack page and prohibits the shadow stack mechanism from accessing non-shadow stack pages. See “Shadow Stack Protection” on page 167.

Access protection checks are performed when a virtual address is translated into a physical address. For those checks, the processor examines the page-level memory-protection bits in the translation tables to determine if the access is allowed. The page table bits involved in these checks are:

- User/Supervisor (U/S)—See “User/Supervisor (U/S) Bit” on page 157.
- Read/Write (R/W)—See “Read/Write (R/W) Bit” on page 156.
- No-Execute (NX)—See “No Execute (NX) Bit” on page 158.
- Memory Protection Key (MPK)—See “Memory Protection Keys (MPK) Bit” on page 166.

Access protection actions taken by the processor are controlled by the following bits:

- Write-Protect enable (CR0.WP)—See “Write Protect (WP) Bit” on page 44.
- No-Execute Enable (EFER.NXE)—See “No-Execute Enable (NXE) Bit” on page 58.
- Supervisor-mode Execution Prevention enable (CR4.SMEP)—See “Supervisor Mode Execution Prevention (SMEP)” on page 51.
- Supervisor-mode Access Prevention enable (CR4.SMAP)—See “Supervisor-Mode Access Prevention(CR4.SMAP) Bit” on page 166.
- Alignment Check Bit (RFLAGS.AC)—See ““Alignment Check (AC) Bit” on page 55.
- Protection Key Enable (CR4.PKE)—See “Protected-Mode Enable (PE) Bit” on page 43.
- Control-flow Enforcement Technology (CR4.CET)—See “CR4 Register” on page 47.

These protection checks are available at all levels of the page-translation hierarchy.

5.6.1 User/Supervisor (U/S) Bit

The U/S bit in the page-translation tables determines the privilege level required to access the page. If U/S=0 in any of the page table entries traversed during a table walk, the page is considered a supervisor page. If U/S=1 in all the page table entries traversed during a table walk, the page is considered a user page. Conceptually, user (non-privileged) pages correspond to a current privilege-level (CPL) of 3, or least-privileged. Supervisor (privileged) pages correspond to a CPL of 0, 1, or 2, all of which are jointly regarded as most-privileged.

[AMD Public Use]

When the processor is running at a CPL of 0, 1, or 2, it can access both user and supervisor pages unless restricted by SMEP or SMAP (See sections “Supervisor-Mode Execution Prevention (CR4.SMEP) Bit” on page 166 and “Supervisor-Mode Access Prevention(CR4.SMAP) Bit” on page 166 for further details). However, when the processor is running at a CPL of 3, it can only access user pages. If an attempt is made to access a supervisor page while the processor is running at CPL = 3, a page-fault exception (#PF) occurs.

See “Privilege-Level Concept” on page 107 for more information on processor privilege levels.

5.6.2 Read/Write (R/W) Bit

The R/W bit in the page-translation tables specifies the access type allowed for the page. If R/W=0 in any of the page table entries traversed during a table walk, the page is read-only. If R/W=1 in all the page table entries traversed during a table walk, the page is read/write. A page-fault exception (#PF) occurs if an attempt is made by user software to write to a read-only page. If supervisor software attempts to write a read-only page, the outcome depends on the value of the CR0.WP bit (described below).

5.6.3 No Execute (NX) Bit

The NX bit provides the ability to mark a page as non-executable. If the NX bit is set at any level of the page-table hierarchy in the table entries traversed during a table walk, the page mapped by those entries is a no-execute page. When no-execute protection is enabled, any attempt to fetch an instruction from a no-execute page results in a page-fault exception (#PF).

The no-execute protection check applies to all privilege levels. It does not distinguish between supervisor and user-level accesses.

The no-execute protection feature is supported only in PAE-paging mode. In 32-bit PAE mode, the NX bit is not supported at the Page Directory Pointer table level. In this mode, the value of the NX bit at the PDP level defaults to 0.

No-execute protection is enabled by setting the NXE bit in the EFER register to 1. Before setting this bit, system software must verify the processor supports the no-execute feature by checking the CPUID NX feature flag (CPUID Fn8000_0001_EDX[NX]).

5.6.4 Write Protect (CR0.WP) Bit

The ability to write to read-only pages is governed by the processor mode and whether write protection is enabled. If write protection is *not* enabled, a processor running at CPL 0, 1, or 2 can write to any physical page, even if it is marked as read-only. Enabling write protection by setting the WP bit in CR0 prevents supervisor code from writing into read-only pages, including read-only user-level pages.

A page-fault exception (#PF) occurs if software attempts to write (at any privilege level) into a read-only page while write protection is enabled.

[AMD Public Use]

5.6.5 Supervisor-Mode Execution Prevention (CR4.SMEP) Bit

When supported and enabled, a page-fault exception (#PF) is generated if the processor attempts to fetch an instruction from a user page while running at CPL 0, 1, or 2.

Supervisor-mode execution prevention is enabled by setting the SMEP bit (bit 20) in the CR4 register to 1. Before setting this bit, system software must verify the processor supports the SMEP feature by checking the SMEP feature flag (CPUID Fn0000_0007_EBX[SMEP]_x0 = 1).

For more information using the CPUID instruction see Section 3.3 “Processor Feature Identification” on page 73.

5.6.6 Supervisor-Mode Access Prevention(CR4.SMAP) Bit

When SMAP is supported and enabled, a page-fault exception (#PF) is generated if the processor attempts to read or write data from a user page and one of the following is true:

- The access is an implicit supervisor-mode access OR
- The access is made while running at CPL 0,1, or 2 and RFLAGS.AC=0.

Some accesses are considered implicit supervisor-mode accesses. Implicit supervisor-mode accesses are subject to the SMAP check regardless of the value of RFLAGS.AC. An implicit supervisor-mode access is one that is considered a supervisor access regardless of the value of CPL.

Supervisor-mode access prevention is enabled by setting the SMAP bit (bit 21) in the CR4 register to 1. Before setting this bit, system software must verify the processor supports the SMAP feature by checking the SMAP feature flag CPUID Fn0000_0007_EBX[SMAP]_x0 (bit 20) = 1.

Shadow stack accesses are not subject to the SMAP check.

For more information on using the CPUID instruction see “Processor Feature Identification” on page 73.

5.6.7 Memory Protection Keys (MPK) Bit

The Memory Protection Key (MPK) feature provides a way for applications to impose page-based data access protections (read/write, read-only or no access), without requiring modification of page tables and subsequent TLB invalidations when the application changes protection domains.

When MPK is enabled (CR4.PKE=1), a protection key is located in bits 62:59 of final page table entry mapping each virtual address. This 4-bit protection key is used as an index (i) into the user-accessible PKRU register which contains 16 access-disable/write-disable (WDi/ADi) pairs.

[AMD Public Use]

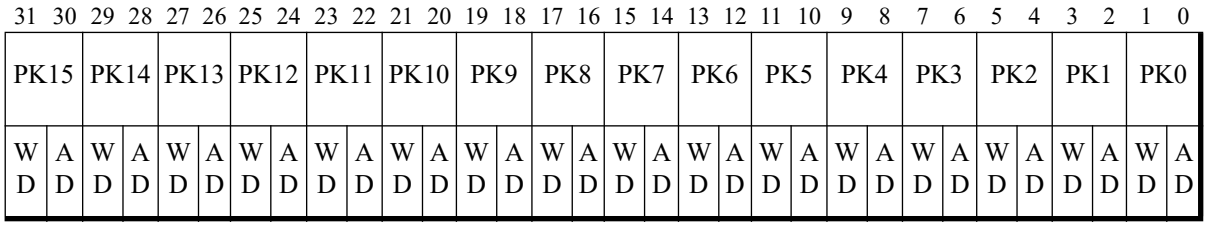


Figure 5-35. PKRU Register

The WDi/ADi pairs operate as follows:

If ADi=0, data access is permitted

- If ADi=1, no data access is permitted (regardless of CPL)
- If WDi == 0, write access is allowed
- If WDi == 1: User-mode write access is not allowed. Supervisor access is controlled by CR0.WP:
  - If CR0.WP=1, supervisor-mode writes are not allowed
  - If CR0.WP=0, supervisor-mode writes are allowed

Software can use the RDPKRU and WRPKRU instructions to read and write the PKRU register. These instructions are not privileged and can be used in user mode or in supervisor mode.

The MPK mechanism is ignored in the following cases:

- if CR4.PKE=0
- if long mode is disabled (EFER.LMA=0)
- for instruction fetches
- for pages marked in the paging structures as supervisor addresses (U/S=0)

5.7 Shadow Stack Protection

When the shadow stack feature is enabled (CR4.CET=1), certain combinations of page-table protection bits are used to distinguish pages containing shadow stacks from ordinary pages. As described in the following sections, the processor restricts the types of memory accesses that can be made to shadow stack pages and prohibits the shadow stack mechanism from accessing non-shadow stack pages. (See “Shadow Stacks” on page 710 for details on the shadow stack feature).

5.7.1 Shadow Stack Accesses

The processor treats certain memory accesses as shadow stack accesses. Shadow stack accesses are generated only by the shadow stack instructions or by the shadow stack mechanism. As with ordinary data accesses, a shadow stack access can be either a supervisor access or a user access, depending on the CPL when the access is made. Shadow stack accesses made when the processor is at CPL 0, 1, or 2 are supervisor-shadow stack accesses, and accesses made at CPL 3 are user-shadow stack accesses.

[AMD Public Use]

(An exception is the WRUSS instruction, whose accesses are always treated as user-shadow stack accesses).

5.7.2 Shadow Stack Pages

Shadow stack accesses are allowed only to linear addresses that are mapped to shadow stack pages. A shadow stack is described by the following combination of page-table protection bits:

- R/W(Read/Write)=0 and D(Dirty)=1 in the final page-table entry that maps the linear address.
- R/W(Read/Write)=1 in all other page-mapping structures leading to the final page-table entry.

The U/S (User/Supervisor) bit in the page-translation tables determines the privilege level required to access a given shadow stack page. If U/S=0, the page is considered a supervisor-shadow stack page and if U/S=1 the page is considered a user-shadow stack page.

5.7.3 Shadow Stack Protection Checks

The processor restricts the types of memory accesses that are allowed to read or write a shadow stack page. The page-level protection bits and the type of memory access are examined to determine if the access is allowed. The following section assumes the memory protection key field allows access to the given page, if memory protection keys are enabled, and that CR0.WP=1 (which is prerequisite for enabling the shadow stack feature).

The following memory accesses are allowed to shadow stack pages:

- User-shadow stack accesses can read or write user-shadow stack pages,
- Supervisor-shadow stack accesses can read or write supervisor-shadow stack pages.

*(**Note**: shadow stack write accesses are allowed to complete, even though the R/W bit is 0).*

- Non-shadow stack reads can read any shadow stack page (subject to U/S page protections).

The following memory accesses are not allowed:

- User-shadow stack access to supervisor-shadow stack pages.
- Supervisor-shadow stack access to user-shadow stack pages.
- Any shadow stack access to a non-shadow stack page.
- Non-shadow stack writes to a shadow stack page.

If the memory access is not allowed, a page-fault exception (#PF) is generated with the paging-protection violation bits (user/supervisor, read/write, or both) set in the error code as appropriate. The SS bit is set in the #PF error code if the page-fault was caused by a shadow stack access. (See “Page-Fault Error Code” on page 265).

[AMD Public Use]

5.8 Protection Across Paging Hierarchy

The privilege level and access type specified at each level of the page-translation hierarchy have a combined effect on the protection of the translated physical page. Enabling and disabling write protection via CR0.WP further qualifies the protection effect on the physical page.

Table 5-2 shows the overall effect that privilege level and access type have on physical-page protection when write protection is disabled (CR0.WP=0). In this case, when *any* translation-table entry is specified as supervisor level, the physical page is a supervisor page and can only be accessed by software running at CPL 0, 1, or 2. Such a page allows read/write access even if all levels of the page-translation hierarchy specify read-only access.

Table 5-2. Physical-Page Protection, CR0.WP=0

Page-Map Level-4 Entry		Page-Directory-Pointer Entry		Page-Directory Entry		Page-Table Entry		Effective Result on Physical Page	
U/S	R/W	U/S	R/W	U/S	R/W	U/S	R/W	U/S	R/W
S	—	—	—	—	—	—	—	S	R/W
—	—	S	—	—	—	—	—		
—	—	—	—	S	—	—	—		
—	—	—	—	—	—	S	—		
U	R	U	—	U	—	U	—	U	R[1]
U	—	U	R	U	—	U	—		
U	—	U	—	U	R	U	—		
U	—	U	—	U	—	U	R		
U	R/W	U	R/W	U	R/W	U	R/W	U	R/W

Note:
S = Supervisor Level (CPL=0, 1, or 2), U = User Level (CPL = 3), R = Read-Only Access, R/W = Read/Write Access, — = Don't Care.

Note:
1. *Supervisor-level programs can access these pages as R/W.*

If *all* table entries in the translation hierarchy are specified as user level the physical page is a user page, and both supervisor and user software can access it. In this case the physical page is read-only if any table entry in the translation hierarchy specifies read-only access. All table entries in the translation hierarchy must specify read/write access for the physical page to be read/write.

Table 5-3 shows the overall effect that privilege level and access type have on physical-page access when write protection is enabled (CR0.WP=1). When any translation-table entry is specified as supervisor level, the physical page is a supervisor page and can only be accessed by supervisor software. In this case, the physical page is read-only if any table entry in the translation hierarchy specifies read-only access. All table entries in the translation hierarchy must specify read/write access for the supervisor page to be read/write.

[AMD Public Use]

Table 5-3. Effect of CR0.WP=1 on Supervisor Page Access

Page-Map Level-4 Entry	Page Directory-Pointer Entry	Page Directory Entry	Page Table Entry	Physical Page
R/W	R/W	R/W	R/W	R/W
R	—	—	—	R
—	R	—	—	
—	—	R	—	
—	—	—	R	
W	W	W	W	W

Note:
R = Read-Only Access Type, W = Read/Write Access Type, — = Don't Care.
Physical page is in supervisor mode, as determined by U/S settings in Table 5-2.

5.8.1 Access to User Pages when CR0.WP=1

As shown in Table 5-2 on page 169, read/write access to user-level pages behaves the same as when write protection is disabled (CR0.WP=0), with one critical difference. When write protection is enabled, supervisor programs cannot write into read-only user pages.

5.9 Effects of Segment Protection

Segment-protection and page-protection checks are performed serially by the processor, with segment-privilege checks performed first, followed by page-protection checks. Page-protection checks are not performed if a segment-protection violation is found. If a violation is found during either segment-protection or page-protection checking, an exception occurs and no memory access is performed. Segment-protection violations cause either a general-protection exception (#GP) or a stack exception (#SS) to occur. Page-protection violations cause a page-fault exception (#PF) to occur.

5.10 Upper Address Ignore

The Upper Address Ignore feature provides a way for software to use bits 63:57 of an address as an arbitrary software-assigned and software-interpreted tag. When this feature is enabled, these address bits are excluded from the canonicality check that's done when the address is used for certain memory references.

5.10.1 Detecting and Enabling Upper Address Ignore

Support for the Upper Address Ignore feature is indicated by CPUID Fn8000_0021_EAX[UpperAddressIgnore](bit 7)=1. The Upper Address Ignore feature is enabled by setting EFER.UAIE (bit 20) in 64-bit mode (EFER.LMA=CS.L=1). EFER.UAIE is ignored outside of 64-bit mode.

[AMD Public Use]

5.10.2 Upper Address Ignore Operation

When Upper Address Ignore is active the processor no longer performs a canonical address check on bits 63:57 of the logical address for memory references that use either the DS or ES segment, it checks only bits 56:48 of the logical address when 4-level paging is enabled. No canonical checks are performed for ES or DS segment memory references when 5-level paging is enabled. For a description of 4-level and 5-level paging see Section 5.3 “Long-Mode Page Translation” on page 141.

Any memory reference made using the segment registers CS, SS, FS or GS still performs the normal canonical address check. This includes indirect jump, call, and return target addresses since they are CS based. Canonical checks for implicit references to IDT, GDT, LDT and TSS are not suppressed when Upper Address Ignore is active.

5.10.3 Address Tag Storage

The following registers, which hold virtual addresses for communication to software, are not guaranteed to hold the address tag in the upper 7 bits:

- CR2
- x87 DP
- IBS Data Cache Linear Address Register

5.10.4 Debug Breakpoint Behavior with Upper Address Ignore

The Debug Breakpoint Address Registers (DR0-3) hold full 64-bit addresses and Upper Address Ignore changes how the address match is performed for data address breakpoints. When Upper Address Ignore is active the value in DR0-3[63:57] is ignored and the linear address of the memory access is only compared against DR0-3[56:0].

[AMD Public Use]

6 System Instructions

System instructions provide control over the resources used to manage the processor operating environment. This includes memory management, memory protection, task management, interrupt and exception handling, system-management mode, software debug and performance analysis, and model-specific features. Most instructions used to access these resources are privileged and can only be executed while the processor is running at CPL=0, although some instructions can be executed at any privilege level.

Table 6-1 summarizes the instructions used for system management. These include all privileged instructions, instructions whose privilege requirement is under the control of system software, non-privileged instructions that are used primarily by system software, and instructions used to transfer control to system software. Most of the instructions listed in Table 6-1 are summarized in this chapter, although a few are introduced elsewhere in this manual, as indicated in the *Reference* column of Table 6-1.

For details on individual system instructions, see "System Instruction Reference" in Volume 3.

Table 6-1. System Management Instructions

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
ARPL	Adjust Requester Privilege Level			X	"Adjusting Access Rights" on page 183
CLAC	Clear Alignment Check Flag	X			"CLAC and STAC Instructions" on page 180
CLGI	Clear Global Interrupt Flag	X			"Global Interrupt Flag, STGI and CLGI Instructions" on page 553
CLI	Clear Interrupt Flag		X		"CLI and STI Instructions" on page 180
CLRSSBSY	Clear Shadow Stack Busy	X			"CLRSSBSY" on page 185
CLTS	Clear Task-Switched Flag in CR0	X			"CLTS Instruction" on page 180
HLT	Halt	X			"Processor Halt" on page 183
INCSSP	Increment SSP			X	"INCSSP" on page 185
INT3	Interrupt to Debug Vector			X	"Breakpoint Instruction (INT3)" on page 410
INVD	Invalidate Caches	X			"Cache Management" on page 183
INVLPG	Invalidate TLB Entry	X			"INVLPG Instruction" on page 184
INVLPGA	Invalidate TLB Entry in a Specified ASID	X			"Invalidate Page, Alternate ASID" on page 552

Note:
1. *The operating system controls the privilege required to use the instruction.*

[AMD Public Use]

Table 6-1. System Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
INVLPGB	Invalidate TLB Entries with Broadcast	X			"INVLPGB Instruction" on page 185
INVPCID	Invalidate TLB Entries in Specified Processor Context	X			"INVPCID Instruction" on page 185
IRET*x*	Interrupt Return (all forms)			X	"Returning From Interrupt Procedures" on page 279
LAR	Load Access-Rights Byte			X	"Checking Access Rights" on page 182
LGDT	Load Global-Descriptor-Table Register	X			"LGDT and LIDT Instructions" on page 181
LIDT	Load Interrupt-Descriptor-Table Register	X			
LLDT	Load Local-Descriptor-Table Register	X			"LLDT and LTR Instructions" on page 182
LMSW	Load Machine-Status Word	X			"LMSW and SMSW Instructions" on page 179
LSL	Load Segment Limit			X	"Checking Segment Limits" on page 182
LTR	Load Task Register	X			"LLDT and LTR Instructions" on page 182
MONITOR	Setup Monitor Address		X		--
MOV CR*n*	Move to/from Control Registers	X			"MOV CRn Instructions" on page 179
MOV DR*n*	Move to/from Debug Registers	X			"Accessing Debug Registers" on page 180
MWAIT	Monitor Wait		X		--
RDFSBASE	Read FS Base Address			X	"RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions" on page 181
RDGSBASE	Read GS Base Address			X	
RDMSR	Read Model-Specific Register	X			"RDMSR and WRMSR Instructions" on page 180
RDPMC	Read Performance-Monitor Counter		X		"RDPMC Instruction" on page 180
RDSSP	Read SSP			X	"RDSSP" on page 185
RDTSC	Read Time-Stamp Counter		X		"RDTSC Instruction" on page 180
RDTSCP	Read Time-Stamp Counter and Processor ID		X		"RDTSCP Instruction" on page 181
RSM	Return from System-Management Mode			X	"Leaving SMM" on page 342

Note:

1. *The operating system controls the privilege required to use the instruction.*

[AMD Public Use]

Table 6-1. System Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
RSTORSSP	Restore SSP			X	“RSTORSSP” on page 185
SAVEPREVSSP	Save Previous SSP			X	“SAVEPREVSSP” on page 185
SETSSBSY	Set Shadow Stack Busy	X			“SETSSBSY” on page 185
SGDT	Store Global-Descriptor-Table Register		X		“SGDT and SIDT Instructions” on page 182
SIDT	Store Interrupt-Descriptor-Table Register		X		
SKINIT	Secure Init and Jump with Attestation	X			“Security” on page 582
SLDT	Store Local-Descriptor-Table Register		X		“SLDT and STR Instructions” on page 182
SMSW	Store Machine-Status Word		X		“LMSW and SMSW Instructions” on page 179
STAC	Set Alignment Check Flag		X		“CLAC and STAC Instructions” on page 180
STI	Set Interrupt Flag		X		“CLI and STI Instructions” on page 180
STGI	Set Global Interrupt Flag	X			“Global Interrupt Flag, STGI and CLGI Instructions” on page 553
STR	Store Task Register		X		“SLDT and STR Instructions” on page 182
SWAPGS	Swap GS and KernelGSbase Registers	X			“SWAPGS Instruction” on page 178
SYSCALL	Fast System Call			X	“SYSCALL and SYSRET” on page 176
SYSENTER	System Call			X	“SYSENTER and SYSEXIT (Legacy Mode Only)” on page 178
SYSEXIT	System Return	X			
SYSRET	Fast System Return	X			“SYSCALL and SYSRET” on page 176
VERR	Verify Segment for Reads			X	“Checking Read/Write Rights” on page 182
VERW	Verify Segment for Writes			X	
VMLOAD	Load State from VMCB	X			“VMSAVE and VMLOAD Instructions” on page 528
VMMCALL	Call VMM	X			“VMMCALL Instruction” on page 553
VMRUN	Run Virtual Machine	X			“VMRUN Instruction” on page 523

Note:

1. *The operating system controls the privilege required to use the instruction.*

[AMD Public Use]

Table 6-1. System Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
VMSAVE	Save State to VMCB	X			"VMSAVE and VMLOAD Instructions" on page 528
WBINVD	Writeback and Invalidate Caches	X			"Cache Management" on page 183
WBNOINVD	Writeback No Invalidate	X			
WRFSBASE	Write FS Base Address			X	"RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions" on page 181
WRGSBASE	Write GS Base Address			X	
WRMSR	Write Model-Specific Register	X			"RDMSR and WRMSR Instructions" on page 180
WRSS	Write to Shadow Stack			X	"WRSS" on page 185
WRUSS	Write to User Shadow Stack	X			"WRUSS" on page 185

Note:
1. *The operating system controls the privilege required to use the instruction.*

The following instructions are summarized in this chapter but are not categorized as system instructions, because of their importance to application programming:

- The CPUID instruction returns information critical to system software in initializing the operating environment. It is fully described in Section 3.3, "Processor Feature Identification," on page 73.
- The PUSHF and POPF instructions set and clear certain rFLAGS bits depending on the processor operating mode and privilege level. These dependencies are described in "POPF and PUSHF Instructions" on page 180.

The MOV, PUSH, and POP instructions can be used to load and store segment registers, as described in "MOV, POP, and PUSH Instructions" on page 181.

User Mode Instruction Prevention (UMIP)

This security mode restricts certain instructions so that they do not reveal information about structures that are controlled by the processor when it is at CPL=0. The presence of the UMIP feature is indicated by CPUID Function 0000_0007, ECX[2]=1. This mode is enabled by setting CR4 bit 11 to a 1. Attempts to set CR4 bit 11 when the UMIP feature is not supported result in a #GP fault. Once CR4[11] is set to 1, the SGDT, SIDT, SLDT, SMSW and STR instructions become available only at CPL=0. Any attempt to execute them with CPL>0 results in a #GP fault with error code 0.

6.1 Fast System Call and Return

Operating systems can use both paging and segmentation to implement protected memory models. Segment descriptors provide the necessary memory protection and privilege checking for segment accesses. By setting segment-descriptor fields appropriately, operating systems can enforce access restrictions as needed.

[AMD Public Use]

A disadvantage of segment-based protection and privilege checking is the overhead associated with loading a new segment selector (and its corresponding descriptor) into a segment register. Even when using the flat-memory model, this overhead still occurs when switching between privilege levels because code segments (CS) and stack segments (SS) are reloaded with different segment descriptors.

To initiate a call to the operating system, an application transfers control to the operating system through a gate descriptor (call, interrupt, trap, or task gate). In the past, control was transferred using either a far CALL instruction or a software interrupt. Transferring control through one of these gates is slowed by the segmentation-related overhead, as is the later return using a far RET or IRET instruction. The following checks are performed when control is transferred in this manner:

- Selectors, gate descriptors, and segment descriptors are in the proper form.
- Descriptors lie within the bounds of the descriptor tables.
- Gate descriptors reference the appropriate segment descriptors.
- The caller, gate, and target privileges all allow the control transfer to take place.
- The stack created by the call has sufficient properties to allow the transfer to take place.

In addition to these call-gate checks, other checks are made involving the task-state segment when a task switch occurs.

6.1.1 SYSCALL and SYSRET

SYSCALL and SYSRET Instructions. SYSCALL and SYSRET are low-latency system call and return instructions. These instructions assume the operating system implements a flat-memory model, which greatly simplifies calls to and returns from the operating system. This simplification comes from eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions). As a result, SYSCALL and SYSRET can take fewer than one-fourth the number of internal clock cycles to complete than the legacy CALL and RET instructions. SYSCALL and SYSRET are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-memory model.

SYSCALL and SYSRET require that the code-segment base, limit, and attributes (except for DPL) are consistent for all application and system processes. Only the DPL is allowed to vary. The processor assumes (but does not check) that the SYSCALL target CS segment descriptor entry has DPL=0 and the SYSRET target CS segment descriptor entry has DPL=3.

For details on the SYSCALL and SYSRET instructions, see “System Instruction Reference” in Volume 3.

Because SYSCALL and SYSRET do not use the program stack to store return addresses, the shadow stack mechanism is not used to validate their return addresses. However, when shadow stacks are enabled, SYSCALL and SYSRET save and restore the current SSP as follows:

- If the shadow stack feature is enabled at the current CPL (typically CPL=3), SYSCALL saves the current SSP to the PL3_SSP MSR
- If shadow stacks are enabled at the target CPL (CPL=0), SYSCALL clears the SSP to 0.

[AMD Public Use]

- If shadow stacks are enabled at CPL=3, SYSRET restores SSP from PL3_SSP.

SYSCALL and SYSRET MSRs. The STAR, LSTAR, and CSTAR registers are model-specific registers (MSRs) used to specify the target address of a SYSCALL instruction as well as the CS and SS selectors of the called and returned procedures. The SFMASK register is used in long mode to specify how rFLAGS is handled by these instructions. Figure 6-1 shows the STAR, LSTAR, CSTAR, and SFMASK register formats.

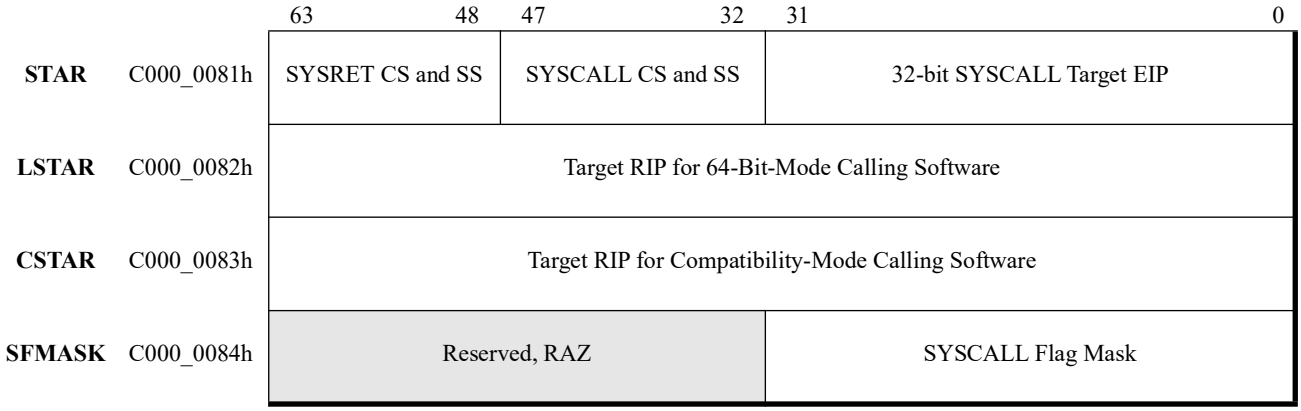


Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs

- *STAR*—The STAR register has the following fields (unless otherwise noted, all bits are read/write):
  - *SYSRET CS and SS Selectors*—Bits 63:48. This field is used to specify both the CS and SS selectors loaded into CS and SS during SYSRET. If SYSRET is returning to 32-bit mode (either legacy or compatibility), this field is copied directly into the CS selector field. If SYSRET is returning to 64-bit mode, the CS selector is set to this field + 16. SS.Sel is set to this field + 8, regardless of the target mode. Because SYSRET always returns to CPL 3, the RPL bits 49:48 should be initialized to 11b.
  - *SYSCALL CS and SS Selectors*—Bits 47:32. This field is used to specify both the CS and SS selectors loaded into CS and SS during SYSCALL. This field is copied directly into CS.Sel. SS.Sel is set to this field + 8. Because SYSCALL always switches to CPL 0, the RPL bits 33:32 should be initialized to 00b.
  - *32-bit SYSCALL Target EIP*—Bits 31:0. This is the target EIP of the called procedure.

  The legacy STAR register is not expanded in long mode to provide a 64-bit target RIP address. Instead, long mode provides two new STAR registers—long STAR (LSTAR) and compatibility STAR (CSTAR)—that hold a 64-bit target RIP.
- *LSTAR and CSTAR*—The LSTAR register holds the target RIP of the called procedure in long mode when the calling software is in 64-bit mode. The CSTAR register holds the target RIP of the called procedure in long mode when the calling software is in compatibility mode. The WRMSR instruction is used to load the target RIP into the LSTAR and CSTAR registers. If the RIP written

[AMD Public Use]

to either of the MSRs is not in canonical form, a #GP fault is generated on the WRMSR instruction.

- *SFMASK*—The SFMASK register is used to specify which RFLAGS bits are cleared during a SYSCALL. In long mode, SFMASK is used to specify which RFLAGS bits are cleared when SYSCALL is executed. If a bit in SFMASK is *set to 1*, the corresponding bit in RFLAGS is *cleared to 0*. If a bit in SFMASK is cleared to 0, the corresponding RFLAGS bit is not modified.

6.1.2 SYSENTER and SYSEXIT (Legacy Mode Only)

SYSENTER and SYSEXIT Instructions. Like SYSCALL and SYSRET, SYSENTER and SYSEXIT are low-latency system call and return instructions designed for use by system and application software implementing a flat-memory model. However, *these instructions are illegal in long mode and result in an undefined opcode exception (#UD) if software attempts to use them*. Software should use the SYSCALL and SYSRET instructions when running in long mode.

SYSENTER and SYSEXIT MSRs. Three model-specific registers (MSRs) are used to specify the target address and stack pointers for the SYSENTER instruction as well as the CS and SS selectors of the called and returned procedures. The register fields are:

- *SYSENTER Target CS*—Holds the CS selector of the called procedure.
- *SYSENTER Target ESP*—Holds the called-procedure stack pointer. The SS selector is updated automatically to point to the next descriptor entry after the SYSENTER Target CS, and ESP is the offset into that stack segment.
- *SYSENTER Target EIP*—Holds the offset into the CS of the called procedure.

Figure 6-2 shows the register formats and their corresponding MSR IDs.

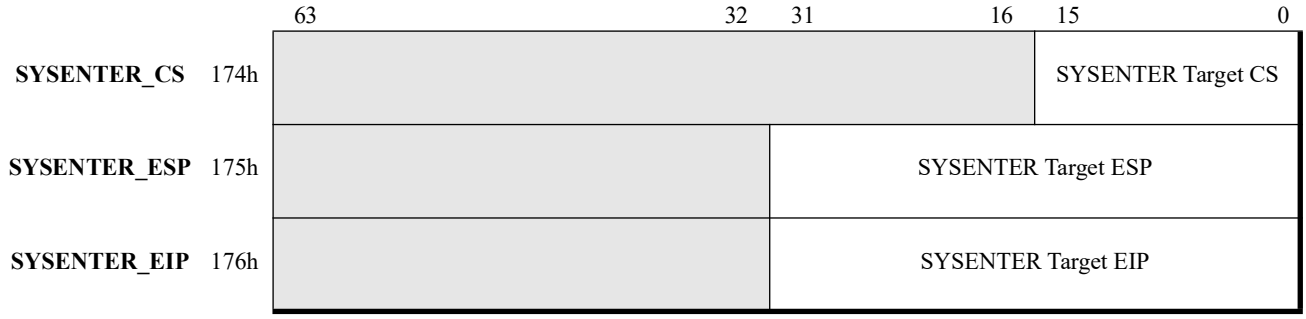


Figure 6-2. SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP MSRs

6.1.3 SWAPGS Instruction

The SWAPGS instruction provides a fast method for system software to load a pointer to system data structures. SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction or as a result of an interrupt or exception. Before returning to application software, SWAPGS can restore an application data-structure pointer that was replaced by the system data-structure pointer.

[AMD Public Use]

SWAPGS exchanges the base-address value located in the KernelGSbase model-specific register (MSR address C000_0102h) with the base-address value located in the hidden portion of the GS selector register (GS.base). This exchange allows the system-kernel software to quickly access kernel data structures by using the GS segment-override prefix during memory references.

The need for SwapGS arises from the requirement that, upon entry to the OS kernel, the kernel needs to obtain a 64-bit pointer to its essential data structures. When using SYSCALL to implement system calls, no kernel stack exists at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures, from which the kernel stack pointer could be read. Thus, the kernel cannot save GPRs or reference memory. SwapGS does not require any GPR or memory operands, so no registers need to be saved before using it. Similarly, when the OS kernel is entered via an interrupt or exception (where the kernel stack is already set up), SwapGS can be used to quickly get a pointer to the kernel data structures.

See “FS and GS Registers in 64-Bit Mode” on page 82 for more information on using the GS.base register in 64-bit mode.

6.2 System Status and Control

System-status and system-control instructions are used to determine the features supported by a processor, gather information about the current execution state, and control the processor operating modes.

6.2.1 Processor Feature Identification (CPUID)

CPUID Instruction. The CPUID instruction provides complete information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this information. System software normally uses the CPUID instruction to determine which optional features are available so the system can be configured appropriately. See Section 3.3, “Processor Feature Identification,” on page 73.

6.2.2 Accessing Control Registers

MOV CR*n* Instructions. The MOV CRn instructions can be used to copy data between the control registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute them.

LMSW and SMSW Instructions. The machine status word is located in CR0 register bits 15:0. The *load machine status word* (LMSW) instruction writes only the least-significant four status-word bits (CR0[3:0]). All remaining status-word bits (CR0[15:4]) are left unmodified by the instruction. The instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

The *store machine status word* (SMSW) instruction stores all 16 status-word bits (CR0[15:0]) into the target GPR or memory location. The instruction is not privileged and can be executed by all software.

[AMD Public Use]

CLTS Instruction. The *clear task-switched bit* instruction (CLTS) clears CR0.TS to 0. The CR0.TS bit is set to 1 by the processor every time a task switch takes place. The bit is useful to system software in determining when the x87 and multimedia register state should be saved or restored. See "Task Switched (TS) Bit" on page 44 for more information on using CR0.TS to manage x87-instruction state. The CLTS instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

6.2.3 Accessing the RFLAGS Register

The RFLAGS register contains both application and system bits. This section describes the instructions used to read and write system bits. Descriptions of instruction effects on application flags can be found in "Flags Register" in Volume 1 and "Instruction Effects on rFLAGS" in Volume 3.

POPF and PUSHF Instructions. The *pop and push rFLAGS* instructions are used for moving data between the rFLAGS register and the stack. They are not strictly system instructions, but their behavior is mode-dependent.

CLI and STI Instructions. The *clear interrupt* (CLI) and *set interrupt* (STI) instructions modify only the RFLAGS.IF bit or RFLAGS.VIF bit. Clearing RFLAGS.IF to 0 causes the processor to ignore maskable interrupts. Setting RFLAGS.IF to 1 causes the processor to allow maskable interrupts.

See "Virtual Interrupts" on page 290 for more information on the operation of these instructions when virtual-8086 mode extensions are enabled (CR4.VME=1).

CLAC and STAC Instructions. The *clear alignment check flag (CLAC) and set alignment check flag (STAC)* instructions modify only the RFLAGS.AC bit.

6.2.4 Accessing Debug Registers

The MOV DR*n* instructions are used to copy data between the debug registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute them. See "Debug Registers" on page 393 for a detailed description of the debug registers.

6.2.5 Accessing Model-Specific Registers

RDMSR and WRMSR Instructions. The *read/write model-specific register* instructions (RDMSR and WRMSR) can be used by privileged software to access the 64-bit MSRs. See "Model-Specific Registers (MSRs)" on page 60 for details about the MSRs.

RDPMC Instruction. The *read performance-monitoring counter* instruction, RDPMC, is used to read the model-specific performance-monitoring counter registers.

RDTSC Instruction. The *read time-stamp counter* instruction, RDTSC, is used to read the model-specific time-stamp counter (TSC) register.

[AMD Public Use]

RDTSCP Instruction. The *read time-stamp counter and processor ID* instruction, RDTSCP, is used to read the model-specific time-stamp counter (TSC) register and the low 32 bits of the TSC_AUX register (MSR C000_0103h).

6.3 Segment Register and Descriptor Register Access

The AMD64 architecture supports the legacy instructions that load and store segment registers and descriptor registers. In some cases the instruction capabilities are expanded to support long mode.

6.3.1 Accessing Segment Registers

MOV, POP, and PUSH Instructions. The MOV and POP instructions can be used to load a selector into a segment register from a general-purpose register or memory (MOV) or from the stack (POP). Any segment register, except the CS register, can be loaded with the MOV and POP instructions. The CS register must be loaded with a far-transfer instruction.

All segment register selectors can be stored in a general-purpose register or memory using the MOV instruction or pushed onto the stack using the PUSH instruction.

When a selector is loaded into a segment register, the processor automatically loads the corresponding descriptor-table entry into the hidden portion of the selector register. The hidden portion contains the base address, limit, and segment attributes.

Segment-load and segment-store instructions work normally in 64-bit mode. The appropriate entry is read from the system descriptor table (GDT or LDT) and is loaded into the hidden portion of the segment descriptor register. However, the contents of data-segment and stack-segment descriptor registers are ignored, except in the case of the FS and GS segment-register base fields—see “FS and GS Registers in 64-Bit Mode” on page 82 for more information.

The ability to use segment-load instructions allows a 64-bit operating system to set up segment registers for a compatibility-mode application before switching to compatibility mode.

6.3.2 Accessing Segment Register Hidden State

WRMSR and RDMSR Instructions. The base address field of the hidden state of the FS and GS registers are mapped to MSRs and may be read and written by privileged software when running in 64-bit mode.

RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions. When supported and enabled, these instructions allow software running at any privilege level to read and write the base address field of the hidden state of the FS and GS segment registers. These instructions are only defined in 64-bit mode.

6.3.3 Accessing Descriptor-Table Registers

LGDT and LIDT Instructions. The *load GDTR* (LGDT) and *load IDTR* (LIDT) instructions load a *pseudo-descriptor* from memory into the GDTR or IDTR registers, respectively.

[AMD Public Use]

LLDT and LTR Instructions. The *load LDTR* (LLDT) and *load TR* (LTR) instructions load a system-segment descriptor from the GDT into the LDTR and TR segment-descriptor registers (hidden portion), respectively.

SGDT and SIDT Instructions. The *store GDTR* (SGDT) and *store IDTR* (SIDT) instructions reverse the operation of the LGDT and LIDT instructions. SGDT and SIDT store a pseudo-descriptor from the GDTR or IDTR register into memory.

SLDT and STR Instructions. In all modes, the *store LDTR* (SLDT) and *store TR* (STR) instructions store the LDT or task selector from the visible portion of the LDTR or TR register into a general-purpose register or memory, respectively. The hidden portion of the LDTR or TR register is not stored.

6.4 Protection Checking

Several instructions are provided to allow software to determine the outcome of a protection check before performing a memory access that could result in a protection violation. By performing the checks before a memory access, software can avoid violations that result in a general-protection exception (#GP).

6.4.1 Checking Access Rights

LAR Instruction. The *load access-rights* (LAR) instruction can be used to determine if access to a segment is allowed, based on privilege checks and type checks. The LAR instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LAR performs a set of access-rights checks and, if successful, loads the segment-descriptor access rights into the destination register. Software can further examine the access-rights bits to determine if access into the segment is allowed.

6.4.2 Checking Segment Limits

LSL Instruction. The *load segment-limit* (LSL) instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LSL performs a set of preliminary access-rights checks and, if successful, loads the segment-descriptor limit field into the destination register. Software can use the limit value in comparisons with pointer offsets to prevent segment limit violations.

6.4.3 Checking Read/Write Rights

VERR and VERW Instructions. The *verify read-rights* (VERR) and *verify write-rights* (VERW) can be used to determine if a target code or data segment (not a system segment) can be read or written from the current privilege level (CPL). The source operand for these instructions is a pointer to the segment selector to be tested. If the tested segment (code or data) is readable from the current CPL, the VERR instruction sets RFLAGS.ZF to 1; otherwise, it is cleared to zero. Likewise, if the tested data segment is writable, the VERW instruction sets the RFLAGS.ZF to 1. A code segment cannot be tested for writability.

[AMD Public Use]

6.4.4 Adjusting Access Rights

ARPL Instruction. The *adjust RPL-field* (ARPL) instruction can be used by system software to prevent access into privileged-data segments by lower-privileged software. This can happen if an application passes a selector to system software and the selector RPL is less than (has greater privilege than) the calling-application CPL. To prevent this surrogate access, system software executes ARPL with the following operands:

- The destination operand is the data-segment selector passed to system software by the application.
- The source operand is the application code-segment selector (available on the system-software stack as a result of the CALL into system software by the application).

ARPL is not supported in 64-bit mode.

6.5 Processor Halt

The *processor halt* instruction (HLT) halts instruction execution, leaving the processor in the halt state. No registers or machine state are modified as a result of executing the HLT instruction. The processor remains in the halt state until one of the following occurs:

- A non-maskable interrupt (NMI).
- An enabled, maskable interrupt (INTR).
- Processor reset (RESET).
- Processor initialization (INIT).
- System-management interrupt (SMI).

6.6 Cache and TLB Management

Cache-management instructions are used by system software to maintain coherency within the memory hierarchy. Memory coherency and caches are discussed in Chapter 7, “Memory System.” Similarly, TLB-management instructions are used to maintain coherency between page translations cached in the TLB and the translation tables maintained by system software in memory. See “Translation-Lookaside Buffer (TLB)” on page 159 for more information.

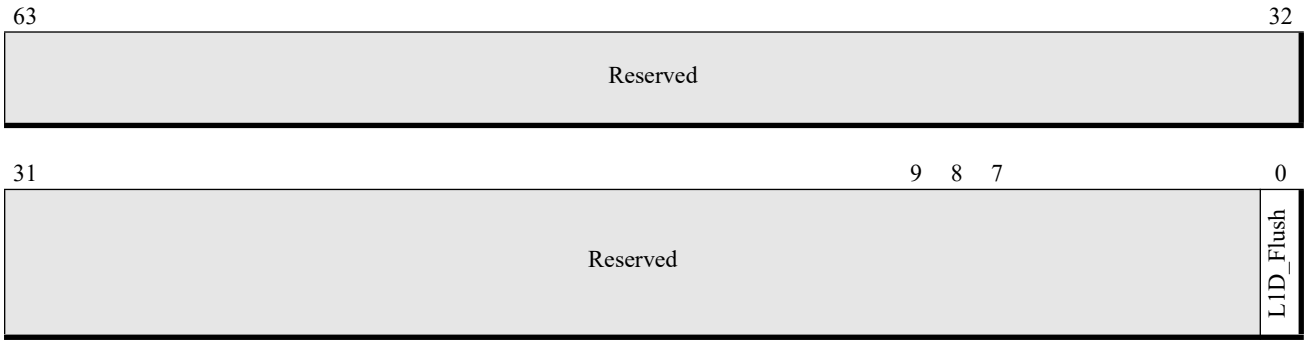
6.6.1 Cache Management

WBINVD and WBNOINVD Instructions. The *writeback and invalidate* (WBINVD) and *writeback no invalidate* (WBNOINVD) instructions are used to write all modified cache lines to memory so that memory contains the most recent copy of data. After the writes are complete, the WBINVD instruction invalidates all cache lines, whereas the WBNOINVD instruction may leave the lines in the cache hierarchy in a non-modified state. These instructions operate on all caches in the memory hierarchy, including caches that are external to the processor. See the instructions' description in Volume 3 for further operational details.

[AMD Public Use]

INVD Instruction. The *invalidate* (INVD) instruction is used to invalidate all cache lines in all caches in the memory hierarchy. Unlike the WBINVD instruction, no modified cache lines are written to memory. The INVD instruction should only be used in situations where memory coherency is not required.

FLUSH_CMD MSR. The FLUSH_CMD (MSR 10Bh) is a write-only register. Attempts to read this register or to write a 1 into any unsupported bits causes a #GP(0) fault. A WRMSR to FLUSH_CMD does not serialize memory operations. However, a write to this register is dispatch serializing and prevents execution of younger instructions until the WRMSR has completed. The format of the FLUSH_CMD register is shown in Figure 6-3 below.



Bits	Mnemonic	Description	Access type
63:1	Reserved		MBZ
0	L1D_Flush	L1 Data Cache write-back and invalidate	W

Figure 6-3. FLUSH_CMD Register

The FLUSH_CMD MSR has the following fields:

- *L1 Data Cache write-back and Invalidate (L1D_FLUSH)*—Bit 0, write only. Setting this bit to 1 causes the processor to perform a write-back and invalidate of the L1 Data Cache. Support for this bit is indicated by CPUID Fn0000_0007_EDX[L1D_FLUSH] (bit 28) = 1.

6.6.2 TLB Invalidation

INVLPG Instruction. The *invalidate TLB entry* (INVLPG) instruction can be used to invalidate specific entries within the TLB. The source operand is a virtual-memory address that specifies the TLB entry to be invalidated. Invalidating a TLB entry does not remove the associated page-table entry from the data cache. See “Translation-Lookaside Buffer (TLB)” on page 159 for more information.

INVLPGA Instruction. The *invalidate TLB entry in a Specified ASID* instruction (INVLPGA) can be used to invalidate TLB entries associated with the specified ASID. See “Invalidate Page, Alternate ASID” on page 552.

[AMD Public Use]

INVLPGB Instruction. The *invalidate TLB with Broadcast* instruction (INVLPGB) can be used to invalidate a specified range of TLB entries on the local processor and broadcast the invalidation to remote processors. See “INVLPGB” in Volume 3.

INVPCID Instruction. The *invalidate TLB entries in Specified PCID* instruction (INVPCID) can be used to invalidate TLB entries of the specified Processor Context ID. See “INVPCID” in Volume 3.

6.7 Shadow Stack Management

The following instructions are available to software for use in managing shadow stacks if the shadow stack feature is present as indicated by CPUID Fn0000_0007_x0_ECX[CET_SS] (bit 7) =1. Except for RDSSP, attempting to execute these instructions when shadow stacks are disabled results in a #UD exception. For more information refer to the detailed instruction descriptions in APM Volume 3.

CLRSSBSY. Validates a shadow stack token and clears the tokens busy bit. This is a privileged instruction.

INCSSP. Increment SSP by ‘n’ stack frames. Used to pop unneeded items from a shadow stack.

RDSSP. Read the SSP into a GPR. Treated as a NOP if shadow stacks are disabled.

RSTORSSP. Used to switch shadow stacks. Expects a ‘shadow stack restore token’ at the top of the new shadow stack. Upon validating this token, sets the token’s busy bit and sets SSP to the top of the new shadow stack.

SAVEPREVSSP. Copies a ‘previous SSP token’ from the current shadow stack back to the previous stack for later use by an RSTORSSP instruction.

SETSSBSY. Validates the shadow stack token pointed to by the PL0_SSP MSR. If valid, sets the busy bit to 1 and sets SSP = PL0_SSP. This is a privileged instruction.

WRSS. Writes the source operand to a shadow stack. This instruction must be enabled in the U_CET and S_CET MSRs, otherwise a #UD is generated.

WRUSS. Writes the source operand to a user shadow stack. This is a privileged instruction.

[AMD Public Use]

7 Memory System

This chapter describes:

- Cache coherency mechanisms
- Cache control mechanisms
- Memory typing
- Memory mapped I/O
- Memory ordering rules
- Serializing instructions

Figure 7-1 on page 187 shows a conceptual picture of a processor and memory system, and how data and instructions flow between the various components. This diagram is not intended to represent a specific microarchitectural implementation but instead is used to illustrate the major memory-system components covered by this chapter.

[AMD Public Use]

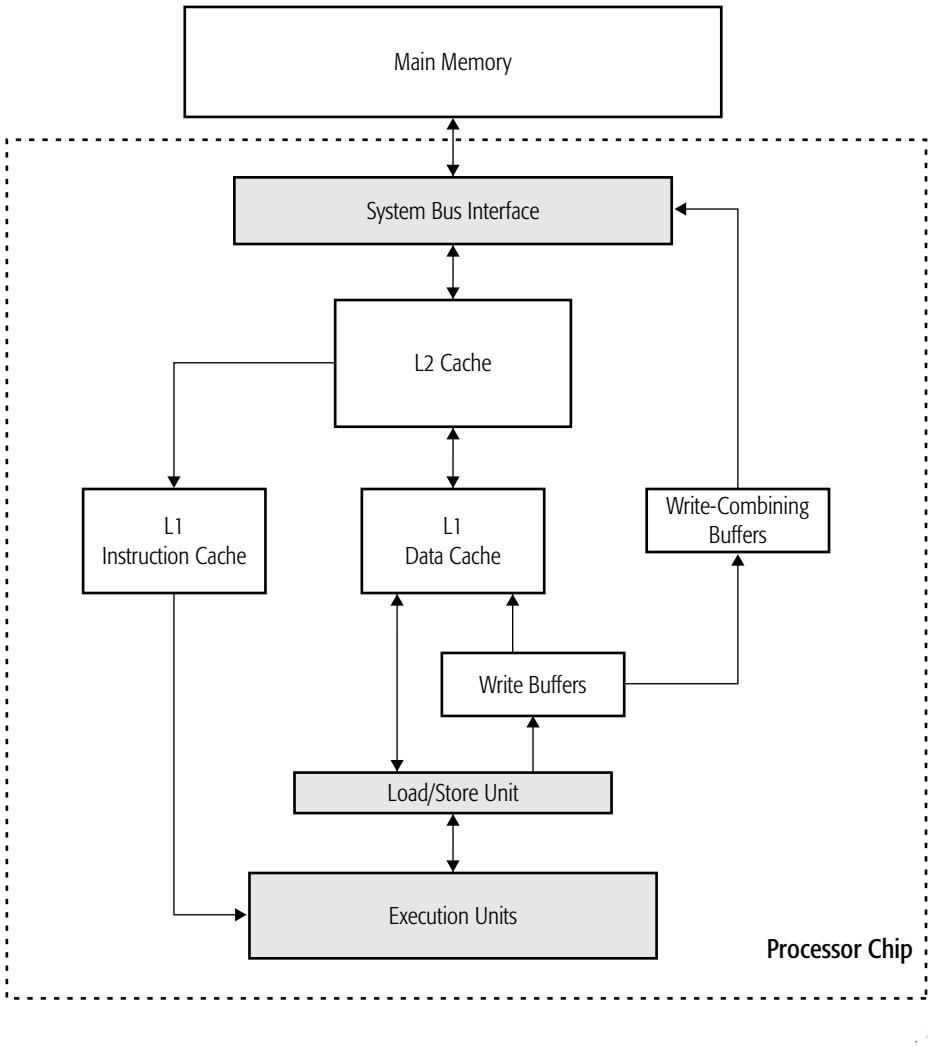


Figure 7-1. Processor and Memory System

The memory-system components described in this chapter are shown as *unshaded* boxes in Figure 7-1. Those items are summarized in the following paragraphs.

Main memory is external to the processor chip and is the memory-hierarchy level farthest from the processor execution units.

Caches are the memory-hierarchy levels closest to the processor execution units. They are much smaller and much faster than main memory, and can be either internal or external to the processor chip. Caches contain copies of the most frequently used instructions and data. By allowing fast access to frequently used data, software can run much faster than if it had to access that data from main memory. Figure 7-1 shows three caches, all internal to the processor:

[AMD Public Use]

- *L1 Data Cache*—The L1 (level-1) data cache holds the data most recently read or written by the software running on the processor.
- *L1 Instruction Cache*—The L1 instruction cache is similar to the L1 data cache except that it holds only the instructions executed most frequently. In some processor implementations, the L1 instruction cache can be combined with the L1 data cache to form a unified L1 cache.
- *L2 Cache*—The L2 (level-2) cache is usually several times larger than the L1 caches, but it is also slower. It is common for L2 caches to be implemented as a unified cache containing both instructions and data. Recently used instructions and data that do not fit within the L1 caches can reside in the L2 cache. The L2 cache can be exclusive, meaning it does not cache information contained in the L1 cache. Conversely, inclusive L2 caches contain a copy of the L1-cached information.

Memory-read operations from cacheable memory first check the cache to see if the requested information is available. A *read hit* occurs if the information is available in the cache, and a *read miss* occurs if the information is not available. Likewise, a *write hit* occurs if the memory write can be stored in the cache, and a *write miss* occurs if it cannot be stored in the cache.

Caches are divided into fixed-size blocks called *cache lines*. The cache allocates lines to correspond to regions in memory of the same size as the cache line, aligned on an address boundary equal to the cache-line size. For example, in a cache with 32-byte lines, the cache lines are aligned on 32-byte boundaries and byte addresses 0007h and 001Eh are both located in the same cache line. The size of a cache line is implementation dependent. Most implementations have either 32-byte or 64-byte cache lines. The implemented cache line size is reported by CPUID Fn8000_0005 and Fn8000_0006 for the various caches, as described in Appendix E of Volume 3.

The process of loading data into a cache is a *cache-line fill*. Even if only a single byte is requested, all bytes in a cache line are loaded from memory. Typically, a cache-line fill must remove (evict) an existing cache line to make room for the new line loaded from memory. This process is called *cache-line replacement*. If the existing cache line was modified before the replacement, the processor performs a cache-line *writeback* to main memory when it performs the cache-line fill.

Cache-line writebacks help maintain *coherency* between the caches and main memory. Internally, the processor can also maintain cache coherency by *internally probing* (checking) the other caches and write buffers for a more recent version of the requested data. External devices can also check processor caches for more recent versions of data by *externally probing* the processor. Throughout this document, the term *probe* is used to refer to external probes, while internal probes are always qualified with the word *internal*.

Write buffers temporarily hold data writes when main memory or the caches are busy with other memory accesses. The existence of write buffers is implementation dependent.

Implementations of the architecture can use *write-combining buffers* if the order and size of non-cacheable writes to main memory is not important to the operation of software. These buffers can combine multiple, individual writes to main memory and transfer the data in fewer bus transactions.

[AMD Public Use]

7.1 Single-Processor Memory Access Ordering

The flexibility with which memory accesses can be ordered is closely related to the flexibility in which a processor implementation can *execute* and *retire* instructions. Instruction execution *creates* results and status and determines whether or not the instruction causes an exception. Instruction retirement *commits* the results of instruction execution, in program order, to software-visible resources such as memory, caches, write-combining buffers, and registers, or it causes an exception to occur if instruction execution created one.

Implementations of the AMD64 architecture retire instructions in program order, but implementations can execute instructions in any order, subject only to data dependencies. Implementations can also *speculatively execute* instructions—executing instructions before knowing they are needed. Internally, implementations manage data reads and writes so that instructions complete in order. However, because implementations can execute instructions out of order and speculatively, the sequence of memory accesses performed by the hardware can appear to be out of program order. The following sections describe the rules governing memory accesses to which processor implementations adhere. These rules may be further restricted, depending on the memory type being accessed. Further, these rules govern single processor operation; see “Multiprocessor Memory Access Ordering” on page 191 for multiprocessor ordering rules.

7.1.1 Read Ordering

Generally, reads do not affect program order because they do not affect the state of software-visible resources other than register contents. However, some system devices might be sensitive to reads. In such a situation software can map a read-sensitive device to a memory type that enforces strong read-ordering, or use read/write barrier instructions to force strong read-ordering.

For cacheable memory types, the following rules govern read ordering:

- Out-of-order reads are allowed to the extent that they can be performed transparently to software, such that the appearance of in-order execution is maintained. Out-of-order reads can occur as a result of out-of-order instruction execution or speculative execution. The processor can read memory and perform cache refills out-of-order to allow out-of-order execution to proceed.
- Speculative reads are allowed. A speculative read occurs when the processor begins executing a memory-read instruction before it knows the instruction will actually complete. For example, the processor can predict a branch will occur and begin executing instructions following the predicted branch before it knows whether the prediction is valid. When one of the speculative instructions reads data from memory, the read itself is speculative. Cache refills may also be performed speculatively.
- Reads can be reordered ahead of writes. Reads are generally given a higher priority by the processor than writes because instruction execution stalls if the read data required by an instruction is not immediately available. Allowing reads ahead of writes usually maximizes software performance.
- A read *cannot* be reordered ahead of a prior write if the read is from the same location as the prior write. In this case, the read instruction stalls until the write instruction completes execution. The

[AMD Public Use]

read instruction requires the result of the write instruction for proper software operation. For cacheable memory types, the write data can be forwarded to the read instruction before it is actually written to memory.

- Instruction fetching constitutes a parallel, asynchronous stream of reads that is independent from and unordered with respect to the read accesses performed by loads in that instruction stream.

7.1.2 Write Ordering

Writes affect program order because they affect the state of software-visible resources. The following rules govern write ordering:

- Generally, out-of-order writes are *not* allowed. Write instructions executed out of order cannot commit (write) their result to memory until all previous instructions have completed in program order. The processor can, however, hold the result of an out-of-order write instruction in a private buffer (not visible to software) until that result can be committed to memory.
- It is possible for writes to *write-combining* memory types to appear to complete out of order, relative to writes into other memory types. See “Memory Types” on page 198 and “Write Combining” on page 205 for additional information.
- Speculative writes are *not* allowed. As with out-of-order writes, speculative write instructions cannot commit their result to memory until all previous instructions have completed in program order. Processors can hold the result in a private buffer (not visible to software) until the result can be committed.
- Write buffering is allowed. When a write instruction completes and commits its result, that result can be buffered until it is actually written to system memory in program order. Although the write buffer itself is not directly accessible by software, the results in the buffer are accessible by subsequent memory accesses to the locations that are buffered, including reads for which only a subset of bytes being accessed are in the buffer. For example, a doubleword read that overlaps a single modified byte in the write buffer can return the buffered value for that byte before that write has been committed to memory.

  In general, any read from cacheable memory returns the net result of all prior globally and locally visible writes to those bytes, as performed in program order. A given implementation may provide bytes from the write buffer to satisfy this, or may stall the read until any overlapping buffered writes have been committed to memory. For cacheable memory types, the write buffer can be read out-of-order and speculatively, just like memory.
- Write combining is allowed. In some situations software can relax the write-ordering rules through the use of a Write Combining memory type or non-temporal store instructions, and allow several writes to be combined into fewer writes to memory. When write-combining is used, it is possible for writes to other memory types to proceed ahead of (out-of-order) memory-combining writes, unless the writes are to the same address. Write-combining should be used only when the order of writes does not affect program order (for example, writes to a graphics frame buffer).

[AMD Public Use]

7.1.3 Read/Write Barriers

When the order of memory accesses must be strictly enforced, software can use read/write barrier instructions to force reads and writes to proceed in program order. Read/write barrier instructions force all prior reads or writes to complete before subsequent reads or writes are executed. The LFENCE, SFENCE, and MFENCE instructions are provided as dedicated read, write, and read/write barrier instructions (respectively). Serializing instructions, I/O instructions, and locked instructions (including the implicitly locked XCHG instruction) can also be used as read/write barriers. Barrier instructions are useful for controlling ordering between differing memory types as well as within one memory type; see Section 7.3.1, “Special Coherency Considerations,” on page 196 for details.

Table 7-2 on page 200 summarizes the memory-access ordering possible for each memory type supported by the AMD64 architecture.

7.2 Multiprocessor Memory Access Ordering

The term memory ordering refers to the sequence in which memory accesses are performed by the memory system, as observed by all processors or programs.

To improve performance of applications, AMD64 processors can speculatively execute instructions out of program order and temporarily hold out-of-order results. However, certain rules are followed with regard to normal cacheable accesses on naturally aligned boundaries to WB memory.

In the examples below, all memory values are initialized to zero.

From the point of view of a program, in ascending order of priority:

- All loads, stores and I/O operations from a single processor appear to occur in program order to the code running on that processor and all instructions appear to execute in program order.
- Successive stores from a single processor are committed to system memory and visible to other processors in program order. A store by a processor cannot be committed to memory before a read appearing earlier in the program has captured its targeted data from memory. In other words, stores from a processor cannot be reordered to occur prior to a load preceding it in program order.

  In this context:

  - Loads do not pass previous loads (loads are not reordered). Stores do not pass previous stores (stores are not reordered)

Processor 0	**Processor 1**
Store A ← 1	Load B
Store B ← 1	Load A

    Load A cannot read 0 when Load B reads 1. (This rule may be violated in the case of loads as part of a string operation, in which one iteration of the string reads 0 for Load A while another iteration reads 1 for Load B.)
  - Stores do not pass loads

[AMD Public Use]

Processor 0	Processor 1
Load A	Load B
Store B ← 1	Store A ← 1

Load A and Load B cannot both read 1.

- Stores from a processor appear to be committed to the memory system in program order; however, stores can be delayed arbitrarily by store buffering while the processor continues operation. Therefore, stores from a processor may not appear to be sequentially consistent.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
…	…
Store A ← 2	Store B ← 2
…	…
Load B	Load A

Both Load A and Load B may read 1. Also, due to possible write combining one or both processors may not actually store a 1 at the designated location.

- Non-overlapping Loads may pass stores.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
Load B	Load A

All combinations of values (00, 01, 10, and 11) may be observed by Processors 0 and 1.

- Where sequential consistency is needed (for example in Dekker's algorithm for mutual exclusion), an MFENCE instruction should be used between the store and the subsequent load, or a locked access, such as XCHG, should be used for the store.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
MFENCE	MFENCE
Load B	Load A

Load A and Load B cannot both read 0.

- Loads that partially overlap prior stores may return the modified part of the load operand from the store buffer, combining globally visible bytes with bytes that are only locally visible. To ensure that such loads return only a globally visible value, an MFENCE or locked access must be used between the store and the dependent load, or the store or load must be performed with a locked operation such as XCHG.

- Stores to different locations in memory observed from two (or more) other processors will appear in the same order to all observers. Behavior such as that shown in this code example,

[AMD Public Use]

Processor 0	Processor 1	Processor X	Processor Y
Store A ← 1	Store B ← 1		
		Load A (1)	Load B (1)
		Load B (0)	Load A (0)

in which processor X sees store A from processor 0 before store B from processor 1, while processor Y sees store B from processor 1 before store A from processor 0, is not allowed.

- Dependent stores between different processors appear to occur in program order, as shown in the code example below.

Processor 0	Processor 1	Processor 2
Store A ← 1		
	Load A (1)	
	Store B ← 1	
		Load B (1)
		Load A (1)

If processor 1 reads a value from A (written by processor 0) before carrying out a store to B, and if processor 2 reads the updated value from B, a subsequent read of A must also be the updated value.

- The local visibility (within a processor) for a memory operation may differ from the global visibility (from another processor). Using a data bypass, a local load can read the result of a local store in a store buffer, before the store becomes globally visible. Program order is still maintained when using such bypasses.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
Load r1 A	Load r3 B
Load r2 B	Load r4 A

Load A in processor 0 can read 1 using the data bypass, while Load A in processor 1 can read 0. Similarly, Load B in processor 1 can read 1 while Load B in processor 0 can read 0. Therefore, the result r1 = 1, r2 = 0, r3 = 1 and r4 = 0 may occur. There are no constraints on the relative order of when the Store A of processor 0 is visible to processor 1 relative to when the Store B of processor 1 is visible to processor 0.

If a very strong memory ordering model is required that does not allow local store-load bypasses, an MFENCE instruction or a synchronizing instruction such as XCHG or a locked read-modify-write should be used between the store and the subsequent load. This enforces a memory ordering stronger than total store ordering.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
MFENCE	MFENCE

[AMD Public Use]

Processor 0	Processor 1
Load r1 A	Load r3 B
Load r2 B	Load r4 A

In this example, the MFENCE instruction ensures that any buffered stores are globally visible before the loads are allowed to execute, so the result r1 = 1, r2 = 0, r3 = 1 and r4 = 0 will not occur.

7.3 Memory Coherency and Protocol

Implementations that support caching support a cache-coherency protocol for maintaining coherency between main memory and the caches. The cache-coherency protocol is also used to maintain coherency in a multiprocessor or multi-mastering system. All processors and all external bus-mastering devices participate in this cache-coherency protocol. In this section, the word processor encompasses any bus-mastering devices as well as other processors that participate in this cache-coherency protocol.

The cache-coherency protocol supported by the AMD64 architecture is the MOESDIF (modified, owned, exclusive, shared, dirty, invalid, forward) protocol. The states of the MOESDIF protocol are:

- *Invalid* - A cache line in the Invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.
- *Exclusive* - A cache line in the Exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
- *Shared* - A cache line in the Shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the Shared state and one other processor may hold a copy of the data in the Owned or Forward state. If no other processor holds it in the Owned state, then the copy in main memory is also the most recent.
- *Forward* - A cache line in the Forward state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the Shared state, as well. The Forward state acts as a performance hint indicating which cache should respond with data when there is a broad casted probe.
- *Modified* - A cache line in the Modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy. The data has usually been changed after it was received into the cache.
- *Dirty* - A cache line in the Dirty state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy. The data has usually not been changed after it was received into the cache.
- *Owned* - A cache line in the Owned state holds the most recent, correct copy of the data. The Owned state is similar to the Shared state in that other processors can hold a copy of the most recent, correct data. Unlike the Shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the Owned state. Any other processors that hold the data are in the Shared state.

[AMD Public Use]

For the purpose of this section a read could be a load, an instruction fetch, or a read during address translation of page tables or other translation structures. A write could be a store, or a change made due to address translation, such as the setting of a page table's Accessed or Dirty bit.

Figure 7-2 on page 170 shows the general MOESDIF state transitions possible with various types of memory accesses. This is a logical software view of the possible state transitions of a cache-line, not a hardware view. Instruction-execution activity and external-bus transactions can both be used to modify the cache MOESDIF state in multiprocessing or multi-mastering systems.

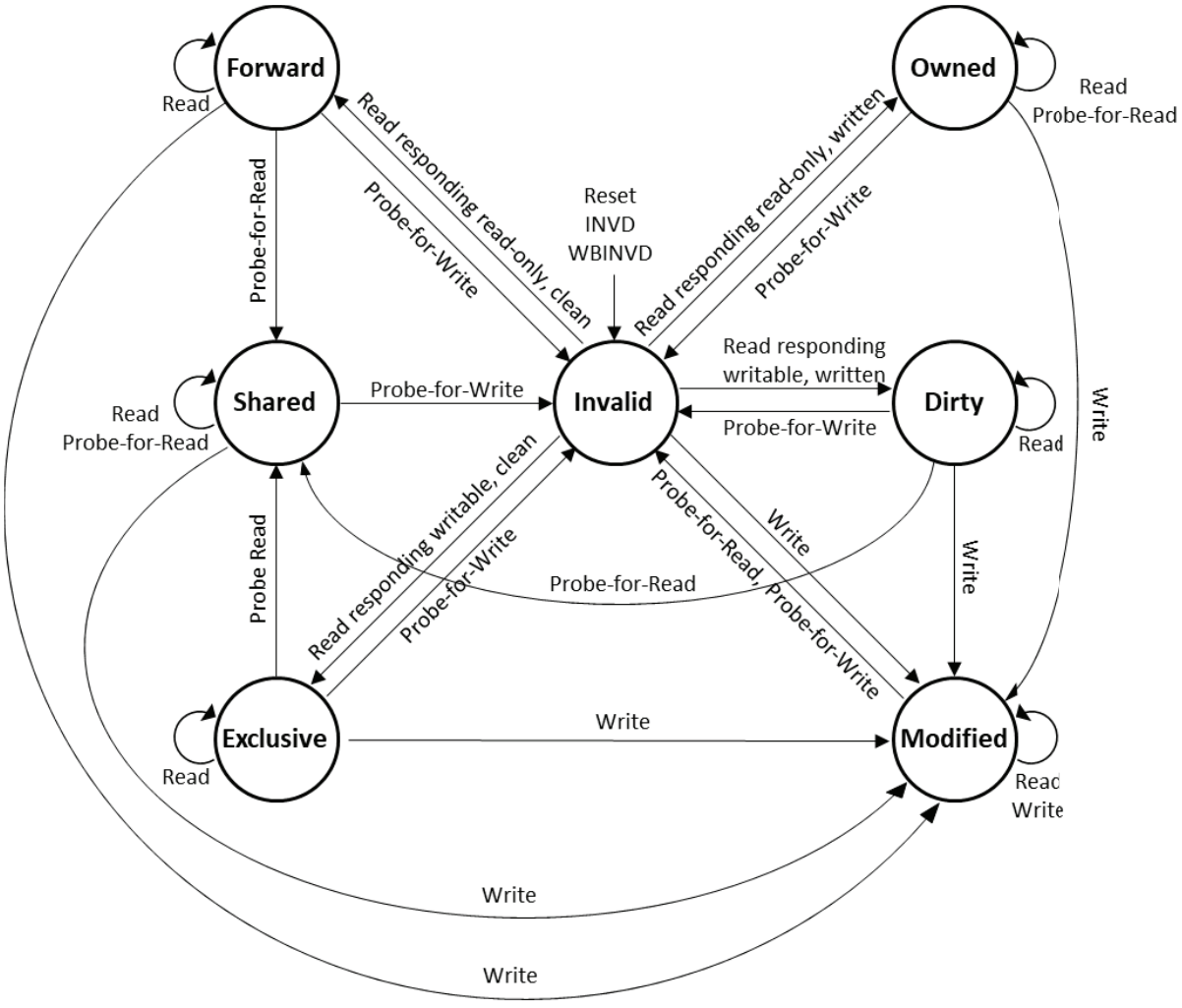


Figure 7-2. MOESDIF State Transitions

[AMD Public Use]

To maintain memory coherency, processors need to acquire the most recent copy of data before caching it internally. That copy can be in main memory or in the internal caches of other devices. When a processor has a cache read-miss or write-miss, it probes the other processors to determine whether the most recent copy of data is held in any of their caches and to perform cache state transitions if necessary. If one of the other caches holds the most recent copy, it may provide it to the requesting processor. The cache-line may be returned to the requesting processor in any of four conditions: read-only and clean, read-only and written, writable and clean, or writable and written. Otherwise, the most recent copy is provided by main memory.

There are two general types of probes:

- Probe-for-read indicates the external processor is requesting the data for read purposes.
- Probe-for-write indicates the external processor is requesting the data for the purpose of modifying it, or a processor has executed a CLFLUSH.

Referring back to Figure 7-2 on page 170, the state transitions involving probes are initiated by other processors or bus-mastering devices. In addition, some read probes are initiated by I/O devices that do not intend to cache the data. Some processor implementations do not change the MOESDIF state if the read probe is initiated by a device that does not intend to cache the data.

Read hits do not cause a MOESDIF-state change. Write hits generally cause a MOESDIF-state change into the Modified state and may require probe-for-write activity to cause cache state transitions on other processors that have the data. If the cache line is already in the Modified state, a write hit does not change its state.

A state transition caused by a read, probe-for-read, or probe-for-write can be caused by prefetching or speculative execution. Starting in family 17h, a transition to the Modified state does not occur speculatively, although the cache may still receive written data from another cache speculatively, putting the line into the Dirty state.

Some implementations may support a subset of the MOESDIF states. The specific operation of external-bus signals and transactions and how they influence a cache MOESDIF state are implementation dependent. For example, an implementation could convert a write miss to a WB memory type into two separate MOESDIF-state changes. The first would be a read-miss placing the cache line in the exclusive state. This would be followed by a write hit into the exclusive cache line, changing the cache-line state to Modified.

7.3.1 Special Coherency Considerations

In some cases, data can be modified in a manner that is impossible for the memory-coherency protocol to handle due to the effects of instruction prefetching. In such situations software must use serializing instructions and/or cache-invalidation instructions to ensure subsequent data accesses are coherent.

An example of this type of a situation is a page-table update followed by accesses to the physical pages referenced by the updated page tables. The following sequence of events shows what can happen when software changes the translation of virtual-page *A* from physical-page *M* to physical-page *N*:

[AMD Public Use]

1. Software invalidates the TLB entry. The tables that translate virtual-page *A* to physical-page *M* are now held only in main memory. They are not cached by the TLB.
2. Software changes the page-table entry for virtual-page A in main memory to point to physical-page *N* rather than physical-page *M*.
3. Software accesses data in virtual-page A.

During Step 3, software expects the processor to access the data from physical-page *N*. However, it is possible for the processor to prefetch the data from physical-page *M* before the page table for virtual-page *A* is updated in Step 2. This is because the physical-memory references for the *page tables* are different than the physical-memory references for the *data*. Because the physical-memory references are different, the processor does not recognize them as requiring coherency checking and believes it is safe to prefetch the data from virtual-page A, which is translated into a read from physical page M. Similar behavior can occur when instructions are prefetched from beyond the page table update instruction.

To prevent this problem, software must use an INVLPG or MOV CR3 instruction immediately after the page-table update to ensure that subsequent instruction fetches and data accesses use the correct virtual-page-to-physical-page translation. It is not necessary to perform a TLB invalidation operation preceding the table update.

7.3.2 Access Atomicity

Cacheable, naturally-aligned single loads or stores of up to a quadword are atomic on any processor model, as are misaligned loads or stores of less than a quadword that are contained entirely within a naturally-aligned quadword. Misaligned load or store accesses typically incur a small latency penalty. Model-specific relaxations of this quadword atomicity boundary, with respect to this latency penalty, may be found in a given processor's Software Optimization Guide.

Misaligned accesses can be subject to interleaved accesses from other processors or cache-coherent devices which can result in unintended behavior. Atomicity for misaligned accesses can be achieved where necessary by using the XCHG instruction or any suitable LOCK-prefixed instruction.

Processors that report CPUID Fn0000_0001_ECX[AVX](bit 28) = 1 extend the atomicity for cacheable, naturally-aligned single loads or stores from a quadword to a double quadword.

7.3.3 Cacheable Locks and Bus Locks

The processor guarantees access atomicity for locked read-modify-write operations. WB memory type, aligned locked read-modify-write (RMW) instructions are referred to as "cacheable locks". Cacheable locks are handled within a single processor and incur minimal or no performance penalty. Non-WB and misaligned locked RMW instructions are referred to as "bus locks" and require system-wide synchronization among all processors to guarantee atomicity. Bus locks may incur significant performance penalties for all processors in the system when any processor performs a non-WB or misaligned locked RMW instruction.

[AMD Public Use]

The processor performs a bus lock for any locked read-modify-write operations with a memory type other than WB, regardless of alignment.

The processor performs a bus lock for any misaligned locked read-modify-write operations, regardless of memory type. The alignment boundary at which a processor considers an access to be misaligned, and thus performs a bus lock instead of a cacheable lock for a locked read-modify-write operation, is implementation-dependent. A bus lock is always performed when an access spans an L1 data cache cacheline alignment boundary, but a processor may perform a bus lock at alignment boundaries smaller than the L1 data cache cacheline alignment boundary. The L1 data cache cacheline size and alignment boundary is specified by CPUID Fn8000_0005_ECX [L1 Data Cache Identifiers] (bits 7:0).

Atomic read-modify-write operations are performed on behalf of the types of operations listed in Table 7-1.

Table 7-1. Atomic Read-Modify-Write Operation

Category	Operations	Comments
Software atomic using a memory operand	XCHG, LOCK prefix + ADD, SUB, AND, OR, XOR, ADC, SBB, INC, DEC, NOT, NEG, BTC, BTR, BTS, XADD, CMPXCHG	Atomic read-modify-write on the memory operand
Segmentation and system data structures	LSL, LAR, VERR, VERW, LDS, LES, LFS, LGS, LSS, LTR, MOV DS, MOV ES, MOV FS, MOV GS, MOV SS, POP DS, POP ES, POP FS, POP GS, POP SS, Task switch	Setting Accessed bit in segment or task descriptor or TSS
Far control transfer	CALL (far), JMP (far), RET (far), IRET, INTn, INT3, INTO, INT1, exception or interrupt or trap gate	Setting Accessed bit in gate descriptor
Paging	Page table walk for instruction fetch or data access	Setting Accessed or Dirty bit in page table entries

The processor provides several methods for reporting bus lock activity to system software and reducing its frequency and performance penalty. Bus lock trap is described in section 8.2.2 on page 250 and section 13.1 on page 392, and SVM Bus Lock Threshold is described in section 15.14.5 on page 548.

7.4 Memory Types

Memory type is an attribute that can be associated with a specific region of virtual or physical memory. Memory type designates certain caching and ordering behaviors for loads and stores to addresses in that region. Most memory types are explicitly assigned, although some are inferred by the hardware from current processor state and instruction context.

The AMD64 architecture defines the following memory types:

- *Uncacheable (UC)*—Reads from, and writes to, UC memory are not cacheable. Reads from UC memory cannot be speculative. Write-combining to UC memory is not allowed. Reads from or

[AMD Public Use]

writes to UC memory cause the write buffers to be written to memory and be invalidated prior to the access to UC memory.

The UC memory type is useful for memory-mapped I/O devices where strict ordering of reads and writes is important. Note that this strong ordering is with respect to UC accesses only; reads to memory types which support speculative operation may bypass non-conflicting UC accesses.

- *Cache Disable (CD)*—The CD memory type is a form of uncacheable memory type that is inferred when the L1 caches are disabled but not invalidated, or for certain conflicting memory type assignments from the Page Attribute Table (PAT) and Memory Type Range Register (MTRR) mechanisms. The former case occurs when caches are disabled by setting CR0.CD to 1 without invalidating the caches with either the INVD or WBINVD instruction for any reference to a region designated as cacheable. The latter case occurs when a specific type has been assigned to a virtual page via PAT, and a conflicting type has been assigned to the mapped physical page via an MTRR (see “Combined Effect of MTRRs and PAT” on page 232 and “Combining Memory Types, MTRRs” on page 577 for details).

  For the L1 data cache and the L2 cache, reads from, and writes to, CD memory that hit the cache, or any other caches in the system, cause the cache line(s) to be invalidated before accessing main memory. If a cache line is in the modified state, the line is written to main memory prior to being invalidated. The access is allowed to proceed after any invalidations are complete.

  For the L1 instruction cache, instruction fetches from CD memory that hit the cache read the cached instructions rather than access main memory. Instruction fetches that miss the cache access main memory and do not cause cache-line replacement. Writes to CD memory that hit in the instruction cache cause the line to be invalidated.

- *Write-Combining (WC)*—Reads from, and writes to, WC memory are not cacheable. Reads from WC memory can be speculative.

  Writes to this memory type can be combined internally by the processor and written to memory as a single write operation to reduce memory accesses. For example, four word writes to consecutive addresses can be combined by the processor into a single quadword write, resulting in one memory access instead of four. Reads from this memory can also be combined internally by the processor if they are to the same cache line. WC reads are not combined across serializing or barrier instructions.

  The WC memory type is useful for graphics-display memory buffers where the order of writes is not important.

- *Write-Combining Plus (WC+)*—WC+ is an uncacheable memory type, and combines writes in write-combining buffers like WC. Unlike WC (but like the CD memory type), accesses to WC+ memory probe the caches on all processors (including the caches of the processor issuing the request) to maintain coherency. This ensures that cacheable writes are observed by WC+ accesses.

- *Write-Protect (WP)*—Reads from WP memory are cacheable and allocate cache lines on a read miss. Reads from WP memory can be speculative.

  Writes to WP memory that hit in the cache do not update the cache. Instead, all writes update memory (write to memory), and writes that hit in the cache invalidate the cache line. Write buffering of WP memory is allowed.

[AMD Public Use]

The WP memory type is useful for shadowed-ROM memory where updates must be immediately visible to all devices that read the shadow locations.

- *Writethrough (WT)*—Reads from WT memory are cacheable and allocate cache lines on a read miss. Reads from WT memory can be speculative.

  All writes to WT memory update main memory, and writes that hit in the cache update the cache line (cache lines remain in the same state after a write that hits a cache line). Writes that miss the cache do not allocate a cache line. Write buffering of WT memory is allowed.

- *Writeback (WB)*—Reads from WB memory are cacheable and allocate cache lines on a read miss. Cache lines can be allocated in the shared, exclusive, or modified states. Reads from WB memory can be speculative.

  All writes that hit in the cache update the cache line and place the cache line in the modified state. Writes that miss the cache allocate a new cache line and place the cache line in the modified state. Writes to main memory only take place during writeback operations. Write buffering of WB memory is allowed.

  The WB memory type provides the highest-possible performance and is useful for most software and data stored in system memory (DRAM).

Table 7-2 shows the memory access ordering possible for each memory type supported by the AMD64 architecture. Table 7-4 on page 203 shows the ordering behavior of various operations on various memory types in greater detail. Table 7-3 on page 201 shows the caching policy for the same memory types.

Table 7-2. Memory Access by Memory Type

Memory Access Allowed		Memory Type				
		UC/CD	WC	WP	WT	WB
Read	Out-of-Order	no	yes	yes	yes	yes
	Speculative	no	yes	yes	yes	yes
	Reorder Before Write	no	yes	yes	yes	yes
Write	Out-of-Order	no	yes	no	no	no
	Speculative	no	no	no	no	no
	Buffering	no	yes	yes	yes	yes
	Combining[1]	no	yes	no	yes	yes

Note:
1. *Write-combining buffers are separate from write (store) buffers.*

[AMD Public Use]

Table 7-3. Caching Policy by Memory Type

Caching Policy	Memory Type					
	UC	CD	WC	WP	WT	WB
Read Cacheable	no	no	no	yes	yes	yes
Write Cacheable	no	no	no	no	yes	yes
Read Allocate	no	no	no	yes	yes	yes
Write Allocate	no	no	no	no	no	yes
Write Hits Update Memory	yes[2]	yes[1]	yes[2]	yes[3]	yes	no

Note:

1. *For the L1 data cache and the L2 cache, if an access hits the cache, the cache line is invalidated. If the cache line is in the modified state, the line is written to main memory and then invalidated. For the L1 instruction cache, read (instruction fetch) hits access the cache rather than main memory.*
2. *The data is not cached, so a cache write hit cannot occur. However, memory is updated.*
3. *Write hits update memory and invalidate the cache line.*

7.4.1 Instruction Fetching from Uncacheable Memory

Instruction fetches from an uncacheable memory type (including those for the CD type which don't hit in the instruction cache) may read a naturally-aligned block of memory no larger than an instruction cache line that contains multiple instructions, and may or may not repeat reads of a given block in the course of extracting instructions from it. In general, the exact sequence of read accesses is not deterministic, regardless of instruction stream contents, aside from the following constraints:

- instruction fetching of branch targets from uncacheable memory will only be done non-speculatively
- sequential instruction fetching will not transition speculatively from a cacheable memory type to an uncacheable memory type
- sequential instruction fetching will not speculatively cross more than one 4KB page boundary

It is recommended that MMIO devices that have read side-effects be separated from memory that's subject to uncacheable instruction fetches by at least one 4KB page.

7.4.2 Memory Barrier Interaction with Memory Types

Memory types other than WB may allow weaker ordering in certain respects. When the ordering of memory accesses to differing memory types must be strictly enforced, software can use the LFENCE, MFENCE or SFENCE barrier instructions to force loads and stores to proceed in program order. Table 7-4 on page 203 summarizes the cases where a memory barrier must be inserted between two memory operations.

The table is read as follows: the ROW is the first memory operation in program order, followed by the COLUMN, which is the second memory operation in program order. Each cell represents the ordered combination of the two memory operations and the letters *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, and *l* within the cell represent the applicable memory ordering rule for that combination. These symbols are described in

[AMD Public Use]

the footnotes below the table. In the table and footnotes, the abbreviation *nt* stands for non-temporal (load or store), *io* stands for input / output, *lf* for LFENCE, *sf* for SFENCE, and *mf* for MFENCE.

7.4.3 Floating Point Instructions and Memory Types

Uncacheable (UC) and Cache Disable (CD) memory types should not be used with x87, MMX, or SIMD (SSE, AVX, and AVX512) instructions. These instructions may read or write a memory location multiple times, which can have undesired side effects with these memory types.

[AMD Public Use]

Table 7-4. Memory Access Ordering Rules

	Second Memory Operation								
First Memory Operation	**Load (wp, wt, wb)**	**Load (uc)**	**Load (wc, wc+)**	**Store (wp, wt, wb)**	**Store (uc)**	**Store (wc, wc+, non-temporal)**	**Load/Store (io)**	**Lock (atomic)**	**Serialize instructions/ Interrupts/Exceptions**
Load (wp, wt, wb)	a	f	b (lf)	c	c	c	d	d	d
Load (uc)	a	f	b (lf)	c	c	c	d	d	d
Load (wc, wc+)	a	f	b (lf)	c	c	c	d	d	d
Store (wp, wt, wb)	e (mf)	f	e (mf)	g	g	h (sf)	d	d	d
Store (uc)	i	f	i	g	g	h (sf)	d	d	d
Store (wc, wc+, non-temporal)	e (mf)	f	e (mf)	j (sf)	g, m	h (sf)	d	d	d
Load/Store (io)	k	k	k	k	k	l	d, k	d, k	d, k
Lock (atomic)	k	k	k	k	k	k	d, k	d, k	d, k
Serialize instruction/ Interrupts/Exceptions	l	l	l	l	l	l	d, l	d, l	d, l

a — A load (wp, wt, wb) may not pass a previous load (wp, wt, wb, wc, wc+, uc).

b — A load (wc, wc+) may pass a previous load (wp, wt, wb, wc, wc+). To ensure memory order, an LFENCE instruction must be inserted between the two loads.

c — A store (wp, wt, wb, uc, wc, wc+, nt) may not pass a previous load (wp, wt, wb, uc, wc, wc+, nt).

d — All previous loads and stores complete to memory or I/O space before a memory access for an I/O, locked or serializing instruction is issued.

e — A load (wp, wt, wb, wc, wc+) may pass a previous non-conflicting store (wp, wt, wb, wc, wc+, nt). To ensure memory order, an MFENCE instruction must be inserted between the store and the load.

f — A load or store (uc) does not pass a previous load or store (wp, wt, wb, uc, wc, wc+, nt).

g — A store (wp, wt, wb, uc) does not pass a previous store (wp, wt, wb, uc).

h — A store (wc, wc+, nt) may pass a previous store (wp, wt, wb) or non-conflicting store (wc, wc+, nt). To ensure memory order, an SFENCE instruction must be inserted between these two stores. A store (wc, wc+, nt) does not pass a previous conflicting store (wc, wc+, nt, uc).

i — A load (wp, wt, wb, wc, wc+) may pass a previous non-conflicting store (uc). To ensure memory order, an MFENCE instruction must be inserted between the store and the load.

j — A store (wp, wt, wb) may pass a previous store (wc, wc+, nt). To ensure memory order, an SFENCE instruction must be inserted between these two stores.

k — All loads and stores associated with the I/O and locked instructions complete to memory (no buffered stores) before a load or store from a subsequent instruction is issued.

[AMD Public Use]

l — All loads and stores complete to memory for the serializing instruction before the subsequent instruction fetch is issued.

m — A store (uc) does not pass a previous store (wc, wc+).

7.5 Buffering and Combining Memory Writes

7.5.1 Write Buffering

Writes to memory (main memory and caches) can be stored internally by the processor in *write buffers* (also known as store buffers) before actually writing the data into a memory location. System performance can be improved by buffering writes, as shown in the following examples:

- When higher-priority memory transactions, such as reads, compete for memory access with writes, writes can be delayed in favor of reads, which minimizes or eliminates an instruction-execution stall due to a memory-operand read.
- When the memory is busy, buffering writes while the memory is busy removes the writes from the instruction-execution pipeline, which frees instruction-execution resources.

The processor manages the write buffer so that it is transparent to software. Memory accesses check the write buffer, and the processor completes writes into memory from the buffer in program order. Also, the processor completely empties the write buffer by writing the contents to memory as a result of performing any of the following operations:

- *SFENCE Instruction*—Executing a store-fence (SFENCE) instruction forces all memory writes before the SFENCE (in program order) to be written into memory (or, for WB type, the cache) before memory writes that follow the SFENCE instruction. The memory-fence (MFENCE) instruction has a similar effect, but it forces the ordering of loads in addition to stores.
- *Serializing Instructions*—Executing a serializing instruction forces the processor to retire the serializing instruction (complete both instruction execution and result writeback) before the next instruction is fetched from memory.
- *I/O instructions*—Before completing an I/O instruction, all previous reads and writes must be written to memory, and the I/O instruction must complete before completing subsequent reads or writes. Writes to I/O-address space (OUT instruction) are never buffered.
- *Locked Instructions*—A locked instruction (an instruction executed using the LOCK prefix) or an XCHG instruction (which is implicitly locked) must complete *after* all previous reads and writes and *before* subsequent reads and writes. Locked writes are never buffered, although locked reads and writes are cacheable.
- *Interrupts and Exceptions*—Interrupts and exceptions, including virtualization intercepts (#VMEXIT), are serializing events that force the processor to write all results from the write buffer to memory before fetching the first instruction from the interrupt or exception service routine.
- *UC Memory Reads*—UC memory reads are not reordered ahead of writes.

[AMD Public Use]

Write buffers can behave similarly to *write-combining buffers* because multiple writes may be collected internally before transferring the data to caches or main memory. See the following section for a description of write combining.

7.5.2 Write Combining

Write-combining memory uses a different buffering scheme than write buffering described above. Writes to write-combining (WC) memory can be combined internally by the processor in a buffer for more efficient transfer to main memory at a later time. For example, 16 doubleword writes to consecutive memory addresses can be combined in the WC buffers and transferred to main memory as a single burst operation rather than as individual memory writes.

The following instructions perform writes to WC memory:

- (V)MASKMOVDQU
- MASKMOVQ
- (V)MOVNTDQ
- MOVNTI
- (V)MOVNTPD
- (V)MOVNTPS
- MOVNTQ
- MOVNTSD
- MOVNTSS

WC memory is not cacheable. A WC buffer writes its contents only to main memory.

The size and number of WC buffers available is implementation dependent. The processor assigns an address range to an empty WC buffer when a WC-memory write occurs. The size and alignment of this address range is equal to the buffer size. All subsequent writes to WC memory that fall within this address range can be stored by the processor in the WC-buffer entry until an event occurs that causes the processor to write the WC buffer to main memory. After the WC buffer is written to main memory, the processor can assign a new address range on a subsequent WC-memory write.

Writes to consecutive addresses in WC memory are not required for the processor to combine them. The processor combines any WC memory write that falls within the active-address range for a buffer. Multiple writes to the same address overwrite each other (in program order) until the WC buffer is written to main memory.

It is possible for writes to proceed out of program order when WC memory is used. For example, a write to cacheable memory that follows a write to WC memory can be written into the cache before the WC buffer is written to main memory. For this reason, and the reasons listed in the previous paragraph, software that is sensitive to the order of memory writes should avoid using WC memory.

WC buffers are written to main memory under the same conditions as the write buffers, namely when:

- Executing a store-fence (SFENCE) instruction.

[AMD Public Use]

- Executing a serializing instruction.
- Executing an I/O instruction.
  - Executing an MMIO access (load or store to UC memory type)
- Executing a locked instruction (an instruction executed using the LOCK prefix).
- Executing an XCHG instruction
- An interrupt or exception occurs.

WC buffers are also written to main memory when:

- A subsequent non-write-combining operation has a write address that matches the WC-buffer active-address range.
- A write to WC memory falls outside the WC-buffer active-address range. The existing buffer contents are written to main memory, and a new address range is established for the latest WC write.

7.6 Memory Caches

The AMD64 architecture supports the use of internal and external caches. The size, organization, coherency mechanism, and replacement algorithm for each cache is implementation dependent. Generally, the existence of the caches is transparent to both application and system software. In some cases, however, software can use cache-structure information to optimize memory accesses or manage memory coherency. Such software can use the extended-feature functions of the CPUID instruction to gather information on the caching subsystem supported by the processor. For more information, see Section 3.3, “Processor Feature Identification,” on page 73.

7.6.1 Cache Organization and Operation

Although the detailed organization of a processor cache depends on the implementation, the general constructs are similar. L1 caches—data and instruction, or unified—and L2 caches usually are implemented as n-way set-associative caches. Figure 7-3 on page 207 shows a typical *logical* organization of an n-way set-associative cache. The physical implementation of the cache can be quite different.

[AMD Public Use]

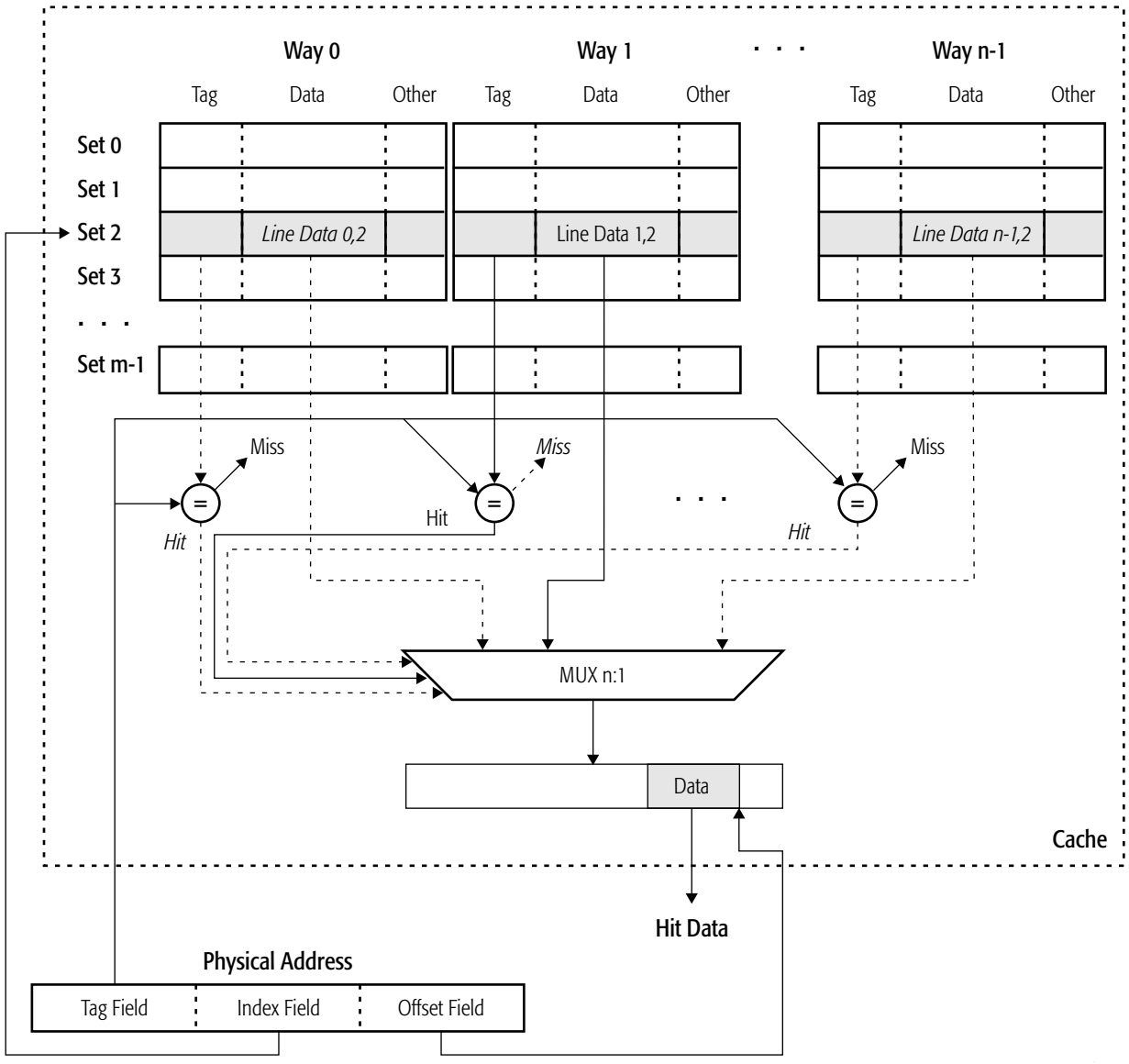


Figure 7-3. Cache Organization Example

As shown in Figure 7-3, the cache is organized as an array of cache lines. Each cache line consists of three parts: a cache-data line (a fixed-size copy of a memory block), a tag, and other information. Rows of cache lines in the cache array are *sets*, and columns of cache lines are *ways*. In an n-way set-associative cache, each set is a collection of n lines. For example, in a four-way set-associative cache, each set is a collection of four cache lines, one from each way.

[AMD Public Use]

The cache is accessed using the physical address of the data or instruction being referenced. To access data within a cache line, the physical address is used to select the set, way, and byte from the cache. This is accomplished by dividing the physical address into the following three fields:

- *Index*—The *index field* selects the cache set (row) to be examined for a hit. All cache lines within the set (one from each way) are selected by the index field.
- *Tag*—The *tag field* is used to select a specific cache line from the cache set. The physical-address tag field is compared with each cache-line tag in the set. If a match is found, a cache hit is signaled, and the appropriate cache line is selected from the set. If a match is not found, a cache miss is signaled.
- *Offset*—The *offset field* points to the first byte in the cache line corresponding to the memory reference. The referenced data or instruction value is read from (or written to, in the case of memory writes) the selected cache line starting at the location selected by the offset field.

In Figure 7-3 on page 207, the physical-address index field is shown selecting Set 2 from the cache. The tag entry for each cache line in the set is compared with the physical-address tag field. The tag entry for Way 1 matches the physical-address tag field, so the cache-line data for Set 2, Way 1 is selected using the n:1 multiplexer. Finally, the physical-address offset field is used to point to the first byte of the referenced data (or instruction) in the selected cache line.

Cache lines can contain other information in addition to the data and tags, as shown in Figure 7-3 on page 207. MOESI state and the state bits associated with the cache-replacement algorithm are typical pieces of information kept with the cache line. Instruction caches can also contain pre-decode or branch-prediction information. The type of information stored with the cache line is implementation dependent.

Self-Modifying Code. Software that stores into its own pending instruction stream with the intent of then executing the modified instructions is classified as self-modifying code. To support self-modifying code, AMD64 processors will flush any lines from the instruction cache that such stores hit, and will additionally check whether an instruction being modified is already in decode or execution behind the store instruction. If so, it will flush the pipeline and restart instruction fetch to acquire and re-decode the updated instruction bytes. No special action is needed by software for such updates to be immediately recognized. As with cache coherency, the check for instructions that are in flight is performed using physical addresses to avoid aliasing issues that could arise with virtual (linear) addresses.

When the modified bytes are in cacheable memory, the data cache may retain a copy of the modified cache line in a shared state, and the instruction cache refill may be satisfied from any suitable place in the memory hierarchy in a model-dependent manner that maintains cache coherency.

Cross-Modifying Code. Software that stores into the active instruction stream of another executing thread with the intent that the other thread subsequently execute the modified instruction stream is classified as cross-modifying code. There are two approaches to consider: asynchronous modification and synchronous modification.

[AMD Public Use]

Asynchronous modification. This is done with a write to the target instruction stream with no particular coordination being done between the writing and receiving threads. The nature of the code being executed by the target thread is such that it is insensitive to the exact timing of the update, for example executing in a known loop until an update to a branch instruction's offset takes it down a new path (or an update to an immediate operand, or opcode, or other instruction field). Such modifications must be done via a single store to the target thread's instruction stream that is contained entirely within a naturally-aligned quadword, and is subject to the constraints given here. A key aspect is that, although the store is performed atomically, the affected quadword may be read more than once in the process of extracting instruction bytes from it. This can result in the following scenarios resulting from a single store:

1. An update to two successive instructions, A and B, to A' and B' may result in execution of an A-B' sequence rather than A'-B'. However it will not result in an A'-B sequence since stores become visible to instruction fetchers in program order, and instruction fetchers read memory sequentially between taken branches.
2. A modification to one instruction A that changes it to two instructions A'-B will only result in execution of A'-B.
3. A modification to two instructions A-B that combines them into one instruction A' may result in a sequence of A-X, where X starts at the point in A' where B previously started.

Note that since stores to the instruction stream are observed by the instruction fetcher in program order, one can do multiple modifications to an area of the target thread's code that is beyond reach of the thread's current control flow, followed by a final asynchronous update that alters the control flow to expose the modified code to fetching and execution.

If the desired action cannot be achieved within these constraints, a synchronous modification approach must be used for reliable operation.

Synchronous modification. This entails a producer-consumer approach to the modification, where the target thread waits on a signal from the modifying thread, such as changing the state of a shared variable, before executing the modified code. The modifying thread writes to the target instruction bytes in any desired manner, then writes the synchronizing variable to release the target thread. Upon release, the target thread must then execute a serializing instruction such as CPUID or MFENCE (a locked operation is not sufficient) before proceeding to the modified code to avoid executing a stale view of the instructions which may have been speculatively fetched. Note that such speculative fetching is a function of branch predictor operation which is completely beyond the control of software.

See Volume 1, Chapter 3, “Semaphores,” for a discussion of instructions that are useful for interprocessor synchronization.

7.6.2 Cache Control Mechanisms

The AMD64 architecture provides a number of mechanisms for controlling the cacheability of memory. These are described in the following sections.

[AMD Public Use]

Cache Disable. Bit 30 of the CR0 register is the cache-disable bit, CR0.CD. Caching is enabled when CR0.CD is cleared to 0, and caching is disabled when CR0.CD is set to 1. When caching is disabled, reads and writes access main memory.

Software can disable the cache while the cache still holds valid data (or instructions). If a read or write hits the L1 data cache or the L2 cache when CR0.CD=1, the processor does the following:

1. Writes the cache line back if it is in the modified or owned state.
2. Invalidates the cache line.
3. Performs a non-cacheable main-memory access to read or write the data.

If an instruction fetch hits the L1 instruction cache when CR0.CD=1, some processor models may read the cached instructions rather than access main memory. When CR0.CD=1, the exact behavior of L2 and L3 caches is model-dependent, and may vary for different types of memory accesses.

The processor also responds to cache probes when CR0.CD=1. Probes that hit the cache cause the processor to perform Step 1. Step 2 (cache-line invalidation) is performed only if the probe is performed on behalf of a memory write or an exclusive read.

Writethrough Disable. Bit 29 of the CR0 register is the *not writethrough* disable bit, CR0.NW. In early x86 processors, CR0.NW is used to control cache writethrough behavior, and the combination of CR0.NW and CR0.CD determines the cache operating mode.

In early x86 processors, clearing CR0.NW to 0 enables writeback caching for main memory, effectively disabling writethrough caching for main memory. When CR0.NW=0, software can disable writeback caching for specific memory pages or regions by using other cache control mechanisms. When software sets CR0.NW to 1, writeback caching is disabled for main memory, while writethrough caching is enabled.

In implementations of the AMD64 architecture, CR0.NW is not used to qualify the cache operating mode established by CR0.CD. Table 7-5 shows the effects of CR0.NW and CR0.CD on the AMD64 architecture cache-operating modes.

Table 7-5. AMD64 Architecture Cache-Operating Modes

CR0.CD	CR0.NW	Cache Operating Mode
0	0	Cache enabled with a writeback-caching policy.
0	1	Invalid setting—causes a general-protection exception (#GP).
1	0	Cache disabled. See “Cache Disable” on page 210.
1	1	

Page-Level Cache Disable. Bit 4 of all paging data-structure entries controls page-level cache disable (PCD). When a data-structure-entry PCD bit is cleared to 0, the page table or physical page pointed to by that entry is cacheable, as determined by the CR0.CD bit. When the PCD bit is set to 1, the page table or physical page is not cacheable. The PCD bit in the paging data-structure base-register

[AMD Public Use]

(bit 4 in CR3) controls the cacheability of the highest-level page table in the page-translation hierarchy.

Page-Level Writethrough Enable. Bit 3 of all paging data-structure entries is the page-level writethrough enable control (PWT). When a data-structure-entry PWT bit is cleared to 0, the page table or physical page pointed to by that entry has a writeback caching policy. When the PWT bit is set to 1, the page table or physical page has a writethrough caching policy. The PWT bit in the paging data-structure base-register (bit 3 in CR3) controls the caching policy of the highest-level page table in the page-translation hierarchy.

The corresponding PCD bit must be cleared to 0 (page caching enabled) for the PWT bit to have an effect.

Memory Typing. Two mechanisms are provided for software to control access to and cacheability of specific memory regions:

- The memory-type range registers (MTRRs) control cacheability based on physical addresses. See “MTRRs” on page 220 for more information on the use of MTRRs.
- The page-attribute table (PAT) mechanism controls cacheability based on virtual addresses. PAT extends the capabilities provided by the PCD and PWT page-level cache controls. See “Page-Attribute Table Mechanism” on page 229 for more information on the use of the PAT mechanism.

System software can combine the use of both the MTRRs and PAT mechanisms to maximize control over memory cacheability.

If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0. Cacheable memory types must be established using the MTRRs in order for memory accesses to be cached.

Cache Control Precedence. The cache-control mechanisms are used to define the memory type and cacheability of main memory and regions of main memory. Taken together, the most restrictive memory type takes precedence in defining the caching policy of memory. The order of precedence is:

1. Uncacheable (UC)
2. Write-combining (WC)
3. Write-protected (WP)
4. Writethrough (WT)
5. Writeback (WB)

For example, assume a large memory region is designated a writethrough type using the MTRRs. Individual pages within that region can have caching disabled by setting the appropriate page-table PCD bits. However, no pages within that region can have a writeback caching policy, regardless of the page-table PWT values.

[AMD Public Use]

7.6.3 Cache and Memory Management Instructions

Data Prefetch. The prefetch instructions are used by software as a hint to the processor that the referenced data is likely to be used in the near future. The processor can preload the cache line containing the data in anticipation of its use. PREFETCH provides a hint that the data is to be read. PREFETCHW provides a hint that the data is to be written. The processor can mark the line as modified if it is preloaded using PREFETCHW.

Memory Ordering. Instructions are provided for software to enforce memory ordering (serialization) in weakly-ordered memory types. These instructions are:

- *SFENCE (store fence)*—forces all memory writes (stores) preceding the SFENCE (in program order) to be written into memory before memory writes following the SFENCE.
- *LFENCE (load fence)*—forces all memory reads (loads) preceding the LFENCE (in program order) to be read from memory before memory reads following the LFENCE. In some systems, LFENCE may be configured to be dispatch serializing. In systems where CPUID Fn8000_0021_EAX[LFenceAlwaysSerializing] (bit 2) = 1, LFENCE is always dispatch serializing.
- *MFENCE (memory fence)*—forces all memory accesses (reads and writes) preceding the MFENCE (in program order) to be written into or read from memory before memory accesses following the MFENCE.

Cache Line Writeback and Flush. The CLFLUSH instruction (writeback, if modified, and invalidate) takes the byte memory-address operand (a linear address), and checks to see if the address is cached. If the address is cached, the entire cache line containing the address is invalidated. If any portion of the cache line is dirty (in the modified or owned state), the entire line is written to main memory before it is invalidated. CLFLUSH affects *all caches* in the memory hierarchy—internal and external to the processor, and across all cores. The CLWB instruction operates in the same manner except it does not invalidate the cache line. The checking and invalidation process continues until the address has been updated in memory and, for CLFLUSH, invalidated in all caches.

In most cases, the underlying memory type assigned to the address has no effect on the behavior of this instruction. However, when the underlying memory type for the address is UC or WC (as defined by the MTRRs), the processor does not proceed with checking all caches to see if the address is cached. In both cases, the address is uncacheable, and invalidation is unnecessary. Write-combining buffers are written back to memory if the corresponding physical address falls within the buffer active-address range.

Cache Writeback and Invalidate. Unlike the CLFLUSH and CLWB instructions, the WBINVD and WBNOINVD instructions operate on the entire cache, rather than a single cache line. The WBINVD and WBNOINVD instructions first write back all cache lines that are dirty (in the modified or owned state) to main memory. After writeback is complete, the WBINVD instruction additionally invalidates all cache lines. The checking and invalidation process continues until all internal caches in the executing core's path to system memory are invalidated. In some implementations this may include caches in other branches of the system's cache hierarchy; see the description of these instructions in

[AMD Public Use]

Volume 3 for more detail. For either instruction, a special bus cycle is transmitted to higher-level external caches directing them to perform a writeback-and-invalidate operation.

Cache Invalidate. The INVD instruction is used to invalidate all cache lines. Unlike the WBINVD instruction, dirty cache lines are not written to main memory. The process continues until all internal caches have been invalidated. A special bus cycle is transmitted to higher-level external caches directing them to perform an invalidation.

The INVD instruction should only be used in situations where memory coherency is not required.

7.6.4 Serializing Instructions

Serializing instructions force the processor to retire the serializing instruction and all previous instructions before the next instruction is fetched. A serializing instruction is retired when the following operations are complete:

- The instruction has executed.
- All registers modified by the instruction are updated.
- All memory updates performed by the instruction are complete.
- All data held in the write buffers have been written to memory.

Serializing instructions can be used as a barrier between memory accesses to force strong ordering of memory operations. Care should be exercised in using serializing instructions because they modify processor state and may affect program flow. The instructions also force execution serialization, which can significantly degrade performance. When strongly-ordered memory accesses are required, but execution serialization is not, it is recommended that software use the memory-ordering instructions described on page 212.

The following are serializing instructions:

- *Non-Privileged Instructions*
  - CPUID
  - IRET
  - RSM
  - MFENCE
- *Privileged Instructions*
  - MOV CRn
  - MOV DRn
  - LGDT, LIDT, LLDT, LTR
  - WRMSR (see note 1)
  - WBINVD, WBNOINVD, INVD
  - INVLPG

[AMD Public Use]

Note 1: *Writes to the following MSRs are not serializing:* SPEC_CTRL, PRED_CMD, all x2APIC MSRs.

A dispatch serializing instruction is a lighter form of ordering than a serializing instruction. A dispatch serializing instruction forces the processor to retire the serializing instruction and all previous instructions before the next instruction is executed. In some systems, LFENCE may be configured to be dispatch serializing. In systems where CPUID Fn8000_0021_EAX[LFenceAlwaysSerializing](bit 2) = 1, LFENCE is always dispatch serializing.

7.6.5 Cache and Processor Topology

Cache and processor topology information is useful in the optimal management of system and application resources. Exposing processor and cache topology information to the programmer allows software to make more efficient use of hardware multithreading resources delivering optimal performance. Shared resources in a specific cache and processor topology may require special consideration in the optimization of multiprocessing software performance.

The processor topology allows software to determine which cores or logical processors are siblings in a compute unit, node, and processor package. For example, a scheduler can then choose to either compact or scatter threads (or processes) to cores in compute units, nodes, or across the cores in the entire physical package in order to optimize for a power and performance profile.

Topology extensions define processor topology at both the node, compute unit and cache level. Topology extensions include cache properties with sharing and the processor topology identified. The result is a simplified extension to the CPUID instruction that describes the processors cache topology and leverages existing industry cache properties folded into AMD’s topology extension description.

Topology extensions definition supports existing and future processors with varying degrees of cache level sharing. Topology extensions also support the description of a simple compute unit with one core or packages where the number of cores in a node and/or compute unit are not an even power of two.

CPUID Function 8000_001D: Cache Topology Definition. CPUID Function 8000_001D describes the hierarchical relationships of cache levels relative to the cores which share these resources. Function 8000_001D is defined to be called iteratively with the value 8000001Dh in EAX and an additional parameter in ECX. To gather information for all cache levels, software must execute the CPUID instruction with 8000001Dh in EAX and ECX set to increasing values beginning with 0 until a value of 0 is returned from EAX[4:0], which indicates no more cache descriptions.

If software dynamically manages cache configuration, it will need to update any stored cache properties for the processor.

CPUID Function 8000_001E: Processor Topology Definition. CPUID Function 8000_001E describes processor topology with component identifiers. To read the processor topology, definition software calls the CPUID instruction with the value 8000001Eh in EAX. After execution, the APIC ID is represented in EAX. EBX contains the compute unit description in the processor, while ECX contains system unique node identification. Software may read this information once for each core.

[AMD Public Use]

CPUID Function 8000_0026: Extended CPU Topology. CPUID Fn8000_0026 reports extended topology level information, including heterogenous topology, for the cores within the system. The topology level is selected by the value passed to the instruction in ECX. To discover the topology of a system, software should execute the CPUID instruction with an EAX value of 80000026h and increasing values of ECX, starting with a value of zero, until CPUID Fn8000_0026_ECX[LevelType](bits 15:8) returns zero. More information about CPUID Fn8000_0026 can be found in Appendix E of Volume 3.

The following CPUID functions provide processor topology information:

- CPUID Fn8000_0001_ECX
- CPUID Fn8000_0008_ECX
- CPUID Fn8000_001D_EAX, EBX, ECX, EDX
- CPUID Fn8000_001E_EAX, EBX, ECX
- CPUID Fn8000_0026_EAX, EBX, ECX, EDX

For more information using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 73.

7.6.6 L3 Cache Range Reservation

The L3 Cache Range Reservation feature allows a portion of the L3 cache to be reserved for a specific system physical address range. This capability is intended to reduce access latency for a specified range of memory by keeping it in the L3 cache as much as possible, which may be useful in some latency-sensitive applications. However, it does not provide a guaranteed level of performance. Since many factors can affect the performance of a particular process, performance evaluation of specific configurations is critical to determining whether to enable the L3 Range Reservation feature. Further details are provided in the following subsections.

CPUID Feature Flag. Support for the L3 Cache Range Reservation feature is indicated by CPUID Fn8000_0020_EBX[L3RR] (bit 4) = 1.

L3 Range Reservation Registers. The following MSRs are used to program L3 Range Reservation:

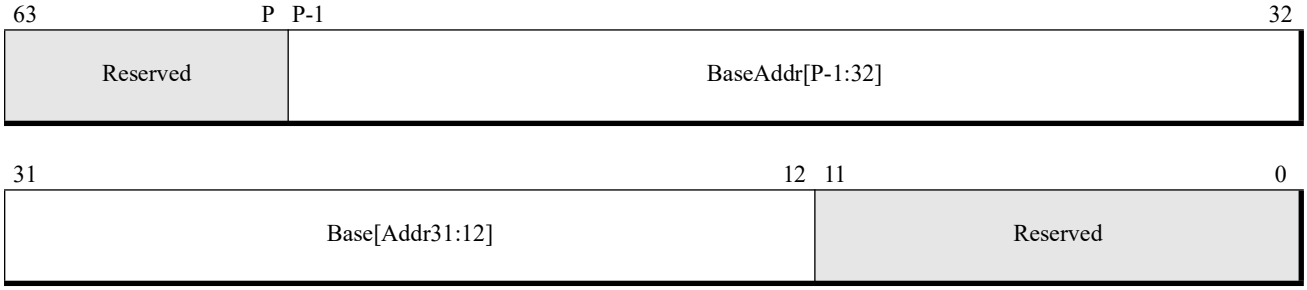
- L3 Range Reserve Base Address Register (C001_1095h)
- L3 Range Reserve Maximum Address Register (C001_1096h)
- L3 Range Reserve Way Mask (C001_109Ah)

L3 Range Reservation MSR layouts are shown below. The MSR bit position numbering uses *P* and *W*, where:

- P is equal to the value of CPUID Fn8000_0008_EAX[PhysAddrSize].
- W is equal to the value of CPUID Fn8000_001D_EBX_x03[CacheNumWays].

[AMD Public Use]

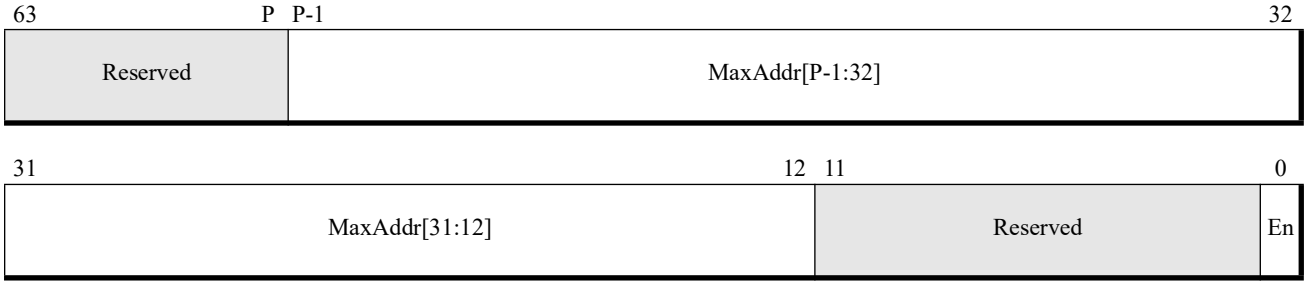
L3 Range Reserve Base Address



Bits	Mnemonic	Description	Access type
63:P	Reserved		MBZ
P-1:12	BaseAddr[P-1:12]	Base Address, 4KB aligned SPA	R/W
11:0	Reserved		MBZ

Figure 7-4. L3 Range Reserve Base Address Register (L3RangeReserveBaseAddr)

L3 Range Reserve Maximum Address

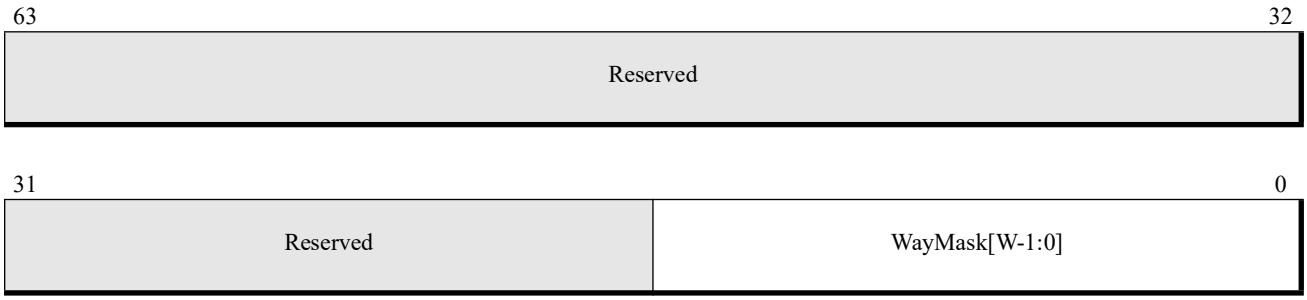


Bits	Mnemonic	Description	Access type
63:P	Reserved		MBZ
P-1:12	MaxAddr[P-1:12]	Maximum Address, 4KB aligned SPA	R/W
11:1	Reserved		MBZ
0	En	Enable	R/W

Figure 7-5. L3 Range Reserve Maximum Address Register (L3RangeReserveMaxAddr)

[AMD Public Use]

L3 Range Reserve Way Mask



Bits	Mnemonic	Description	Access type
63:W	Reserved		MBZ
W-1:0	WayMask[W-1:0]	Way Mask, WayMask[n]=1 allocates L3 way n to range reserved addresses.	R/W

Figure 7-6. L3 Range Reserve Way Mask Register (L3RangeReserveWayMask)

L3 Range Reservation Register Programming. A system physical address *A* is in the reserved range if:

BaseAddr[P-1:12] <= A[P-1:12] < MaxAddr[P-1:12]

The L3 ways enabled for use by cache lines in the reserved range will be referred to as the reserved region of the cache.

To enable the L3 Range Reservation feature, software must follow the following sequence of MSR writes:

1. Write MSR C001_109Ah to program the WayMask[W-1:0] field.
2. Write MSR C001_1095h to program the BaseAddr[P-1:12] field.
3. Write MSR C001_1096h to program the MaxAddr[P-1:12] field and set En to 1.

To disable the L3 Range Reservation feature, set En to zero.

See “Programming Considerations” on page 218 for other requirements and considerations.

Range Reservation used as Range Lock. The L3 Range Reservation feature can be used to lock the reserved range of addresses in the L3 cache in the reserved region. To accomplish this, software must calculate the size of the L3 cache using the following fields of CPUID information:

- CacheNumSets from CPUID Fn8000_001D_ECX_x03.
- CacheLineSize from CPUID Fn8000_001D_EBX_x03.

Using the above parameters, the L3 cache size per-way in bytes is calculated as:

*L3SizePerWay = CacheNumSets * (CacheLineSize + 1).*

[AMD Public Use]

Software can determine the size of the reserved range it wishes to lock in bytes and allocate as many L3-ways as needed by programming the *WayMask* field in MSR C001_109Ah. Software must set CR0.CD on all logical processors sharing the L3 cache and execute the WBINVD instruction to flush the L3 cache before enabling the Range Reservation feature. If the size of the reserved range is less than the number of bytes of L3 cache allocated by the *WayMask* field in MSR C001_109Ah, the reserved range will remain "locked" in cache. The reserved range follows coherency rules around evictions, probes, and cache flushes. If the size of the reserved range does not fit in the allocated L3 ways as programmed in the *WayMask* field in MSR C001_109Ah, the cache lines within the reserved range are replaced using normal L3 cache replacement policy, in those allocated L3 ways. However, the reserved region is only used by cache lines in the reserved range.

Range Reservation Operation. The scope of L3 range reservation is limited to an L3 cache and the cores sharing that L3 cache. Setting up a reserved range reserves a portion of the corresponding cache for use by a range of system physical addresses but does not bring the data or instructions at those addresses into the cache. The cache is filled with the data or instructions from the reserved address region as those addresses are accessed by the associated logical processors, either through explicit references by software, by hardware-based prefetching or by speculative execution of instructions which may or may not eventually retire. After the L3 range reservation is enabled, the cache lines with addresses in the reserved range are cached in the reserved region of the L3 cache. The cache lines with addresses outside the reserved range are cached in the unreserved region of the L3 cache.

Whether L3 range reservation is enabled or disabled, a cache line in the reserved range will still behave coherently with respect to all coherent operations. Placing a cache line in a reserved location does not exempt it from the rules of coherency. In case where a cache line outside the reserved range is present in the reserved region at the time the L3 range reservation feature is enabled or where a cache line in the reserved range is present in unreserved regions of the cache, normal coherent accesses by any processor in the system will still read the correct, current data from these locations.

Programming Considerations. The Range Reservation configuration registers should not be modified while cacheable traffic is in flight. The recommended way to meet this requirement is to set the CR0.CD bit on all logical processors which are connected to the cache being configured. For the L3 cache, that includes all the logical processors sharing an L3 cache and for the L2 cache that includes the two logical processors which share a physical core and L2 cache.

If a given range is reserved in multiple L3 caches in a system, the cache lines can be cached in each of them as long as none of the logical processors accessing the data modify it. Modifying the cache lines will remove them from all other caches. However, once the modification is complete the cache lines can once again be re-cached in multiple caches if accessed by those processors. The cache lines thus accessed are again placed in the reserved region of the L3 cache upon access. It is not necessary to disable and re-enable the address range in order to re-establish the reserved region; the affected addresses will be brought into the reserved cache regions again as they are accessed, although the initial accesses to those ranges will incur the latency penalty associated with retrieving the data or instructions from memory or another cache.

Setting up a reserved range does not affect the cacheability rules of the addresses in that range. For example, if the page tables specify that the memory type for a page is UC, including that page within a

[AMD Public Use]

range reserved region will not cause the addresses on that page to become cacheable. Instead, a portion of the cache will become unusable by any address since the addresses for which that portion of cache is reserved cannot be installed in the cache and all other addresses are prevented from being installed there.

When Secure Memory Encryption (SME) or Secure Encrypted Virtualization (SEV) is enabled, the L3 range reservation feature cannot be used on system physical addresses that are encrypted. See “Secure Memory Encryption” on page 240 and “Secure Encrypted Virtualization” on page 613 for details.

Reserving a way of the L3 cache for range reservation exempts those ways from being used by the L3 QoS Allocation feature. The ways that overlap between any of the *L3 QoS Allocation Mask* MSRs (MSR 0x0000_0C90h through 0x0000_0C9Fh) will be unavailable for use by the L3 QoS Allocation feature and only available for use by the L3 Range Reservation feature.

7.7 Memory-Type Range Registers

The AMD64 architecture supports three mechanisms for software access-control and cacheability-control over memory regions. These mechanisms can be used in place of similar capabilities provided by external chipsets used with early x86 processors.

This section describes a control mechanism that uses a set of programmable model-specific registers (MSRs) called the *memory-type-range registers* (MTRRs). The MTRR mechanism provides system software with the ability to manage hardware-device memory mapping. System software can characterize physical-memory regions by type (e.g., ROM, flash, memory-mapped I/O) and assign hardware devices to the appropriate physical-memory type.

Another control mechanism is implemented as an extension to the page-translation capability and is called the *page attribute table* (PAT). It is described in “Page-Attribute Table Mechanism” on page 229. Like the MTRRs, PAT provides system software with the ability to manage hardware-device memory mapping. With PAT, however, system software can characterize physical pages and assign virtually-mapped devices to those physical pages using the page-translation mechanism. PAT may be used in conjunction with the MTRR mechanism to maximize flexibility in memory control.

Finally, control mechanisms are provided for managing memory-mapped I/O. These mechanisms employ extensions to the MTRRs and a separate feature called the *top-of-memory registers*. The MTRR extensions include additional MTRR type-field encodings for fixed-range MTRRs and variable-range I/O range registers (IORRs). These mechanisms are described in “Memory-Mapped I/O” on page 234.

7.7.1 MTRR Type Fields

The MTRR mechanism provides a means for associating a physical-address range with a memory type (see “Memory Types” on page 198). The MTRRs contain a type field used to specify the memory type in effect for a given physical-address range.

[AMD Public Use]

There are two variants of the memory type-field encodings: standard and extended. Both the standard and extended encodings use type-field bits 2:0 to specify the memory type. For the standard encodings, bits 7:3 are reserved and must be zero. For the extended encodings, bits 7:5 are reserved, but bits 4:3 are defined as the RdMem and WrMem bits. “Extended Fixed-Range MTRR Type-Field Encodings” on page 234 describes the function of these extended bits and how software enables them. Only the fixed-range MTRRs support the extended type-field encodings. Variable-range MTRRs use the standard encodings.

Table 7-6 on page 220 shows the memory types supported by the MTRR mechanism and their encoding in the MTRR type fields referenced throughout this section. Unless the extended type-field encodings are explicitly enabled, the processor uses the type values shown in Table 7-6.

Table 7-6. MTRR Type Field Encodings

Type Value	Type Name	Type Description
00h	UC—Uncacheable	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed
01h	WC—Write-Combining	All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed
04h	WT—Writethrough	Reads allocate cache lines on a cache miss. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.
05h	WP—Write-Protect	Reads allocate cache lines on a cache miss. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.
06h	WB—Writeback	Reads allocate cache lines on a cache miss, and can allocate to either the shared, exclusive, or modified state. Writes allocate to the modified state on a cache miss.

If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). *Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0.* Cacheable memory types must be established using the MTRRs to enable memory accesses to be cached.

7.7.2 MTRRs

Both fixed-size and variable-size address ranges are supported by the MTRR mechanism. The fixed-size ranges are restricted to the lower 1 Mbyte of physical-address space, while the variable-size ranges can be located anywhere in the physical-address space.

Figure 7-7 on page 221 shows an example mapping of physical memory using the fixed-size and variable-size MTRRs. The areas shaded gray are not mapped by the MTRRs. Unmapped areas are set to the software-selected default memory type.

[AMD Public Use]

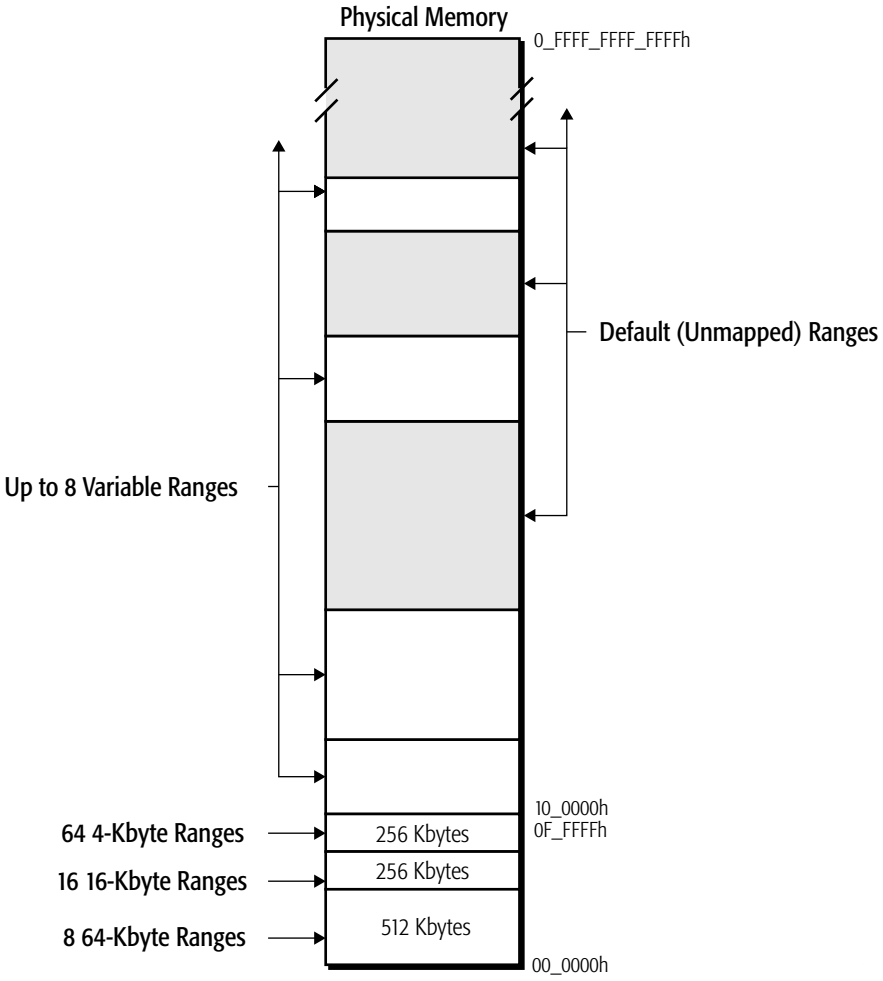


Figure 7-7. MTRR Mapping of Physical Memory

MTRRs are 64-bit model-specific registers (MSRs). They are read using the RDMSR instruction and written using the WRMSR instruction. See “Memory-Typing MSRs” on page 768 for a listing of the MTRR MSR numbers. The following sections describe the types of MTRRs and their function.

Fixed-Range MTRRs. The fixed-range MTRRs are used to characterize the first 1 Mbyte of physical memory. Each fixed-range MTRR contains eight type fields for characterizing a total of eight memory ranges. Fixed-range MTRRs support extended type-field encodings as described in “Extended Fixed-Range MTRR Type-Field Encodings” on page 234. The extended type field allows a fixed-range MTRR to be used as a fixed-range IORR. Figure 7-8 on page 222 shows the format of a fixed-range MTRR.

[AMD Public Use]

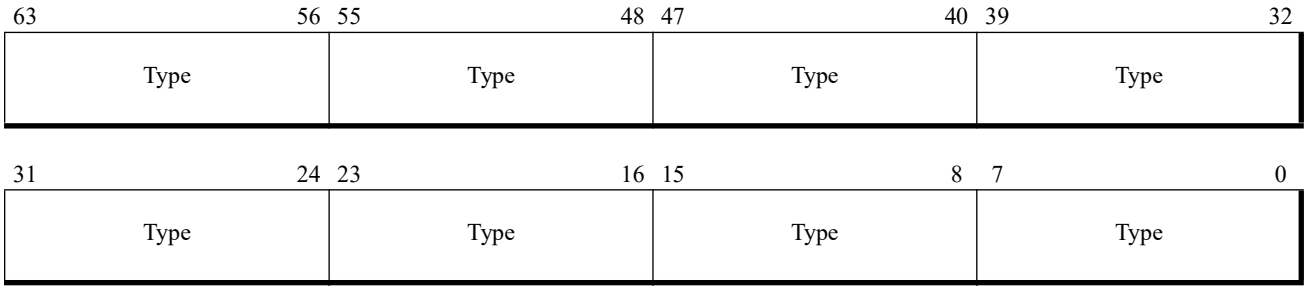


Figure 7-8. Fixed-Range MTRR

For the purposes of memory characterization, the first 1 Mbyte of physical memory is segmented into a total of 88 non-overlapping memory ranges, as follows:

- The 512 Kbytes of memory spanning addresses 00_0000h to 07_FFFFh are segmented into eight 64-Kbyte ranges. A single MTRR is used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 08_0000h to 0B_FFFFh are segmented into 16 16-Kbyte ranges. Two MTRRs are used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 0C_0000h to 0F_FFFFh are segmented into 64 4-Kbyte ranges. Eight MTRRs are used to characterize this address space.

Table 7-7 shows the address ranges corresponding to the type fields within each fixed-range MTRR. The gray-shaded heading boxes represent the bit ranges for each type field in a fixed-range MTRR. See Table 7-6 on page 220 for the type-field encodings.

Table 7-7. Fixed-Range MTRR Address Ranges

Physical Address Range (in hexadecimal)								Register Name
63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0	
70000–7FFFF	60000–6FFFF	50000–5FFFF	40000–4FFFF	30000–3FFFF	20000–2FFFF	10000–1FFFF	00000–0FFFF	**MTRRfix64K_00000**
9C000–9FFFF	98000–9BFFF	94000–97FFF	90000–93FFF	8C000–8FFFF	88000–8BFFF	84000–87FFF	80000–83FFF	**MTRRfix16K_80000**
BC000–BFFFF	B8000–BBFFF	B4000–B7FFF	B0000–B3FFF	AC000–AFFFF	A8000–ABFFF	A4000–A7FFF	A0000–A3FFF	**MTRRfix16K_A0000**
C7000–C7FFF	C6000–C6FFF	C5000–C5FFF	C4000–C4FFF	C3000–C3FFF	C2000–C2FFF	C1000–C1FFF	C0000–C0FFF	**MTRRfix4K_C0000**
CF000–CFFFF	CE000–CEFFF	CD000–CDFFF	CC000–CCFFF	CB000–CBFFF	CA000–CAFFF	C9000–C9FFF	C8000–C8FFF	**MTRRfix4K_C8000**
D7000–D7FFF	D6000–D6FFF	D5000–D5FFF	D4000–D4FFF	D3000–D3FFF	D2000–D2FFF	D1000–D1FFF	D0000–D0FFF	**MTRRfix4K_D0000**
DF000–DFFFF	DE000–DEFFF	DD000–DDFFF	DC000–DCFFF	DB000–DBFFF	DA000–DAFFF	D9000–D9FFF	D8000–D8FFF	**MTRRfix4K_D8000**

[AMD Public Use]

Table 7-7. Fixed-Range MTRR Address Ranges (continued)

Physical Address Range (in hexadecimal)								Register Name
63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0	
E7000–E7FFF	E6000–E6FFF	E5000–E5FFF	E4000–E4FFF	E3000–E3FFF	E2000–E2FFF	E1000–E1FFF	E0000–E0FFF	**MTRRfix4K_E0000**
EF000–EFFFF	EE000–EEFFF	ED000–EDFFF	EC000–ECFFF	EB000–EBFFF	EA000–EAFFF	E9000–E9FFF	E8000–E8FFF	**MTRRfix4K_E8000**
F7000–F7FFF	F6000–F6FFF	F5000–F5FFF	F4000–F4FFF	F3000–F3FFF	F2000–F2FFF	F1000–F1FFF	F0000–F0FFF	**MTRRfix4K_F0000**
FF000–FFFFF	FE000–FEFFF	FD000–FDFFF	FC000–FCFFF	FB000–FBFFF	FA000–FAFFF	F9000–F9FFF	F8000–F8FFF	**MTRRfix4K_F8000**

Variable-Range MTRRs. The variable-range MTRRs can be used to characterize any address range within the physical-memory space, including all of physical memory. Up to eight address ranges of varying sizes can be characterized using the MTRR. Two variable-range MTRRs are used to characterize each address range: MTRRphysBase*n* and MTRRphysMask*n* (*n* is the address-range number from 0 to 7). For example, address-range 3 is characterized using the MTRRphysBase3 and MTRRphysMask3 register pair.

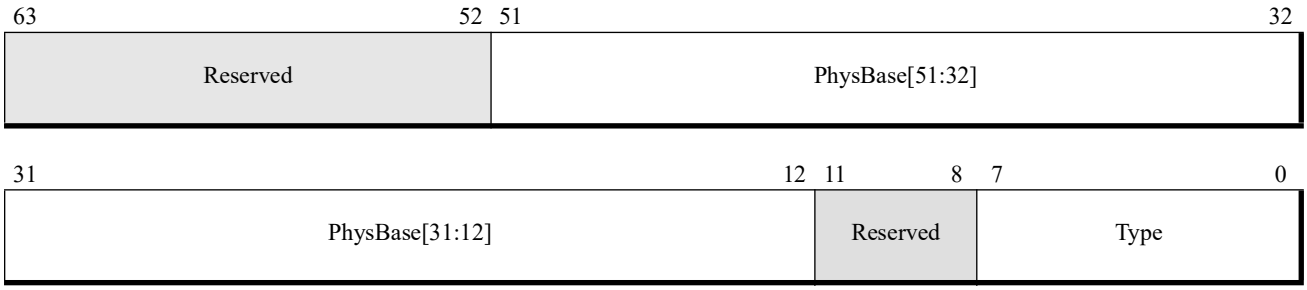
Figure 7-9 shows the format of the MTRRphysBase*n* register and Figure 7-10 on page 224 shows the format of the MTRRphysMask*n* register. The fields within the register pair are read/write.

MTRRphysBase*n* Registers. The fields in these variable-range MTRRs, shown in Figure 7-9, are:

- *Type*—Bits 7:0. The memory type used to characterize the memory range. See Table 7-6 on page 220 for the type-field encodings. Variable-range MTRRs do not support the extended type-field encodings.
- *Range Physical Base-Address (PhysBase)*—Bits 51:12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11:0 are assumed to be 0.

Note that a given processor may implement less than the architecturally-defined physical address size of 52 bits.

[AMD Public Use]

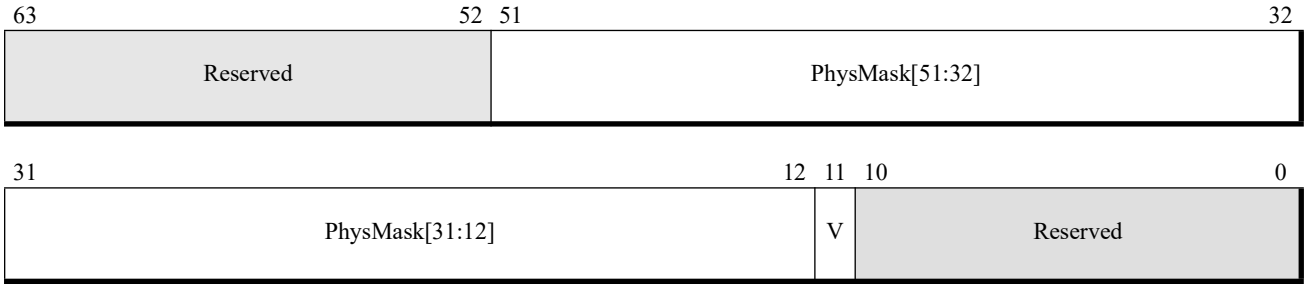


Bits	Mnemonic	Description	Access type
63:52	Reserved		MBZ
51:12	PhysBase	Range Physical Base Address	R/W
11:8	Reserved		MBZ
7:0	Type	Default Memory Type	R/W

Figure 7-9. MTRRphysBase*n* Register

MTRRphysMask*n* Registers. The fields in these variable-range MTRRs, shown in Figure 7-10, are:

- *Valid (V)*—Bit 11. Indicates that the MTRR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0 the register pair is not used.
- *Range Physical Mask (PhysMask)*—Bits 51:12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11:0 of PhysMask are assumed to be 0.



Bits	Mnemonic	Description	Access type
63:52	Reserved		MBZ
51:12	PhysMask	Range Physical Mask	R/W
11	V	MTRR Pair Enable (Valid)	R/W
10:0	Reserved		MBZ

Figure 7-10. MTRRphysMask*n* Register

PhysMask and PhysBase are used together to determine whether a target physical-address falls within the specified address range. PhysMask is logically ANDed with PhysBase and separately ANDed with

[AMD Public Use]

the upper 40 bits of the target physical-address. If the results of the two operations are identical, the target physical-address falls within the specified memory range. The pseudo-code for the operation is:

```
MaskBase = PhysMask AND PhysBase
MaskTarget = PhysMask AND Target_Address[51:12]
IF MaskBase == MaskTarget
   target address is in range
ELSE
   target address is not in range
```

Variable Range Size and Alignment. The size and alignment of variable memory-ranges (MTRRs) and I/O ranges (IORRs) are restricted as follows:

- The boundary on which a variable range is aligned must be equal to the range size. For example, a memory range of 16 Mbytes must be aligned on a 16-Mbyte boundary.
- The range size must be a power of 2 (2^n, $52 > n > 11$), with a minimum allowable size of 4 Kbytes. For example, 4 Mbytes and 8 Mbytes are allowable memory range sizes, but 6 Mbytes is not allowable.

PhysMask and PhysBase Values. Software can calculate the PhysMask value using the following procedure:

1. Subtract the memory-range physical base-address from the upper physical-address of the memory range.
2. Subtract the value calculated in Step 1 from the physical memory size.
3. Truncate the lower 12 bits of the result in Step 2 to create the PhysMask value to be loaded into the MTRRphysMask*n* register. Truncation is performed by right-shifting the value 12 bits.

For example, assume a 32-Mbyte memory range is specified within the 52-bit physical address space, starting at address 200_0000h. The upper address of the range is 3FF_FFFFh. Following the process outlined above yields:

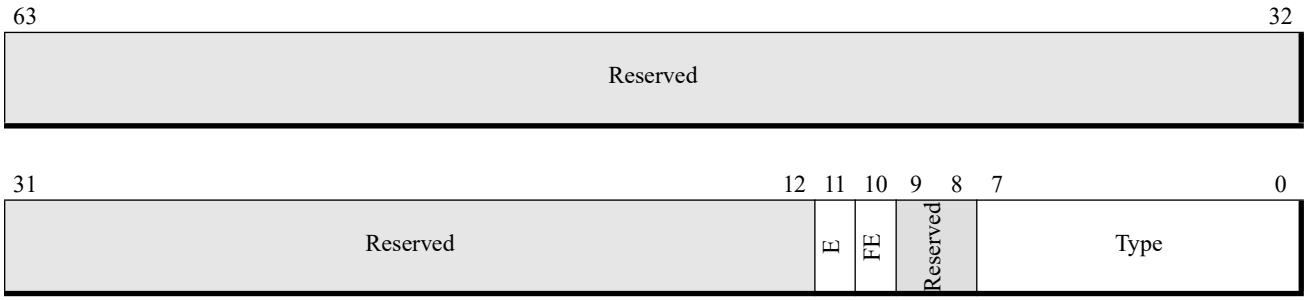
1. 3FF_FFFFh–200_0000h = 1FF_FFFFh
2. F_FFFF_FFFF_FFFF–1FF_FFFFh = F_FFFF_FE00_0000h
3. Right shift (F_FFFF_FE00_0000h) by 12 = FF_FFFF_E000h

In this example, the 40-bit value loaded into the PhysMask field is FF_FFFF_E000h.

Software must also truncate the lower 12 bits of the physical base-address before loading it into the PhysBase field. In the example above, the 40-bit PhysBase field is 00_0000_2000h.

Default-Range MTRRs. Physical addresses that are not within ranges established by fixed-range and variable-range MTRRs are set to a default memory-type using the MTRRdefType register. The format of this register is shown in Figure 7-11.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:12	Reserved		MBZ
11	E	MTRR Enable	R/W
10	FE	Fixed Range Enable	R/W
9:8	Reserved		MBZ
7:0	Type	Default Memory Type	R/W

Figure 7-11. MTRRdefType Register Format

The fields within the MTRRdefType register are read/write. These fields are:

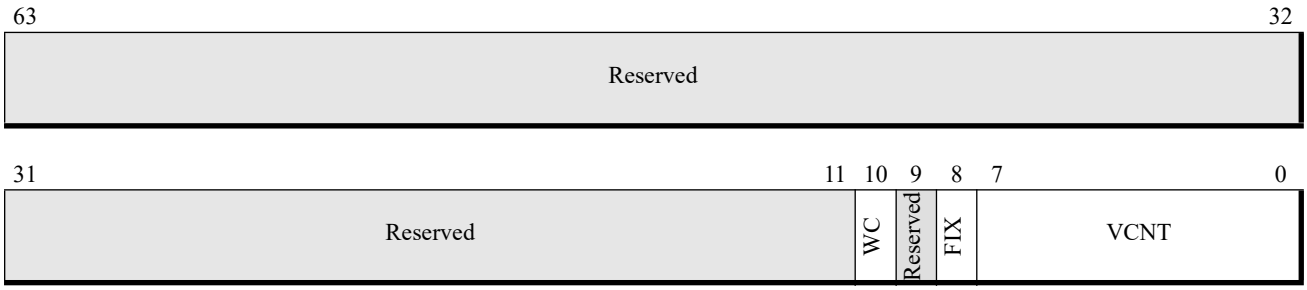
- *Type*—Bits 7:0. The default memory-type used to characterize physical-memory space. See Table 7-6 on page 220 for the type-field encodings. The extended type-field encodings are not supported by this register.
- *Fixed-Range Enable (FE)*—Bit 10. All fixed-range MTRRs are enabled when FE is set to 1. Clearing FE to 0 disables all fixed-range MTRRs. Setting and clearing FE has no effect on the variable-range MTRRs. The FE bit has no effect unless the E bit is set to 1 (see below).
- *MTRR Enable (E)*—Bit 11. This is the MTRR memory typing enable bit. The memory typing capabilities of all fixed-range and variable-range MTRRs are enabled when E is set to 1. Clearing E to 0 disables the memory typing capabilities of all fixed-range and variable-range MTRRs and sets the default memory-type to uncacheable (UC) regardless of the value of the Type field. This bit does not affect the operation of the RdMem and WrMem fields.

7.7.3 Using MTRRs

Identifying MTRR Features. Software determines whether a processor supports the MTRR mechanism by executing the CPUID instruction with either function 0000_0001h or function 8000_0001h. If MTRRs are supported, bit 12 in the EDX register is set to 1 by CPUID. See “Processor Feature Identification” on page 73 for more information on the CPUID instruction.

The MTRR capability register (MTRRcap) is a read-only register containing information describing the level of MTRR support provided by the processor. Figure 7-12 shows the format of this register. If MTRRs are supported, software can read MTRRcap using the RDMSR instruction. Attempting to write to the MTRRcap register causes a general-protection exception (#GP).

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:11	Reserved		R
10	WC	Write Combining	R
9	Reserved		R
8	FIX	Fixed-Range Registers	R
7:0	VCNT	Variable-Range Register Count	R

Figure 7-12. MTRR Capability Register Format

The MTRRcap register field are:

- *Variable-Range Register Count (VCNT)*—Bits 7:0. The VCNT field contains the number of variable-range register pairs supported by the processor. For example, a processor supporting eight register pairs returns a 08h in this field.
- *Fixed-Range Registers (FIX)*—Bit 8. The FIX bit indicates whether or not the fixed-range registers are supported. If the processor returns a 1 in this bit, *all* fixed-range registers are supported. If the processor returns a 0 in this bit, *no* fixed-range registers are supported.
- *Write-Combining (WC)*—Bit 10. The WC bit indicates whether or not the write-combining memory type is supported. If the processor returns a 1 in this bit, WC memory is supported, otherwise it is not supported.

7.7.4 MTRRs and Page Cache Controls

When paging and the MTRRs are both enabled, the address ranges defined by the MTRR registers can span multiple pages, each of which can characterize memory with different types (using the PCD and PWT page bits). When caching is enabled (CR0.CD=0 and CR0.NW=0), the *effective memory type* is determined as follows:

1. If the page is defined as cacheable and writeback (PCD=0 and PWT=0), then the MTRR defines the effective memory-type.
2. If the page is defined as not cacheable (PCD=1), then UC is the effective memory-type.
3. If the page is defined as cacheable and writethrough (PCD=0 and PWT=1), then the MTRR defines the effective memory-type *unless* the MTRR specifies WB memory, in which case WT is the effective memory-type.

[AMD Public Use]

Table 7-8 lists the MTRR and page-level cache-control combinations and their combined effect on the final memory-type, if the PAT register holds the default settings.

Table 7-8. Combined MTRR and Page-Level Memory Type with Unmodified PAT MSR

MTRR Memory Type	Page PCD Bit	Page PWT Bit	Effective Memory-Type
UC	—	—	UC
WC	0	—	WC
	1	0	WC[1]
	1	1	UC
WP	0	—	WP
	1	—	UC
WT	0	—	WT
	1	—	UC
WB	0	0	WB
	0	1	WT
	1	—	UC

Note:
1. The effective memory-type resulting from the combination of PCD=1, PWT=0, and an MTRR WC memory type is implementation dependent.

Large Page Sizes. When paging is enabled, software can use large page sizes (2 Mbytes and 4 Mbytes) in addition to the more typical 4-Kbyte page size. When large page sizes are used, it is possible for multiple MTRRs to span the memory range within a single large page. Each MTRR can characterize the regions within the page with different memory types. If this occurs, the effective memory-type used by the processor within the large page is undefined.

Software can avoid the undefined behavior in one of the following ways:

- Avoid using multiple MTRRs to characterize a single large page.
- Use multiple 4-Kbyte pages rather than a single large page.
- If multiple MTRRs must be used within a single large page, software can set the MTRR type fields to the same value.
- If the multiple MTRRs must have different type-field values, software can set the large page PCD and PWT bits to the most restrictive memory type defined by the multiple MTRRs.

Overlapping MTRR Registers. If the address ranges of two or more MTRRs overlap, the following rules are applied to determine the memory type used to characterize the overlapping address range:

1. Fixed-range MTRRs, which characterize only the first 1 Mbyte of physical memory, have precedence over variable-range MTRRs.

[AMD Public Use]

2. If two or more variable-range MTRRs overlap, the following rules apply:
   a. If the memory types are identical, then that memory type is used.
   b. If at least one of the memory types is UC, the UC memory type is used.
   c. If at least one of the memory types is WT, and the only other memory type is WB, then the WT memory type is used.
   d. If the combination of memory types is not listed Steps A through C immediately above, then the memory type used is undefined.

7.7.5 MTRRs in Multi-Processing Environments

In multi-processing environments, the MTRRs located in all processors must characterize memory in the same way. Generally, this means that identical values are written to the MTRRs used by the processors. This also means that values CR0.CD and the PAT must be consistent across processors. Failure to do so may result in coherency violations or loss of atomicity. Processor implementations *do not* check the MTRR settings in other processors to ensure consistency. It is the responsibility of system software to initialize and maintain MTRR consistency across all processors.

7.8 Page-Attribute Table Mechanism

The page-attribute table (PAT) mechanism extends the page-table entry format and enhances the capabilities provided by the PCD and PWT page-level cache controls. PAT (and PCD, PWT) allow memory-type characterization based on the virtual (linear) address. The PAT mechanism provides the same memory-typing capabilities as the MTRRs but with the added flexibility of the paging mechanism. Software can use both the PAT and MTRR mechanisms to maximize flexibility in memory-type control.

7.8.1 PAT Register

Like the MTRRs, the PAT register is a 64-bit model-specific register (MSR). The format of the PAT registers is shown in Figure 7-13. See “Memory-Typing MSRs” on page 768 for more information on the PAT MSR number and reset value.

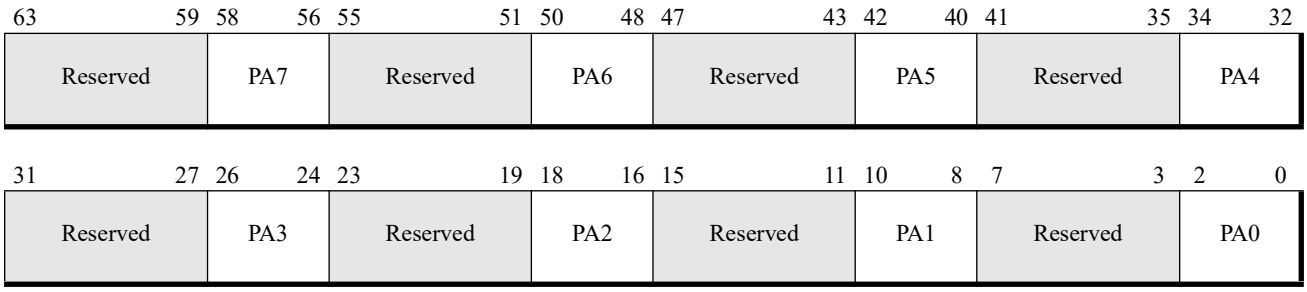


Figure 7-13. PAT Register

[AMD Public Use]

The PAT register contains eight page-attribute (PA) fields, numbered from PA0 to PA7. The PA fields hold the encoding of a memory type, as found in Table 7-9 on page 230. The PAT type-encodings match the MTRR type-encodings, with the exception that PAT adds the 07h encoding. The 07h encoding corresponds to a *UC−* type. The UC− type (07h) is identical to the UC type (00h) except it can be overridden by an MTRR type of WC.

Software can write any supported memory-type encoding into any of the eight PA fields. An attempt to write anything but zeros into the reserved fields causes a general-protection exception (#GP). An attempt to write an unsupported type encoding into a PA field also causes a #GP exception.

The PAT register fields are initiated at processor reset to the default values shown in Table 7-10 on page 231.

Table 7-9. PAT Type Encodings

Type Value	Type Name	Type Description
00h	UC—Uncacheable	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed.
01h	WC—Write-Combining	All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed.
04h	WT—Writethrough	Reads allocate cache lines on a cache miss, but only to the shared state. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.
05h	WP—Write-Protect	Reads allocate cache lines on a cache miss, but only to the shared state. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.
06h	WB—Writeback	Reads allocate cache lines on a cache miss, and can allocate to either the shared or exclusive state. Writes allocate to the modified state on a cache miss.
07h	UC– (UC minus)	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed. Can be overridden by an MTRR with the WC type.

7.8.2 PAT Indexing

PA fields in the PAT register are selected using three bits from the page-table entries. These bits are:

- *PAT (page attribute table)*—The PAT bit is bit 7 in 4-Kbyte PTEs; it is bit 12 in 2-Mbyte and 4-Mbyte PDEs. Page-table entries that don't have a PAT bit (PML4 entries, for example) assume PAT = 0.
- *PCD (page cache disable)*—The PCD bit is bit 4 in all page-table entries. The PCD from the PTE or PDE is selected depending on the paging mode.
- *PWT (page writethrough)*—The PWT bit is bit 3 in all page-table entries. The PWT from the PTE or PDE is selected depending on the paging mode.

[AMD Public Use]

Table 7-10 on page 231 shows the various combinations of the PAT, PCD, and PWT bits used to select a PA field within the PAT register. Table 7-10 also shows the default memory-type values established in the PAT register by the processor after a reset. The default values correspond to the memory types established by the PCD and PWT bits alone in processor implementations that do not support the PAT mechanism. In such implementations, the PAT field in page-table entries is reserved and cleared to 0. See "Page-Translation-Table Entry Fields" on page 155 for more information on the page-table entries.

Table 7-10. PAT-Register PA-Field Indexing

Page Table Entry Bits			PAT Register Field	Default Memory Type
PAT	PCD	PWT		
0	0	0	PA0	WB
0	0	1	PA1	WT
0	1	0	PA2	UC–[1]
0	1	1	PA3	UC
1	0	0	PA4	WB
1	0	1	PA5	WT
1	1	0	PA6	UC–[1]
1	1	1	PA7	UC

Note:
1. Can be overridden by WC memory type set by an MTRR.

7.8.3 Identifying PAT Support

Software determines whether a processor supports the PAT mechanism by executing the CPUID instruction with either function 0000_0001h or function 8000_0001h. If PAT is supported, bit 16 in the EDX register is set to 1 by CPUID. See Section 3.3, "Processor Feature Identification," on page 73 for more information on the CPUID instruction.

If PAT is supported by a processor implementation, it is *always enabled*. The PAT mechanism cannot be disabled by software. Software can effectively avoid using PAT by:

- Not setting PAT bits in page-table entries to 1.
- Not modifying the reset values of the PA fields in the PAT register.

In this case, memory is characterized using the same types that are used by implementations that do not support PAT.

7.8.4 PAT Accesses

In implementations that support the PAT mechanism, all memory accesses that are translated through the paging mechanism use the PAT index bits to specify a PA field in the PAT register. The memory type stored in the specified PA field is applied to the memory access. The process is summarized as:

1. A virtual address is calculated as a result of a memory access.

[AMD Public Use]

2. The virtual address is translated to a physical address using the page-translation mechanism.
3. The PAT, PCD and PWT bits are read from the corresponding page-table entry during the virtual-address to physical-address translation.
4. The PAT, PCD and PWT bits are used to select a PA field from the PAT register.
5. The memory type is read from the appropriate PA field.
6. The memory type is applied to the physical-memory access using the translated physical address.

Page-Translation Table Access. The PAT bit exists only in the PTE (4K paging) or PDEs (2/4 Mbyte paging). In the remaining upper levels (PML5, PML4, PDP, and 4KB PDEs), only the PWT and PCD bits are used to index into the first 4 entries in the PAT register. The resulting memory type is used for the next lower paging level.

7.8.5 Combined Effect of MTRRs and PAT

The memory types established by the PAT mechanism can be combined with MTRR-established memory types to form an effective memory-type. The combined effect of MTRR and PAT memory types are shown in Figure 7-11. In the AMD64 architecture, reserved and undefined combinations of MTRR and PAT memory types result in undefined behavior. If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC).

Table 7-11. Combined Effect of MTRR and PAT Memory Types

PAT Memory Type	MTRR Memory Type	Effective Memory Type
UC	UC	UC
UC	WC, WP, WT, WB	CD
UC–	UC	UC
	WC	WC
	WP, WT, WB	CD
WC	—	WC
WP	UC	UC
	WC	CD
	WP	WP
	WT	CD
	WB	WP
WT	UC	UC
	WC, WP	CD
	WT, WB	WT

[AMD Public Use]

Table 7-11. Combined Effect of MTRR and PAT Memory Types (continued)

PAT Memory Type	MTRR Memory Type	Effective Memory Type
WB	UC	UC
	WC	WC
	WP	WP
	WT	WT
	WB	WB

7.8.6 PATs in Multi-Processing Environments

In multi-processing environments, values of CR0.CD and the PAT must be consistent across all processors and the MTRRs in all processors must characterize memory in the same way. In other words, matching address ranges and cachability types are written to the MTRRs for each processor.

Failure to do so may result in coherency violations or loss of atomicity. Processor implementations *do not* check the MTRR, CR0.CD and PAT values in other processors to ensure consistency. It is the responsibility of system software to initialize and maintain consistency across all processors.

7.8.7 Changing Memory Type

A physical page should not have differing cacheability types assigned to it through different virtual mappings; they should be either all of a cacheable type (WB, WT, WP) or all of a non-cacheable type (UC, WC). Otherwise, this may result in a loss of cache coherency, leading to stale data and unpredictable behavior. For this reason, certain precautions must be taken when changing the memory type of a page. In particular, when changing from a cachable memory type to an uncachable type the caches must be flushed, because speculative execution by the processor may have resulted in memory being cached even though it was not programmatically referenced. The following table summarizes the serialization requirements for safely changing memory types.

[AMD Public Use]

Table 7-12. Serialization Requirements for Changing Memory Types

		New Type				
		WB	**WT**	**WP**	**UC**	**WC**
Old Type	WB	–	a	a	b	b
	WT	a	–	a	b	b
	WP	a	a	–	b	b
	UC	a	a	a	–	a
	WC	a	a	a	a	–

Note:

a. Remove the previous mapping (make it not present in the page tables); Flush the TLBs including the TLBs of other processors that may have used the mapping, even speculatively; Create a new mapping in the page tables using the new type.

b. In addition to the steps described in note a, software should flush the page from the caches of any processor that may have used the previous mapping. This must be done after the TLB flushing in note a has been completed.

7.9 Memory-Mapped I/O

Processor implementations can independently direct reads and writes to either system memory or memory-mapped I/O. The method used for directing those memory accesses is implementation dependent. In some implementations, separate system-memory and memory-mapped I/O buses can be provided at the processor interface. In other implementations, system memory and memory-mapped I/O share common data and address buses, and system logic uses sideband signals from the processor to route accesses appropriately. Refer to AMD data sheets and application notes for more information about particular hardware implementations of the AMD64 architecture.

The I/O range registers (IORRs), and the top-of-memory registers allow system software to specify where memory accesses are directed for a given address range. The MTRR extensions are described in the following section. “IORRs” on page 236 describes the IORRs and “Top of Memory” on page 238 describes the top-of-memory registers. *In implementations that support these features, the default action taken when the features are disabled is to direct memory accesses to memory-mapped I/O.*

7.9.1 Extended Fixed-Range MTRR Type-Field Encodings

The fixed-range MTRRs support extensions to the type-field encodings that allow system software to direct memory accesses to system memory or memory-mapped I/O. The extended MTRR type-field encodings use previously reserved bits 4:3 to specify whether reads and writes to a physical-address range are to system memory or to memory-mapped I/O. The format for this encoding is shown in Figure 7-14 on page 235. The new bits are:

- *WrMem*—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.
- *RdMem*—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.

[AMD Public Use]

The type subfield (bits 2:0) allows the encodings specified in Table 7-6 on page 220 to be used for memory characterization.

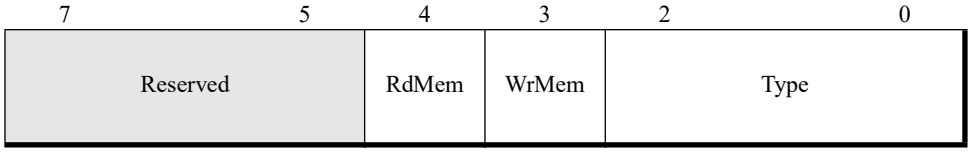


Figure 7-14. Extended MTRR Type-Field Format (Fixed-Range MTRRs)

These extensions are enabled using the following bits in the SYSCFG MSR:

- *MtrrFixDramEn*—Bit 18. When set to 1, RdMem and WrMem attributes are enabled. When cleared to 0, these attributes are disabled. *When disabled, accesses are directed to memory-mapped I/O space.*
- *MtrrFixDramModEn*—Bit 19. When set to 1, software can read and write the RdMem and WrMem bits. When cleared to 0, writes do not modify the RdMem and WrMem bits, and reads return 0.

To use the MTRR extensions, system software must first set MtrrFixDramModEn=1 to allow modification to the RdMem and WrMem bits. After the attribute bits are properly initialized in the fixed-range registers, the extensions can be enabled by setting MtrrFixDramEn=1.

RdMem and WrMem allow the processor to independently direct reads and writes to either system memory or memory-mapped I/O. The RdMem and WrMem controls are particularly useful when shadowing ROM devices located in memory-mapped I/O space. It is often useful to shadow such devices in RAM system memory to improve access performance, but writes into the RAM location can corrupt the shadowed ROM information. The MTRR extensions solve this problem. System software can create the shadow location by setting WrMem = 1 and RdMem = 0 for the specified memory range and then copy the ROM location into itself. Reads are directed to the memory-mapped ROM, but writes go to the same physical addresses in system memory. After the copy is complete, system software can change the bit values to WrMem = 0 and RdMem = 1. Now reads are directed to the faster copy located in system memory, and writes are directed to memory-mapped ROM. The ROM responds as it would normally to a write, which is to ignore it.

Not all combinations of RdMem and WrMem are supported for each memory type encoded by bits 2:0. Table 7-13 on page 236 shows the allowable combinations. The behavior of reserved encoding combinations (shown as gray-shaded cells) is undefined and results in unpredictable behavior.

[AMD Public Use]

Table 7-13. Extended Fixed-Range MTRR Type Encodings

RdMem	WrMem	Type	Implication or Potential Use
0	0	0 (UC)	UC I/O
		1 (WC)	WC I/O
		4 (WT)	WT I/O
		5 (WP)	WP I/O
		6 (WB)	Reserved
0	1	0 (UC)	Used while creating a shadowed ROM
		1 (WC)	
		4 (WT)	Reserved
		5 (WP)	
		6 (WB)	
1	0	0 (UC)	Used to access a shadowed ROM
		1 (WC)	Reserved
		4 (WT)	
		5 (WP)	WP Memory (Can be used to access shadowed ROM)
		6 (WB)	Reserved
1	1	0 (UC)	UC Memory
		1 (WC)	WC Memory
		4 (WT)	WT Memory
		5 (WP)	Reserved
		6 (WB)	WB Memory

7.9.2 IORRs

The IORRs operate similarly to the variable-range MTRRs. The IORRs specify whether reads and writes in any physical-address range map to system memory or memory-mapped I/O. Up to two address ranges of varying sizes can be controlled using the IORRs. A pair of IORRs are used to control each address range: IORRBase*n* and IORRMask*n* (*n* is the address-range number from 0 to 1).

Figure 7-15 on page 237 shows the format of the IORRBase*n* registers and Figure 7-16 on page 238 shows the format of the IORRMask*n* registers. The fields within the register pair are read/write.

The intersection of the IORR range with the equivalent effective MTRR range follows the same type encoding table (Table 7-13) as the fixed-range MTRR, where the RdMem/WrMem and memory type are directly tied together.

IORRBase*n* Registers. The fields in these IORRs are:

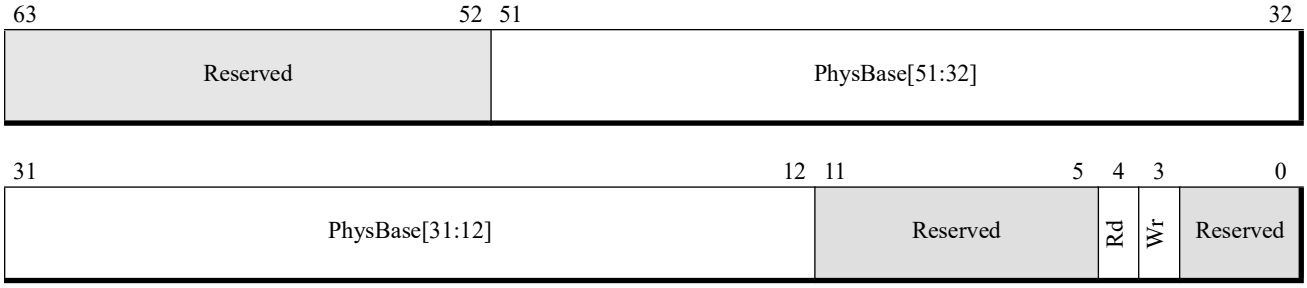
- *WrMem*—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.

[AMD Public Use]

- *RdMem*—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.
- *Range Physical-Base-Address (PhysBase)*—Bits 51:12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11:0 are assumed to be 0.

Note that a given processor may implement less than the architecturally-defined physical address size of 52 bits.

The format of these registers is shown in Figure 7-15.



Bits	Mnemonic	Description	Access type
63:52	Reserved		IGN
51:12	PhysBase	Range Physical Base Address	R/W
11:5	Reserved		IGN
4	Rd	RdMem Enable	R/W
3	Wr	WrMem Enable	R/W
2:0	Reserved		IGN

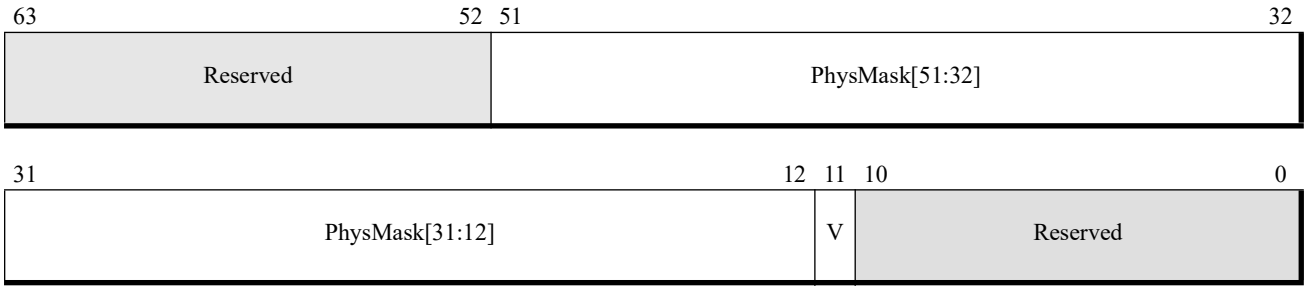
Figure 7-15. IORRBase*n* Register

IORRMask*n* Registers. The fields in these IORRs are:

- *Valid (V)*—Bit 11. Indicates that the IORR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0 the register pair is not used for memory-mapped I/O control (disabled).
- *Range Physical-Mask (PhysMask)*—Bits 51:12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11:0 of PhysMask are assumed to be 0.

The format of these registers is shown in Figure 7-16 on page 238.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:52	Reserved		IGN
51:12	PhysMask	Range Physical Mask	R/W
11	V	I/O Register Pair Enable (Valid)	R/W
10:0	Reserved		IGN

Figure 7-16. IORRMask*n* Register

The operation of the PhysMask and PhysBase fields is identical to that of the variable-range MTRRs. See page 224 for a description of this operation.

7.9.3 IORR Overlapping

The use of overlapping IORRs is not recommended. If overlapping IORRs are specified, the resulting behavior is implementation-dependent.

7.9.4 Top of Memory

The *top-of-memory* registers, TOP_MEM and TOP_MEM2, allow system software to specify physical addresses ranges as memory-mapped I/O locations. Processor implementations can direct accesses to memory-mapped I/O differently than system I/O, and the precise method depends on the implementation. System software specifies memory-mapped I/O regions by writing an address into each of the top-of-memory registers. The memory regions specified by the TOP_MEM registers are aligned on 8-Mbyte boundaries as follows:

- Memory accesses from physical address 0 to one less than the value in TOP_MEM are directed to system memory.
- Memory accesses from the physical address specified in TOP_MEM to FFFF_FFFFh are directed to memory-mapped I/O.
- Memory accesses from physical address 1_0000_0000h to one less than the value in TOP_MEM2 are directed to system memory.
- Memory accesses from the physical address specified in TOP_MEM2 to the maximum physical address supported by the system are directed to memory-mapped I/O.

[AMD Public Use]

Figure 7-17 on page 239 shows how the top-of-memory registers organize memory into separate system-memory and memory-mapped I/O regions.

The intersection of the top-of-memory range with the equivalent effective MTRR range follows the same type encoding table (Table 7-13 on page 236) as the fixed-range MTRR, where the RdMem/WrMem and memory type are directly tied together.

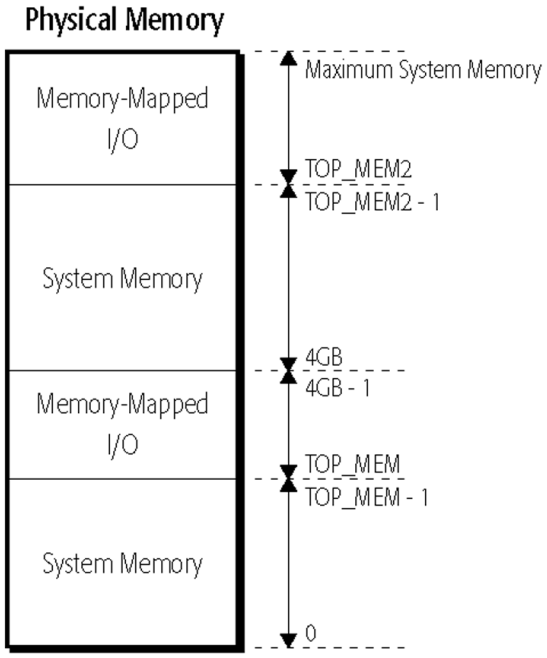


Figure 7-17. Memory Organization Using Top-of-Memory Registers

Figure 7-18 shows the format of the TOP_MEM and TOP_MEM2 registers. Bits 51:23 specify an 8-Mbyte aligned physical address. All remaining bits are reserved and ignored by the processor. System software should clear those bits to zero to maintain compatibility with possible future extensions to the registers. The TOP_MEM registers are model-specific registers. See “Memory-Typing MSRs” on page 768 for information on the MSR address and reset values for these registers.

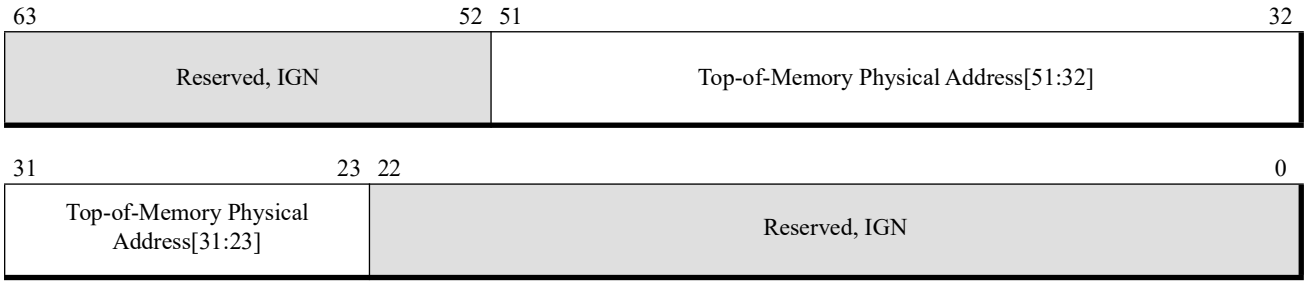


Figure 7-18. Top-of-Memory Registers (TOP_MEM, TOP_MEM2)

[AMD Public Use]

The TOP_MEM register is enabled by setting the MtrrVarDramEn bit in the SYSCFG MSR (bit 20) to 1 (one). The TOP_MEM2 register is enabled by setting the MtrrTom2En bit in the SYSCFG MSR (bit 21) to 1 (one). The registers are disabled when their respective enable bits are cleared to 0. When the top-of-memory registers are disabled, memory accesses default to memory-mapped I/O space.

Note that a given processor may implement fewer than the architecturally-defined number of physical address bits.

7.10 Secure Memory Encryption

Software running in non-virtualized (native) mode can utilize the Secure Memory Encryption (SME) feature to mark individual pages of memory as encrypted through the page tables. A page of memory marked encrypted will be automatically decrypted when read by software and automatically encrypted when written to DRAM. SME may therefore be used to protect the contents of DRAM from physical attacks on the system.

All memory encrypted using SME is encrypted with the same AES key which is created randomly each time a system is booted. The memory encryption key cannot be read or modified by software.

For details on using memory encryption in virtualized environments, please see Section 15.34, “Secure Encrypted Virtualization,” on page 613.

7.10.1 Determining Support for Secure Memory Encryption

Support for memory encryption features is reported in CPUID Fn8000_001F[EAX]. Bit 0 indicates support for Secure Memory Encryption. When this feature is present, CPUID Fn8000_001F[EBX] supplies additional information regarding the use of memory encryption such as which page table bit is used to mark pages as encrypted.

Additionally, in some implementations, the physical address size of the processor may be reduced when memory encryption features are enabled, for example from 48 to 43 bits. In this case the upper physical address bits are treated as reserved when the feature is enabled except where otherwise indicated. When memory encryption is supported in an implementation, CPUID Fn8000_001F[EBX] reports any physical address size reduction present. Bits reserved in this mode are treated the same as other page table reserved bits, and will generate a page fault if found to be non-zero when used for address translation.

Complete CPUID details for encrypted memory features can be found in Volume 3, section E.4.17.

7.10.2 Enabling Memory Encryption Extensions

Prior to using SME, memory encryption features must be enabled by setting SYSCFG MSR bit 23 (MemEncryptionModEn) to 1. In implementations where the physical address size of the processor is reduced when memory encryption features are enabled, software must ensure it is executing from addresses where these upper physical address bits are 0 prior to setting SYSCFG[MemEncryptionModEn]. Memory encryption is then further controlled via the page tables.

[AMD Public Use]

Note that software should keep the value of SYSCFG[MemEncryptionModEn] consistent across all CPU cores in the system. Failure to do so may lead to unexpected results.

7.10.3 Supported Operating Modes

SME is supported in all CPU modes when CR4.PAE=1 and paging is enabled. This includes long mode as well as legacy PAE-enabled protected mode.

7.10.4 Page Table Support

Software utilizes the page tables to indicate if a memory page is encrypted or unencrypted. The location of the specific attribute bit (C-bit, or enCrypted bit) used is implementation-specific but may be determined by referencing CPUID Fn8000_001F[EBX] (see Volume 3, section E.4.17 for details). In some implementations, the bit used may be a physical address bit (e.g., address bit 47), especially in cases where the physical address size is reduced by hardware when memory encryption features are enabled.

To mark a memory page for encryption when stored in DRAM, software sets the C-bit to 1 for the page. If the C-bit is 0, the page is not encrypted when stored in DRAM. The C bit can be applied to translation table entries for any size of page - 4KB, 2MB, or 1GB.

Note that it is possible for the page tables themselves to be located in encrypted memory. For instance, if the C-bit is set in a PML4 entry, the PDP table it points to (and thus all PDPEs in that table) will be loaded from encrypted memory.

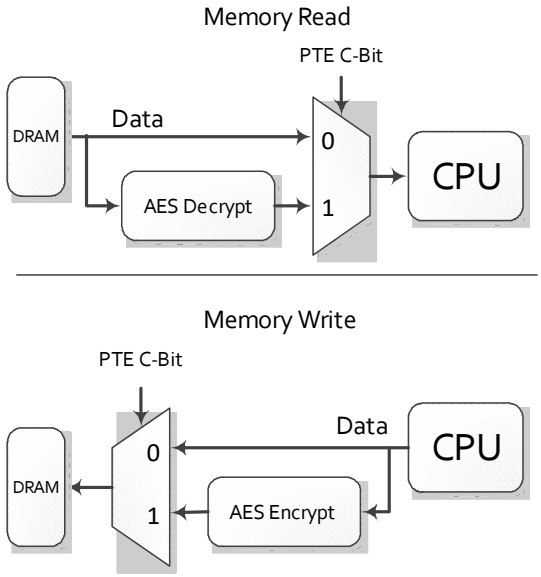


Figure 7-19. Encrypted Memory Accesses

[AMD Public Use]

7.10.5 I/O Accesses

In implementations where the physical address size is reduced when memory encryption features are enabled, memory range checks (e.g. MTRR/TOM/IORR/etc.) to determine memory types or DRAM/MMIO are performed using the reduced physical address size. In particular, the C-bit is not considered a physical address bit and is masked by hardware for purposes of these checks.

Additionally, any pages corresponding to MMIO addresses must be configured with the C-bit clear. Encrypted I/O pages are not allowed and accesses with the C-bit set will be ignored.

7.10.6 Restrictions

In some hardware implementations, coherency between the encrypted and unencrypted mappings of the same physical page are not enforced. In such a system, prior to changing the value of the C-bit for a page, software should flush the page from all CPU caches in the system. If a hardware implementation supports coherency across encryption domains as indicated by CPUID Fn8000_001F_EAX[10] then this flush is not required.

Simply changing the value of a C-bit on a page will not automatically encrypt the existing contents of a page, and any data in the page prior to the C-bit modification will become unintelligible. To set the C-bit on a page and cause its contents to become encrypted so the data remains accessible, see Section 7.10.8, “Encrypt-in-Place,” on page 242.

In legacy PAE mode, if the C-bit location is in the upper 32 bits of the page table entry, the first level page table (the PDP table) cannot be located in encrypted memory. This is because when the CPU is in 32-bit PAE mode, the CR3 value is only 32-bits in length.

7.10.7 SMM Interaction

SME is available when the processor is executing in SMM, once it has enabled paging. Any physical address bit restrictions that exist due to memory encryption features being enabled remain in place while in SMM.

7.10.8 Encrypt-in-Place

It is possible to perform an in-place encryption of data in physical memory. This technique is useful for setting the C-bit on a page while maintaining visibility to the page's contents such as during SME initialization. This is accomplished by creating two linear mappings of the same page where one mapping has the C-bit set to 0 and the other has the C-bit set to 1. To avoid possible data corruption, software should use the following algorithm for performing in-place encryption of memory:

1. Create two linear mappings X and Y that map to the same physical page. Mapping X has C-bit=0 and uses the WP (Write Protect) memory type. Mapping Y has C-bit=1 and uses the WB (Write-Back) memory type.
2. Perform a WBINVD on all cores in the system.

[AMD Public Use]

3. Copy N bytes from mapping X to a temporary buffer in conventionally-mapped memory (for which the C bit may or may not be set, as desired). N must be equal to the L1 cache line size as specified by CPUID Fn8000_0005[ECX].
4. Write N bytes from the temporary buffer to Y. Note that the initial cache refill of the line for this step will cause it to be decrypted, which corrupts the contents since it is not yet encrypted. This step restores the original contents. (If the line were evicted before this step was completed, the unwritten portion would get corrupted by the outgoing encryption, which is why the line can't be copied in-place, but rather must be copied from the temporary buffer.)
5. Repeat steps 3-4 until the entire page has been copied.

7.10.9 Secure Multi-Key Memory Encryption

The Multi-Key Secure Memory Encryption (SME-MK) feature is an extension of the Secure Memory Encryption (SME) feature. This feature allows software (OS or hypervisor) to load encryption keys into hardware memory controllers and to select which key is used to encrypt each page of memory. Encryption key loading also requires the use of the AMD Secure Processor (AMD-SP).

Support for SME-MK is reported by CPUID Fn8000_0023_EAX[MemHmk] (bit 0) = 1. When this feature is present, the number of simultaneously available host encryption key IDs (EncrKeyId) is reported by CPUID Fn8000_0023_EBX[MaxMemHmkEncrKeyID] (bits 15:0).

The MEM-MK feature is enabled by setting SYSCFG MSR bit 26 (HostMultiKeyMemEncrModeEn) to 1 and bit 23 (MemEncryptionModEn) to 0. In implementations where the physical address size of the processor is reduced when memory encryption features are enabled, software must ensure it is executing from addresses where these upper physical address bits are 0 prior to enabling the MEM-MK feature. Memory encryption is then further controlled via the page tables. Software should keep the value of SYSCFG MSR bits HostMultiKeyMemEncrModeEn and MemEncryptionModEn consistent across all CPU cores in the system. Failure to do so may lead to unexpected results.

When SME-MK is enabled, all system physical addresses have an EncrKeyID field defined as part of the system physical address (this includes CR3, the nested page table base, VMCB addresses, etc.). The EncrKeyID field specifies the index of the encryption key to use. An EncrKeyID value of zero indicates that encryption is not used for that access. The physical address space may be reduced and the upper physical address bits used to specify the EncrKeyID (the physical address size reduction is reported by CPUID Fn8000_001F_EBX[PhysAddrReduction]). EncrKeyID width (EncrKeyIdWidth) is equal to log2(CPUID Fn8000_0023_EBX[MaxMemHmkEncrKeyID] + 1). System physical address bits [CBitP:(CBitP-EncrKeyIdWidth + 1)] contain EncrKeyID, where CBitP is equal to CPUID Fn8000_001F_EBX[CbitPosition] (5:0).

[AMD Public Use]

8 Exceptions and Interrupts

Exceptions and interrupts force control transfers from the currently-executing program to a system-software service routine that handles the interrupting event. These routines are referred to as *exception handler*s and *interrupt handlers*, or collectively as *event handlers*. Typically, interrupt events can be handled by the service routine transparently to the interrupted program. During the control transfer to the service routine, the processor stops executing the interrupted program and saves its return pointer. The system-software service routine that handles the exception or interrupt is responsible for saving the state of the interrupted program. This allows the processor to restart the interrupted program after system software has handled the event.

When an exception or interrupt occurs, the processor uses the interrupt vector number as an index into the interrupt-descriptor table (IDT). An IDT is used in all processor operating modes, including real mode (also called real-address mode), protected mode, and long mode.

Exceptions and interrupts come from three general sources:

- *Exceptions* occur as a result of software execution errors or other internal-processor errors. Exceptions also occur during non-error situations, such as program single stepping or address-breakpoint detection. Exceptions are considered *synchronous* events because they are a direct result of executing the interrupted instruction.
- *Software interrupts* occur as a result of executing interrupt instructions. Unlike exceptions and external interrupts, software interrupts allow intentional triggering of the interrupt-handling mechanism. Like exceptions, software interrupts are synchronous events.
- *External interrupts* are generated by system logic in response to an error or some other event outside the processor. They are reported over the processor bus using external signaling. External interrupts are *asynchronous* events that occur independently of the interrupted instruction.

Throughout this section, the term *masking* can refer to either disabling or delaying an interrupt. For example, masking external interrupts *delays* the interrupt, with the processor holding the interrupt as pending until it is unmasked. With floating-point exceptions (SSE and x87), masking *prevents* an interrupt from occurring and causes the processor to perform a default operation on the exception condition.

8.1 General Characteristics

Exceptions and interrupts have several different characteristics that depend on how events are reported and the implications for program restart.

8.1.1 Precision

Precision describes how the exception is related to the interrupted program:

- *Precise* exceptions are reported on a predictable instruction boundary. This boundary is generally the first instruction that has not completed when the event occurs. All previous instructions (in

[AMD Public Use]

program order) are allowed to complete before transferring control to the event handler. The pointer to the instruction boundary is saved automatically by the processor. When the event handler completes execution, it returns to the interrupted program and restarts execution at the interrupted-instruction boundary.

- *Imprecise* exceptions are not guaranteed to be reported on a predictable instruction boundary. The boundary can be any instruction that has not completed when the interrupt event occurs. Imprecise events can be considered asynchronous, because the source of the interrupt is not necessarily related to the interrupted instruction. Imprecise exception and interrupt handlers typically collect machine-state information related to the interrupting event for reporting through system-diagnostic software. The interrupted program is not restartable.

8.1.2 Instruction Restart

As mentioned above, precise exceptions are reported on an instruction boundary. The instruction boundary can be reported in one of two locations:

- Most exceptions report the boundary *before* the instruction causing the exception. In this case, all previous instructions (in program order) are allowed to complete, but the interrupted instruction is not. *No program state is updated as a result of partially executing an interrupted instruction.*
- Some exceptions report the boundary *after* the instruction causing the exception. In this case, all previous instructions—including the one executing when the exception occurred—are allowed to complete.

  Program state can be updated when the reported boundary is after the instruction causing the exception. This is particularly true when the event occurs as a result of a task switch. In this case, the general registers, segment-selector registers, page-base address register, and LDTR are all updated by the hardware task-switch mechanism. The event handler cannot rely on the state of those registers when it begins execution and must be careful in validating the state of the segment-selector registers before restarting the interrupted task. This is not an issue in long mode, however, because the hardware task-switch mechanism is disabled in long mode.

8.1.3 Types of Exceptions

There are three types of exceptions, depending on whether they are precise and how they affect program restart:

- *Faults* are precise exceptions reported on the boundary before the instruction causing the exception. Generally, faults are caused by an error condition involving the faulted instruction. Any machine-state changes caused by the faulting instruction are discarded so that the instruction can be restarted. The saved rIP points to the faulting instruction.
- *Traps* are precise exceptions reported on the boundary following the instruction causing the exception. The trapped instruction is completed by the processor and all state changes are saved. The saved rIP points to the instruction following the faulting instruction.
- *Aborts* are imprecise exceptions. Because they are imprecise, aborts typically do not allow reliable program restart.

[AMD Public Use]

8.1.4 Masking External Interrupts

General Masking Capabilities. Software can *mask* the occurrence of certain exceptions and interrupts. Masking can delay or even prevent triggering of the exception-handling or interrupt-handling mechanism when an interrupt-event occurs. External interrupts are classified as maskable or nonmaskable:

- *Maskable interrupts* trigger the interrupt-handling mechanism only when RFLAGS.IF=1. Otherwise they are held pending for as long as the RFLAGS.IF bit is cleared to 0.
- *Nonmaskable interrupts* (NMI) are unaffected by the value of the RFLAGS.IF bit. However, the occurrence of an NMI masks further NMIs until an IRET instruction is executed to completion or, in the event of a task switch, to the completion of the outgoing TSS update. An exception raised during execution of the IRET prior to these points will result in NMI continuing to be masked for the duration of the exception handler, until the exception handler completes an IRET.

Masking During Stack Switches. The processor delays recognition of maskable external interrupts and debug exceptions during certain instruction sequences that are often used by software to switch stacks. The typical programming sequence used to switch stacks is:

1. Load a stack selector into the SS register.
2. Load a stack offset into the ESP register.

If an interrupting event occurs after the selector is loaded but before the stack offset is loaded, the interrupted-program stack pointer is invalid during execution of the interrupt handler.

To prevent interrupts from causing stack-pointer problems, the processor does not allow external interrupts or debug exceptions to occur until the instruction immediately following the MOV SS or POP SS instruction completes execution.

The recommended method of performing this sequence is to use the LSS instruction. LSS loads both SS and ESP, and the instruction inhibits interrupts until both registers are updated successfully.

8.1.5 Masking Floating-Point and Media Instructions

Any x87 floating-point exceptions can be masked and reported later using bits in the x87 floating-point status register (FSW) and the x87 floating-point control register (FCW). The floating-point exception-pending exception is used for unmasked x87 floating-point exceptions (see Section “#MF—x87 Floating-Point Exception-Pending (Vector 16)” on page 259).

The SIMD floating-point exception is used for unmasked SSE floating-point exceptions (see Section “#XF—SIMD Floating-Point Exception (Vector 19)” on page 261). SSE floating-point exceptions are masked using the MXCSR register. The exception mechanism is not triggered when these exceptions are masked. Instead, the processor handles the exceptions in a default manner.

[AMD Public Use]

8.1.6 Disabling Exceptions

Disabling an exception prevents the exception condition from being recognized, unlike masking an exception which prevents triggering the exception mechanism after the exception is recognized. Some exceptions can be disabled by system software running at CPL=0, using bits in the CR0 register or CR4 register:

- Alignment-check exception (see Section “#AC—Alignment-Check Exception (Vector 17)” on page 260).
- Device-not-available exception (see Section “#NM—Device-Not-Available Exception (Vector 7)” on page 253).
- Machine-check exception (see Section “#MC—Machine-Check Exception (Vector 18)” on page 261).

The debug-exception mechanism provides control over when specific breakpoints are enabled and disabled. See Section “Setting Breakpoints” on page 404 for more information on how breakpoint controls are used for triggering the debug-exception mechanism.

8.2 Vectors

Specific exception and interrupt sources are assigned a fixed vector-identification number (also called an “interrupt vector” or simply “vector”). The interrupt vector is used by the interrupt-handling mechanism to locate the system-software service routine assigned to the exception or interrupt. Up to 256 unique interrupt vectors are available. The first 32 vectors are reserved for predefined exception and interrupt conditions. Software-interrupt sources can trigger an interrupt using any available interrupt vector.

Table 8-1 on page 248 lists the supported interrupt vector numbers, the corresponding exception or interrupt name, the mnemonic, the source of the interrupt event, and a summary of the possible causes.

[AMD Public Use]

Table 8-1. Interrupt Vector Source and Cause

Vector	Exception/Interrupt	Mnemonic	Cause
0	Divide-by-Zero-Error	#DE	DIV, IDIV, AAM instructions
1	Debug	#DB	Instruction accesses and data accesses
2	Non-Maskable-Interrupt	#NMI	External NMI signal
3	Breakpoint	#BP	INT3 instruction
4	Overflow	#OF	INTO instruction
5	Bound-Range	#BR	BOUND instruction
6	Invalid-Opcode	#UD	Invalid instructions
7	Device-Not-Available	#NM	x87 instructions
8	Double-Fault	#DF	Exception during the handling of another exception or interrupt
9	Coprocessor-Segment-Overrun	—	Unsupported (Reserved)
10	Invalid-TSS	#TS	Task-state segment access and task switch
11	Segment-Not-Present	#NP	Segment register loads
12	Stack	#SS	SS register loads and stack references
13	General-Protection	#GP	Memory accesses and protection checks
14	Page-Fault	#PF	Memory accesses when paging enabled
15	Reserved	—	
16	x87 Floating-Point Exception-Pending	#MF	x87 floating-point instructions
17	Alignment-Check	#AC	Misaligned memory accesses
18	Machine-Check	#MC	Model specific
19	SIMD Floating-Point	#XF	SSE floating-point instructions
20	Reserved	—	
21	Control-Protection Exception	#CP	RET/IRET or other control transfer
22–27	Reserved	—	
28	Hypervisor Injection Exception	#HV	Event injection
29	VMM Communication Exception	#VC	Virtualization event
30	Security Exception	#SX	Security-sensitive event in host
31	Reserved	—	
0–255	External Interrupts (Maskable)	#INTR	External interrupts
0–255	Software Interrupts	—	INT*n* instruction

Table 8-2 on page 249 shows how each interrupt vector is classified. Reserved interrupt vectors are indicated by the gray-shaded rows.

[AMD Public Use]

Table 8-2. Interrupt Vector Classification

Vector	Interrupt (Exception)	Type	Precise	Class[2]
0	Divide-by-Zero-Error	Fault	yes	Contributory
1	Debug	Fault or Trap		Benign
2	Non-Maskable-Interrupt	—	—	
3	Breakpoint	Trap	yes	
4	Overflow			
5	Bound-Range	Fault		
6	Invalid-Opcode			
7	Device-Not-Available			
8	Double-Fault	Abort	no	
9	Coprocessor-Segment-Overrun			
10	Invalid-TSS	Fault	yes	Contributory
11	Segment-Not-Present			
12	Stack			
13	General-Protection			
14	Page-Fault			Benign or Contributory
15	Reserved			
16	x87 Floating-Point Exception-Pending	Fault	no	Benign
17	Alignment-Check		yes	
18	Machine-Check	Abort	no	
19	SIMD Floating-Point	Fault	yes	
20	Reserved			
21	Control Protection	Fault	yes	Contributory
22–27	Reserved			
28	Hypervisor Injection Exception	–	–	Benign
29	VMM Communication Exception	Fault	yes	Contributory
30	Security Exception	–	yes	Contributory
31	Reserved			
0–255	External Interrupts (Maskable)	—[1]	—[1]	Benign
0–255	Software Interrupts			

Note:

1. *External interrupts are not classified by type or whether or not they are precise.*
2. *See Section "#DF—Double-Fault Exception (Vector 8)" on page 253 for a definition of benign and contributory classes.*

The following sections describe each interrupt in detail. The format of the error code reported by each interrupt is described in Section "Error Codes" on page 264.

[AMD Public Use]

8.2.1 #DE—Divide-by-Zero-Error Exception (Vector 0)

A #DE exception occurs when the denominator of a DIV instruction or an IDIV instruction is 0. A #DE also occurs if the result is too large to be represented in the destination.

#DE cannot be disabled.

Error Code Returned. None.

Program Restart. #DE is a fault-type exception. The saved instruction pointer points to the instruction that caused the #DE.

8.2.2 #DB—Debug Exception (Vector 1)

When the debug-exception mechanism is enabled, a #DB exception can occur under any of the following circumstances:

- Instruction execution.
- Instruction single stepping.
- Data read.
- Data write.
- I/O read.
- I/O write.
- Task switch.
- Debug-register access, or *general detect fault (debug register access when DR7.GD=1).*
- Executing the INT1 instruction (opcode 0F1h).

#DB conditions are enabled and disabled using the debug-control register, DR7 and RFLAGS.TF. Each #DB condition is described in more detail in Section “Setting Breakpoints” on page 404.

The processor can be configured to generate a #DB exception as a trap following the successful execution of a bus lock when the processor is at privilege level > 0. Section 13.1.3.6 on page 409 describes the bus lock trap mechanism in further detail.

The conditions under which a bus lock occurs are described in Section 7.3.3 on page 197.

Error Code Returned. None. #DB information is returned in the debug-status register, DR6.

Program Restart. #DB can be either a fault-type or trap-type exception. In the following cases, the saved instruction pointer points to the instruction that caused the #DB:

- Instruction execution.
- Invalid debug-register access, or *general detect.*

In all other cases, the instruction that caused the #DB is completed, and the saved instruction pointer points to the instruction after the one that caused the #DB.

[AMD Public Use]

The RFLAGS.RF bit can be used to restart an instruction following an instruction breakpoint resulting in a #DB. In most cases, the processor clears RFLAGS.RF to 0 after every instruction is successfully executed. However, in the case of the IRET, JMP, CALL, and INT*n (*through a task gate) instructions, RFLAGS.RF is not cleared to 0 until the *next* instruction successfully executes.

When a non-debug exception occurs (or when a string instruction is interrupted), the processor normally sets RFLAGS.RF to 1 in the rFLAGS *image* that is pushed on the interrupt stack. A subsequent IRET back to the interrupted program pops the rFLAGS image off the stack and into the RFLAGS register, with RFLAGS.RF=1. The interrupted instruction executes without causing an instruction breakpoint, after which the processor clears RFLAGS.RF to 0.

However, when a #DB exception occurs, the processor clears RFLAGS.RF to 0 in the rFLAGS image that is pushed on the interrupt stack. The #DB handler has two options:

- Disable the instruction breakpoint completely.
- Set RFLAGS.RF to 1 in the interrupt-stack rFLAGS image. The instruction breakpoint condition is ignored immediately after the IRET, but reoccurs if the instruction address is accessed later, as can occur in a program loop.

8.2.3 NMI—Non-Maskable-Interrupt Exception (Vector 2)

An NMI exception occurs as a result of system logic signaling a non-maskable interrupt to the processor.

Error Code Returned. None.

Program Restart. NMI is an interrupt. The processor recognizes an NMI at an instruction boundary. The saved instruction pointer points to the instruction immediately following the boundary where the NMI was recognized.

Masking. NMI cannot be masked. However, when an NMI is recognized by the processor, recognition of subsequent NMIs are disabled until an IRET instruction is executed.

8.2.4 #BP—Breakpoint Exception (Vector 3)

A #BP exception occurs when an INT3 instruction is executed. The INT3 is normally used by debug software to set instruction breakpoints by replacing instruction-opcode bytes with the INT3 opcode.

#BP cannot be disabled.

Error Code Returned. None.

Program Restart. #BP is a trap-type exception. The saved instruction pointer points to the byte after the INT3 instruction. This location can be the start of the next instruction. However, if the INT3 is used to replace the first opcode bytes of an instruction, the restart location is likely to be in the middle of an instruction. In the latter case, the debug software must replace the INT3 byte with the correct instruction byte. The saved RIP instruction pointer must then be decremented by one before returning

[AMD Public Use]

to the interrupted program. This allows the program to be restarted correctly on the interrupted-instruction boundary.

8.2.5 #OF—Overflow Exception (Vector 4)

An #OF exception occurs as a result of executing an INTO instruction while the overflow bit in RFLAGS is set to 1 (RFLAGS.OF=1).

#OF cannot be disabled.

Error Code Returned. None.

Program Restart. #OF is a trap-type exception. The saved instruction pointer points to the instruction following the INTO instruction that caused the #OF.

8.2.6 #BR—Bound-Range Exception (Vector 5)

A #BR exception can occur as a result of executing the BOUND instruction. The BOUND instruction compares an array index (first operand) with the lower bounds and upper bounds of an array (second operand). If the array index is not within the array boundary, the #BR occurs.

#BR cannot be disabled.

Error Code Returned. None.

Program Restart. #BR is a fault-type exception. The saved instruction pointer points to the BOUND instruction that caused the #BR.

8.2.7 #UD—Invalid-Opcode Exception (Vector 6)

A #UD exception occurs when an attempt is made to execute an invalid or undefined opcode. The validity of an opcode often depends on the processor operating mode. A #UD occurs under the following conditions:

- Execution of any reserved or undefined opcode in any mode.
- Execution of the UD0, UD1 or UD2 instructions.
- Use of the LOCK prefix on an instruction that cannot be locked.
- Use of the LOCK prefix on a lockable instruction with a non-memory target location.
- Execution of an instruction with an invalid-operand type.
- Execution of the SYSENTER or SYSEXIT instructions in long mode.
- Execution of any of the following instructions in 64-bit mode: AAA, AAD, AAM, AAS, BOUND, CALL (opcode 9A), DAA, DAS, DEC, INC, INTO, JMP (opcode EA), LDS, LES, POP (DS, ES, SS), POPA, PUSH (CS, DS, ES, SS), PUSHA, SALC.
- Execution of the ARPL, LAR, LLDT, LSL, LTR, SLDT, STR, VERR, or VERW instructions when protected mode is not enabled, or when virtual-8086 mode is enabled.

[AMD Public Use]

- Execution of any legacy SSE instruction when CR4.OSFXSR is cleared to 0. (For further information, see Section “FXSAVE/FXRSTOR Support (OSFXSR)” on page 50.
- Execution of any SSE instruction (uses ZMM/YMM/XMM registers), or 64-bit media instruction (uses MMX™ registers) when CR0.EM = 1.
- Execution of any SSE floating-point instruction (uses ZMM/YMM/XMM registers) that causes a numeric exception when CR4.OSXMMEXCPT = 0.
- Use of the DR4 or DR5 debug registers when CR4.DE = 1.
- Execution of RSM when not in SMM mode.

See the specific instruction description (in the other volumes) for additional information on invalid conditions.

#UD cannot be disabled.

Error Code Returned. None.

Program Restart. #UD is a fault-type exception. The saved instruction pointer points to the instruction that caused the #UD.

8.2.8 #NM—Device-Not-Available Exception (Vector 7)

A #NM exception occurs under any of the following conditions:

- An FWAIT/WAIT instruction is executed when CR0.MP=1 and CR0.TS=1.
- Any x87 instruction other than FWAIT is executed when CR0.EM=1.
- Any x87 instruction is executed when CR0.TS=1. The CR0.MP bit controls whether the FWAIT/WAIT instruction causes an #NM exception when TS=1.
- Any 128-bit or 64-bit media instruction when CR0.TS=1.

#NM can be enabled or disabled under the control of the CR0.MP, CR0.EM, and CR0.TS bits as described above. See Section 3.1.1 for more information on the CR0 bits used to control the #NM exception.

Error Code Returned. None.

Program Restart. #NM is a fault-type exception. The saved instruction pointer points to the instruction that caused the #NM.

8.2.9 #DF—Double-Fault Exception (Vector 8)

A #DF exception can occur when a second exception occurs during the handling of a prior (first) exception or interrupt handler.

Usually, the first and second exceptions can be handled sequentially without resulting in a #DF. In this case, the first exception is considered *benign*, as it does not harm the ability of the processor to handle the second exception.

[AMD Public Use]

In some cases, however, the first exception adversely affects the ability of the processor to handle the second exception. These exceptions contribute to the occurrence of a #DF, and are called *contributory* exceptions. If a contributory exception is followed by another contributory exception, a double-fault exception occurs. Likewise, if a page fault is followed by another page fault or a contributory exception, a double-fault exception occurs.

Table 8-3 shows the conditions under which a #DF occurs. Page faults are either benign or contributory, and are listed separately. See the "Class" column in Table 8-2 on page 249 for information on whether an exception is benign or contributory.

Table 8-3. Double-Fault Exception Conditions

First Interrupting Event	Second Interrupting Event
Contributory Exceptions • Divide-by-Zero-Error Exception • Invalid-TSS Exception • Segment-Not-Present Exception • Stack Exception • General-Protection Exception	Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception
Page Fault Exception	Page Fault Exception Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception

If a third interrupting event occurs while transferring control to the #DF handler, the processor shuts down. Only an NMI, RESET, or INIT can restart the processor in this case. However, if the processor shuts down as it is executing an NMI handler, the processor can only be restarted with RESET or INIT.

#DF cannot be disabled.

Error Code Returned. Zero.

Program Restart. #DF is an abort-type exception. The saved instruction pointer is undefined, and the program cannot be restarted.

8.2.10 Coprocessor-Segment-Overrun Exception (Vector 9)

This interrupt vector is reserved. It is for a discontinued exception originally used by processors that supported external x87-instruction coprocessors. On those processors, the exception condition is caused by an invalid-segment or invalid-page access on an x87-instruction coprocessor-instruction operand. On current processors, this condition causes a general-protection exception to occur.

Error Code Returned. Not applicable.

Program Restart. Not applicable.

[AMD Public Use]

8.2.11 #TS—Invalid-TSS Exception (Vector 10)

A #TS exception occurs when an invalid reference is made to a segment selector as part of a task switch. A #TS also occurs during a privilege-changing control transfer (through a call gate or an interrupt gate), if a reference is made to an invalid stack-segment selector located in the TSS. Table 8-4 lists the conditions under which a #TS occurs and the error code returned by the exception mechanism.

#TS cannot be disabled.

Error Code Returned. See Table 8-4 for a list of error codes returned by the #TS exception.

Program Restart. #TS is a fault-type exception. If the exception occurs before loading the segment selectors from the TSS, the saved instruction pointer points to the instruction that caused the #TS. However, most #TS conditions occur due to errors with the loaded segment selectors. When an error is found with a segment selector, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the #TS exception mechanism. In this case, the saved instruction pointer points to the first instruction in the new task.

In long mode, a #TS cannot be caused by a task switch, because the hardware task-switch mechanism is disabled. A #TS occurs only as a result of a control transfer through a gate descriptor that results in an invalid stack-segment reference using an SS selector in the TSS. In this case, the saved instruction pointer always points to the control-transfer instruction that caused the #TS.

Table 8-4. Invalid-TSS Exception Conditions

Selector Reference	Error Condition	Error Code
Task-State Segment	TSS limit check on a task switch	TSS Selector Index
	TSS limit check on an inner-level stack pointer	
LDT Segment	LDT does not point to GDT	LDT Selector Index
	LDT reference outside GDT	
	GDT entry is not an LDT descriptor	
	LDT descriptor is not present	
Code Segment	CS reference outside GDT or LDT	CS Selector Index
	Privilege check (conforming DPL > CPL)	
	Privilege check (non-conforming DPL ≠ CPL)	
	Type check (CS not executable)	
Data Segment	Data segment reference outside GDT or LDT	DS, ES, FS or GS Selector Index
	Type check (data segment not readable)	
Stack Segment	SS reference outside GDT or LDT	SS Selector Index
	Privilege check (stack segment descriptor DPL ≠ CPL)	
	Privilege check (stack segment selector RPL ≠ CPL)	
	Type check (stack segment not writable)	

[AMD Public Use]

8.2.12 #NP—Segment-Not-Present Exception (Vector 11)

An #NP occurs when an attempt is made to load a segment or gate with a clear present bit, as described in the following situations:

- Using the MOV, POP, LDS, LES, LFS, or LGS instructions to load a segment selector (DS, ES, FS, and GS) that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Far transfer to a CS that is not present.
- Referencing a gate descriptor containing a clear present bit.
- Referencing a TSS descriptor containing a clear present bit. This includes attempts to load the TSS descriptor using the LTR instruction.
- Attempting to load a descriptor containing a clear present bit into the LDTR using the LLDT instruction.
- Loading a segment selector (CS, DS, ES, FS, or GS) as part of a task switch, with the segment descriptor referenced by the segment selector having a clear present bit. In long mode, an #NP cannot be caused by a task switch, because the hardware task-switch mechanism is disabled.

When loading a stack-segment selector (SS) that references a descriptor with a clear present bit, a stack exception (#SS) occurs. For information on the #SS exception, see the next section, "#SS—Stack Exception (Vector 12)."

#NP cannot be disabled.

Error Code Returned. The segment-selector index of the segment descriptor causing the #NP exception.

Program Restart. #NP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that loaded the segment selector resulting in the #NP. See Section "Exceptions During a Task Switch" on page 264 for a description of the consequences when this exception occurs during a task switch.

8.2.13 #SS—Stack Exception (Vector 12)

An #SS exception can occur in the following situations:

- Implied stack references in which the stack address is not in canonical form. Implied stack references include all push and pop instructions, and any instruction using RSP or RBP as a base register.
- Attempting to load a stack-segment selector that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Any stack access that fails the stack-limit check.

#SS cannot be disabled.

Error Code Returned. The error code depends on the cause of the #SS, as shown in Table 8-5 on page 257:

[AMD Public Use]

Table 8-5. Stack Exception Error Codes

Stack Exception Cause	Error Code
Stack-segment descriptor present bit is clear	SS Selector Index
Stack-limit violation	0
Stack reference using a non-canonical address	0

Program Restart. #SS is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #SS. See Section “Exceptions During a Task Switch” on page 264 for a description of the consequences when this exception occurs during a task switch.

8.2.14 #GP—General-Protection Exception (Vector 13)

Table 8-6 describes the general situations that can cause a #GP exception. The table is not an exhaustive, detailed list of #GP conditions, but rather a guide to the situations that can cause a #GP. If an invalid use of an AMD64 architectural feature results in a #GP, the specific cause of the exception is described in detail in the section describing the architectural feature.

#GP cannot be disabled.

Error Code Returned. As shown in Table 8-6, a selector index is reported as the error code if the #GP is due to a segment-descriptor access. In all other cases, an error code of 0 is returned.

Program Restart. #GP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #GP. See Section “Exceptions During a Task Switch” on page 264 for a description of the consequences when this exception occurs during a task switch.

Table 8-6. General-Protection Exception Conditions

Error Condition	Error Code
Any segment privilege-check violation, while loading a segment register.	Selector Index
Any segment type-check violation, while loading a segment register.	
Loading a null selector into the CS, SS, or TR register.	
Accessing a gate-descriptor containing a null segment selector.	
Referencing an LDT descriptor or TSS descriptor located in the LDT.	
Attempting a control transfer to a busy TSS (except IRET).	
In 64-bit mode, loading a non-canonical base address into the GDTR or IDTR.	
In long mode, accessing a system or call-gate descriptor whose extended type field is not 0.	
In long mode, accessing a system descriptor containing a non-canonical base address.	
In long mode, accessing a gate descriptor containing a non-canonical offset.	
In long mode, accessing a gate descriptor that does not point to a 64-bit code segment.	
In long mode, accessing a 16-bit gate descriptor.	
In long mode, attempting a control transfer to a TSS or task gate.	

[AMD Public Use]

Table 8-6. General-Protection Exception Conditions (continued)

Error Condition	Error Code
Any segment limit-check or non-canonical address violation (except when using the SS register).	0
Accessing memory using a null segment register.	
Writing memory using a read-only segment register.	
Attempting to execute an SSE instruction specifying an unaligned memory operand.	
Attempting to execute code that is past the CS segment limit or at a non-canonical RIP.	
Executing a privileged instruction while CPL is > 0.	
Executing an instruction that is more than 15 bytes long.	
Writing a 1 into any register field that is reserved, must be zero (MBZ).	
Using WRMSR to write a read-only MSR.	
Using WRMSR to write a non-canonical value into an MSR that must be canonical.	
Using WRMSR to set an invalid type encoding in an MTRR or the PAT MSR.	0
Enabling paging while protected mode is disabled.	
Setting CR0.NW=1 while CR0.CD=0.	
Any long-mode consistency-check violation.	

8.2.15 #PF—Page-Fault Exception (Vector 14)

A #PF exception can occur during a memory access in any of the following situations:

- A page-translation-table entry or physical page involved in translating the memory access is not present in physical memory. This is indicated by a cleared present bit (P=0) in the translation-table entry.
- An attempt is made by the processor to load the instruction TLB with a translation for a non-executable page.
- The memory access fails the paging-protection checks (user/supervisor, read/write, or both).
- A reserved bit in one of the page-translation-table entries is set to 1. A #PF occurs for this reason only when CR4.PSE=1 *or* CR4.PAE=1.
- A data access to a user-mode address caused a protection key violation.

#PF cannot be disabled.

CR2 Register. The virtual (linear) address that caused the #PF is stored in the CR2 register. The legacy CR2 register is 32 bits long. The CR2 register in the AMD64 architecture is 64 bits long, as shown in Figure 8-1 on page 259. In AMD64 implementations, when either software or a page fault causes a write to the CR2 register, only the low-order 32 bits of CR2 are used in legacy mode; the processor clears the high-order 32 bits.

[AMD Public Use]

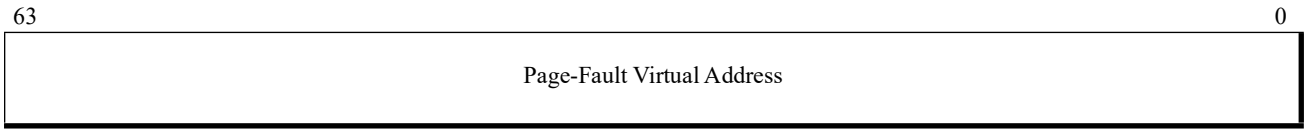


Figure 8-1. Control Register 2 (CR2)

Error Code Returned. The page-fault error code is pushed onto the page-fault exception-handler stack. See Section “Page-Fault Error Code” on page 265 for a description of this error code.

Program Restart. #PF is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #PF. See Section “Exceptions During a Task Switch” on page 264 for a description of what can happen if this exception occurs during a task switch.

8.2.16 #MF—x87 Floating-Point Exception-Pending (Vector 16)

The #MF exception is used to handle unmasked x87 floating-point exceptions. An #MF occurs when *all* of the following conditions are true:

- CR0.NE=1.
- An unmasked x87 floating-point exception is pending. This is indicated by an exception bit in the x87 floating-point status-word register being set to 1
- The corresponding mask bit in the x87 floating-point control-word register is cleared to 0.
- The FWAIT/WAIT instruction or any waiting floating-point instruction is executed.

If there is an exception mask bit (in the FPU control word) set, the exception is not reported. Instead, the x87-instruction unit responds in a default manner and execution proceeds normally.

The x87 floating-point exceptions reported by the #MF exception are (including mnemonics):

- IE—Invalid-operation exception (also called #I), which is either:
  - IE alone—Invalid arithmetic-operand exception (also called #IA), or
  - SF and IE together—x87 Stack-fault exception (also called #IS).
- DE—Denormalized-operand exception (also called #D).
- ZE—Zero-divide exception (also called #Z).
- OE—Overflow exception (also called #O).
- UE—Underflow exception (also called #U).
- PE—Precision exception (also called #P or inexact-result exception).

Error Code Returned. None. Exception information is provided by the x87 status-word register. See “x87 Floating-Point Programming” in Volume 1 for more information on using this register.

Program Restart. #MF is a fault-type exception. The #MF exception is not precise, because multiple instructions and exceptions can occur before the #MF handler is invoked. Also, the saved instruction

[AMD Public Use]

pointer does not point to the instruction that caused the exception resulting in the #MF. Instead, the saved instruction pointer points to the x87 floating-point instruction or FWAIT/WAIT instruction that is about to be executed when the #MF occurs. The address of the *last instruction* that caused an x87 floating-point exception is in the x87 instruction-pointer register. See "x87 Floating-Point Programming" in Volume 1 for information on accessing this register.

Masking. Each type of x87 floating-point exception can be masked by setting the appropriate bits in the x87 control-word register. See "x87 Floating-Point Programming" in Volume 1 for more information on using this register.

#MF can also be disabled by clearing the CR0.NE bit to 0. See Section "Numeric Error (NE) Bit" on page 44 for more information on using this bit.

8.2.17 #AC—Alignment-Check Exception (Vector 17)

An #AC exception occurs when an unaligned-memory data reference is performed while alignment checking is enabled.

After a processor reset, #AC exceptions are disabled. Software enables the #AC exception by setting the following register bits:

- CR0.AM=1.
- RFLAGS.AC=1.

When the above register bits are set, an #AC can occur only when CPL=3. #AC never occurs when CPL < 3.

Table 8-7 lists the data types and the alignment boundary required to avoid an #AC exception when the mechanism is enabled.

Table 8-7. Data-Type Alignment

Supported Data Type	Required Alignment (Byte Boundary)
Word	2
Doubleword	4
Quadword	8
Bit string	2, 4 or 8 (depends on operand size)
256-bit media	16
128-bit media	16
64-bit media	8
Segment selector	2
32-bit near pointer	4
32-bit far pointer	2
48-bit far pointer	4

[AMD Public Use]

Table 8-7. Data-Type Alignment (continued)

Supported Data Type	Required Alignment (Byte Boundary)
x87 Floating-point single-precision	4
x87 Floating-point double-precision	8
x87 Floating-point extended-precision	8
x87 Floating-point save areas	2 or 4 (depends on operand size)

Error Code Returned. Zero.

Program Restart. #AC is a fault-type exception. The saved instruction pointer points to the instruction that caused the #AC.

8.2.18 #MC—Machine-Check Exception (Vector 18)

The #MC exception is model specific. Processor implementations are not required to support the #MC exception, and those implementations that do support #MC can vary in how the #MC exception mechanism works.

The exception is enabled by setting CR4.MCE to 1. The machine-check architecture can include model-specific masking for controlling the reporting of some errors. Refer to Chapter 9, “Machine Check Architecture,” for more information.

Error Code Returned. None. Error information is provided by model-specific registers (MSRs) defined by the machine-check architecture.

Program Restart. #MC is an abort-type exception. If the RIPV flag (RIP valid) is set in the MCG_Status MSR, the saved CS and rIP point to the instruction at which the interrupted process thread can be restarted. If RIPV is clear, there is no reliable way to restart the program. If the EIPV flag (EIP valid) is set in the MCG_Status MSR, the saved CS and rIP point to the process thread that caused the error. If EIPV is clear, the CS:rIP of the instruction causing the failure is not known or the machine check is not related to a specific instruction.

8.2.19 #XF—SIMD Floating-Point Exception (Vector 19)

The #XF exception is used to handle unmasked SSE floating-point exceptions. A #XF exception occurs when all of the following conditions are true:

- A SSE floating-point exception occurs. The exception causes the processor to set the appropriate exception-status bit in the MXCSR register to 1.
- The exception-mask bit in the MXCSR that corresponds to the SSE floating-point exception is clear (=0).
- CR4.OSXMMEXCPT=1, indicating that the operating system supports handling of SSE floating-point exceptions.

[AMD Public Use]

The exception-mask bits are used by software to specify the handling of SSE floating-point exceptions. When the corresponding mask bit is cleared to 0, an exception occurs under the control of the CR4.OSXMMEXCPT bit. However, if the mask bit is set to 1, the SSE floating-point unit responds in a default manner and execution proceeds normally.

The CR4.OSXMMEXCPT bit specifies the interrupt vector to be taken when an unmasked SSE floating-point exception occurs. When CR4.OSXMMEXCPT=1, the #XF interrupt vector is taken when an exception occurs. When CR4.OSXMMEXCPT=0, the #UD (undefined opcode) interrupt vector is taken when an exception occurs.

The SSE floating-point exceptions reported by the #XF exception are (including mnemonics):

- IE—Invalid-operation exception (also called #I).
- DE—Denormalized-operand exception (also called #D).
- ZE—Zero-divide exception (also called #Z).
- OE—Overflow exception (also called #O).
- UE—Underflow exception (also called #U).
- PE—Precision exception (also called #P or inexact-result exception).

Each type of SSE floating-point exception can be masked by setting the appropriate bits in the MXCSR register. #XF can also be disabled by clearing the CR4.OSXMMEXCPT bit to 0.

Error Code Returned. None. Exception information is provided by the SSE floating-point MXCSR register. See “Instruction Reference” in Volume 4 for more information on using this register.

Program Restart. #XF is a fault-type exception. Unlike the #MF exception, the #XF exception is precise. The saved instruction pointer points to the instruction that caused the #XF.

8.2.20 #CP—Control-Protection Exception (Vector 21)

A #CP exception is generated when shadow stacks are enabled (CR4.CET=1) and any of the following situations occur:

- For RET or IRET instructions, the return addresses on the shadow stack and the data stack do not match.
- An invalid supervisor shadow stack token is encountered by the CALL, RET, IRET, SETSSBSY or RSTORSSP instructions or during the delivery of an interrupt or exception.
- For inter-privilege RET and IRET instructions, the SSP is not 8-byte aligned, or the previous SSP from shadow stack is not 4-byte aligned or, in legacy or compatibility mode, is not less than 4GB.
- A task switch initiated by IRET where the incoming SSP is not aligned to 4 bytes or is not less than 4GB.

Error Code Returned. The #CP error code is pushed onto the control-protection exception-handler stack. See Section “Control-Protection Error Code” on page 265 for a description of this error code.

[AMD Public Use]

Program Restart. #CP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #CP. See Section “Exceptions During a Task Switch” on page 264 for a description of what can happen if this exception occurs during a task switch.

8.2.21 #HV—Hypervisor Injection Exception (Vector 28)

The #HV exception may be injected by the hypervisor into a secure guest VM to notify the VM of pending events. See Section 15.36.16 for details.

8.2.22 #VC—VMM Communication Exception (Vector 29)

The #VC exception is generated when certain events occur inside a secure guest VM. See Section 15.35.5 for more details.

8.2.23 #SX—Security Exception (Vector 30)

The #SX exception is generated by security-sensitive events under SVM. See Section 15.28 for details.

8.2.24 User-Defined Interrupts (Vectors 32–255)

User-defined interrupts can be initiated either by system logic or software. They occur when:

- System logic signals an external interrupt request to the processor. The signaling mechanism and the method of communicating the interrupt vector to the processor are implementation dependent.
- Software executes an INT*n* instruction. The INT*n* instruction operand provides the interrupt vector number.

Both methods can be used to initiate an interrupt into vectors 0 through 255. However, because vectors 0 through 31 are defined or reserved by the AMD64 architecture, software should not use vectors in this range for purposes other than their defined use.

Error Code Returned. None.

Program Restart. The saved instruction pointer depends on the interrupt source:

- External interrupts are recognized on instruction boundaries. The saved instruction pointer points to the instruction immediately following the boundary where the external interrupt was recognized.
- If the interrupt occurs as a result of executing the INT*n* instruction, the saved instruction pointer points to the instruction after the INT*n*.

Masking. The ability to mask user-defined interrupts depends on the interrupt source:

- External interrupts can be masked using the RFLAGS.IF bit. Setting RFLAGS.IF to 1 enables external interrupts, while clearing RFLAGS.IF to 0 inhibits them.
- Software interrupts (initiated by the INT*n* instruction) cannot be disabled.

[AMD Public Use]

8.3 Exceptions During a Task Switch

An exception can occur during a task switch while loading a segment selector. Page faults can also occur when accessing a TSS. In these cases, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the appropriate exception mechanism. No other checks are performed. When this happens, the saved instruction pointer points to the first instruction in the new task.

In long mode, an exception cannot occur during a task switch, because the hardware task-switch mechanism is disabled.

8.4 Error Codes

The processor exception-handling mechanism reports error and status information for some exceptions using an error code. The error code is pushed onto the stack by the exception mechanism during the control transfer into the exception handler. The error code formats are described in the following sections.

8.4.1 Selector-Error Code

Figure 8-2 shows the format of the selector-error code.

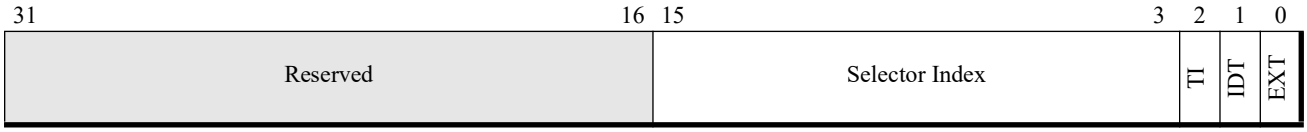


Figure 8-2. Selector Error Code

The information reported by the selector-error code includes:

- *EXT*—Bit 0. If this bit is set to 1, the exception source is external to the processor. If cleared to 0, the exception source is internal to the processor.
- *IDT*—Bit 1. If this bit is set to 1, the error-code selector-index field references a gate descriptor located in the interrupt-descriptor table (IDT). If cleared to 0, the selector-index field references a descriptor in either the global-descriptor table (GDT) or local-descriptor table (LDT), as indicated by the TI bit.
- *TI*—Bit 2. If this bit is set to 1, the error-code selector-index field references a descriptor in the LDT. If cleared to 0, the selector-index field references a descriptor in the GDT. The TI bit is relevant only when the IDT bit is cleared to 0.
- *Selector Index*—Bits 15:3. The selector-index field specifies the index into either the GDT, LDT, or IDT, as specified by the IDT and TI bits.

Some exceptions return a zero in the selector-error code.

[AMD Public Use]

8.4.2 Page-Fault Error Code

Figure 8-4 shows the format of the page-fault error code.

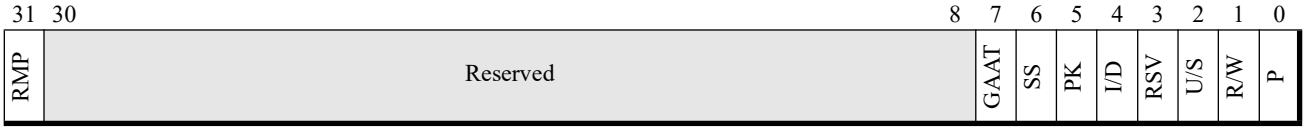


Figure 8-3. Page-Fault Error Code

The information reported by the page-fault error code includes:

- *P*—Bit 0. If this bit is cleared to 0, the page fault was caused by a not-present page. If this bit is set to 1, the page fault was caused by a page-protection violation.
- *R/W*—Bit 1. If this bit is cleared to 0, the access that caused the page fault is a memory read. If this bit is set to 1, the memory access that caused the page fault was a write. This bit does not necessarily indicate the cause of the page fault was a read or write violation.
- *U/S*—Bit 2. If this bit is cleared to 0, an access in supervisor mode (CPL=0, 1, or 2) caused the page fault. If this bit is set to 1, an access in user mode (CPL=3) caused the page fault. This bit does not necessarily indicate the cause of the page fault was a privilege violation.
- *RSV*—Bit 3. If this bit is set to 1, the page fault is a result of the processor reading a 1 from a reserved field within a page-translation-table entry. This type of page fault occurs only when CR4.PSE=1 or CR4.PAE=1. If this bit is cleared to 0, the page fault was not caused by the processor reading a 1 from a reserved field.
- *I/D*—Bit 4. If this bit is set to 1, it indicates that the access that caused the page fault was an instruction fetch. Otherwise, this bit is cleared to 0. This bit is only defined if no-execute feature is enabled (EFER.NXE=1 && CR4.PAE=1).
- *PK*—Bit 5. If this bit is set to 1, it indicates that a data access to a user-mode address caused a protection key violation. This fault only occurs if memory protection keys are enabled (CR4.PKE=1).
- *SS*—Bit 6. If this bit is set to 1, the page fault was caused by a shadow stack access. This bit is only set when the shadow stack feature is enabled (CR4.CET=1).
- *GAAT*—Bit 7. If this bit is set to 1, the page fault occurred during a GAAT table walk and was caused by a not-present page (P=0) or a reserved bit error (RSV=1).
- *RMP*—Bit 31. If this bit is set to 1, the page fault is a result of the processor encountering an RMP violation. This type of page fault only occurs when SYSCFG[SecureNestedPagingEn]=1. If this bit is cleared to 0, the page fault was not caused by an RMP violation. See section 15.36.10 for additional information.

8.4.3 Control-Protection Error Code

Figure 8-4 shows the format of the #CP error code.

[AMD Public Use]

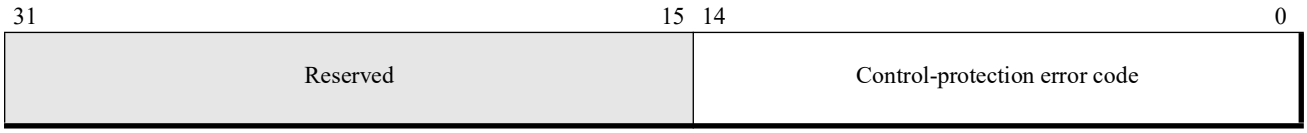


Figure 8-4. Control-Protection Error Code

The control-protection error codes are defined below:

Table 8-8. Control-Protection Error Codes

Error Code Value	Name	Cause
1	NEAR-RET	A RET (near) instruction encountered a return address mismatch.
2	FAR-RET/IRET	A RET (far) or IRET instruction encountered a return address mismatch.
4	RSTORSSP	An RSTORSSP instruction encountered an invalid shadow stack restore token.
5	SETSSBSY	A SETSSBSY instruction encountered an invalid supervisor shadow stack token.

8.5 Priorities

To allow for consistent handling of multiple-interrupt conditions, simultaneous interrupts are prioritized by the processor. The AMD64 architecture defines priorities between groups of interrupts, and interrupt prioritization within a group is implementation dependent. Table 8-9 shows the interrupt priorities defined by the AMD64 architecture.

When simultaneous interrupts occur, the processor transfers control to the highest-priority interrupt handler. Lower-priority interrupts from external sources are held pending by the processor, and they are handled after the higher-priority interrupt is handled. Lower-priority interrupts that result from internal sources are discarded. Those interrupts reoccur when the high-priority interrupt handler completes and transfers control back to the interrupted instruction. Software interrupts are discarded as well, and reoccur when the software-interrupt instruction is restarted.

Table 8-9. Simultaneous Interrupt Priorities

Interrupt Priority	Interrupt Condition	Interrupt Vector
(High)	Processor Reset	—
0	Machine-Check Exception	18
1	External Processor Initialization (INIT)	—
	SMI Interrupt	
	External Clock Stop (Stpclk)	

[AMD Public Use]

Table 8-9. Simultaneous Interrupt Priorities (continued)

Interrupt Priority	Interrupt Condition	Interrupt Vector
2	Data, and I/O Breakpoint (Debug Register)	1
	Single-Step Execution Instruction Trap (RFLAGS.TF=1)	
3	Non-Maskable Interrupt	2
4	Maskable External Interrupt (INTR)	32–255
5	Instruction Breakpoint (Debug Register)	1
	Code-Segment-Limit Violation[1]	13
	Instruction-Fetch Page Fault[1]	14
6	Invalid Opcode Exception[1]	6
	Device-Not-Available Exception	7
	Instruction-Length Violation (> 15 Bytes)	13
7	Divide-by-zero Exception	0
	Invalid-TSS Exception	10
	Segment-Not-Present Exception	11
	Stack Exception	12
	General-Protection Exception	13
	Data-Access Page Fault	14
	Floating-Point Exception-Pending Exception	16
	Alignment-Check Exception	17
	SIMD Floating-Point Exception	19
	Control-Protection Exception	21
	Hypervisor Injection Exception	28
	VMM Communication Exception	29

Note:

1. This reflects the relative priority for faults encountered when fetching the first byte of an instruction. In the fetching and decoding of subsequent bytes of an instruction, if those bytes span the segment limit or cross into a non-executable or not-present page, the fetch will result in a #GP(0) fault or #PF as appropriate, preventing those bytes from being accessed. However, if the instruction can be determined to be invalid based just on the bytes preceding that boundary, a #UD fault may take priority. This behavior is model-dependent.

8.5.1 Floating-Point Exception Priorities

Floating-point exceptions (SSE and x87 floating-point) can be handled in one of two ways:

- Unmasked exceptions are reported in the appropriate floating-point status register, and a software-interrupt handler is invoked. See Section “#MF—x87 Floating-Point Exception-Pending (Vector 16)” on page 259 and Section “#XF—SIMD Floating-Point Exception (Vector 19)” on page 261 for more information on the floating-point interrupts.

[AMD Public Use]

- Masked exceptions are also reported in the appropriate floating-point status register. Instead of transferring control to an interrupt handler, however, the processor handles the exception in a default manner and execution proceeds.

If the processor detects more than one exception while executing a single floating-point instruction, it prioritizes the exceptions in a predictable manner. When responding in a default manner to masked exceptions, it is possible that the processor acts only on the high-priority exception and ignores lower-priority exceptions. In the case of vector (SIMD) floating-point instructions, priorities are set on sub-operations, not across all operations. For example, if the processor detects and acts on a QNaN operand in one sub-operation, the processor can still detect and act on a denormal operand in another sub-operation.

When reporting SSE floating-point exceptions before taking an interrupt or handling them in a default manner, the processor first classifies the exceptions as follows:

- *Input exceptions* include SNaN operand (#I), invalid operation (#I), denormal operand (#D), or zero-divide (#Z). Using a NaN operand with a maximum, minimum, compare, or convert instruction is also considered an input exception.
- *Output exceptions* include numeric overflow (#O), numeric underflow (#U), and precision (#P).

Using the above classification, the processor applies the following procedure to report the exceptions:

1. The exceptions for all sub-operations are prioritized.
2. The exception conditions for all sub-operations are logically ORed together to form a single set of exceptions covering all operations. For example, if two sub-operations produce a denormal result, only one denormal exception is reported.
3. If the set of exceptions includes any *unmasked* input exceptions, all input exceptions are reported in MCXSR, and no output exceptions are reported. Otherwise, all input and output exceptions are reported in MCXSR.
4. If any exceptions are unmasked, control is transferred to the appropriate interrupt handler.

Table 8-10 on page 268 lists the priorities for simultaneous floating-point exceptions.

Table 8-10. Simultaneous Floating-Point Exception Priorities

Exception Priority	Exception Condition	
(High) 0	SNaN Operand	#I
	NaN Operand of Maximum, Minimum, Compare, and Convert Instructions (Vector Floating-Point)	
	Stack Overflow (x87 Floating-Point)	
	Stack Underflow (x87 Floating-Point)	
1	QNaN Operand	—
2	Invalid Operation (Remaining Conditions)	#I
	Zero Divide	#Z

[AMD Public Use]

Table 8-10. Simultaneous Floating-Point Exception Priorities (continued)

Exception Priority	Exception Condition	
3	Denormal Operand	#D
4	Numeric Overflow	#O
	Numeric Underflow	#U
5 (Low)	Precision	#P

8.5.2 External Interrupt Priorities

The AMD64 architecture allows software to define up to 15 external interrupt-priority classes. Priority classes are numbered from 1 to 15, with priority-class 1 being the lowest and priority-class 15 the highest. The organization of these priority classes is implementation dependent. A typical method is to use the upper four bits of the interrupt vector number to define the priority. Thus, interrupt vector 53h has a priority of 5 and interrupt vector 37h has a priority of 3.

A new control register (CR8) is introduced by the AMD64 architecture for managing priority classes. This register, called the *task-priority register* (TPR), uses its four low-order bits to specify a task priority. The remaining 60 bits are reserved and must be written with zeros. Figure 8-5 shows the format of the TPR.

The TPR is available only in 64-bit mode.

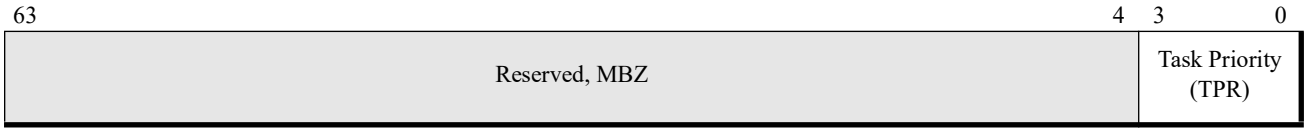


Figure 8-5. Task Priority Register (CR8)

System software can use the TPR register to temporarily block low-priority interrupts from interrupting a high-priority task. This is accomplished by loading TPR with a value corresponding to the highest-priority interrupt that is to be blocked. For example, loading TPR with a value of 9 (1001b) blocks all interrupts with a priority class of 9 or less, while allowing all interrupts with a priority class of 10 or more to be recognized. Loading TPR with 0 enables all external interrupts. Loading TPR with 15 (1111b) disables all external interrupts. The TPR is cleared to 0 on reset.

System software reads and writes the TPR using a MOV CR8 instruction. The MOV CR8 instruction requires a privilege level of 0. Programs running at any other privilege level cannot read or write the TPR, and an attempt to do so results in a general-protection exception (#GP).

A serializing instruction is not required after loading the TPR, because a new priority level is established when the MOV instruction completes execution. For example, assume two sequential TPR loads are performed, in which a low value is first loaded into TPR and immediately followed by a load

[AMD Public Use]

of a higher value. Any pending, lower-priority interrupt enabled by the first MOV CR8 is recognized between the two MOVs.

The TPR is an architectural abstraction of the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external system device, or it can be integrated on the chip like the local advanced programmable interrupt controller (APIC). Typically, the IC contains a priority mechanism similar, if not identical to, the TPR. The IC, however, is implementation dependent, and the underlying priority mechanisms are subject to change. The TPR, by contrast, is part of the AMD64 architecture.

Effect of IC on TPR. The features of the implementation-specific IC can impact the operation of the TPR. For example, the TPR might affect interrupt delivery only if the IC is enabled. Also, the mapping of an external interrupt to a specific interrupt priority is an implementation-specific behavior of the IC.

While the CR8 register provides the same functionality as the TPR at offset 80h of the local APIC, software should only use one mechanism to access the TPR. For example, updating the TPR with a write to the local APIC offset 0x80 but then reading it with a MOV CR8 is not guaranteed to return the same value that was written by the local APIC write.

8.6 Real-Mode Interrupt Control Transfers

In real mode, the IDT is a table of 4-byte entries, one entry for each of the 256 possible interrupts implemented by the system. The real mode IDT is often referred to as an *interrupt vector table*, or IVT. Table entries contain a far pointer (CS:IP pair) to an exception or interrupt handler. The base of the IDT is stored in the IDTR register, which is loaded with a value of 00h during a processor reset. Figure 8-6 on page 271 shows how the real-mode interrupt handler is located by the interrupt mechanism.

[AMD Public Use]

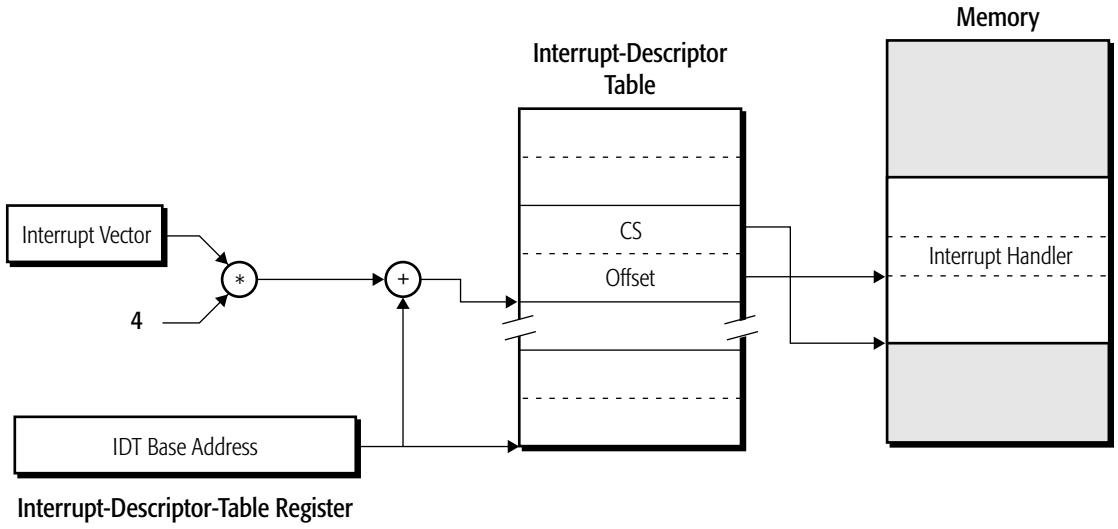


Figure 8-6. Real-Mode Interrupt Control Transfer

When an exception or interrupt occurs in real mode, the processor performs the following:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears EFLAGS.IF to 0 and EFLAGS.TF to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the IDT by scaling the interrupt vector by four and adding the result to the value in the IDTR.
5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

Figure 8-7 on page 272 shows the stack after control is transferred to the interrupt handler in real mode.

[AMD Public Use]

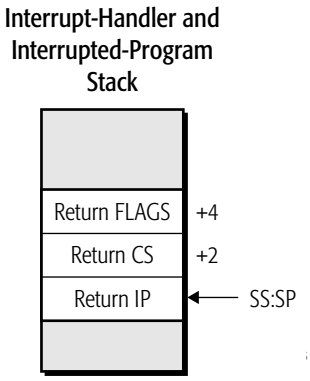


Figure 8-7. Stack After Interrupt in Real Mode

An IRET instruction is used to return to the interrupted program. When an IRET is executed, the processor performs the following:

1. Pops the saved CS value off the stack and into the CS register. The saved IP value is popped into RIP[15:0].
2. Pops the FLAGS value off of the stack and into EFLAGS[15:0].
3. Execution begins at the saved CS.IP location.

8.7 Legacy Protected-Mode Interrupt Control Transfers

In protected mode, the interrupt mechanism transfers control to an exception or interrupt handler through gate descriptors. In protected mode, the IDT is a table of 8-byte gate entries, one for each of the 256 possible interrupt vectors implemented by the system. Three gate types are allowed in the IDT:

- Interrupt gates
- Trap gates
- Task gates

If a reference is made to any other descriptor type in the IDT, a general-protection exception (#GP) occurs.

Interrupt-gate control transfers are similar to CALLs and JMPs through call gates. The interrupt mechanism uses gates (interrupt, trap, and task) to establish protected entry-points into the exception and interrupt handlers.

The remainder of this chapter discusses control transfers through interrupt gates and trap gates. If the gate descriptor in the IDT is a task gate, a TSS-segment selector is referenced, and a task switch occurs. See Chapter 12, “Task Management,” for more information on the hardware task-switch mechanism.

[AMD Public Use]

8.7.1 Locating the Interrupt Handler

When an exception or interrupt occurs, the processor scales the interrupt vector number by eight and uses the result as an offset into the IDT. If the gate descriptor referenced by the IDT offset is an interrupt gate or a trap gate, it contains a segment-selector and segment-offset field (see Section “Legacy Segment Descriptors” on page 90 for a detailed description of the gate-descriptor format and fields). These two fields perform the same function as the pointer operand in a far control-transfer instruction. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The gate-descriptor segment-offset field is the instruction-pointer offset into the interrupt-handler code segment. The code-segment base taken from the code-segment descriptor is added to the gate-descriptor segment-offset field to create the interrupt-handler virtual address (linear address).

Figure 8-8 on page 274 shows how the protected-mode interrupt handler is located by the interrupt mechanism.

[AMD Public Use]

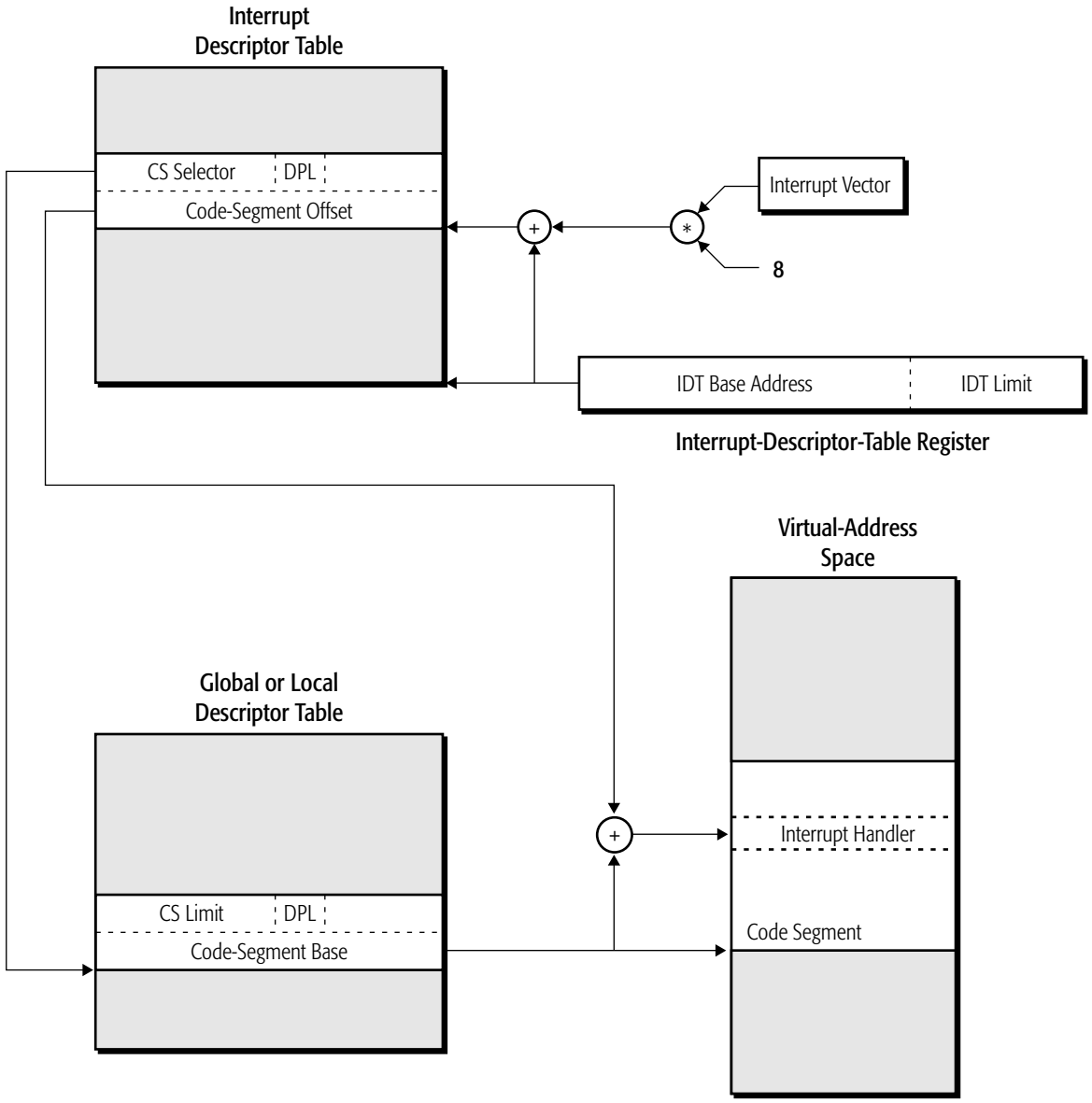


Figure 8-8. Protected-Mode Interrupt Control Transfer

8.7.2 Interrupt To Same Privilege

When a control transfer to an exception or interrupt handler at the same privilege level occurs (through an interrupt gate or a trap gate), the processor performs the following:

1. Pushes the EFLAGS register onto the stack.
2. Clears the TF, NT, RF, and VM bits in EFLAGS to 0.
3. The processor handles EFLAGS.IF based on the gate-descriptor type:

[AMD Public Use]

- If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
- If the gate descriptor is a trap gate, EFLAGS.IF is not modified.

4. Saves the return CS register and EIP register (RIP[31:0]) by pushing them onto the stack. The CS value is padded with two bytes to form a doubleword.
5. If the interrupt has an associated error code, the error code is pushed onto the stack.
6. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.
7. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-9 shows the stack after control is transferred to the interrupt handler.

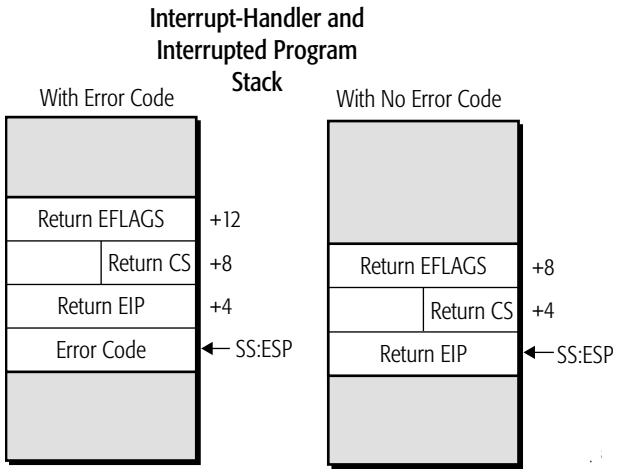


Figure 8-9. Stack After Interrupt to Same Privilege Level

8.7.3 Interrupt To Higher Privilege

When a control transfer to an exception or interrupt handler running at a higher privilege occurs (numerically lower CPL value), the processor performs a stack switch using the following steps:

1. The target CPL is read by the processor from the target code-segment DPL and used as an index into the TSS for selecting the new stack pointer (SS:ESP). For example, if the target CPL is 1, the processor selects the SS:ESP for privilege-level 1 from the TSS.
2. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.
3. Pushes the EFLAGS register onto the new stack.
4. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.
5. The processor handles the EFLAGS.IF bit based on the gate-descriptor type:

[AMD Public Use]

- If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
- If the gate descriptor is a trap gate, EFLAGS.IF is not modified.

6. Saves the return-address pointer (CS:EIP) by pushing it onto the stack. The CS value is padded with two bytes to form a doubleword.
7. If the interrupt vector number has an error code associated with it, the error code is pushed onto the stack.
8. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.
9. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-10 shows the new stack after control is transferred to the interrupt handler.

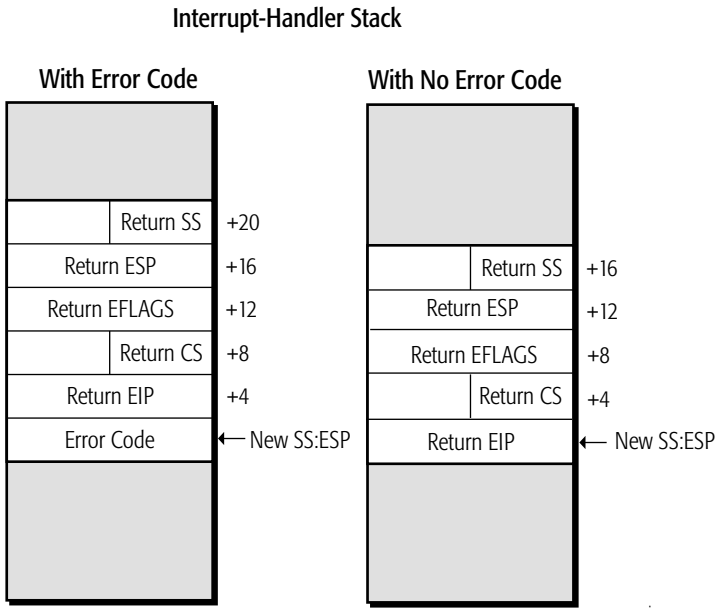


Figure 8-10. Stack After Interrupt to Higher Privilege

8.7.4 Privilege Checks

Before loading the CS register with the interrupt-handler code-segment selector (located in the gate descriptor), the processor performs privilege checks similar to those performed on call gates. The checks are performed when either conforming or nonconforming interrupt handlers are referenced:

1. The processor reads the gate DPL from the interrupt-gate or trap-gate descriptor. The gate DPL is the *minimum privilege-level* (numerically-highest value) needed by a program to access the gate. The processor compares the CPL with the gate DPL. The CPL must be numerically *less-than or equal-to* the gate DPL for this check to pass.

[AMD Public Use]

2. The processor compares the CPL with the interrupt-handler code-segment DPL. For this check to pass, the CPL must be numerically *greater-than or equal-to* the code-segment DPL. This check prevents control transfers to less-privileged interrupt handlers.

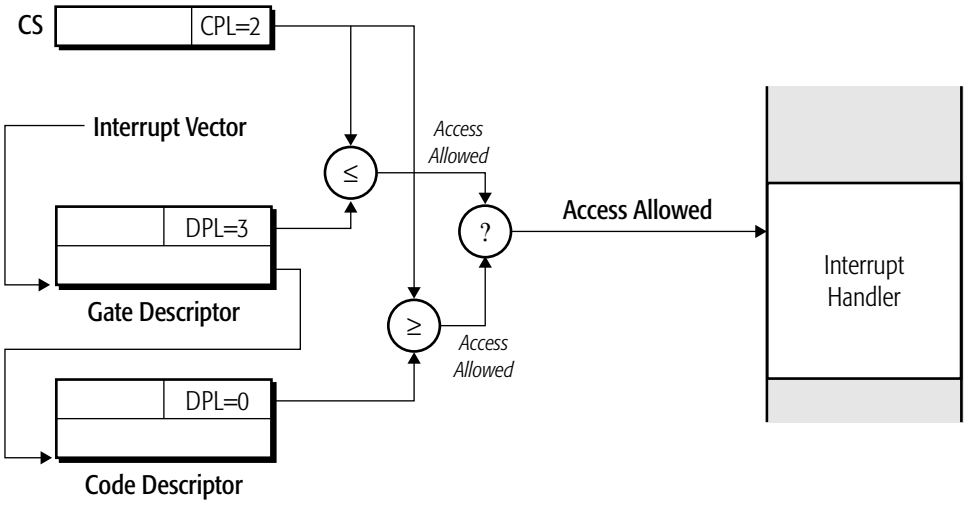
Unlike call gates, no RPL comparison takes place. This is because the gate descriptor is referenced in the IDT using the interrupt vector number rather than a selector, and no RPL field exists in the interrupt vector number.

Exception and interrupt handlers should be made reachable from software running at any privilege level that requires them. If the gate DPL value is too low (requiring more privilege), or the interrupt-handler code-segment DPL is too high (runs at lower privilege), the interrupt control transfer can fail the privilege checks. Setting the gate DPL=3 and interrupt-handler code-segment DPL=0 makes the exception handler or interrupt handler reachable from any privilege level.

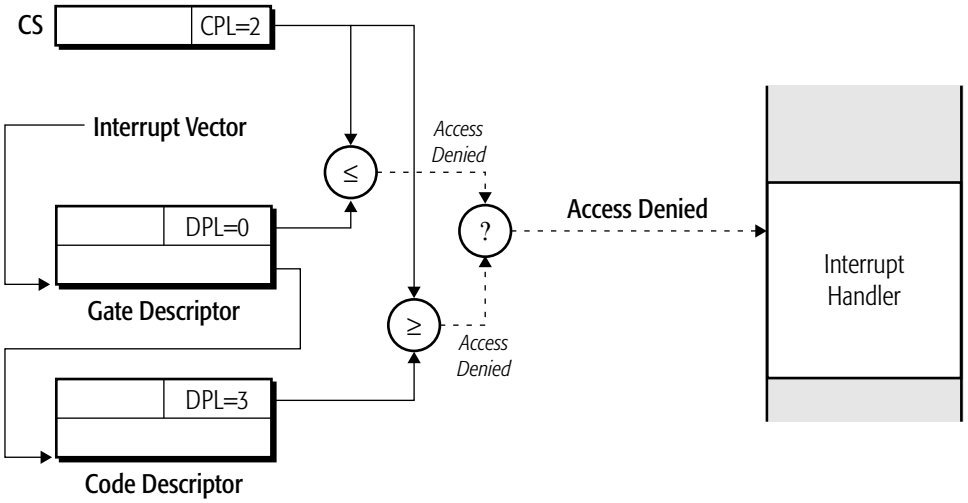
Figure 8-11 on page 278 shows two examples of interrupt privilege checks. In Example 1, both privilege checks pass:

- The interrupt-gate DPL is at the lowest privilege (3), which means that software running at any privilege level (CPL) can access the interrupt gate.
- The interrupt-handler code segment is at the highest-privilege level, as indicated by DPL=0. This means software running at any privilege can enter the interrupt handler through the interrupt gate.

[AMD Public Use]



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

Figure 8-11. Privilege-Check Examples for Interrupts

In Example 2, both privilege checks fail:

- The interrupt-gate DPL specifies that only software running at privilege-level 0 can access the gate. The current program does not have a high enough privilege level to access the interrupt gate, since its CPL is set at 2.

[AMD Public Use]

- The interrupt handler has a lower privilege (DPL=3) than the currently-running software (CPL=2). Transitions from more-privileged software to less-privileged software are not allowed, so this privilege check fails as well.

Although both privilege checks fail, only one such failure is required to deny access to the interrupt handler.

8.7.5 Returning From Interrupt Procedures

A return to an interrupted program should be performed using the IRET instruction. An IRET is a far return to a different code segment, with or without a change in privilege level. The actions of an IRET in both cases are described in the following sections.

IRET, Same Privilege. Before performing the IRET, the stack pointer must point to the return EIP. If there was an error code pushed onto the stack as a result of the exception or interrupt, that error code should have been popped off the stack earlier by the handler. The IRET reverses the actions of the interrupt mechanism:

1. Pops the return pointer off of the stack, loading both the CS register and EIP register (RIP[31:0]) with the saved values. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that an equal-privilege control transfer is occurring.
2. Pops the saved EFLAGS image off of the stack and into the EFLAGS register.
3. Transfers control to the return program at the target CS:EIP.

IRET, Less Privilege. If an IRET changes privilege levels, the return program must be at a lower privilege than the interrupt handler. The IRET in this case causes a stack switch to occur:

1. The return pointer is popped off of the stack, loading both the CS register and EIP register (RIP[31:0]) with the saved values. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that a lower-privilege control transfer is occurring.
2. The saved EFLAGS image is popped off of the stack and loaded into the EFLAGS register.
3. The return-program stack pointer is popped off of the stack, loading both the SS register and ESP register (RSP[31:0]) with the saved values.
4. Control is transferred to the return program at the target CS:EIP.

8.7.6 Shadow Stack Support for Interrupts and Exceptions

The operation of the shadow stack for an interrupt control transfer depends whether or not the interrupt handler runs at the same privilege or at a higher privilege level than the CPL when the interrupt or exception occurred.

Interrupt Control Transfer to Same Privilege Level. When a control transfer to a interrupt handler at the same privilege occurs, and the shadow stack feature is enabled for the current CPL, the processor pushes the interrupted procedure's CS and LIP (CS.base + EIP) onto the current shadow stack.

[AMD Public Use]

Interrupt Control Transfer to Higher Privilege Level. When a control transfer to a interrupt handler at a higher privilege occurs, the actions taken by the processor depend on the CPL when the interrupt or exception occurred.

- If the CPL = 3:
  - if shadow stacks are enabled in user-mode, the current SSP is saved to MSR PL3_SSP.
  - if shadow stacks are enabled in supervisor mode, a new SSP is loaded from MSR PLn_SSP (where n = the target CPL 0, 1 or 2).
  - the shadow stack token located at the base of the new shadow stack is checked, and if valid, the token's busy bit is set to 1.

  Note: the CS and LIP are not pushed onto the new shadow stack.
- If the CPL =1 or 2, and shadow stacks are enabled in supervisor mode:
  - the new SSP is loaded from MSR PLn_SSP (where n = the target CPL 0 or 1).
  - the shadow stack token located at the base of the new shadow stack is checked, and if valid, the token's busy bit is set to 1.
  - the CS, LIP and old SSP are pushed onto the new shadow stack.

For a detailed description of shadow stack operations, see Section “Shadow Stacks” on page 710.

8.8 Virtual-8086 Mode Interrupt Control Transfers

This section describes interrupt control transfers as they relate to virtual-8086 mode. Virtual-8086 mode is not supported by long mode. Therefore, the control-transfer mechanism described here is not applicable to long mode.

When a software interrupt occurs (not external interrupts, INT1, or INT3) while the processor is running in virtual-8086 mode (EFLAGS.VM=1), the control transfer that occurs depends on three system controls:

- *EFLAGS.IOPL*—This field controls interrupt handling based on the CPL. See Section “I/O Privilege Level Field (IOPL) Field” on page 54 for more information on this field.

  Setting IOPL<3 redirects the interrupt to the general-protection exception (#GP) handler.
- *CR4.VME*—This bit enables virtual-mode extensions. See Section “Virtual-8086 Mode Extensions (VME)” on page 48 for more information on this bit.
- *TSS Interrupt-Redirection Bitmap*—The TSS interrupt-redirection bitmap contains 256 bits, one for each possible INT*n* vector (software interrupt). When CR4.VME=1, the bitmap is used by the processor to direct interrupts to the handler provided by the currently-running 8086 program (bitmap entry is 0), or to the protected-mode operating-system interrupt handler (bitmap entry is 1). See Section “Legacy Task-State Segment” on page 374 for information on the location of this field within the TSS.

[AMD Public Use]

If IOPL<3, CR4.VME=1, and the corresponding interrupt redirection bitmap entry is 0, the processor uses the virtual-interrupt mechanism. See Section “Virtual Interrupts” on page 290 for more information on this mechanism.

Table 8-11 summarizes the actions of the above system controls on interrupts taken when the processor is running in virtual-8086 mode.

Table 8-11. Virtual-8086 Mode Interrupt Mechanisms

EFLAGS.IOPL	CR4.VME	TSS Interrupt Redirection Bitmap Entry	Interrupt Mechanism
0, 1, or 2	0	—	General-Protection Exception
	1	1	
	1	0	Virtual Interrupt
3	0	—	Protected-Mode Handler
	1	1	
	1	0	Virtual 8086 Handler

8.8.1 Protected-Mode Handler Control Transfer

Control transfers to protected-mode handlers from virtual-8086 mode differ from standard protected-mode transfers in several ways. The processor follows these steps in making the control transfer:

1. Reads the CPL=0 stack pointer (SS:ESP) from the TSS.
2. Pushes the GS, FS, DS, and ES selector registers onto the stack. Each push is padded with two bytes to form a doubleword.
3. Clears the GS, FS, DS, and ES selector registers to 0. This places a null selector in each of the four registers
4. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.
5. Pushes the EFLAGS register onto the new stack.
6. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.
7. Handles EFLAGS.IF based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.
8. Pushes the return-address pointer (CS:EIP) onto the stack. The CS value is padded with two bytes to form a doubleword.
9. If the interrupt has an associated error code, pushes the error code onto the stack.
10. Loads the segment-selector field from the gate descriptor into the CS register, and loads the offset field from the gate descriptor into the EIP register.

[AMD Public Use]

11. Begins execution of the interrupt handler with the instruction referenced by the new CS:EIP.

Figure 8-12 shows the new stack after control is transferred to the interrupt handler with an error code.

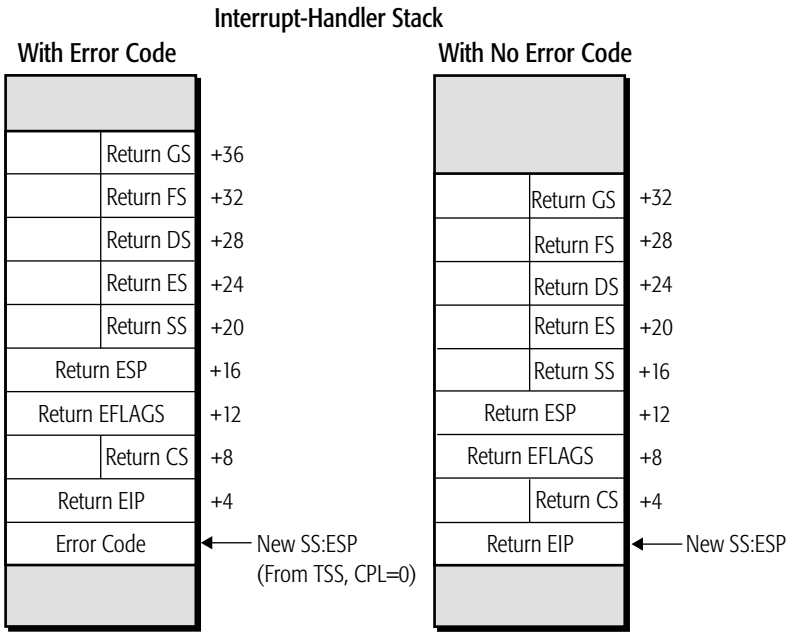


Figure 8-12. Stack After Virtual-8086 Mode Interrupt to Protected Mode

An IRET from privileged protected-mode software (CPL=0) to virtual-8086 mode reverses the stack-build process. After the return pointer, EFLAGS, and return stack-pointer are restored, the processor restores the ES, DS, FS, and GS registers by popping their values off the stack.

8.8.2 Virtual-8086 Handler Control Transfer

When a control transfer to an 8086 handler occurs from virtual-8086 mode, the processor creates an interrupt-handler stack identical to that created when an interrupt occurs in real mode (see Figure 8-7 on page 272). The processor performs the following actions during a control transfer:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears the EFLAGS.IF and EFLAGS.RF bits to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the 8086 IDT by scaling the interrupt vector by four and adding the result to the virtual (linear) address 0. The value in the IDTR is not used.
5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

An IRET from the 8086 handler back to virtual-8086 mode reverses the stack-build process.

[AMD Public Use]

8.9 Long-Mode Interrupt Control Transfers

The long-mode architecture expands the legacy interrupt-mechanism to support 64-bit operating systems and applications. These changes include:

- All interrupt handlers are 64-bit code and operate in 64-bit mode.
- The size of an interrupt-stack push is fixed at 64 bits (8 bytes).
- The interrupt-stack frame is aligned on a 16-byte boundary.
- The stack pointer, SS:RSP, is pushed unconditionally on interrupts, rather than conditionally based on a change in CPL.
- The SS selector register is loaded with a null selector as a result of an interrupt, if the CPL changes.
- The IRET instruction behavior changes, to unconditionally pop SS:RSP, allowing a null SS to be popped.
- A new interrupt stack-switch mechanism, called the interrupt-stack table or IST, is introduced.
- When shadow stacks are enabled, a new shadow stack-switch mechanism, called the Interrupt Shadow Stack Table or ISST, is introduced.

8.9.1 Interrupt Gates and Trap Gates

Only long-mode interrupt and trap gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy 32-bit interrupt-gate and 32-bit trap-gate types (0Eh and 0Fh, as described in Section “System Descriptors” on page 101) are redefined in long mode as 64-bit interrupt-gate and 64-bit trap-gate types. 32-bit and 16-bit interrupt-gate and trap-gate types do not exist in long mode, and software is prohibited from using task gates. If a reference is made to any gate other than a 64-bit interrupt gate or a 64-bit trap gate, a general-protection exception (#GP) occurs.

The long-mode gate types are 16 bytes (128 bits) long. They are an extension of the legacy-mode gate types, allowing a full 64-bit segment offset to be stored in the descriptor. See Section “Legacy Segment Descriptors” on page 90 for a detailed description of the gate-descriptor format and fields.

8.9.2 Locating the Interrupt Handler

When an interrupt occurs in long mode, the processor multiplies the interrupt vector number by 16 and uses the result as an offset into the IDT. The gate descriptor referenced by the IDT offset contains a segment-selector and a 64-bit segment-offset field. The gate-descriptor segment-offset field contains the complete virtual address for the interrupt handler. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The code-segment descriptor is only used for privilege-checking purposes and for placing the processor in 64-bit mode. The code segment-descriptor base field, limit field, and most attributes are ignored.

Figure 8-13 shows how the long-mode interrupt handler is located by the interrupt mechanism.

[AMD Public Use]

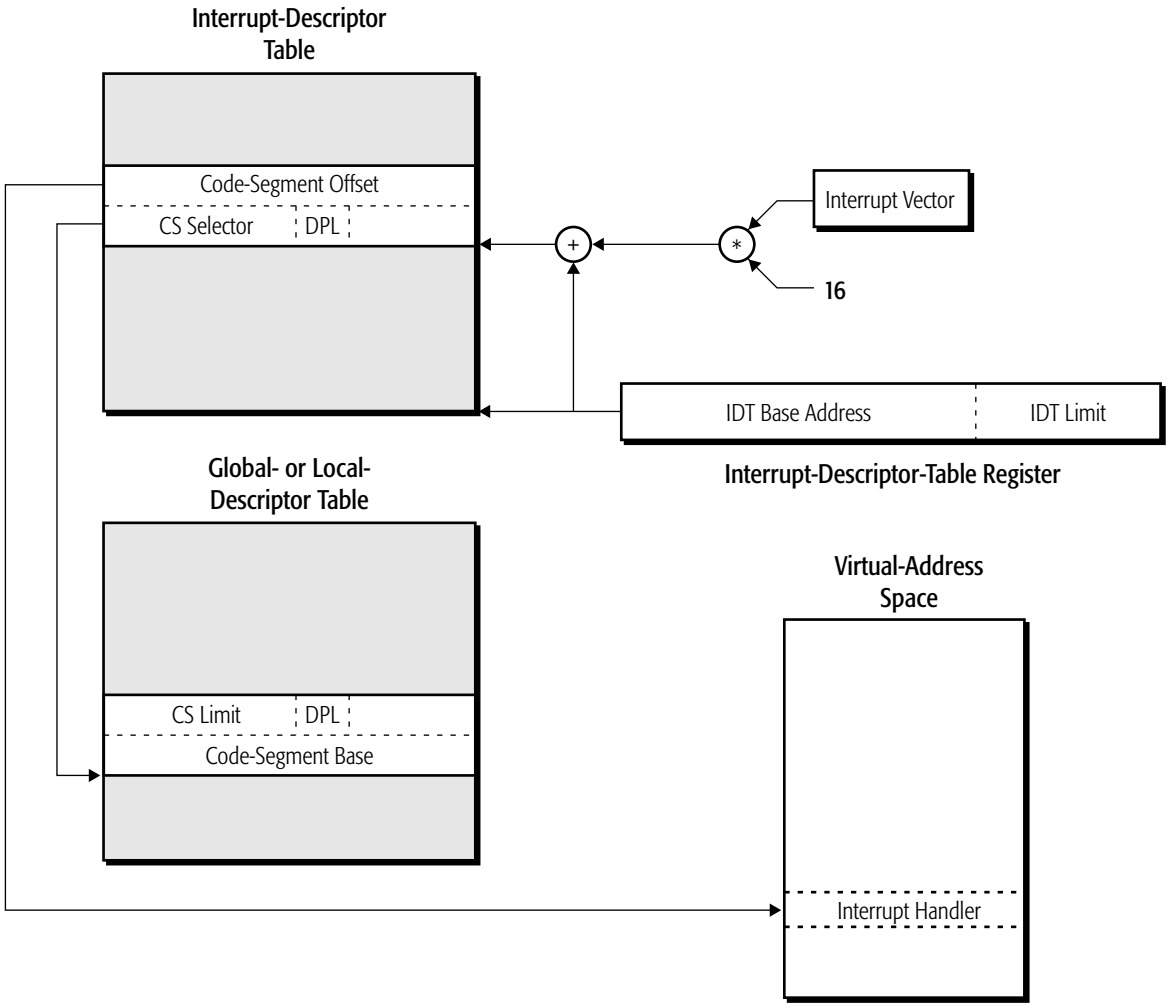


Figure 8-13. Long-Mode Interrupt Control Transfer

8.9.3 Interrupt Stack Frame

In long mode, the return-program stack pointer (SS:RSP) is always pushed onto the interrupt-handler stack, regardless of whether or not a privilege change occurs. Although the SS register is not used in 64-bit mode, SS is pushed to allow returns into compatibility mode. Pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stack-frame size for all interrupts, except for error codes. Interrupt service-routine entry points that handle interrupts generated by non-error-code interrupts can push an error code on the stack for consistency.

In long mode, when a control transfer to an interrupt handler occurs, the processor performs the following:

1. Aligns the new interrupt-stack frame by masking RSP with FFFF_FFFF_FFFF_FFF0h.

[AMD Public Use]

2. If IST field in interrupt gate is not 0, reads IST pointer into RSP.
3. If a privilege change occurs, the target DPL is used as an index into the long-mode TSS to select a new stack pointer (RSP).
4. If a privilege change occurs, SS is cleared to zero indicating a null selector.
5. Pushes the return stack pointer (old SS:RSP) onto the new stack. The SS value is padded with six bytes to form a quadword.
6. Pushes the 64-bit RFLAGS register onto the stack. The upper 32 bits of the RFLAGS image on the stack are written as zeros.
7. Clears the TF, NT, and RF bits in RFLAGS bits to 0.
8. Handles the RFLAGS.IF bit according to the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, RFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, RFLAGS.IF is not modified.
9. Pushes the return CS register and RIP register onto the stack. The CS value is padded with six bytes to form a quadword.
10. If the interrupt vector number has an error code associated with it, pushes the error code onto the stack. The error code is padded with four bytes to form a quadword.
11. Loads the segment-selector field from the gate descriptor into the CS register. The processor checks that the target code-segment is a 64-bit mode code segment.
12. Loads the offset field from the gate descriptor into the target RIP. The interrupt handler begins execution when control is transferred to the instruction referenced by the new RIP.

Figure 8-14 on page 286 shows the stack after control is transferred to the interrupt handler.

[AMD Public Use]

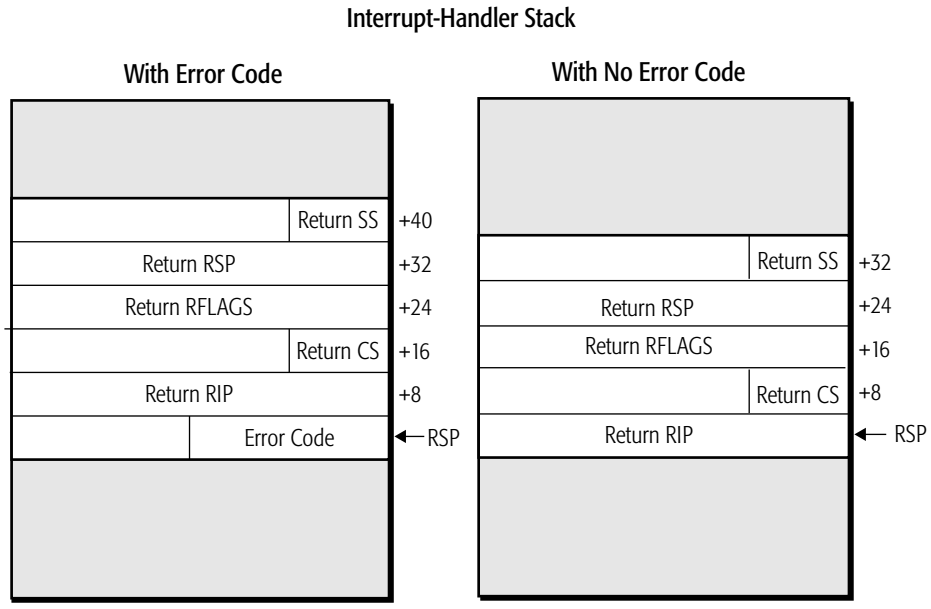


Figure 8-14. Long-Mode Stack After Interrupt—Same Privilege

Interrupt-Stack Alignment. In legacy mode, the interrupt-stack pointer can be aligned at any address boundary. Long mode, however, aligns the stack on a 16-byte boundary. This alignment is performed by the processor in hardware before pushing items onto the stack frame. The previous RSP is saved unconditionally on the new stack by the interrupt mechanism. A subsequent IRET instruction automatically restores the previous RSP.

Aligning the stack on a 16-byte boundary allows optimal performance for saving and restoring the 16-byte XMM registers. The interrupt handler can save and restore the XMM registers using the faster 16-byte aligned loads and stores (MOVAPS), rather than unaligned loads and stores (MOVUPS). Although the RSP alignment is always performed in long mode, it is only of consequence when the interrupted program is already running at CPL=0, and it is generally used only within the operating-system kernel. The operating system should put 16-byte aligned RSP values in the TSS for interrupts that change privilege levels.

Stack Switch. In long mode, the stack-switch mechanism differs slightly from the legacy stack-switch mechanism (see Section “Interrupt To Higher Privilege” on page 275). When stacks are switched during a long-mode privilege-level change resulting from an interrupt, a new SS descriptor is *not* loaded from the TSS. Long mode only loads an inner-level RSP from the TSS. However, the SS selector is loaded with a null selector, allowing nested control transfers, including interrupts, to be handled properly in 64-bit mode. The SS.RPL is set to the new CPL value. See Section “Nested IRETs to 64-Bit Mode Procedures” on page 289 for additional information.

[AMD Public Use]

The interrupt-handler stack that results from a privilege change in long mode looks identical to a long-mode stack when no privilege change occurs. Figure 8-15 shows the stack after the switch is performed and control is transferred to the interrupt handler.

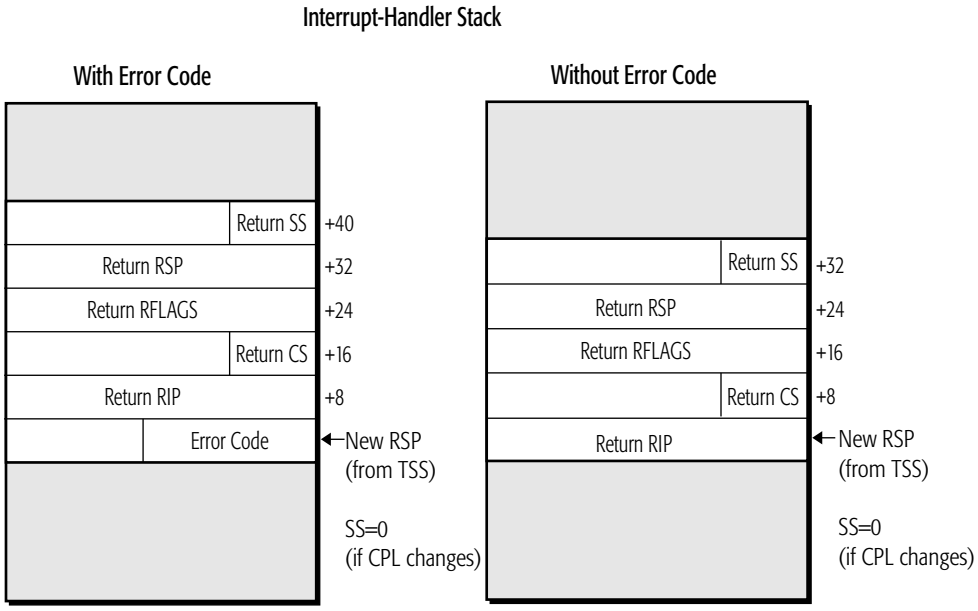


Figure 8-15. Long-Mode Stack After Interrupt—Higher Privilege

8.9.4 Interrupt-Stack Table

In long mode, a new interrupt-stack table (IST) mechanism is introduced as an alternative to the modified legacy stack-switch mechanism described above. The IST mechanism provides a method for specific interrupts, such as NMI, double-fault, and machine-check, to always execute on a known-good stack. In legacy mode, interrupts can use the hardware task-switch mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the hardware task-switch mechanism is not supported in long mode.

When enabled, the IST mechanism unconditionally switches stacks. It can be enabled on an individual interrupt vector basis using a new field in the IDT gate-descriptor entry. This allows some interrupts to use the modified legacy mechanism, and others to use the IST mechanism. The IST mechanism is only available in long mode.

The IST mechanism uses new fields in the 64-bit TSS format and the long-mode interrupt-gate and trap-gate descriptors:

- Figure 12-8 on page 380 shows the format of the 64-bit TSS and the location of the seven IST pointers. The 64-bit TSS offsets from 24h to 5Bh provide space for seven IST pointers, each of which are 64 bits (8 bytes) long.

[AMD Public Use]

- The long-mode interrupt-gate and trap-gate descriptors define a 3-bit IST-index field in bits 2:0 of byte +4. Figure 4-24 on page 104 shows the format of long-mode interrupt-gate and trap-gate descriptors and the location of the IST-index field.

To enable the IST mechanism for a specific interrupt, system software stores a non-zero value in the interrupt gate-descriptor IST-index field. If the IST index is zero, the modified legacy stack-switching mechanism (described in the previous section) is used.

Figure 8-16 shows how the IST mechanism is used to create the interrupt-handler stack. When an interrupt occurs and the IST index is non-zero, the processor uses the index to select the corresponding IST pointer from the TSS. The IST pointer is loaded into the RSP to establish a new stack for the interrupt handler. The SS register is loaded with a null selector if the CPL changes and the SS.RPL is set to the new CPL value. After the stack is loaded, the processor pushes the old stack pointer, RFLAGS, the return pointer, and the error code (if applicable) onto the stack. Control is then transferred to the interrupt handler.

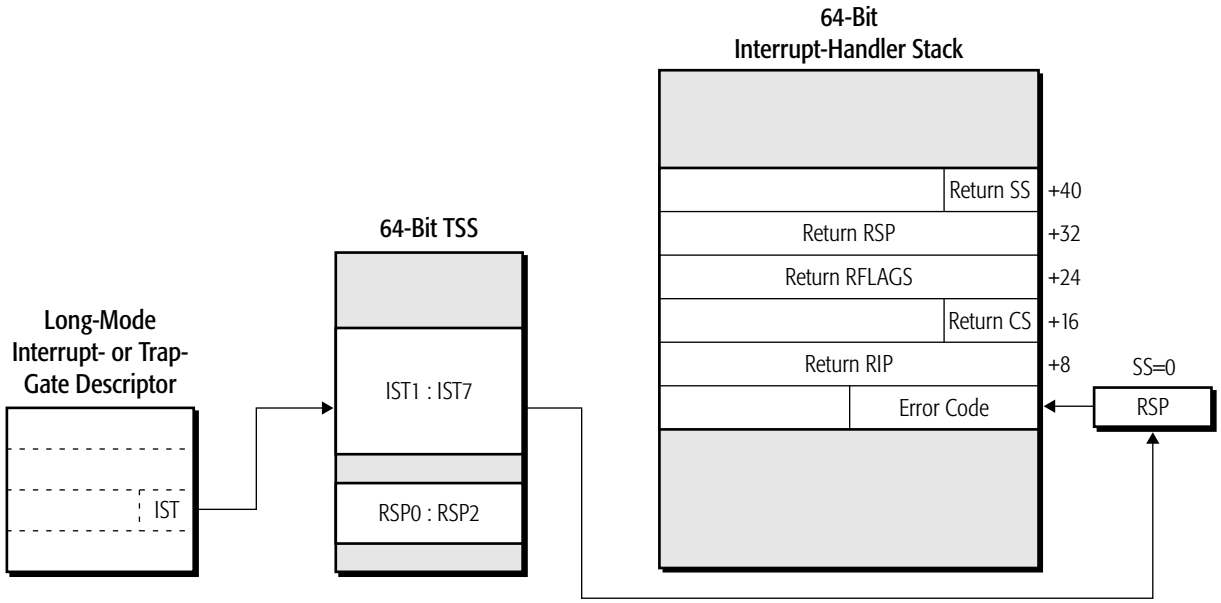


Figure 8-16. Long-Mode IST Mechanism

Software must make sure that an interrupt or exception handler using an IST pointer doesn't take another exception using the same IST pointer, as this will result in the first stack exception frame being overwritten.

[AMD Public Use]

Interrupt Shadow Stack Table. When the shadow stack feature is enabled in long mode (EFER.LMA=1), a mechanism similar to the IST is provided to switch shadow stacks. This mechanism, the Interrupt Shadow Stack Table (ISST) is described in Section “Shadow Stacks” on page 710.

8.9.5 Returning From Interrupt Procedures

As with legacy mode, a return to an interrupted program in long mode should be performed using the IRET instruction. However, in long mode, the IRET semantics are different from legacy mode:

- *In 64-bit mode*, IRET pops the return-stack pointer unconditionally off the interrupt-stack frame and into the SS:RSP registers. This reverses the action of the long-mode interrupt mechanism, which saves the stack pointer whether or not a privilege change occurs. IRET also allows a null selector to be popped off the stack and into the SS register. See Section “Nested IRETs to 64-Bit Mode Procedures” on page 289 for additional information.
- *In compatibility mode*, IRET behaves as it does in legacy mode. The SS:ESP is popped off the stack only if a control transfer to less privilege (numerically greater CPL) is performed. Otherwise, it is assumed that a stack pointer is not present on the interrupt-handler stack.

The long-mode interrupt mechanism always uses a 64-bit stack when saving values for the interrupt handler, and the interrupt handler is always entered in 64-bit mode. To work properly, an IRET used to exit the 64-bit mode interrupt-handler requires a series of eight-byte pops off the stack. This is accomplished by using a 64-bit operand-size prefix with the IRET instruction. The default stack size assumed by an IRET in 64-bit mode is 32 bits, so a 64-bit REX prefix is needed by 64-bit mode interrupt handlers.

Nested IRETs to 64-Bit Mode Procedures. In long mode, an interrupt causes a null selector to be loaded into the SS register if the CPL changes (this is the same action taken by a far CALL in long mode). If the interrupt handler performs a far call, or is itself interrupted, the null SS selector is pushed onto the stack frame, and another null selector is loaded into the SS register. Using a null selector in this way allows the processor to properly handle returns nested within 64-bit-mode procedures and interrupt handlers.

The null selector enables the processor to properly handle nested returns to 64-bit mode (which do not use the SS register), and returns to compatibility mode (which do use the SS register). Normally, an IRET that pops a null selector into the SS register causes a general-protection exception (#GP) to occur. However, in long mode, the null selector indicates the existence of nested interrupt handlers and/or privileged software in 64-bit mode. Long mode allows an IRET to pop a null selector into SS from the stack under the following conditions:

- The target mode is 64-bit mode.
- The target CPL<3.

In this case, the processor does not load an SS descriptor, and the null selector is loaded into SS without causing a #GP exception.

[AMD Public Use]

8.10 Virtual Interrupts

The term *virtual interrupts* includes two classes of extensions to the interrupt-handling mechanism:

- *Virtual-8086 Mode Extensions (VME)*—These allow virtual interrupts and interrupt redirection in virtual-8086 mode. VME has no effect on protected-mode programs.
- *Protected-Mode Virtual Interrupts (PVI)*—These allow virtual interrupts in protected mode when CPL=3. Interrupt redirection is not available in protected mode. PVI has no effect on virtual-8086-mode programs.

Because virtual-8086 mode is not supported in long mode, VME extensions are not supported in long mode. PVI extensions are, however, supported in long mode.

8.10.1 Virtual-8086 Mode Extensions

The virtual-8086-mode extensions (VME) enable performance enhancements for 8086 programs running as protected tasks in virtual-8086 mode. These extensions are enabled by setting CR4.VME (bit 0) to 1. The extensions enabled by CR4.VME are:

- Virtualizing control and notification of maskable external interrupts with the EFLAGS VIF (bit 19) and VIP (bit 20) bits.
- Selective interception of software interrupts (INT*n* instructions) using the TSS interrupt redirection bitmap (IRB).

Background. Legacy-8086 programs expect to have full access to the EFLAGS interrupt flag (IF) bit, allowing programs to enable and disable maskable external interrupts. When those programs run in virtual-8086 mode under a multitasking protected-mode environment, it can disrupt the operating system if programs enable or disable interrupts for their own purposes. This is particularly true if interrupts associated with one program can occur during execution of another program. For example, a program could request that an area of memory be copied to disk. System software could suspend the program before external hardware uses an interrupt to acknowledge that the block has been copied. System software could subsequently start a second program which enables interrupts. This second program could receive the external interrupt indicating that the memory block of the first program has been copied. If that were to happen, the second program would probably be unprepared to handle the interrupt properly.

Access to the IF bit must be managed by system software on a task-by-task basis to prevent corruption of system resources. In order to completely manage the IF bit, system software must be able to interrupt all instructions that can read or write the bit. These instructions include STI, CLI, PUSHF, POPF, INT*n*, and IRET. These instructions are part of an instruction class that is *IOPL-sensitive*. The processor takes a general-protection exception (#GP) whenever an IOPL-sensitive instruction is executed and the EFLAGS.IOPL field is less than the CPL. Because all virtual-8086 programs run at CPL=3, system software can interrupt all instructions that modify the IF bit by setting IOPL<3.

System software maintains a virtual image of the IF bit for each virtual-8086 program by emulating the actions of IOPL-sensitive instructions that modify the IF bit. When an external maskable-interrupt

[AMD Public Use]

occurs, system software checks the state of the IF image for the current virtual-8086 program to determine whether the program is masking interrupts. If the program is masking interrupts, system software saves the interrupt information until the virtual-8086 program attempts to re-enable interrupts. When the virtual-8086 program unmasks interrupts with an IOPL-sensitive instruction, system software traps the action with the #GP handler.

The performance of a processor can be significantly degraded by the overhead of trapping and emulating IOPL-sensitive instructions, and the overhead of maintaining images of the IF bit for each virtual-8086 program. This performance loss can be eliminated by running virtual-8086 programs with IOPL set to 3, thus allowing changes to the real IF flag from any privilege level. Unfortunately, this can leave critical system resources unprotected.

In addition to the performance problems caused by virtualizing the IF bit, software interrupts (INT*n* instructions) cannot be masked by the IF bit or virtual copies of the IF bit. The IF bit only affects maskable external interrupts. Software interrupts in virtual-8086 mode are normally directed to the real mode interrupt vector table (IVT), but it can be desirable to redirect certain interrupts to the protected-mode interrupt-descriptor table (IDT).

The virtual-8086-mode extensions are designed to support both external interrupts and software interrupts, with mechanisms that preserve high performance without compromising protection. Virtualization of external interrupts is supported using two bits in the EFLAGS register: the virtual-interrupt flag (VIF) bit and the virtual-interrupt pending (VIP) bit. Redirection of software interrupts is supported using the interrupt-redirection bitmap (IRB) in the TSS. A separate TSS can be created for each virtual-8086 program, allowing system software to control interrupt redirection independently for each virtual-8086 program.

VIF and VIP Extensions for External Interrupts. When VME extensions are enabled, the IF-modifying instructions normally trapped by system software are allowed to execute. However, instead of modifying the IF bit, they modify the EFLAGS VIF bit. This leaves control over maskable interrupts to the system software. It can also be used as an indicator to system software that the virtual-8086 program is able to, or is expecting to, receive external interrupts.

When an unmasked external interrupt occurs, the processor transfers control from the virtual-8086 program to a protected-mode interrupt handler. If the interrupt handler determines that the interrupt is for the virtual-8086 program, it can check the state of the VIF bit in the EFLAGS value pushed on the stack for the virtual-8086 program. If the VIF bit is set (indicating the virtual-8086 program attempted to unmask interrupts), system software can allow the interrupt to be handled by the appropriate virtual-8086 interrupt handler.

If the VIF bit is clear (indicating the virtual-8086 program attempted to mask interrupts) and the interrupt is for the virtual-8086 program, system software can hold the interrupt pending. System software holds an interrupt pending by saving appropriate information about the interrupt, such as the interrupt vector, and setting the virtual-8086 program's VIP bit in the EFLAGS image on the stack. When the virtual-8086 program later attempts to set IF, the previously set VIP bit causes a general-protection exception (#GP) to occur. System software can then pass the saved interrupt information to the virtual-8086 interrupt handler.

[AMD Public Use]

To summarize, when the VME extensions are enabled (CR4.VME=1), the VIF and VIP bits are set and cleared as follows:

- *VIF Bit*—This bit is set and cleared by the processor in virtual-8086 mode in response to an attempt by a virtual-8086 program to set and clear the EFLAGS.IF bit. VIF is used by system software to determine whether a maskable external interrupt should be passed on to the virtual-8086 program, emulated by system software, or held pending. VIF is also cleared during software interrupts through interrupt gates, with the original VIF value preserved in the EFLAGS image on the stack.
- *VIP Bit*—System software sets and clears this bit in the EFLAGS image saved on the stack after an interrupt. It can be set when an interrupt occurs for a virtual-8086 program that has a clear VIF bit. The processor examines the VIP bit when an attempt is made by the virtual-8086 program to set the IF bit. If VIP is set when the program attempts to set IF, a general-protection exception (#GP) occurs *before* execution of the IF-setting instruction. System software must clear VIP to avoid repeated #GP exceptions when returning to the interrupted instruction.

The VIF and VIP bits can be used by system software to minimize the overhead associated with managing maskable external interrupts because virtual copies of the IF flag do not have to be maintained by system software. Instead, VIF and VIP are maintained during context switches along with the remaining EFLAGS bits.

Table 8-12 on page 294 shows how the behavior of instructions that modify the IF bit are affected by the VME extensions.

Interrupt Redirection of Software Interrupts. In virtual-8086 mode, software interrupts (INT*n* instructions) are trapped using a #GP exception handler if the IOPL is less than 3 (the CPL for virtual-8086 mode). This allows system software to interrupt and emulate 8086-interrupt handlers. System software can set the IOPL to 3, in which case the INT*n* instruction is vectored through a gate descriptor in the protected-mode IDT. System software can use the gate to control access to the virtual-8086 mode interrupt vector table (IVT), or to redirect the interrupt to a protected-mode interrupt handler.

When VME extensions are enabled, for INT*n* instructions to execute normally, vectoring directly to a virtual-8086 interrupt handler through the virtual-8086 IVT (located at address 0 in the virtual-address space of the task). For security or performance reasons, however, it can be necessary to intercept INT*n* instructions on a vector-specific basis to allow servicing by protected-mode interrupt handlers. This is performed by using the interrupt-redirection bitmap (IRB), located in the TSS and enabled when CR4.VME=1. The IRB is available only in virtual-8086 mode.

Figure 12-6 on page 375 shows the format of the TSS, with the interrupt redirection bitmap located near the top. The IRB contains 256 bits, one for each possible software-interrupt vector. The most-significant bit of the IRB controls interrupt vector 255, and is located immediately before the IOPB base. The least-significant bit of the IRB controls interrupt vector 0.

The bits in the IRB function as follows:

[AMD Public Use]

- When set to 1, the INT*n* instruction behaves as if the VME extensions are not enabled. The interrupt is directed through the IDT to a protected-mode interrupt handler if IOPL=3. If IOPL<3, the INT*n* causes a #GP exception.
- When cleared to 0, the INT*n* instruction is directed through the IVT for the virtual-8086 program to the corresponding virtual-8086 interrupt handler.

Only software interrupts can be redirected using the IRB mechanism. External interrupts are asynchronous events that occur outside the context of a virtual-8086 program. Therefore, external interrupts require system-software intervention to determine the appropriate context for the interrupt. The VME extensions described in Section “VIF and VIP Extensions for External Interrupts” on page 291 are provided to assist system software with external-interrupt intervention.

8.10.2 Protected Mode Virtual Interrupts

The protected-mode virtual-interrupt (PVI) bit in CR4 enables support for interrupt virtualization in protected mode. When enabled, the processor maintains program-specific VIF and VIP bits similar to the manner defined by the virtual-8086 mode extensions (VME). However, unlike VME, only the STI and CLI instructions are affected by the PVI extension. When a program is running at CPL=3, it can use STI and CLI to set and clear its copy of the VIF flag without causing a general-protection exception. The last section of Table 8-12 on page 294 describes the behavior of instructions that modify the IF bit when PVI extensions are enabled.

The interrupt redirection bitmap (IRB) defined by the VME extensions is not supported by the PVI extensions.

8.10.3 Effect of Instructions that Modify EFLAGS.IF

Table 8-12 on page 294 shows how the behavior of instructions that modify the IF bit are affected by the VME and PVI extensions. The table columns specify the following:

- *Operating Mode*—the processor mode in effect when the instruction is executed.
- *Instruction*—the IF-modifying instruction.
- *IOPL*—the value of the EFLAGS.IOPL field.
- *VIP*—the value of the EFLAGS.VIP bit.
- *#GP*—indicates whether the conditions in the first four columns cause a general-protection exception (#GP) to occur.
- *Effect on IF Bit*—indicates the effect the conditions in the first four columns have on the EFLAGS.IF bit and the image of EFLAGS.IF on the stack.
- *Effect on VIF Bit*—indicates the effect the conditions in the first four columns have on the EFLAGS.VIF bit and the image of EFLAGS.VIF on the stack.

[AMD Public Use]

Table 8-12. Effect of Instructions that Modify the IF Bit

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Real Mode *CR0.PE=0* *EFLAGS.VM=0* *CR4.VME=0* *CR4.PVI=0*	CLI			no	IF = 0	
	STI				IF = 1	
	PUSHF				EFLAGS.IF Stack Image = IF	
	POPF				IF = EFLAGS.IF stack image	
	INT*n*				EFLAGS.IF Stack Image = IF IF = 0	
	IRET				IF = EFLAGS.IF Stack Image	
Protected Mode *CR0.PE=1* *EFLAGS.VM=0* *CR4.VME=x* *CR4.PVI=0*	CLI	≥CPL		no	IF = 0	
		<CPL		yes	—	
	STI	≥CPL		no	IF = 1	
		<CPL		yes	—	
	PUSHF	x		no	EFLAGS.IF Stack Image = IF	
	POPF	≥CPL			IF = EFLAGS.IF Stack Image	
		<CPL			No Change	
	INT*n* gate	x			EFLAGS.IF Stack Image = IF IF = 0	
	IRET				IF = EFLAGS.IF Stack Image	
	IRETD					

Note:
Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:
1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

[AMD Public Use]

Table 8-12. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Virtual-8086 Mode *CR0.PE=1* *EFLAGS.VM=1* *CR4.VME=0* *CR4.PVI=x*	CLI	3		no	IF = 0	
		< 3		yes	—	
	STI	3		no	IF = 1	
		< 3		yes	—	
	PUSHF	3		no	EFLAGS.IF Stack Image = IF	
		< 3		yes	—	
	POPF	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	—	
	INT*n* gate	3		no	EFLAGS.IF Stack Image = IF IF = 0	
		< 3		yes	—	
	IRET	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	—	
	IRETD	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	—	

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

[AMD Public Use]

Table 8-12. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Virtual-8086 Mode *with VME Extensions* *CR0.PE=1* *EFLAGS.VM=1* *CR4.VME=1* *CR4.PVI=x*	CLI	3	x	no	IF = 0	No Change
		<3			No Change	VIF = 0
	STI	3	x	no	IF = 1	No Change
		<3	0	no	No Change	VIF = 1
			1	yes	—	
	PUSHF	3	x	no	EFLAGS.IF Stack Image = IF	Not Pushed
		<3			Not Pushed	EFLAGS.IF Stack Image = VIF
	PUSHFD	3	x	no	EFLAGS.IF Stack Image = IF	EFLAGS.VIF Stack Image = VIF
		<3		yes	—	
	POPF	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3	0	no	No Change	VIF = EFLAGS.IF Stack Image
			1	yes	—	
	POPFD	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3		yes	—	
	INT*n* gate	3	x	no	EFLAGS.IF Stack Image = IF IF = 0	No Change
		<3			No Change	EFLAGS.IF Stack Image = VIF VIF = 0
	IRET	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3	0	no	No Change	VIF = EFLAGS.IF Stack Image
			1	no[1]	No Change	VIF = EFLAGS.IF Stack Image
				yes[2]	—	
	IRETD	3	x	no	IF = EFLAGS.IF Stack Image	VIF = EFLAGS.IF Stack Image
		<3		yes	—	

Note:
Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:
1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

[AMD Public Use]

Table 8-12. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Protected Mode *with PVI Extensions* *CR0.PE=1* *EFLAGS.VM=0* *CR4.VME=x* *CR4.PVI=1* *CPL=3*	CLI	3	x	no	IF = 0	No Change
		<3			No Change	VIF = 0
	STI	3	x	no	IF = 1	No Change
		<3	0	no	No Change	VIF = 1
			1	yes	—	
	PUSHF	x	x	no	EFLAGS.IF Stack Image = IF	Not Pushed
	PUSHFD					EFLAGS.VIF Stack Image = VIF
	POPF				IF = EFLAGS.IF Stack Image	No Change
	POPFD					VIF = 0
	INT*n* gate				EFLAGS.IF Stack Image = IF IF = 0 (if interrupt gate)	No Change
	IRET				IF = EFLAGS.IF Stack Image	No Change
	IRETD					VIF = EFLAGS.VIF Stack Image

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

[AMD Public Use]

9 Machine Check Architecture

The AMD64 Machine Check Architecture (MCA) plays a vital role in the reliability, availability, and serviceability (RAS) of AMD processors, as well as the RAS of the computer systems in which they are embedded. MCA defines the facilities by which processor and system hardware errors are logged and reported to system software. This allows system software to serve a strategic role in recovery from and diagnosis of hardware errors.

Error checking hardware is configured and information about detected error conditions is conveyed via an architecturally-defined set of registers. The system programming interface of MCA is described below in Section 9.3 “Machine Check Architecture MSRs” on page 303.

9.1 Introduction

All computer systems are susceptible to errors—results that are contrary to the system design. Errors can be categorized as *soft* or *hard*. Soft errors are caused by transient interference and are not necessarily indicative of any damage to the computer circuitry. These external events include noise from electromagnetic radiation and the incursion of sub-atomic particles that cause bit cell storage capacitors to change state.

Hard errors are repeatable malfunctions that are generally attributable to physical damage to computer circuitry. Damage may be caused by external forces (for example, voltage surges) or wear processes inherent in the circuit technology. Damaged circuit elements can manifest symptoms similar to those that are caused by soft error processes. An increase in the frequency of errors attributable to one circuit element may indicate that the element has sustained damage or is wearing-out and may, in the future, cause a hard error.

9.1.1 Reliability, Availability, and Serviceability

This section describes the concepts of reliability, availability, and serviceability (RAS) and shows how they are interrelated.

The rate at which errors occur in a computer system is a measure of the system’s *reliability*. *Availability* is the percentage of time that the system is available to do useful work. Errors that prevent a computer system from continued operation result in *down-time*, that is, periods of unavailability. Down-time includes the amount of time required to restore the system to operation. This may include the time to diagnose a failure, determine the field replaceable unit (FRU) containing the faulty circuitry, carry out the repair action required to replace the identified FRU, and restart the system. This time directly impacts the system’s availability and is a measure of the system’s *serviceability*.

The availability of a computer system can be increased without decreasing performance or significantly increasing cost through the judicious addition of data and control path redundancy in concert with dedicated error-checking hardware. Together, redundancy and error checking *detect* and

[AMD Public Use]

often *correct* hardware errors. When errors are corrected by hardware, system operation continues without any perceptible disruption or loss in performance.

Another important technique that can prevent down-time is *error containment*. Error containment limits the propagation of an erroneous data. This enhances system availability by limiting the effects of errors to a subset of software or hardware resources. System software may either correct the error and resume the interrupted program or, if the error cannot be corrected, terminate software processes that cannot continue due to the error.

Error logging enhances serviceability by providing information that is used to identify the FRU that contains the failed circuitry. The mechanical design of the computer system can enhance serviceability (and thus availability) by making the task of physically replacing a failed FRU quicker and easier.

9.1.2 Error Detection, Logging, and Reporting

Error detection requires specific error-checking hardware that compares the actual result of some data transfer or transformation to the expected result. Any disparity indicates that an error has occurred. Error detection is controlled through implementation-specific means. Disabling detection is normally only appropriate when hardware is being debugged in the laboratory.

When an error is detected, hardware autonomously acts to either correct the error or contain the propagation of the corrupting effects of an uncorrected error. For some error sources, hardware action can be disabled by software through the MCA interface.

As hardware acts to correct or contain a detected error, it gathers information about the error to aid in recovery, diagnosis, and repair. The architecture provides software control of error *logging* and *reporting*. The following describes the characteristics of each:

- Logging

  Logging involves saving information about the error in specific MCA registers. If the error reporting bank associated with the error source is enabled, logging occurs; if disabled, error information is generally discarded (there are implementation-specific exceptions).
- Reporting

  An uncorrected error may be reported to system software via a machine-check exception, if error reporting for the specific error source is enabled.

Reporting is the hardware-initiated action of interrupting the processor using a machine-check exception (#MC). Reporting for each specific error type can be enabled or disabled by system software though the MCA register interface. Even if reporting for an error type is disabled, logging may continue.

Disabling reporting can negatively impact both error containment and error recovery (see the next section) and should be avoided.

[AMD Public Use]

Hardware categorizes errors into three classes. These are:

- *corrected*
- *uncorrected*
- *deferred*

The following sections describe the characteristics of each of these error classes:

If an error can be corrected by hardware, no immediate action by software is required. In this case, information is logged, if enabled, to aid in later diagnosis and possible repair.

If correction is not possible, the error is classified as uncorrected. The occurrence of an uncorrected error requires immediate action by system software to either correct the error and resume the interrupted program or, if software-based correction is not possible, to determine the extent of the impact of the uncorrected error to any executing instruction stream or the architectural state of the processor or system and take actions to contain the error condition by terminating corrupted software processes.

For errors that are not corrected, but have no immediate impact on the architectural state of the system, processor core, or any current thread of execution, the error may be classified by hardware as a deferred error. Information about deferred errors is logged, if enabled, but not reported via a machine-check exception. Instead hardware monitors the error and escalates the error classification to uncorrected at the point in time where the error condition is about to impact the execution of an instruction stream or cause the corruption of the processor core or system architectural state.

This escalation results in a #MC exception, assuming that reporting for that error source is enabled. If software can correct the error, it may be possible to resume the affected program. If not, software can terminate the affected program rather than bringing down the entire system. This is referred to as *error localization*.

A common example of deferred error processing and localization is the conversion of globally uncorrected DRAM errors to process-specific consumed memory errors. In this example, uncorrected ECC-protected data that has not yet been consumed by any processor core is tagged as “poison.” Hardware reports the uncorrected data as a localized error via a #MC exception when it is about to be used (“consumed”) by an instruction execution stream.

In contrast, an error that cannot be contained and is of such severity that it has compromised the continued operation of a processor core requires immediate action to terminate system processing and may result in a hardware-enforced *shutdown*. In the shutdown state, the execution of instructions by that processor core is halted. See Section 8.2.9 “#DF—Double-Fault Exception (Vector 8)” on page 253 for a description of the shutdown processor state.

If supported, system software can chose to configure and enable hardware to generate an interrupt when a deferred error is first detected. Corrected errors may be counted as they are logged. If supported and enabled, exceeding a software-configured count threshold may be signalled via an interrupt. These notification mechanisms are independent of machine-check reporting.

[AMD Public Use]

Specific details on hardware error detection, logging, and reporting are implementation-dependent and are described in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

9.1.3 Error Recovery

When errors cannot be corrected by hardware, *error recovery* comes into play. Error recovery, as defined by MCA, always involves software intervention. Logged information about the uncorrected error condition that caused the exception allows system software to take actions to either correct the error and resume the interrupted execution stream or terminate software processes (or higher-level software constructs) that are known to be affected by the uncorrected error.

From a system perspective, all errors are either recoverable or unrecoverable. The following outlines the characteristics of each:

- *Recoverable*—Hardware has determined that the architectural state of the processor experiencing the uncorrected error has not been compromised. Software execution can continue if system software can determine the extent of the error and take actions to either:
  - *correct* the error and *resume* the interrupted stream of execution or,
  - if this is not possible, *terminate* software processes that have incurred a loss of architectural state and *continue* other software processes that are unaffected by the error.
- *Unrecoverable*—Hardware has determined that the architectural state of the processor experiencing the uncorrected error has been corrupted. Software execution cannot reliably continue.

  Software saves any diagnostic information that it may be able to gather and halts.

The fact that an error is recoverable does not mean that recovery software will be able to resume program execution. If it is unable to determine the extent of the corruption or if it determines that essential state information has been lost, it may only be able to save information about the error and halt processing.

System software has many options to recover from an uncorrected error. The following is a partial list of possible actions that system software might take:

- If it can be determined that the corruption caused by the uncorrected error is contained within a software process, software can kill the process.
- If the uncorrected error has corrupted the architectural state of a virtual machine, the VMM can rebuild the container (using only hardware resources that are known to be good) and reboot the guest operating system.
- If the uncorrected error is a part of a block of data being transferred to or from an I/O device, the data transfer can be flushed and retried or terminated with an error.
- If the uncorrected error is due to a hard link failure, software can reconfigure the network to route information around the failed link.

[AMD Public Use]

- If the uncorrected error is in a cache and the cache line containing the uncorrected (known bad) data is in the shared state, software can invalidate the line so that it will be reloaded from memory or another cache that has the line in the owned state.

Many more error scenarios are recoverable depending on the effectiveness of hardware error containment, the logging capabilities of the system, and the sophistication of the recovery software that acts on the information conveyed through the MCA reporting structure.

If recovery software is unable to restore a valid system architectural state at some level of software abstraction (process, guest operating system, virtual machine, or virtual machine monitor), the uncorrected error is considered *system fatal*. In this situation, system software must halt the execution of instructions. A system reset is required to restore the system to a known-good architectural state.

9.2 Determining Machine-Check Architecture Support

Support for the machine-check architecture is implementation-dependent. System software executes the CPUID instruction to determine whether a processor implements the machine-check exception (#MC) and the global MCA MSRs. The CPUID Fn0000_0001_EDX[MCE] feature bit indicates support for the machine-check exception and the CPUID Fn0000_0001_EDX[MCA] feature bit indicates support for the base set of global machine-check MSRs.

Once system software determines that the base set of MCA MSRs is available, it determines the implemented number of machine-check reporting banks by reading the machine-check capabilities register (MCG_CAP), which is the first of the global MCA MSRs.

For a processor implementation to provide an architecturally compliant MCA interface, it must provide support for the machine-check exception, the global machine-check MSRs, the watchdog timer (see “CPU Watchdog Timer Register” on page 306.), and at least one bank of the machine-check reporting registers.

Support for the deferred reporting and software-based containment of uncorrected data errors is indicated by the feature bit CPUID Fn8000_0007_EBX[SUCCOR]. See “Machine-Check Recovery” on page 309.

Support for recoverable MCA overflow conditions is indicated by feature bit CPUID Fn8000_0007_EBX[McaOverflowRecov]. See the discussion of recoverable status overflow in Section 9.3.2.1 “MCA Overflow” on page 309.

Implementation-specific information concerning the machine-check mechanism can be found in the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product. For more information on using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 73.

[AMD Public Use]

9.3 Machine Check Architecture MSRs

The AMD64 Machine-Check Architecture defines the set of model-specific registers (MCA MSRs) used to log and report hardware errors. These registers are:

- Global status and control registers:
  - Machine-check global-capabilities register (MCG_CAP)
  - Machine-check global-status register (MCG_STATUS)
  - Machine-check global-control register (MCG_CTL)
  - Machine-check exception configuration register (MCA_INTR_CFG)
- One or more error-reporting register banks, each containing:
  - Machine-check control register (MC*i*_CTL)
  - Machine-check status register (MC*i*_STATUS)
  - Machine-check address register (MC*i*_ADDR)
  - At least one machine-check miscellaneous error-information register (MC*i*_MISC0)

  Each error-reporting register bank is associated with a specific processor unit (or group of processor units).
- CPU Watchdog Timer register (CPU_WATCHDOG_TIMER)
- MCA Query Registers (MCA_QUERY[3:0])

The error-reporting registers retain their values through a warm reset. (A warm reset occurs while power to the processor is stable. This in contrast to a cold reset, which occurs during the application of power after a period of power loss.) This preservation of error information allows the platform firmware or other system-boot software to recover and report information associated with the error when the processor is forced into a shutdown state.

The RDMSR and WRMSR instructions are used to read and write the machine-check MSRs. See “Machine-Check MSRs” on page 770 for a listing of the machine-check MSR numbers and their reset values. The following sections describe each MCA MSR and its function.

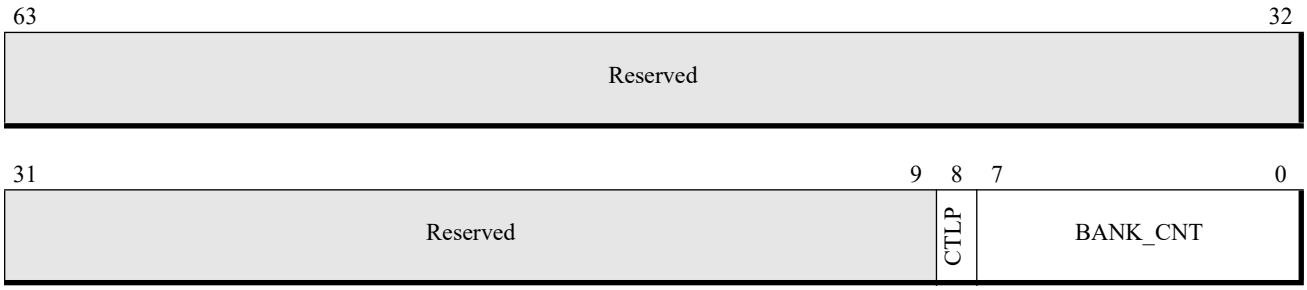
9.3.1 Global Status and Control Registers

The global status and control MSRs are the MCG_CAP, MCG_STATUS, MCG_CTL, MCA_INTR_CFG and MCA_QUERY registers.

9.3.1.1 Machine-Check Global-Capabilities Register

Figure 9-1 shows the format of the machine-check global-capabilities register (MCG_CAP). MCG_CAP is a read-only register that specifies the machine-check mechanism capabilities supported by the processor implementation.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:9	Reserved		R
8	CTLP	MCG_CTL register present	R
7:0	BANK_CNT	Number of reporting banks	R

Figure 9-1. MCG_CAP Register

The fields within the MCG_CAP register are:

- *BANK_CNT (MCi Bank Count)*—Bits 7:0. This field specifies how many error-reporting register banks are supported by the processor implementation.
- *CTLP(MCG_CTL Register Present)*—Bit 8. This bit specifies whether or not the Machine-Check Global-Control (MCG_CTL) Register is supported by the processor. When the bit is set to 1, the register is supported. When the bit is cleared to 0, the register is unsupported. The MCG_CTL register is described on page 305.

All remaining bits in the MCG_CAP register are reserved. Writing values to the MCG_CAP register produces undefined results.

9.3.1.2 Machine-Check Global-Status Register

Figure 9-2 shows the format of the machine-check global-status register (MCG_STATUS). MCG_STATUS provides basic information about the processor state after the occurrence of a machine-check error.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:3	Reserved		RAZ
2	MCIP	Machine Check In-Progress	R/W
1	EIPV	Error IP Valid Flag	R/W
0	RIPV	Restart IP Valid Flag	R/W

Figure 9-2. MCG_STATUS Register

The fields within the MCG_STATUS register are:

- *Restart-IP Valid (RIPV)*—Bit 0. When this bit is set to 1, the interrupted program can be reliably restarted at the instruction addressed by the instruction pointer pushed onto the stack by the machine-check error mechanism. If this bit is cleared to 0, the interrupted program cannot be reliably restarted.
- *Error-IP Valid (EIPV)*—Bit 1. When this bit is set to 1, the process thread that was interrupted by the #MC is responsible for the machine-check error, which occurred at the privilege level indicated by the CS selector that was pushed on the stack by the exception. If this bit is cleared to 0, it is possible that the interrupted thread is not responsible for the machine-check error.
- *Machine Check In-Progress (MCIP)*—Bit 2. When this bit is set to 1, it indicates that a machine-check error is in progress. If another machine-check error occurs while this bit is set, the processor enters the shutdown state. The processor sets this bit whenever a machine check exception is generated. Software is responsible for clearing it after the machine check exception is handled.

All remaining bits in the MCG_STATUS register are reserved.

9.3.1.3 Machine-Check Global-Control Register

Figure 9-3 shows the format of the machine-check global-control register (MCG_CTL). MCG_CTL is used by software to enable or disable the logging and reporting of machine-check errors from the implemented error-reporting banks. Depending on the implementation, detected errors from some error sources associated with a reporting bank that is disabled are still logged. Setting all bits to 1 in this register enables all implemented error-reporting register banks to log errors.

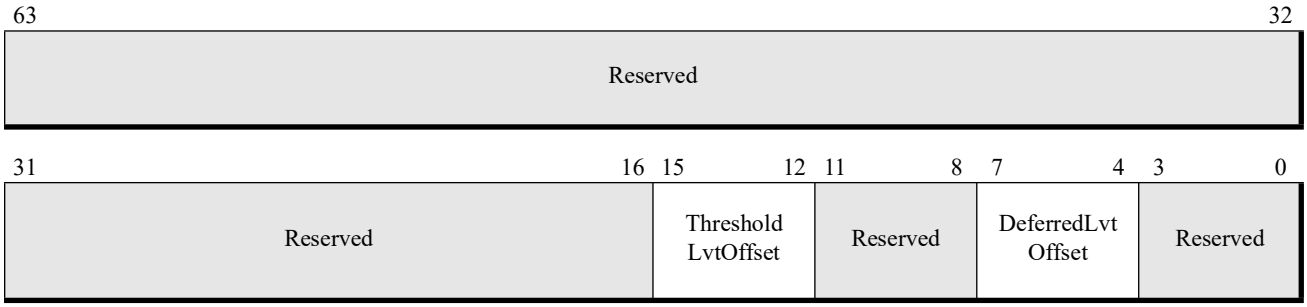
[AMD Public Use]



Figure 9-3. MCG_CTL Register

9.3.1.4 Machine-Check Exception Configuration Register

In addition to the standard Machine Check exception (#MC) signaling, certain machine check events can raise an interrupt via the APIC LVT mechanism. Configuration of these interrupts are provided, in part, in register MCA_INTR_CFG located at MSR C000_0410, see Figure 9-4. See Section 16.4 “Local Interrupts” on page 666 for details on configuring LVT entries.



Bits	Mnemonic	Description	Access type
63:16	Reserved		RAZ
15:12	ThresholdLvtOffset	For error thresholding interrupts, specifies the address of the LVT entry in the APIC registers as follows: LVT address = (ThresholdLvtOffset < < 4) + 500h (see APIC[570:500]).	R/W
11:8	Reserved		RAZ
7:4	DeferredLvtOffset	For deferred error interrupts, specifies the address of the LVT entry in the APIC registers as follows: LVT address = (DeferredLvtOffset < < 4) + 500h (see APIC[570:500]).	R/W
3:0	Reserved		RAZ

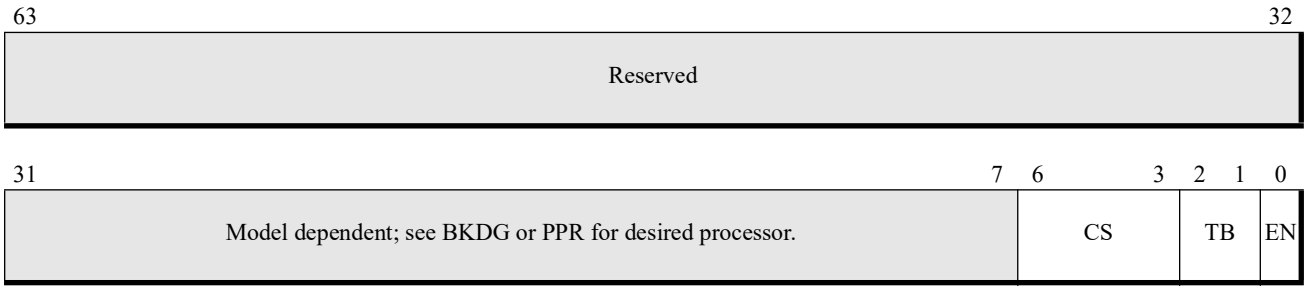
Figure 9-4. MCA_INTR_CFG Register

9.3.1.5 CPU Watchdog Timer Register

The CPU watchdog timer is used to generate a machine check condition when an instruction does not complete within a time period specified by the CPU Watchdog Timer register. The timer restarts the count each time an instruction completes, when enabled by the *CPU Watchdog Timer Enable* bit. The time period is determined by the *Count Select* and *Time Base* fields. The timer does not count during halt or stop-grant.

The format of the CPU watchdog timer is shown in Figure 9-5.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:7	Reserved		SBZ
6:3	CS	CPU Watchdog Timer Count Select	R/W
2:1	TB	CPU Watchdog Timer Time Base	R/W
0	EN	CPU Watchdog Timer Enable	R/W

Figure 9-5. CPU Watchdog Timer Register Format

CPU Watchdog Timer Enable (EN) - Bit 0. This bit specifies whether the CPU Watchdog Timer is enabled. When the bit is set to 1, the timer increments and generates a machine check when the timer expires. When cleared to 0, the timer does not increment and no machine check is generated.

CPU Watchdog Timer Time Base (TB) - Bits 2:1. Specifies the time base for the time-out period indicated in the *Count Select* field. The allowable time base values are provided in Table 9-1.

Table 9-1. CPU Watchdog Timer Time Base

TB[1:0]	Time Base
00b	1 millisecond
01b	1 microsecond
10b	Reserved
11b	Reserved

CPU Watchdog Timer Count Select (CS) - Bits 6:3. Specifies the time period required for the CPU Watchdog Timer to expire. The time period is this value times the time base specified in the *Time Base* field. The allowable values are shown in Table 9-2.

Table 9-2. CPU Watchdog Timer Count Select

CS[3:0]	Value
0000b	4095
0001b	2047
0010b	1023
0011b	511
0100b	255
0101b	127

[AMD Public Use]

Table 9-2. CPU Watchdog Timer Count Select (continued)

0110b	63
0111b	31
1000b	8191
1001b	16383
1010b–1111b	Reserved

9.3.1.6 MCA Query Registers

Processors that set CPUID Fn8000_0007_EBX[MCA_QUERY_MSR] (bit 15) = 1 support MCA Query MSRs. The MCA Query MSRs are read-only and attempt to write results in a #GP exception. Each bit in the MCA Query MSRs reports the current value of MCi_STATUS[VAL] and allows software to quickly identify MCA banks with errors without having to read MCA bank's status register.

- MCA_QUERY0 (MSR C000_0420h): MCA banks 0 to 63
- MCA_QUERY1 (MSR C000_0421h): reserved for future use
- MCA_QUERY2 (MSR C000_0422h): reserved for future use
- MCA_QUERY3 (MSR C000_0423h): reserved for future use

9.3.2 Error-Reporting Register Banks

Each error-reporting register bank contains the following registers:

- Machine-check control register (MC*i*_CTL).
- Machine-check status register (MC*i*_STATUS).
- Machine-check address register (MC*i*_ADDR).
- Machine-check miscellaneous error-information register 0 (MC*i*_MISC0).

The *i* in each register name corresponds to the number of a supported register bank. Each error-reporting register bank is normally associated with a specific execution unit. The number of error-reporting register banks is implementation-specific. For more information, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

Software reads the MCG_CAP register to determine the number of supported register banks. The first error-reporting register (MC0_CTL) always starts with MSR address 400h, followed by MC0_STATUS (401h), MC0_ADDR (402h), and MC0_MISC0 (403h). The addresses of any additional error-reporting MSRs are assigned sequentially starting at 404h through the remaining supported register banks.

[AMD Public Use]

9.3.2.1 MCA Overflow

If an error occurs within an error reporting bank while the status register for that bank contains valid data (MC*i*_STATUS[VAL] = 1), an MCA overflow condition results. In this situation, information about the new error will either be discarded or will replace the information about the prior error.

Hardware sets the MC*i*_STATUS[OVER] bit to indicate this condition has occurred and follows a set of rules to determine whether to overwrite the previously logged error information or discard the new error information. These rules are shown in Table 9-3 below.

Table 9-3. Error Logging Priorities

		Previous Error Type		
		Corrected	Deferred	Uncorrected
Current Error Type	**Corrected**	Discard Current	Discard Current	Discard Current
	Deferred	Overwrite Previous	Discard Current	Discard Current
	Uncorrected	Overwrite Previous	Overwrite Previous	Discard Current

Note(s):

1. *Logging a deferred error has priority over the retention of information concerning a prior corrected error.*
2. *Logging an uncorrected error has priority over the retention of information concerning either a prior deferred or corrected error.*
3. *Valid Information concerning an uncorrected error is not overwritten by any subsequent errors.*

If the VAL bit is not set, hardware writes the appropriate logging registers based on the type of error (writing the MC*i*_STATUS register last) and then sets the VAL bit to indicate to software that the information currently contained in the MC*i*_STATUS register is valid. Software clears the VAL bit after reading the contents of this register (after reading and saving valid information stored in any of the other logging registers) to indicate to hardware that it has saved the information, making the registers available to log the next error.

If survivable MCA overflow is supported by the implementation (as indicated by CPUID Fn8000_0007_EBX[McaOverflowRecov] = 1), the state of the MC*i*_STATUS[PCC] bit indicates whether system execution can continue. If a particular processor does not support survivable MCA overflow and overflow occurs, software must halt instruction execution on that processor core regardless of the state of the PCC bit because critical information may have been lost as a result of the overflow. See the description of the Machine-Check Status registers below for more information on the PCC bit.

9.3.2.2 Machine-Check Recovery

Machine Check Recovery is a feature allowing recovery of the system when the hardware cannot correct an error. Machine Check Recovery is supported when CPUID Fn8000_0007_EBX[SUCCOR]=1.

When Machine Check Recovery is supported and an uncorrected error has been detected that the hardware can contain to the task or process to which the machine check has been delivered, it logs a context-synchronous uncorrectable error (MCi_STATUS[UC]=1, MCi_STATUS[PCC]=0). The rest

[AMD Public Use]

of the system is unaffected and may continue running if supervisory software can terminate only the affected process context.

9.3.2.3 Machine-Check Control Registers

The machine-check control registers (MC*i*_CTL), as shown in Figure 9-6, contain an enable bit for each error source within an error-reporting register bank. Setting an enable bit to 1 enables error reporting for the specific feature controlled by the bit, and clearing the bit to 0 disables error reporting for the feature. It is recommended that the value FFFF_FFFF_FFFF_FFFFh be programmed into each MC*i*_CTL register.

Disabling the reporting of errors from error sources that are capable of detecting uncorrected errors can compromise future error recovery and is not recommended. Other implementation-specific values are documented in the product's *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* .



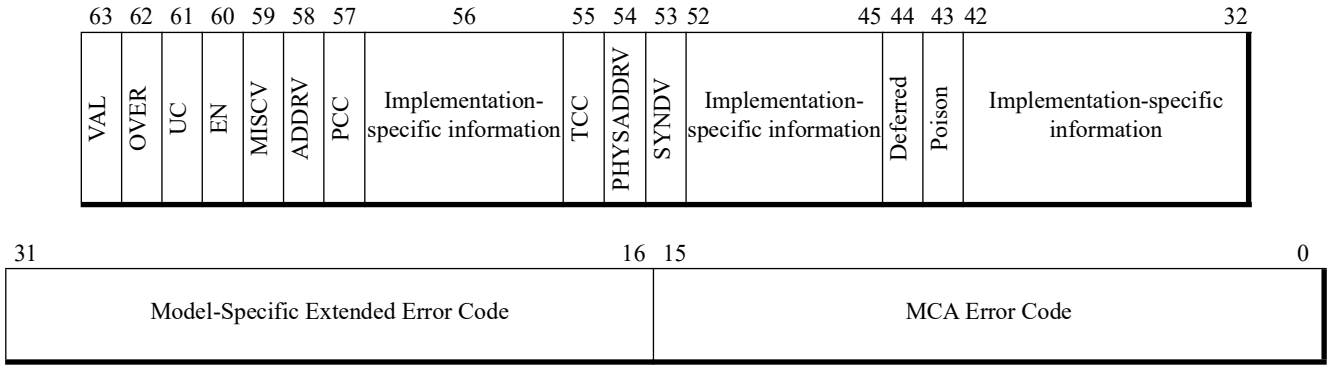
Figure 9-6. MC*i*_CTL Register

9.3.2.4 Machine-Check Status Registers

Each error-reporting register bank includes a machine-check status register (MC*i*_STATUS) that the processor uses to log error information. Hardware writes the status register bits when an error is detected, and sets the VAL bit of the register to 1, indicating that the status information is valid. Error reporting for the error source associated with the detected error *does not* need to be enabled in the MC*i*_CTL Register for the processor to write the status register. Error reporting must be enabled for the error to be reported via a #MC exception. Software is responsible for clearing the status register after the exception has been handled. Attempting to write a value other than 0 to an MC*i*_STATUS register in an implemented MCA bank may raise a general-protection (#GP) exception.

Figure 9-7 on page 311 shows the format of the MC*i*_STATUS register.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63	VAL	Valid	R/W*
62	OVER	Status Register Overflow	R/W*
61	UC	Uncorrected Error	R/W*
60	EN	Error Condition Enabled	R/W*
59	MISCV	Miscellaneous-Error Register Valid	R/W*
58	ADDRV	Error-Address Register Valid	R/W*
57	PCC	Processor-Context Corrupt	R/W*
56		Implementation-specific information	R/W*
55	TCC	Task-Context Corrupt	R/W*
54	PHYSADDRV	Error-Address Register Physical Address Valid	R/W*
53	SYNDV	Syndrome Register Valid	R/W*
52:45		Implementation-specific information	R/W*
44	Deferred	Deferred error	R/W*
43	Poison	Poisoned data consumed	R/W*
42:32		Implementation-specific information	R/W*
31:16		Model-Specific Extended Error Code	R/W*
15:0		MCA Error Code	R/W*

* System software can only clear this bit to 0.

Figure 9-7. MC*i*_STATUS Register

The fields within the MC*i*_STATUS register are:

- *MCA Error Code*—Bits 15:0. This field encodes information about the error, including:
  - The type of transaction that caused the error.
  - The memory-hierarchy level involved in the error.
  - The type of request that caused the error.
  - Other information concerning the transaction type.

  See the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product for information on the format and encoding of the MCA error code.

[AMD Public Use]

- *Model-Specific Extended Error Code*—Bits 31:16. This field encodes model-specific information about the error. For further information, see the documentation for particular implementations of the architecture.
- *Implementation-specific Information*—Bits 56, 54:45, 42:32. These bit ranges hold model-specific error information. Software should not rely on the field definitions in these ranges being consistent between processor implementations. For details see the *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for desired implementations of the architecture.
- *Poison*—Bit 43. When set to 1, this bit indicates that the uncorrected error condition being reported is due to the attempted use of data that was previous detected as in error (and could not be corrected) and marked as known-bad.
- *Deferred*—Bit 44. When set to 1, this bit indicates that hardware has determined that the error condition being logged has not affected the execution of any instruction stream and that action by system software to prevent or correct an error is not required. No machine-check exception is signaled. Hardware will monitor the error and log an uncorrected error when the execution of any thread of execution is impacted.
- *SYNDV*—Bit 53. When set to 1, this bit indicates that the contents of the corresponding syndrome register (MCA_SYND) are valid. When this bit is cleared, the contents of MCA_SYND are not valid.
- *PHYSADDRV*—Bit 54. When set to 1 and ADDRV is set, this bit indicates that the contents of the corresponding address register (MCi_ADDR) contains a system physical address. Otherwise, the type of address in MCi_ADDR is implementation specific.
- *TCC*—Bit 55. When set to 1, this bit indicates that the hardware context of the process thread to which the error was reported may have been corrupted. Continued operation of the thread may have unpredictable results. When this bit is cleared, the hardware context of the process thread to which the error was reported is not corrupted and recovery of the process thread is possible. This bit is only meaningful when MCA_STATUS[PCC]=0.
- *PCC*—Bit 57. When set to 1, this bit indicates that the processor state is likely to be corrupt due to an uncorrected error. In this case, it is possible that software cannot reliably continue execution. When this bit is cleared, the processor state is not corrupted and recovery is still possible. If the PCC bit is set in any error bank, the processor will clear RIPV and EIPV in the MCG_STATUS register.
- *ADDRV*—Bit 58. When set to 1, this bit indicates that the contents of the corresponding error-reporting address register (MC*i*_ADDR) are valid. When this bit is cleared, the contents of MC*i*_ADDR are not valid.
- *MISCV*—Bit 59. When set to 1, this bit indicates that additional information about the error is saved in the corresponding error-reporting miscellaneous register (MC*i*_MISC0). When cleared, this bit indicates that the contents of the MC*i*_MISC0 register are not valid.
- *EN*—Bit 60. When set to 1, this bit indicates that the error condition is enabled in the corresponding error-reporting control register (MC*i*_CTL). Errors disabled by MC*i*_CTL do not cause a machine-check exception.

[AMD Public Use]

- *UC*—Bit 61. When set to 1, this bit indicates that the logged error status is for an uncorrected error. When cleared, the error class is determined by looking at the Deferred bit; the error is a Corrected error if the Deferred bit is clear or a Deferred error if the Deferred bit is set. (See Section 9.1.2, “Error Detection, Logging, and Reporting,” on page 299, for more detail on these error classes.)
- *OVER*—Bit 62. This bit is set to 1 by the processor if the VAL bit is already set to 1 as the processor attempts to write error information into MC*i*_STATUS. In this situation, the machine-check mechanism handles the contents of MC*i*_STATUS as follows:
  - For processor implementations that log errors for disabled reporting banks, status for an enabled error replaces status for a disabled error.
  - Status for a deferred error replaces status for a corrected error.
  - Status for an uncorrected error replaces status for a corrected or deferred error.
  - Status for an enabled uncorrected error is never replaced.

  See Section 9.3.2.1 “MCA Overflow” on page 309 for more information on this field.
- *VAL*—Bit 63. This bit is set to 1 by the processor if the contents of MC*i*_STATUS are valid. Software should clear the VAL bit after reading the MC*i*_STATUS register, otherwise a subsequent machine-check error sets the OVER bit as described above.

When a machine-check error occurs, the processor writes an error code into the appropriate MC*i*_STATUS register MCA error-code field. The MC*i*_STATUS[VAL] bit is set to 1, indicating that the MC*i*_STATUS register contents are valid.

MCA error-codes are used to report errors in the memory hierarchy, the system bus, and the system-interconnection logic. Error-codes are divided into subfields that are used to describe the cause of an error. The information is implementation-specific. For further information, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

9.3.2.5 Machine-Check Address Registers

Each error-reporting register bank includes a machine-check address register (MC*i*_ADDR) that the processor uses to report the address or location associated with the logged error. The address field can hold a virtual (linear) address, a physical address, or a value indicating an internal physical location, depending on the type of error. For further information, see the documentation for particular implementations of the architecture. The contents of this register are valid only if the ADDRV bit in the corresponding MC*i*_STATUS register is set to 1.

9.3.2.6 Machine-Check Miscellaneous-Error Information Register 0 (MC*i*_MISC0)

Each error-reporting register bank includes the Machine-Check Miscellaneous 0 register that the processor uses to report additional error information.

In some implementations, the MC*i*_MISC0 register is used for error thresholding. Thresholding is a mechanism provided by hardware to:

- count detected errors, and

[AMD Public Use]

- (optionally) generate an APIC-based interrupt when a programmed number of errors has been counted.

Processor hardware counts detected errors and ensures that multiple error sources do not share the same thresholding register. Software can use corrected error counts to help predict which components might soon fail (begin generating uncorrectable errors) and schedule their replacement.

Threshold counters increment for error sources that are enabled for logging.

9.3.2.7 Additional Machine-Check Miscellaneous-Error Information Registers (MC*i*_MISC*j*)

The MC*i*_MISC0[BLKP] field is used to point to any additional MC*i*_MISC*j* registers, where j > 0. If this field is zero, no additional MC*i*_MISC registers are implemented. If this field is one, and CPUID Fn8000_0007_EBX[ScalableMca]=1, refer to Section 9.3.3, “Machine-Check Architecture Extension Registers,” on page 316 for addresses of additional MC*i*_MISC*j* registers.

If the MC*i*_MISC0[BLKP] field is non-zero and CPUID Fn8000_0007_EBX[ScalableMca]=0, up to 8 additional MC*i*_MISC*j* registers can be implemented for the error-reporting bank *i* (for a total of 9). These registers are allocated in contiguous blocks of 8, with MC*i*_MISC1 addressed by:

MC*i*_MISC1 address = C000_0400h + (MC*i*_MISC0[BLKP] << 3)

This is illustrated in Figure 9-8 below.

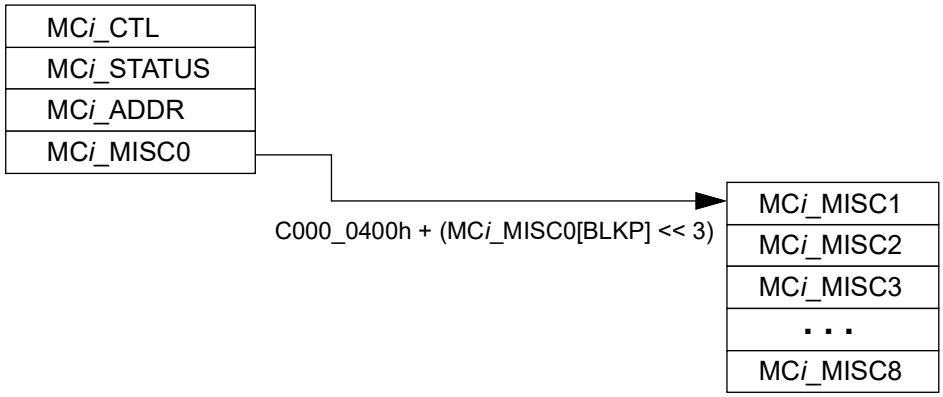
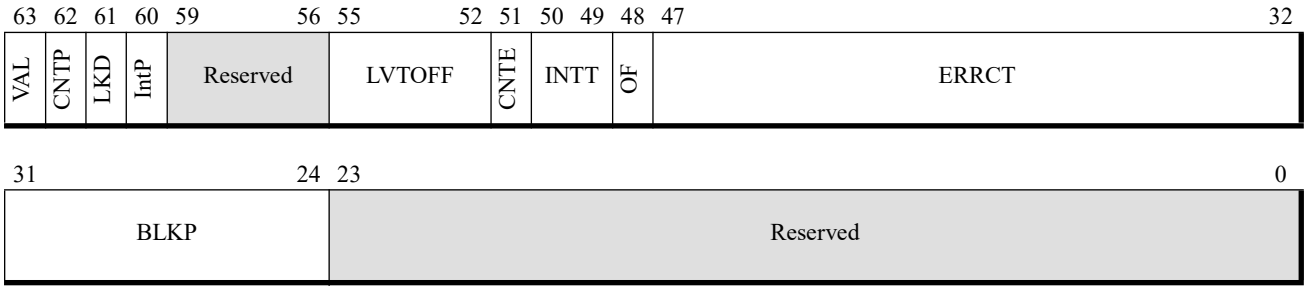


Figure 9-8. MC*i*_MISC1 Addressing

The format of implemented MC*i*_MISC*j* registers depends upon their use and use can vary from one implementation to another. Figure 9-9 below illustrates the format of a miscellaneous error information register when used as an error thresholding register.

All miscellaneous error information registers will contain the VAL field in bit position 63. MC*i*_MISC0 must contain the BLKP field in bits 31:24.

[AMD Public Use]



Bits	Mnemonic	Description	Access type	Reset
63	VAL	Valid	R	1b
62	CNTP	Counter Present	R	1b
61	LKD	Locked	R/W	0b
60	IntP	Thresholding Interrupt Supported	R	Implementation Dependent
59:56	Reserved			
55:52	LVTOFF	LVT Offset	R/W	0000b
51	CNTE	Counter Enable	R/W	0b
50:49	INTT	Interrupt Type	R/W	00b
48	OF	Overflow	R/W	Undefined
47:32	ERRCT	Error Counter	R/W	Undefined
31:24	BLKP	Block pointer for additional MISC registers	R	Undefined
23:0	Reserved			

Figure 9-9. Miscellaneous Information Register (Thresholding Register Format)

The fields within the MC*i*_MISC*j* register are:

- *Valid (VAL)*—Bit 63. When set to 1, indicates that the counter present (CNTP) and block pointer (BLKP) fields in this register are valid.
- *Counter Present (CNTP)*—Bit 62. When set to 1, indicates the presence of a threshold counter.
- *Locked (LKD)*—Bit 61. When set to 1, indicates that the threshold counter is not available for OS use. If this is the case, writes to bits 60:0 of this register are ignored and do not generate a fault. Software must check the Locked bit before writing into the thresholding register.

  This field is write-enabled by MSR C001_0015h Hardware Configuration Register [MCSTATUSWrEn].
- *IntP (Thresholding Interrupt Supported)*—Bit 60. When set, this bit indicates that the reporting of threshold overflow via interrupt is supported. Interrupt type is determined by the setting of the INTT field.
- *LVT Offset (LVTOFF)*—Bits 55:52. This field specifies the address of the APIC LVT entry to deliver the threshold counter interrupt. Software must initialize the APIC LVT entry before enabling the threshold counter to generate the APIC interrupt; otherwise, undefined behavior may result. This field is aliased to MCA_INTR_CFG[ThresholdLvtOffest].

[AMD Public Use]

- *Counter Enable (CNTE)*—Bit 51. When set to 1, counting of implementation-dependent errors is enabled; otherwise, counting is disabled.
- *Interrupt Type (INTT)*—Bits 50:49. The value of this field specifies the type of interrupt signaled when the value of the overflow bit changes from 0 to 1.
  - 00b = No interrupt
  - 01b = APIC-based interrupt
  - 10b = Reserved
  - 11b = Reserved
- *Overflow (OF)*—Bit 48. The value of this field is maintained through a warm reset. This bit is set by hardware when the error counter increments to its maximum implementation-supported value (from FFFEh to FFFFh for the maximum implementation-supported value). This is defined as the threshold level. When the overflow bit is set, the interrupt selected by the interrupt type field is generated. Software must reset this bit to zero in the interrupt handler routine when they update the error counter.
- *Error Counter (ERRCT)*—Bits 47:32. This field is maintained through a warm reset. The size of the threshold counter is implementation-dependent. Implementations with less than 16 bits fill the most significant unimplemented bits with zeros.

  Software enumerates the counter bits to discover the size of the counter and the threshold level (when counter increments to the maximum count implemented). Software sets the starting error count as follows:

  Starting error count = threshold level – desired software error count to cause overflow

  The error counter is incremented by hardware when errors for the associated error counter are logged. When this counter overflows, it stays at the maximum error count (with no rollover). MCA_MISCx[ErrCt] will not increment on errors that set MCA_STATUS[Poison].
- *Block pointer for additional MISC registers (BLKP)*—Bits 31:24. This field is only valid when valid (VAL) bit is set. When non-zero, this field is used to indicate the presence of additional MCi_MISC registers.

Other formats for miscellaneous information registers are implementation-dependent, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product for more details.

9.3.3 Machine-Check Architecture Extension Registers

This section describes the Machine Check Architecture Extension (MCAX). MCAX is available on a processor when CPUID Fn8000_0007_EBX[ScalableMca] returns 1.

A processor that supports MCAX supports up to 64 MCA banks per thread, with 16 MCAX registers per bank, at MSRs C000_2[3FF:000]. MSRs C000_2[FFF:400] are reserved for future use.

Software can identify an MCA bank as MCAX-capable by checking for MCA_CONFIG[Mcax]=1. An MCAX bank contains the following MCAX registers: MCA_CTL, MCA_STATUS,

[AMD Public Use]

MCA_ADDR, MCA_MISC0, MCA_CONFIG, MCA_IPID, MCA_SYND, MCA_DESTAT, MCA_MISC[4:1], and MCA_SYND[2:1]. See Figure 9-10 for address mapping of registers in a MCAX bank.

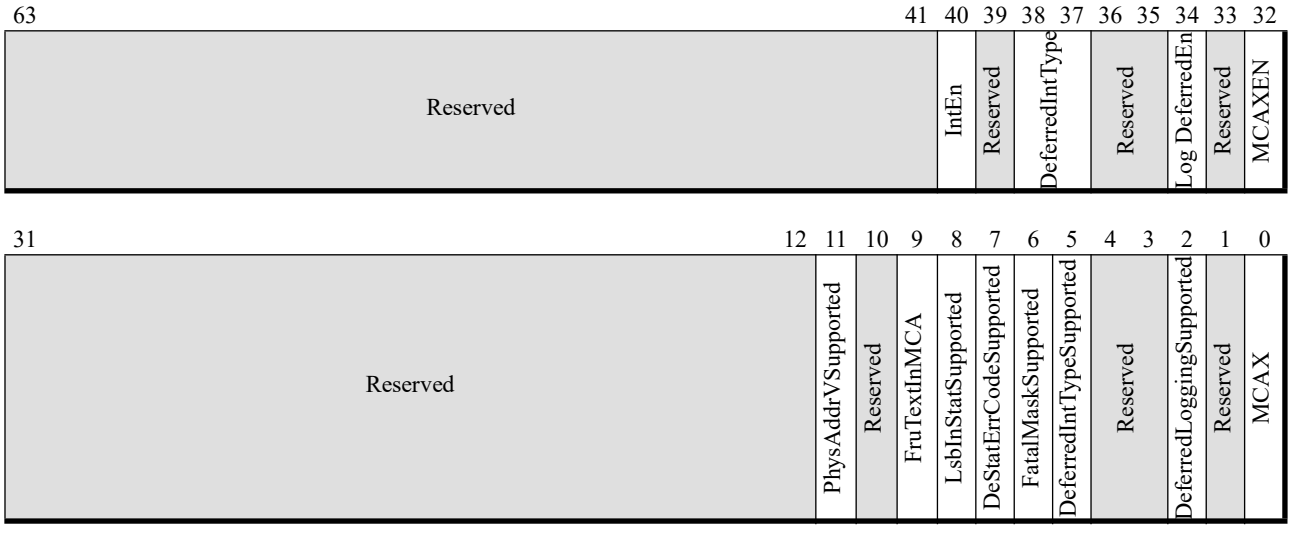
MCAX bank registers MCA_CTL, MCA_STATUS, MCA_ADDR, and MCA_MISC0 are aliased to MCA bank registers MCi_CTL, MCi_STATUS, MCi_ADDR, and MCi_MISC0 in MCA banks 0 to 31. The format of MCA_CTL, MCA_STATUS, MCA_ADDR, and MCA_MISC0 is the same as the format of the corresponding MCA bank register.

MCA bank number (decimal)	MCAX Bank registers (MSR C000_2xxx)				MCAX Bank registers (MSR C000_2xxx)							
	MCA_CTL	MCA_STATUS	MCA_ADDR	MCA_MISC0	MCA_CONFIG	MCA_IPID	MCA_SYND	Reserved	MCA_DESTAT	MCA_DEADDR	MCA_MISC[1:4]	MCA_SYND[1:2]
0	000h	001h	002h	003h	004h	005h	006h	007h	008h	009h	00Ah:00Dh	00Eh:00Fh
n	000h + n*10h	001h + n*10h	002h + n*10h	003h + n*10h	004h + n*10h	005h + n*10h	006h + n*10h	007h + n*10h	008h + n*10h	009h + n*10h	(00Ah:00Dh) + n*10h	(00Eh:00Fh) + n*10h

Figure 9-10. Address Mapping of Registers in MCAX Bank

9.3.3.1 MCA Configuration Register

The MCA_CONFIG register holds configuration information for the MCA bank.



Bits	Mnemonic	Description	Access Type
63:41	Reserved		
40	IntEn	Interrupt Enable	R/W
39	Reserved		

[AMD Public Use]

Bits	Mnemonic	Description	Access Type
38:37	DeferredIntType	Deferred Error Interrupt Type	R/W
36:35	Reserved		
34	LogDeferredEn	Deferred Error Logging Enable	R/W
33	Reserved		
32	McaxEn	MCAX Enable	R/W
31:12	Reserved		
11	PhysAddrVSupported	PHYSADDRV bits are supported	R
10	Reserved		
9	FruTextInMCA	MCA FruText Supported	R/W
8	LsbInStatSupported	Address LSB in MCA_STATUS Supported	R
7	DeStatErrCodeSupported	Deferred Error Status Error Code Supported	R
6	FatalMaskSupported	System fatal error event supported.	R
5	DeferredIntTypeSupported	Deferred Interrupt Type Supported	R
4:3	Reserved		
2	DeferredLoggingSupported	Deferred Error Logging Supported	R
1	Reserved		
0	MCAX	MCAX Capable	R

IntEn—Bits 40. When this bit is set to 0, an interrupt will not occur for corrected errors. When this bit is set to 1 and IntPresent=1, this bank will generate an interrupt on a corrected error to the interrupt vector configured in MCA_INTR_CFG[ThresholdLvtOffset].

DeferredIntType—Bits 38:37. When MCA_CONFIG[McaxEn]=1, specifies the type of interrupt to generate on a deferred error.

00b	No interrupt
01b	APIC based interrupt
10b	SMI
11b	Reserved

LogDeferredEn—Bit 34. Enable logging of deferred errors in MCA_STATUS. 0=Log deferred errors only in MCA_DESTAT and MCA_DEADDR. 1=Log deferred errors in MCA_STATUS and MCA_ADDR in addition to MCA_DESTAT and MCA_DEADDR. This bit does not affect logging of deferred errors in MCA_SYND or MCA_MISCx.

McaxEn—Bit 32. Enable MCAX feature set. 1=System software acknowledges support for the MCAX feature set. 0=System software has not acknowledged support for the MCAX feature set; all uncorrected errors and fatal errors cause a system fatal event.

PhysAddrVSupported—Bit 11. PHYSADDRV bits in MCA_STATUS and MCA_DESTAT are supported.

FruTextInMca—Bit 9. 1=MCA_SYND1 and MCA_SYND2 contain an ASCII-formatted Field Replaceable Unit (FRU) identifier for the error logged in MCA_STATUS or MCA_DESTAT. 0=MCA_SYND1 and MCA_SYND2 contain implementation-specific information.

[AMD Public Use]

LsbInStatSupported—Bit 8. 1=MCA_STATUS[29:24] and MCA_DESTAT[29:24] contain the least significant valid bit of the address logged in MCA_ADDR and MCA_DEADDR.

DeStatErrCodeSupported—Bit 7. 1=MCA_DESTAT contains an MCA Error Code and a Model-specific Extended Error Code for the error logged in MCA_DESTAT.

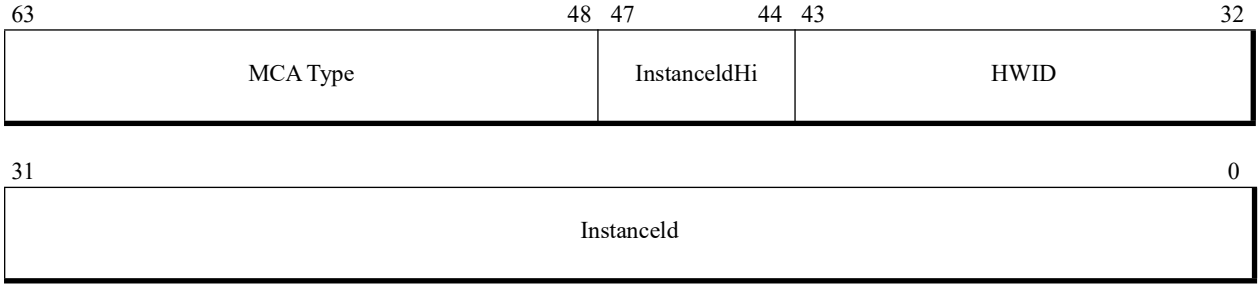
DeferredIntTypeSupported—Bit 5. 1=MCA_CONFIG[DeferredIntType] controls the type of interrupt generated on a deferred error. Deferred errors are supported in this bank only if MCA_CONFIG[DeferredLoggingSupported]=1.

DeferredLoggingSupported—Bit 2. 1= MCA_CONFIG[LogDeferredEn] in this bank controls the logging behavior for deferred errors. 0= MCA_CONFIG[LogDeferredEn] is not supported.

MCAX—Bit 0. MCAX: Set by hardware. This MCA bank provides Machine Check Architecture Extensions.

9.3.3.2 MCA IP Identification

This register holds information which identifies the specific bank type: the McaType value, HWID, and the InstanceID. By providing those values as part of the MCA, the bank type information is available to the machine check handler for parsing and for diagnosis.



Bits	Mnemonic	Description	Access type
63:48	McaType	MCA bank type.	R/W
47:44	InstanceIdHi	The high bits of the Instance ID.	R/W
43:32	Hwid	Hardware ID value.	R/W
31:0	InstanceID	The low bits of the Instance ID.	R/W

Figure 9-11. MCA_IPID Register

McaType—Bits 63:48. MCA bank type. This field is used to identify the MCA bank type in conjunction with MCA_IPID[Hwid]. For more information, see the Processor Programming Reference Manual applicable to your product.

InstanceIdHi—Bits 47:44. The high bits of the Instance identification for the MCA bank. This field is used to identify the MCA bank instance in conjunction with MCA_IPID[InstanceId]. For more information, see the Processor Programming Reference Manual applicable to your product.

[AMD Public Use]

Hwid—Bits 43:32. MCA bank hardware identification. For more information, see the Processor Programming Reference Manual applicable to your product.

InstanceId—Bits 31:0. The Instance identification for the MCA bank. This field is used to identify the MCA bank instance.

9.3.3.3 MCA Syndrome Register

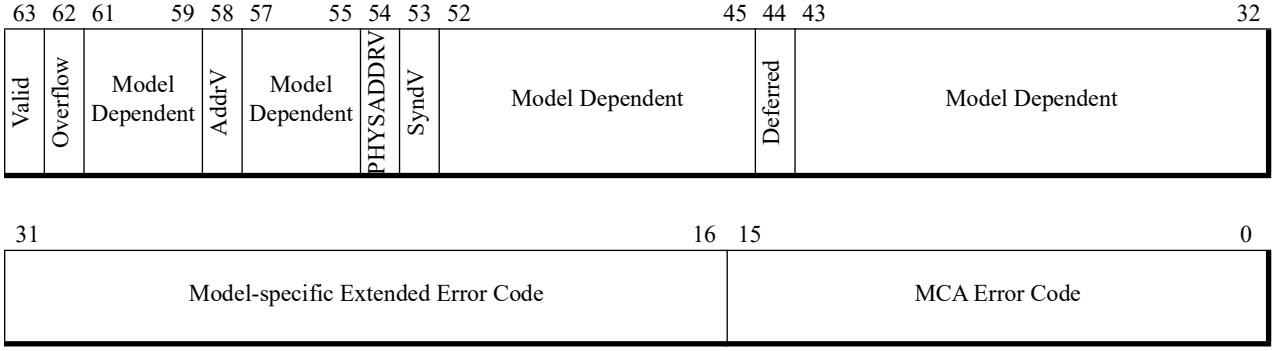
The MCA_SYND register stores a syndrome associated with the error logged in MCA_STATUS or MCA_DESTAT. The "syndrome" may include syndrome values associated with an error correcting code or other information about the error. The contents of this register are valid if MCA_STATUS[SYNDV] bit is set to 1 or MCA_DESTAT[SYNDV] bit is set to 1.

The format and contents of this register are implementation dependent. For more information, see the Processor Programming Reference Manual applicable to your product.

9.3.3.4 MCA Deferred Error Status Register

This register holds status information for deferred errors. This register ensures that software will see a deferred error, even when a later error of higher severity occurs. If the error being logged is a deferred error, then the error will be logged to MCA_DESTAT. See Section 9.3.2.4, "Machine-Check Status Registers," on page 310 for more detailed descriptions of the register fields.

When the deferred error has been processed by the deferred error handler, MCA_DESTAT should be cleared. If MCA_STATUS also contains a deferred error, MCA_STATUS should be cleared.



Bits	Mnemonic	Description	Access type
63	Valid	A valid error is contained in this register.	R/W
62	Overflow	One or more deferred errors was not logged.	R/W
61:59		Model Dependent.	
58	AddrV	An address is contained in MCA_DEADDR.	R/W
57:56		Model Dependent.	
54	PhysAddrV	A system physical address is contained in MCA_DEADDR	R/W
53	SyndV	Syndrome valid. MCA_SYND contains error syndrome information associated with this deferred error.	R/W
52:45		Model Dependent.	
44	Deferred	Error is a deferred error.	R/W

[AMD Public Use]

Bits	Mnemonic	Description	Access type
43:32		Model Dependent.	
31:16	Model-Specific Extended Error Code	This field encodes model-specific information about the error. Valid if MCA_CONFIG[DeStatErrCodeSupported]=1, else Reserved.	R/W
15:0	MCA Error Code	This field encodes information about the error logged in MCA_DESTAT. Valid if MCA_CONFIG[DeStatErrCodeSupported]=1, else Reserved.	R/W

Figure 9-12. MCA_DESTAT Register

9.3.3.5 MCA Deferred Error Address Register

The MCA_DEADDR register provides the address associated with an error logged in MCA_DESTAT.

The format of this register is the same as MCA_ADDR.

The register is only meaningful if MCA_DESTAT[Valid]=1 and MCA_DESTAT[ADDRV]=1.

9.3.3.6 MCA Miscellaneous Registers 1 - 4

Set by hardware.

The format of MCA_MISC[4-1] is the same as the format of MCA_MISC0.

9.3.3.7 MCA Syndrome Registers 1 - 2

The MCA_SYND[2-1] registers store information associated with the error in MCA_STATUS or MCA_DESTAT. The contents of these registers are valid if MCA_STATUS[SYNDV] is set or MCA_DESTAT[SYNDV] is set.

The format and contents of this register is implementation dependent. For more information, see the Processor Programming Reference Manual applicable to your product.

9.4 Initializing the Machine-Check Mechanism

Following a processor reset, all machine-check error-reporting enable bits are disabled. System software must enable these bits before machine-check errors can be reported. Generally, system software should initialize the machine-check mechanism using the following process:

- Execute the CPUID instruction and verify that the processor supports the machine-check exception (MCE) and machine-check registers (MCA). Software should not proceed with initializing the machine-check mechanism if the machine-check registers are not supported.
- If the machine-check registers are supported, system software should take the following steps:
  - Check to see if the CTLP bit in the MCG_CAP register is set to 1. If it is, then the MCG_CTL register is supported by the processor. If the MCG_CTL register is supported, software should set its enable bits to 1 for the machine-check features it uses. Software can load MCG_CTL with all 1s to enable all available machine-check reporting banks.

[AMD Public Use]

- Read the COUNT field from the MCG_CAP register to determine the number of error-reporting register banks supported by the processor. For each error-reporting register bank, software should set the enable bits to 1 in the MC*i*_CTL register for the error types it wants the processor to report. Software can write each MC*i*_CTL with all 1s to enable all error-reporting mechanisms.

  Not enabling reporting banks that may be involved in the reporting of uncorrected errors can lead to the loss of system reliability and error recoverability.
- Check the VAL bit on each implemented MC*i*_STATUS register. It is possible that valid error-status information has already been logged in the MC*i*_STATUS registers at the time software is attempting to initialize them. The status can reflect errors logged prior to a warm reset or errors recorded during the system power-up and boot process. Before clearing the MC*i*_STATUS registers, software should examine their contents and log any errors found.
- After saving any valid error information contained in the MC*i*_STATUS, MC*i*_ADDR, and any implemented miscellaneous error information registers for each implemented reporting bank, software should clear all status fields in the MC*i*_STATUS register for each bank by writing all 0s to the register.

- As a final step in the initialization process, system software should enable the machine-check exception by setting CR4[MCE] to 1.

A machine-check condition that occurs while CR4[MCE] is cleared will result in the processor core entering the shutdown state.

9.5 Using MCA Features

System software can detect and handle logged errors using three methods:

1. Polling

   Software can periodically examine the machine-check status registers for errors, and save any error information found. Uncorrected errors found during polling will require some type of immediate response to initiate recovery or shutdown.

2. Enabling machine-check reporting

   When reporting is enabled, any uncorrected error that occurs causes control to be transferred to the machine-check exception handler. The exception handler can be designed for a specific processor implementation or can be generalized to work on multiple implementations.

3. Setting up and enabling interrupts for deferred and corrected errors

   In many implementations, MCA hardware can be configured to generate an interrupt hardware on the detection of a deferred error or when a programmed corrected error threshold is reached.

These methods are not mutually exclusive.

[AMD Public Use]

9.5.1 Determining the Scope of Detected Errors

Table 9-4 details the actions that recovery software should take and the level of recovery possible based on status information returned in the MCi_STATUS and MCG_STATUS registers.

Table 9-4. Error Scope

MC*i*_STATUS				Error Scope
PCC	TCC	UC	Deferred	
1	—	1	—	System fatal error. Error has corrupted the processor core architectural state. System processing must be terminated.
0	0	1	—	Recoverable error. If software can correct the error, the interrupted program can be resumed.
0	1	1	—	Containable error. The interrupted instruction stream cannot be resumed. System-level recovery may be possible if software can localize the error and terminate any affected software processes.
0	0	0	1	Deferred error. Immediate software action is not required. A latent error has been discovered, but not yet consumed. Error handling software may attempt to correct this data error, or prevent access by processes which map the data, or make the physical resource containing the data inaccessible.
0	0	0	0	Hardware corrected error. No software action is required. Error information should be saved for analysis.

9.5.2 Handling Machine Check Exceptions

The processor uses the interrupt control-transfer mechanism to invoke an exception handler after a machine-check exception occurs. This requires system software to initialize the interrupt-descriptor table (IDT) with either an interrupt gate or a trap gate that references the interrupt handler. See “Legacy Protected-Mode Interrupt Control Transfers” on page 272 and “Long-Mode Interrupt Control Transfers” on page 283 for more information on interrupt control transfers.

At a minimum, the machine-check exception handler must be capable of logging errors for later examination. This can be a sufficient implementation for some handlers. More thorough exception-handler implementations can analyze the error to determine if it is unrecoverable, and whether it can be recovered in software.

Machine-check exception handlers that attempt recovery must be thorough in their analysis and their corrective actions. The following guidelines should be used when writing such a handler:

- The status registers in all the enabled error-reporting register banks must be examined to identify the cause of the machine-check exception. Read the COUNT field from MCG_CAP to determine the number of status registers supported by the processor.
- Check the valid bit in each status register (MC*i*_STATUS[VAL]). The MC*i*_STATUS register does not need to be examined when its valid bit is clear. Alternatively, read MCA_QUERY to determine valid MC*i*_STATUS registers.

[AMD Public Use]

- Check the valid MC*i*_STATUS registers to see if error recovery is possible. Error recovery is not possible when:
  - The processor-context corrupt bit (MC*i*_STATUS[PCC]) is set to 1.
  - The error-overflow status bit (MC*i*_STATUS[OVER]) is set and the processor does not support recoverable MC*i*_STATUS overflow (as indicated by feature bit CPUID Fn8000_0007_EBX[McaOverflowRecov] = 0).
  - The processor does not support Machine Check Recovery as indicated by feature bit CPUID Fn8000_0007_EBX[SUCCOR].

  If error recovery is not possible, the handler should log the error information and return to the system software responsible for shutting down the processor core.
- Check the MC*i*_STATUS[UC] bit to see if the processor corrected the error. If UC is set, the processor did not correct the error and the exception handler must correct the error before restarting the interrupted program.
  - If MCA Recovery is supported:
    - Check MCA_STATUS[TCC].
    - If TCC is set, the context of the process executing on the interrupted thread may be corrupt and the thread cannot be recovered. The rest of the system is unaffected; it is possible to terminate only the affected process thread.
    - If TCC is clear, the context of the process thread executing on the interrupted logical core is not corrupt. Recovery of the process thread may be possible, but only if the uncorrected error condition is first corrected by software; otherwise, the interrupted process thread must be terminated.

  If the handler cannot correct the error or the MCG_STATUS[RIPV] bit is cleared, it should not return control to the interrupted program, but should log the error information and terminate the software process that was about to consume the uncorrected data. If the error has compromised the state of a guest operating system, the guest should be restarted. If the state of the virtual machine has been corrupted, the virtual machine must be reinitialized.
- When identifying the error condition, portable exception handlers should examine only the architecturally defined fields of the MC*i*_STATUS register.
- If the MCG_STATUS[RIPV] bit is set, the interrupted program can be restarted reliably at the instruction pointer address pushed onto the exception handler stack. If RIPV = 0, the interrupted program cannot be restarted reliably at that location, although it can be restarted at that location for debugging purposes.
- When logging errors, particularly those that are not recoverable, check the MCG_STATUS[EIPV] bit to see if the instruction-pointer address pushed onto the exception handler stack is related to the process thread interrupted by the machine-check exception. If EIPV = 0, the address is not guaranteed to be related to the interrupted process thread.
- Before exiting the machine-check handler, clear the MCG_STATUS[MCIP] bit. MCIP indicates a machine-check exception occurred. If this bit is set when another machine-check exception occurs, the processor enters the shutdown state.

[AMD Public Use]

- When an exception handler is able to, at a minimum, successfully log an error condition, the MC*i*_STATUS registers should be cleared before exiting the machine-check handler. Software is responsible for clearing at least the MC*i*_STATUS[VAL] bits.
- Additional machine-check exception-handler portability can be added by having the handler use the CPUID instruction to identify the processor and its capabilities. Implementation-specific software can be added to the machine-check exception handler based on the processor information reported by CPUID.

9.5.3 Reporting Corrected Errors

Machine-check exceptions do not occur if the error is corrected by the processor. If system software wishes to detect and save information concerning corrected machine-check errors, a system-service routine must be provided to check the contents of the machine-check status registers for corrected errors. The service routine can be invoked by system software on a periodic basis, or by an error-thresholding interrupt.

A service routine that gathers error information for corrected errors should perform the following:

- Examine the status register (MC*i*_STATUS) in each of the enabled error-reporting register banks. For each MC*i*_STATUS register with a set valid bit (VAL=1), the service routine should:
  - Save the contents of the MC*i*_STATUS register.
  - Save the contents of the corresponding MC*i*_ADDR register if MC*i*_STATUS[ADDRV] = 1.
  - Save the contents of the corresponding MC*i*_MISC register if MC*i*_STATUS[MISCV] = 1.
- Once the information found in the error-reporting register banks is saved, the MC*i*_STATUS register should be cleared. This allows the processor to properly report any subsequent errors in the MC*i*_STATUS registers.
- The service routine can save the time-stamp counter with each error logged. This can help in determining how frequently errors occur. For further information, see “Time-Stamp Counter” on page 425.
- In multiprocessor configurations, the service routine can save the processor-node identifier. This can help locate a failing multiprocessor-system component, which can then be isolated from the rest of the system. For further information, see the documentation for particular implementations of the architecture.

[AMD Public Use]

10 System-Management Mode

System-management mode (SMM) is an operating mode designed for system-control activities like power management. Normally, these activities are transparent to conventional operating systems and applications. SMM is used by platform firmware and specialized low-level device drivers, rather than the operating system.

The SMM interrupt-handling mechanism differs substantially from the standard interrupt-handling mechanism described in Chapter 8, “Exceptions and Interrupts.” SMM is entered using a special external interrupt called the *system-management interrupt* (SMI). After an SMI is received by the processor, the processor saves the processor state in a separate address space, called *SMRAM*. The SMM-handler software and data structures are also located in the SMRAM space. Interrupts and exceptions that ordinarily cause control transfers to the operating system are disabled when SMM is entered. The processor exits SMM, restores the saved processor state, and resumes normal execution by using a special instruction, *RSM*.

In SMM, address translation is disabled and addressing is similar to real mode. SMM programs can address up to 4 Gbytes of physical memory. See “SMM Operating-Environment” on page 336 for additional information on memory addressing in SMM.

The following sections describe the components of the SMM mechanism:

- *“SMM Resources” on page 327*—this section describes SMRAM, the SMRAM save-state area used to hold the processor state, and special SMRAM save-state entries used in support of SMM.
- *“Using SMM” on page 336*—this section describes the mechanism of entering and exiting SMM. It also describes SMM memory allocation, addressing, and interrupts and exceptions.

Of these mechanisms, only the format of the SMRAM save-state area differs between the AMD64 architecture and the legacy architecture.

Note: *Model-independent aspects of SMM operation are described here; see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual (PPR) of a given processor family for possible model-specific details.*

10.1 SMM Differences

There are functional differences between the SMM support in the AMD64 architecture and the SMM support found in previous architectures. These are:

- The SMRAM state-save area layout is changed to hold the 64-bit processor state.
- The initial processor state upon entering SMM is expanded to reflect the 64-bit nature of the processor.
- New conditions exist that can cause a processor shutdown while in SMM.

[AMD Public Use]

- The auto-halt restart and I/O-instruction restart entries in the SMRAM state-save area are one byte each instead of two bytes each.
- SMRAM caching considerations are modified because the legacy FLUSH# external signal (writeback, if modified, and invalidate) is not supported on implementations of the AMD64 architecture.
- Some previous AMD x86 processors saved and restored the CR2 register in the SMRAM state-save area. This register is not saved by the SMM implementation in the AMD64 architecture. SMM handlers that save and restore CR2 must perform the operation in software.

10.2 SMM Resources

The SMM resources supported by the processor consist of SMRAM, the SMRAM state-save area, and special entries within the SMRAM state-save area. In addition to the save-state area, SMRAM includes space for the SMM handler.

10.2.1 SMRAM

SMRAM is the memory-address space accessed by the processor when in SMM. The default size of SMRAM is 64 Kbytes and can range in size between 32 Kbytes and 4 Gbytes. System logic can use physically separate SMRAM and main memory, directing memory transactions to SMRAM after recognizing SMM is entered, and redirecting memory transactions back to system memory after recognizing SMM is exited. When separate SMRAM and main memory are used, the system designer needs to provide a method of mapping SMRAM into main memory so that the SMI handler and data structures can be loaded.

Figure 10-1 on page 328 shows the default SMRAM memory map. The default SMRAM code segment (CS) has a base address of 0003_0000h (the base address is automatically scaled by the processor using the CS-selector register, which is set to the value 3000h). This default SMRAM-base address is known as *SMBASE*. A 64-Kbyte memory region, addressed from 0003_0000h to 0003_FFFFh, makes up the default SMRAM memory space. The top 32 Kbytes (0003_8000h to 0003_FFFFh) must be supported by system logic, with physical memory covering that entire address range. The top 512 bytes (0003_FE00h to 0003_FFFFh) of this address range are the default *SMM state-save area*. The default entry point for the SMM interrupt handler is located at 0003_8000h.

[AMD Public Use]

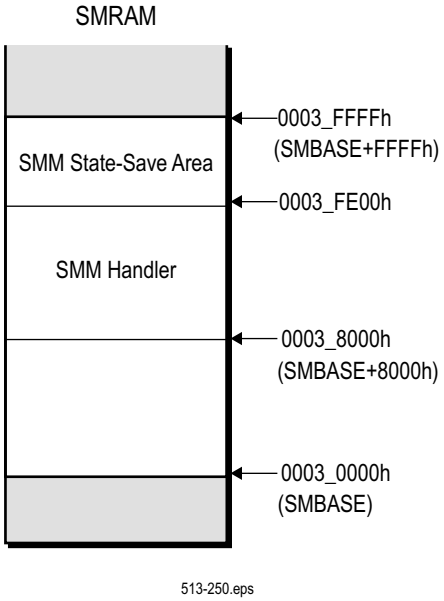


Figure 10-1. Default SMRAM Memory Map

10.2.2 SMBASE Register

The format of the SMBASE register is shown in Figure 10-2. SMBASE is an internal processor register that holds the value of the SMRAM-base address. SMBASE is set to 30000h after a processor reset.

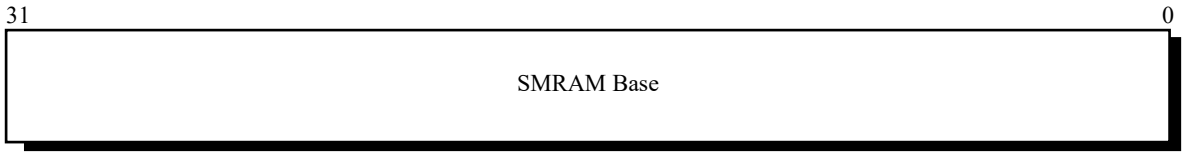


Figure 10-2. SMBASE Register

In some operating environments, relocation of SMRAM to a higher memory area can provide more low memory for legacy software. SMBASE relocation is supported when the SMM-base relocation bit in the SMM-revision identifier (bit 17) is set to 1. In processors implementing the AMD64 architecture, SMBASE relocation is always supported.

Software can only modify SMBASE (relocate the SMRAM-base address) by entering SMM, modifying the SMBASE image stored in the SMRAM state-save area, and exiting SMM. The SMM-handler entry point must be loaded at the new memory location specified by SMBASE+8000h. The next time SMM is entered, the processor saves its state in the new state-save area at SMBASE+0FE00h, and begins executing the SMM handler at SMBASE+8000h. The new SMBASE address is used for every SMM until it is changed, or a hardware reset occurs.

[AMD Public Use]

When SMBASE is used to relocate SMRAM to an address above 1 Mbyte, 32-bit address-size override prefixes must be used to access this memory. This is because addressing in SMM behaves as it does in real mode, with a 16-bit default operand size and address size. The values in the 16-bit segment-selector registers are left-shifted four bits to form a 20-bit segment-base address. Without using address-size overrides, the maximum computable address is 10FFEFh.

Because SMM memory-addressing is similar to real-mode addressing, the SMBASE address must be less than 4 Gbytes.

10.2.3 SMRAM State-Save Area

When an SMI occurs, the processor saves its state in the 512-byte SMRAM state-save area during the control transfer into SMM. The format of the state-save area defined by the AMD64 architecture is shown in Table 10-1. This table shows the offsets in the SMRAM state-save area relative to the SMRAM-base address. The state-save area is located between offset 0_FE00h (SMBASE+0_FE00h) and offset 0_FFFFh (SMBASE+0_FFFFh). Software should not modify offsets specified as read-only or reserved, otherwise unpredictable results can occur.

Table 10-1. AMD64 Architecture SMM State-Save Area

Offset (Hex) from SMBASE	Contents		Size	Allowable Access
FE00h	ES	Selector	Word	Read-Only
FE02h		Attributes	Word	
FE04h		Limit	Doubleword	
FE08h		Base	Quadword	
FE10h	CS	Selector	Word	Read-Only
FE12h		Attributes	Word	
FE14h		Limit	Doubleword	
FE18h		Base	Quadword	
FE20h	SS	Selector	Word	Read-Only
FE22h		Attributes	Word	
FE24h		Limit	Doubleword	
FE28h		Base	Quadword	
FE30h	DS	Selector	Word	Read-Only
FE32h		Attributes	Word	
FE34h		Limit	Doubleword	
FE38h		Base	Quadword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

[AMD Public Use]

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents		Size	Allowable Access
FE40h	FS	Selector	Word	Read-Only
FE42h		Attributes	Word	
FE44h		Limit	Doubleword	
FE48h		Base	Quadword	
FE50h	GS	Selector	Word	Read-Only
FE52h		Attributes	Word	
FE54h		Limit	Doubleword	
FE58h		Base	Quadword	
FE60h–FE63h	GDTR	Reserved	4 Bytes	Read-Only
FE64h		Limit	Word	
FE66h–FE67h		Reserved	2 Bytes	
FE68h		Base	Quadword	
FE70h	LDTR	Selector	Word	Read-Only
FE72h		Attributes	Word	
FE74h		Limit	Doubleword	
FE78h		Base	Quadword	
FE80h–FE83h	IDTR	Reserved	4 Bytes	Read-Only
FE84h		Limit	Word	
FE86h–FE87h		Reserved	2 Bytes	
FE88h		Base	Quadword	
FE90h	TR	Selector	Word	Read-Only
FE92h		Attributes	Word	
FE94h		Limit	Doubleword	
FE98h		Base	Quadword	
FEA0h	I/O Instruction Restart RIP		Quadword	Read-Only
FEA8h	I/O Instruction Restart RCX		Quadword	Read-Only
FEB0h	I/O Instruction Restart RSI		Quadword	Read-Only
FEB8h	I/O Instruction Restart RDI		Quadword	Read-Only
FEC0h	I/O Instruction Restart Dword		Doubleword	Read-Only
FEC4h–FEC7h	Reserved		4 Bytes	—
FEC8h	I/O Instruction Restart		Byte	Read/Write
FEC9h	Auto-Halt Restart		Byte	
FECAh—FECFh	Reserved		6 Bytes	—
FED0h	EFER		Quadword	Read-Only

Note:

1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

[AMD Public Use]

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FED8h	SVM Guest	Quadword	Read-Only
FEE0h	SVM Guest VMCB Physical Address	Quadword	
FEE8h	SVM Guest Virtual Interrupt	Quadword	
FEF0h—FEFBh	Reserved	12 Bytes	—
FEFCh	SMM-Revision Identifier[1]	Doubleword	Read-Only
FF00h	SMBASE	Doubleword	Read/Write
FF04h—FF17h	Reserved	20 Bytes	—
FF18h	SSP	Quadword	Read/Write
FF20h	SVM Guest PAT	Quadword	Read-Only
FF28h	SVM Host EFER	Quadword	
FF30h	SVM Host CR4	Quadword	
FF38h	SVM Host CR3	Quadword	
FF40h	SVM Host CR0	Quadword	
FF48h	CR4	Quadword	Read-Only
FF50h	CR3	Quadword	
FF58h	CR0	Quadword	
FF60h	DR7	Quadword	Read-Only
FF68h	DR6	Quadword	
FF70h	RFLAGS	Quadword	Read/Write
FF78h	RIP	Quadword	Read/Write
FF80h	R15	Quadword	
FF88h	R14	Quadword	
FF90h	R13	Quadword	
FF98h	R12	Quadword	
FFA0h	R11	Quadword	
FFA8h	R10	Quadword	
FFB0h	R9	Quadword	
FFB8h	R8	Quadword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

[AMD Public Use]

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FFC0h	RDI	Quadword	Read/Write
FFC8h	RSI	Quadword	
FFD0h	RBP	Quadword	
FFD8h	RSP	Quadword	
FFE0h	RBX	Quadword	
FFE8h	RDX	Quadword	
FFF0h	RCX	Quadword	
FFF8h	RAX	Quadword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

A number of other registers are not saved or restored automatically by the SMM mechanism. See "Saving Additional Processor State" on page 339 for information on using these registers in SMM.

As a reference for legacy processor implementations, the legacy SMM state-save area format is shown in Table 10-2. *Implementations of the AMD64 architecture do not use this format.*

Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FE00h—FEF7h	Reserved	248 Bytes	—
FEF8h	SMBASE	Doubleword	Read/Write
FEFCh	SMM-Revision Identifier	Doubleword	Read-Only
FF00h	I/O Instruction Restart	Word	Read/Write
FF02h	Auto-Halt Restart	Word	
FF04h—FF87h	Reserved	132 Bytes	—
FF88h	GDT Base	Doubleword	Read-Only
FF8Ch—FF93h	Reserved	Quadword	—
FF94h	IDT Base	Doubleword	Read-Only
FF98h—FFA7h	Reserved	16 Bytes	—

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

[AMD Public Use]

Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture) (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FFA8h	ES	Doubleword	Read-Only
FFACh	CS	Doubleword	
FFB0h	SS	Doubleword	
FFB4h	DS	Doubleword	
FFB8h	FS	Doubleword	
FFBCh	GS	Doubleword	
FFC0h	LDT Base	Doubleword	Read-Only
FFC4h	TR	Doubleword	
FFC8h	DR7	Doubleword	Read-Only
FFCCh	DR6	Doubleword	
FFD0h	EAX	Doubleword	Read/Write
FFD4h	ECX	Doubleword	
FFD8h	EDX	Doubleword	
FFDCh	EBX	Doubleword	
FFE0h	ESP	Doubleword	
FFE4h	EBP	Doubleword	
FFE8h	ESI	Doubleword	
FFECh	EDI	Doubleword	
FFF0h	EIP	Doubleword	Read/Write
FFF4h	EFLAGS	Doubleword	Read/Write
FFF8h	CR3	Doubleword	Read-Only
FFFCh	CR0	Doubleword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

10.2.4 SMM-Revision Identifier

The SMM-revision identifier specifies the SMM version and the available SMM extensions implemented by the processor. Software reads the SMM-revision identifier from offset FEFCh in the SMM state-save area of SMRAM. This offset location is compatible with earlier versions of SMM. Software must not write to this location. Doing so can produce undefined results. Figure 10-3 on page 334 shows the format of the SMM-revision identifier.

[AMD Public Use]

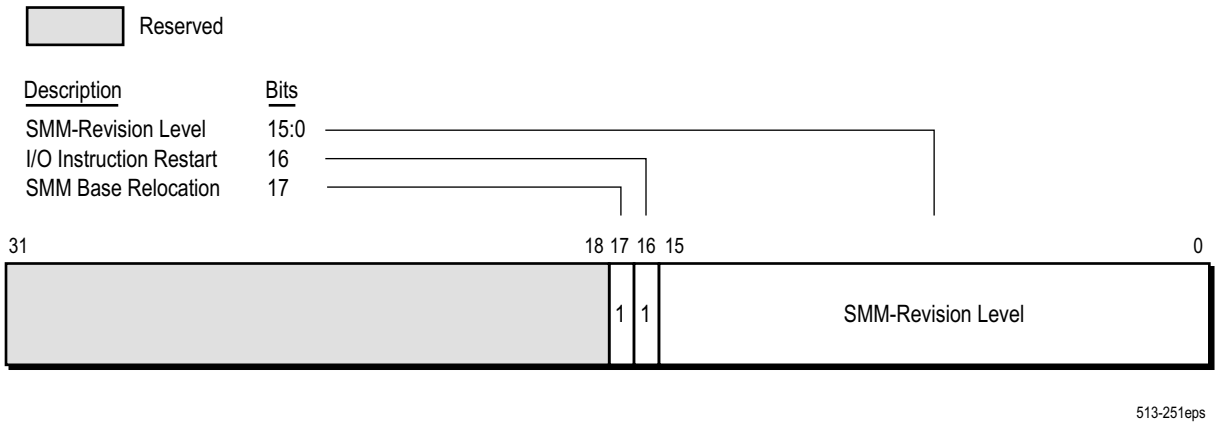


Figure 10-3. SMM-Revision Identifier

The fields within the SMM-revision identifier are:

- *SMM-revision Level*—Bits 15:0. Specifies the version of SMM supported by the processor. The SMM-revision level is of the form 0_xx64h, where *xx* starts with 00 and is incremented for later revisions to the SMM mechanism.
- *I/O Instruction Restart*—Bit 16. When set to 1, the processor supports restarting I/O instructions that are interrupted by an SMI. This bit is always set to 1 by implementations of the AMD64 architecture. See “I/O Instruction Restart” on page 340 for information on using this feature.
- *SMM Base Relocation*—Bit 17. When set to 1, the processor supports relocation of SMRAM. This bit is always set to 1 by implementations of the AMD64 architecture. See “SMBASE Register” on page 328 for information on using this feature.

All remaining bits in the SMM-revision identifier are reserved.

10.2.5 SMRAM Protected Areas

Two areas are provided as safe areas for SMM code and data that are not readily accessible by non-SMM applications. The SMI handler can be located in one of these two ranges, or it can be located outside of these ranges. The handler is placed in the desired range by setting SMBASE accordingly.

The ASeg range is located at a fixed address from A_0000h to B_FFFFh. The TSeg range is located at a variable base specified by the SMM_ADDR MSR with a variable size specified by the SMM_MASK MSR. These ranges must never overlap.

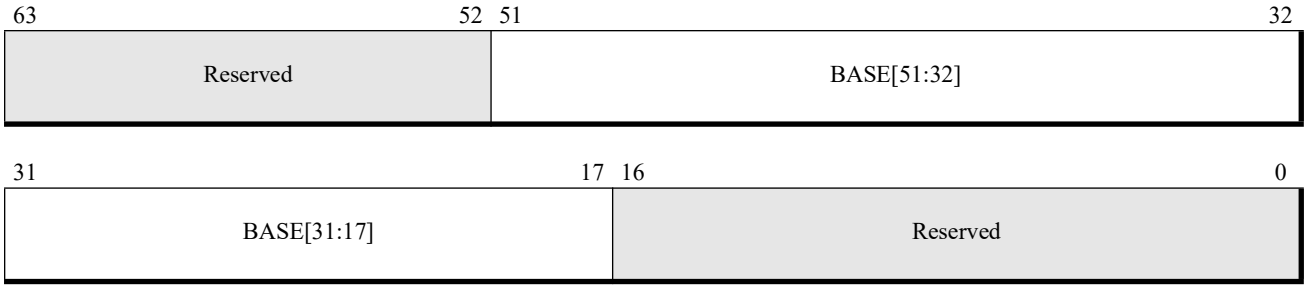
Each CPU memory access is in the TSeg range if the following is true:

Phys Addr[51:17] & SMM_MASK[51:17] = SMM_ADDR[51:17] & SMM_MASK[51:17].

For example, if the TSeg range spans 256 Kbytes starting at address 10_0000h, then SMM_ADDR =0010_0000h and SMM_MASK=FFFC_0000h. This results in a TSeg address range from 0010_0000

[AMD Public Use]

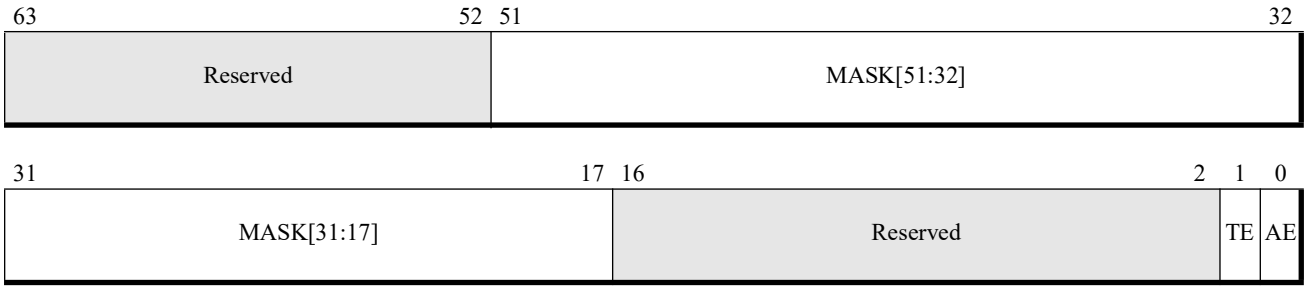
to 0013_FFFFh. The TSeg range must be aligned to a 128 Kbyte boundary and the minimum TSeg size is 128 Kbytes.



Bits	Mnemonic	Description	Access type
63:52	Reserved		RAZ
51:17	BASE	SMM TSeg Base Address	R/W
16:0	Reserved		RAZ

Figure 10-4. SMM_ADDR Register Format

- *SMM TSeg Base Address (BASE)*—Bits 51:17. Specifies the base address of the TSeg range of protected addresses.



Bits	Mnemonic	Description	Access type
63:52	Reserved		RAZ
51:17	MASK	TSeg Mask	R/W
16:2	Reserved		RAZ
1	TE	Tseg Address Range Enable	R/W
0	AE	Aseg Address Range Enable	R/W

Figure 10-5. SMM_MASK Register Format

- *ASeg Address Range Enable (AE)*—Bit 0. Specifies whether the ASeg address range is enabled for protection. When the bit is set to 1, the ASeg address range is enabled for protection. When cleared to 0, the ASeg address range is disabled for protection.

[AMD Public Use]

- *TSeg Address Range Enable (TE)*—Bit 1. Specifies whether the TSeg address range is enabled for protection. When the bit is set to 1, the TSeg address range is enabled for protection. When cleared to 0, the TSeg address range is disabled for protection.
- *TSeg Mask (MASK)*—Bits 51:17. Specifies the mask used to determine the TSeg range of protected addresses. The physical address is in the TSeg range if the following is true:

  Phys Addr[51:17] & SMM_MASK[51:17] = SMM_ADDR[51:17] & SMM_MASK[51:17].

Note that a processor is not required to implement all 52 bits of the physical address.

10.3 Using SMM

10.3.1 System-Management Interrupt (SMI)

SMM is entered using the system-management interrupt (SMI). SMI is an external non-maskable interrupt that operates differently from and independently of other interrupts. SMI has priority over all other external interrupts, including NMI (see “Priorities” on page 266 for a list of the interrupt priorities). SMIs are disabled when in SMM, which prevents reentrant calls to the SMM handler.

When an SMI is received by the processor, the processor stops fetching instructions and waits for currently-executing instructions to complete and write their results. The SMI also waits for all buffered memory writes to update the caches or system memory. When these activities are complete, the processor uses implementation-dependent external signaling to acknowledge back to the system that it has received the SMI.

10.3.2 SMM Operating-Environment

The SMM operating-environment is similar to real mode, except that the segment limits in SMM are 4 Gbytes rather than 64 Kbytes. This allows an SMM handler to address memory in the range from 0h to 0FFFF_FFFFh. As with real mode, segment-base addresses are restricted to 20 bits in SMM, and the default operand-size and address-size is 16 bits. To address memory locations above 1 Mbyte, the SMM handler must use the 32-bit operand-size-override and address-size-override prefixes.

After saving the processor state in the SMRAM state-save area, a processor running in SMM sets the segment-selector registers and control registers into a state consistent with real mode. Other registers are also initialized upon entering SMM, as shown in Table 10-3.

Table 10-3. SMM Register Initialization

Register		Initial SMM Contents
CS	Selector	SMBASE right-shifted 4 bits
	Base	SMBASE
	Limit	FFFF_FFFFh
	Attr	Read-Write-Execute

[AMD Public Use]

Table 10-3. SMM Register Initialization (continued)

Register		Initial SMM Contents
DS, ES, FS, GS, SS	Selector	0000h
	Base	0000_0000_0000_0000h
	Limit	FFFF_FFFFh
	Attr	Read-Write
RIP		0000_0000_0000_8000h
RFLAGS		0000_0000_0000_0002h
CR0		PE, EM, TS, PG bits cleared to 0. All other bits are unmodified.
CR4		0000_0000_0000_0000h
DR7		0000_0000_0000_0400h
EFER		0000_0000_0000_0000h

10.3.3 Exceptions and Interrupts

All hardware interrupts are disabled upon entering SMM, but exceptions and software interrupts are not disabled. If necessary, the SMM handler can re-enable hardware interrupts. Software that handles interrupts in SMM should consider the following:

- *SMI*—If an SMI occurs while the processor is in SMM, it is latched by the processor. The latched SMI occurs when the processor leaves SMM.
- *NMI*—If an NMI occurs while the processor is in SMM, it is latched by the processor, but the NMI handler is not invoked until the processor leaves SMM with the execution of an RSM instruction. A pending NMI causes the handler to be invoked immediately after the RSM completes and before the first instruction in the interrupted program is executed.

  An SMM handler can unmask NMI interrupts by simply executing an IRET. Upon completion of the IRET instruction, the processor recognizes the pending NMI, and transfers control to the NMI handler. Once an NMI is recognized within SMM using this technique, subsequent NMIs are recognized until SMM is exited. Later SMIs cause NMIs to be masked, until the SMM handler unmasks them.
- *Exceptions*—Exceptions (internal processor interrupts) are not disabled and can occur while in SMM. Therefore, the SMM-handler software should be written to avoid generating exceptions.
- *Software Interrupts*—The software-interrupt instructions (BOUND, INT*n*, INT3, and INTO) can be executed while in SMM. However, it is not recommended that the SMM handler use these instructions.
- *Maskable Interrupts*—RFLAGS.IF is cleared to 0 by the processor when SMM is entered. Software can re-enable maskable interrupts while in SMM, but it must follow the guidelines listed below for handling interrupts.

[AMD Public Use]

- *Debug Interrupts*—The processor disables the debug interrupts when SMM is entered by clearing DR7 to 0 and clearing RFLAGS.TF to 0. The SMM handler can re-enable the debug facilities while in SMM, but it must follow the guidelines listed below for handling interrupts.
- *INIT*—The processor does not recognize INIT while in SMM.

Because the RFLAGS.IF bit is cleared when entering SMM, the HLT instruction should not be executed in SMM without first setting the RFLAGS.IF bit to 1. Setting this bit to 1 allows the processor to exit the halt state by using an external maskable interrupt.

In the cases where an SMM handler must accept and handle interrupts and exceptions, several guidelines must be followed:

- Interrupt handlers must be loaded and accessible before enabling interrupts.
- A real-mode interrupt vector table located at virtual (linear) address 0 is required.
- Segments accessed by the interrupt handler cannot have a base address greater than 20 bits because of the real-mode addressing used in SMM. In SMM, the 16-bit value stored in the segment-selector register is left-shifted four bits to form the 20-bit segment-base address, like real mode.
- Only the IP (rIP[15:0]) is pushed onto the stack as a result of an interrupt in SMM, because of the real-mode addressing used in SMM. If the SMM handler is interrupted at a code-segment offset above 64 Kbytes, then the return address on the stack must be adjusted by the interrupt-handler, and a RET instruction with a 32-bit operand-size override must be used to return to the SMM handler.
- If the interrupt-handler is located below 1 Mbyte, and the SMM handler is located above 1 Mbyte, a RET instruction cannot be used to return to the SMM handler. In this case, the interrupt handler can adjust the return pointer on the stack, and use a far CALL to transfer control back to the SMM handler.

10.3.4 Invalidating the Caches

The processor can cache SMRAM-memory locations. If the system implements physically separate SMRAM and system memory, it is possible for SMRAM and system memory locations to alias into identical cache locations. In some processor implementations, the cache contents must be written to memory and invalidated when SMM is entered *and* exited. This prevents the processor from using previously-cached main-memory locations as aliases for SMRAM-memory locations when SMM is entered, and vice-versa when SMM is exited.

Implementations of the AMD64 architecture *do not require cache invalidation* when entering and exiting SMM. Internally, the processor keeps track of SMRAM and system-memory accesses separately and properly handles situations where aliasing occurs. Cached system memory and SMRAM locations can persist across SMM mode changes. Removal of the requirement to writeback and invalidate the cache simplifies SMM entry and exit and allows SMM code to execute more rapidly.

[AMD Public Use]

10.3.5 Saving Additional Processor State

Several registers are not saved or restored automatically by the SMM mechanism. These are:

- The 128-bit media instruction registers.
- The 64-bit media instruction registers.
- The x87 floating-point registers.
- The page-fault linear-address register (CR2).
- The task-priority register (CR8).
- The debug registers, DR0, DR1, DR2, and DR3.
- The memory-type range registers (MTRRs).
- Model-specific registers (MSRs).

These registers are not saved because SMM handlers do not normally use or modify them. If an SMI results in a processor reset (due to powering down the processor, for example) or the SMM handler modifies the contents of the unsaved registers, the SMM handler should save and restore the original contents of those registers. The unsaved registers, along with those stored in the SMRAM state-save area, need to be saved in a non-volatile storage location if a processor reset occurs. The SMM handler should execute the CPUID instruction to determine the feature set available in the processor, and be able to save and restore the registers required by those features. For more information on using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 73.

The SMM handler can execute any of the 128-bit media, 64-bit media, or x87 instructions. A simple method for saving and restoring those registers is to use the FXSAVE and FXRSTOR instructions, respectively, if it is supported by the processor. See “Saving Media and x87 Execution Unit State” on page 351 for information on saving and restoring those registers.

Floating-point exceptions can occur when the SMM handler uses media or x87 floating-point instructions. If the SMM handler uses floating-point exception handlers, they must follow the usage guidelines established in “Exceptions and Interrupts” on page 337. A simple method for dealing with floating-point exceptions while in SMM is to simply mask all exception conditions using the appropriate floating-point control register. When the exceptions are masked, the processor handles floating-point exceptions internally in a default manner, and allows execution to continue uninterrupted.

10.3.6 Operating in Protected Mode and Long Mode

Software can enable protected mode from SMM and it can also enable and activate long mode. An SMM handler can use this capability to enter 64-bit mode and save additional processor state that cannot be accessed from outside 64-bit mode (for example, the most-significant 32 bits of CR2).

10.3.7 Auto-Halt Restart

The auto-halt restart entry is located at offset FEC9h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM.

[AMD Public Use]

When entering SMM, the processor loads the auto-halt restart entry to indicate whether SMM was entered from the halt state, as follows:

- Bit 0 indicates the processor state upon entering SMM:
  - When set to 1, the processor entered SMM from the halt state.
  - When cleared to 0, the processor did not enter SMM from the halt state.
- Bits 7:1 are cleared to 0.

The SMM handler can write the auto-halt restart entry to specify whether the return from SMM should take the processor back to the halt state or to the instruction-execution state specified by the SMM state-save area. The values written are:

- *Clear to 00h*—The processor returns to the state specified by the SMM state-save area.
- *Set to any non-zero value*—The processor returns to the halt state.

If the return from SMM takes the processor back to the halt state, the HLT instruction is not re-executed. However, the halt special bus-cycle is driven on the processor bus after the RSM instruction executes.

The result of entering SMM from a non-halt state and returning to a halt state is not predictable.

10.3.8 I/O Instruction Restart

The I/O-instruction restart entry is located at offset FEC8h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM. The I/O-instruction restart mechanism is supported when the I/O-instruction restart bit (bit 16) in the SMM-revision identifier is set to 1. This bit is always set to 1 in the AMD64 architecture.

When an I/O instruction is interrupted by an SMI, the I/O-instruction restart entry specifies whether the interrupted I/O instruction should be re-executed following an RSM that returns from SMM. Re-executing a trapped I/O instruction is useful, for example, when an I/O write is performed to a powered-down disk drive. When this occurs, the system logic monitoring the access can issue an SMI to have the SMM handler power-up the disk drive and retry the I/O write. The SMM handler does this by querying system logic and detecting the failed I/O write, asking system logic to initiate the disk-drive power-up sequence, enabling the I/O instruction restart mechanism, and returning from SMM. Upon returning from SMM, the I/O write to the disk drive is restarted.

When an SMI occurs, the processor always clears the I/O-instruction restart entry to 0. If the SMI interrupted an I/O instruction, then the SMM handler can modify the I/O-instruction restart entry as follows:

- *Clear to 00h (default value)*—The I/O instruction is not restarted, and the instruction following the interrupted I/O-instruction is executed. When a REP (repeat) prefix is used with an I/O instruction, it is possible that the next instruction to be executed is the next I/O instruction in the repeat loop.
- *Set to any non-zero value*—The I/O instruction is restarted.

[AMD Public Use]

While in SMM, the handler must determine the cause of the SMI and examine the processor state at the time the SMI occurred to determine whether or not an I/O instruction was interrupted. Implementations provide state information in the SMM save-state area to assist in this determination:

- I/O Instruction Restart DWORD—indicates whether the SMI interrupted an I/O instruction, and saves extra information describing the I/O instruction.
- I/O Instruction Restart RIP—the RIP of the interrupted I/O instruction.
- I/O Instruction Restart RCX—the RCX of the interrupted I/O instruction.
- I/O Instruction Restart RSI—the RSI of the interrupted I/O instruction.
- I/O Instruction Restart RDI—the RDI of the interrupted I/O instruction.

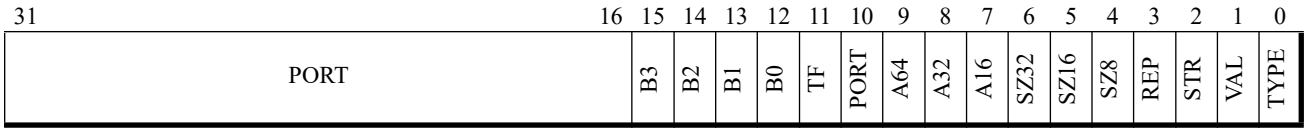


Figure 10-6. I/O Instruction Restart Dword

The fields in the I/O Instruction Restart DWORD are as follows:

- B3—DR6.B3 status
- B2—DR6.B2 status
- B1—DR6.B1 status
- B0 —DR6.B0 status
- TF—The I/O instruction was interrupted with Single Stepping (EFLAGS.TF = 1)
- PORT—Intercepted I/O port
- A64—64-bit address
- A32—32-bit address
- A16—16-bit address
- SZ32—32-bit I/O port size
- SZ16—16-bit I/O port size
- SZ8—8-bit I/O port size
- REP—Repeated port access
- STR—String based port access (INS, OUTS)
- VAL—Valid (SMI was detected during an I/O instruction.)
- TYPE—Access type (0 = OUT instruction, 1 = IN instruction).

10.3.9 SMM Page Configuration Lock

The SMM Page Configuration Lock feature allows the SMM handler to lock the paging configuration. The feature is enabled by setting HWCR.SMM_PGCFG_LOCK=1 (bit 33). Once locked, the paging

[AMD Public Use]

configuration cannot be modified until SMM is exited using the RSM instruction. The processor clears HWCR.SMM_PGCFG_LOCK when completing the RSM instruction. If page configuration locking is needed when the processor enters SMM again in the future, HWCR.SMM_PGCFG_LOCK must be set again by the SMM handler.

When SMM Page Configuration Lock is enabled, the following will result in a #GP exception:

- Writing Extended Feature Register (EFER) using the WRMSR instruction.
- Writing CR0, CR3, or CR4 using the MOV CRn instruction.

Attempting to set HWCR.SMM_PGCFG_LOCK when not in SMM results in a #GP exception. Before setting HWCR.SMM_PGCFG_LOCK, system software must verify the processor supports the SMM Page Configuration Lock feature by checking that CPUID Fn8000_0021_EAX[SmmPgCfgLock] (bit 3) = 1. For more information on using the CPUID instruction see Section 3.3, “Processor Feature Identification,” on page 73.

10.4 Leaving SMM

Software leaves SMM and returns to the interrupted program by executing the RSM instruction. RSM causes the processor to load the interrupted state from the SMRAM state-save area and then transfer control back to the interrupted program. RSM cannot be executed in any mode other than SMM, otherwise an invalid-opcode exception (#UD) occurs.

An RSM causes a processor shutdown if an invalid-state condition is found in the SMRAM state-save area. Only an external reset, external processor-initialization, or non-maskable external interrupt (NMI) can cause the processor to leave the shutdown state. The invalid SMRAM state-save-area conditions that can cause a processor shutdown during an RSM are:

- CR0.PE=0 and CR0.PG=1.
- CR0.CD=0 and CR0.NW=1.
- CR0.WP=0 and CR4.CET=1.
- Certain reserved bits are set to 1, including:
  - Any CR0 bit in the range 63:32 is set to 1.
  - Any unsupported bit in CR3 is set to 1.
  - Any unsupported bit in CR4 is set to 1.
  - Any DR6 bit or DR7 bit in the range 63:32 is set to 1.
  - Any unsupported bit in EFER is set to 1.
- Invalid returns to long mode, including:
  - EFER.LME=1, CR0.PG=1, and CR4.PAE=0.
  - EFER.LME=1, CR0.PG=1, CR4.PAE=1, CS.L=1, and CS.D=1.
- Invalid return to Virtual-8086 Mode, including:
  - rFLAGS.VM=1 and CR4.CET=1 and U_CET.SS=1.

[AMD Public Use]

- Invalid return to Process Context Identifiers (PCID) enabled outside of Long Mode:
  - CR4.PCIDE=1 and (EFER.LME=0 or CR0.PG=0 or CR4.PAE=0).
- The SSM revision identifier is modified.

Some SMRAM state-save-area conditions are ignored, and the registers, or bits within the registers, are restored in a default manner by the processor. This avoids a processor shutdown when an invalid condition is stored in SMRAM. The default conditions restored by the processor are:

- CPL is forced to 0 if the SMRAM value of CR0.PE is zero and forced to 3 if the SMRAM value of RFLAGS.VM is one.
- The EFER.LMA register bit is set to the value obtained by logically ANDing the SMRAM values of EFER.LME, CR0.PG, and CR4.PAE.
- The RFLAGS.VM register bit is set to the value obtained by logically ANDing the SMRAM values of RFLAGS.VM, CR0.PE, and the inverse of EFER.LMA.
- The base values of FS, GS, GDTR, IDTR, LDTR, and TR are restored in canonical form. Those values are sign-extended to bit 63 using the most-significant implemented bit.
- Unimplemented segment-base bits in the CS, DS, ES, and SS registers are cleared to 0.
- SSP is canonicalized (i.e., sign-extended to bit 63).

10.5 Multiprocessor Considerations

For multiprocessor operation, each logical processor must be given a separate SMBASE value so that the save-state areas do not overlap. For systems with fewer than 64 logical processors it is sufficient to stagger the SMBASE values by 512 bytes. Note that this also offsets the SMI entry point by the same amount for each processor. With 64 or more logical processors, the entry points will start to collide with the save-state areas. Staggering the SMBASE values by 1024 bytes results in 512-byte entry point areas interleaved with the 512-byte state-save areas, and so provides scaling beyond 63 logical processors.

Further details on multiprocessor aspects of SMM may be found in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* for a given processor family.

[AMD Public Use]

11 SSE, MMX, and x87 Programming

This chapter describes the system-software implications of supporting applications that use the Streaming SIMD Extensions (SSE), MMX™, and x87 instructions. Throughout this chapter, these instructions are collectively referred to as *media and x87* (media/x87) instructions. A complete listing of the instructions that fall in this category—and the detailed operation of each instruction—can be found in Volumes 4 and 5. Refer to Volume 1 for information on using these instructions in application software.

The SSE instruction set is comprised of the *legacy SSE* instruction set which includes the SSE1, SSE2, SSE3, SSSE3, SSE4A, SSE4.1, and SSE4.2 subsets and the *extended SSE* instruction set which includes the AVX, FMA4, XOP, and AVX512 subsets. Many of the extended SSE instructions support both 128-bit and 256-bit data types.

11.1 Overview of System-Software Considerations

Processor implementations can support different combinations of the SSE, MMX, and x87 instruction sets. Two sets of registers—independent of the general-purpose registers—support these instructions. The SSE instructions operate on the ZMM/YMM/XMM registers, and the 64-bit media and x87-instructions operate on the aliased MMX/x87 registers. The SSE and x87 floating-point instruction sets have distinct status registers, control registers, exception vectors, and system-software control bits for managing the operating environment. System software that supports use of these instructions must be able to manage these resources properly including:

- Detecting support for the instruction set, and enabling any optional features, as necessary.
- Saving and restoring the processor media or x87 state.
- Execution of floating-point instructions (media or x87) can produce exceptions. System software must supply exception handlers for all unmasked floating-point exceptions.

11.2 Determining Media and x87 Feature Support

Support for the architecturally defined subsets within the media and x87 instructions is implementation dependent. System software executes the CPUID instruction to determine whether a processor implements any of these features (see Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction). After CPUID is executed feature support can be determined by examining specific bit fields returned in the EAX, ECX, and EDX registers.

The following table summarizes the architecturally defined SSE subsets and state management instructions and gives the feature bits returned by the CPUID function. If the indicated bit is set, the feature is supported by the processor.

[AMD Public Use]

Table 11-5. SSE Subsets – CPUID Feature Identifiers

CPUID Fn	Field Name	Field Bit	Instruction Subset
Legacy SSE			
0000_0001h	EDX[SSE]	EDX[25]	Original Streaming SIMD Extensions (SSE1)
0000_0001h	EDX[SSE2]	EDX[26]	SSE2
0000_0001h	ECX[SSE3]	ECX[0]	SSE3
0000_0001h	ECX[SSSE3]	ECX[9]	SSSE3
0000_0001h	ECX[SSE41]	ECX[19]	SSE4.1
0000_0001h	ECX[SSE42]	ECX[20]	SSE4.2
8000_0001h	ECX[SSE4A]	ECX[6]	SSE4A: EXTRQ, INSERTQ, MOVNTSS, and MOVNTSD instructions
Extended SSE			
0000_0001h	ECX[AVX]	ECX[28]	AVX
8000_0001h	ECX[XOP]	ECX[11]	AMD XOP
0000_0001h	ECX[FMA]	ECX[12]	FMA
8000_0001h	ECX[FMA4]	ECX[16]	AMD FMA4
MMX			
0000_0001h or 8000_0001h	EDX[MMX]	EDX[23]	Original MMX™ Instructions
8000_0001h	EDX[MmxExt]	EDX[22]	AMD Extensions to MMX
8000_0001h	EDX[3DNow]	EDX[31]	AMD 3DNow!™
8000_0001h	EDX[3DNowExt]	EDX[30]	AMD Extensions to 3DNow!
x87			
0000_0001h or 8000_0001h	EDX[FPU]	EDX[0]	x87 instruction set and facilities
Context Management Instructions			
0000_0001h or 8000_0001h	EDX[FXSR]	EDX[24]	FXSAVE / FXRSTOR instructions
8000_0001h	EDX[FFXSR]	EDX[25]	Hardware optimizations for FXSAVE / FXRSTOR
0000_0001h	ECX[XSAVE]	ECX[26]	XSAVE / XRSTOR instructions
0000_000Dh ECX=01h	EAX[XSAVEOPT]	EAX[0]	XSAVEOPT

Some instructions may be listed in more than one subset. If software attempts to execute an instruction belonging to an unsupported instruction subset, an invalid-opcode exception (#UD) occurs. Refer to Appendix D, “Instruction Subsets and CPUID Feature Flags” in Volume 3 for specific information.

[AMD Public Use]

11.3 Enabling SSE Instructions

Use of the 512-bit, 256-bit, and 128-bit media instructions by application software requires system software support. System software must determine which SSE subsets are supported, enable those that are to be used, and supply code to handle the various exceptions that may occur during the execution of these instructions. The legacy SSE instructions and the extended SSE instructions often require unique exception handling.

11.3.1 Enabling Legacy SSE Instruction Execution

When legacy SSE instructions are supported, system software must set CR4.OSFXSR to let the processor know that the software supports the FXSAVE/FXRSTOR instructions. When the processor detects CR4.OSFXSR = 1, it allows execution of the legacy SSE instructions. If system software does not set CR4.OSFXSR, any attempt to execute these instructions causes an invalid-opcode exception (#UD). System software must also *clear* the CR0.EM (emulate coprocessor) bit to 0, otherwise an attempt to execute a legacy SSE instruction causes a #UD exception. An attempt to execute either FXSAVE or FXRSTOR when CR0.EM is set results in a #NM exception.

System software should also *set* the CR0.MP (monitor coprocessor) bit to 1. When CR0.EM=0 and CR0.MP=1, all media instructions, x87 instructions, and the FWAIT/WAIT instructions cause a device-not-available exception (#NM) when the CR0.TS bit is set. System software can use the #NM exception to perform lazy context switching, saving and restoring media and x87 state only when necessary after a task switch. See “CR0 Register” on page 42 for more information.

11.3.2 Enabling Extended SSE Instruction Execution

After the steps specified above are completed to enable legacy SSE instruction execution, additional steps are required to enable the extended SSE instructions and state management. System software must carry out the following process:

- Confirm that the hardware supports the XSAVE, XRSTOR, XSETBV, and XGETBV instructions and the XCR0 register by executing the CPUID instruction function 0000_0001h. If CPUID Fn0000_0001_ECX[XSAVE] is set, hardware support is verified.
- Optionally confirm hardware support of the XSAVEOPT instruction by executing CPUID function 0000_000Dh, sub-function 1 (ECX = 1). If CPUID Fn0000_000D_EAX_x1[XSAVEOPT] is set, the processor supports the XSAVEOPT instruction. XSAVEOPT is a performance optimized version of XSAVE.
- Confirm that hardware supports the extended SSE instructions by verifying XFeatureSupportedMask[2:0] = 111b. XFeatureSupportedMask is accessed via the CPUID instruction function 0000_000Dh, sub-function 0 (ECX = 0). XFeatureSupportedMask[31:0] is returned in the EAX register.

  If CPUID Fn0000_000D_EAX_x0[2:0] = 111b, hardware supports x87, legacy SSE, and extended SSE instructions. Bit 0 of EAX signifies x87 floating-point and MMX support, bit 1 signifies legacy SSE support, and bit 2 signifies extended SSE support. Support for both x87 and legacy SSE instructions are required for processors that support the extended SSE instructions.

[AMD Public Use]

- Set CR4[OSXSAVE] (bit 18) to enable the use of the XSETBV and XGETBV instructions. XSETBV is a privileged instruction that writes the XCRn registers. XCR0 is used to manage media and x87 processor state using the XSAVE, XSAVEOPT, and XRSTOR instructions.
- Enable the x87/MMX, legacy SSE, and extended SSE instructions and processor state management by setting the x87, SSE, and YMM bits of XCR0. This is done via the privileged instruction XSETBV. Enabling extended SSE capabilities without enabling legacy SSE capabilities is not allowed. The x87 flag (bit 0) must be set when writing XCR0.
- Determine the XSAVE/XRSTOR memory save area size requirement. The field XFeatureEnabledSizeMax specifies the size requirement in bytes based on the currently enabled extended features and is returned in the EBX register after execution of CPUID Function 0000_000Dh, sub-function 0 (ECX = 0).
- Allocate the save/restore area based on the information obtained in the previous step.

For a detailed description of the XSETBV and XGETBV instructions, see individual instruction reference pages in Volume 4.

For more information on using the CPUID instruction to obtain processor feature information, see Section 3.3, “Processor Feature Identification,” on page 73.

11.3.3 SIMD Floating-Point Exception Handling

System software must supply an exception handler if unmasked SSE floating-point exceptions are allowed to occur. When an unmasked exception is detected, the processor transfers control to the SIMD floating-point exception (#XF) handler provided by the operating system. System software must let the processor know that the #XF handler is available by setting CR4.OSXMMEXCPT to 1. If this bit is set to 1, the processor transfers control to the #XF handler when it detects an unmasked exception, otherwise a #UD exception occurs. When the processor detects a masked exception, it handles it in a default manner regardless of the CR4.OSXMMEXCPT value.

11.4 Media and x87 Processor State

The media and x87 processor state includes the contents of the registers used by SSE, MMX, and x87 instructions. System software that supports such applications must be capable of saving and restoring these registers.

11.4.1 SSE Execution Unit State

Figure 11-1 shows registers whose contents are affected by execution of SSE instructions, including:

- ZMM/YMM/XMM0-31—Thirty-two 512-bit/256-bit/128-bit SSE registers. In legacy and compatibility modes, software access is limited to the first eight registers.
- MXCSR—The 32-bit Media eXtensions Control and Status Register.

All of these registers are visible to application software. Refer to “Streaming SIMD Extensions Media and Scientific Programming” in Volume 1 for more information on these registers.

[AMD Public Use]

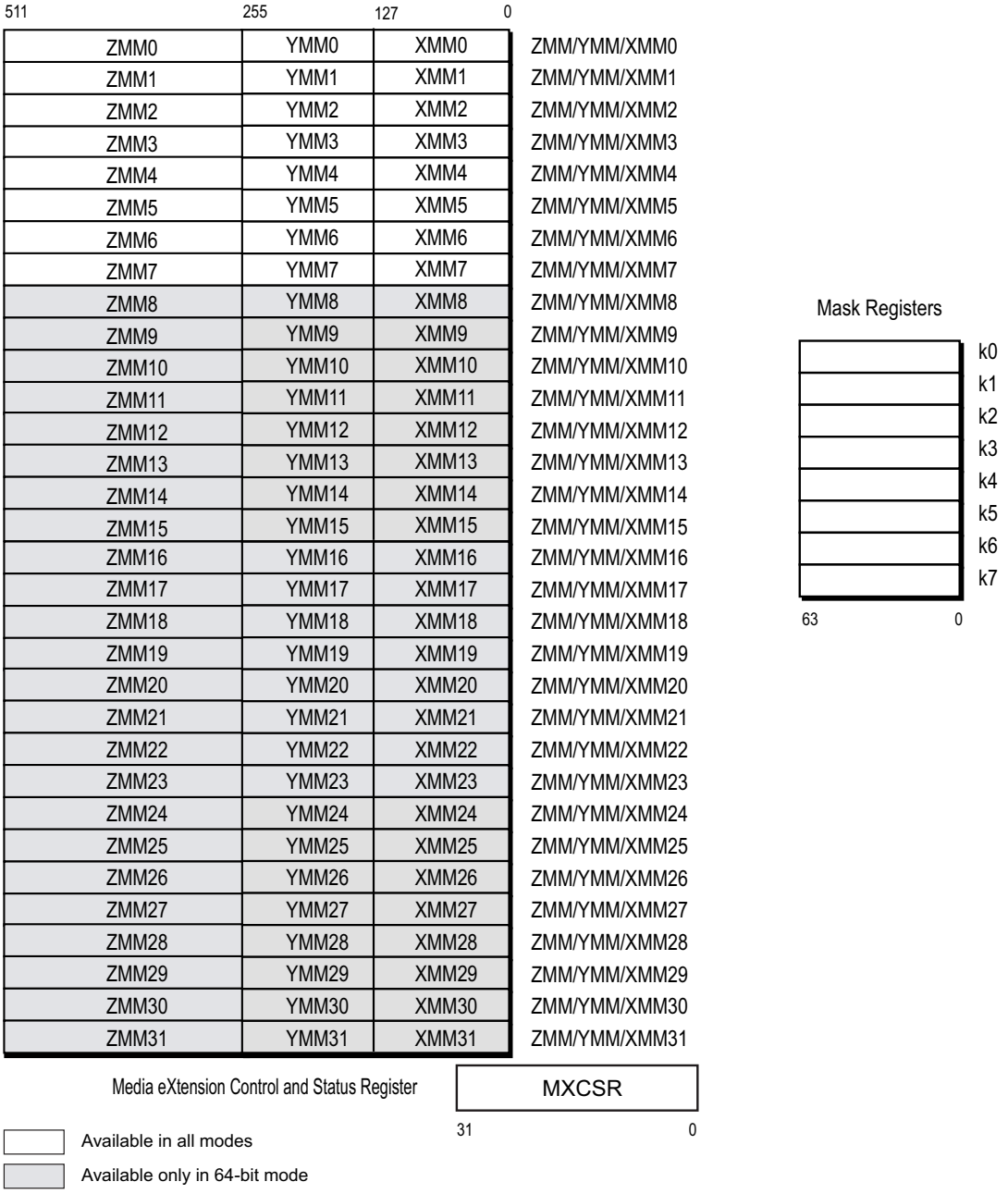


Figure 11-1. SSE Execution Unit State

11.4.2 MMX Execution Unit State

Figure 11-2 on page 349 shows register contents that are affected by execution of 64-bit media instructions, including:

- *mmx0–mmx7*—Eight 64-bit media registers.

[AMD Public Use]

- *FSW*—Two fields (TOP and ES) in the 16-bit x87 status word register.
- *FTW*—The 16-bit x87 tag word.

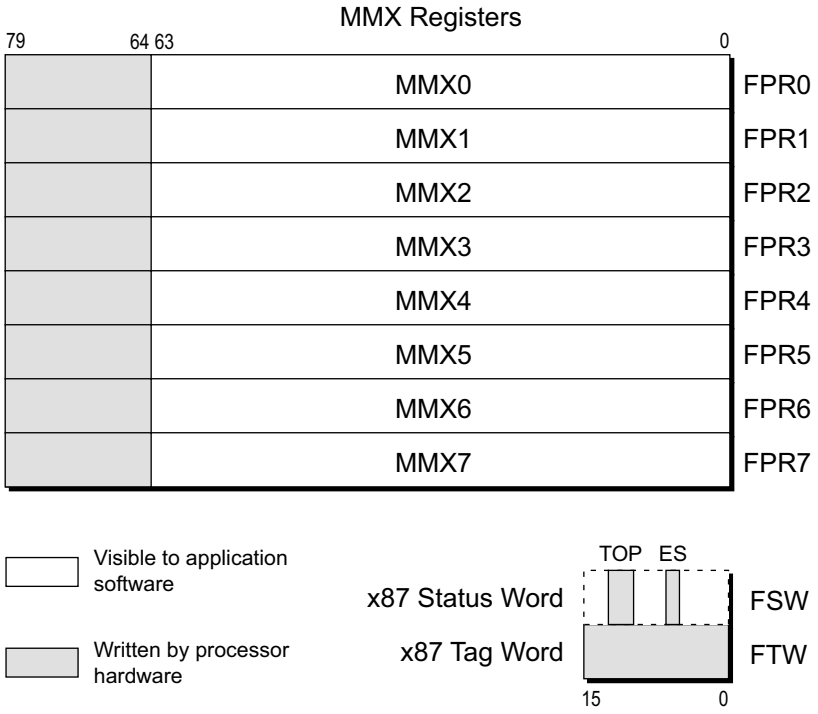


Figure 11-2. MMX Execution Unit State

The 64-bit media instructions and x87 floating-point instructions share the same physical data registers. Figure 11-2 shows how the 64-bit registers (MMX0–MMX7) are aliased onto the low 64 bits of the 80-bit x87 floating-point physical data registers (FPR0–FPR7). Refer to “64-Bit Media Programming” in Volume 1 for more information on these registers.

Of the registers shown in Figure 11-2, only the eight 64-bit MMX registers are visible to 64-bit media application software. The processor maintains the contents of the two fields of the x87 status word—top-of-stack-pointer (TOP) and exception summary (ES)—and the 16-bit x87 tag word during execution of 64-bit media instructions, as described in “Actions Taken on Executing 64-Bit Media Instructions” in Volume 1.

64-bit media instructions do not generate x87 floating-point exceptions, nor do they set any status flags. However, 64-bit media instructions can trigger an unmasked floating-point exception caused by a previously executed x87 instruction. 64-bit media instructions do this by reading the x87 FSW.ES bit to determine whether such an exception is pending.

11.4.3 x87 Execution Unit State

Figure 11-3 on page 351 shows the registers whose contents are affected by execution of x87 floating-point instructions. These registers include:

[AMD Public Use]

- *fpr0–fpr7*—Eight 80-bit floating-point physical registers.
- *FCW*—The 16-bit x87 control word register.
- *FSW*—The 16-bit x87 status word register.
- *FTW*—The 16-bit x87 tag word.
- *Last x87 Instruction Pointer*—This value is a pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the last non-control x87 floating-point instruction executed.
- *Last x87 Data Pointer*—The pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the data operand referenced by the last non-control x87 floating-point instruction executed, if that instruction referenced memory; if it did not, then this value is implementation dependent.
- *Last x87 Opcode*—An 11-bit permutation of the instruction opcode from the last non-control x87 floating-point instruction executed.

Of the registers shown in Figure 11-3 on page 351, only FPR0–FPR7, FCW, and FSW are directly updated by x87 application software. The processor maintains the contents of the FTW, instruction and data pointers, and opcode registers during execution of x87 instructions. Refer to “Registers” in Volume 1 for more information on these registers.

The 11-bit instruction opcode register holds a permutation of the two-byte instruction opcode from the last non-control x87 instruction executed by the processor. (For a definition of *non-control x87 instruction*, see “Control” in Chapter 6 of Volume 1.) The opcode field is formed as follows:

- Opcode Register Field[10:8] = First x87 opcode byte[2:0].
- Opcode Register Field[7:0] = Second x87 opcode byte[7:0].

For example, the x87 opcode D9 F8h is stored in the opcode register as 001_1111_1000b. The low-order three bits of the first opcode byte, D9h (1101_1001b), are stored in opcode-register bits 10:8. The second opcode byte, F8h (1111_1000b), is stored in bits 7:0 of the opcode register. The high-order five bits of the first opcode byte (1101_1b) are not needed because they are identical for all x87 instructions.

[AMD Public Use]

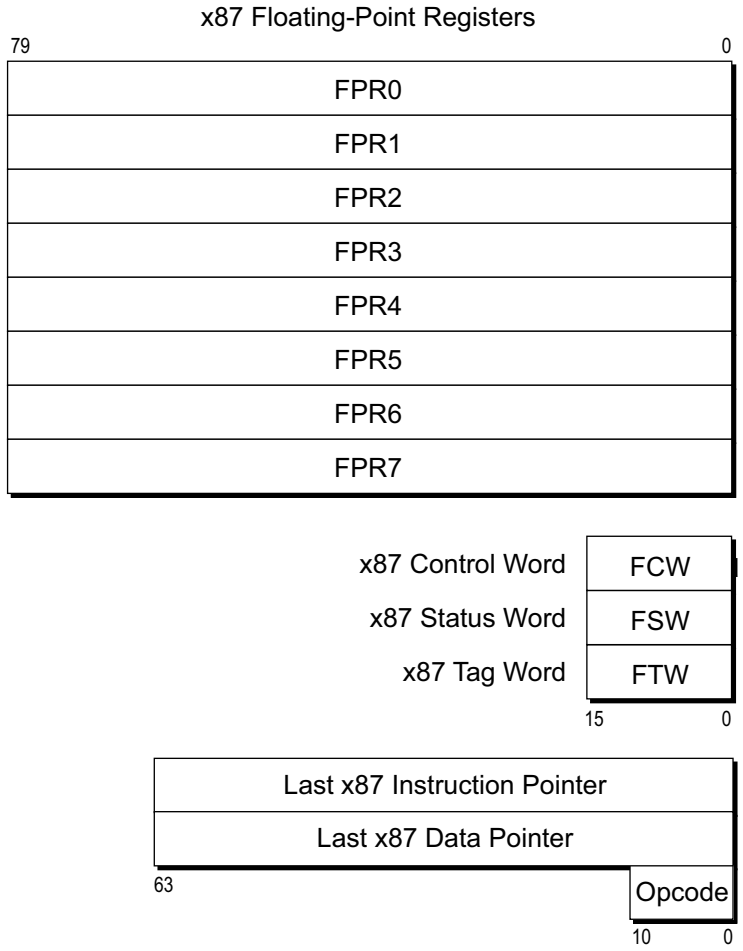


Figure 11-3. x87 Execution Unit State

11.4.4 Saving Media and x87 Execution Unit State

In most cases, operating systems, exception handlers, and device drivers should save and restore the media and/or x87 processor state between task switches or other interventions in the execution of 128-bit, 64-bit, or x87 procedures. Application programs are also free to save and restore state at any time.

In general, system software should use the FXSAVE and FXRSTOR instructions to save and restore the entire media and x87 processor state. The FSAVE/FNSAVE and FRSTOR instructions can be used for saving and restoring the x87 state. Because the 64-bit media registers are physically aliased onto the x87 registers, the FSAVE/FNSAVE and FRSTOR instructions can also be used to save and restore the 64-bit media state. However, FSAVE/FNSAVE and FRSTOR do not save or restore the 128-bit media state.

[AMD Public Use]

11.4.4.1 FSAVE/FNSAVE and FRSTOR Instructions

The FSAVE/FNSAVE and FRSTOR instructions save and restore the entire register state for 64-bit media instructions and x87 floating-point instructions. The FSAVE instruction stores the register state, but only after handling any pending unmasked-x87 floating-point exceptions. The FNSAVE instruction stores the register state but skips the reporting and handling of these exceptions. The state of all MMX/FPR registers is saved, as well as all other x87 state (the control word register, status word register, tag word, instruction pointer, data pointer, and last opcode). After saving this state, the tag state for all MMX/FPR registers is changed to *empty* and is thus available for a new procedure.

Starting on page 353, Figure 11-4 through Figure 11-7 show the memory formats used by the FSAVE/FNSAVE and FRSTOR instructions when storing the x87 state in various processor modes and using various effective-operand sizes. This state includes:

- *x87 Data Registers*
  - FPR0–FPR7 80-bit physical data registers.
- *x87 Environment*
  - FCW: x87 control word register
  - FSW: x87 status word register
  - FTW: x87 tag word
  - Last x87 instruction pointer
  - Last x87 data pointer
  - Last x87 opcode

The eight data registers are stored in the 80 bytes following the environment information. Instead of storing these registers in their physical order (FPR0–FPR7), the processor stores the registers in the their stack order, ST(0)–ST(7), beginning with the top-of-stack, ST(0).

[AMD Public Use]

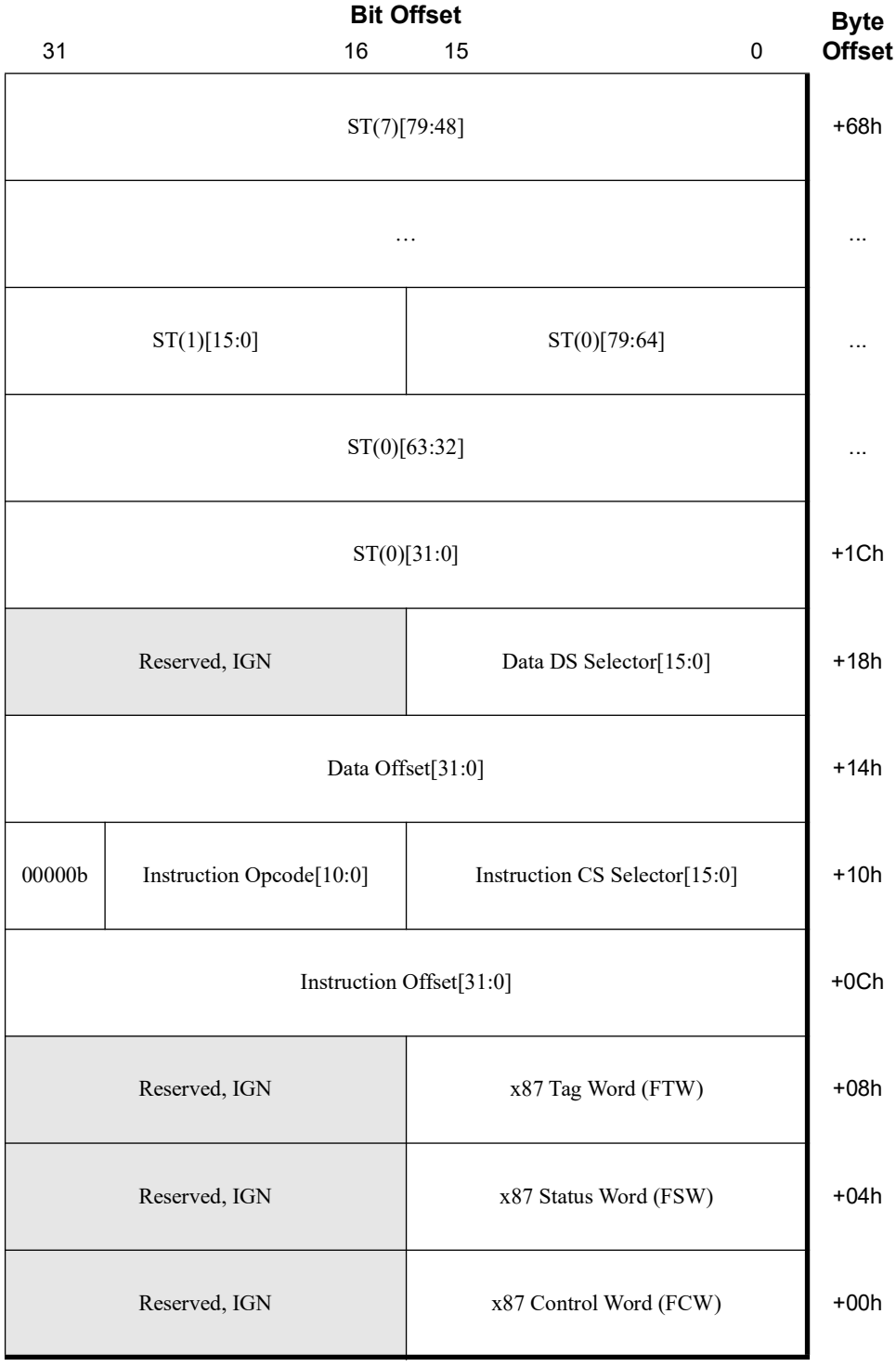


Figure 11-4. FSAVE/FNSAVE Image (32-Bit, Protected Mode)

[AMD Public Use]

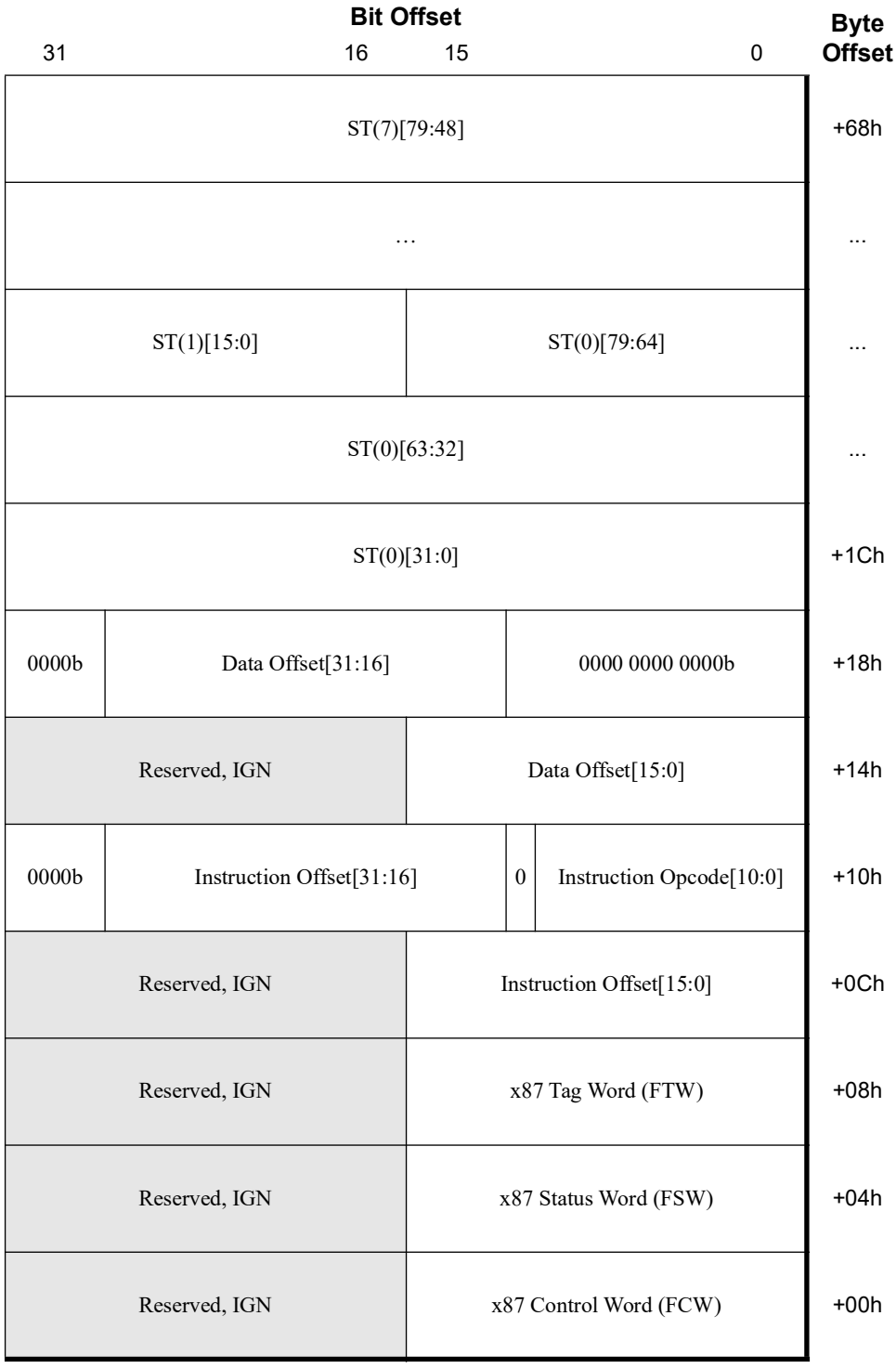


Figure 11-5. FSAVE/FNSAVE Image (32-Bit, Real/Virtual-8086 Modes)

[AMD Public Use]



Figure 11-6. FSAVE/FNSAVE Image (16-Bit, Protected Mode)

[AMD Public Use]



Figure 11-7. FSAVE/FNSAVE Image (16-Bit, Real/Virtual-8086 Modes)

11.4.4.2 FLDENV/FNLDENV and FSTENV Instructions

The FLDENV/FNLDENV and FSTENV instructions load and store only the x87 floating-point environment. These instructions, unlike the FSAVE/FNSAVE and FRSTOR instructions, do not save or restore the x87 data registers. The FLDENV/FSTENV instructions do not save the full 64-bit data and instruction pointers. 64-bit applications should use FXSAVE/FXRSTOR, rather than FLDENV/FSTENV. The format of the saved x87 environment images for protected mode and real/virtual mode are the same as those of the first 14-bytes of the FSAVE/FNSAVE images for 16-bit operands or 32/64-bit operands, respectively. See Figure 11-4 on page 353, Figure 11-5 on page 354, Figure 11-6 on page 355, and Figure 11-7.

[AMD Public Use]

11.4.4.3 FXSAVE and FXRSTOR Instructions

The FXSAVE and FXRSTOR instructions save and restore the entire 128-bit media, 64-bit media, and x87 state. These instructions usually execute faster than FSAVE/FNSAVE and FRSTOR because they do not normally save and restore the x87 exception pointers (last-instruction pointer, last data-operand pointer, and last opcode). The only case in which they do save the exception pointers is the relatively rare case in which the exception-summary bit in the x87 status word (FSW.ES) is set to 1, indicating that an unmasked exception has occurred. The FXSAVE and FXRSTOR memory format contains fields for storing these values.

Unlike FSAVE and FNSAVE, the FXSAVE instruction does not alter the x87 tag word. Therefore, the contents of the shared 64-bit MMX and 80-bit FPR registers can remain valid after an FXSAVE instruction (or any other value the tag bits indicated before the save). Also, FXSAVE (like FNSAVE) does not check for pending unmasked-x87 floating-point exceptions.

Figure 11-9 on page 363 shows the memory format of the media x87 state in long mode. If a 32-bit operand size is used in 64-bit mode, the memory format is the same, except that RIP and RDS are stored as *sel:offset* pointers, as shown in Figure 11-10 on page 364.

For more information on the FXSAVE and FXRSTOR instructions, see individual instruction listings in “64-Bit Media Instruction Reference” of Volume 5.

11.5 XSAVE/XRSTOR Instructions

The XSAVE, XSAVEOPT, XRSTOR, XGETBV, and XSETBV instructions and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor states and provide additional functionality. These instructions do not obviate the FXSAVE/FXRSTOR instructions. For more information about FXSAVE/FXRSTOR, see *“FXSAVE and FXRSTOR Instructions” in Volume 2*. For detailed descriptions of FXSAVE and FXRSTOR, see individual instruction listings in AMD64 Architecture Programmer’s Manual *“Volume 5: 64-Bit Media and x87 Floating-Point Instructions.”*

The CPUID instruction is used to identify features supported in processor hardware. Extended control registers are used to enable and disable the handling of processor states associated with supported hardware features and to communicate to an application whether an operating system supports a particular feature that has a processor state specific to it.

11.5.1 CPUID Enhancements

- CPUID Fn0000_00001_ECX[XSAVE] indicates that the processor supports XSAVE/XRSTOR instructions and at least one XCR.
- CPUID Fn0000_00001_ECX[OSXSAVE] indicates whether the operating system has enabled extensible state management and supports processor extended state management.
- CPUID Fn0000_0000D enumerates processor states (including legacy x87 FPU states, SSE states, and processor extended states), the offset, and the size of the save area for each processor extended state. Sub-functions (ECX > 0) provide details concerning features and support of processor states enumerated in the root function.

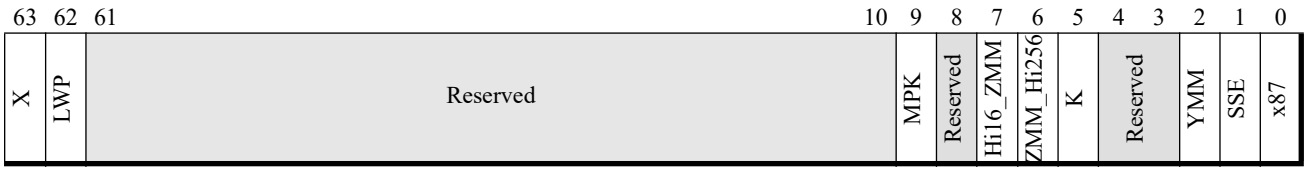
[AMD Public Use]

11.5.2 XCR0

XCR0 (XFEATURE_ENABLED_MASK) is set up by privileged software to enable the saving and restoring of extended processor architectural state information supported by a specific processor. Clearing defined bit fields in this mask inhibits the XSAVE instruction from saving (and XRSTOR from restoring) this state information.

XCR0 is accessed with the XSETBV and XGETBV instructions.

XCR0 is defined as follows:



Bits	Mnemonic	Description	Access type
63	X	Reserved specifically for XCR0 bit vector expansion.	MBZ
62	LWP	When set, Lightweight Profiling (LWP) extensions are enabled and XSAVE/XRSTOR supports LWP state management.	R/W
61:10	Reserved		MBZ
9	MPK	When set, PKRU state management is supported by XSAVE/XRSTOR.	R/W
8	Reserved		MBZ
7	Hi16_ZMM	When set, ZMM16-ZMM31 register state management is supported by XSAVE/XRSTOR. Must be set to enable AVX512 extensions.	R/W
6	ZMM_Hi256	When set, ZMM0-ZMM15 register state management, upper half, is supported by XSAVE/XRSTOR. Must be set to enable AVX512 extensions.	R/W
5	K	When set, K0-K7 opmask register state management is supported by XSAVE/XRSTOR. Must be set to enable AVX512 extensions.	R/W
4:3	Reserved		MBZ
2	YMM	When set, 256-bit SSE state management is supported by XSAVE/XRSTOR. Must be set to enable AVX extensions.	R/W
1	SSE	When set, 128-bit SSE state management is supported by XSAVE/XRSTOR. This bit must be set if YMM is set. Must be set to enable AVX extensions.	R/W
0	x87	x87 FPU state management is supported by XSAVE/XRSTOR. Must be set to 1.	R/W

Figure 11-8. XCR0 Format

Hardware initializes XCR0 to 0000_0000_0000_0001h. On writing this register, software must ensure that bits marked MBZ in Figure 11-8 are clear and that XCR0[7:5,2:0] are either 000_001b, 000_011b, 000_111b, or 111_111b. An attempt to write data that violates these rules results in a #GP.

[AMD Public Use]

11.5.3 Extended Save Area

The XSAVE/XRSTOR save area extends the legacy 512-byte FXSAVE/FXRSTOR memory image to provide a compatible register state management environment as well as an upward migration path. The save area is architecturally defined to be extendable and enumerated by the sub-functions of CPUID Fn 0000_000Dh. Figure 11-6 shows the format of the XSAVE/XRSTOR area.

Table 11-6. Extended Save Area Format

Save Area	Offset (Byte)	Size (Bytes)
FPU/SSE Save Area	0	512
Header	512	64
Reserved, (Ext_Save_Area_2)	CPUID Fn 0000_000D_EBX_x02	CPUID Fn 0000_000D_EAX_x02
Reserved, (Ext_Save_Area_3)	CPUID Fn 0000_000D_EBX_x03	CPUID Fn 0000_000D_EAX_x03
Reserved, (Ext_Save_Area_4)	CPUID Fn 0000_000D_EBX_x04	CPUID Fn 0000_000D_EAX_x04
Reserved, (…)	…	…
Note: *Bytes 464–511 are available for software use. XRSTOR ignores bytes 464–511 of an XSAVE image.*		

The register fields of the first 512 bytes of the XSAVE/XRSTOR area are the same as those of the FXSAVE/FXRSTOR area, but the 512-byte area is organized as x87 FPU states, MXCSR (including MXCSR_MASK), and XMM registers. The layout of the save area is fixed and may contain non-contiguous individual save areas because a processor does not support certain extended states or because system software does not support certain processor extended states. The save area is not compacted when features are not saved or are not supported by the processor or by system software.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 73.

11.5.4 Instruction Functions

CR4.OSXSAVE and XCR0 can be read at all privilege levels but written only at ring 0.

- XGETBV reads XCR0.
- XSETBV writes XCR0, ring 0 only.
- XRSTOR restores states specified by bitwise AND of a mask operand in EDX:EAX with XCR0.
- XSAVE (and XSAVEOPT) saves states specified by bitwise AND of a mask operand in EDX:EAX with XCR0.

11.5.5 ZMM/YMM States and Supported Operating Modes

Extended instructions operate on ZMM/YMM states by means of extended (XOP/VEX/EVEX) prefix encoding. When a processor supports ZMM/YMM states, the states exist in all operating modes, but interfaces to access the ZMM/YMM states may vary by mode. Processor support for extended prefix encoding is independent of processor support of ZMM/YMM states.

[AMD Public Use]

Instructions that use extended prefix encoding are generally supported in long and protected modes, but are not supported in real or virtual 8086 modes, or when entering SMM mode. Bits 511:128 of the ZMM/YMM register state are maintained across transitions into and out of these modes. The XSAVE/XRSTOR instructions function in all operating modes; XRSTOR can modify ZMM/YMM register state in any operating mode, using state information from the XSAVE/XRSTOR area.

11.5.6 Extended SSE Execution State Management

Operating system software must use the XSAVE/XRSTOR instructions for extended SSE execution state management. XSAVEOPT, a performance optimized version of XSAVE, may be used instead of XSAVE once the XSAVE/XRSTOR save area is initialized. In the following discussion XSAVEOPT may be substituted for the instruction XSAVE. The instructions also provide an interface to manage XMM/MXCSR states and x87 FPU states in conjunction with processor extended states. An operating system must enable extended SSE execution state management prior to the execution of extended SSE instructions. Attempting to execute an extended SSE instruction without enabling execution state management causes a #UD exception.

11.5.6.1 Enabling Extended SSE Instruction Execution

To enable extended SSE instruction execution and state management, system software must carry out the following process:

- Confirm that the hardware supports the XSAVE, XRSTOR, XSETBV, and XGETBV instructions and the XCR0 register by executing the CPUID instruction function 0000_0001h. If CPUID Fn0000_0001_ECX[XSAVE] is set, hardware support is verified.
- Optionally confirm hardware support of the XSAVEOPT instruction by executing CPUID function 0000_000Dh, sub-function 1 (ECX = 1). If CPUID Fn0000_000D_EAX_x1[XSAVEOPT] is set, the processor supports the XSAVEOPT instruction. XSAVEOPT is a performance optimized version of XSAVE. (SDCR-3580)
- Confirm that hardware supports the extended SSE instructions by verifying XFeatureSupportedMask[2:0] = 111b. XFeatureSupportedMask is accessed via the CPUID instruction function 0000_000Dh, sub-function 0 (ECX = 0).

  If CPUID Fn0000_000D_EAX_x0[2:0] = 111b, hardware supports x87, legacy SSE, and extended SSE instructions. Bit 0 of EAX signifies x87 floating-point and MMX support, bit 1 signifies legacy SSE support, and bit 2 signifies extended SSE support. Support for both x87 and legacy SSE instructions are required for processors that support the extended SSE instructions.

  Confirm that hardware supports the AVX512 instructions by verifying XFeatureSupportedMask[7:5] = 111b.
- Set CR4[OSXSAVE] (bit 18) to enable the use of the XSETBV and XGETBV instructions. XSETBV is a privileged instruction that writes the XCRn registers. XCR0 is used to manage media and x87 processor state using the XSAVE, XSAVEOPT, and XRSTOR instructions.
- Enable the x87/MMX, legacy SSE, and extended SSE instructions and processor state management by setting the x87, SSE, YMM, ZMM, and K bits of XCR0. Enabling extended SSE

[AMD Public Use]

capabilities without enabling legacy SSE capabilities is not allowed. The x87 flag (bit 0) must be set when writing XCR0.

- Determine the XSAVE/XRSTOR memory save area size requirement. The field XFeatureEnabledSizeMax specifies the size requirement in bytes based on the currently enabled extended features and is returned in the EAX register after execution of CPUID Function 0000_000Dh, sub-function 0 (ECX = 0).
- Allocate the save/restore area based on the information obtained in the previous step.

For more information on the XSETBV and XGETBV instructions, see individual instruction descriptions in Volume 4. XCR0 fields are defined in Section 11.5.2 above.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 73.

11.5.7 Saving Processor State

The XSTATE header starts at byte offset 512 in the save area. XSTATE_BV is the first 64-bit field in the header. The order of bit vectors in XSTATE_BV matches the order of bit vectors in XCR0. The XSAVE instruction sets bits in the XSTATE_BV vector field when it writes the corresponding processor extended state to a save area in memory. XSAVE modifies only bits for processor states specified by bitwise AND of the XSAVE bit mask operand in EDX:EAX with XCR0. If software modifies the save area image of a particular processor state component directly, it must also set the corresponding bit of XSTATE_BV. If the bit is not set, directly modified state information in a save area image may be ignored by XRSTOR.

XSAVEOPT, a performance optimized version of the XSAVE instruction, may be used (if supported) in lieu of the XSAVE instruction once the XSAVE/XRSTOR save area has been initialized via the execution of the XSAVE instruction.

11.5.8 Restoring Processor State

When XRSTOR is executed, processor state components are updated only if the corresponding bits in the mask operand (EDX:EAX) and XCR0 are both set. For each updated component, when the corresponding bit in the XSTATE_BV field in the save area header is set, the component is loaded

[AMD Public Use]

from the save area in memory. When the XSTATE_BV bit is cleared, the state is set to the hardware-specified initial values shown in Table 11-7.

Table 11-7. XRSTOR Hardware-Specified Initial Values

Component	Initial Value
x87	FCW = 037Fh FSW = 0000h Empty/Full = 00h (FTW = FFFFh) x87 Error Pointers = 0 ST0 - ST7 = 0
XMM	XMM0 - XMM15 = 0, if 64-bit mode XMM0 - XMM7 = 0, if !64-bit mode
YMM_HI	YMM_HI0 -Y MM_HI15 = 0, if 64-bit mode YMM_HI0-YMM_HI7 = 0, if !64-bit mode
K	K0-K7 = 0
ZMM_Hi256	ZMM_HI0-YMM_HI15 = 0, if 64-bit mode ZMM_HI0-YMM_HI7 = 0, if !64-bit mode
HI16_ZMM	ZMM16-ZMM31 = 0
LWP	LWP disabled
MPK	PKRU = 0
CET_S	PL0_SSP = PL1_SSP = PL2_SSP = 0
CET_U	U_CET = 0, PL3_SSP = 0

11.5.9 MXCSR State Management

The MXCSR has no hardware-specified initial state; it is read from the save area in memory whenever either XMM or YMM_HI are updated.

11.5.10 Mode-Specific XSAVE/XRSTOR State Management

Some state is conditionally saved or updated, depending on processor state:

- On processors where CPUID Fn8000_0008_EBX[2] is 0, the x87 error pointers are not saved or restored if the state saved or loaded from memory doesn't have a pending #MF. On processors where CPUID Fn8000_0008_EBX[2] is 1, the error pointers are always restored from the save area (and if in 64-bit mode the CS and DS portions of the error pointer registers are zeroed), and the error pointer fields in the save area are zeroed if there is no pending #MF, else the error pointer offset registers are written to the save area.
- XMM8–XMM15 are not saved or restored in non-64-bit mode.
- YMM_HI8–YMM_HI15 are not saved or restored in non-64-bit mode.
- ZMM8–ZMM31[511:256] are not saved or restored in non-64-bit mode.

[AMD Public Use]

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
Reserved, IGN ... Reserved, IGN																+1F0h … +1A0h
XMM15																+190h
XMM14																+180h
XMM13																+170h
XMM12																+160h
XMM11																+150h
XMM10																+140h
XMM9																+130h
XMM8																+120h
XMM7																+110h
XMM6																+100h
XMM5																+F0h
XMM4																+E0h
XMM3																+D0h
XMM2																+C0h
XMM1																+B0h
XMM0																+A0h
Reserved, IGN								ST(7)								+90h
Reserved, IGN								ST(6)								+80h
Reserved, IGN								ST(5)								+70h
Reserved, IGN								ST(4)								+60h
Reserved, IGN								ST(3)								+50h
Reserved, IGN								ST(2)								+40h
Reserved, IGN								ST(1)								+30h
Reserved, IGN								ST(0)								+20h
MXCSR_MASK								MXCSR				RDP[1]				+10h
RIP[1]								FOP		0	FTW	FSW			FCW	+00h

1. Stored as sel:offset if operand size is 32 bits. 32bit sel:offset format of the pointers is shown in figure 11-10.

Figure 11-9. FXSAVE and FXRSTOR Image (64-bit Mode)

[AMD Public Use]

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
Reserved, IGN ... Reserved, IGN																+1F0h ... +120h
XMM7																+110h
XMM6																+100h
XMM5																+F0h
XMM4																+E0h
XMM3																+D0h
XMM2																+C0h
XMM1																+B0h
XMM0																+A0h
Reserved, IGN								ST(7)								+90h
Reserved, IGN								ST(6)								+80h
Reserved, IGN								ST(5)								+70h
Reserved, IGN								ST(4)								+60h
Reserved, IGN								ST(3)								+50h
Reserved, IGN								ST(2)								+40h
Reserved, IGN								ST(1)								+30h
Reserved, IGN								ST(0)								+20h
MXCSR_MASK				MXCSR				rsrvd, IGN		DS			DP			+10h
rsrvd, IGN		CS		EIP				FOP		0	FTW		FSW		FCW	+00h

Figure 11-10. FXSAVE and FXRSTOR Image (Non-64-bit Mode)

Software can read and write all fields within the FXSAVE and FXRSTOR memory image. These fields include:

- *FCW*—Bytes 01h–00h. x87 control word.
- *FSW*—Bytes 03h–02h. x87 status word.
- *FTW*—Byte 04h. x87 tag word. See “FXSAVE Format for x87 Tag Word” on page 365 for additional information on the FTW format saved by the FXSAVE instruction.
- (Byte 05h contains the value 00h.)
- *FOP*—Bytes 07h–06h. last x87 opcode.
- *Last x87 Instruction Pointer*—A pointer to the last non-control x87 floating-point instruction executed by the processor:

[AMD Public Use]

- *RIP (64-bit format)*—Bytes 0Fh–08h. 64-bit offset into the code segment (used without a CS selector).
- *EIP (32-bit format)*—Bytes 0Bh–08h. 32-bit offset into the code segment.
- *CS (32-bit format)*—Bytes 0Dh–0Ch. Segment selector portion of the pointer.

• *Last x87 Data Pointer*—If the last non-control x87 floating point instruction referenced memory, this value is a pointer to the data operand referenced by the last non-control x87 floating-point instruction executed by the processor:

- *RDP (64-bit format)*—Bytes 17h–10h. 64-bit offset into the data segment (used without a DS selector).
- *DP (32-bit format)*—Bytes 13h–10h. 32-bit offset into the data segment.
- *DS (32-bit format)*—Bytes 15h–14h. Segment selector portion of the pointer.

If the last non-control x87 instruction did not reference memory, then the value in the pointer is implementation dependent.

• *MXCSR*—Bytes 1Bh–18h. 128-bit media-instruction control and status register. This register is saved only if CR4.OSFXSR is set to 1.

• *MXCSR_MASK*—Bytes 1Fh–1Ch. Set bits in MXCSR_MASK indicate supported feature bits in MXCSR. For example, if bit 6 (the DAZ bit) in the returned MXCSR_MASK field is set to 1, the DAZ mode and the DAZ flag in MXCSR *are* supported. Cleared bits in MXCSR_MASK indicate reserved bits in MXCSR. If software attempts to set a reserved bit in the MXCSR register, a #GP exception will occur. To avoid this exception, after software clears the FXSAVE memory image and executes the FXSAVE instruction, software should use the value returned by the processor in the MXCSR_MASK field when writing a value to the MXCSR register, as follows:

- *MXCSR_MASK = 0:* If the processor writes a zero value into the MXCSR_MASK field, the denormals-are-zeros (DAZ) mode and the DAZ flag in MXCSR are *not* supported. Software should use the default mask value, 0000_FFBFh (bit 6, the DAZ bit, and bits 31:16 cleared to 0), to mask any value it writes to the MXCSR register to ensure that all reserved bits in MXCSR are written with 0, thus avoiding a #GP exception.
- *MXCSR_MASK ≠ 0:* If the processor writes a non-zero value into the MXCSR_MASK field, software should AND this value with any value it writes to the MXCSR register.

• *MMXn/FPRn*—Bytes 9Fh–20h. Shared 64-bit media and x87 floating-point registers. As in the case of the x87 FSAVE instruction, these registers are stored in stack order ST(0)–ST(7). The upper six bytes in the memory image for each register are reserved.

• *XMMn*—Bytes 11Fh–A0h. 128-bit media registers. These registers are saved only if CR4.OSFXSR is set to 1.

11.5.10.1 FXSAVE Format for x87 Tag Word

Rather than saving the entire x87 tag word, FXSAVE saves a single-byte encoded version. FXSAVE encodes each of the eight two-bit fields in the x87 tag word as follows:

• Two-bit values of 00, 01, and 10 are encoded as a 1, indicating the corresponding x87 FPR*n* register holds a value.

[AMD Public Use]

- A two-bit value of 11 is encoded as a 0, indicating the corresponding x87 FPR*n* is empty.

For example, assume an FSAVE instruction saves an x87 tag word with the value 83F1h. This tag-word value describes the x87 FPRn contents as follows:

x87 Register	FPR7	FPR6	FPR5	FPR4	FPR3	FPR2	FPR1	FPR0
Tag Word Value (hex)	8		3		F		1	
Tag Value (binary)	10	00	00	11	11	11	00	01
Meaning	Special	Valid	Valid	Empty	Empty	Empty	Valid	Zero

When an FXSAVE is used to write the x87 tag word to memory, it encodes the value as E3h. This encoded version describes the x87 FPRn contents as follows:

x87 Register	FPR7	FPR6	FPR5	FPR4	FPR3	FPR2	FPR1	FPR0
Encoded Tag Byte (hex)	E				3			
Tag Value (binary)	1	1	1	0	0	0	1	1
Meaning	Valid	Valid	Valid	Empty	Empty	Empty	Valid	Valid

If necessary, software can decode the single-bit FXSAVE tag-word fields into the two-bit field FSAVE uses by examining the contents of the corresponding FPR registers saved by FXSAVE. Table 11-8 on page 367 shows how the FPR contents are used to find the equivalent FSAVE tag-field value. The *fraction* column refers to fraction portion of the extended-precision significand (bits 62:0). The *integer bit* column refers to the integer-portion of the significand (bit 63). See Chapter 11, “SSE, MMX, and x87 Programming,” on page 344 for more information on floating-point numbering formats.

[AMD Public Use]

Table 11-8. Deriving FSAVE Tag Field from FXSAVE Tag Field

Encoded FXSAVE Tag Field	Exponent	Integer Bit[2]	Fraction[1]	Type of Value	Equivalent FSAVE Tag Field
1 (Valid)	All 0s	0	All 0s	Zero	01 (Zero)
		0	Not all 0s	Denormal	10 (Special)
		1	All 0s	Pseudo Denormal	
		1	Not all 0s		
	Neither all 0s nor all 1s	0	don't care	Unnormal	
		1		Normal	00 (Valid)
	All 1s	0		Pseudo Infinity or Pseudo NaN	10 (Special)
		1	All 0s	Infinity	
			Not all 0s	NaN	
0 (Empty)	don't care			Empty	11 (Empty)

Note(s):

1. *Bits 62:0 of the significand. Bit 62, the most-significant bit of the fraction, is also called the M bit.*
2. *Bit 63 of the significand, also called the J bit.*

11.5.10.2 Performance Considerations

When system software supports multi-tasking, it must be able to save the processor state for one task and load the state for another. For performance reasons, the media and/or x87 processor state is usually saved and loaded only when necessary. System software can save and load this state at the time a task switch occurs. However, if the new task does not use the state, loading the state is unnecessary and reduces performance.

The task-switch bit (CR0.TS) is provided as a *lazy* context-switch mechanism that allows system software to save and load the processor state only when necessary. When CR0.TS=1, a device-not-available exception (#NM) occurs when an attempt is made to execute a 128-bit media, 64-bit media, or x87 instruction. System software can use the #NM exception handler to save the state of the previous task, and restore the state of the current task. Before returning from the exception handler to the media or x87 instruction, system software must clear CR0.TS to 0 to allow the instruction to be executed. Using this approach, the processor state is saved only when the registers are used.

In legacy mode, the hardware task-switch mechanism sets CR0.TS=1 during a task switch (see "Task Switched (TS) Bit" on page 44 for more information). In long mode, the hardware task-switching is not supported, and the CR0.TS bit is not set by the processor. Instead, the architecture assumes that system software handles all task-switching and state-saving functions. If CR0.TS is to be used in long mode for controlling the save and restore of media or x87 state, system software must set and clear it explicitly.

[AMD Public Use]

12 Task Management

This chapter describes the hardware task-management features. All of the legacy x86 task-management features are supported by the AMD64 architecture in legacy mode, but most features are not available in long mode. Long mode, however, requires system software to initialize and maintain certain task-management *resources*. The details of these resource-initialization requirements for long mode are discussed in "Task-Management Resources" on page 369.

12.1 Hardware Multitasking Overview

A task (also called a *process*) is a program that the processor can execute, suspend, and later resume executing at the point of suspension. During the time a task is suspended, other tasks are allowed to execute. Each task has its own execution space, consisting of:

- Code segment and instruction pointer.
- Data segments.
- Stack segments for each privilege level.
- General-purpose registers.
- rFLAGS register.
- Local-descriptor table.
- Task register, and a link to the previously-executed task.
- I/O-permission and interrupt-permission bitmaps.
- Pointer to the page-translation tables (CR3).

The state information defining this execution space is stored in the task-state segment (TSS) maintained for each task.

Support for hardware multitasking is provided in legacy mode. Hardware multitasking provides automated mechanisms for switching tasks, saving the execution state of the suspended task, and restoring the execution state of the resumed task. When hardware multitasking is used to switch tasks, the processor takes the following actions:

- Suspends execution of the task, allowing any executing instructions to complete and save their results.
- Saves the task execution state in the task TSS.
- Loads the execution state for the new task from its TSS.
- Begins executing the new task at the location specified in the new task TSS.

Software can switch tasks by branching to a new task using the CALL or JMP instructions. Exceptions and interrupts can also switch tasks if the exception or interrupt handlers are themselves separate tasks. IRET can be used to return to an earlier task.

[AMD Public Use]

12.2 Task-Management Resources

The hardware-multitasking features are available when protected mode is enabled (CR0.PE=1). Protected-mode software execution, by definition, occurs as part of a task. While system software is not required to use the hardware-multitasking features, it is required to initialize certain task-management resources for at least one task (the current task) when running in protected mode. This single task is needed to establish the protected-mode execution environment. The resources that must be initialized are:

- *Task-State Segment (TSS)*—A segment that holds the processor state associated with a task.
- *TSS Descriptor*—A segment descriptor that defines the task-state segment.
- *TSS Selector*—A segment selector that references the TSS descriptor located in the GDT.
- *Task Register*—A register that holds the TSS selector and TSS descriptor for the current task.

Figure 12-1 on page 370 shows the relationship of these resources to each other in both 64-bit and 32-bit operating environments.

[AMD Public Use]

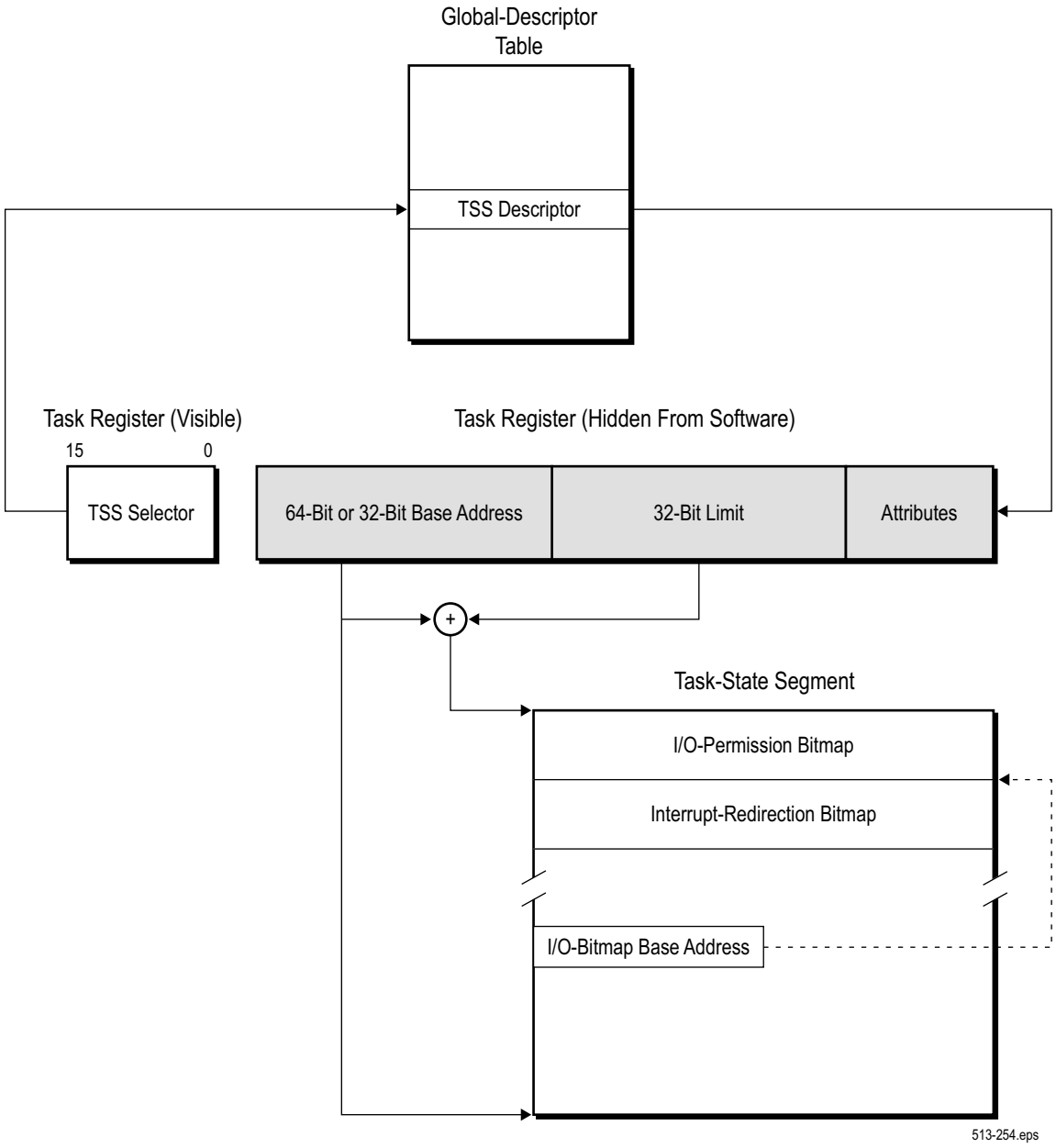


Figure 12-1. Task-Management Resources

A fifth resource is available in legacy mode for use by system software that uses the hardware-multitasking mechanism to manage more than one task:

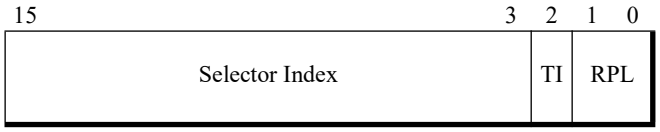
- *Task-Gate Descriptor*—This form of gate descriptor holds a reference to a TSS descriptor and is used to control access between tasks.

The task-management resources are described in the following sections.

[AMD Public Use]

12.2.1 TSS Selector

TSS selectors are selectors that point to task-state segment descriptors in the GDT. Their format is identical to all other segment selectors, as shown in Figure 12-2.



Bits	Mnemonic	Description
15:3		Selector Index
2	TI	Table Indicator
1:0	RPL	Requestor Privilege Level

Figure 12-2. Task-Segment Selector

The selector format consists of the following fields:

Selector Index. Bits 15:3. The selector-index field locates the TSS descriptor in the global-descriptor table.

Table Indicator (TI) Bit. Bit 2. The TI bit must be cleared to 0, which indicates that the GDT is used. TSS descriptors cannot be located in the LDT. If a reference is made to a TSS descriptor in the LDT, a general-protection exception (#GP) occurs.

Requestor Privilege-Level (RPL) Field. Bits 1:0. RPL represents the privilege level (CPL) the processor is operating under at the time the TSS selector is loaded into the task register.

12.2.2 TSS Descriptor

The TSS descriptor is a system-segment descriptor, and it can be located only in the GDT. The format for an 8-byte, legacy-mode and compatibility-mode TSS descriptor can be found in “System Descriptors” on page 96. The format for a 16-byte, 64-bit mode TSS descriptor can be found in “System Descriptors” on page 101.

The fields within a TSS descriptor (all modes) are described in “Descriptor Format” on page 90. The following additional information applies to TSS descriptors:

- *Segment Limit*—When shadow stacks are not enabled (CR4.CET=0), a TSS descriptor must have a segment limit value of at least 67h, which defines a minimum TSS size of 68h (104 decimal) bytes. If shadow stacks are enabled (CR4.CET=1), the TSS segment limit must be at least 06Bh (for a minimum size of 108 decimal bytes), in order to accommodate the 32-bit shadow stack pointer (SSP). If the limit is less than the specified values, an invalid-TSS exception (#TS) occurs during the task switch. When an I/O-permission bitmap, interrupt-redirection bitmap, or additional state information is included in the TSS, the limit must be set to a value large enough to enclose that information. In this case, if the TSS limit is not large enough to hold the additional information, a

[AMD Public Use]

#GP exception occurs when an attempt is made to access beyond the TSS limit. No check for the larger limit is performed during the task switch.

- *Type*—Four system-descriptor types are defined as TSS types, as shown in Table 4-5 on page 96. Bit 9 is used as the descriptor busy bit (B). This bit indicates that the task is busy when set to 1, and available when cleared to 0. Busy tasks are the currently running task and any previous (outer) tasks in a nested-task hierarchy. Task recursion is not supported, and a #GP exception occurs if an attempt is made to transfer control to a busy task. See "Nesting Tasks" on page 389 for additional information.

  In long mode, the 32-bit TSS types (available and busy) are redefined as 64-bit TSS types, and only 64-bit TSS descriptors can be used. Loading the task register with an available 64-bit TSS causes the processor to change the TSS descriptor type to indicate a busy 64-bit TSS. Because long mode does not support task switching, the TSS-descriptor busy bit is never cleared by the processor to indicate an available 64-bit TSS.

  Sixteen-bit TSS types are illegal in long mode. A general-protection exception (#GP) occurs if a reference is made to a 16-bit TSS.

12.2.3 Task Register

The *task register* (TR) points to the TSS location in memory, defines its size, and specifies its attributes. As with the other descriptor-table registers, the TR has two portions. A *visible* portion holds the TSS selector, and a *hidden* portion holds the TSS descriptor. When the TSS selector is loaded into the TR, the processor automatically loads the TSS descriptor from the GDT into the hidden portion of the TR.

The TR is loaded with a new selector using the LTR instruction. The TR is also loaded during a task switch, as described in "Switching Tasks" on page 382.

Figure 12-3 shows the format of the TR in legacy mode.

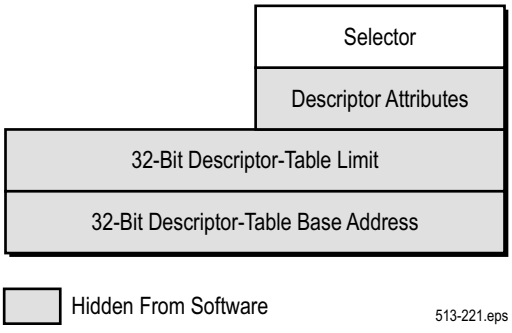


Figure 12-3. TR Format, Legacy Mode

Figure 12-4 shows the format of the TR in long mode (both compatibility mode and 64-bit mode).

[AMD Public Use]

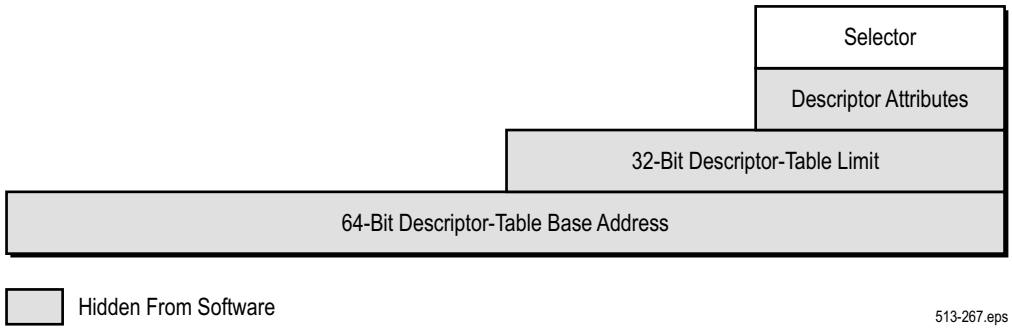


Figure 12-4. TR Format, Long Mode

The AMD64 architecture expands the TSS-descriptor base-address field to 64 bits so that system software running in long mode can access a TSS located anywhere in the 64-bit virtual-address space. The processor ignores the 32 high-order base-address bits when running in legacy mode. Because the TR is loaded from the GDT, the system-segment descriptor format has been expanded to 16 bytes by the AMD64 architecture in support of 64-bit mode. See “System Descriptors” on page 101 for more information on this expanded format. The high-order base-address bits are only loaded from 64-bit mode using the LTR instruction. Figure 12-5 shows the relationship between the TSS and GDT.

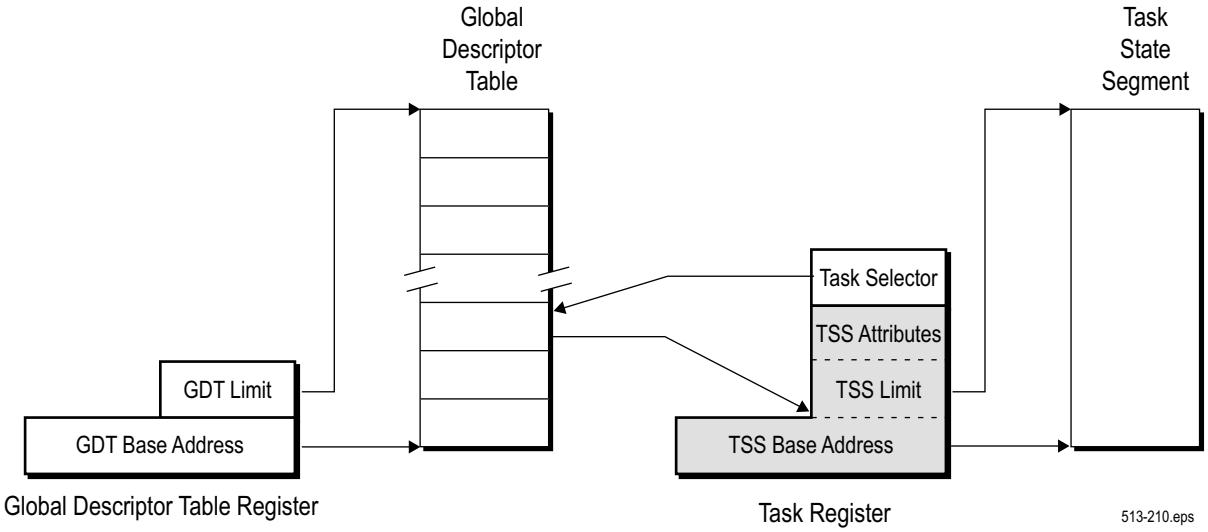


Figure 12-5. Relationship between the TSS and GDT

Long mode requires the use of a 64-bit TSS type, and this type must be loaded into the TR by executing the LTR instruction in *64-bit mode*. Executing the LTR instruction in 64-bit mode loads the TR with the full 64-bit TSS base address from the 16-byte TSS descriptor format (compatibility mode

[AMD Public Use]

can only load 8-byte system descriptors). A processor running in either compatibility mode or 64-bit mode uses the full 64-bit TR.base address.

12.2.4 Legacy Task-State Segment

The task-state segment (TSS) is a data structure in memory that the processor uses to save and restore the execution state for a task when a task switch occurs. Figure 12-6 on page 375 shows the format of a legacy 32-bit TSS.

[AMD Public Use]

Bit Offset 31	16	15	0	Byte Offset
I/O-Permission Bitmap (IOPB) (Up to 8 Kbytes)				IOPB Base
Interrupt-Redirection Bitmap (IRB) (Eight 32-Bit Locations)				
- –	Operating-System Data Structure		- –	
SSP				+68h
I/O-Permission Bitmap Base Address		Reserved, IGN	T	+64h
Reserved, IGN		LDT Selector		+60h
Reserved, IGN		GS		+5Ch
Reserved, IGN		FS		+58h
Reserved, IGN		DS		+54h
Reserved, IGN		SS		+50h
Reserved, IGN		CS		+4Ch
Reserved, IGN		ES		+48h
EDI				+44h
ESI				+40h
EBP				+3Ch
ESP				+38h
EBX				+34h
EDX				+30h
ECX				+2Ch
EAX				+28h
EFLAGS				+24h
EIP				+20h
CR3				+1Ch
Reserved, IGN		SS2		+18h
ESP2				+14h
Reserved, IGN		SS1		+10h
ESP1				+0Ch
Reserved, IGN		SS0		+08h
ESP0				+04h
Reserved, IGN		Link (Prior TSS Selector)		+00h

Figure 12-6. Legacy 32-bit TSS

[AMD Public Use]

The 32-bit TSS contains three types of fields:

- *Static fields* are read by the processor during a task switch when a new task is loaded, but are not written by the processor when a task is suspended.
- *Dynamic fields* are read by the processor during a task switch when a new task is loaded, and are written by the processor when a task is suspended.
- *Software-defined fields* are read and written by software, but are not read or written by the processor. All but the first 104 bytes of a TSS can be defined for software purposes, minus any additional space required for the optional I/O-permission bitmap and interrupt-redirection bitmap.

TSS fields are not read or written by the processor when the LTR instruction is executed. The LTR instruction loads the TSS descriptor into the TR and marks the task as busy, but it does not cause a task switch.

The TSS fields used by the processor in legacy mode are:

- *Link*—Bytes 01h–00h, dynamic field. Contains a copy of the task selector from the previously-executed task. See “Nesting Tasks” on page 389 for additional information.
- *Stack Pointers*—Bytes 1Bh–04h, static field. Contains the privilege 0, 1, and 2 stack pointers for the task. These consist of the stack-segment selector (SS*n)*, and the stack-segment offset (ESP*n*).
- *CR3*—Bytes 1Fh–1Ch, static field. Contains the page-translation-table base-address (CR3) register for the task.
- *EIP*—Bytes 23h–20h, dynamic field. Contains the instruction pointer (EIP) for the next instruction to be executed when the task is restored.
- *EFLAGS*—Bytes 27h–24h, dynamic field. Contains a copy of the EFLAGS image at the point the task is suspended.
- *General-Purpose Registers*—Bytes 47h–28h, dynamic field. Contains a copy of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI values at the point the task is suspended.
- *Segment-Selector Registers*—Bytes 59h–48h, dynamic field. Contains a copy of the ES, CS, SS, DS, FS, and GS, values at the point the task is suspended.
- *LDT Segment-Selector Register*—Bytes 63h–60h, static field. Contains the local-descriptor-table segment selector for the task.
- *T (Trap) Bit*—Bit 0 of byte 64h, static field. This bit, when set to 1, causes a debug exception (#DB) to occur on a task switch. See “Breakpoint Instruction (INT3)” on page 410 for additional information.
- *Shadow Stack Pointer (SSP)*—Bytes 6Bh-68h, static field. Contains the 32-bit SSP for the incoming task. Note that the SSP of the outgoing task is not saved in this field.
- *I/O-Permission Bitmap Base Address*—Bytes 67h–66h, static field. This field represents a 16-bit offset into the TSS. This offset points to the beginning of the I/O-permission bitmap, and the end of the interrupt-redirection bitmap.
- *I/O-Permission Bitmap*—Static field. This field specifies protection for I/O-port addresses (up to the 64K ports supported by the processor), as follows:

[AMD Public Use]

- Whether the port can be accessed at any privilege level.
- Whether the port can be accessed outside the privilege level established by EFLAGS.IOPL.
- Whether the port can be accessed when the processor is running in virtual-8086 mode.

Because one bit is used per 8-byte I/O-port, this bitmap can take up to 8 Kbytes of TSS space. The bitmap can be located anywhere within the first 64 Kbytes of the TSS, as long as it is above byte 103. The last byte of the bitmap must contain all ones (0FFh). See “I/O-Permission Bitmap” on page 377 for more information.

- *Interrupt-Redirection Bitmap*—Static field. This field defines how each of the 256-possible software interrupts is directed in a virtual-8086 environment. One bit is used for each interrupt, for a total bitmap size of 32 bytes. The bitmap can be located anywhere above byte 103 within the first 64 Kbytes of the TSS. See “Interrupt Redirection of Software Interrupts” on page 292 for information on using this field.

The TSS can be paged by system software. System software that uses the hardware task-switch mechanism must guarantee that a page fault does not occur during a task switch. Because the processor only reads and writes the first 104 TSS bytes during a task switch, this restriction only applies to those bytes. The simplest approach is to align the TSS on a page boundary so that all critical bytes are either present or not present. Then, if a page fault occurs when the TSS is accessed, it occurs before the first byte is read. If the page fault occurs after a portion of the TSS is read, the fault is unrecoverable.

I/O-Permission Bitmap. The I/O-permission bitmap (IOPB) allows system software to grant less-privileged programs access to individual I/O ports, overriding the effect of RFLAGS.IOPL for those devices. When an I/O instruction is executed, the processor checks the IOPB only if the processor is in virtual x86 mode or the CPL is greater than the RFLAGS.IOPL field. Each bit in the IOPB corresponds to a byte I/O port. A word I/O port corresponds to two consecutive IOPB bits, and a doubleword I/O port corresponds to four consecutive IOPB bits. Access is granted to an I/O port of a given size when *all* IOPB bits corresponding to that port are clear. If any bits are set, a #GP occurs.

The IOPB is located in the TSS, as shown by the example in Figure 12-7 on page 378. Each TSS can have a different copy of the IOPB, so access to individual I/O devices can be granted on a task-by-task basis. The I/O-permission bitmap base-address field located at byte 66h in the TSS is an offset into the TSS locating the start of the IOPB. If all 64K I/O ports are supported, the IOPB base address must not be greater than 0DFFFh, otherwise accesses to the bitmap cause a #GP to occur. An extra byte must be present after the last IOPB byte. This byte must have all bits set to 1 (0FFh). This allows the processor to read two IOPB bytes each time an I/O port is accessed. By reading two IOPB bytes, the processor can check all bits when unaligned, multi-byte I/O ports are accessed.

[AMD Public Use]

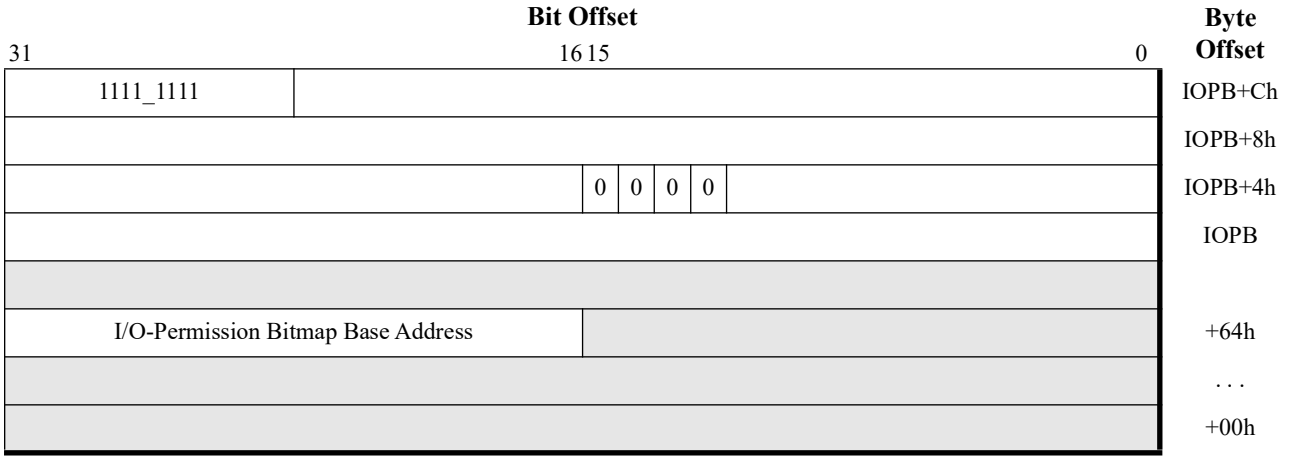


Figure 12-7. I/O-Permission Bitmap Example

Bits in the IOPB sequentially correspond to I/O port addresses. The example in Figure 12-7 shows bits 12 through 15 in the second doubleword of the IOPB cleared to 0. Those bit positions correspond to byte I/O ports 44h through 47h, or alternatively, doubleword I/O port 44h. Because the bits are cleared to zero, software running at any privilege level can access those I/O ports.

By adjusting the TSS limit, it may happen that some ports in the I/O-address space have no corresponding IOPB entry. Ports not represented by the IOPB will cause a #GP exception. Referring again to Figure 12-7, the last IOPB entry is at bit 23 in the fourth IOPB doubleword, which corresponds to I/O port 77h. In this example, all ports from 78h and above will cause a #GP exception, as if their permission bit was set to 1.

12.2.5 64-Bit Task State Segment

Although the hardware task-switching mechanism is not supported in long mode, a 64-bit task state segment (TSS) must still exist. System software must create at least one 64-bit TSS for use after activating long mode, and it must execute the LTR instruction, *in 64-bit mode*, to load the TR register with a pointer to the 64-bit TSS that serves both 64-bit-mode programs and compatibility-mode programs.

The legacy TSS contains several fields used for saving and restoring processor-state information. The legacy fields include general-purpose register, EFLAGS, CR3 and segment-selector register state, among others. Those legacy fields are not supported by the 64-bit TSS. System software must save and restore the necessary processor-state information required by the software-multitasking implementation (if multitasking is supported). Figure 12-8 on page 380 shows the format of a 64-bit TSS.

The 64-bit TSS holds several pieces of information important to long mode that are not directly related to the task-switch mechanism:

- *RSPn*—Bytes 1Bh–04h**.** The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0 through 2.

[AMD Public Use]

- *ISTn*—Bytes 5Bh–24h**.** The full 64-bit canonical forms of the interrupt-stack-table (IST) pointers. See “Interrupt-Stack Table” on page 287 for a description of the IST mechanism.
- *I/O Map Base Address*—Bytes 67h–66h**.** The 16-bit offset to the I/O-permission bit map from the 64-bit TSS base. The function of this field is identical to that in a legacy 32-bit TSS. See “I/O-Permission Bitmap” on page 377 for more information.

[AMD Public Use]

Bit Offset 31 16 15 0	Byte Offset
I/O-Permission Bitmap (IOPB) (Up to 8 Kbytes)	IOPB Base
- -	
I/O Map Base Address / Reserved, IGN	+64h
Reserved, IGN	+60h +5Ch
IST7[63:32]	+58h
IST7[31:0]	+54h
IST6[63:32]	+50h
IST6[31:0]	+4Ch
IST5[63:32]	+48h
IST5[31:0]	+44h
IST4[63:32]	+40h
IST4[31:0]	+3Ch
IST3[63:32]	+38h
IST3[31:0]	+34h
IST2[63:32]	+30h
IST2[31:0]	+2Ch
IST1[63:32]	+28h
IST1[31:0]	+24h
Reserved, IGN	+20h +1Ch
RSP2[63:32]	+18h
RSP2[31:0]	+14h
RSP1[63:32]	+10h
RSP1[31:0]	+0Ch
RSP0[63:32]	+08h
RSP0[31:0]	+04h
Reserved, IGN	+00h

Figure 12-8. Long Mode TSS Format

[AMD Public Use]

12.2.6 Task Gate Descriptor (Legacy Mode Only)

Task-gate descriptors hold a selector reference to a TSS and are used to control access between tasks. Unlike a TSS descriptor or other gate descriptors, a task gate can be located in any of the three descriptor tables (GDT, LDT, and IDT). Figure 12-9 shows the format of a task-gate descriptor.

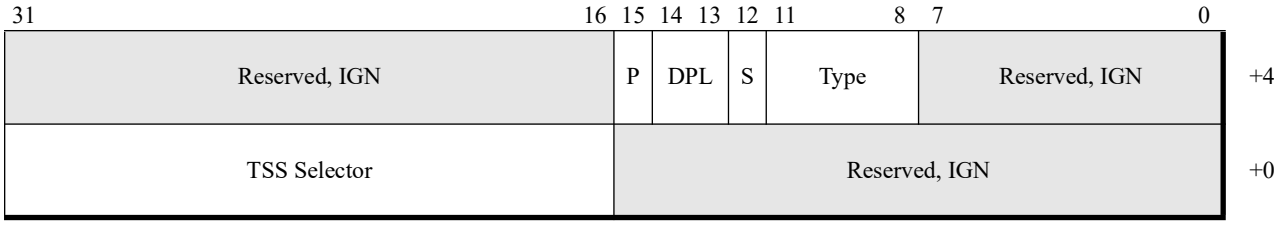


Figure 12-9. Task-Gate Descriptor, Legacy Mode Only

The task-gate descriptor fields are:

- *System (S) and Type*—Bits 12 and 11:8 (respectively) of byte +4. These bits are encoded by software as 00101b to indicate a task-gate descriptor type.
- *Present (P)*—Bit 15 of byte +4. The segment-present bit indicates the segment referenced by the gate descriptor is loaded in memory. If a reference is made to a segment when P=0, a segment-not-present exception (#NP) occurs. This bit is set and cleared by system software and is never altered by the processor.
- *Descriptor Privilege-Level (DPL)*—Bits 14:13 of byte +4. The DPL field indicates the gate-descriptor privilege level. DPL can be set to any value from 0 to 3, with 0 specifying the most privilege and 3 the least privilege.

12.3 Hardware Task-Management in Legacy Mode

This section describes the operation of the task-switch mechanism when the processor is running in legacy mode. None of these features are supported in long mode (either compatibility mode or 64-bit mode).

12.3.1 Task Memory-Mapping

The hardware task-switch mechanism gives system software a great deal of flexibility in managing the sharing and isolation of memory—both virtual (linear) and physical—between tasks.

Segmented Memory. The segmented memory for a task consists of the segments that are loaded during a task switch and any segments that are later accessed by the task code. The hardware task-switch mechanism allows tasks to either share segments with other tasks, or to access segments in isolation from one another. Tasks that share segments actually share a virtual-address (linear-address) space, but they do not necessarily share a physical-address space. When paging is enabled, the virtual-to-physical mapping for each task can differ, as is described in the following section. Shared segments

[AMD Public Use]

do share physical memory when paging is disabled, because virtual addresses are used as physical addresses.

A number of options are available to system software that shares segments between tasks:

- Sharing segment descriptors using the GDT. All tasks have access to the GDT, so it is possible for segments loaded in the GDT to be shared among tasks.
- Sharing segment descriptors using a single LDT. Each task has its own LDT, and that LDT selector is automatically saved and restored in the TSS by the processor during task switches. Tasks, however, can share LDTs simply by storing the same LDT selector in multiple TSSs. Using the LDT to manage segment sharing and segment isolation provides more flexibility to system software than using the GDT for the same purpose.
- Copying shared segment descriptors into multiple LDTs. Segment descriptors can be copied by system software into multiple LDTs that are otherwise not shared between tasks. Allowing segment sharing at the segment-descriptor level, rather than the LDT level or GDT level, provides the greatest flexibility to system software.

In all three cases listed above, the actual data and instructions are shared between tasks only when the tasks' virtual-to-physical address mappings are identical.

Paged Memory. Each task has its own page-translation table base-address (CR3) register, and that register is automatically saved and restored in the TSS by the processor during task switches. This allows each task to point to its own set of page-translation tables, so that each task can translate virtual addresses to physical addresses independently. Page translation must be enabled for changes in CR3 values to have an effect on virtual-to-physical address mapping. When page translation is disabled, the tables referenced by CR3 are ignored, and virtual addresses are equivalent to physical addresses.

12.3.2 Switching Tasks

The hardware task-switch mechanism transfers program control to a new task when any of the following occur:

- A CALL or JMP instruction with a selector operand that references a task gate is executed. The task gate can be located in either the LDT or GDT.
- A CALL or JMP instruction with a selector operand that references a TSS descriptor is executed. The TSS descriptor must be located in the GDT.
- A software-interrupt instruction (INT*n*) is executed that references a task gate located in the IDT.
- An exception or external interrupt occurs, and the vector references a task gate located in the IDT.
- An IRET is executed while the EFLAGS.NT bit is set to 1, indicating that a return is being performed from an inner-level task to an outer-level task. The new task is referenced using the selector stored in the current-task link field. See "Nesting Tasks" on page 389 for additional information. The RET instruction *cannot* be used to switch tasks.

When a task switch occurs, the following operations are performed automatically by the processor:

[AMD Public Use]

- The processor performs privilege-checking to determine whether the currently-executing program is allowed to access the target task. If this check fails, the task switch is aborted without modifying the processor state, and a general-protection exception (#GP) occurs. The privilege checks performed depend on the cause of the task switch:
  - If the task switch is initiated by a CALL or JMP instruction through a TSS descriptor, the processor checks that both the currently-executing program CPL and the TSS-selector RPL are numerically less-than or equal-to the TSS-descriptor DPL.
  - If the task switch takes place through a task gate, the CPL and task-gate RPL are compared with the task-gate DPL, and no comparison is made using the TSS-descriptor DPL. See “Task Switches Using Task Gates” on page 387.
  - Software interrupts, hardware interrupts, and exceptions all transfer control without checking the task-gate DPL.
  - The IRET instruction transfers control without checking the TSS-descriptor DPL.
- The processor performs limit-checking on the target TSS descriptor to verify that the TSS limit is greater than or equal to:
  - 67h (at least 104 bytes), when shadow stacks are not enabled (CR4.CET=0).
  - 6Bh (at least 108 bytes), when shadow stacks are enabled (CR4.CET=1).

  If this check fails, the task switch is aborted without modifying the processor state, and an invalid-TSS exception (#TS) occurs.
- If shadow stacks are enabled at the current CPL and the task switch was initiated by an IRET instruction, the current SSP must be aligned to 8 bytes. If this check fails, the task switch is aborted without modifying the processor state, and an #TS(current task) exception is generated.
- The current-task state is saved in the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers.

  Up to this point, any exception that occurs aborts the task switch without changing the processor state. From this point forward, any exception that occurs does so in the context of the new task. If an exception occurs in the context of the new task during a task switch, the processor finishes loading the new-task state without performing additional checks. The processor transfers control to the #TS handler after this state is loaded, but before the first instruction is executed in the new task. When a #TS occurs, it is possible that some of the state loaded by the processor did not participate in segment access checks. The #TS handler must verify that all segments are accessible before returning to the interrupted task.
- The task register (TR) is loaded with the new-task TSS selector, and the hidden portion of the TR is loaded with the new-task descriptor. The TSS now referenced by the processor is that of the new task.
- The current task is marked as busy. The previous task is marked as available or remains busy, based on the type of linkage. See “Nesting Tasks” on page 389 for more information.
- CR0.TS is set to 1. This bit can be used to save other processor state only when it becomes necessary. For more information, see the next section, “Saving Other Processor State.”
- If shadow stacks are enabled (CR4.CET=1), the following shadow stack actions are performed:

[AMD Public Use]


```
saveCsLipSsp = FALSE
checkCsLip   = FALSE
Read CS and EFLAGS from the incoming TSS
newCPL = (EFLAGS.VM == 1) ? 3 : CS.RPL

IF (task switch was initiated by CALL, interrupt or exception)
   {
   IF (ShadowStacksEnabled at current CPL)
      {
      IF ((current CPL == 3) && (newCPL < 3))
         PL3_SSP = SSP     // switching from user to supv
      ELSE
         {
         saveCsLipSsp = TRUE  // all other priv changes
         tempSSP = SSP
         tempLIP = CS.base + EIP
         tempCS  = CS.sel
         }
      }
   } //end task switch initated by CALL,int, or exception

ELSEIF (task switch was initiated by IRET)
   {
   IF (ShadowStacksEnabled at current CPL)
      {
      // pop CS, LIP and SSP from shadow stack
      IF ((newCPL == current CPL) || (newCPL < 3))
         {
         // no priv change, or supv to user/supv
         tempCS  = SSTK_READ_MEM.d [SSP+16]
         tempLIP = SSTK_READ_MEM.d [SSP+8]
         tempSSP = SSTK_READ_MEM.d [SSP]
         SSP = SSP + 24
```

[AMD Public Use]


```
        checkCsLip = TRUE
        }
      // check shadow stack token and clear busy bit
       temp_Token = SSTK_READ_MEM.q [SSP]    // read supervisor sstk token
       expected_Token = SSP OR 0x01          // busy bit must be set
       IF (temp_token == expected token)
            SSTK_WRITE_MEM.q [SSP] = SSP     // token OK, clear busy bit
       SSP = 0
      }      // end shadow stacks enabled at current CPL
   } //end task switch initiated by IRET
```

- The new-task state is loaded from the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers. The processor clears the segment-descriptor present (P) bits (in the hidden portion of the segment registers) to prevent access into the new segments, until the task switch completes successfully.
- The LDTR and CR3 registers are loaded from the TSS, changing the virtual-to-physical mapping from that of the old task to the new task. Because this is done in the middle of accessing the new TSS, system software must ensure that TSS addresses are translated identically in all tasks.
- The descriptors for all previously-loaded segment selectors are loaded into the hidden portion of the segment registers. This sets or clears the P bits for the segments as specified by the new descriptor values.
- If shadow stacks are enabled (CR4.CET=1), the following shadow stack actions are performed:

```
IF (ShadowStacksEnabled at current CPL)
   {
   IF (EFLAGS.VM == 1)
        EXCEPTION [#TSS(new task selector)]

   IF (task switch was initiated by a CALL, JMP, interrupt or exception)
      {
      newSSP = SSTK_READ_MEM.d [TSS offset 0x68]  // read new SSP
      IF (newSSP[2:0] != 0)                       // must be 8-byte aligned
           EXCEPTION [#TSS(new task selector)]
      // check token and set busy
```

[AMD Public Use]


```
      temp_Token = SSTK_READ_MEM.q [newSSP]      // read sstk token
      expected_Token = SSP                       // busy bit must be clear
      IF (temp_token != expected_token)          // token must be valid
            EXCEPTION [#TSS(new task selector)]
      SSTK_WRITE_MEM.q [SSP] = SSP OR 0x01      // valid token, set busy bit
      SSP = newSSP

      IF (saveCsLipSsP == TRUE) // push old CS,LIP,SSP onto new sstk
         {
         SSTK_WRITE_MEM.q [SSP-24] = tempCS
         SSTK_WRITE_MEM.q [SSP-16] = tempLIP
         SSTK_WRITE_MEM.q [SSP-8]  = tempSSP
         SSP = SSP - 24
         }
      } // end task switch initiated by CALL, JMP, interrupt or exception
   } // end shadow stacks enabled at current CPL

ELSEIF (task switch was initiated by an IRET)
   {
   IF (checkCsLip == TRUE)
      {                              // check CS, LIP against shadow stack
      IF (tempCS != CS)
            EXCEPTION [#CP(RETF/IRET)]  // CS must match
      IF (tempLIP != (CS.base + EIP))
            EXCEPTION [#CP(RETF/IRET)]  // LIP must match
      }

   IF ShadowStackEnabled at newCPL
      {
      IF (!checkCsLip)
```

[AMD Public Use]


```
      tempSSP = PL3_SSP;
   IF (tempSSP[1:0] != 0)           // SSP must be 4-byte aligned
         EXCEPTION [#CP(RETF/IRET)]
   IF (tempSSP[63:32] != 0)         // and SSP must be <4 GB
         EXCEPTION [#CP(RETF/IRET)]
   SSP = tempSSP
   }  // end shadow stacks enabled at new CPL
} // end task swtich initiated by IRET
```

If the above steps complete successfully, the processor begins executing instructions in the new task beginning with the instruction referenced by the CS:EIP far pointer loaded from the new TSS. The privilege level of the new task is taken from the new CS segment selector's RPL.

Saving Other Processor State. The processor does not automatically save the registers used by the media or x87 instructions. Instead, the processor sets CR0.TS to 1 during a task switch. Later, when an attempt is made to execute any of the media or x87 instructions while TS=1, a device-not-available exception (#NM) occurs. System software can then save the previous state of the media and x87 registers and clear the CR0.TS bit to 0 before executing the next media/x87 instruction. As a result, the media and x87 registers are saved only when necessary after a task switch.

12.3.3 Task Switches Using Task Gates

When a control transfer to a new task occurs through a task gate, the processor reads the task-gate DPL (DPL_G) from the task-gate descriptor. Two privilege checks, both of which must pass, are performed on DPL_G before the task switch can occur successfully:

- The processor compares the CPL with DPL_G. The CPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $CPL \leq DPL_G$.
- The processor compares the RPL in the task-gate selector with DPL_G. The RPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $RPL \leq DPL_G$.

Unlike call-gate control transfers, the processor does not read the DPL from the target TSS descriptor (DPL_S) and compare it with the CPL when a task gate is used.

Figure 12-10 on page 389 shows two examples of task-gate privilege checks. In Example 1, the privilege checks pass:

- The task-gate DPL (DPL_G) is at the lowest privilege (3), specifying that software running at any privilege level (CPL) can access the gate.
- The selector referencing the task gate passes its privilege check because the RPL is numerically less than or equal to DPL_G.

[AMD Public Use]

In Example 2, both privilege checks fail:

- The task-gate DPL (DPL_G) specifies that only software at privilege-level 0 can access the gate. The current program does not have enough privilege to access the task gate, because its CPL is 2.
- The selector referencing the task-gate descriptor does not have a high enough privilege to complete the reference. Its RPL is numerically greater than DPL_G.

Although both privilege checks failed in the example, if only one check fails, access into the target task is denied.

Because the legacy task-switch mechanism is not supported in long mode, *software cannot use task gates in long mode*. Any attempt to transfer control to another task using a task gate in long mode causes a general-protection exception (#GP) to occur.

[AMD Public Use]

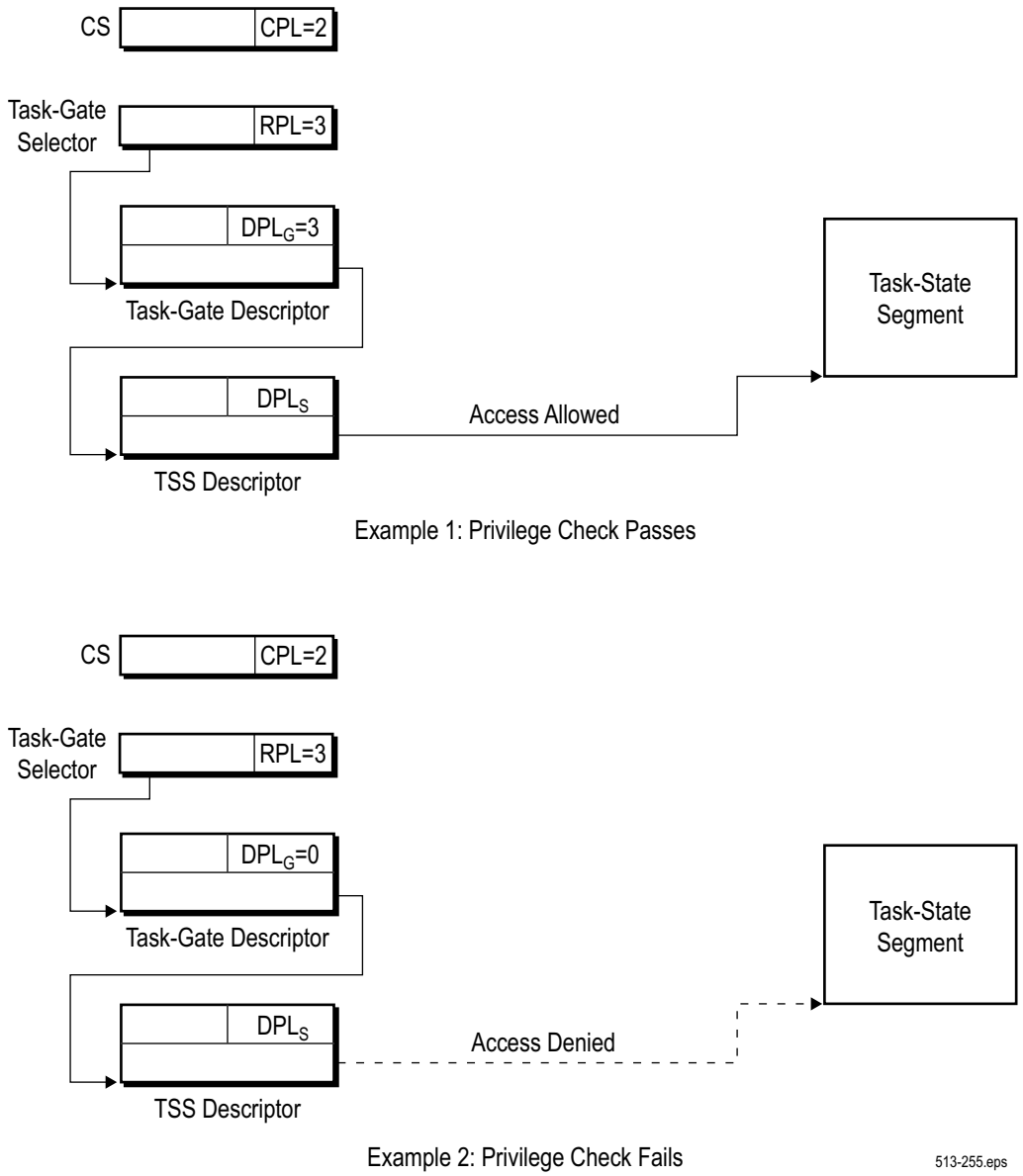


Figure 12-10. Privilege-Check Examples for Task Gates

12.3.4 Nesting Tasks

The hardware task-switch mechanism supports task nesting through the use of EFLAGS *nested-task* (NT) bit and the TSS link-field. The manner in which these fields are updated and used during a task switch depends on how the task switch is initiated:

- The JMP instruction does not update EFLAGS.NT or the TSS link-field. Task nesting is not supported by the JMP instruction.

[AMD Public Use]

- The CALL instruction, INT*n* instructions, interrupts, and exceptions can only be performed from outer-level tasks to inner-level tasks. All of these operations set the EFLAGS.NT bit for the new task to 1 during a task switch, and copy the selector for the previous task into the new-task link field.
- An IRET instruction which returns to another task only occurs when the EFLAGS.NT bit for the current task is set to 1, and only can be performed from an inner-level task to an outer-level task. When an IRET results in a task switch, the new task is referenced using the selector stored in the current-TSS link field. The EFLAGS.NT bit for the current task is cleared to 0 during the task switch.

Table 12-1 summarizes the effect various task-switch initiators have on EFLAGS.NT, the TSS link-field, and the TSS-busy bit. (For more information on the busy bit, see the next section, "Preventing Recursion.")

Table 12-1. Effects of Task Nesting

Task-Switch Initiator	Old Task			New Task		
	EFLAGS.NT	Link (Selector)	Busy	EFLAGS.NT	Link (Selector)	Busy
JMP	—	—	Clear to 0 (was 1)	—	—	Set to 1
CALL INT*n* Interrupt Exception	—		— (Was 1)	Set to 1	Old Task	Set to 1
IRET	Clear to 0 (was 1)	—	Clear to 0 (was 1)	—		

Note:
"—" indicates no change is made.

Programs running at any privilege level can set EFLAGS.NT to 1 and execute the IRET instruction to transfer control to another task. System software can keep control over improperly nested-task switches by initializing the link field of all TSSs that it creates. That way, improperly nested-task switches always transfer control to a known task.

Preventing Recursion. Task recursion is not allowed by the hardware task-switch mechanism. If recursive-task switches were allowed, they would replace a previous task-state image with a newer image, discarding the previous information. To prevent recursion from occurring, the processor uses the busy bit located in the TSS-descriptor type field (bit 9 of byte +4). Use of this bit depends on how the task switch is initiated:

- The JMP instruction clears the busy bit in the old task to 0 and sets the busy bit in the new task to 1. A general-protection exception (#GP) occurs if an attempt is made to JMP to a task with a set busy bit.
- The CALL instruction, INT*n* instructions, interrupts, and exceptions set the busy bit in the new task to 1. The busy bit in the old task remains set to 1, preventing recursion through task-nesting

[AMD Public Use]

levels. A general-protection exception (#GP) occurs if an attempt is made to switch to a task with a set busy bit.

- An IRET to another task (EFLAGS.NT must be 1) clears the busy bit in the old task to 0. The busy bit in the new task is not altered, because it was already set to 1.

Table 12-1 on page 390 summarizes the effect various task-switch initiators have on the TSS-busy bit.

[AMD Public Use]

13 Software Debug and Performance Resources

Testing, debug, and performance optimization consume a significant portion of the time needed to develop a new computer or software product and move it successfully into production. To stay competitive, product developers need tools that allow them to rapidly detect, isolate, and correct problems before a product is shipped. The goal of the debug and performance features incorporated into processor implementations of the AMD64 architecture is to support the tool chain solutions used in software and hardware product development.

The debug and performance resources that can be supported by AMD64 architecture implementations include:

- *Software Debug*—Software-debug facilities include the debug registers (DR0–DR7), debug exception, and breakpoint exception. Additional features are provided using model-specific registers (MSRs). These registers are used to set breakpoints on branches, interrupts, and exceptions and to single step from one branch to the next. The software-debug capability is described in the following section.
- *Performance Monitoring Counters*—Performance monitoring counters (PMCs) are provided to count specific processor hardware events. A set of control registers allow the selection of events to be monitored and a corresponding set of counter registers track the frequency of monitored events. These counters are described in Section 13.2 “Performance Monitoring Counters” on page 413.
- *Instruction-Based Sampling*— Instruction-based sampling is a hardware-based facility that enables system software to capture specific data concerning instruction fetch and instruction execution operation based on random sampling. This facility is described in Section 13.3 “Instruction-Based Sampling” on page 426.
- *Lightweight Profiling*—AMD64 architecture provides instructions that allow user-level programs to manage the gathering of instruction statistics using very little overhead. This facility is described in Section 13.4 “Lightweight Profiling” on page 460.

Although a subset of the facilities listed are available in all processor implementations, the remainder are optional. Support for optional facilities is indicated via CPUID feature bits. The means of determining support for each architected facility is described along with the facility in the sections that follow.

A given processor product may include additional debug and performance monitoring capabilities beyond those which are architecturally-defined. For details see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

13.1 Software-Debug Resources

Software can program breakpoints into the debug registers, causing a debug exception (#DB) when matches occur on instruction-memory addresses, data-memory addresses, or I/O addresses. The breakpoint exception (#BP) is also supported to allow software to set breakpoints by placing INT3

[AMD Public Use]

instructions in the instruction memory for a program. Program control is transferred to the breakpoint exception (#BP) handler when an INT3 instruction is executed.

In addition to the debug features supported by the debug registers (DR0–DR7), the processor also supports features supported by model-specific registers (MSRs). Together, these capabilities provide a rich set of breakpoint conditions, including:

- *Breakpoint On Address Match*—Breakpoints occur when the address stored in a address-breakpoint register matches the address of an instruction or data reference. Up to four address-match breakpoint conditions can be set by software.
- *Single Step All Instructions*—Breakpoints can be set to occur on every instruction, allowing a debugger to examine the contents of registers as a program executes.
- *Single Step Control Transfers*—Breakpoints can be set to occur on control transfers, such as calls, jumps, interrupts, and exceptions. This can allow a debugger to narrow a problem search to a specific section of code before enabling single stepping of all instructions.
- *Breakpoint On Any Instruction*—Breakpoints can be set on any specific instruction using either the address-match breakpoint condition or using the INT3 instruction to force a breakpoint when the instruction is executed.
- *Breakpoint On Task Switch*—Software forces a #DB exception to occur when a task switch is performed to a task with the T bit in the TSS set to 1. Debuggers can use this capability to enable or disable debug conditions for a specific task.
- *Breakpoint On Bus Lock* —The processor generates a #DB exception as a trap following the successful execution of a locked read-modify-write operation that requires a bus lock and the current privilege level (CPL) was > 0. See Section 13.1.3.6 on page 409 for more information about Bus Lock traps.

Problem areas can be identified rapidly using the information supplied by the debug registers when breakpoint conditions occur:

- Special conditions that cause a #DB exception are recorded in the DR6 debug-status register, including breakpoints due to task switches and single stepping. The DR6 register also identifies which address-breakpoint register (DR0–DR3) caused a #DB exception due to an address match. When combined with the DR7 debug-control register settings, the cause of a #DB exception can be identified.
- To assist in analyzing the instruction sequence a processor follows in reaching its current state, the source and destination addresses of control-transfer events are saved by the processor. These include branches (calls and jumps), interrupts, and exceptions. Debuggers can use this information to narrow a problem search to a specific section of code before single stepping all instructions.

13.1.1 Debug Registers

The AMD64 architecture supports the legacy debug registers, DR0–DR7. These registers are expanded to 64 bits by the AMD64 architecture. In legacy mode and in compatibility mode, only the lower 32 bits are used. In these modes, writes to a debug register fill the upper 32 bits with zeros, and

[AMD Public Use]

reads from a debug register return only the lower 32 bits. In 64-bit mode, all 64 bits of the debug registers are read and written. Operand-size prefixes are ignored.

The debug registers can be read and written only when the current-protection level (CPL) is 0 (most privileged). Attempts to read or write the registers at a lower-privilege level (CPL>0) cause a general-protection exception (#GP).

Several debug registers described below are model-specific registers (MSRs). See “Software-Debug MSRs” on page 771 for a listing of the debug-MSR numbers and their reset values. Some processor implementations include additional MSRs used to support implementation-specific software debug features. For more information on these registers and their capabilities, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

13.1.1.1 Address-Breakpoint Registers (DR0-DR3)

Figure 13-1 shows the format of the four address-breakpoint registers, DR0-DR3. Software can load a virtual (linear) address into any of the four registers, and enable breakpoints to occur when the address matches an instruction or data reference. The MOV DR*n* instructions *do not* check that the virtual addresses loaded into DR0–DR3 are in canonical form. Breakpoint conditions are enabled using the debug-control register, DR7 (see “Debug-Control Register (DR7)” on page 396).

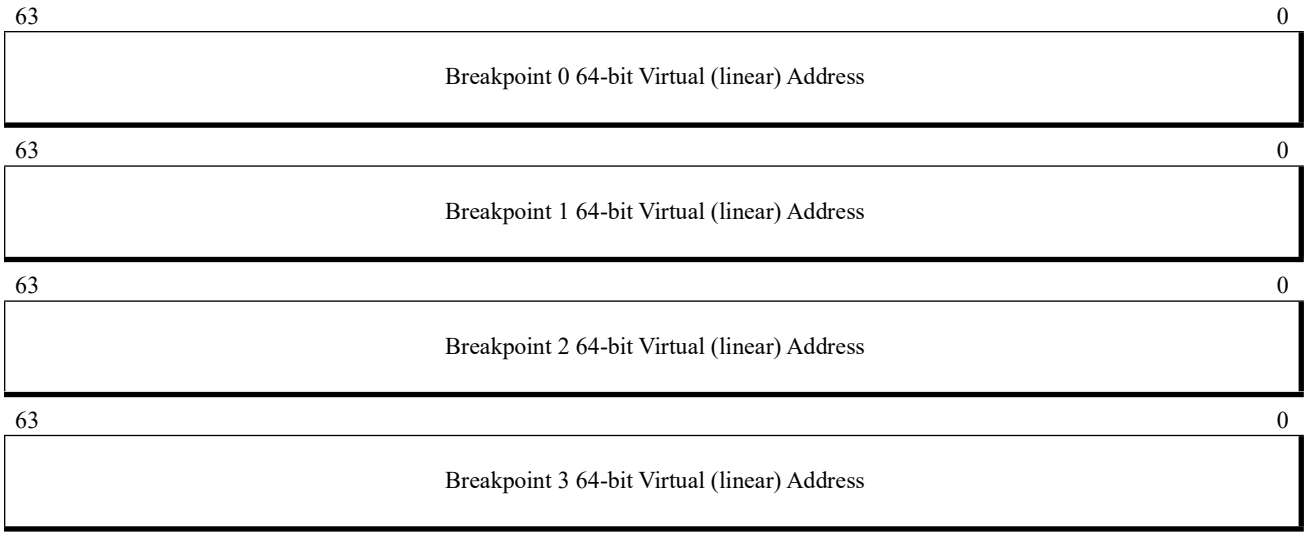


Figure 13-1. Address-Breakpoint Registers (DR0–DR3)

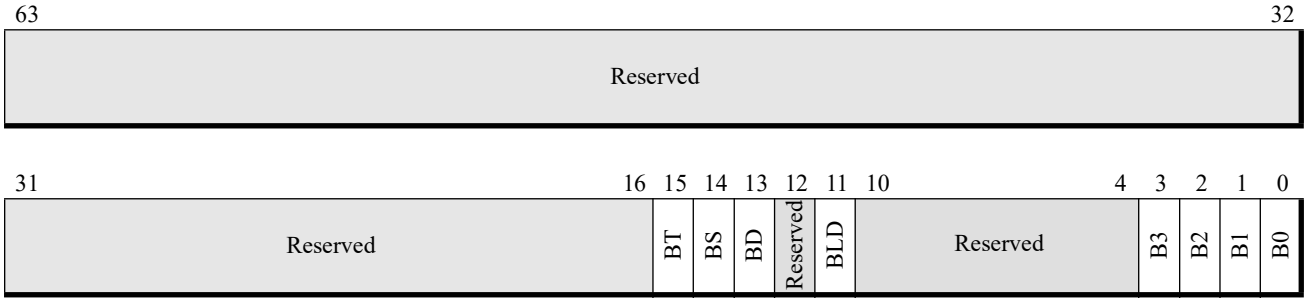
13.1.1.2 Reserved Debug Registers (DR4, DR5)

The DR4 and DR5 registers are reserved and should not be used by software. These registers are aliased to the DR6 and DR7 registers, respectively. When the debug extensions are enabled (CR4[DE] = 1) attempts to access these registers cause an invalid-opcode exception (#UD).

[AMD Public Use]

13.1.1.3 Debug-Status Register (DR6)

Figure 13-2 on page 395 shows the format of the debug-status register, DR6. Debug status is loaded into DR6 when an enabled debug condition is encountered that causes a #DB exception.



Bits	Mnemonic	Description	Access type
63:32	Reserved		MBZ
31:16	Reserved		RA1
15	BT	Breakpoint Task Switch	R/W
14	BS	Breakpoint Single Step	R/W
13	BD	Breakpoint Debug Access Detected	R/W
12	Reserved		RAZ
11	BLD	Bus Lock Detected	R/W
10:4	Reserved		RA1
3	B3	Breakpoint #3 Condition Detected	R/W
2	B2	Breakpoint #2 Condition Detected	R/W
1	B1	Breakpoint #1 Condition Detected	R/W
0	B0	Breakpoint #0 Condition Detected	R/W

Figure 13-2. Debug-Status Register (DR6)

Bits 15:13 of the DR6 register are not cleared by the processor and must be cleared by software after the contents have been read. Register fields are:

- *Breakpoint-Condition Detected (B3–B0)*—Bits 3:0. The processor updates these four bits on every debug breakpoint or general-detect condition. A bit is set to 1 if the corresponding address-breakpoint register detects an enabled breakpoint condition, as specified by the DR7 L*n,* G*n,* R/W*n* and LEN*n* controls, and is cleared to 0 otherwise. For example, B1 (bit 1) is set to 1 if an address-breakpoint condition is detected by DR1.
- *Bus Lock Detected (BLD)*—Bit 11. The processor clears this bit if #DB was generated due to a bus lock. Other sources of #DB do not modify this bit.
- *Debug-Register-Access Detected (BD)*—Bit 13. The processor sets this bit to 1 if software accesses any debug register (DR0–DR7) while the general-detect condition is enabled (DR7[GD] = 1).
- *Single Step (BS)*—Bit 14. The processor sets this bit to 1 if the #DB exception occurs as a result of single-step mode (rFLAGS[TF] = 1). Single-step mode has the highest-priority among debug

[AMD Public Use]

exceptions. Other status bits within the DR6 register can be set by the processor along with the BS bit.

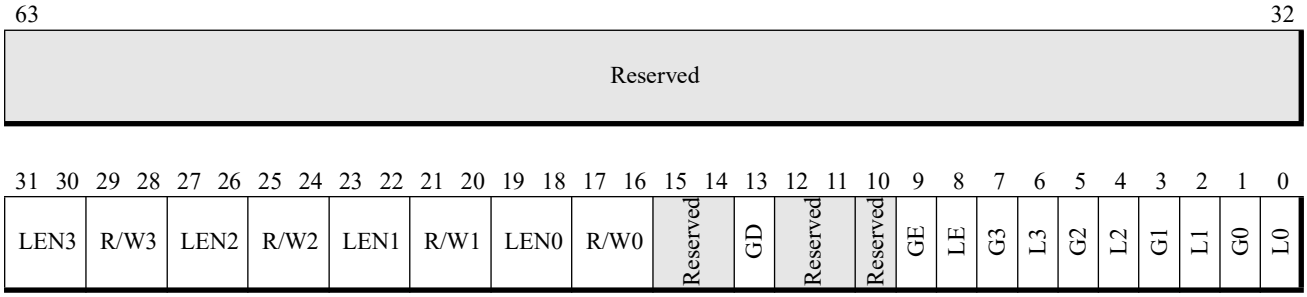
- *Task-Switch (BT)*—Bit 15. The processor sets this bit to 1 if the #DB exception occurred as a result of task switch to a task with a TSS T-bit set to 1.

All remaining bits in the DR6 register are reserved. Reserved bits 31:16 and 11:4 must all be set to 1, while reserved bit 12 must be cleared to 0. In 64-bit mode, the upper 32 bits of DR6 are reserved and must be written with zeros. Writing a 1 to any of the upper 32 bits results in a general-protection exception, #GP(0).

13.1.1.4 Debug-Control Register (DR7)

Figure 13-3 shows the format of the debug-control register, DR7. DR7 is used to establish the breakpoint conditions for the address-breakpoint registers (DR0–DR3) and to enable debug exceptions for each address-breakpoint register individually. DR7 is also used to enable the general-detect breakpoint condition.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:32	Reserved		MBZ
31:30	LEN3	Length of Breakpoint #3	R/W
29:28	R/W3	Type of Transaction(s) to Trap	R/W
27:26	LEN2	Length of Breakpoint #2	R/W
25:24	R/W2	Type of Transaction(s) to Trap	R/W
23:22	LEN1	Length of Breakpoint #1	R/W
21:20	R/W1	Type of Transaction(s) to Trap	R/W
19:18	LEN0	Length of Breakpoint #0	R/W
17:16	R/W0	Type of Transaction(s) to Trap	R/W
15:14	Reserved		RAZ
13	GD	General Detect Enabled	R/W
12:11	Reserved		RAZ
10	Reserved		RA1
9	GE	Global Exact Breakpoint Enabled	R/W
8	LE	Local Exact Breakpoint Enabled	R/W
7	G3	Global Exact Breakpoint #3 Enabled	R/W
6	L3	Local Exact Breakpoint #3 Enabled	R/W
5	G2	Global Exact Breakpoint #2 Enabled	R/W
4	L2	Local Exact Breakpoint #2 Enabled	R/W
3	G1	Global Exact Breakpoint #1 Enabled	R/W
2	L1	Local Exact Breakpoint #1 Enabled	R/W
1	G0	Global Exact Breakpoint #0 Enabled	R/W
0	L0	Local Exact Breakpoint #0 Enabled	R/W

Figure 13-3. Debug-Control Register (DR7)

The fields within the DR7 register are all read/write. These fields are:

- *Local-Breakpoint Enable (L3–L0)*—Bits 6, 4, 2, and 0 (respectively). Software individually sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DR*n*) detects a breakpoint condition while executing the *current* task. For example, if L1 (bit 2) is set to 1 and an address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are cleared to 0 by the processor when a hardware task-switch occurs.
- *Global-Breakpoint Enable (G3–G0)*—Bits 7, 5, 3, and 1 (respectively). Software sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DR*n*) detects a breakpoint condition while executing *any* task. For example, if G1 (bit 3) is set to 1 and an

[AMD Public Use]

address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are never cleared to 0 by the processor.

- *Local-Enable (LE)*—Bit 8. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing the *current* task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.
- *Global-Enable (GE)*—Bit 9. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing *any* task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.
- *General-Detect Enable (GD)*—Bit 13. Software sets this bit to 1 to cause a debug exception to occur when an attempt is made to execute a MOV DR*n* instruction to any debug register (DR0–DR7). This bit is cleared to 0 by the processor when the #DB handler is entered, allowing the handler to read and write the DR*n* registers. The #DB exception occurs before executing the instruction, and DR6[BD] is set by the processor. Software debuggers can use this bit to prevent the currently-executing program from interfering with the debug operation.
- *Read/Write (R/W3–R/W0)*—Bits 29:28, 25:24, 21:20, and 17:16 (respectively). Software sets these fields to control the breakpoint conditions used by the corresponding address-breakpoint registers (DR*n*). For example, control-field R/W1 (bits 21:20) controls the breakpoint conditions for the DR1 register. The R/W*n* control-field encodings specify the following conditions for an address-breakpoint to occur:
  - 00—Only on instruction execution.
  - 01—Only on data write.
  - 10—This encoding is further qualified by CR4[DE] as follows:
    . CR4[DE] = 0—Condition is undefined.
    . CR4[DE] = 1—Only on I/O read or I/O write.
  - 11—Only on data read or data write.
- *Length (LEN3–LEN0)*—Bits 31:30, 27:26, 23:22, and 19:18 (respectively). Software sets these fields to control the range used in comparing a memory address with the corresponding address-breakpoint register (DR*n*). For example, control-field LEN1 (bits 23:22) controls the breakpoint-comparison range for the DR1 register.

  The value in DR*n* defines the low-end of the address range used in the comparison. LEN*n* is used to mask the low-order address bits in the corresponding DR*n* register so that they are not used in the address comparison. To work properly, breakpoint boundaries must be aligned on an address corresponding to the range size specified by LEN*n*. The LEN*n* control-field encodings specify the following address-breakpoint-comparison ranges:
  - 00—1 byte.
  - 01—2 byte, must be aligned on a word boundary.
  - 10—8 byte, must be aligned on a quadword boundary. (Long mode only; otherwise undefined.)

[AMD Public Use]

- 11—4 byte, must be aligned on a doubleword boundary.

If the R/W*n* field is used to specify instruction breakpoints (R/W*n*=00), the corresponding LEN*n* field must be set to 00. Setting LEN*n* to any other value produces undefined results.

All remaining bits in the DR7 register are reserved. Reserved bits 15:14 and 12:11 must all be cleared to 0, while reserved bit 10 must be set to 1. In 64-bit mode, the upper 32 bits of DR7 are reserved and must be written with zeros. Writing a 1 to any of the upper 32 bits results in a general-protection #GP(0) exception.

13.1.1.5 64-Bit-Mode Extended Debug Registers

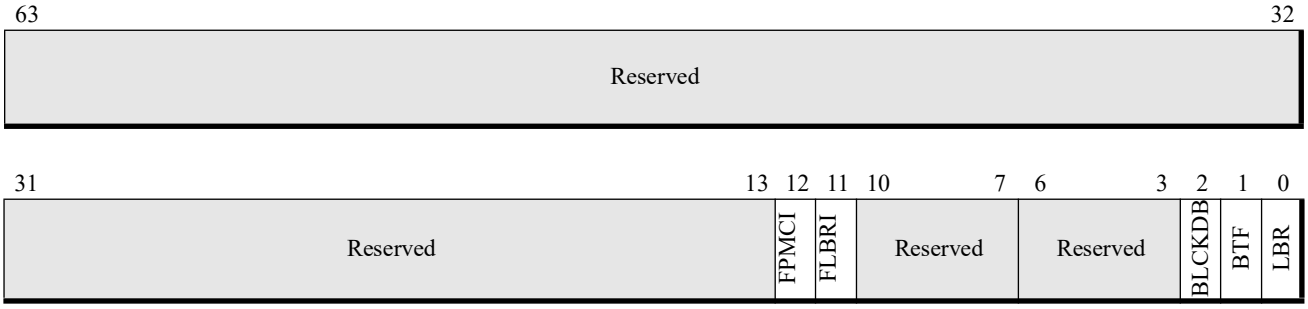
In 64-bit mode, additional encodings for debug registers are available. The R bit of the REX prefix is used to modify the ModRM *reg* field when that field encodes a control register. These additional encodings enable the processor to address DR8–DR15.

Access to the DR8–DR15 registers is implementation-dependent. The architecture does not require any of these extended debug registers to be implemented. Any attempt to access an unimplemented register results in an invalid-opcode exception (#UD).

13.1.1.6 Debug-Control MSR (DebugCtl)

Figure 13-4 on page 400 shows the format of the debug-control MSR (DebugCtl). DebugCtl provides additional debug controls over control-transfer recording and single stepping, and external-breakpoint reporting and trace messages. DebugCtl is read and written using the RDMSR and WRMSR instructions.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:13	Reserved		RAZ
12	FPMCI	Freeze Core Performance Monitor Counters on PMC interrupt	R/W
11	FLBRI	Freeze LBR Stack on PMC interrupt	R/W
10:7	Reserved		RAZ
6:3	Reserved		MBZ
2	BLCKDB	Bus Lock #DB trap enabled	R/W
1	BTF	Branch Single Step	R/W
0	LBR	Last-Branch Record	R/W

Figure 13-4. Debug-Control MSR (DebugCtl)

The fields within the DebugCtl register are:

- *Last-Branch Record (LBR)*—Bit 0, read/write. Software sets this bit to 1 to cause the processor to record the source and target addresses of the last control transfer taken before a debug exception occurs. The recorded control transfers include branch instructions, interrupts, and exceptions. See "Control-Transfer Breakpoint Features" on page 411 for more details on the registers. See Figure 13-5 on page 401 for the format of the control-transfer recording MSRs.
- *Branch Single Step (BTF)*—Bit 1, read/write. Software uses this bit to change the behavior of the rFLAGS[TF] bit. When this bit is cleared to 0, the rFLAGS[TF] bit controls instruction single stepping, (normal behavior). When this bit is set to 1, the rFLAGS[TF] bit controls single stepping on control transfers. The single-stepped control transfers include branch instructions, interrupts, and exceptions. Control-transfer single stepping requires both BTF = 1 and rFLAGS[TF] = 1. See "Control-Transfer Breakpoint Features" on page 411 for more details on control-transfer single stepping.
- *Bus Lock #DB Trap (BLCKDB)*—Bit 2, read/write. Software sets this bit to enable generation of a #DB trap following successful exection of a bus lock when CPL is > 0. See Section 7.3.3 on page 197 for information on the conditions that cause a bus lock. Section 13.1.3.6 on page 409 describes bus lock #DB trap in more detail.
- *Freeze LBR Stack on PMC Interrupt (FLBRI)*—Bit 11, read/write. Software sets this bit to freeze the Last Branch Record (LBR) Stack when one of the Core Performance Counters overflows and is

[AMD Public Use]

configured to signal a PMC overflow event. Support for this feature is indicated by CPUID Fn8000_0022_EAX[LbrAndPmcFreeze](bit 2) = 1.

- *Freeze PMC on PMC interrupt (FPMCI)*—Bit 12, read/write. Software sets this bit to freeze the Core Performance Counters when one of these counters overflows and is configured to signal a PMC overflow event. Support for this feature is indicated by CPUID Fn8000_0022_EAX[LbrAndPmcFreeze](bit 2) = 1.

All remaining bits in the DebugCtl register are reserved.

13.1.1.7 Control-Transfer Recording MSRs

Figure 13-5 on page 401 shows the format of the 64-bit control-transfer recording MSRs: LastBranchToIP, LastBranchFromIP, LastIntToIP, and LastIntFromIP. These registers are loaded automatically by the processor when the DebugCtl[LBR] bit is set to 1. These MSRs are read-only.

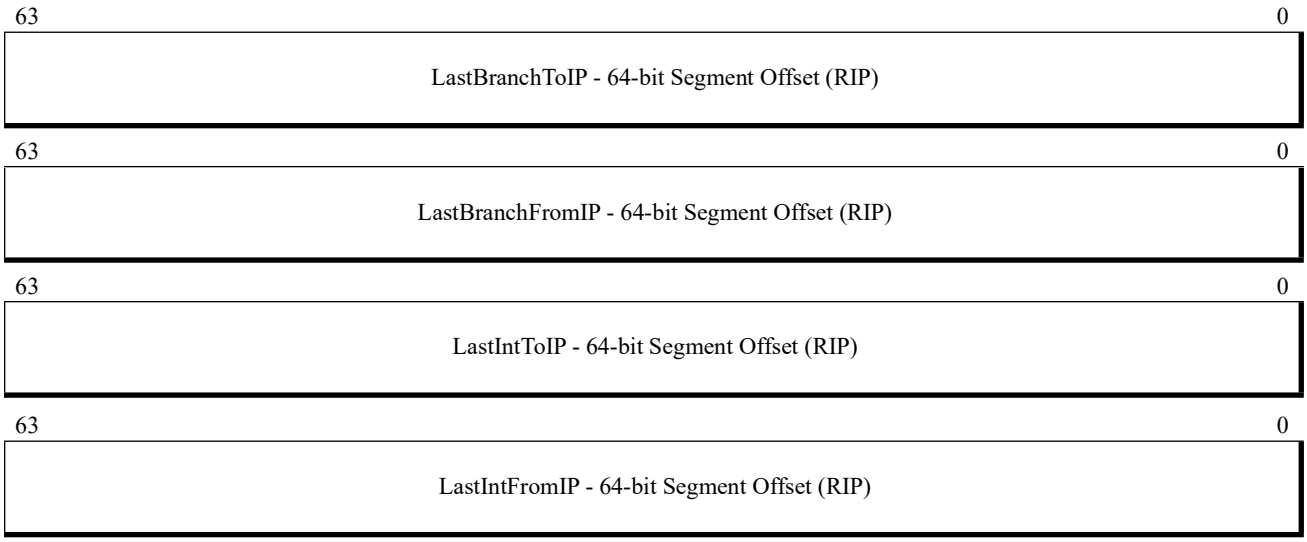
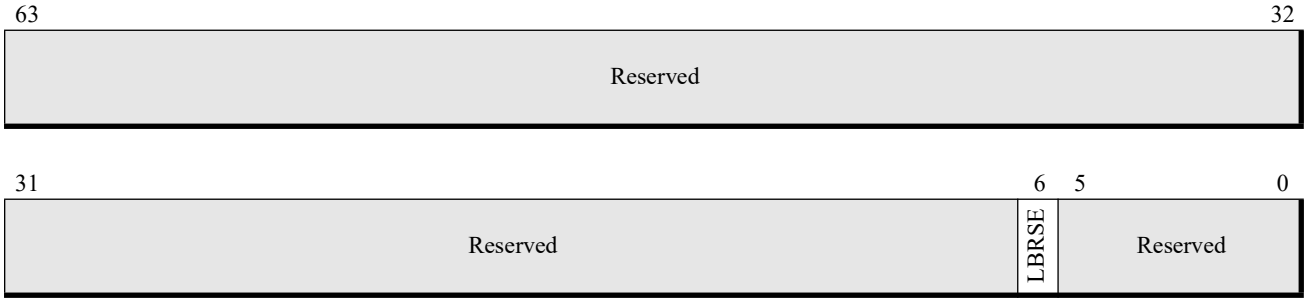


Figure 13-5. Control-Transfer Recording MSRs

13.1.1.8 Debug-Extension-Control MSR (DebugExtnCtl)

Figure 13-6 shows the format of the Debug-Extension-Control MSR (DebugExtnCtl). DebugExtnCtl is supported on processors that support Last Branch Record Stack (see Section 13.1.8, “Last Branch Record Stack,” on page 412) and provides control over enabling Last Branch Record Stack recording.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:7	Reserved		MBZ
6	LBRSE	LBR Stack Enable	R/W
5:0	Reserved		MBZ

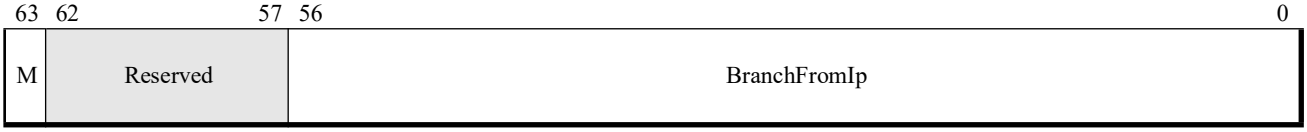
Figure 13-6. Debug-Extension-Control MSR (DebugExtnCtl)

The fields within the DebugExtnCtl register are:

- *Last-Branch Record Stack Enable (LBRSE)*—Bit 6, read/write. Software sets this bit to 1 to cause the source and target address of control transfer instructions to be recorded on the LBR stack. The processor clears this bit when #DB exception occurs. Support for LBR Stack is indicated by CPUID Fn8000_0022_EAX[LbrStack](bit 1) = 1.

13.1.1.9 Last Branch Stack Registers

Figure 13-7 and Figure 13-8 show the format of the LastBranchStackFromIp and LastStackToIp MSRs. These MSRs provide access to the Last Branch Record Stack. These MSRs are supported on processors with CPUID Fn8000_0022_EAX[LbrStack] = 1. There are LBR_SIZE = CPUID Fn8000_0022_EBX[LbrStackSize] instances for each MSR. The Last Branch Record functionality is described in Section 13.1.8, “Last Branch Record Stack,” on page 412.



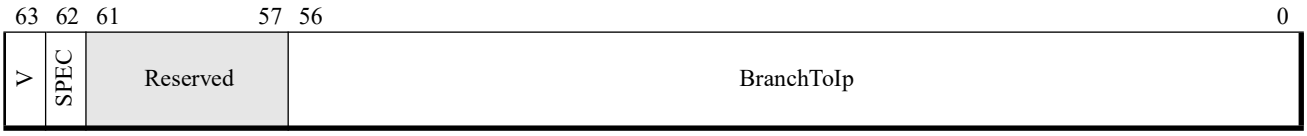
Bits	Mnemonic	Description	Access type
63	M	Mispredict	R/W
62:57	Reserved		
56:0	BranchFromIp	Segment offset of the recorded branch	R/W

Figure 13-7. LastBranchStackFromIp MSR

[AMD Public Use]

The fields within LastBranchStackFromIp are:

- *Mispredict*—Bit 63, read/write. The processor sets this bit to 1 when the recorded branch was mispredicted.
- *BranchFromIp*—Bits 56:0, read/write. The processor records either the segment offset of the branch itself or the instruction preceding the branch. It can only record the segment offset of an instruction preceding the branch if that instruction is not a branch itself. Recording the IP of an instruction preceding the branch is the result of branch fusion. If it is desired that BranchFromIp only records the address of branches, Branch Fusion needs to be disabled by also enabling Legacy LBR via DbgCfg[LBR].



Bits	Mnemonic	Description	Access type
63	V	Valid	R/W
62	SPEC	Speculative	R/W
61:57	Reserved		
56:0	BranchToIp	BranchToIp	R/W

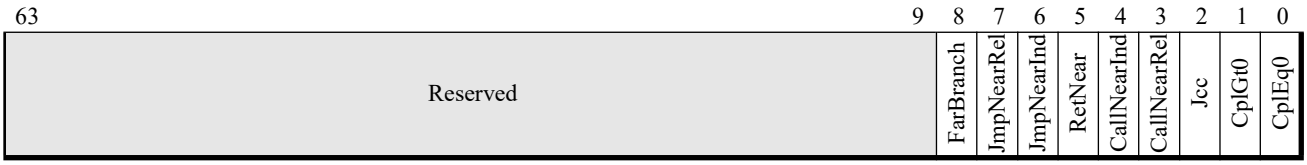
Figure 13-8. LastBranchStackToIp MSR

The fields within LastBranchStackToIp are:

- *Valid*—Bit 63, read/write. The processor sets this bit to 1 when the recorded branch entry is valid.
- *Speculative*—Bit 62, read/write. The processor sets this bit when the branch entry was recorded when a speculative performance feature was active.
- *BranchToIp*—Bits 56:0, read/write. The processor records the segment offset of the branch target.

Figure 13-9 shows the format of the LastBranchStackSelect MSR. LastBranchStackSelect controls the recording of branches to the Last Branch Record Stack. This MSR is supported on processors with CPUID Fn8000_0022_EAX[LbrStack] = 1.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:9	Reserved		RAZ
8	FarBranch	Suppress recording of far branches	R/W
7	JmpNearRel	Suppress recording of near relative jumps	R/W
6	JumpNearInd	Suppress recording of near indirect jumps	R/W
5	RetNear	Suppress recording of near returns	R/W
4	CallNearInd	Suppress recording of indirect near calls	R/W
3	CallNearRel	Suppress recording of relative near calls	R/W
2	Jcc	Suppress recording of conditional branches	R/W
1	CplGt0	Suppress recording of branches ending in CPL > 0	R/W
0	CplEq0	Suppress recording of branches ending in CPL = 0	R/W

Figure 13-9. LastBranchStackSelect MSR

The fields within LbrSelect are:

- *CplEq0*—Bit 0, read/write. When set, the LBR Stack does not record branches ending in CPL = 0.
- *CplGt0*—Bit 1, read/write. When set, the LBR Stack does not record branches ending in CPL > 0.
- *Jcc*—Bit 2, read/write. When set, the LBR Stack does not record conditional near branches (Jcc, JrCXZ, LOOP, and LOOPcc).
- *CallNearRel*—Bit 3, read/write. When set, the LBR Stack does not record direct (relative) near calls.
- *CallNearInd*—Bit 4, read/write. When set, the LBR Stack does not record indirect near calls.
- *RetNear*—Bit 5, read/write. When set, the LBR Stack does not record near returns.
- *JmpNearInd*—Bit 6, read/write. When set, the LBR Stack does not record near indirect jumps.
- *JmpNearRel*—Bit 7, read/write. When set, the LBR Stack does not record near direct (relative) jumps.
- *FarBranch*—Bit 8, read/write. When set, the LBR Stack does not record far branches.

13.1.2 Setting Breakpoints

Breakpoints can be set to occur on either instruction addresses or data addresses using the breakpoint-address registers, DR0–DR3 (DR*n*). The values loaded into these registers represent the breakpoint-location virtual address. The debug-control register, DR7, is used to enable the breakpoint registers and to specify the type of access and the range of addresses that can trigger a breakpoint.

[AMD Public Use]

Software enables the DR*n* registers using the corresponding local-breakpoint enable (L*n*) or global-breakpoint enable (G*n*) found in the DR7 register. L*n* is used to enable breakpoints only while the current task is active, and it is cleared by the processor when a task switch occurs. G*n* is used to enable breakpoints for all tasks, and it is never cleared by the processor.

The R/W*n* fields in DR7, along with the CR4[DE] bit, specify the type of access required to trigger a breakpoint when an address match occurs on the corresponding DR*n* register. Breakpoints can be set to occur on instruction execution, data reads and writes, and I/O reads and writes. The R/W*n* and CR4[DE] encodings used to specify the access type are described on page 398 of "Debug-Control Register (DR7)."

The LEN*n* fields in DR7 specify the size of the address range used in comparison with data or instruction addresses. LEN*n* is used to mask the low-order address bits in the corresponding DR*n* register so that they are not used in the address comparison. Breakpoint boundaries must be aligned on an address corresponding to the range size specified by LEN*n*. Assuming the access type matches the type specified by R/W*n*, a breakpoint occurs if any accessed byte falls within the range specified by LEN*n*. For instruction breakpoints, LEN*n* must specify a single-byte range. The LEN*n* encodings used to specify the address range are described on page 398 of "Debug-Control Register (DR7)."

Table 13-1 shows several examples of data accesses, and whether or not they cause a #DB exception to occur based on the breakpoint address in DR*n* and the breakpoint-address range specified by LEN*n*. In this table, R/W*n* always specifies read/write access.

Table 13-1. Breakpoint-Setting Examples

Data-Access Address (hexadecimal)	Access Size (bytes)	Byte-Addresses in Data-Access (hexadecimal)	Breakpoint-Address Range (hexadecimal)	Result
DR*n*=F000, LEN*n*=00 (1 Byte)				
DR*n*=F005, LEN*n*=10 (8 Bytes)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F000–F007	#DB
EFFE	2	EFFE, EFFF		—
	4	EFFE, EFFF, F000, F001		#DB
F000	1	F000		
F001	2	F001, F002		
F005	4	F005, F006, F007, F008		

Note:
"—" indicates no #DB occurs.

[AMD Public Use]

Table 13-1. Breakpoint-Setting Examples *(continued)*

Data-Access Address (hexadecimal)	Access Size (bytes)	Byte-Addresses in Data-Access (hexadecimal)	Breakpoint-Address Range (hexadecimal)	Result
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F000	#DB
EFFE	2	EFFE, EFFF		—
	4	EFFE, EFFF, F000, F001		#DB
F000	1	F000		
F001	2	F001, F002		—
F005	4	F005, F006, F007, F008		
DR*n*=F004, LEN*n*=11 (4 Bytes)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F004–F007	—
EFFE	2	EFFE, EFFF		
	4	EFFE, EFFF, F000, F001		
F000	1	F000		
F001	2	F001, F002		
F005	4	F005, F006, F007, F008		#DB
DR*n*=F005, LEN*n*=10 (8 Bytes)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F000–F007	#DB
EFFE	2	EFFE, EFFF		—
	4	EFFE, EFFF, F000, F001		#DB
F000	1	F000		
F001	2	F001, F002		
F005	4	F005, F006, F007, F008		

Note:
"—" indicates no #DB occurs.

13.1.3 Using Breakpoints

A debug exception (#DB) occurs when an enabled-breakpoint condition is encountered during program execution. The debug-handler must check the debug-status register (DR6), the conditions enabled by the debug-control register (DR7), and the debug-control MSR (DebugCtl), to determine the #DB cause. The #DB exception corresponds to interrupt vector 1. See "#DB—Debug Exception (Vector 1)" on page 250.

Instruction breakpoints and general-detect conditions cause the #DB exception to occur *before* the instruction is executed, while all other breakpoint and single-stepping conditions cause the #DB exception to occur *after* the instruction is executed. Table 13-2 summarizes where the #DB exception occurs based on the breakpoint condition.

[AMD Public Use]

Table 13-2. Breakpoint Location by Condition

Breakpoint Condition	Breakpoint Location
Instruction	Before Instruction is Executed
General Detect	
Data Write Only	After Instruction is Executed[1]
Data Read or Data Write	
I/O Read or I/O Write	
Single Step[1]	After Instruction is Executed
Task Switch	

Note:
1. Repeated operations (REP prefix) can breakpoint between iterations.

Instruction breakpoints and general-detect conditions have a lower interrupt-priority than the other breakpoint and single-stepping conditions (see “Priorities” on page 266). Data-breakpoint conditions on the *previous* instruction occur before an instruction-breakpoint condition on the *next* instruction. However, if instruction and data breakpoints can occur as a result of executing a *single* instruction, the instruction breakpoint occurs first (before the instruction is executed), followed by the data breakpoint (after the instruction is executed).

13.1.3.1 Instruction Breakpoints

Instruction breakpoints are set by loading a breakpoint-address register (DR*n*) with the desired instruction virtual-address, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 00b to specify that the contents of DR*n* are to be compared only with the virtual address of the next instruction to be executed.
- LEN*n* must be set to 00b.

When a #DB exception occurs due to an instruction breakpoint-address in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs before the instruction is executed, and the breakpoint-instruction address is pushed onto the debug-handler stack. If multiple instruction breakpoints are set, the debug handler can use the B*n* field to identify which register caused the breakpoint.

Returning from the debug handler causes the breakpoint instruction to be executed. Before returning from the debug handler, the rFLAGS[RF] bit should be set to 1 to prevent a re-occurrence of the #DB exception due to the instruction-breakpoint condition. The processor ignores instruction-breakpoint conditions when rFLAGS[RF] = 1, until after the next instruction (in this case, the breakpoint instruction) is executed. After the next instruction is executed, the processor clears rFLAGS[RF].

[AMD Public Use]

13.1.3.2 Data Breakpoints

Data breakpoints are set by loading a breakpoint-address register (DR*n*) with the desired data virtual-address, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 01b to specify that the data virtual-address is compared with the contents of DR*n* only during a memory-write. Setting this field to 11b specifies that the comparison takes place during both memory reads and memory writes.
- LEN*n* is set to 00b, 01b, 11b, or 10b to specify an address-match range of one, two, four, or eight bytes, respectively. Long mode must be active to set LEN*n* to 10b.

When a #DB exception occurs due to a data breakpoint address in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the data-access instruction is executed, which means that the original data is overwritten by the data-access instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

Because the breakpoint occurs after the data-access instruction is executed, the address of the instruction following the data-access instruction is pushed onto the debug-handler stack. Repeated string instructions, however, can trigger a breakpoint before all iterations of the repeat loop have completed. When this happens, the address of the string instruction is pushed onto the stack during a #DB exception if the repeat loop is not complete. A subsequent IRET from the #DB handler returns to the string instruction, causing the remaining iterations to be executed. Most implementations cannot report breakpoints exactly for repeated string instructions, but instead report the breakpoint on an iteration later than the iteration where the breakpoint occurred.

13.1.3.3 I/O Breakpoints

I/O breakpoints are set by loading a breakpoint-address register (DR*n*) with the I/O-port address to be trapped, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 10b to specify that the I/O-port address is compared with the contents of DR*n* only during execution of an I/O instruction. This encoding of R/W*n* is valid only when debug extensions are enabled (CR4[DE] = 1).
- LEN*n* is set to 00b, 01b, or 11b to specify the breakpoint occurs on a byte, word, or doubleword I/O operation, respectively.

The I/O-port address specified by the I/O instruction is zero extended by the processor to 64 bits before comparing it with the DR*n* registers.

When a #DB exception occurs due to an I/O breakpoint in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the instruction is executed, which means that the original data is overwritten by the breakpoint instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

[AMD Public Use]

Because the breakpoint occurs after the instruction is executed, the address of the instruction following the I/O instruction is pushed onto the debug-handler stack, in most cases. In the case of INS and OUTS instructions that use the repeat prefix, however, the breakpoint occurs after the first iteration of the repeat loop. When this happens, the I/O-instruction address can be pushed onto the stack during a #DB exception if the repeat loop is not complete. A subsequent return from the debug handler causes the next I/O iteration to be executed. If the breakpoint condition is still set, the #DB exception reoccurs after that iteration is complete.

13.1.3.4 Task-Switch Breakpoints

Breakpoints can be set in a task TSS to raise a #DB exception after a task switch. Software enables a task breakpoint by setting the T bit in the TSS to 1. When a task switch occurs into a task with the T bit set, the processor completes loading the new task state. Before the first instruction is executed, the #DB exception occurs, and the processor sets DR6[BT] to 1, indicating that the #DB exception occurred as a result of task breakpoint.

The processor does not clear the T bit in the TSS to 0 when the #DB exception occurs. Software must explicitly clear this bit to disable the task breakpoint. Software should never set the T-bit in the debug-handler TSS if a separate task is used for #DB exception handling, otherwise the processor loops on the debug handler.

13.1.3.5 General-Detect Condition

General-detect is a special debug-exception condition that occurs when software running at any privilege level attempts to access any of the DR*n* registers while DR7[GD] is set to 1. When a #DB exception occurs due to the general-detect condition, the processor clears DR7[GD] and sets DR6[BD] to 1. Clearing DR7[GD] allows the debug handler to access the DR*n* registers without causing infinite #DB exceptions.

A debugger enables general detection to prevent other software from accessing and interfering with the debug registers while they are in use by the debugger. The exception is taken before executing the MOV DR*n* instruction so that the DR*n* contents are not altered.

13.1.3.6 Bus Lock Trap

The processor can be configured to generate a #DB exception when a bus lock occurs. Software enables bus lock trap by setting DebugCtl MSR[BLCKDB] (bit 2) to 1. When bus lock trap is enabled, the processor generates a #DB exception following the successful execution of a locked read-modify-write instruction that requires a bus lock when CPL is > 0. The processor indicates that this #DB was caused by a bus lock by clearing DR6[BLD] (bit 11). DR6[11] previously had been defined to be always 1, so DR6[11]=0 uniquely identifies the source of the #DB as a bus lock trap. All other #DB exceptions leave DR6[BLD] unmodified, so to maintain DR6[BLD]=0 as the unique identifier of a bus lock trap, system software should set DR6[BLD]=1 before returning to the interrupted task.

See Section 7.3.3 on page 197 for information on the conditions that cause a bus lock.

[AMD Public Use]

Note that implicit supervisor-level accesses do not generate a bus lock trap, even when CPL is > 0. An implicit supervisor-mode access is one that is considered a supervisor access regardless of the value of CPL.

13.1.4 Single Stepping

Single-step breakpoints are enabled by setting the rFLAGS[TF] bit to 1. TF may be set by the IRET, POPF or SYSRET instructions, with an IRET executed by a debugger being the typical use case. When IRET sets TF, it causes a #DB exception to be taken immediately *after* the *target* of the IRET is executed, returning control to the debugger and thereby single-stepping the target instruction. Setting TF with a POPF instruction also causes a one-instruction delayed #DB exception. When TF is set by SYSRET however, a #DB exception is taken *before* the target instruction executes, hence SYSRET does not provide the single-stepping behavior of IRET.

When a #DB exception occurs due to single stepping, the processor clears rFLAGS[TF] before entering the debug handler, so that the debug handler itself is not single stepped. The processor also sets DR6[BS] to 1, which indicates that the #DB exception occurred as a result of single stepping. The rFLAGS image pushed onto the debug-handler stack has the TF bit set, and single stepping resumes when a subsequent IRET pops the stack image into the rFLAGS register.

Single-step breakpoints have a higher priority than external interrupts. If an external interrupt occurs during single stepping, control is transferred to the #DB handler first, causing the rFLAGS[TF] bit to be cleared. Next, before the first instruction in the debug handler is executed, the processor transfers control to the pending-interrupt handler. This allows external interrupts to be handled outside of single-step mode.

The INT*n*, INT3, and INTO instructions clear the rFLAGS[TF] bit when they are executed. If a debugger is used to single-step software that contains these instructions, it must emulate them instead of executing them.

The single-step mechanism can also be set to single step only control transfers, rather than single step every instruction. See “Single Stepping Control Transfers” on page 412 for additional information.

13.1.5 Breakpoint Instruction (INT3)

The INT3 instruction, or the INT*n* instruction with an operand of 3, can be used to set breakpoints that transfer control to the breakpoint-exception (#BP) handler rather than the debug-exception handler. When a debugger uses the breakpoint instructions to set breakpoints, it does so by replacing the first bytes of an instruction with the breakpoint instruction. The debugger replaces the breakpoint instructions with the original-instruction bytes to clear the breakpoint.

INT3 is a single-byte instruction while INT*n* with an operand of 3 is a two-byte instruction. The instructions have slightly different effects on the breakpoint exception-handler stack. See “#BP—Breakpoint Exception (Vector 3)” on page 251 for additional information on this exception.

[AMD Public Use]

13.1.6 Control-Transfer Breakpoint Features

A control transfers is accomplished by using one of following instructions:

- JMP, CALL, RET
- J*cc*, J*r*CXZ, LOOP, LOOP*cc*
- JMPF, CALLF, RETF
- INT*n*, INT 3, INTO, ICEBP
- Exceptions, IRET
- SYSCALL, SYSRET, SYSENTER, SYSEXIT
- INTR, NMI, SMI, RSM

13.1.6.1 Recording Control Transfers

Software enables control-transfer recording by setting DebugCtl[LBR] to 1. When this bit is set, the processor updates the recording MSRs automatically when control transfers occur:

- *LastBranchFromIP and LastBranchToIP Registers*—On branch instructions, the LastBranchFromIP register is loaded with the segment offset of the branch instruction, and the LastBranchToIP register is loaded with the first instruction to be executed after the branch. On interrupts and exceptions, the LastBranchFromIP register is loaded with the segment offset of the interrupted instruction, and the LastBranchToIP register is loaded with the offset of the interrupt or exception handler.
- *LastIntFromIP and LastIntToIP Registers*—The processor loads these from the LastBranchFromIP register and the LastBranchToIP register, respectively, when most interrupts and exceptions are taken. These two registers are not updated, however, when #DB or #MC exceptions are taken, or the ICEBP instruction is executed.

The processor automatically disables control-transfer recording when a debug exception (#DB) occurs by clearing DebugCtl[LBR] to 0. The contents of the control-transfer recording MSRs are not altered by the processor when the #DB occurs. Before exiting the debug-exception handler, software can set DebugCtl[LBR] to 1 to re-enable the recording mechanism.

Debuggers can trace a control transfer backward from a bug to its source using the recording MSRs and the breakpoint-address registers. The debug handler does this by updating the breakpoint registers from the recording MSRs after a #DB exception occurs, and restarting the program. The program takes a #DB exception on the previous control transfer, and this process can be repeated. The debug handler cannot simply copy the contents of the recording MSR into the breakpoint-address register. The recording MSRs hold segment offsets, while the debug registers hold virtual (linear) addresses. The debug handler must calculate the virtual address by reading the code-segment selector (CS) from the interrupt-handler stack, then reading the segment-base address from the CS descriptor, and adding that base address to the offset in the recording MSR. The calculated virtual-address can then be used as a breakpoint address.

[AMD Public Use]

13.1.6.2 Single Stepping Control Transfers

Software can enable control-transfer single stepping by setting DebugCtl[BTF] to 1 and rFLAGS[TF] to 1. The processor automatically disables control-transfer single stepping when a debug exception (#DB) occurs by clearing DebugCtl[BTF]. rFLAGS[TF] is also cleared when a #DB exception occurs. Before exiting the debug-exception handler, software must set both DebugCtl[BTF] and rFLAGS[TF] to 1 to restart single stepping.

When enabled, this single-step mechanism causes a #DB exception to occur on every branch instruction, interrupt, or exception. Debuggers can use this capability to perform a “coarse” single step across blocks of code (bound by control transfers), and then, as the problem search is narrowed, switch into a “fine” single-step mode on every instruction (DebugCtl[BTF] = 0 and rFLAGS[TF] = 1).

Debuggers can use both the single-step mechanism and recording mechanism to support full backward and forward tracing of control transfers.

13.1.7 Debug Breakpoint Address Masking

The Breakpoint Address Extension feature extends the DR[0-3] breakpoint capabilities. Processors with this extension support address mask registers corresponding to each of the DR[0-3] breakpoint registers, in the form of DR[0-3]_ADDR_MASK MSRs. These masks may be used to increase the range of breakpoints by excluding address bits from the breakpoint match. Each bit set to one excludes the corresponding address bit from the breakpoint comparison. Mask bits 11:0 may be used to expand instruction fetch breakpoint ranges up to a 4KB page, while mask bits 31:12 have no effect on instruction breakpoints. For DR0 only, the full mask field (31:0) may be used to qualify data breakpoint matches. This extension is signified by CPUID Fn8000_0001_ECX[26]=1.

An additional extension expands the data breakpoint masking capability of DR0 to the other breakpoint registers, and extends instruction breakpoint masking to bits 31:0 for all registers. This is signified by CPUID Fn8000_0001_ECX[30]=1.

13.1.8 Last Branch Record Stack

Processors with CPUID Fn8000_0022_EAX[LbrStack] = 1 support Last Branch Record Stack (LBR Stack). The LBR Stack records the From and To IP for the LBR_ SIZE = CPUID Fn8000_0022_EBX[LbrStackSize] most recent control-transfer instructions.

LBR Stack recording is enabled by setting DebugExtnCtl [LBRSE] = 1. LBR Stack recording is independent of Last Branch Record (LBR). DebugCtl[LBR] does not need to be set to 1 for the processor to start recording to the LBR Stack.

Software can access the contents of the LBR stack using LastBranchStackFromIp and LastBranchStackToIp MSRs. The format of these MSRs is shown in Figure 13-7 on page 402 and Figure 13-8 on page 403.

Table 13-3, “Types of Branch Records” shows the types of branch records based on the values of LastBranchStackToIp[SPEC] and LastBranchStackToIp[V].

[AMD Public Use]

Table 13-3. Types of Branch Records

LastBranchStackToIp[V]	LastBranchStackToIp[SPEC]	Description of LBR Stack entry
1	0	Normal recorded branch.
1	1	Branch was recorded when a speculative performance feature was active and successful.
0	1	Branch was recorded, but the speculative performance feature was not successful.
0	0	No branch has yet been recorded in this entry since SW last cleared the V and SPEC bits.

Software can use the LastBranchStackSelect MSR (Figure 13-9 on page 404) to specify the branch types for which LBR Stack recording is suppressed.

13.2 Performance Monitoring Counters

The AMD64 architecture supports a set of hardware-based performance-monitoring counters (PMCs) that can be utilized to measure the frequency or duration of certain hardware events. MSRs allow the selection of events to be monitored and include a set of corresponding counter registers that accumulate a count of monitored events.

Software tools can use these counters to identify performance bottlenecks, such as sections of code that have high cache-miss rates or frequently mispredicted branches. This information can then be used as a guide for improving overall performance or eliminating performance problems through software optimizations or hardware-design improvements.

Software performance analysis tools often require a means to time-stamp an event or measure elapsed time between two events. The time-stamp counter provides this capability. See Section 13.2.4 “Time-Stamp Counter” on page 425.

The registers used in support of performance monitoring are model-specific registers (MSRs). See “Model-Specific Registers (MSRs)” on page 60 for a general discussion of MSRs and “Performance-Monitoring MSRs” on page 772 for a listing of the performance-monitoring MSR numbers and their reset values.

13.2.1 Performance Counter MSRs

The legacy architecture defines four performance counters (PerfCtr*n*) and corresponding event-select registers (PerfEvtSel*n*). Extensions add Northbridge and L2 cache performance monitoring counters. Each *PerfCtr register counts events selected by the corresponding *PerfEvtSel register.

[AMD Public Use]

An architectural extension augments the number of performance and event-select registers by adding two more processor counter / event-select pairs. Further extensions add four counter / event-select pairs dedicated to counting Northbridge (NB) events and four counter / event-select pairs dedicated to counting L2 cache (L2I) events.

Core logic includes instruction execution pipelines, execution units, and caches closest to the execution hardware. The NB includes logic that routes data traffic between caches, external I/O devices, and a system memory controller which reads and writes system memory (usually implemented as external DRAM). The L2 cache is a cache that is further away from the processor core than the L1 cache or caches. This cache is normally larger than the L1 cache(s) and requires more processor cycles to access. An L2 cache may be shared between physical processor cores.

All implementations support the base set of four performance counter / event-select pairs. Support for the extended performance monitoring registers and the performance-related events selectable via the *PerfEvtSel registers vary by implementation and are described in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* for that processor.

Core performance counters are used to count processor core events, such as data-cache misses, or the duration of events, such as the number of clocks it takes to return data from memory after a cache miss. During event counting, hardware increments a counter each time it detects an occurrence of a specified event. During duration measurement, hardware counts the number of processor clock cycles required to complete a specific hardware function.

NB performance counters are used to count events that occur within the Northbridge. The L2I performance counters are used to count events associated with accessing the L2 cache.

Performance counters and event-select registers are implemented as machine-specific registers (MSRs). The base set of four PerfCtr and PerfEvtSel registers are accessed via a legacy set of MSRs and the extended set of six core PerfCtr / PerfEvtSel registers are accessed via a different set. Extended core PerfCtr / PerfEvtSel registers 0–3 alias the legacy set.

Support for the extended set of core PerfCtr registers and associated PerfEvtSel registers, as well as the sets of Northbridge and L2 cache counter / event-select pairs are indicated by CPUID feature bits. See "Detecting Hardware Support for Performance Counters" on page 423. The MSR address assignments for the legacy and extended performance / event-select pairs are listed in Appendix A, Section A.6, "Performance-Monitoring MSRs" on page 772.

The length, in bits, of the performance counters is implementation-dependent, but the maximum length supported is 64 bits. Figure 13-10 shows the format of the performance counter registers.

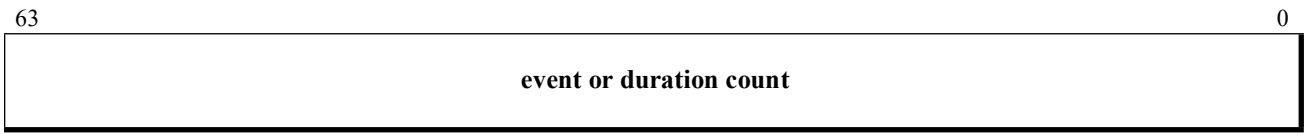


Figure 13-10. Performance Counter Format

[AMD Public Use]

For a given processor, all implemented performance counter registers can be read and written by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. The architecture also provides an instruction, RDPMC, which may be employed by user-mode software to read the architected core, Northbridge, and L2 performance counters.

The RDPMC instruction loads the contents of the architected performance counter register specified by the index value contained in the ECX register, into the EDX register and the EAX register. The high 32 bits are returned in EDX, and the low 32 bits are returned in EAX. RDPMC can be executed only at CPL = 0, unless system software enables use of the instruction at all privilege levels. RDPMC can be enabled for use at all privilege levels by setting CR4[PCE] (the *performance-monitor counter-enable* bit) to 1. When CR4[PCE] = 0 and CPL > 0, attempts to execute RDPMC result in a general-protection exception (#GP). For more information on the RDPMC instruction, see the instruction reference page in Volume 3 of this manual.

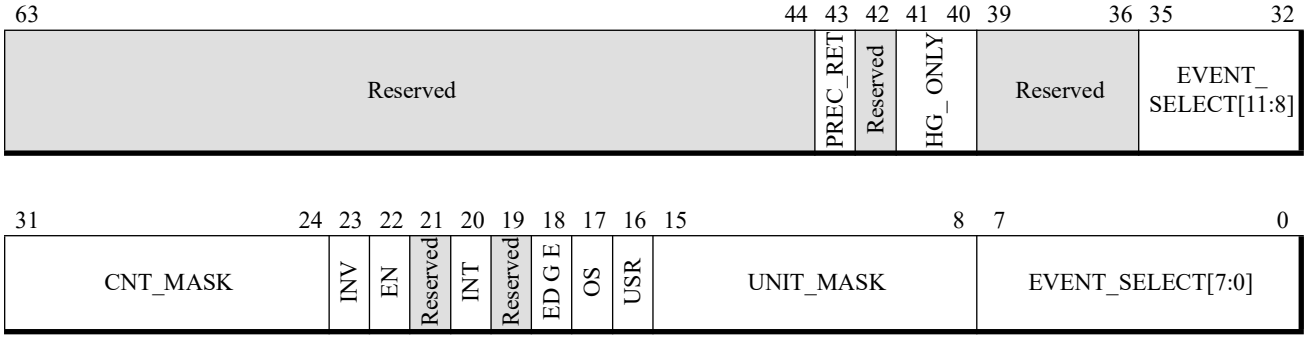
Writing the performance counters can be useful if software wants to count a specific number of events, and then trigger an interrupt when that count is reached. An interrupt can be triggered when a performance counter overflows (see “Counter Overflow” on page 424 for additional information). Software should use the WRMSR instruction to load the count as a two’s-complement negative number into the performance counter. This causes the counter to overflow after counting the appropriate number of times.

The performance counters are not guaranteed to produce identical measurements each time they are used to measure a particular instruction sequence, and they should not be used to take measurements of very small instruction sequences. The RDPMC instruction is not serializing, and it can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause events to be counted before the counter value is loaded into EDX:EAX. The following sections describe the core performance event-select and the Northbridge performance event-select registers.

Core Performance Event-Select Registers

The core performance event-select registers (PerfEvtSel*n*) are 64-bit registers used to specify the events counted by the core performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-11 below shows the format of the core PerfEvtSel register.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:44	Reserved		RAZ
43	PREC_RET	Enable precise counting of retire events.	R/W
42	Reserved		RAZ
41:40	HG_ONLY	Host/Guest Only	R/W
39:36	Reserved		RAZ
35:32	EVENT_SELECT[11:8]	Event select bits 11:8	R/W
31:24	CNT_MASK	Counter Mask	R/W
23	INV	Invert Comparison	R/W
22	EN	Counter Enable	R/W
21	Reserved		RAZ
20	INT	Interrupt Enable	R/W
19	Reserved		RAZ
18	EDGE	Edge Detect	R/W
17	OS	Operating-System Mode	R/W
16	USR	User Mode	R/W
15:8	UNIT_MASK	Unit Mask	R/W
7:0	EVENT_SELECT[7:0]	Event select bits 7:0	R/W

Figure 13-11. Core Performance Event-Select Register (PerfEvtSel*n*)

The fields shown in Figure 13-11 above are further described below:

- *PREC_RET (Enable precise counting of retire events)*—Bit 43, read/write. If CPUID Fn 8000_0021h_EAX[PreciseRetirePmc2Manual] is set, PREC_RET should be set when counting retired events. If CPUID Fn 8000_0021h_EAX[PreciseRetirePmc2Manual] is clear, this bit is Reserved. PREC_RET is only available in PerfEvtSel2 and Reserved for other PerfEvtSel*n* registers.
- *HG_ONLY (Host/Guest Only)*—Bits 41:40, read/write. This field qualifies events to be counted based on virtualization operating mode (guest or host). The following table defines how HG_ONLY qualifies the counting of events:

[AMD Public Use]

Table 13-4. Host/Guest Only Bits

Host Mode (Bit 41)	Guest Mode (Bit 40)	Events Counted
0	0	All events, irrespective of guest or host mode
0	1	Guest events, if EFER[SVME] = 1
1	0	Host events, if EFER[SVME] = 1
1	1	Guest and host events, if EFER[SVME] = 1

- *EVENT_SELECT[11:8] (Event Select)*—Bits 35:32, read/write. This field extends the EVENT_SELECT field from 8 bits to 12 bits. See EVENT_SELECT[7:0] below.
- *CNT_MASK (Counter Mask)*—Bits 31:24, read/write. Used with INV bit to control the counting of multiple events that occur within one clock cycle. The table below describes this:

Table 13-5. Count Control Using CNT_MASK and INV

CNT_MASK	INV	Increment Value
00h	–	Corresponding PerfCtr[*n*] register is incremented by the number of events occurring in a clock cycle. If the number of events is equal to or greater than 32, the count register is incremented by 32.
FFh:01h[1]	0	Corresponding PerfCtr[*n*] register is incremented by 1, if the number of events occurring in a clock cycle is greater than or equal to the CNT_MASK value.
	1	Corresponding PerfCtr[*n*]register is incremented by 1, if the number of events occurring in a clock cycle is less than the CNT_MASK value.

Note 1: Maximum CNT_MASK value (in the range FFh:01h is implementation dependent. Consult applicable BIOS and Kernel Developer's Guide (BKDG) or Processor Programming Reference Manual (PPR) .

- INV (Invert Comparison), read/write. Used with CNT_MASK field to control the counting of multiple events within one clock cycle. See table above.
- EN (Counter Enable), read/write. Software sets this bit to 1 to enable the PerfEvtSel*n* register, and counting in the corresponding PerfCtr*n* register. Clearing this bit to 0 disables the register pair.
- INT (Interrupt Enable), read/write. Software sets this bit to 1 to enable an interrupt to occur when the performance counter overflows (see "Counter Overflow" on page 424 for additional information). Clearing this bit to 0 disables the triggering of the interrupt.
- EDGE (Edge Detect), read/write. Software sets this bit to 1 to count the number of edge transitions from the negated to asserted state. This feature is useful when coupled with event-duration monitoring, as it can be used to calculate the average time spent in an event. Clearing this bit to 0 disables edge detection.
- OS (Operating-System Mode) and USR (User Mode), read/write. Software uses these bits to control the privilege level at which event counting is performed according to Table 13-6.

[AMD Public Use]

Table 13-6. Operating-System Mode and User Mode Bits

OS (Bit 17)	USR (Bit 16)	Event Counting
0	0	No counting.
0	1	Only at CPL > 0.
1	0	Only at CPL = 0.
1	1	At all privilege levels.

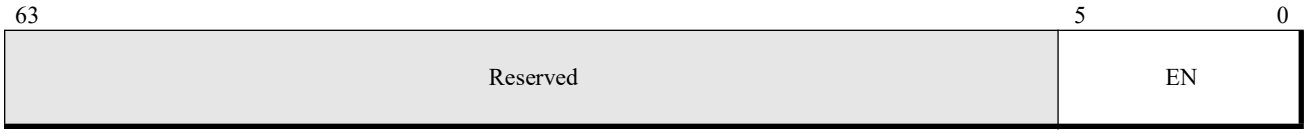
- *UNIT_MASK (Unit Mask)*—Bits 15:8, read/write. This field further specifies or qualifies the event specified by the EVENT_SELECT field. Depending on implementation, it may be used to specify a sub-event within the class specified by the EVENT_SELECT field or it may act as bit mask and be used to specify a number of events within the class to be monitored simultaneously.
- *EVENT_SELECT[7:0] (Event Select [7:0])*—Bits 7:0, read/write. This field concatenated with EVENT_SELECT[11:8] specifies the event or event duration to be counted by the corresponding PerfCtr[*n*] register. The events that can be monitored are implementation dependent. In some implementations, support for a specific EVENT_SELECT value may restricted to a subset of the available performance counters. For more information, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

The core performance event-select registers can be read and written only by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL > 0 causes a general-protection exception to occur.

Core Performance Global Control Register

Core Performance Counter Global Control Register (PerfCntrGlobalCtl) is a 64-bit register used to provide additional enable bits for each of the available core performance counters (PerfCnt*n*). The number of available core performance counters is reported by CPUID Fn8000_0022_EBX[NumCorePmc]. PerfCntrGlobalCtl provides a way to simultaneously enable or disable multiple core performance counters with a single WRMSR instruction. To enable PerfCnt*n*, both bit *n* of PerfCntrGlobalCtl and the EN field of the counter's PerfEvtSel register must be set to 1. Clearing either bit causes the corresponding count register to stop counting.

Figure 13-12 shows the format of the core PerfCntrGlobalCtl register.



[AMD Public Use]

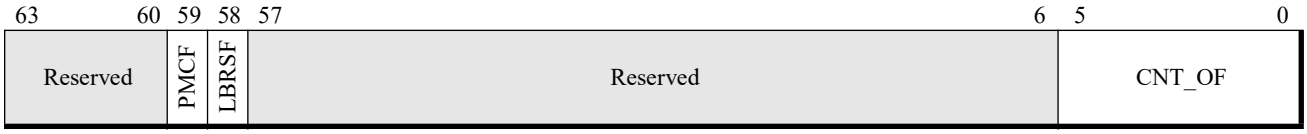
Bits	Mnemonic	Description	Access type
63:6	Reserved		RAZ
5:0	EN	Global enable for PerfCnt[5:0]	R/W

Figure 13-12. Performance Counter Global Control (PerfCntrGlobalCtl)

Core Performance Counter Status Registers

A set of three registers is provided to monitor and manage core performance counter status. The Core Performance Counter Status Register (PerfCntrGlobalStatus) contains overflow bits for the Core Performance Counters and freeze status bits for the LBR stack and performance counters. Software can clear individual status bits by writing 1's to corresponding bit positions in the PerfCntrGlobalStatusClr register, and set individual status bits by writing 1's to corresponding bit positions in the PerfCntrGlobalStatusSet register. Hardware normally sets these bits; setting or clearing them with software may cause certain hardware actions as described below.

Performance Counter Global Status Register



Bits	Mnemonic	Description	Access type
63:60	Reserved		RO
59	PMCF	Performance Counter Freeze	RO
58	LBRSF	Last Branch Record Stack Freeze	RO
57:6	Reserved		RO
5:0	CNT_OF	Counter overflow for PerfCnt[5:0]	RO

Figure 13-13. Performance Counter Global Status Register (PerfCntrGlobalStatus)

The fields shown in Figure 13-13 are further described below:

- *Performance Counter Freeze (PMCF)*—Bit 59, read-only. When set, this indicates that the Core Performance Counter registers have been frozen due to an overflow interrupt triggered for at least one of the core performance counters. This feature is enabled via DebugCtl[FPCI] as described in "Debug-Control MSR (DebugCtl)" on page 399. Software must use PerfCntrGlobalStatusClr[C_PMCF] to clear this bit to allow the Core Performance Counters to resume counting.
- *Last Branch Record Stack Freeze (LBRSF)*—Bit 58, read-only. When set, this indicates that the Last Branch Record Stack has been frozen due to an overflow interrupt triggered for at least one of the core performance counters. This feature is enabled via DebugCtl[FLBRI] as described in "Debug-Control MSR (DebugCtl)" on page 399. Software must use

[AMD Public Use]

PerfCntrGlobalStatusClr[C_LBRSF] to clear this bit to allow the Last Branch Record Stack to resume recording branches.

- *Core Performance Counter Overflow (CNT_OF)*—Bits 5:0, read-only. Each available Core Performance Counter corresponds to one bit in this field. When set, it indicates that the corresponding performance counter has overflowed. The CNT_OF bit will be set for a counter even when the counter is not configured to trigger an interrupt on overflow. Software must use PerfCntrGlobalStatusSet[C_CNT_OF] to clear these indications.

Software may use PerfCntrGlobalStatusSet and PerfCntrGlobalStatusClr registers to modify the contents of the PerfCntrGlobalStatus register for context switching purposes.

Performance Counter Global Status Set Register



Bits	Mnemonic	Description	Access type
63:60	Reserved		WO
59	S_PMCF	Set Performance Counter Freeze	WO
58	S_LBRSF	Set Last Branch Record Freeze	WO
57:6	Reserved		WO
5:0	S_CNT_OF	Set selected bits of PerfCntrGlobalStatus[CNT_OF]	WO

Figure 13-14. Performance Counter Global Status Set Register (PerfCntrGlobalStatusSet)

The fields shown in Figure 13-14 are further described below:

- *Set Core Performance Counter Freeze (S_PMCF)*—Bit 59, write-only. When written as 1, PerfCntrGlobalStatus[PMCF] is set to 1 and the Core Performance Monitor Counters are frozen.
- *Set Last Branch Record Stack Freeze (S_LBRSF)*—Bit 58, write-only. When written as 1, PerfCntrGlobalStatus[LBRSF] is set to 1 and the Last Branch Record Stack is frozen.
- *Set Core Performance Counter Overflow (S_CNT_OF)*—Bits 5:0, write-only. For each bit written as 1, the corresponding bit in PerfCntrGlobalStatus[CNT_OF] is set to 1. Setting a bit in PerfCntrGlobalStatus[CNT_OF] through PerfCntrGlobalStatusSet[CNT_OF] does not trigger an interrupt, freeze core performance counters, freeze the Last Branch Record stack or trigger any other action that the hardware may take when a core performance counter overflows.

[AMD Public Use]

Performance Counter Global Status Clear Register



Bits	Mnemonic	Description	Access type
63:60	Reserved		WO
59	C_PMCF	Clear Performance Counter Freeze	WO
58	C_LBRSF	Clear Last Branch Record Freeze	WO
57:6	Reserved		WO
5:0	C_CNT_OF	Clear selected bits of PerfCntrGlobalStatus[CNT_OF]	WO

Figure 13-15. Performance Counter Global Status Clear Register (PerfCntrGlobalStatusClr)

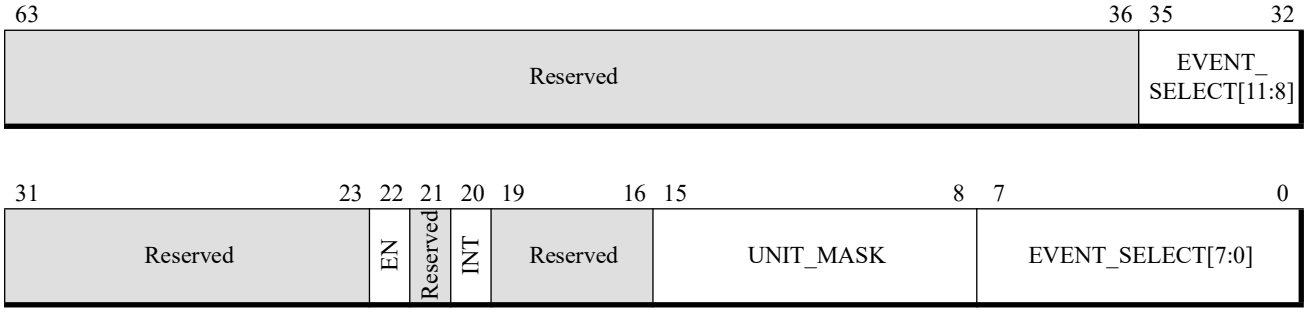
The fields shown in Figure 13-15 are further described below:

- *Clear Performance Counter Freeze (C_PMCF)*—Bit 59, write-only. When written as 1, PerfCntrGlobalStatus[PMCF] is cleared to 0 and core performance counters, that are enabled to count, continue counting.
- *Clear Last Branch Record Stack Freeze (C_LBRSF)*—Bit 58, write-only. When written as 1, PerfCntrGlobalStatus[LBRSF] is cleared to 0 and the Last Branch Record Stack continues to record branches if enabled.
- *Clear Core Performance Counter Overflow (C_CNT_OF)*—Bits 5:0, write-only. For each bit written as 1, the corresponding bit in PerfCntrGlobalStatus[CNT_OF] is cleared to 0. Software should clear the Performance Counter Overflow bits when:
  - Handling a core performance counter overflow interrupt
  - Disabling a core performance counter
  - Resetting a core performance counter

Northbridge Performance Event-Select Registers

The Northbridge (NB) performance event-select registers (NB_PerfEvtSel*n*) are 64-bit registers used to specify the events counted by the Northbridge performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-16 below shows the format of the NB_PerfEvtSel*n* register.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:36	Reserved		RAZ
35:32	EVENT_SELECT[11:8]	Event select bits [11:8]	R/W
31:23	Reserved		RAZ
22	EN	Counter Enable	R/W
21	Reserved		RAZ
20	INT	Interrupt Enable	R/W
19:16	Reserved		RAZ
15:8	UNIT_MASK	Unit Mask	R/W
7:0	EVENT_SELECT[7:0]	Event select bits [7:0]	R/W

Figure 13-16. Northbridge Performance Event-Select Register (NB_PerfEvtSel*n*)

The Northbridge performance event-select registers can be read and written only by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL > 0 causes a general-protection exception to occur.

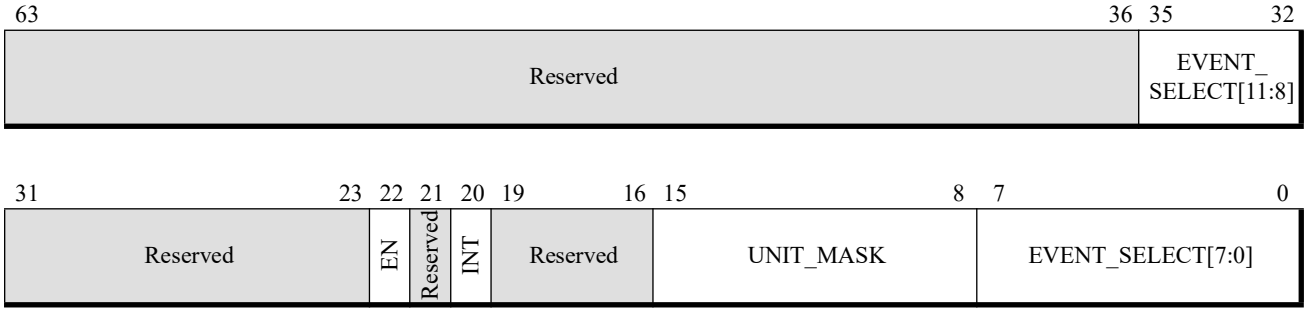
For more information on the defined fields within the NB_PerfEvtSel*n* registers, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

Support for NB_PerfEvtSel0-3 and NB_PerfCtr0-3 is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtNB] (bit 24) = 1. Processors that set CPUID Fn8000_0022_EAX[PerfMonV2] (bit 0) = 1 report the total number of available Northbridge performance counters in CPUID Fn8000_0022_EBX[NumPerfCtrNB] (bits 15:10).

L2 Cache (L2I) Performance Event-Select Registers

The L2 cache performance event-select registers (L2I_PerfEvtSel*n*) are 64-bit registers used to specify the events counted by the L2 cache performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-17 below shows the format of the L2I_PerfEvtSel*n* register.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:36	Reserved		RAZ
35:32	EVENT_SELECT[11:8]	Event select bits [11:8]	R/W
31:23	Reserved		RAZ
22	EN	Counter Enable	R/W
21	Reserved		RAZ
20	INT	Interrupt Enable	R/W
19:16	Reserved		RAZ
15:8	UNIT_MASK	Unit Mask	R/W
7:0	EVENT_SELECT[7:0]	Event select bits [7:0]	R/W

Figure 13-17. L2 Cache Performance Event-Select Register (L2I_PerfEvtSel*n*)

The L2 cache performance event-select registers can be read and written only by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL > 0 causes a general-protection exception to occur.

For more information on the defined fields within the L2I_PerfEvtSel*n* registers, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

Instructions Retired Performance counter

This is a dedicated counter that is always counting instructions retired. It exists at MSR address C000_00E9. It is enabled by setting a 1 to HWCR[30] and its support is indicated by CPUID Fn8000_0008_EBX[1].

13.2.2 Detecting Hardware Support for Performance Counters

Support for extended core, Northbridge, and L2 cache performance counters is implementation-dependent. Support on a given processor implementation can be verified using the CPUID instruction.

CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1 indicates support for the six architecturally defined extended core performance counters and their associated event-select registers. CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1 indicates support for the four architecturally defined Northbridge performance counter / event-select pairs and

[AMD Public Use]

CPUID Fn8000_0001_ECX[PerfCtrExtL2I] = 1 indicates support for the four architecturally defined L2 cache performance counter / event-select pairs.

See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

Processors that return a non-zero value in CPUID Fn8000_0022_EBX[NumPerfCtrCore] report the number of available core performance counters in CPUID Fn8000_0022_EBX[NumPerfCtrCore].

Processors that return a non-zero value in CPUID Fn8000_0022_EBX[NumPerfCtrNB] report the number of available Northbridge performance counters in CPUID Fn8000_0022_EBX[NumPerfCtrNB].

A given processor may implement other performance measurement MSRs with similar capabilities even if one or more of the optional architected facilities are not supported.

CPUID Fn8000_0022_EAX[PerfMonV2] = 1 indicates support for PerfCntrGlobalCtl MSR, PerfCntrGlobalStatus MSR, PerfCntrGlobalStatusClr MSR and PerfCntrGlobalStatusSet MSR.

CPUID Fn8000_0022_EAX[LbrAndPmcFreeze] = 1 indicates support for LBR Stack and Core Performance Counter Freeze on PMC overflow.

13.2.3 Using Performance Counters

13.2.3.1 Starting and Stopping

Performance measurement using the PerfCtr*n*, NB_PerfCtr*n*, and L2I_PerfCtr*n* registers is initiated by setting the corresponding *PerfEvtSel*n*[EN] bit to 1. Counting is stopped by clearing the *PerfEvtSel*n*[EN] bit. Software must initialize the remaining *PerfEvtSel*n* fields with the appropriate setup information before or at the same time EN is set. Counting begins when the WRMSR instruction that sets *PerfEvtSel*n*[EN] to 1 completes execution. Counting stops when the WRMSR instruction that clears the EN bit completes execution.

13.2.3.2 Counter Overflow

Some processor implementations support an interrupt-on-overflow capability that allows an interrupt to occur when one of the *PerfCtr*n* registers overflows. The source and type of interrupt is implementation dependent. Some implementations cause a debug interrupt to occur, while others make use of the local APIC to specify the interrupt vector and trigger the interrupt when an overflow occurs. Software enables or disables the triggering of an interrupt on counter overflow by setting or clearing the *PerfEvtSel*n*[INT] bit.

If system software makes use of the interrupt-on-overflow capability, an interrupt handler must be provided that can record information relevant to the counter overflow. Before returning from the interrupt handler, the performance counter can be re-initialized to its previous state so that another interrupt occurs when the appropriate number of events are counted.

[AMD Public Use]

13.2.4 Time-Stamp Counter

The time-stamp counter (TSC) is used to count processor-clock cycles. The TSC is cleared to 0 after a processor reset. After a reset, the TSC is incremented at a rate corresponding to the baseline frequency of the processor (which may differ from actual processor frequency in low power modes of operation). Each time the TSC is read, it returns a monotonically-larger value than the previous value read from the TSC. When the TSC contains all ones, it wraps to zero. The TSC in a 1-GHz processor counts for almost 600 years before it wraps. Figure 13-18 shows the format of the 64-bit time-stamp counter (TSC).

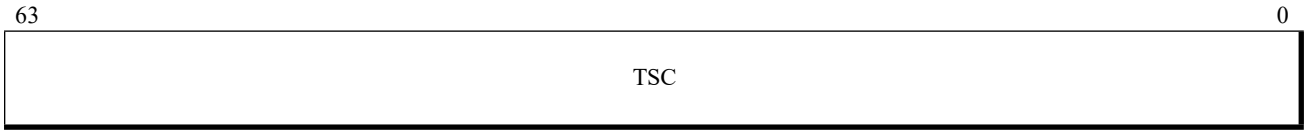


Figure 13-18. Time-Stamp Counter (TSC)

The TSC is a model-specific register that can also be read using one of the special *read time-stamp counter* instructions, RDTSC (Read Time-Stamp Counter) or RDTSCP (Read Time-Stamp Counter and Processor ID). The RDTSC and RDTSCP instructions load the contents of the TSC into the EDX register and the EAX register. The high 32 bits are loaded into EDX, and the low 32 bits are loaded into EAX. The RDTSC and RDTSCP instructions can be executed at any privilege level and from any processor mode. However, system software can disable the RDTSC or RDTSCP instructions for programs that run at CPL > 0 by setting CR4[TSD] (the *time-stamp disable* bit) to 1. When CR4[TSD] = 1 and CPL > 0, attempts to execute RDSTC or RDSTCP result in a general-protection exception (#GP).

The TSC register can be read and written using the RDMSR and WRMSR instructions, respectively. The programmer should use the CPUID instruction to determine whether these features are supported. If EDX bit 4 (as returned by CPUID function 1) is set, then the processor supports TSC, the RDTSC instruction and CR4[TSD]. If EDX bit 27 returned by CPUID function 8000_0001h is set, then the processor supports the RDTSCP instruction.

The TSC register can be used by performance-analysis applications, along with the performance-monitoring registers, to help determine the relative frequency of an event or its duration. Software can also use the TSC to time software routines to help identify candidates for optimization. In general, the TSC should not be used to take very short time measurements, because the resulting measurement is not guaranteed to be identical each time it is made. The RDTSC instruction (unlike the RDTSCP instruction) is not serializing, and can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause additional cycles to be counted before the TSC value is loaded into EDX:EAX.

When using the TSC to measure elapsed time, programmers must be aware that for some implementations, the rate at which the TSC is incremented varies based on the processor power management state (Pstate). For other implementations, the TSC increment rate is fixed and is not

[AMD Public Use]

subject to power-management related changes in processor frequency. CPUID Fn 8000_0007h_EDX[TscInvariant] = 1 indicates that the TSC increment rate is a constant.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 73.

13.3 Instruction-Based Sampling

Instruction-Based Sampling (IBS) is a hardware facility that can be used to gather specific metrics related to processor instruction fetch and instruction execution activity. Data capture is performed by hardware at a sampling interval specified by values programmed in IBS sampling control registers. The IBS facility can be utilized by software to perform code profiling based on statistical sampling.

There are two independent data gathering components of IBS: instruction fetch sampling and instruction execution sampling. Instruction fetch sampling provides information about instruction address translation look-aside buffer (ITLB) and instruction cache behavior for a randomly selected fetch block, under the control of the IBS Fetch Control Register. Instruction execution sampling provides information about instruction execution behavior by tracking the execution of a single operation (op) that is randomly selected, under the control of the IBS Execution Control Register.

When the programmed interval for fetch sampling has expired, the fetch sampling component of IBS selects and tags the next fetch block. IBS hardware records specific performance information about the tagged fetch. In a similar manner, when the programmed interval for op sampling has expired, the op sampling component of IBS selects and tags the next op being dispatched for execution.

When data collection for the tagged fetch or op is complete, the hardware signals an interrupt. An interrupt handler can then read the performance information that was captured for the fetch or op in IBS MSRs, save it, and re-enable the hardware to take the next sample.

More information about the IBS facility and how software can use it to perform code profiling can be found in the *Software Optimization Guide* for your specific product. The *Software Optimization Guide for AMD Family 15h Processors* is order #47414.

Support for the IBS feature is indicated by the CPUID Fn8000_0001h_ECX[IBS]. For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 73.

13.3.1 IBS Fetch Sampling

When a processor fetches an instruction, it is actually reading a contiguous range of instruction bytes that contains the instruction from memory or from cache. This range of bytes loaded by the processor in one operation is called a *fetch block*. The size and address-alignment characteristics of the fetch block are implementation-dependent. In the following discussion, the term *instruction fetch* or simply *fetch* refers to this operation of reading a fetch block.

Instruction fetch sampling records the following performance information for each tagged fetch:

[AMD Public Use]

- If the fetch completed or was aborted
- The number of core clock cycles spent on the fetch
- If the fetch hit or missed the instruction cache
- If the instruction fetch hit or missed the instruction TLBs
- The fetch address translation page size
- The linear and physical address associated with the fetch

IBS selects and tags a fetch at a programmable rate. When enabled by the IBS Fetch Control Register (IbsFetchEn = 1 and IbsFetchVal = 0) and not disabled by IBS Fetch Control register 2 (IbsFetchCtl2[IbsDis] = 0), an internal 20-bit fetch interval counter increments for every attempted fetch operation. When the value in bits 19:4 of the fetch counter equal the value in the IbsFetchMaxCnt field of the IBS Fetch Control Register, the next fetch block is tagged for data collection. IbsFetchCtl2[IbsFetchDis] is suported on processors which set CPUID Fn8000_0001B_EAX[IbsDis] (bit 13) = 1. It allows software to start and stop IBS Fetch Sampling with a write to a register not updated by the processor thus eliminating a read-modify-write hazard which exists for modifying bits in IBS Fetch Control while IBS Fetch Sampling is active.

When the tagged fetch completes or is aborted, the status of the fetch is written to the IBS Fetch Control Register and the associated linear address and physical address are written in the IBS Fetch Linear Address Register and IBS Fetch Physical Address Register, respectively. The IbsFetchVal bit is set in the IBS Fetch Control Register and an interrupt is generated as specified by the local APIC.

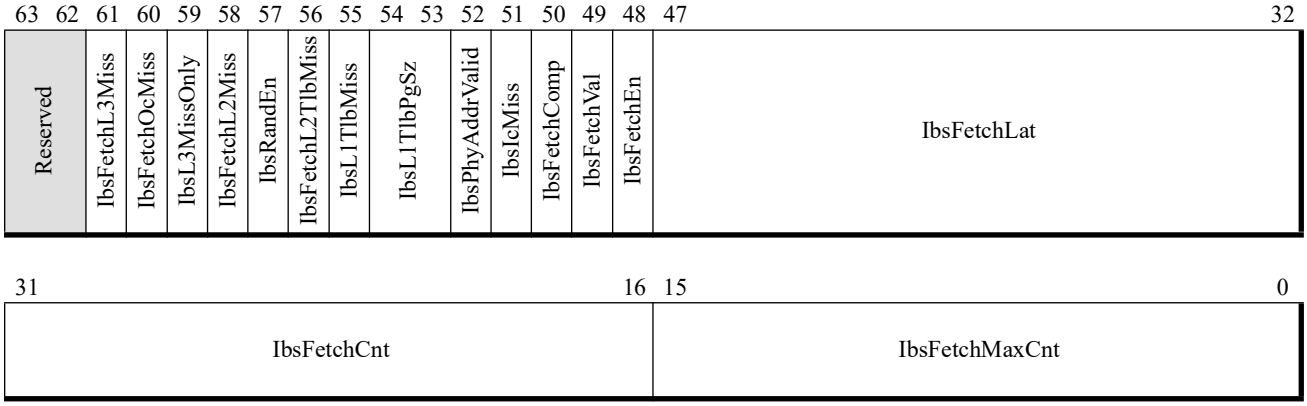
The interrupt service routine saves the performance information stored in the IBS fetch registers. Software can then initiate another sample by resetting the IbsFetchVal bit in the IBS Fetch Control Register. Hardware initializes bits 19:4 of the internal fetch interval counter with the value in the IbsFetchCnt field. If the IbsFetchCtl[IbsRandEn] bit is set, bits 3:0 of the fetch interval counter are re-initialized by hardware with a pseudo-random value; otherwise bits 3:0 are cleared.

13.3.2 IBS Fetch Sampling Registers

The IBS fetch sampling registers consist of the status and control register (IBS Fetch Control Register) and the associated fetch address registers (IBS Fetch Linear Address Register and IBS Fetch Physical Address Register). The IBS fetch sampling registers are accessed using the RDMSR and WRMSR instructions.

[AMD Public Use]

IBS Fetch Control Register (IbsFetchCtl) (MSR C001_1030h)



Bits	Mnemonic	Description	Access type
63:62	Reserved		RAZ
61	IbsFetchL3Miss	IBS Fetch L3 Cache Miss	R/W
60	IbsFetchOcMiss	IBS Fetch Op Cache Miss	R/W
59	IbsL3MissOnly	IBS Fetch L3 Miss Filtering	R/W
58	IbsFetchL2Miss	IBS Fetch L2 Cache Miss	R/W
57	IbsRandEn	IBS Randomize Tagging Enable	R/W
56	IbsFetchL2TlbMiss	IBS Fetch L2 TLB Miss	R/W
55	IbsL1TlbMiss	IBS Fetch L1 TLB Miss	R/W
54:53	IbsL1TlbPgSz	IBS Fetch L1 TLB Page Size	R/W
52	IbsPhyAddrValid	IBS Fetch Physical Address Valid	R/W
51	IbsIcMiss	IBS Instruction Cache Miss	R/W
50	IbsFetchComp	IBS Fetch Complete	R/W
49	IbsFetchVal	IBS Fetch Valid	R/W
48	IbsFetchEn	IBS Fetch Enable	R/W
47:32	IbsFetchLat	IBS Fetch Latency	R/W
31:16	IbsFetchCnt	IBS Fetch Count	R/W
15:0	IbsFetchMaxCnt	IBS Fetch Maximum Count	R/W

Figure 13-19. IBS Fetch Control Register (IbsFetchCtl)

The fields shown in Figure 13-19 are further described below:

- *IbsFetchL3Miss (IBS Fetch L3 Cache Miss)*—Bit 61, read/write. This bit is set when the tagged fetch missed in the L3 cache. This bit is supported when CPUID Fn8000_001B_EAX[IbsL3MissFiltering](bit 11) = 1.
- *IbsFetchOcMiss (IBS Fetch Op Cache Miss)*—Bit 60, read/write. This bit is set if the tagged fetch missed in the Op Cache. This bit is supported when CPUID Fn8000_001B_EAX[IbsL3MissFiltering](bit 11) = 1.

[AMD Public Use]

- *IbsL3MissOnly (IBS Fetch L3 Miss Filtering)*—Bit 59, read/write. This bit controls L3 Miss Filtering. When set, Fetch IBS only sends interrupts for samples with an L3 miss. See Section 13.3.5, “IBS Filtering,” on page 444 for a description of Fetch IBS Filtering.
- *IbsFetchL2Miss (IBS Fetch L2 Cache Miss)*—Bit 58, read/write. This bit is set if the tagged fetch missed in the L2 cache.
- *IbsRandEn (IBS Randomize Tagging Enable)*—Bit 57, read/write. Software sets this bit to 1 to add variability to the interval at which fetch operations are selected for tagging. When set, bits 3:0 of the fetch interval counter are set to a pseudo-random value when the IbsFetchCtl register is written. Clearing this bit causes bits 3:0 of the fetch interval counter to be reset to zero.
- *IbsFetchL2TlbMiss (IBS Fetch L2 TLB Miss)*—Bit 56, read/write. This bit is set if the tagged fetch missed in the Level 2 TLB.
- *IbsL1TlbMiss (IBS Fetch L1 TLB Miss)*—Bit 55, read/write. This bit is set if the tagged fetch missed in the L1 TLB.
- *IbsL1TlbPgSz[1:0] (IBS Fetch L1 TLB Page Size)*—Bits 54:53, read/write. This field indicates the page size of the translation in the L1 TLB for the tagged fetch. This field is valid only if IbsPhyAddrValid = 1. The table below defines the encoding of this two-bit field:

Value	Page Size
00b	4 Kbyte
01b	2 Mbyte
10b	1 Gbyte
11b	4 Kbyte coalesced

  Some implementations might not support all page sizes. Note: The page size in the L1 TLB might not match the page size in the page table.
- *IbsPhyAddrValid (IBS Fetch Physical Address Valid)*—Bit 52, read/write. This bit is set if the physical address of the tagged fetch is valid. When this bit is set, the IbsL1TlbPgSz field and the contents of the IBS Fetch Physical Address Register (see definition of this register below) are both valid.
- *IbsIcMiss (IBS Instruction Cache Miss)*—Bit 51, read/write. This bit is set if the tagged fetch missed in the instruction cache.
- *IbsFetchComp (IBS Fetch Complete)*—Bit 50, read/write. This bit is set if the tagged fetch completes and data is available for use by the instruction decoder.
- *IbsFetchVal (IBS Fetch Valid)*—Bit 49, read/write. This bit is set if the tagged fetch either completes or is aborted. When the bit is set, captured data for the tagged fetch is available and the fetch interval counter stops. An interrupt is generated as specified by the local APIC. The interrupt handler should read and save the captured performance data before clearing the bit.
- *IbsFetchEn (IBS Fetch Enable)*—Bit 48, read/write. Software sets this bit to enable fetch sampling. Clearing this bit to 0 disables fetch sampling.
- *IbsFetchLat[15:0] (IBS Fetch Latency)*—Bits 47:32, read/write. This 16-bit field indicates the number of core clock cycles from the initiation of the fetch to the delivery of the instruction bytes

[AMD Public Use]

to the core. If the fetch is aborted before it completes, this field returns the number of clock cycles from the initiation of the fetch to its abortion.

- *IbsFetchCnt[15:0] (IBS Fetch Count)*—Bits 31:16, read/write. This 16-bit field returns the current value of bits 19:4 of the fetch interval counter on a read. Bits 19:4 of the fetch interval counter are set to this value on a write.
- *IbsFetchMaxCnt[15:0] (IBS Fetch Maximum Count)*—Bits 15:0, read/write. This 16-bit field specifies the maximum count value of bits 19:4 of the fetch interval counter. When the value in bits 19:4 of the fetch counter equals the value in this field, the next fetch block is tagged for profiling.

IBS Fetch Linear Address Register (IbsFetchLinAd) (MSR C001_1031h)

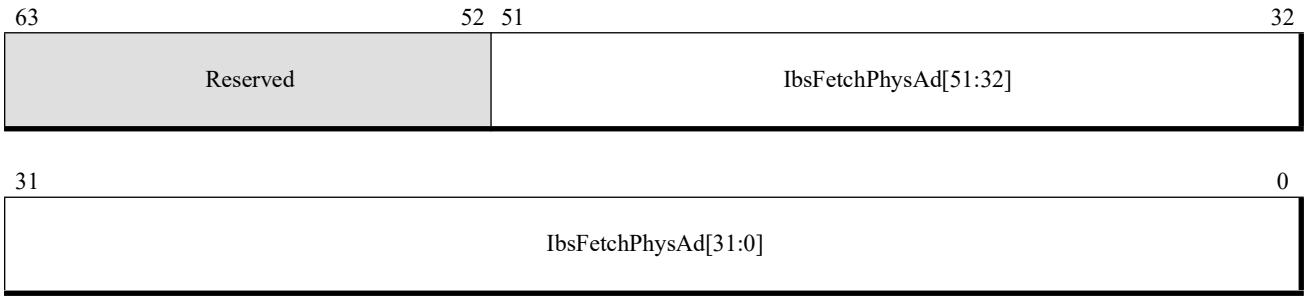


Bits	Mnemonic	Description	Access type
63:0	IbsFetchLinAd	IBS Fetch Linear Address	RO

Figure 13-20. IBS Fetch Linear Address Register (IbsFetchLinAd)

This is a read-only register. Reading the IbsFetchLinAd MSR returns the 64-bit linear address of the tagged fetch. This address may correspond to the first byte of an AMD64 instruction or the start of the fetch block. The address is valid only if the IbsFetchCtl[IbsFetchVal]=1. The address is in canonical form.

IBS Fetch Physical Address Register (IbsFetchPhyAd) (MSR C001_1032h)



[AMD Public Use]

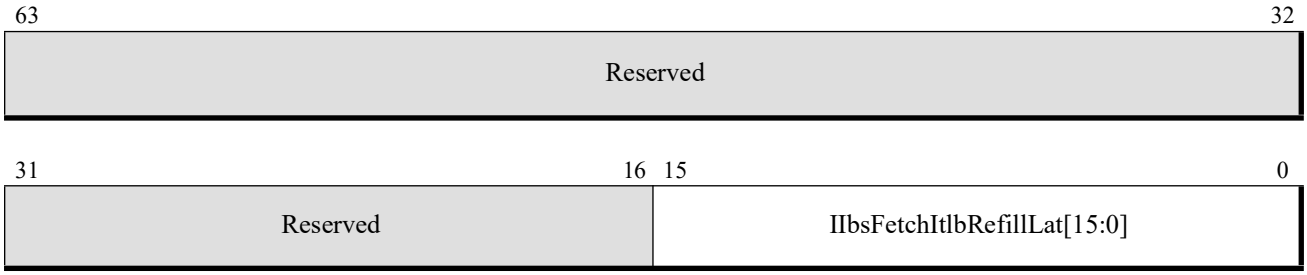
Bits	Mnemonic	Description	Access type
63:52	Reserved		RAZ
51:0	IbsFetchPhysAd	IBS Fetch Physical Address	RO

Figure 13-21. IBS Fetch Physical Address Register (IbsFetchPhysAd)

This is a read-only register. Reading the IbsFetchPhysAd MSR returns the 52-bit physical address of the tagged fetch. This address may correspond to the first byte of an AMD64 instruction or the start of the fetch block. The address is valid only if both the IbsPhyAddrValid and the IbsFetchVal bits of the IbsFetchCtl register are set. Otherwise, the contents of this register are undefined. The indicated size of 52 bits is an architectural limit. Specific processors may implement fewer bits.

IBS Fetch Extended Control Register (IbsFetchExtdCtl) (MSR C001_103Ch)

Support for IbsFetchExtdCtl is indicated by CPUID Fn8000_0001B_EAX[IbsFetchCtlExt] (bit 9) = 1.



Bits	Mnemonic	Description	Access type
63:16	Reserved		RAZ
15:0	IbsFetchItlbRefillLat	IBS Data Cache Physical Address	R/W

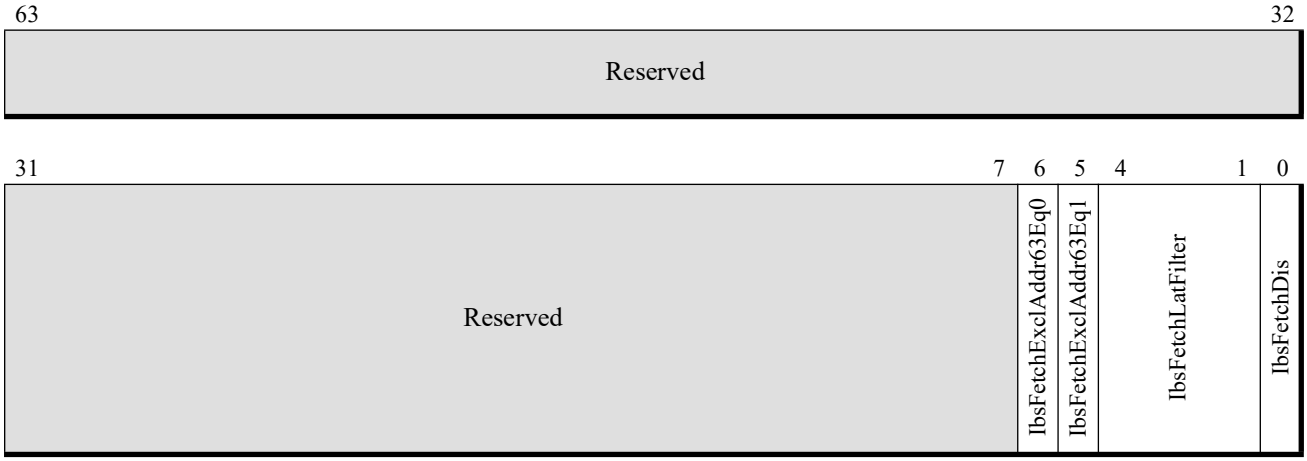
Figure 13-22. IBS Fetch Extended Control Register (IbsFetchExtdCtl)

The fields of IbsFetchExtdCtl are further described below

- *IbsFetchItlbRefillLat (IBS Fetch ITLB Refill Latency)*—Bits 15:0, read/write. This 16-bit field indicates the number of core clock cycles the fetch engine is stalled for an ITLB reload for the sampled fetch. If there is no reload, the reported latency is zero.

[AMD Public Use]

IBS Fetch Control Register 2 (IbsFetchCtl2) (MSR C001_103Fh)



Bits	Mnemonic	Description	Access type
63:7	Reserved		RAZ
6	IbsFetchExclAddr63Eq0	Exclude IBS Fetch samples with fetch address bit 63 cleared	R/W
5	IbsFetchExclAddr63Eq1	Exclude IBS Fetch samples with fetch address bit 63 set	R/W
4:1	IbsFetchLatFilter	IBS Fetch Latency filter threshold	R/W
0	IbsFetchDis	IBS Fetch Sampling Disable	R/W

Figure 13-23. IBS Fetch Control Register 2 (IbsFetchCtl2)

The fields of IbsFetchCtl2 are further described below:

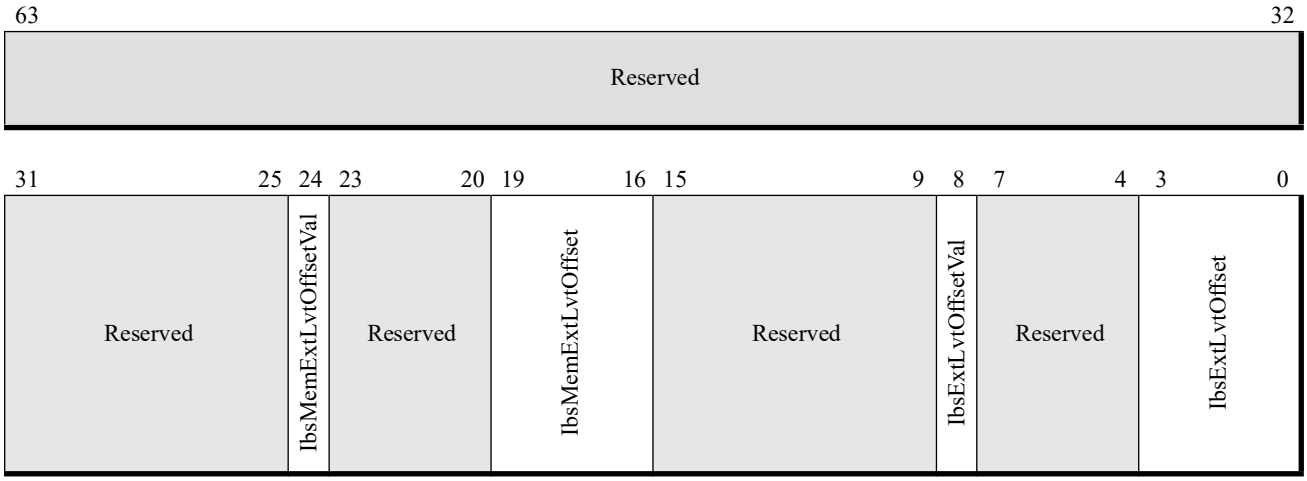
- *IbsFetchExclAddr63Eq0 (Exclude IBS Fetch samples with fetch address bit 63 cleared)*—Bit 6, read/write. This bit controls IBS Fetch Filtering. When set, Fetch IBS excludes samples when the logical fetch address (IbsFetchLinAd) has bit 63 cleared. IbsFetchExclAddr63Eq0 is supported when CPUID Fn8000_0001B_EAX[IbsAddrBit63Filtering] (bit 15) = 1.
- *IbsFetchExclAddr63Eq1 (Exclude IBS Fetch samples with fetch address bit 63 set)*—Bit 5, read/write. This bit controls IBS Fetch Filtering. When set, Fetch IBS excludes samples when the logical fetch address (IbsFetchLinAd) has bit 63 set. IbsFetchExclAddr63Eq1 is supported when CPUID Fn8000_0001B_EAX[IbsAddrBit63Filtering] (bit 15) = 1.
- *IbsFetchLatFilter (IBS Fetch Latency filter threshold)*—Bits 4:1, read/write. This bit controls IBS Fetch Filtering. When programmed with a non-zero value, Fetch IBS only reports samples when the fetch latency (IbsFetchCtl[IbsFetchLat]) is greater or equal to this field (IbsFetchLatFilter) times 128. When programmed with a non-zero value and IBS L3 Miss Filtering is active IbsFetchCtl[IbsL3MissOnly] = 1), the Fetch IBS sample must be an L3 miss IbsFetchCtl[IbsL3Miss] = 1) and the fetch latency threshold condition must be met for Fetch IBS to report a sample. This field is supported when CPUID Fn8000_0001B_EAX[IbsFetchLatencyFiltering] (bit 14) = 1.

[AMD Public Use]

- *IbsFetchDis (IBS Fetch Sampling Disable)*—Bit 0, read/write. When set, IBS Fetch Sampling is disabled. IBS Fetch Sampling is enabled if IbsFetchDis = 0, IbsFetchCtl[IbsFetchEn] = 1, and IbsFetchCtl[IbsFetchVal] = 0. This bit is supported when CPUID Fn8000_0001B_EAX[IbsDis] (bit 13) = 1.

IBS Control Register (IbsCtl) (MSR C001_103Ah)

The IBS Control Register specifies the extended local vector table (LVT) entries used to deliver IBS Fetch, Execution Sampling, and Memory Profiler interrupts. This register is read-only; attempts to write it generate a general-protection exception (#GP).



Bits	Mnemonic	Description	Access type
63:25	Reserved		RO
24	IbsMemExtLvtOffsetVal	Memory Profiler IBS Extended LVT Offset Valid	RO
23:20	Reserved		RO
19:16	IbsMemExtLvtOffset	Memory Profiler IBS Extended LVT Offset	RO
15:9	Reserved		RO
8	IbsExtLvtOffsetVal	IBS Extended LVT Offset Valid	RO
7:4	Reserved		RO
3:0	IbsExtLvtOffset	IBS Extended LVT Offset	RO

Figure 13-24. IBS Control Register (IbsCtl)

The fields shown in Figure 13-24 are further described below:

- *IbsMemExtLvtOffsetVal (Memory Profiler IBS Extended LVT Offset Valid)*—Bit 24, read-only. When this bit is set, the IbsMemExtLvtOffset field contains a valid extended LVT offset used for the delivery of IBS Memory Profiler interrupts. This field is supported when CPUID Fn8000_001B_EAX[IbsMemProfilerV1](bit 18) = 1.
- *IbsMemExtLvtOffset[3:0] (Memory Profiler IBS Extended LVT Offset)*—Bits 19:16, read-only. This field specifies the offset into the extended local APIC vector table used to deliver IBS

[AMD Public Use]

Memory Profiler interrupts. The APIC register address of the corresponding extended LVT entry is computed as (IbsMemExtLvtOffset << 4) + 500h. This field is supported when CPUID Fn8000_001B_EAX[IbsMemProfilerV1](bit 18) = 1.

- *IbsExtLvtOffsetVal (IBS Extended LVT Offset Valid)*—Bit 8, read-only. When this bit is set, the IbsExtLvtOffset field contains a valid extended LVT offset used for the delivery of IBS Fetch and Execution Sampling interrupts.
- *IbsExtLvtOffset[3:0] (IBS Extended LVT Offset)*—Bits 3:0, read-only. This field specifies the offset into the extended local APIC vector table used to deliver IBS Fetch and Execution Sampling interrupts. The APIC register address of the corresponding extended LVT entry is computed as (IbsExtLvtOffset << 4) + 500h.

13.3.3 IBS Execution Sampling

Instruction execution performance is measured by tagging an op associated with an instruction. The tagged op joins other ops in a queue waiting to be dispatched and executed. Instructions that decode to more than one op may return different performance data depending upon which op associated with the instruction is tagged. IBS returns the following performance information for each retired tagged op:

- Branch status for branch ops.
- For a load or store op:
  - Whether the load or store missed in the data cache.
  - Whether the load or store address hit or missed in the TLBs.
  - The linear and physical address of the data operand associated with the load or store operation.
  - Source information for cache, DRAM, MMIO, or I/O accesses.

IBS selects and tags an op at a programmable rate. When enabled by the IBS Execution Control Register (IbsOpEn = 1 and IbsOpVal = 0) and not disabled by IBS Execution Control Register 2 (IbsOpCtl2[IbsOpDis] = 0), an internal 27-bit op interval counter increments either once for every core clock cycle, if IbsOpCntCtl is cleared, or once for every dispatched op, if IbsOpCntCtl is set IbsOpCtl2[IbsOpDis] is supported on processors which set CPUID Fn8000_0001B_EAX[IbsDis] (bit 13) = 1. It allows software to start and stop IBS Execution Sampling with a write to a register not updated by the processor thus eliminating a read-modify-write hazard which exists for modifying bits in IBS Execution Control while IBS Execution Sampling is active.

When the value in bits 26:4 of the op counter equals the value in the IbsOpMaxCnt field of the IBS Execution Control Register, and no op tagged for execution sampling is in flight, an op is tagged in the next cycle if IbsOpCntCtl is cleared, or the op that causes the op counter to equal IbsOpMaxCnt is tagged if IbsOpCntCtl is set. Once the op counter reaches IbsOpMaxCnt, it resets to a small random value and continues counting cycles or ops. This reset happens even when no op is tagged. When the tagged op retires, the execution status of the op is written to the IBS execution registers, and the IbsOpVal bit of the IBS Execution Control Register is set. When this is complete, an interrupt is signaled to the local APIC. The local APIC should be programmed to deliver this interrupt to the processor core.

[AMD Public Use]

The interrupt service routine must save the performance information stored in IBS execution registers. Software can then initiate another sample by resetting the IbsOpVal bit in the IBS Execution Control Register.

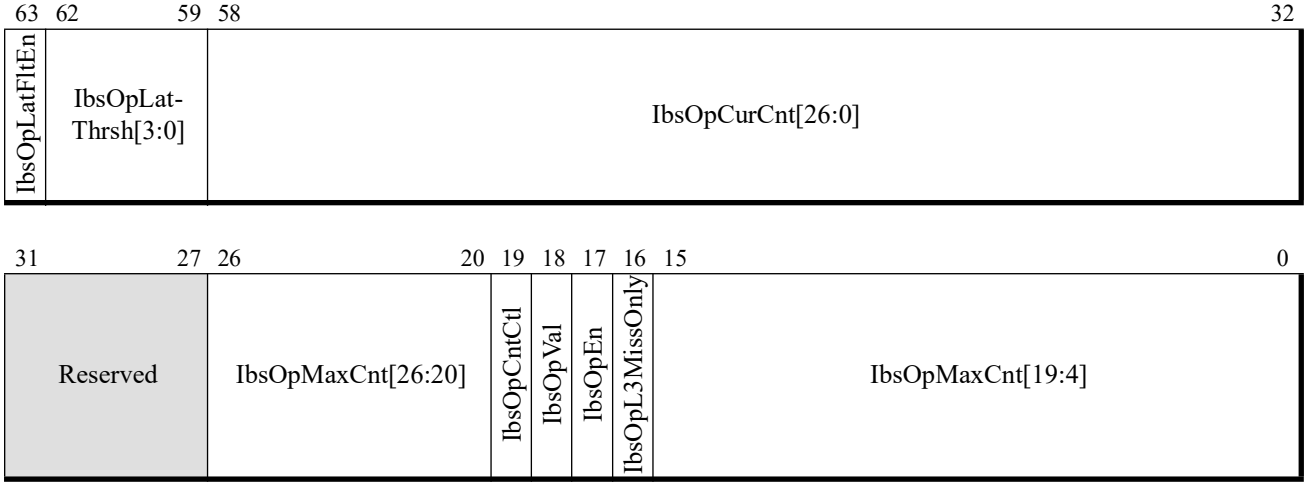
Aborted ops do not produce an IBS execution sample. If the tagged op aborts (i.e., does not retire), hardware resets bits 26:7 of the op interval counter to zero, and bits 6:0 to a random value. The op counter continues to increment and another op is selected when the value in bits 26:4 of the op interval counter equals the value in the IbsOpMaxCnt field.

Randomization of sampling interval: A degree of randomization of the sampling interval is necessary to ensure fairness in sampling, especially for loop-intensive code. For execution sampling this must be done by software. This is accomplished when writing the IbsOpCtl register to clear the IbsOpVal bit and initiate a new sampling interval. At that time software can provide a small random number (4-6 bits) in the IbsOpCurCnt field to offset the starting count, thereby randomizing the point at which the count reaches the IbsOpMaxCnt value and triggers a sample.

13.3.4 IBS Execution Sampling Registers

The IBS execution sampling registers consist of the control register (IBS Execution Control Register), the linear address register (IBS Op Linear Address Register), and three execution data registers (IBS Op Data 1–3). The IBS execution sampling registers are accessed using the RDMSR and WRMSR instructions.

IBS Execution Control Register (IbsOpCtl) (MSR C001_1033h)



[AMD Public Use]

Bits	Mnemonic	Description	Access type
63	IbsOpLatFltEn	IBS Op Load Latency Filter Enable	R/W
62:59	IbsOpLatThrsh[3:0]	IBS Op Load Latency Threshold	R/W
58:32	IbsOpCurCnt[26:0]	IBS Op Current Count, bits 26:0	R/W
31:27	Reserved		RAZ
26:20	IbsOpMaxCnt[26:20]	IBS Op Maximum Count, bits 26:20	R/W
19	IbsOpCntCtl	IBS Op Counter Control	R/W
18	IbsOpVal	IBS Op Sample Valid	R/W
17	IbsOpEn	IBS Op Sampling Enable	R/W
16	IbsOpL3MissOnly	IBS Op L3 Miss Filtering	R/W
15:0	IbsOpMaxCnt[19:4]	IBS Op Maximum Count, bits 19:4	R/W

Figure 13-25. IBS Execution Control Register (IbsOpCtl)

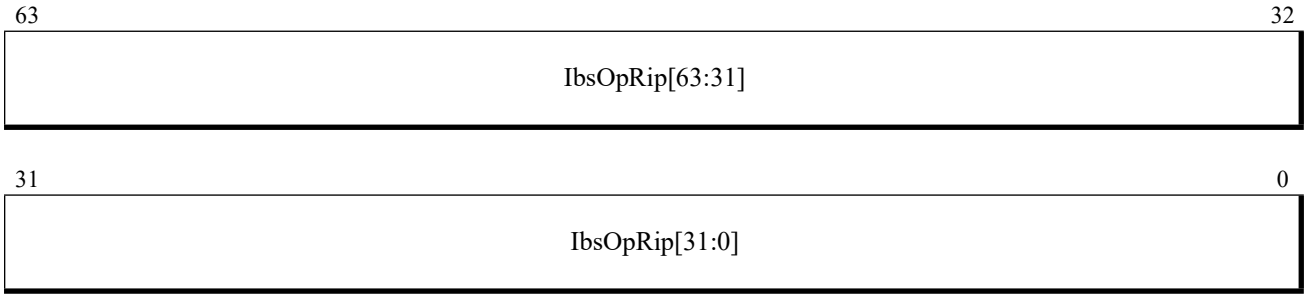
The fields shown in Figure 13-25 are further described below:

- *IbsOpLatFltEn (IBS Op Load Latency Filter Enable)*—Bit 63, read/write. This bit controls Load Latency Filtering. When set, IBS Execution Sampling only generates an interrupt for load operations whose latency equals or exceeds the threshold specified by the IbsOpLatThrsh field. This bit is supported when CPUID Fn8000_001B_EAX[IbsLoadLatencyFiltering] (bit 12) = 1.
- *IbsOpLatThrsh[3:0] (IBS Op Load Latency Threshold)*—Bits 62:59, read/write. This field specifies the minimum load latency, in units of 128 core clock cycles, required for an execution sample to generate an interrupt when load latency filtering is enabled. The threshold is computed as (IbsOpLatThrsh + 1) * 128 cycles. This field is valid only when IbsOpLatFltEn is set to 1. This field is supported when CPUID Fn8000_001B_EAX[IbsLoadLatencyFiltering](bit 12) = 1.
- *IbsOpCurCnt[26:0] (IBS Op Current Count)*—Bits 58:32, read/write. This field returns the current value of the op counter, and provides the starting value of the counter when software writes this register to clear the IbsOpVal bit and start another sampling interval.
- *IbsOpMaxCnt[26:20] (IBS Op Maximum Count[26:20])*—Bits 26:20, read/write. This field is used to specify the most significant 7 bits of the IbsOpMaxCnt.
- *IbsOpCntCtl (IBS Op Counter Control)*—Bit 19, read/write. This bit controls op tagging. When this bit is zero, IBS counts core clock cycles in order to select an op for tagging. When this bit is one, IBS counts dispatched ops in order to select an op for tagging.
- *IbsOpVal (IBS Op Sample Valid)*—Bit 18, read/write. This bit is set when a tagged op retires and indicates that new instruction execution data is available. The op counter stops counting. An interrupt is generated as specified by the local APIC. The software interrupt handler captures the *performance data before clearing the bit to enable the hardware to take another sample.*
- *IbsOpEn (IBS Op Sample Enable)*—Bit 17, read/write. Software sets this bit to enable IBS execution sampling. Clearing this bit disables IBS execution sampling.
- *IbsOpL3MissOnly (IBS Op L3 Miss Filtering)*—Bit 16, read/write. This bit controls L3 Miss Filtering. When set, Execution IBS only sends interrupts for samples with an L3 miss. See Section 13.3.5, “IBS Filtering,” on page 444 for a description of Execution IBS Filtering.

[AMD Public Use]

- *IbsOpMaxCnt[19:4] (IBS Op Maximum Count[19:4])*—Bits 15:0, read/write. This field specifies the maximum count value for bits 19:4 of the op interval counter. When the value in bits 26:4 of the op interval counter equal the value specified by the concatenation of the IbsOpMaxCnt[26:20] field with this field, the next op is tagged for profiling.

IBS Op Linear Address Register (IbsOpRip) (MSR C001_1034h)



Bits	Mnemonic	Description	Access type
63:0	IbsOpRip	IBS Op Linear Address	R/W

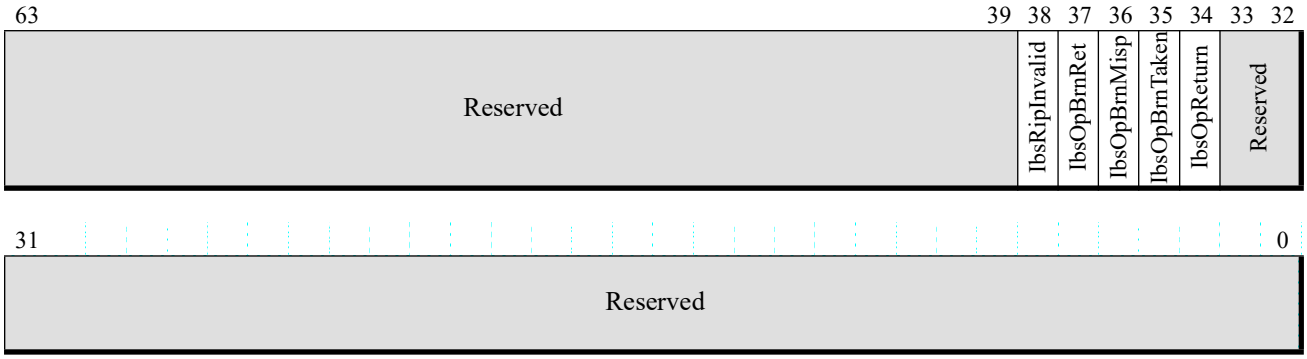
Figure 13-26. IBS Op Linear Address Register (IbsOpRip)

IbsOpRip[63:0] (IBS Op Linear Address)—Bits 63:0, read/write. Specifies the linear address for the instruction from which the tagged op was issued. The address is valid only if the IbsOpCtl[IbsOpVal] bit is set and the IbsOpData1[IbsRipInvalid] bit is cleared. The address is in canonical form.

[AMD Public Use]

IBS Op Data 1 Register (IbsOpData1) (MSR C001_1035h)

The IBS Op Data 1 Register provides core cycle counts for tagged ops and performance data for tagged ops which perform a branch.



Bits	Mnemonic	Description	Access type
63:39	Reserved		RAZ
38	IbsRipInvalid	IbsOpRip Register Invalid	R/W
37	IbsOpBrnRet	IBS Op Branch Retired	R/W
36	IbsOpBrnMisp	IBS Op Branch Mispredicted	R/W
35	IbsOpBrnTaken	IBS Op Branch Taken	R/W
34	IbsOpReturn	IBS Op RET	R/W
33:0	Reserved		RAZ

Figure 13-27. IBS Op Data 1 Register (IbsOpData1)

The fields shown in Figure 13-27 are further described below:

- *IbsRipInvalid (IbsOpRip Register Invalid)*—Bit 38, read/write. If this bit is set, the contents of the IbsOpRip register are not valid.
- *IbsOpBrnRet (IBS Op Branch Retired)*—Bit 37, read/write. This bit is set if the tagged op performs a branch that retired.
- *IbsOpBrnMisp (IBS Op Branch Mispredicted)*—Bit 36, read/write. This bit is set if the tagged op performs a retired mispredicted branch.
- *IbsOpBrnTaken (IBS Op Branch Taken)*—Bit 35, read/write. This bit is set if the tagged op performs a retired taken branch.
- *IbsOpReturn (IBS Op RET)*—Bit 34, read/write. This bit is set if the tagged op performs a retired subroutine return (RET).

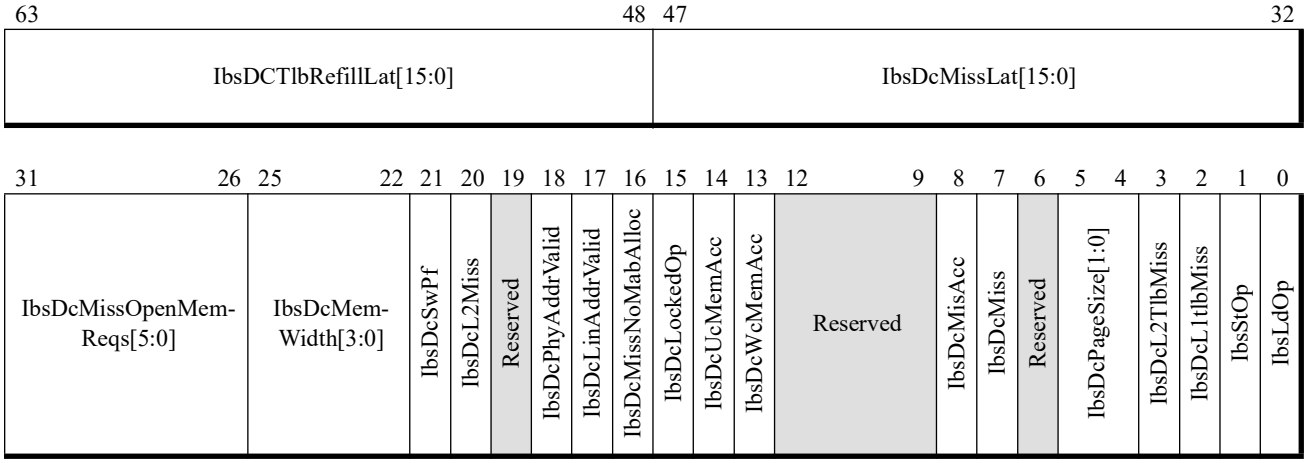
[AMD Public Use]

IBS Op Data 2 Register (IbsOpData2) (MSR C001_1036h)

The IBS Op Data 2 Register captures Northbridge-related performance data. The information captured is implementation-dependent. See the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product for details.

IBS Op Data 3 Register (IbsOpData3) (MSR C001_1037h)

Data Cache (first-level cache) performance data is captured in IBS Op Data 3 Register. If a load or store operation crosses a 128-bit boundary, the data returned in this register is the data for the access to the data below the 128-bit boundary.



[AMD Public Use]

Bits	Mnemonic	Description	Access type
63:48	IbsDCTlbRefillLat[15:0]	IBS Data Cache TLB Refill Latency	R/W
47:32	IbsDcMissLat[15:0]	IBS Data Cache Miss Latency	R/W
31:26	IbsDcMissOpenMemReqs[5:0]	IBS Data Cache Outstanding Memory Requests at Fill	R/W
25:22	IbsDcMemWidth[3:0]	IBS Data Cache Memory Access Width	R/W
21	IbsDcSwPf	IBS Data Cache Software Prefetch	R/W
20	IbsDcL2Miss	IBS Data Cache L2 Cache Miss	R/W
19	Reserved		RAZ
18	IbsDcPhyAddrValid	IBS Data Cache Physical Address Valid	R/W
17	IbsDcLinAddrValid	IBS Data Cache Linear Address Valid	R/W
16	IbsDcMissNoMabAlloc	IBS Data Cache Miss Without MAB Allocation	R/W
15	IbsDcLockedOp	IBS Data Cache Locked Op	R/W
14	IbsDcUcMemAcc	IBS Data Cache UC Memory Access	R/W
13	IbsDcWcMemAcc	IBS Data Cache WC Memory Access	R/W
12:9	Reserved		RAZ
8	IbsDcMisAcc	IBS Data Cache Misaligned Access Penalty	R/W
7	IbsDcMiss	IBS Data Cache Miss	R/W
6	Reserved		RAZ
5:4	IbsDcPageSize[1:0]	IBS Data Cache Page Size	R/W
3	IbsDcL2TlbMiss	IBS Data Cache L2 TLB Miss	R/W
2	IbsDcL1tlbMiss	IBS Data Cache L1 TLB Miss	R/W
1	IbsStOp	IBS Store Operation	R/W
0	IbsLdOp	IBS Load Operation	R/W

Figure 13-28. IBS Op Data 3 Register (IbsOpData3)

Note: bits 5:4 replace the existing IbsDcL1tlbHit1G (bit 5) and IbsDcL1tlbHit2M (bit 4) entries on processors where CPUID Fn8000_001B_EAX[IbsUpdtdDtlbStats](bit 19) = 1.

The fields shown in Figure 13-28 are further described below:

- *IbsDCTlbRefillLat[15:0] (IBS Data Cache TLB Refill Latency)*—Bits 63:48, read/write. This 16-bit field indicates the number of core clock cycles from when a data cache TLB refill is triggered to when the refill completes. This field is valid only when IbsDcPhyAddrValid is set to 1.
- *IbsDcMissLat[15:0] (IBS Data Cache Miss Latency)*—Bits 47:32, read/write. This field indicates the number of core clock cycles from when a miss is detected in the data cache to when the data is delivered to the core. The value is not valid for data cache store operations.
- *IbsDcMissOpenMemReqs[5:0] (IBS Data Cache Outstanding Memory Requests at Fill)*—Bits 31:26, read/write. This field indicates the number of allocated, valid Miss Address Buffer (MAB) entries at the time the MAB entry for the tagged data cache miss op is deallocated. A value of 00h indicates that no information is available.

[AMD Public Use]

- *IbsDcMemWidth[3:0] (IBS Data Cache Memory Access Width)*—Bits 25:22, read/write. This field indicates the size of the load or store memory access in bytes. The table below defines the encoding of this field:

Value	Access Size
0h	No information available
1h	Byte (1 byte)
2h	Word (2 bytes)
3h	Doubleword (4 bytes)
4h	Quadword (8 bytes)
5h	Octword (16 bytes)
6h	32 bytes
7h	64 bytes
8h-Fh	Reserved

- *IbsDcSwPf (IBS Data Cache Software Prefetch)*—Bit 21, read/write. This bit is set when the tagged op belongs to a software prefetch instruction.
- *IbsDcL2Miss (IBS Data Cache L2 Cache Miss)*—Bit 20, read/write. This bit is set when the tagged load or store op missed in the L2 cache.
- *IbsDcPhyAddrValid (IBS Data Cache Physical Address Valid)*—Bit 18, read/write. If this bit and IbsDcLinAddrValid are set the physical address in the IBS DC Physical Address Register is valid for a load or store operation.
- *IbsDcLinAddrValid (IBS Data Cache Linear Address Valid)*—Bit 17, read/write. This bit is set if the linear address in the IBS DC Linear Address Register is valid for a load or store operation.
- *IbsDcMissNoMabAlloc (IBS Data Cache Miss Without MAB Allocation)*—Bit 16, read/write. This bit is set when the tagged load or store op missed in the data cache but hit on an already allocated Miss Address Buffer (MAB) entry. In this case a new MAB entry was not allocated for the operation.
- *IbsDcLockedOp (IBS Data Cache Locked Op)*—Bit 15, read/write. This bit is set if the tagged load or store operation was a locked operation.
- *IbsDcUcMemAcc (IBS Data Cache UC Memory Access)*—Bit 14, read/write. This bit is set if the tagged load or store operation accessed uncacheable memory.
- *IbsDcWcMemAcc (IBS Data Cache WC Memory Access)*—Bit 13, read/write. This bit is set if the tagged load or store operation accessed write combining memory.
- *IbsDcMisAcc (IBS Data Cache Misaligned Access Penalty)*—Bit 8, read/write. This bit is set if a tagged load or store operation incurred a performance penalty due to a misaligned access.
- *IbsDcMiss (IBS Data Cache Miss)*—Bit 7, read/write. This bit is set if the cache line used by the tagged load or store operation was not present in the data cache.
- IbsDcPageSize[1:0] (IBS Data Cache Page Size)--Bits 5:4, read/write. This field indicates the page size of the translation used by the data cache for the tagged load or store op. This field is valid only when IbsDcPhyAddrValid is set to 1. The table below defines the encoding of this two-bit field:

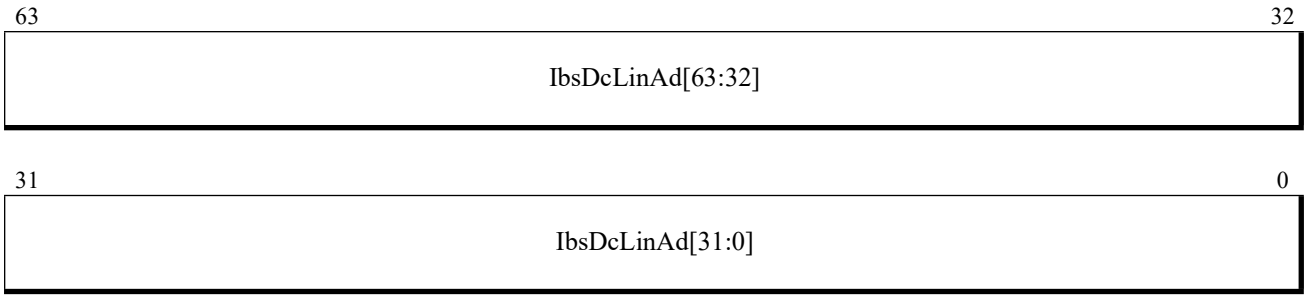
[AMD Public Use]

Value	Page Size
00b	4 Kbyte
01b	2 Mbyte
10b	1 Gbyte
11b	Reserved

This field is supported when CPUID Fn8000_001B_EAX[IbsUpdtdDtlbStats] (bit 19) = 1.

- *IbsDcL1tlbHit1G (IBS Data Cache L1 TLB Hit 1-Gbyte Page)*—Bit 5, read/write. This bit is set if the physical address for the tagged load or store operation was present in a 1-Gbyte page table entry in the data cache L1 TLB. This bit not supported when CPUID Fn8000_001B_EAX[IbsUpdtdDtlbStats] (bit 19) = 1.
- *IbsDcL1tlbHit2M (IBS Data Cache L1 TLB Hit 2-Mbyte Page)*—Bit 4, read/write. This bit is set if the physical address for the tagged load or store operation was present in a 2-Mbyte page table entry in the data cache L1 TLB. This bit not supported when CPUID Fn8000_001B_EAX[IbsUpdtdDtlbStats] (bit 19) = 1.
- *IbsDcL2TlbMiss (IBS Data Cache L2 TLB Miss)*—Bit 3, read/write. This bit is set when a translatio the tagged load or store op was not present in the L2 TLB. This bit is valid only when IbsDcPhyAddrValid is set to 1.
- *IbsDcL1tlbMiss (IBS Data Cache L1 TLB Miss)*—Bit 2, read/write. This bit is set if the physical address for the tagged load or store operation was not present in the data cache L1 TLB.
- *IbsStOp (IBS Store Op)*—Bit 1, read/write. This bit is set if the tagged op was a store.
- *IbsLdOp (IBS Load Op)*—Bit 0, read/write. This bit is set if the tagged op was a load.

IBS Data Cache Linear Address Register (IbsDcLinAd) (MSR C001_1038h)



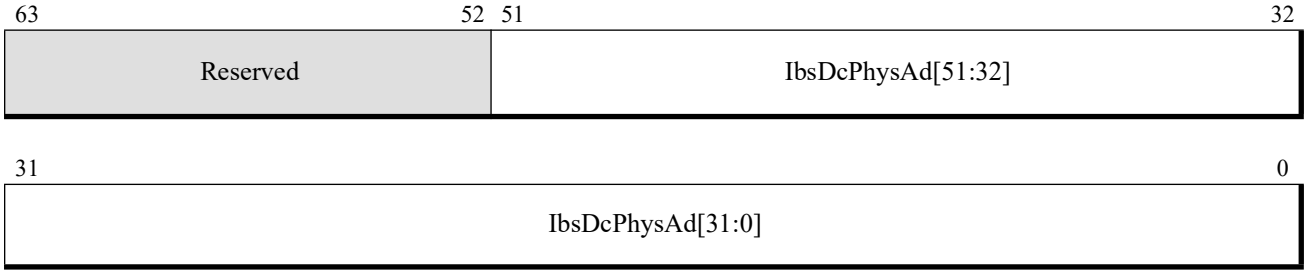
Bit	Mnemonic	Description	Access type
63:0	IbsDcLinAd	IBS Data Cache Linear Address	R/W

Figure 13-29. IBS Data Cache Linear Address Register (IbsDcLinAd)

IbsDcLinAd[63:0] (IBS Data Cache Linear Address)—Bits 63:0, read/write. Specifies the linear address of the tagged op's memory operand. The address is valid only if IbsOpData3[IbsDcLinAddrValid] is set. The address is in canonical form.

[AMD Public Use]

IBS Data Cache Physical Address Register (IbsDcPhysAd) (MSR C001_1039h)

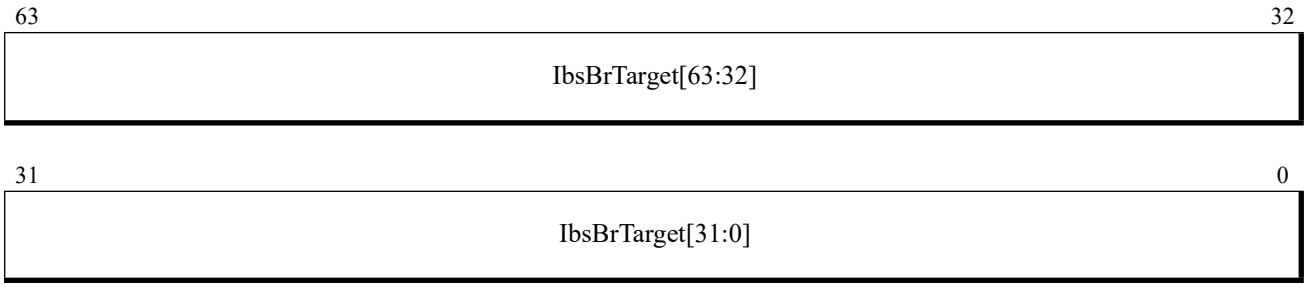


Bits	Mnemonic	Description	Access type
63:52	Reserved		RAZ
51:0	IbsDcPhysAd	IBS Data Cache Physical Address	R/W

Figure 13-30. IBS Data Cache Physical Address Register (IbsDcPhysAd)

IbsDcPhysAd (IBS Data Cache Physical Address)—Bits 51:0, read/write. Specifies the physical address of the tagged op's memory operand. The address is valid only if IbsOpData3[IbsDcPhyAddrValid] and IbsOpData3[IbsDcLinAddrValid] are set.

IBS Branch Target Address Register (IbsBrTarget) (MSR C001_103Bh)

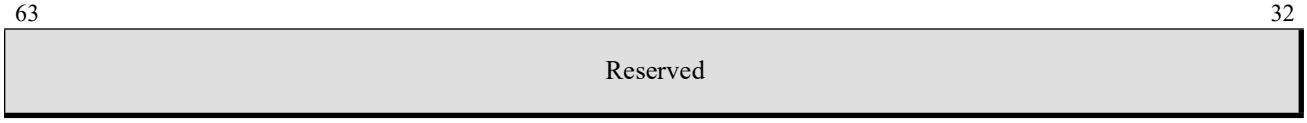


Bits	Mnemonic	Description	Access type
63:0	IbsBrTarget	IBS branch target linear address	R/W

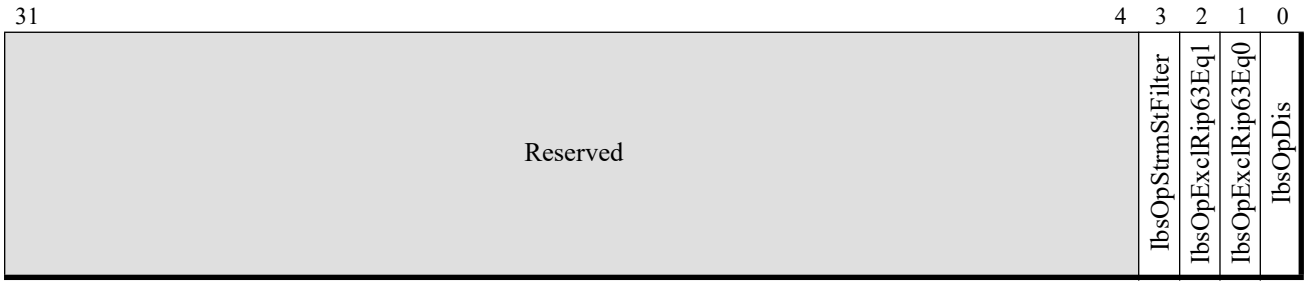
Figure 13-31. IBS Branch Target Address Register (IbsBrTarget)

IbsBrTarget (IBS Branch Target)—Bits 63:0, read/write. Specifies the 64-bit linear address for the branch target. The address is in canonical form. For non-branch instructions, the value supplied in this register is invalid. For a conditional branch not taken, the value supplied in this register is the fall-through address.

IBS Execution Control Register 2 (IbsOpCtl2) (MSR C001_103Eh)



[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:4	Reserved		RAZ
3	IbsOpStrmStFilter	IBS Op Streaming Store Filter Control	R/W
2	IbsOpExclRip63Eq1	Exclude IBS Op samples with RIP bit 63 set	R/W
1	IbsOpExclRip63Eq0	Exclude IBS Op samples with RIP bit 63 cleared	R/W
0	IbsOpDis	IBS Execution Sampling disable	R/W

Figure 13-32. IBS Execution Control Register 2 (IbsOpCtl2)

The fields of IbsOpCtl2 are further described below:

- *IbsOpStrmStFilter (IBS Op Streaming Store Filter Control)*—Bit 3, read/write. This bit controls Execution IBS Filtering. When set, samples for streaming stores (IbsOpData2[StrmSt] = 1) are reported. For more detail on IBS Filtering see section “IBS Filtering”. IbsOpStrmStFilter is supported when CPUID Fn8000_001B_EAX[IbsStrmStAndRmtSocket] (bit 16) = 1.
- *IbsOpExclRip63Eq1 (Exclude IBS Execution samples with RIP bit 63 set)*—Bit 2, read/write. This bit controls IBS Execution Filtering. When set, Execution IBS excludes samples when the logical instruction address (IbsOpRip) has bit 63 set or IbsOpRip is not valid (IbsOpData1[IbsRipInvalid] = 1). IbsOpExclRip63Eq1 is supported when CPUID Fn8000_0001B_EAX[IbsAddrBit63Filtering] (bit 15) = 1.
- *IbsOpExclRip63Eq0 (Exclude IBS Execution samples with RIP bit 63 cleared)*—Bit 1, read/write. This bit controls IBS Execution Filtering. When set, Execution IBS excludes samples when the logical instruction address (IbsOpRip) has bit 63 cleared or IbsOpRip is not valid (IbsOpData1[IbsRipInvalid] = 1). IbsOpExclRip63Eq0 is supported when CPUID Fn8000_0001B_EAX[IbsAddrBit63Filtering] (bit 15) = 1.
- *IbsOpDis (IBS Execution Sampling Disable)*—Bit 0, read/write. When set, IBS Execution Sampling is disabled. IBS Execution Sampling is enabled if IbsOpDis = 0, IbsOpCtl[IbsOpEn] = 1, and IbsOpCtl[IbsOpVal] = 0. This bit is supported when CPUID Fn8000_0001B_EAX[IbsDis] (bit 13) = 1.

13.3.5 IBS Filtering

Processors that set CPUID Fn8000_001B_EAX[IbsL3MissFiltering](bit 11) = 1 support IBS Filtering for both IBS Fetch Sampling and IBS Execution Sampling. IBS Filtering changes the behavior of the IBS hardware such that interrupts are only generated for samples that match certain criteria. If the

[AMD Public Use]

sample does not match the selected criteria the IBS hardware restarts and selects a new sample. This results in reduced interrupt overhead if software is only interested in samples with certain characteristics.

Fetch IBS Filtering

When Fetch IBS collects a sample that does not match the selected filter criteria it does not set IbsFetchCtl[IbsFetchVal] and instead clears IbsFetchCtl[IbsFetchCnt] and bits 19:7 of the internal fetch counter. Bits 6:0 of the internal fetch counter are loaded with a pseudo-random value if IbsFetchCtl[IbsRandEn] is set and cleared if IbsFetchCtl[IbsRandEn] is not set. It is recommended that software sets IbsFetchCtl[IbsRandEn] when IBS Filtering is active. After its reset, the internal fetch counter increments for every attempted fetch. When the internal fetch counter's bits [19:4] match IbsFetchCtl[IbsFetchMaxCnt] the next fetch is selected for profiling. When the selected fetch completes or is aborted, the IBS hardware evaluates it against the IBS Filtering criteria and either sets IbsFetchCtl[IbsFetchVal] and signals an interrupt or restarts Fetch IBS in the manner described above.

Fetch IBS Filtering Pseudo Code

The following pseudo-code illustrates the processor's Fetch IBS Filtering behavior considering available IBS filter criteria.

The action `restart IBS` refers to the processor discarding a sample and restarting Fetch IBS. The action `report valid IBS sample` refers to the processor setting IbsFetchCtl[IbsFetchVal] and signaling an IBS interrupt or writing an IBS sample into the IBS Buffer. CPUID checks for the support of IBS filtering capabilities have been omitted for clarity and can be found in Table 13-7.

```
// fetch address bit 63 filtering
IF (IbsFetchCtl2[IbsFetchExclAddr63Eq0]==1 &&
      IbsFetchLinAd[63]==0)
      restart IBS

// fetch address bit 63 filtering
ELSIF (IbsFetchCtl2[IbsFetchExclAddr63Eq1]==1 &&
      IbsFetchLinAd[63]==1)
      restart IBS

// fetch latency filtering
ELSIF (IbsFetchCtl[IbsFetchLat] < IbsFetchCtl2[IbsFetchLatFilter]*128)
      restart IBS

// no L3-miss filtering enabled
elsif (IbsFetchCtl[IbsL3MissOnly]==0)
      report valid IBS sample

// L3-miss filtering
elsif (IbsFetchCtl[IbsL3MissOnly]==1 &&
      IbsFetchCtl[IbsFetchL3Miss]==1
      report valid IBS sample
ELSE
      restart IBS
```

[AMD Public Use]

Execution IBS Filtering

When Execution IBS collects a sample that does not match the selected filter criteria it does not set IbsOpCtl[IbsOpVal] and instead clears IbsOpCtl[IbsOpCurCnt] and bits 26:0 of the internal op counter. Bits 6:0 of the internal op counter are loaded with a pseudo-random value. After its reset, the internal op counter increments for every cycle (IbsOpCtl[IbsOpCntCtl] = 0) or dispatched op (IbsOpCtl[IbsOpCntCtl] = 1). When the internal op counter's bits [26:4] match IbsOpCtl[IbsOpMaxCnt] the next op is selected for profiling. When profiling is complete, the IBS hardware evaluates it against the IBS Filtering criteria and either sets IbsOpCtl[IbsOpVal] and signals an interrupt or restarts Execution IBS in the manner described above.

Execution and Memory Profiler IBS Filtering Pseudo Code

The following pseudo-code illustrates the processor's Fetch and Memory Profiler IBS Filtering behavior considering available IBS filter criteria. The action `restart IBS` refers to the processor discarding a sample and restarting Execution or Memory Profiler IBS. The action `report valid IBS sample` refers to the processor setting IbsFetchCtl[IbsFetchVal] or IbsMemCtl[IbsMemVal] and signaling an IBS interrupt or writing an IBS sample into the IBS Buffer. CPUID checks for the support of IBS filtering capabilities have been omitted for clarity and can be found in Table 13-7.

Execution and Memory Profiler IBS Filtering Pseudocode

```
IF (IbsOpCtl2[IbsOpExclRip63Eq0]==1 &&
      (IbsOpRip[63]==0 || IbsOpData1[IbsRipInvalid]==1))
      restart IBS

ELSIF (IbsOpCtl2[IbsOpExclRip63Eq1]==1 &&
      (IbsOpRip[63]==1 || IbsOpData1[IbsRipInvalid]==1))
      restart IBS

// no load latency, L3-miss or streaming store filtering is active
ELSIF (IbsOpCtl[IbsOpLatFltEn]==0 &&
      IbsOpCtl1[IbsOpL3MissOnly]==0 &&
      IbsOpCtl2[IbsOpStrmStFilter]==0)
      report valid IBS sample

// load latency filtering without L3 miss filtering
ELSIF (IbsOpCtl[IbsOpLatFltEn]==1 &&
      IbsOpCtl[IbsOpL3MissOnly]==0 &&
      IbsOpData3[IbsLdOp]==1 &&
      IbsOpData3[IbsSwPf]==0 &&
      (IbsOpData3[IbsDcMissLat] > (IbsOpData3[IbsOpLatThrsh]+1) * 128)
      report valid IBS sample

// load latency filtering with L3 miss filtering
ELSIF (IbsOpCtl[IbsOpLatFltEn]==1 &&
      IbsOpCtl[IbsOpL3MissOnly]==1 &&
      IbsOpData3[IbsL2Miss]==1 &&
      IbsOpData2[DataSrc[4:0]] > 01h) &&
      IbsOpData3[IbsLdOp]==1 &&
```

[AMD Public Use]


```
      IbsOpData3[IbsSwPf]==0 &&
      (IbsOpData3[IbsDcMissLat] > (IbsOpCtl[IbsOpLatThrsh]+1) * 128)
      report valid IBS sample

// L3-miss filtering without load latency filtering
ELSIF (IbsOpCtl[IbsOpLatFltEn]==0 &&
      IbsOpCtl[IbsOpL3MissOnly]==1 &&
      IbsOpData3[IbsL2Miss]==1 &&
      IbsOpData2[DataSrc[4:0]] > 01h)
      report valid IBS sample

// streaming store filtering
ELSIF (IbsOpCtl2[IbsOpStrmStFilter]==1 &&
      IbsOpData2[StrmSt]==1)
      report valid IBS sample
ELSE

restart IBS
```

Supported IBS Filter Criteria

Table 13-7 shows the supported filter criteria and how they are activated.

Table 13-7. Supported IBS Filtering Criteria

Filter Criterion	Active When	Supported If
Fetch IBS only reports samples with L3 miss	IbsFetchCtl[IbsL3MissOnly] = 1	CPUID Fn8000_001B_EAX[IbsL3MissFiltering](bit 11) = 1
Execution IBS only reports samples with L3 miss	IbsOpCtl[IbsOpL3MissOnly] = 1	CPUID Fn8000_001B_EAX[IbsL3MissFiltering](bit 11) = 1
Fetch IBS only reports samples meeting fetch latency threshold	IbsFetchCtl2[IbsFetchLatFilter] != 0	CPUID Fn8000_001B_EAX[IbsFetchLatencyFiltering](bit 14) = 1
Fetch IBS excludes samples where fetch address bit 63 is cleared	IbsFetchCtl2[IbsFetchExclAddr63Eq0] = 1	CPUID Fn8000_001B_EAX[IbsAddrBit63Filtering](bit 15) = 1
Fetch IBS excludes samples where fetch address bit 63 is set	IbsFetchCtl2[IbsFetchExclAddr63Eq1] = 1	CPUID Fn8000_001B_EAX[IbsAddrBit63Filtering](bit 15) = 1
Execution IBS only reports samples meeting a load latency threshold	IbsOpCtl[IbsOpLatFltEn] = 1	CPUID Fn8000_001B_EAX[IbsLoadLatencyFiltering] (bit 12) = 1

[AMD Public Use]

Table 13-7. Supported IBS Filtering Criteria

Execution IBS excludes samples where RIP bit 63 is cleared	IbsOpCtl2[IbsOpExclRip63Eq0] = 1	CPUID Fn8000_001B_EAX[IbsAddrBit63Filtering](bit 15) = 1
Execution IBS excludes samples where RIP bit 63 is set	IbsOpCtl2[IbsOpExclRip63Eq1] = 1	CPUID Fn8000_001B_EAX[IbsAddrBit63Filtering](bit 15) = 1
Execution IBS only reports samples for streaming stores	IbsOpCtl2[IbsOpStrmStFilter] = 1	CPUID Fn8000_001B_EAX[IbsStrmStAndRmtSocket](bit 16) = 1

13.3.6 IBS Buffering

IBS Buffering is a feature which reduces the software overhead for processing IBS sample interrupts. With IBS Buffering the processor suppresses interrupts for Fetch and Execution IBS and instead writes the samples (IBS register values) into a memory buffer. Support for IBS Buffering Version 1 is indicated by CPUID Fn8000_001B_EAX[IbsBufferV1] (bit 17) = 1. IBS Buffering is not supported for the IBS Memory Profiler.

Enabling and Using IBS Buffering

To use IBS Buffering, software configures the size of the IBS memory buffer in IbsBufferSize[IbsBufferSize], writes bits 63:12 of the IBS Buffer linear starting address into IbsBufferBase[IbsBufferLinAddr] and sets IbsBufferBase[IbsBufferEn]=1. The processor assumes that the buffer start address is aligned on a 4-Kbyte boundary with the low-order 12 address bits 11:0 assumed to be 0. IBS buffering is active if IbsBufferFail and IbsBufferFull bits in the IBS Buffer Tail register are clear.

Once IBS Buffering is enabled and active, the processor can write Fetch and Execution IBS samples into the IBS Buffer. When a new Fetch or Execution IBS sample is available and IBS Buffering is active the processor does not send an IBS interrupt but instead attempts to write the IBS sample into the IBS Buffer. When IbsBufferBase[TscRecordEn]=1, the processor also writes a TSC record before it writes a Fetch or Execution IBS record or both if they become ready close in time. The format of Fetch IBS, Execution IBS and TSC records is described in“IBS Buffer Records on page 452”.

Once the buffer write succeeds, the processor advances IbsBufferTail by the number of quadwords written to the IBS Buffer. If the remaining buffer capacity (IbsBufferSize[IbsBufferSize] – IbsBufferTail[IbsBufferTail]) is insufficient to write the maximum number of records that could be written at any time, assuming both Fetch and Execution IBS are enabled, the processor sets IbsBufferTail[IbsBufferFull] = 1 and signals an IBS interrupt as specified by the local APIC. The processor may also set IbsFetchCtl[IbsFetchVal]=1 and IbsOpCtl[IbsOpVal]=1 to temporarily halt IBS sampling.

When servicing an IBS interrupt with IBS buffering enabled, software should not interpret IbsFetchCtl[IbsFetchVal] = 1 or IbsOpCtl[IbsOpVal] = 1 as valid samples in the Fetch and Execution IBS register but instead check IbsBufferTail[IbsBufferFull] and IbsBufferTail[IbsBufferFail].

[AMD Public Use]

When IbsBufferTail[IbsBufferFull] is set, the processor successfully wrote all IBS sample information into the memory buffer and software should empty the IBS buffer followed by writing IbsBufferTail[IbsBufferTail] to zero and clearing IbsBufferTail[IbsBufferFull]. When IbsBufferTail[IbsBufferFull] is cleared, the processor clears IbsFetchCtl[IbsFetchVal] and IbsOpCtl[IbsOpVal] to resume temporarily halted IBS activity.

IBS Buffering is disabled by INIT.

Failed IBS Buffer Writes

An IBS buffer write can fail due to a fault or other reasons. When an IBS buffer write fails, the processor sets IbsBufferTail[IbsBufferFail] indicating that IBS sample data has not been written and is available in the Fetch IBS and/or Execution IBS registers as indicated by their respective IbsVal bits. Failed IBS Buffer writes do not write a TSC record. When setting IbsBufferTail[IbsBufferFail] the processor signals an interrupt as specified by the local APIC. On a failed buffer write, the processor does not set IbsBufferTail[IbsBufferFull].

If an IBS buffer write fails due to a fault, the processor does not raise the fault exception when attempting the IBS buffer write. When processing a failed IBS buffer write interrupt, software should perform a write, equal to the size of the largest IBS buffer entry that may be written by IBS Buffering, to the current IBS buffer tail pointer location to trigger the exception.

IBS Buffer writes automatically fail when the processor is in SMM or paging is disabled (CR0.PG = 0). With IBS Buffering enabled, software should only check IbsFetchCtl[IbsFetchVal] and IbsOpCtl[IbsOpVal] when IbsBufferTail[IbsBufferFail] is set.

Recommended IBS Interrupt Handling Pseudo-Code with IBS Buffering

The following pseudo-code shows the recommended software flow when handling IBS interrupts with IBS Buffering enabled.

```
IF IbsBufferTail[IbsBufferFull] == 1
      process IBS buffer
      write IbsBufferTail[IbsBufferTail] = 0
      write IbsBufferTail[IbsBufferFull] = 0
ELSIF IbsBufferTail[IbsBufferFail] == 1
      Disable Fetch IBS and Execution IBS
      Write 64 bytes to IBS buffer base + IBS buffer tail
      IF IbsBufferTail[IbsBufferTail] > 0
            process IBS buffer
            write IbsBufferTail[IbsBufferTail] = 0
      IF IbsFetchCtl[IbsFetchVal] == 1
            Process Fetch IBS sample
            Write IbsFetchCtl[IbsFetchVal] = 0
      IF IbsOpCtl[IbsOpVal] == 1
            Process Execution IBS sample
            Write IbsOpCtl[IbsOpVal] = 0
      write IbsBufferTail[IbsBufferFail] = 0
      Reenable Fetch IBS and Execution IBS
ELSE
```

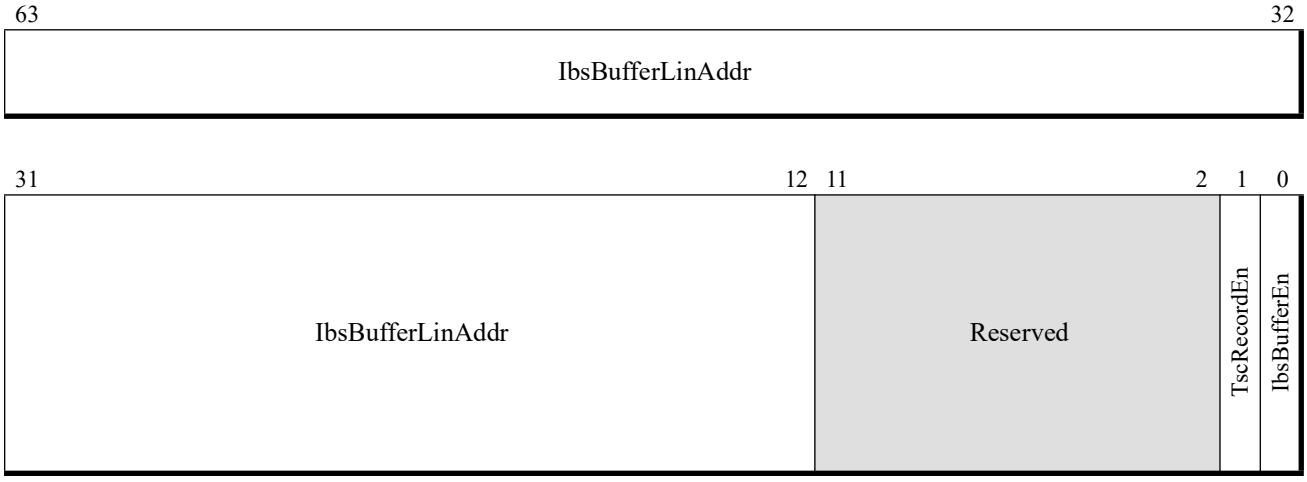
[AMD Public Use]


```
// No Fetch or Execution IBS data available for consumption
```

IBS Buffering Registers

The IBS Buffering Version 1 registers consist of the IBS Buffer Base register (IbsBufferBase), the IBS Buffer Size register (IbsBufferSize), and the IBS Buffer Tail register (IbsBufferTail). These registers can be accessed with WRMSR and RDMSR instructions.

IBS Buffer Base (IbsBufferBase) (MSR C001_0390h)



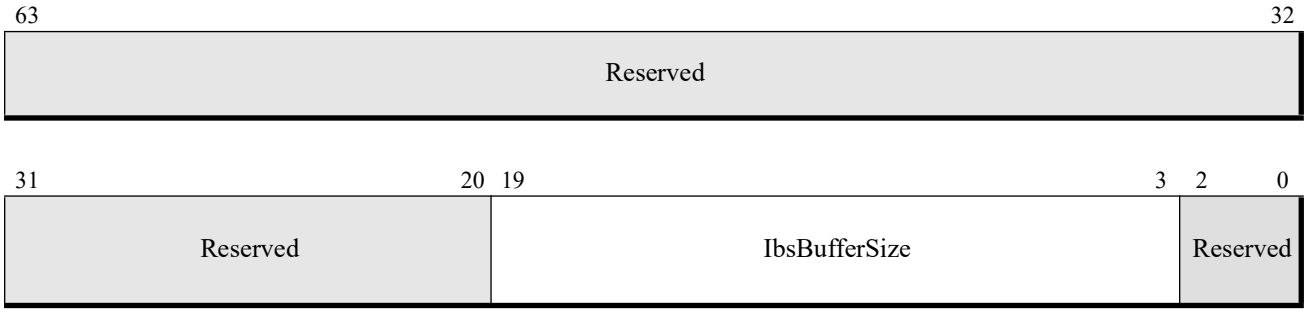
Bits	Mnemonic	Description	Access Type
63:12	IbsBufferLinAddr	IBS Buffer linear starting address	R/W
11:2	Reserved		MBZ
1	TscRecordEn	TSC Record Enable	R/W
0	IbsBufferEn	IBS Buffering Enable	R/W

The fields of IbsBufferBase are further described below:

- *IbsBufferLinAddr (IBS Buffer linear starting address*)—Bits 63:12, read/write. This field contains bits 63:12 of the IBS Buffer linear starting address, The processor assumes that the IBS Buffer is aligned at a 4K boundary and that bits 11:0 of its linear starting address are zero. The address must be canonical relative to the processor's maximum linear address width otherwise the write result in a general-protection exception (#GP).
- *TscRecordEn (TSC Record Enable)*—Bit 1, read/write. When set, the processor writes a TSC (Time Stamp Counter) record into the IBS buffer before it writes a Fetch and/or Execution IBS sample.
- *IbsBufferEn (IBS Buffering Enable)*—Bit 0, read/write. When set, IBS Buffering is enabled. See section "Enabling and Using IBS Buffering" on page 448 for a description of how to configure and use IBS Buffering.

[AMD Public Use]

IBS Buffer Size (IbsBufferSize) (MSR C001_0391h)

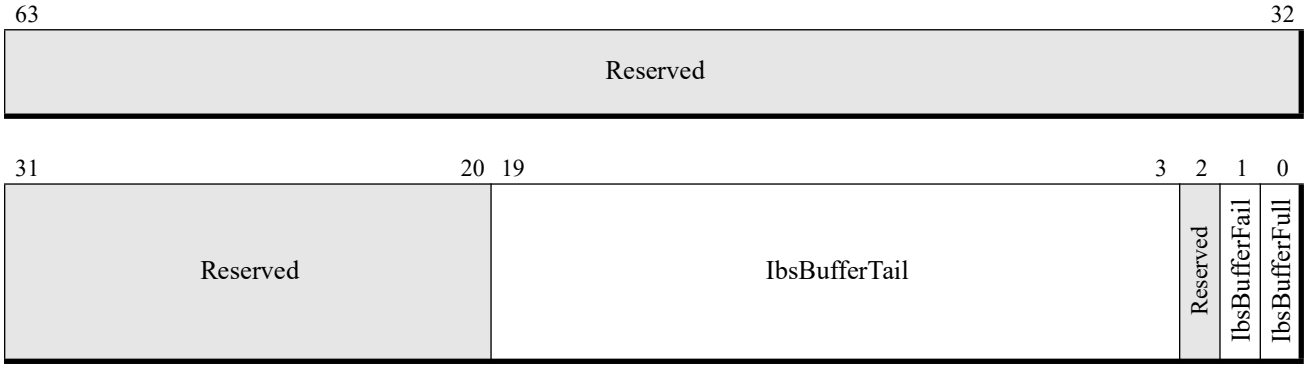


Bits	Mnemonic	Description	Access Type
63:20	Reserved		MBZ
19:3	IbsBufferSize	IBS Buffer Size	R/W
2:0	Reserved		MBZ

The fields of IbsBufferSize are described below.

- *IbsBufferSize (IBS Buffer Size)*—Bits 19:3, read/write. This field specifies the size of the IBS memory buffer in quadwords.

IBS Buffer Tail (IbsBufferTail) (MSR C001_0392h)



Bits	Mnemonic	Description	Access Type
63:20	Reserved		MBZ
19:3	IbsBufferTail	IBS Buffer tail pointer	R/W
2	Reserved		MBZ
1	IbsBufferFail	IBS Buffer write fail indication	R/W
0	IbsBufferFull	IBS Buffer full indication	R/W

The fields of IbsBufferTail are described below:

- *IbsBufferTail (IBS Buffer tail pointer)*—Bits 19:3, read/write. This field returns the current value of the IBS Buffer tail pointer which points to the quadword within the IBS Buffer that would be

[AMD Public Use]

written next by the processor. After processing a full IBS Buffer, software needs to write this field to zero to empty the IBS buffer and restart IBS Buffering.

- *IbsBufferFail (IBS Buffer write fail indication)*—Bit 1. The processor sets this bit when the write of IBS sample data into the IBS Buffer failed. See section “Failed IBS Buffer Writes” for more detail.
- *IbsBufferFull (IBS Buffer full indication)*—Bit 0. The processor sets this bit after it has written one or multiple IBS samples and determined that the remaining IBS buffer capacity (IbsBufferSize[IbsBufferSize] – IbsBufferTail[IbsBufferTail]) is not sufficient for the most IBS sample information that might be written next.

IBS Buffer Records

IBS Buffer records are always multiple of 8 bytes (quadword). The first quadword contains a header which specifies the type of IBS Buffer record and its size in bytes. Subsequent bytes contain IBS sample information as read from the respective IBS registers.

Processors that set CPUID_Fn8000_001B_EAX[IbsBufferV1] (bit 17) = 1 support the following three types of IBS Buffer records shown below. Future versions of IBS buffering may extend these records by adding additional bytes. Software should always use the size read from the IBS Buffer record to advance to the next IBS Buffer entry. When IBS Buffering is supported the IbsBrTarget and IbsFetchCtlExt MSRs are also supported.

Table 13-8. Fetch IBS Record

Offset in bytes	Size	Description
0	byte	Type = 1, Fetch IBS record
1	byte	Size = 40, Fetch IBS record size in bytes
2 .. 7	byte	Reserved
8	quadword	IbsFetchCtl (MSR C001_1030h)
16	quadword	IbsFetchLinAd (MSR C001_1031h)
24	quadword	IbsFetchPhysAd (MSR C001_1032h)
32	quadword	IbsFetchCtlExt (MSR C001_103Ch)

Table 13-9. Execution IBS Record

Offset in bytes	Size	Description
0	byte	Type = 2, Execution IBS record
1	byte	Size = 64, Execution IBS Buffer Record size in bytes
2 .. 7	byte	Reserved
8	quadword	IbsOpRip (MSR C001_1034h)
16	quadword	IbsOpData1 (MSR C001_1035h)

[AMD Public Use]

Offset in bytes	Size	Description
24	quadword	IbsOpData2 (MSR C001_1036h)
32	quadword	IbsOpData3 (MSR C001_1037h)
40	quadword	IbsDcLinAd (MSR C001_1038h)
48	quadword	IbsDcPhysAd (MSR C001_1039h)
56	quadword	IbsBrTarget (MSR C001_103Bh)

Table 13-10. Timestamp Counter (TSC) Record

Offset in bytes	Size	Description
0	byte	Type = 3, TSC record
1	byte	Size = 16, TSC Record size in bytes
2 .. 7	byte	Reserved
8	quadword	Time Stamp Counter (MSR 10h)

13.3.7 IBS Memory Profiler

The IBS Memory Profiler provides software with an IBS facility that can be used to profile memory operations. Support for IBS Memory Profiler Version 1 is indicated by CPUID Fn8000_0001B_EAX[IbsMemProfilerV1] (bit 18) = 1. The Memory Profiler IBS facility utilizes the same programming model as IBS Execution Sampling which is described in Section 13.3.3, “IBS Execution Sampling,” on page 434. The IBS Memory Profiler does not collect information on memory access for instruction fetch. Hardware virtualization and IBS Buffering are not supported for IBS Memory Profiler Version 1.

The following mapping between IBS Execution Sampling and IBS Memory Profiler registers exists:

Table 13-11. IBS Execution Sampling and Memory Profiler Register Equivalents

IBS Execution Sampling Register	IBS Memory Profiler Register	Description
IbsOpCtl	IbsMemCtl	Control register 1
IbsOpCtl2	IbsMemCtl2	Control register 2
IbsOpRip	IbsMemRip	Instruction Linear Address Register
IbsOpData1	–	Not supported for IBS Memory Profiler
IbsOpData2	IbsMemData2	Northbridge and L2 miss related performance data
IbsOpData3	IbsMemData3	Data Cache (first-level) cache performance data
IbsDcLinAd	IbsMemLinAd	Linear address of the data access
IbsDcPhysAd	IbsMemPhysAd	Physical address of the data access
IbsBrTarget	–	Not supported for IBS Memory Profiler

[AMD Public Use]

All IBS Filtering capabilities supported for IBS Execution Sampling are also supported for IBS Memory Profiler. Due to the IBS Memory Profiler's focus on memory operations, it is recommended to use the IBS Memory Profiler with at least one of these IBS Filtering controls:

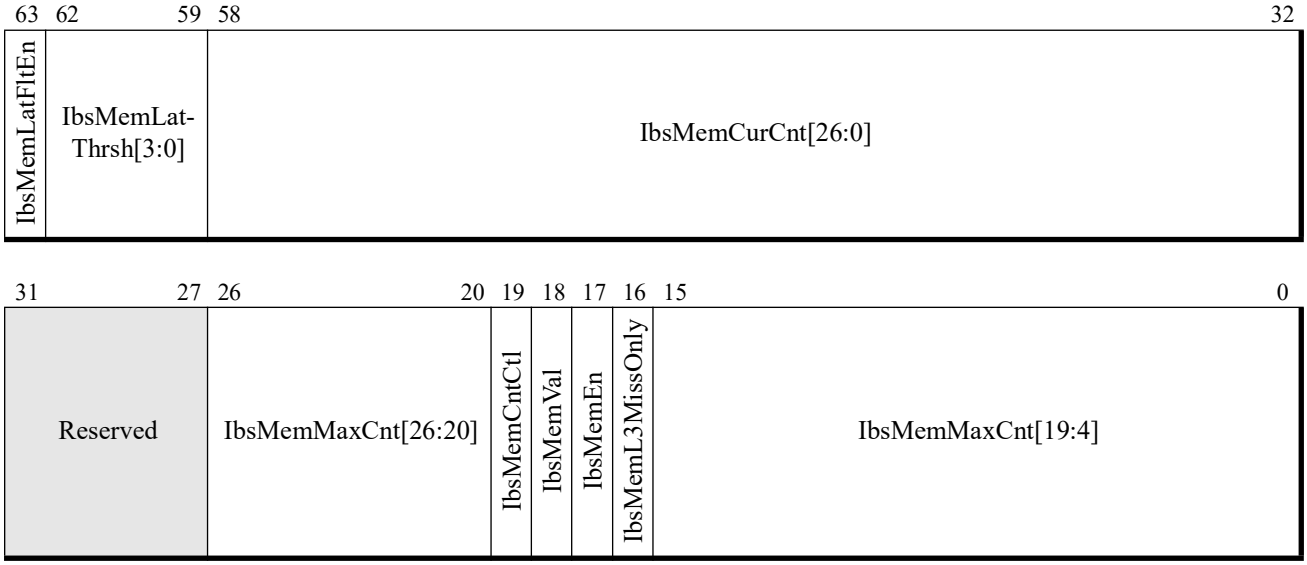
- *IbsMemCtl1[IbsMemL3MissOnly] = 1* – L3 Miss Filtering
- *IbsMemCtl[IbsMemLatFltEn] = 1* – Load Latency Filtering
- *IbsMemCtl2[IbsMemStrmStFilter] = 1* – Streaming Store Filtering

When a sample has been collected and the IBS Memory Profiler sets IbsMemCtl[IbsMemVal] = 1 an interrupt is signaled to the local APIC. The interrupt source differs from the interrupt source used for Fetch and Execution IBS and can be discovered in IbsCtl[IbsMemExtLvtOffset] when IbsCtl[IbsMemExtLvtOffsetVal] = 1.

IBS Memory Profiler Registers

Unless stated otherwise, support for these registers, bits and fields is indicated by CPUID Fn8000_0001B_EAX[IbsMemProfilerV1] (bit 18) = 1.

IBS Memory Profiler Control Register (IbsMemCtl) (MSR C001_0380h)



Bits	Mnemonic	Description	Access Type
63	IbsMemLatFltEn	IBS Mem Load Latency Filter Enable	R/W
62:59	IbsMemLatThrsh[3:0]	IBS Mem Load Latency Threshold	R/W
58:32	IbsMemCurCnt[26:0]	IBS Mem Current Op Count, bits 26:0	R/W
31:27	Reserved		RAZ
26:20	IbsMemMaxCnt[26:20]	IBS Mem Maximum Op Count, bits 26:20	R/W
19	IbsMemCntCtl	IBS Mem Op Counter Control	R/W
18	IbsMemVal	IBS Mem Sample Valid	R/W
17	IbsMemEn	IBS Mem Sampling Enable	R/W
16	IbsMemL3MissOnly	IBS Mem L3 Miss Filtering	R/W

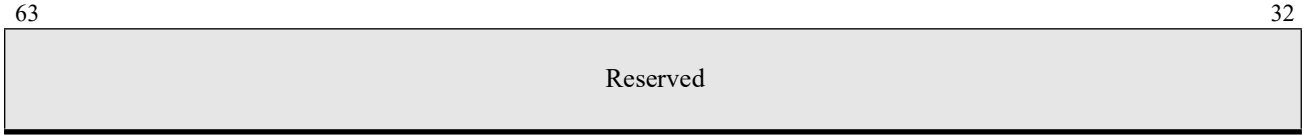
[AMD Public Use]

Bits	Mnemonic	Description	Access Type
15:0	IbsMemMaxCnt[19:4]	IBS Mem Maximum Op Count, bits 19:4	R/W

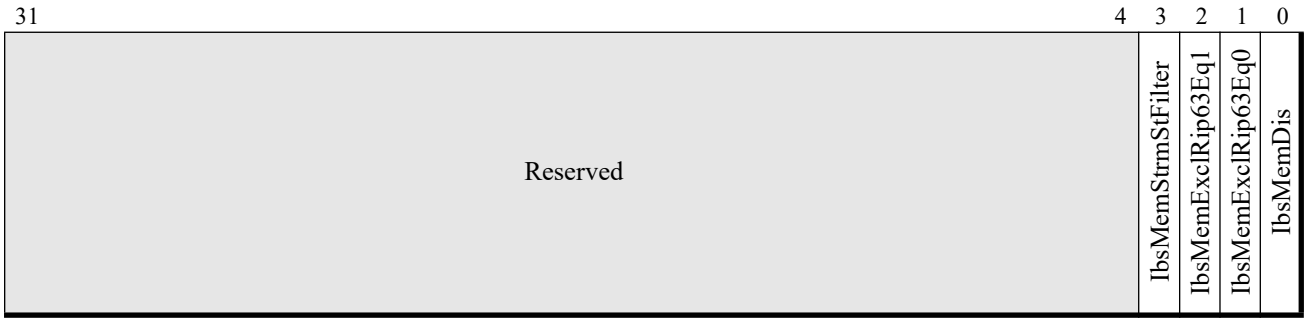
The fields of IbsMemCtl are further described below:

- *IbsMemLatFltEn (IBS Mem Load Latency Filter Enable)* – Bit 63, read/write. When set, Load Latency Filtering for the IBS Memory Profiler is enabled.
- *IbsMemLatThrsh (IBS Mem Load Latency Threshold)* – Bits 62:59, read/write. When IbsMemLatFltEn = 1, this field specifies the load latency threshold. Memory Profiler IBS samples meet the load latency filter criterium if they are a load op (IbsMemData3[IbsMemLdOp] = 1), not a software prefetch op (IbsMemData3[IbsMemSwPf] = 0) and their load latency (IbsMemData3[IbsDcMissLat]) is greater than (IbsMemLatThrsh + 1) * 128. See *Section 13.3.5, "IBS Filtering," on page 444* for IBS Filtering capabilities.
- *IbsMemCurCnt[26:0] (IBS Mem Op Current Count)* – Bits 58:32, read/write. This field returns the current value of the op counter, and provides the starting value of the counter when software writes this register to clear the IbsOpVal bit and start another sampling interval.
- *IbsMemMaxCnt[26:20] (IBS Mem Op Maximum Count[26:20])* – Bits 26:20, read/write. This field is used to specify the most significant 7 bits of IbsMemMaxCnt.
- *IbsMemCntCtl (IBS Mem Op Counter Control)* – Bit 19, read/write. This bit controls op tagging. When this bit is zero, IBS Memory Profiler counts core clock cycles to select an op for tagging. When this bit is one, IBS counts dispatched ops to select an op for tagging.
- *IbsMemVal (IBS Mem Sample Valid)* – Bit 18, read/write. This bit is set when a tagged op retires and passes the enabled IBS filter criteria. It indicates that new IBS Memory Profiler sample data is available. The op counter stops counting. An interrupt is generated as specified by the local APIC. The software interrupt handler captures the performance data before clearing the bit to enable the hardware to take another sample.
- *IbsMemEn (IBS Mem Sample Enable)* – Bit 17, read/write. Software sets this bit to enable IBS Memory Profiler sampling. Clearing this bit disables IBS Memory Profiler sampling.
- *IbsMemL3MissOnly (IBS Mem L3 Miss Filtering)* – Bit 16, read/write. This bit controls L3 Miss Filtering. When set, IBS Memory Profiler only sends interrupts for samples with an L3 miss. See *Section 13.3.5, "IBS Filtering," on page 444* for a description of Execution IBS Filtering which also applies to Memory Profiler IBS Filtering.
- *IbsMemMaxCnt[19:4] (IBS Mem Op Maximum Count[19:4])* – Bits 15:0, read/write. This field specifies the maximum count value for bits 19:4 of the op interval counter. When the value in bits 26:4 of the op interval counter equals the value specified by the concatenation of the IbsOpMaxCnt[26:20] field with this field, the next op is tagged for profiling.

IBS Memory Profiler Control Register 2 (IbsMemCtl2) (MSRC001_0381h)



[AMD Public Use]

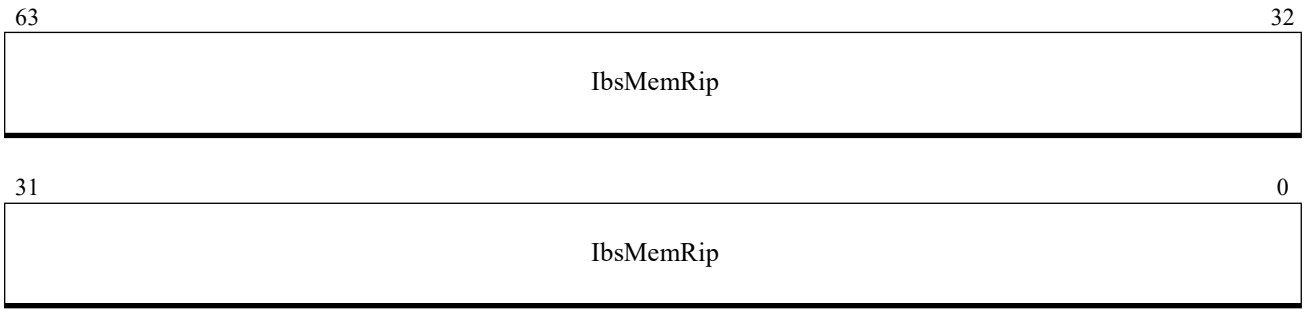


Bits	Mnemonic	Description	Access Type
63:4	Reserved		RAZ
3	IbsMemStrmStFilter	IBS Mem Streaming Store Filter Control	R/W
2	IbsMemExclRip63Eq1	Exclude IBS Mem samples with RIP bit 63 set	R/W
1	IbsMemExclRip63Eq0	Exclude IBS Mem samples with RIP bit 63 cleared	R/W
0	IbsMemDis	IBS Mem Sampling disable	R/W

The fields of IbsMemCtl2 are further described below:

- *IbsMemStrmStFilter (IBS Mem Streaming Store Filter Control)* – Bit 3, read/write. This bit controls Memory Profiler IBS Filtering. When set, samples for streaming stores (IbsMemData2[StrmSt] = 1) are reported. For more detail on IBS Filtering see *Section 13.3.5, "IBS Filtering," on page 444*.
- *IbsMemExclRip63Eq1 (Exclude IBS Mem samples with RIP bit 63 set)* – Bit 2, read/write. This bit controls Memory Profiler IBS Filtering. When set, Memory Profiler IBS excludes samples when the logical instruction address (IbsMemRip) has bit 63 set.
- *IbsMemExclRip63Eq0 (Exclude IBS Mem samples with RIP bit 63 cleared)* – Bit 1, read/write. This bit controls Memory Profiler IBS Filtering. When set, Memory Profiler IBS excludes samples when the logical instruction address (IbsMemRip) has bit 63 cleared.
- *IbsMemDis (IBS Mem Sampling Disable)* – Bit 0, read/write. When set, Memory Profiler IBS is disabled. Memory Profiler IBS is only enabled if IbsMemDis = 0, IbsMemCtl[IbsMemEn] = 1, and IbsMemCtl[IbsMemVal] = 0.

IBS Memory Profiler RIP Register (IbsMemRip) (MSR C001_0386h)



[AMD Public Use]

Bits	Mnemonic	Description	Access Type
63:0	IbsMemRip	IBS Mem Op Linear Address	R/W

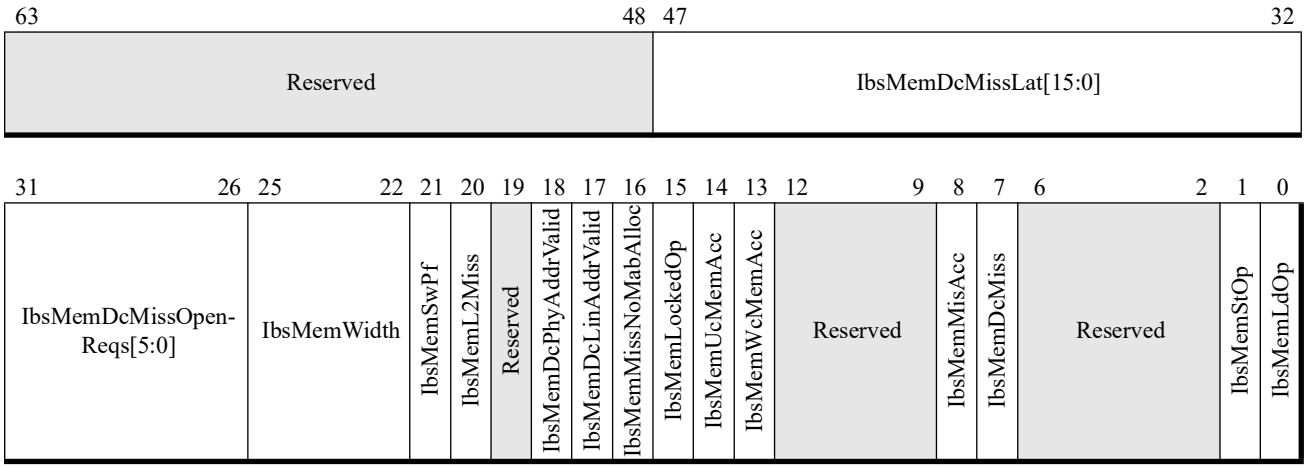
The fields of IbsMemRip are further described below:

- *IbsMemRip[63:0] (IBS Mem Op Linear Address)* – Bits 63:0, read/write. Specifies the linear address for the instruction from which the tagged op was issued. The address is valid only if the IbsMemCtl[IbsMemVal] bit is set. The address is in canonical form.

IBS Memory Profiler Data 2 Register (IbsMemData2) (MSR C001_0382h)

The IbsMemData2 Register captures Northbridge and system related performance data. The information captured is implementation-dependent. See the *Processor Programming Reference Manual (PPR)* applicable to your product for details.

IBS Memory Profiler Data 3 Register (IbsMemData3) (MSR C001_0383h)



Bits	Mnemonic	Description	Access Type
63:48	Reserved		RAZ
47:32	IbsMemDcMissLat[15:0]	IBS Mem Data Cache Miss Latency	R/W
31:26	IbsMemDcMissOpenReqs[5:0]	IBS Mem Outstanding Memory Requests on DC Fill	R/W
25:22	IbsMemWidth	IBS Mem Access Size	R/W
21	IbsMemSwPf	IBS Mem Software Prefetch	R/W
20	IbsMemL2Miss	IBS Mem L2 Cache Miss	R/W
19	Reserved		RAZ
18	IbsMemDcPhyAddrValid	IBS Mem Data Cache Physical Address Valid	R/W
17	IbsMemDcLinAddrValid	IBS Mem Data Cache Linear Address Valid	R/W
16	IbsMemMissNoMabAlloc	IBS Mem No Miss Buffer Allocation	R/W
15	IbsMemLockedOp	IBS Mem Locked Op	R/W
14	IbsMemUcMemAcc	IBS Mem UC Memory Access	R/W
13	IbsMemWcMemAcc	IBS Mem WC Memory Access	R/W
12:9	Reserved		RAZ

[AMD Public Use]

Bits	Mnemonic	Description	Access Type
8	IbsMemMisAcc	IBS Mem Misaligned Access Penalty	R/W
7	IbsMemDcMiss	IBS Mem Data Cache Miss	R/W
6:2	Reserved		RAZ
1	IbsMemStOp	IBS Mem Store Op	R/W
0	IbsMemLdOp	IBS Mem Load Op	R/W

The fields of IbsMemData3 are further described below:

- *IbsMemDcMissLat[15:0] (IBS Mem Data Cache Miss Latency)* - Bits 47:32, read/write. This field indicates the number of core clock cycles from when a miss is detected in the data cache to when the data is delivered to the core. The value is not valid for data cache store operations.
- *IbsMemDcMissOpenReqs[5:0] (IBS Mem Outstanding Memory Requests on DC Fill)* – Bits 31:26, read/write. The number of outstanding data cache miss requests when the data cache fill data for the tagged data cache miss op is received. The count Includes the fill request by the sampled op. A value of 0 means that no information is provided.
- *IbsMemWidth[3:0] (IBS Mem Access Size)* – Bits 25:22, read/write. The encoded number of bytes the tagged load or store is accessing.

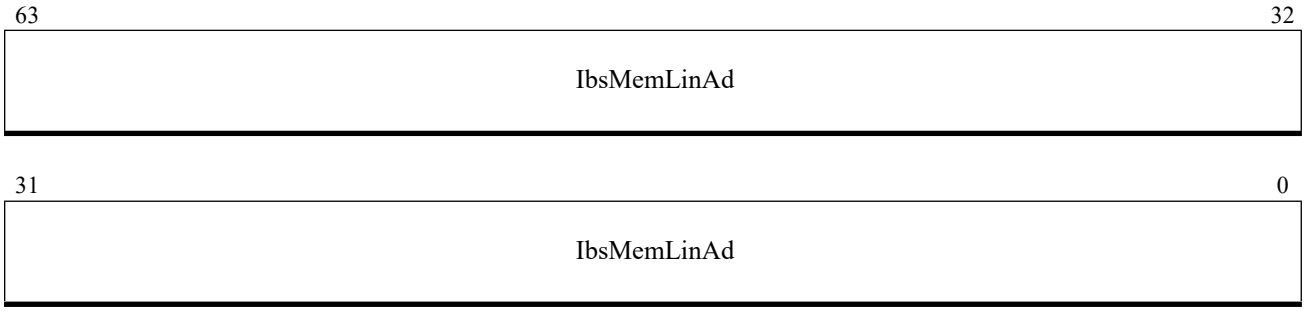
Value	Memory Access Size
0h	No information provided
1h	byte
2h	word
3h	doubleword (4 bytes)
4h	quadword (8 bytes)
5h	16 bytes
6h	32 bytes
7h	64 bytes
8h-Fh	Reserved

- *IbsMemSwPf (IBS Mem Software Prefetch)* – Bit 21, read/write. This bit is set when the tagged op belongs to a data prefetch instruction.
- *IbsMemL2Miss (IBS Mem L2 Cache Miss)* – Bit 20, read/write. This bit is set if the cache line used by the tagged load or store operation was not present in the L2 cache.
- *IbsMemDcPhyAddrValid (IBS Mem Data Cache Physical Address Valid)* – Bit 18, read/write. This bit is set if the physical address in the IBS Memory Profiler DC Physical Address Register (IbsMemPhysAd) is valid for a load or store operation.
- *IbsMemDcLinAddrValid (IBS Mem Data Cache Linear Address Valid)* – Bit 17, read/write. This bit is set if the physical address in the IBS Memory Profiler DC Linear Address Register (IbsMemLinAd) is valid for a load or store operation.

[AMD Public Use]

- *IbsMemMissNoMabAlloc (IBS Mem No Miss Buffer Allocation)* – Bit 16, read/write. This bit is set if the tagged load or store operation matched an already outstanding cache miss and no new miss buffer entry was allocated.
- *IbsMemLockedOp (IBS Mem Locked Op)* – Bit 15, read/write. This bit is set if the tagged load or store operation was a locked operation.
- *IbsMemUcMemAcc (IBS Mem UC Memory Access)* – Bit 14, read/write. This bit is set if the tagged load or store operation accessed uncacheable memory.
- *IbsMemWcMemAcc (IBS Mem Cache WC Memory Access)* – Bit 13, read/write. This bit is set if the tagged load or store operation accessed Write-Combining memory.
- *IbsMemMisAcc (IBS Memory Misaligned Access Penalty)* – Bit 8, read/write. This bit is set if a tagged load or store operation incurred a performance penalty due to misaligned access.
- *IbsMemDcMiss (IBS Data Cache Miss)* – Bit 7, read/write. This bit is set if the cache line used by the tagged load or store operation was not present in the data cache.
- *IbsMemStOp (IBS Mem Store Op)* – Bit 1, read/write. This bit is set if the tagged op was a store.
- *IbsMemLdOp (IBS Mem Load Op)* – Bit 0, read/write. This bit is set if the tagged op was a load.

IBS Memory Profiler DC Linear Address Register (IbsMemLinAd) (MSR C001_0384h)

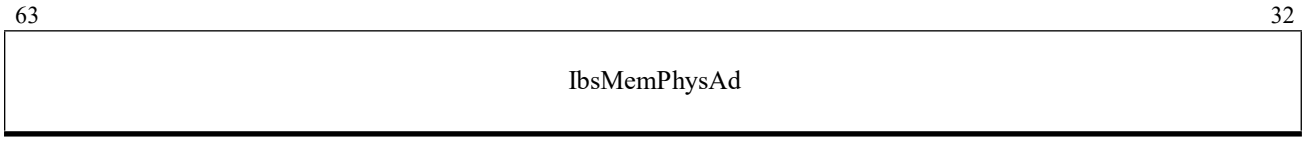


Bits	Mnemonic	Description	Access Type
63:0	IbsMemLinAd	IBS Mem Linear Address	R/W

The fields of IbsMemLinAd are further described below:

- *IbsMemLinAd[63:0] (IBS Mem Linear Address)* – Bits 63:0, read/write. Specifies the linear address of the tagged op's memory operand. The address is valid only if IbsMemData3[IbsMemDcLinAddrValid] is set. The address is in canonical form.

IBS Memory Profiler DC Physical Address Register (IbsMemPhysAd) (MSR C001_0385h)



[AMD Public Use]

31	0
IbsMemPhysAd	

Bits	Mnemonic	Description	Access Type
63:0	IbsMemPhysAd	IBS Mem Physical Address	R/W

The fields of IbsMemPhysAd are further described below:

- *IbsMemPhysAd[63:0] (IBS Mem Physical Address)* – Bits 63:0, read/write. Specifies the physical address of the tagged op's memory operand. The address is valid only if IbsMemData3[IbsMemPhysAdVal] is set.

13.4 Lightweight Profiling

Lightweight Profiling (LWP) is an AMD64 extension that allows user mode processes to gather performance data about themselves with very low overhead. LWP is supported in both long mode and legacy mode. Modules such as managed runtime environments and dynamic optimizers can use LWP to monitor the running program with high accuracy and high resolution. They can quickly discover performance problems and opportunities and immediately act on this information.

LWP allows a program to gather performance data and examine it either by polling or by taking an occasional interrupt. It introduces minimal additional state to the CPU and the process. LWP differs from the existing performance counters and from Instruction Based Sampling (IBS) because it collects large quantities of data before taking an interrupt. This substantially reduces the overhead of using performance feedback. An application can avoid the need to enable and process interrupts by polling the LWP data.

A program can control LWP data collection entirely in user mode. It can start, stop, and reconfigure profiling without calling the kernel.

LWP runs within the context of a thread, so it can be used by multiple processes in a system at the same time without interference. This also means that if one thread is using LWP and another is not, the latter thread incurs no profiling overhead.

LWP can be programmed to run in one of two modes: *synchronized mode* or *continuous mode*. In synchronized mode the recording of events stops when the buffer set up to hold event records becomes full. In continuous mode, the storing of events wraps in the buffer overwriting older records.

13.4.1 Overview

When enabled, LWP hardware monitors one or more events during the execution of user-mode code and periodically inserts event records into a ring buffer in the address space of the running process. If performance timestamping is supported and enabled, each event record captured is timestamped using the value read from the Performance Timestamp Counter (PTSC). Timestamping is enabled by setting

[AMD Public Use]

the Flags.PTSC bit of the Lightweight Profiling Control Block (LWPCB). When the ring buffer is filled beyond a user-specified threshold, the hardware can cause an interrupt which the operating system (OS) uses to signal a process to empty the ring buffer. With proper OS support, the interrupt can even be delivered to a separate process or thread.

LWP only counts instructions that retire in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. LWP does not count hardware events while the processor is in system management mode (SMM) and while entering or leaving SMM.

Once LWP is enabled, each user-mode thread uses the LLWPCB and SLWPCB instructions to control LWP operation. These instructions refer to a data structure in application memory called the Lightweight Profiling Control Block, or LWPCB, to specify the profiling parameters and to interact with the LWP hardware. The LWPCB in turn points to a buffer in memory in which LWP stores event records.

Each thread in a multi-threaded process must configure LWP separately. A thread has its own ring buffer and counters which are context switched with the rest of the thread state. However, a single monitor thread could collect and process LWP data from multiple other threads.

LWP may be set up to run in one of two modes:

- Synchronized Mode

  LWP runs in synchronized mode when it is started with LWPCB.Flags.CONT = 0. In this mode, LWP will not advance the ring buffer pointer when the event ring buffer is full. It simply increments LWPCB.MissedEvents to count the number of missed event records. In synchronized mode, a thread can remove event records from the ring buffer by advancing the ring buffer tail pointer without stopping LWP in the executing thread. If the buffer had been full, event records will again be written and the ring buffer pointer will be advanced.
- Continuous Mode

  LWP runs in continuous mode when it is started with LWPCB.Flags.CONT = 1. In this mode, LWP will store an event record even when the event ring buffer is full, wrapping around in the ring buffer and overwriting the oldest event record. In continuous mode, LWPCB.MissedEvents counts the number of times that such wrapping has occurred. The only reliable way to read events from the ring buffer when LWP is in continuous mode is to stop LWP in the running thread before accessing the LWPCB and the ring buffer contents. Support for continuous mode is indicated by CPUID Fn8000_001C_EAX[LwpCont].

During profiling, the LWP hardware monitors and reports on one or more types of events. Following are the steps in this process:

1. **Count—**Each time an instruction is retired, LWP decrements its internal event counters for all of the events associated with the instruction. An instruction can cause zero, one, or multiple events. For instance, an indirect jump through a pointer in memory counts as an instruction retired, a branch retired, and may also cause up to two DCache misses (or more, if there is a TLB miss) and up to two ICache misses.

[AMD Public Use]

- Some events may have filters or conditions on them that regulate counting. For instance, the application may configure LWP so that only cache miss events with latency greater than a specified minimum are eligible to be counted.

2. **Gather—**When an event counter becomes negative, the event should be reported. LWP gathers an event record and, if enabled, samples the value in the PTSC to be included in the record as the TimeStamp value. The event's counter may continue to count below zero until the record is written to the event ring buffer.

   For most events, such as instructions retired, LWP gathers an event record describing the instruction that caused the counter to become negative. However, it is valid for LWP to gather event record data for the *next* instruction that causes the event, or to take other measures to capture a record. Some of these options are described with the individual events.

   - An implementation can choose to gather event information on one or many events at any one time. If multiple event counters become negative, an advanced LWP implementation might gather one event record per event and write them sequentially. A basic LWP implementation may choose one of the eligible events. Other events continue counting but wait until the first event record is written. LWP picks the next eligible instructions for the waiting events. This situation should be extremely uncommon if software chooses large event interval values.
   - LWP may discard an event occurrence. For instance, if the LWPCB or the event ring buffer needs to be paged in from disk, LWP might choose not to preserve the pending event data. If an event is discarded, LWP gathers an event record for the next instruction to cause the event.
   - Similarly, if LWP needs to replay an instruction to gather a complete event record, the replay may abort instead of retiring. The event counter continues counting below zero and LWP gathers an event record for the next instruction to cause the event.

3. **Store—**When a complete event record is gathered, LWP stores it into the event ring buffer in the process' address space and advances the ring buffer head pointer.

   - LWP checks to see if the ring buffer is full, i.e., if advancing the ring buffer head pointer would make it equal to the tail pointer. If the buffer is full, LWP increments the 64-bit counter LWPCB.MissedEvents. If LWP is running in synchronized mode, it does not advance the head pointer. If LWP is running in continuous mode, it always advances the head pointer and LWPCB.MissedEvents counts the number of times that the buffer wrapped.
   - If more than one event record reaches the Store stage simultaneously, only one need be stored. Though LWP might store all such event records, it may delay storing some event records or it may discard the information and proceed to choose the next eligible instruction for the discarded event type(s). This behavior is implementation dependent.
   - The store need not complete synchronously with the instruction retiring. In other words, if LWP buffers the event record contents, the Store stage (and subsequent stages) may complete some number of cycles after the tagged instruction retires. The data about the event and the instruction are precise, but the Report and Reset steps (below) may complete later.

[AMD Public Use]

4. **Report—**If LWP threshold interrupts are enabled and the space used in the event ring buffer exceeds a user-defined threshold, LWP initiates an interrupt. The OS can use this to signal the process to empty the ring buffer. Note that the interrupt may occur significantly later than the event that caused the threshold to be reached.
5. **Reset—**For each event that was stored, the counter is reset to its programmed interval. If requested by the application, LWP applies randomization to the low order bits of the interval. Counting for that event continues. Reset happens if the ring buffer head pointer was advanced or if the missed event counter was incremented. If the event counter went below -1, indicating that additional events occurred between the selected event and the time it was reported, that overrun value reduces the reset value so as to preserve the statistical distribution of events.

   For all events except the LWPVAL instruction, the hardware may impose a minimum on the reset value of an event counter. This prevents the system from spending too much time storing samples rather than making forward progress on the application. Any minimum imposed by the hardware can be detected by examining the EventInterval*n* fields in the LWPCB after enabling LWP.

An application should periodically remove event records from the ring buffer and advance the tail pointer. (If the application does not process the event records quickly enough or often enough, the LWP hardware will detect that the ring buffer is full and will miss events.) There are two ways to process the gathered events: interrupts or polling.

The application can wait until a threshold interrupt occurs to process the event records in the ring buffer. This requires OS or driver support. (As a consequence, interrupts can only be enabled if a kernel mode routine allows it; see “LWP_CFG—LWP Configuration MSR” on page 477) One usage model is to associate the LWP interrupt with a semaphore or mutex. When the interrupt occurs, the OS or driver signals the associated object. A thread waiting on the object wakes up and empties the ring buffer. Other models are possible, of course.

Alternatively, the application can have a thread that periodically polls the ring buffer. The polling thread need not be part of the process that is using LWP. It can be in a separate process that shares the memory containing the LWP control block and ring buffer.

Access to the ring buffer uses a lockless protocol between the LWP hardware and the application. The hardware owns the head pointer and the area in the ring buffer from the head pointer up to (but not including) the tail pointer. The application must not modify the head pointer nor rely on any data in the area of the ring buffer owned by the hardware. If the application has a stale value for the head pointer, it may miss an existing event record but it will never read invalid data. When the application is done emptying the ring buffer, it should refresh its copy of the head pointer to see if the LWP hardware has added any new event records.

Similarly, the application owns the tail pointer and the area in the ring buffer from the tail pointer up to (but not including) the head pointer. The hardware will never modify the tail pointer or overwrite the data in that region of the ring buffer. If the hardware has a stale value for the tail pointer, it may consider that the ring buffer is full or at its threshold, but it will never overwrite valid data. Instead, it refreshes its copy of the tail pointer and rechecks to see if the full or threshold condition still applies.

[AMD Public Use]

When LWP is in continuous mode, this lockless protocol does not work, since the LWP hardware may overwrite the event records in the ring buffer when it advances the head pointer past the tail pointer. Because of this, the application must stop LWP before removing event records from the ring buffer. This prevents the hardware from wrapping through the ring buffer asynchronously from the application's attempt to remove data from it.

To use continuous mode properly, the application should set LWPCB.MissedEvents to 0 and set the head and tail pointers to the start of the ring buffer before starting LWP. To empty the ring buffer, the application should stop LWP. If LWPCB.MissedEvents is 0, the buffer did not wrap and there are event records starting at the tail pointer and continuing up to (but not including) the head pointer. If MissedEvents is not 0, the buffer wrapped and there are event records starting with the oldest one pointed to by the head pointer and continuing (possibly wrapping) all the way around to the newest one just before the head pointer.

13.4.2 Events and Event Records

When a monitored event overflows its event counter, LWP puts an event record into the LWP event ring buffer. If event timestamping is supported and enabled, each event record will include a TimeStamp value. This value is a copy of the contents of Performance Timestamp Counter (PTSC) zero-extended if necessary to 64 bits.

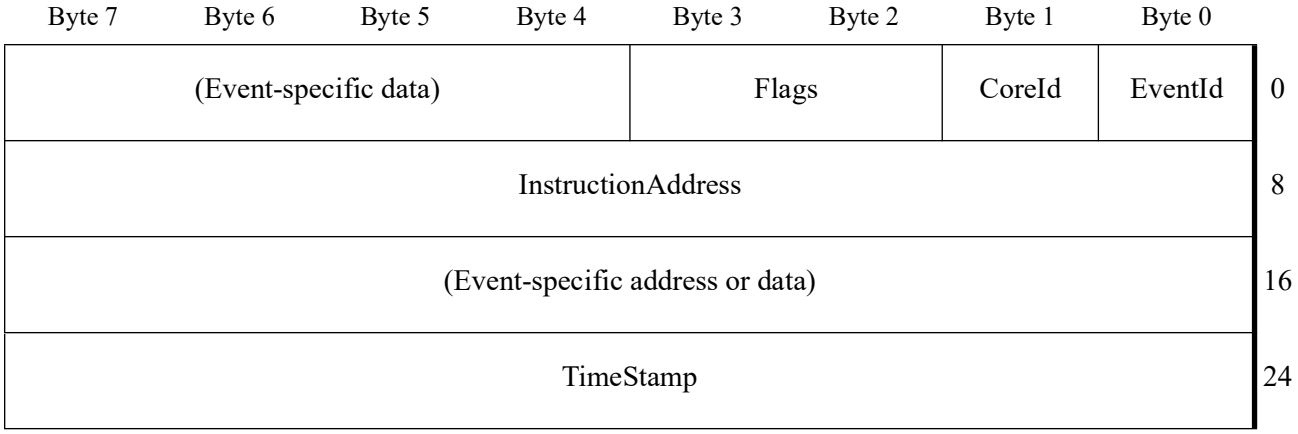
The PTSC is a free-running counter that increments at a constant rate of 100MHz and is synchronized across all cores on a node to within +/-1. This counter starts when the processor is initialized and cannot be reset or modified. It is at least 40 bits wide. Privileged code can read the PTSC value via the RDMSR instruction. The size of the counter is indicated by the 2-bit field CPUID Fn8000_0008_ECX[PerfTscSize]. A value of 00b means that the PTSC is 40 bits wide; 01b means 48 bits, 10b means 56 bits, and 11b indicates a full 64 bits.

The PTSC can be correlated to the architectural TSC that runs at the P0 frequency. An application can read the TSC and PTSC, wait a 1000 clock periods or so, then read them again. The ratio of the differences is the scaling factor for the counters.

The event record size is fixed but may vary based on implementation. The event record size for a given processor is discovered by executing CPUID Fn8000_001C and extracting the value of the LwpEventSize field. (See "Detecting LWP Capabilities" on page 474). Current implementations fix the record size at 32 bytes and this size is used in the record format specifications below.

Reserved fields and fields that are not defined for a particular event are set to zero when LWP writes an event record.

[AMD Public Use]



Bytes	Field	Description
0	EventId	Event identifier specifying the event record type. Valid identifiers are 1 to 255. 0 is an invalid identifier.
1	CoreId	CPU core identifier value from COREID field of LWP_CFG (see "LWP_CFG—LWP Configuration MSR" on page 477). For multicore systems, this typically identifies the core on which LWP is running. This allows software to aggregate event records from multiple threads into a single data structure without losing CPU information. It also allows software to detect when a thread has migrated from one core to another.
3–2	Flags	Event-specific flags.
7–4		Event-specific data.
15–8	InstructionAddress	The Effective Address of the instruction that triggered this event record. This is the value before adding in the CS base address. If the base is non-zero, software must track it. (Modern operating systems use a CS base of zero, and CS is unused in long mode.)
23–16		Event-specific address or other data.
31–24	TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-33. Generic Event Record

Table 13-12 below lists the event identifiers for the events defined in version 1 of LWP. They are described in detail in the following sections.

Table 13-12. EventId Values

EventId	Description
0	Reserved – invalid event
1	Programmed value sample
2	Instructions retired
3	Branches retired
4	DCache misses

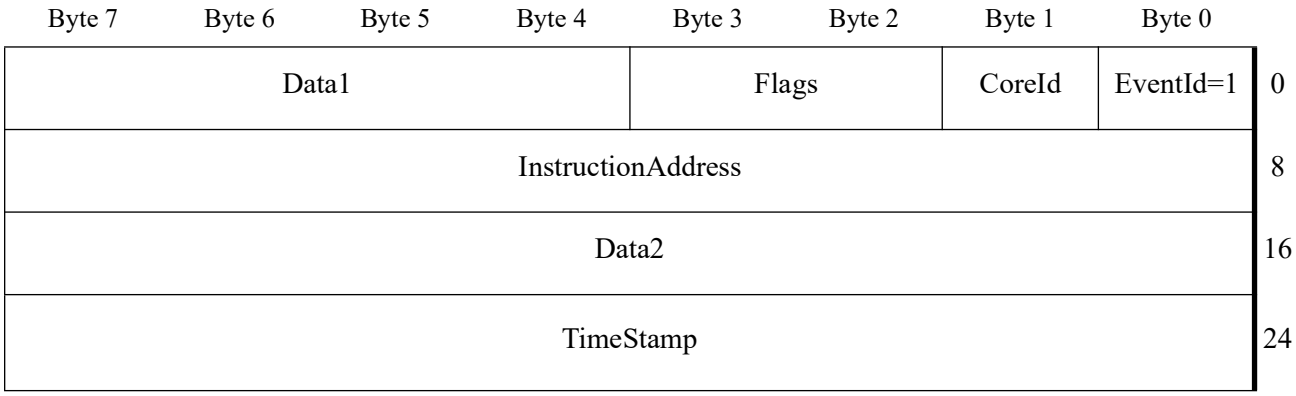
[AMD Public Use]

Table 13-12. EventId Values (continued)

EventId	Description
5	CPU clocks not halted
6	CPU reference clocks not halted
255	Programmed event

13.4.2.1 Programmed Value Sample

LWP decrements the event counter each time the program executes the LWPVAL instruction (see “LWPVAL—Insert Value Sample in LWP Ring Buffer” on page 481). When the counter becomes negative, it stores an event record with an EventId of 1. The data in the event record come from the operands to the instruction as detailed in the instruction description.



Bytes	Field	Description
0	EventId	Event identifier = 1
1	CoreId	CPU core identifier from LWP_CFG
3–2	Flags	Immediate value (bottom 16 bits)
7–4	Data1	Reg/mem value
15–8	InstructionAddress	Instruction address of LWPVAL instruction
23–16	Data2	Reg value (zero extended if running in legacy mode)
31–24	TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

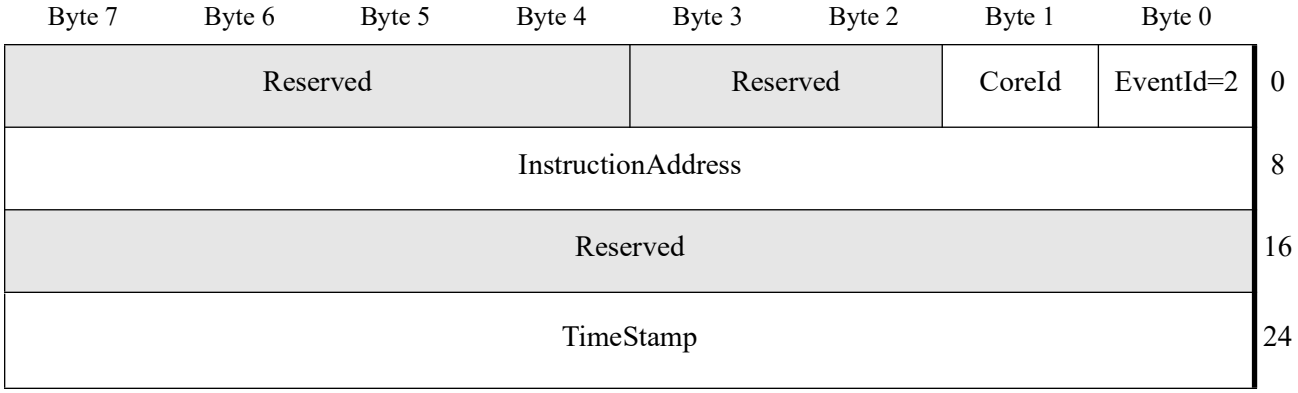
Figure 13-34. Programmed Value Sample Event Record

13.4.2.2 Instructions Retired

LWP decrements the event counter each time an instruction retires. When the counter becomes negative, it stores a generic event record with an EventId of 2.

Instructions are counted if they execute entirely in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. All user mode instructions are counted, including LWPVAL and LWPINS.

[AMD Public Use]



Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 2
1	7:0	CoreId	CPU identifier from LWP_CFG
7–2			Reserved
15–8		InstructionAddress	Instruction address
23–16			Reserved
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-35. Instructions Retired Event Record

13.4.2.3 Branches Retired

LWP decrements the event counter each time a transfer of control retires, regardless of whether or not it is taken. When the counter becomes negative, it stores an event record with an EventId of 3.

Control transfer instructions that are counted are:

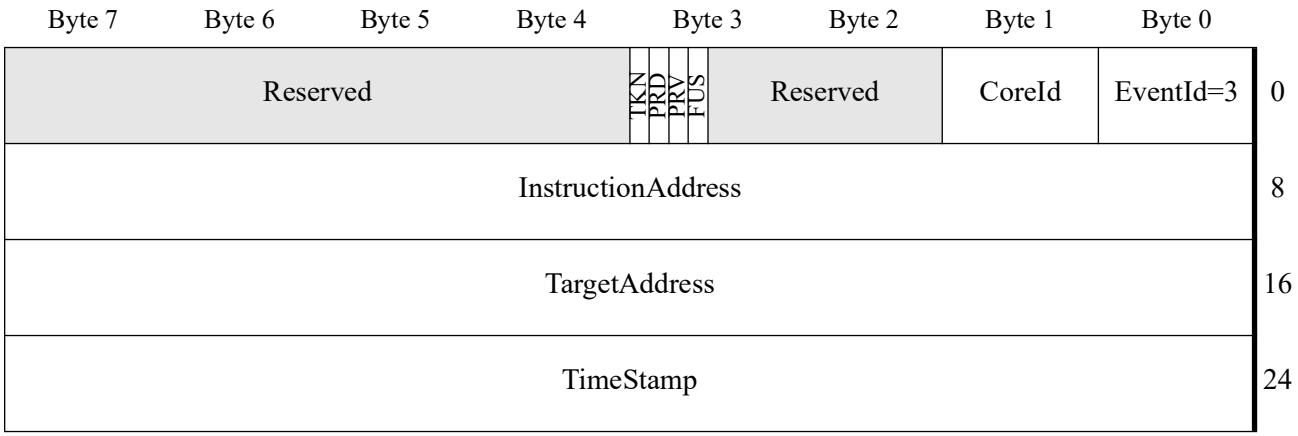
- JMP (near), Jcc, JCXZ, JEXCZ, JRCXZ, and JrCXZ
- LOOP, LOOPE, LOOPNE, and LPOOPcc
- CALL (near) and RET (near)

LWP does not count JMP (far), CALL (far), RET (far), traps, or interrupts (whether synchronous or asynchronous), nor does it count operations that switch to or from ring 3, SMM, or SVM, such as SYSCALL, SYSENTER, SYSEXIT, SYSRET, VMMCALL, INT, or INTO.

Some implementations of the AMD64 architecture perform an optimization called “fusing” when a compare operation (or other operation that sets the condition codes) is followed immediately by a conditional branch. The processor fuses these into a single operation internally before they are executed. While this is invisible to the programmer, the address of the actual branch is not available for LWP to report when the (fused) instruction retires. In this case, LWP sets the FUS bit in the event record and reports the address of the operation that set the condition codes. If FUS is set, software can find the address of the actual branch by decoding the instruction at the reported InstructionAddress and

[AMD Public Use]

adding its length to that address. (Note that fused instructions do count as 2 instructions for the Instructions Retired event, since there were 2 x86 instructions originally.)



Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 3
1	7:0	CoreId	CPU core identifier from LWP_CFG
3–2	11:0		Reserved
3	4	FUS	1—Fused operation. InstructionAddress points to a compare operation (or other operation that sets the condition codes) immediately preceding the branch. 0—InstructionAddress points to the branch instruction.
3	5	PRV	1—PRD bit is valid 0—Prediction information is not available Some implementations of LWP may be unable to capture branch prediction information on some or all branches.
3	6	PRD	1—Branch was predicted correctly 0—Mispredicted If PRV = 0, the value of PRD is unpredictable and should be ignored. For unconditional branches, PRD=1 if PRV=1.
3	7	TKN	1—Branch was taken 0—Branch not taken Always 1 for unconditional branches.
7–4			Reserved
15–8		InstructionAddress	Instruction address
23–16		TargetAddress	Address of instruction executed after branch. This is the target if the branch was taken and the fall-through address if the branch was a not-taken conditional branch. TargetAddress is the Effective Address value before adding in the CS base address.
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-36. Branch Retired Event Record

[AMD Public Use]

13.4.2.4 DCache Misses

LWP decrements the event counter each time a load from memory causes a DCache miss whose latency exceeds the LwpCacheLatency threshold and/or whose data come from a level of the cache or memory hierarchy that is selected for counting. When the counter becomes negative, LWP stores an event record with an EventId of 4.

A misaligned access that causes two misses on a single load decrements the event counter by 1 and, if it reports an event, the data are for the lowest address that missed. LWP only counts loads directly caused by the instruction. It does not count cache misses that are indirectly due to TLB walks, LDT or GDT references, TLB misses, etc. Cache misses caused by LWP itself accessing the LWPCB or the event ring buffer are not counted.

Measuring Latency

The x86 architecture allows multiple loads to be outstanding simultaneously. An implementation of LWP might not have a full latency counter for every load that is waiting for a cache miss to be resolved. Therefore, an implementation may apply any of the following simplifications. Software using LWP should be prepared for this.

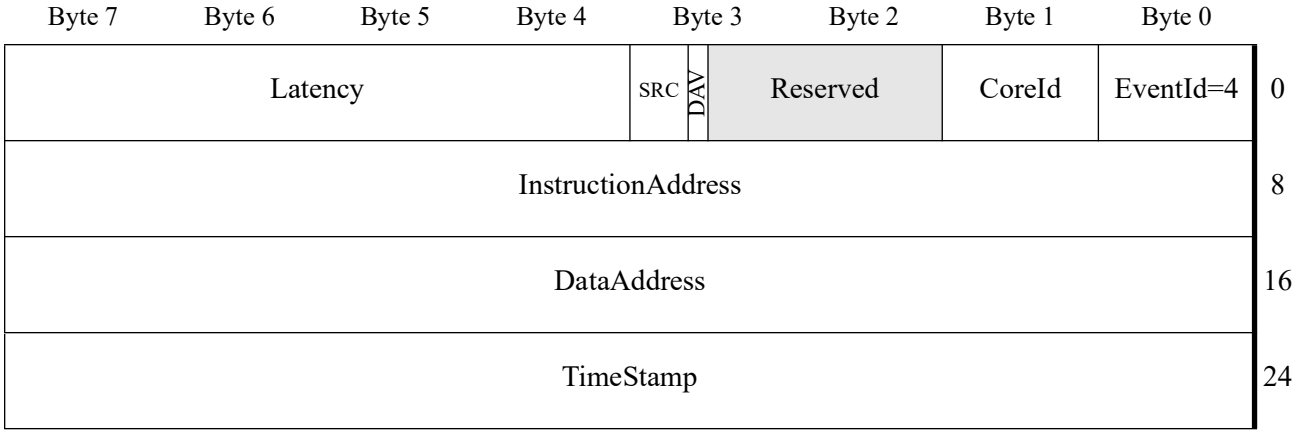
- The implementation may round the latency to a multiple of 2^*j*. This is a small power of 2, and the value of *j* must be 1 to 4. For example, in the rest of this section, assume that *j* = 4, so 2^*j* = 16. The low 4 bits of latency reported in the event record will be 0. The actual latency counter is incremented by 16 every 16 cycles of waiting. The value of *j* is returned as LwpLatencyRnd (see “Detecting LWP Capabilities” on page 474).
- The implementation may do an approximation when starting to count latency. If counting is in increments of 16, the 16 cycles need not start when the load begins to wait. The implementation may bump the latency value from 0 to 16 any time during the first 16 cycles of waiting.
- The implementation may cap total latency to 2^*n*-16 (where *n* >= 10). The latency counter is thus a saturating counter that stops counting when it reaches its maximum value. For example, if *n* = 10, the latency value will count from 0 to 1008 in steps of 16 and then stop at 1008. (If *n* = 10, each counter is only 6 bits wide.) The value of *n* is returned as LwpLatencyMax (see “Detecting LWP Capabilities” on page 474).

Note that the latency threshold used to filter events is a multiple of 16. This value is used in the comparison that decides whether a cache miss event is eligible to be counted.

Reporting the DCache Miss Data Address

The event record for a DCache miss reports the linear address of the data (after adding in the segment base address, if any). The way an implementation records the linear address affects the exact event that is reported and the amount of time it takes to report a cache miss event. The implementation may report the event immediately, report the next eligible event once the counter becomes negative, or replay the instruction.

[AMD Public Use]



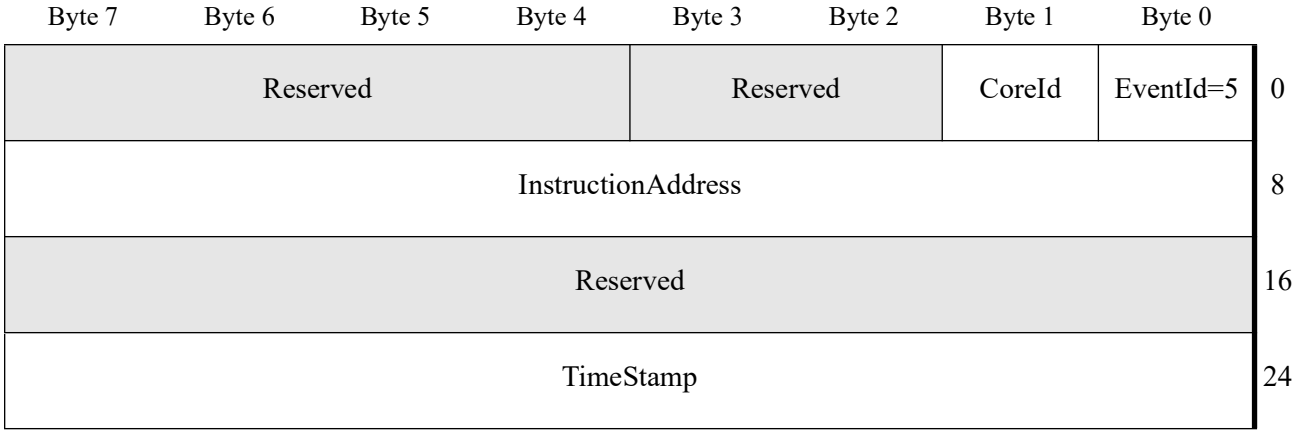
Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 4
1	7:0	CoreId	CPU identifier from LWP_CFG
2–3	11:0		Reserved
3	4	DAV	1—DataAddress is valid 0—Address is unavailable
3	5:7	SRC	Data source for the requested data 0 No valid status 1 Local L3 cache 2 Remote CPU or L3 cache 3 DRAM 4 Reserved (for Remote cache) 5 Reserved 6 Reserved 7 Other (MMIO/Config/PCI/APIC)
7–4		Latency	Total latency of cache miss (in cycles)
15–8		InstructionAddress	Instruction address
23–16		DataAddress	Address of memory reference (if flag bit 28 = 1)
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-37. DCache Miss Event Record

13.4.2.5 CPU Clocks not Halted

LWP decrements the event counter each clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter becomes negative, it stores a generic event record with an EventId of 5. This counter varies in real-time frequency as the core clock frequency changes.

[AMD Public Use]



Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 5
1	7:0	CoreId	CPU identifier from LWP_CFG
7–2			Reserved
15–8		InstructionAddress	Instruction address
23–16			Reserved
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

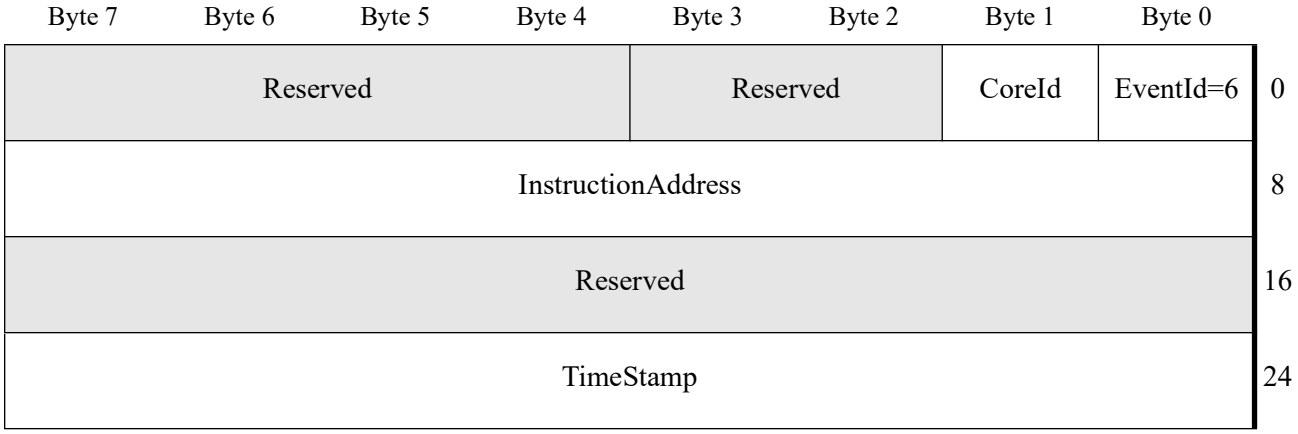
Figure 13-38. CPU Clocks not Halted Event Record

13.4.2.6 CPU Reference Clocks not Halted

LWP decrements the event counter each reference clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter becomes negative, it stores a generic event record with an EventId of 6.

The reference clock runs at a constant frequency that is independent of the core frequency and of the performance state. The reference clock frequency is processor dependent. The processor may implement this event by subtracting the ratio of (reference clock frequency / core clock frequency) each core clock cycle.

[AMD Public Use]



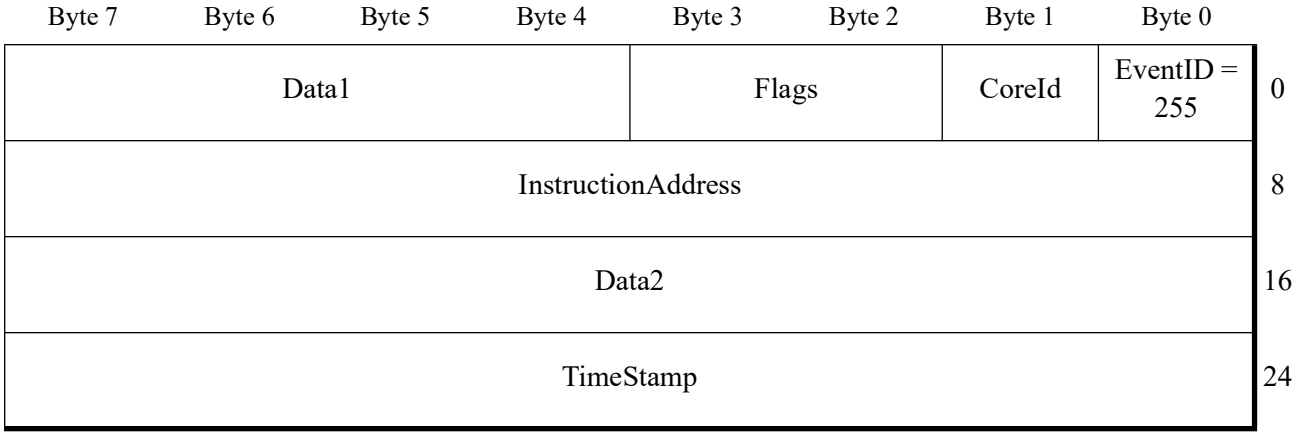
Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 6
1	7:0	CoreId	CPU identifier from LWP_CFG
2–7			Reserved
15–8		InstructionAddress	Instruction address
23–16			Reserved
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-39. CPU Reference Clocks not Halted Event Record

13.4.2.7 Programmed Event

When a program executes the LWPINS instruction (see “LWPINS—Insert User Event Record in LWP Ring Buffer” on page 482), the processor stores an event record with an event identifier of 255. The data in the event record come from the operands to the instruction as detailed in the instruction description.

[AMD Public Use]



Bytes	Field	Description
0	EventId	Event identifier = 255
1	CoreId	CPU identifier from LWP_CFG
3–2	Flags	Imm16 value
7–4	Data1	Reg/mem value
15–8	InstructionAddress	Instruction address of LWPINS instruction
23–16	Data2	Reg value (zero extended if running in legacy mode)
31–24	TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-40. Programmed Event Record

13.4.2.8 Other Events

The overall design of LWP allows easy extension to the list of events that it can monitor. The following are possibilities for events that may be added in future versions of LWP:

- DTLB misses
- FPU operations
- ICache misses
- ITLB misses

13.4.3 Detecting LWP

An application uses the CPUID instruction to identify whether Lightweight Profiling is present and which of its capabilities are available for use. An operating system uses CPUID to determine whether LWP is supported on the hardware and to determine which features of LWP are supported and can be made available to applications.

[AMD Public Use]

13.4.3.1 Detecting LWP Presence

LWP is supported on a processor if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. This bit is identical to the value of CPUID Fn0000_000D_EDX_x0[bit 30], which is bit 62 of the XFeatureSupportedMask and indicates XSAVE support for LWP. A system can check either of those bits to determine if LWP is supported. Since LWP requires XSAVE, software can assume that this bit being set implies that CPUID Fn0000_0001_ECX[XSAVE] (bit 26) is also set.

13.4.3.2 Detecting LWP XSAVE Area

The size of the LWP extended state save area used by XSAVE/XRSTOR is 128 bytes (080h). This value is returned by CPUID Fn0000_000D_ EAX_x3E (ECX=62).

The offset of the LWP save area from the beginning of the XSAVE/XRSTOR area is 832 bytes (340h). This value is returned by CPUID Fn0000_000D_ EBX_x3E (ECX=62).

The size of the LWP save area is included in the XFeatureSupportedSizeMax value returned by CPUID Fn0000_000D_ECX_x0 (ECX=0).

If LWP is enabled in the XCR0, the size of the LWP save area is included in the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0).

13.4.3.3 Detecting LWP Capabilities

The values returned by CPUID Fn8000_001C indicate the capabilities of LWP. See Table 13-13, “Lightweight Profiling CPUID Values” for a listing of the returned values.

Bit 0 of EAX is a copy of bit 62 from XCR0 and indicates whether LWP is available for use by applications. If it is 1, the processor supports LWP and the operating system has enabled LWP for applications.

Bits 31:1 returned in EAX are taken from the LWP_CFG MSR and reflect the LWP features that are available for use. These are a subset of the bits returned in EDX, which reflect the full capabilities of LWP on current processor. The operating system can make a subset of LWP available if it cannot handle all supported features. For instance, if the OS cannot handle an LWP threshold interrupt, it can disable the feature. User-mode software must assume that the bits in EAX describe the features it can use. Operating systems should use the bits from EDX to determine the supported capabilities of LWP and make all or some of those features available.

Under SVM, if a VMM allows the migration of guests among processors that all support LWP, it must arrange for CPUID to report the logical AND of the supported feature bits over all processors in the migration pool. Other CPUID values must also be reported as the “least common denominator” among the processors.

[AMD Public Use]

Table 13-13. Lightweight Profiling CPUID Values

Reg	Bits	Field	Description
EAX	0	LwpAvail	1—LWP is available to application programs. The hardware and the operating system support LWP. 0—LWP is not available. This bit is a copy of bit 62 of the XCR0 register.
	1	LwpVAL	LWPVAL instruction (EventId = 1) is available.
	2	LwpIRE	Instructions retired event (EventId = 2) is available.
	3	LwpBRE	Branch retired event (EventId = 3) is available.
	4	LwpDME	DCache miss event (EventId = 4) is available.
	5	LwpCNH	CPU clocks not halted event (EventId = 5) is available.
	6	LwpRNH	CPU reference clocks not halted event (EventId = 6) is available.
	28:7		Reserved
	29	LwpCont	Sampling in continuous mode is available.
	30	LwpPTSC	Performance Time Stamp Counter in event records is available.
	31	LwpInt	Interrupt on threshold overflow is available.
EBX	7:0	LwpCbSize	Size in quadwords of the LWPCB. This value is at least (LwpEventOffset / 8) + LwpMaxEvents but an implementation may require a larger control block.
	15:8	LwpEventSize	Size in bytes of an event record in the LWP event ring buffer. (32 for LWP Version 1.)
	23:16	LwpMaxEvents	Maximum supported EventId value (not including EventId 255 used by the LWPINS instruction). Not all events between 1 and LwpMaxEvents are necessarily supported.
	31:24	LwpEventOffset	Offset from the start of the LWPCB to the EventInterval1 field. Software uses this value to locate the area of the LWPCB that describes events to be sampled. This permits expansion of the initial fixed region of the LWPCB. LwpEventOffset is always a multiple of 8.

[AMD Public Use]

Table 13-13. Lightweight Profiling CPUID Values

Reg	Bits	Field	Description
ECX	4:0	LwpLatencyMax	Number of bits in cache latency counters (10 to 31). 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	5	LwpDataAddress	1—Cache miss event records report the data address of the reference. 0—Data address is not reported. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	8:6	LwpLatencyRnd	The amount by which cache latency is rounded. The bottom LwpLatencyRnd bits of latency information will be zero. The actual number of bits implemented for the counter is (LwpLatencyMax – LwpLatencyRnd). Must be 0 to 4. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	15:9	LwpVersion	Version of LWP implementation. (1 for LWP Version 1.)
	23:16	LwpMinBufferSize	Minimum size of the LWP event ring buffer, in units of 32 event records. At least 32*LwpMinBufferSize records must be allocated for the LWP event ring buffer, and hence the size of the ring buffer must be at least 32 * LwpMinBufferSize * LwpEventSize bytes. If 0, there is no minimum.
	27:24		Reserved
	28	LwpBranchPrediction	1—Branches Retired events can be filtered based on whether the branch was predicted properly. The values of NMB and NPB in the LWPCB enable filtering based on prediction. 0—NMB and NPB fields of the LWPCB are ignored. 0 if Branches Retired event is not supported (EDX[LwpBRE] = 0).
	29	LwpIpFiltering	1—IP filtering is supported. 0—IP filtering is not supported. The IPI, IPF, BaseIP, and LimitIP fields of the LWPCB are ignored.
	30	LwpCacheLevels	1—Cache-related events can be filtered by the cache level that returned the data. The value of CLF in the LWPCB enables cache level filtering. 0—CLF is ignored. An implementation must support filtering either by latency or by cache level. It may support both. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	31	LwpCacheLatency	1—Cache-related events can be filtered by latency. The value of MinLatency in the LWPCB controls filtering. 0—MinLatency is ignored. An implementation must support filtering either by latency or by cache level. It may support both. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
EDX	0	LwpAvail	LWP is supported. If 0, the remainder of the data returned by CPUID should be ignored. This bit is a copy of CPUID Fn8000_0001_ECX[LWP] (bit 15).
	1	LwpVAL	LWPVAL instruction (EventId = 1) is supported.

[AMD Public Use]

Table 13-13. Lightweight Profiling CPUID Values

Reg	Bits	Field	Description
	2	LwpIRE	Instructions retired event (EventId = 2) is supported.
	3	LwpBRE	Branch retired event (EventId = 3) is supported.
	4	LwpDME	DCache miss event (EventId = 4) is supported.
	5	LwpCNH	CPU clocks not halted event (EventId = 5) is supported.
	6	LwpRNH	CPU reference clocks not halted event (EventId = 6) is supported.
	28:7		Reserved
	29	LwpCont	Sampling in continuous mode is supported.
	30	LwpPTSC	Performance Time Stamp Counter in event records is supported.
	31	LwpInt	Interrupt on threshold overflow is supported.

For more information on using the CPUID instruction, refer to Section 3.3, “Processor Feature Identification,” on page 73.

13.4.4 LWP Registers

The XCR0 register and the LWP model-specific registers describe and control the LWP hardware. The MSRs are available if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. LWP can only be used if the system has made support for LWP state management available in XCR0.

13.4.4.1 XCR0 Support

LWP requires that the processor support the XSAVE/XRSTOR instructions to manage LWP state, along with the XSETBV/XGETBV instructions that manage the enabled state mask. An operating system uses XSETBV to set bit 62 of XCR0 to indicate that it supports management of LWP state and allows applications to use LWP. When the system makes LWP available by setting bit 62 of XCR0, LWP is initially disabled (LWP_CBADDR is zero).

See “Guidelines for Operating Systems” on page 500 for details on how to implement LWP support in an operating system.

13.4.4.2 LWP_CFG—LWP Configuration MSR

LWP_CFG (MSR C000_0105h) controls which features of LWP are available on the processor. The operating system loads LWP_CFG at start-up time (or at the time an LWP driver is loaded) to indicate its level of support for LWP. Only bits for supported features (those that are set in CPUID Fn8000_001C_EDX) can be turned on in LWP_CFG. Attempting to set other bits causes a #GP fault.

User code can examine LWP_CFG bits 31:1 by reading CPUID Fn8000_001C_EAX.

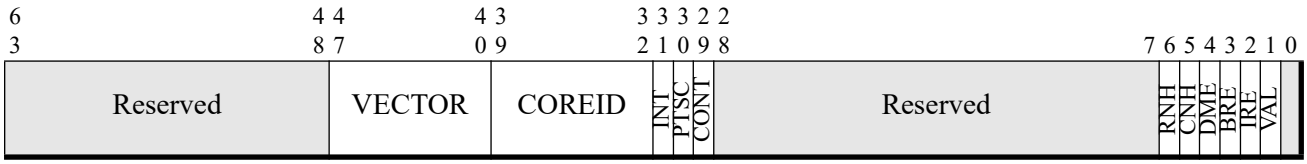
Bits 39:32 of LWP_CFG contains the COREID value that LWP will store into the CoreId field of every event record written by this core. The operating system should initialize this value to be the local APIC number, obtained from CPUID Fn0000_0001_EBX[LocalApicId] (bits 31:24). COREID is

[AMD Public Use]

present so that when LWP is used in a virtualized environment, it has access to the core number without needing to enter the hypervisor. On systems that support x2APIC, local APIC numbers may be more than 8 bits wide. The operating system may then assign LWP COREID values that are small and identify the core within a cluster. If the system has more than 256 cores, there will be unavoidable duplication of COREID values.

Bits 47:40 of LWP_CFG specify the vector number that LWP will use when it signals a ring buffer threshold interrupt.

The reset value of LWP_CFG is 0.



Bits	Field	Description
0		Reserved
1	VAL	Allow the LWPVAL instruction.
2	IRE	Allow LWP to count instructions retired.
3	BRE	Allow LWP to count branches retired.
4	DME	Allow LWP to count DCache misses.
5	CNH	Allow LWP to count CPU clocks not halted.
6	RNH	Allow LWP to count CPU reference clocks not halted.
28:7		Reserved
29	CONT	Enable continuous mode. If 0, LWP will always use synchronized mode.
30	PTSC	Enable storing Performance Time Stamp Counter (PTSC) in the TimeStamp field of event records, if PTSC is available.
31	INT	Allow LWP to generate an interrupt when threshold is exceeded.
39:32	COREID	Value to store in CoreId field when writing an event record.
47:40	VECTOR	Interrupt vector number to use for LWP Threshold interrupts. Must be provided if INT=1.
63:48		Reserved

Figure 13-41. LWP_CFG—Lightweight Profiling Features MSR

13.4.4.3 LWP_CBADDR—LWPCB Address MSR

LWP_CBADDR (MSR C000_0106h) provides access to the internal copy of the LWPCB linear address.

RDMSR from this register returns the current LWPCB address without performing any of the operations described for the SLWPCB instruction.

WRMSR to this register with a non-zero value generates a #GP fault; use LLWPCB or XRSTOR to load an LWPCB address.

[AMD Public Use]

Writing a zero to LWP_CBADDR immediately disables LWP, discarding any internal state. For instance, an operating system can write a zero to stop LWP when it terminates a thread.

Note that LWP_CBADDR contains the linear address of the control block. All references to the LWPCB that are made by microcode during the normal operation of LWP ignore the DS segment register.

The reset value of LWP_CBADDR is 0. This means that when the system sets bit 62 of XCR0 to make LWP available, it is initially disabled.

13.4.5 LWP Instructions

This section describes the instructions included in the AMD64 architecture to support LWP. These instructions raise #UD if LWP is not supported or if bit 62 of XCR0 is 0 indicating that LWP is not available.

The LLWPCB instruction enables or disables Lightweight Profiling and controls the events being profiled. The SLWPCB instruction queries the current state of Lightweight Profiling.

LWP provides two instructions for inserting user data into the event ring buffer. The LWPINS instruction unconditionally stores an event record into the ring buffer, while the LWPVAL instruction uses an LWP event counter to sample program values at defined intervals.

The instructions LLWPCB, SLWPCB, LWPINS, and LWPVAL are also described in the chapter "General-Purpose Instruction Reference" of Volume 3. Refer to reference pages for the individual instruction for information on instruction encoding, flags affected, and exception behavior.

13.4.5.1 LLWPCB—Load LWPCB Address

Parses the Lightweight Profiling Control Block at the address contained in the specified register. If the LWPCB is valid, writes the address into the LWP_CBADDR MSR and enables Lightweight Profiling.

The LWPCB must be in memory that is readable and writable in user mode. For better performance, it should be aligned on a 64-byte boundary in memory and placed so that it does not cross a page boundary, though neither of these suggestions is required.

Action

1. If LWP is not available or if the machine is not in protected mode, LLWPCB immediately causes a #UD exception.
2. If LWP is already enabled, the processor flushes the LWP state to memory in the old LWPCB. See "SLWPCB—Store LWPCB Address" on page 481 for details on saving the active LWP state.

   If the flush causes a #PF exception, LWP remains enabled with the old LWPCB still active. Note that the flush is done before LWP attempts to access the new LWPCB.
3. If the specified LWPCB address is 0, LWP is disabled and the execution of LLWPCB is complete.
4. The LWPCB address is non-zero. LLWPCB validates it as follows:

[AMD Public Use]

- If any part of the LWPCB or the ring buffer is beyond the data segment limit, LLWPCB causes a #GP exception.
- If the ring buffer size is below the implementation's minimum ring buffer size, LLWPCB causes a #GP exception.
- While doing these checks, LWP reads and writes the LWPCB, which may cause a #PF exception.

If any of these exceptions occurs, LLWPCB aborts and LWP is left disabled. Usually, the operating system will handle a #PF exception by making the memory available and returning to retry the LLWPCB instruction. The #GP exceptions indicate application programming errors.

5. LWP converts the LWPCB address and the ring buffer address to linear address form by adding the DS base address and stores the addresses internally.
6. LWP examines the LWPCB.Flags field to determine which events should be enabled and whether threshold interrupts should be taken. It clears the bits for any features that are not available and stores the result back to LWPCB.Flags to inform the application of the actual LWP state.
7. For each event being enabled, LWP examines the EventInterval*n* value and, if necessary, sets it to an implementation-defined minimum. (The minimum event interval for LWPVAL is zero.) It loads its internal counter for the event from the value in EventCounter*n*. A zero or negative value in EventCounter*n* means that the next event of that type will cause an event record to be stored. To count every j^{th} event, a program should set EventInterval*n* to *j-1* and EventCounter*n* to some starting value (where *j-1* is a good initial count). If the counter value is larger than the interval, the first event record will be stored after a larger number of events than subsequent records.
8. LWP is started. The execution of LLWPCB is complete.

Notes

If none of the bits in the LWPCB.Flags specifies an available event, LLWPCB still enables LWP to allow the use of the LWPINS instruction. However, no other event records will be stored.

A program can temporarily disable LWP by executing SLWPCB to obtain the current LWPCB address, saving that value, and then executing LLWPCB with a register containing 0. It can later re-enable LWP by executing LLWPCB with a register containing the saved address.

When LWP is enabled, it is typically an error to execute LLWPCB with the address of the active LWPCB. When the hardware flushes the existing LWP state into the LWPCB, it may overwrite fields that the application may have set to new LWP parameter values. The flushed values will then be loaded as LWP is restarted. To reuse an LWPCB, an application should stop LWP by passing a zero to LLWPCB, then prepare the LWPCB with new parameters and execute LLWPCB again to restart LWP.

Internally, LWP keeps the linear address of the LWPCB and the ring buffer. If the application changes the value of DS, LWP will continue to collect samples even if the new DS value would no longer allows it to access the LWPCB or the ring buffer. However, a #GP fault will occur if the application uses XRSTOR to restore LWP state saved by XSAVE. Programs should avoid using XSAVE/XRSTOR on LWP state if DS has changed. This only applies when the CPL ≠ 0; kernel mode

[AMD Public Use]

operation of XRSTOR is unaffected by changes to DS. See “XSAVE/XRSTOR” on page 493 for details.

Operating system and hypervisor code that runs when the CPL ≠ 3 should use XSAVE and XRSTOR to control LWP rather than using LLWPCB (see below). Use WRMSR to write 0 to LWP_CBADDR to immediately stop LWP without saving its current state (see “LWP_CBADDR—LWPCB Address MSR” on page 478).

It is possible to execute LLWPCB when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Furthermore, if LWP is enabled when a kernel executes LLWPCB, both the old and new control blocks and ring buffers must be accessible. Using LLWPCB in these situations is not recommended.

13.4.5.2 SLWPCB—Store LWPCB Address

Flushes LWP state to memory and returns the current effective address of the LWPCB in the specified register.

If LWP is not currently enabled, SLWPCB sets the specified register to zero.

The flush operation stores the internal event counters for active events and the current ring buffer head pointer into the LWPCB. If there is an unwritten event record pending, it is written to the event ring buffer.

If LWP_CBADDR is not zero, the value returned is an effective address that is calculated by subtracting the current DS.Base address from the linear address kept in LWP_CBADDR. Note that if DS has changed between the time LLWPCB was executed and the time SLWPCB is executed, this might result in an address that is not currently accessible by the application.

SLWPCB generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

It is possible to execute SLWPCB when the CPL ≠ 3 or when SMM is active, but if the LWPCB pointer is not zero, the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF fault. Using SLWPCB in these situations is not recommended.

13.4.5.3 LWPVAL—Insert Value Sample in LWP Ring Buffer

Decrements the event counter associated with the Programmed Value Sample event (see “Programmed Value Sample” on page 466). If the resulting counter value is negative, inserts an event record into the LWP event ring buffer in memory and advances the ring buffer pointer. If the counter is not negative and the ModRM operand specifies a memory location, that location is not accessed.

The event record has an EventId of 1. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value

[AMD Public Use]

(third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 13-34 on page 466.

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the event counter is reset to the interval for the event (subject to randomization). If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in continuous mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in synchronized mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, and the head pointer is not advanced.

LWPVAL generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPVAL does nothing if LWP is not enabled or if the Programmed Value Sample event is not enabled in LWPCB.Flags. This allows LWPVAL instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPVAL when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPVAL in these situations is not recommended.

LWPVAL can be used by a program to perform value profiling. This is the technique of sampling the value of some program variable at a predetermined frequency. For example, a managed runtime might use LWPVAL to sample the value of the divisor for a frequently executed divide instruction in order to determine whether to generate specialized code for a common division. It might sample the target location of an indirect branch or call to see if one destination is more frequent than others. Since LWPVAL does not modify any registers or condition codes, it can be inserted harmlessly between any instructions.

Note

When LWPVAL completes (whether or not it stored an event record in the event ring buffer), it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store an event record. If LWPVAL also stored an event record, the buffer will contain two records with the same instruction address (but different EventId values).

13.4.5.4 LWPINS—Insert User Event Record in LWP Ring Buffer

Inserts a record into the LWP event ring buffer in memory and advances the ring buffer pointer.

The record has an EventId of 255. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 13-40 on page 473.

[AMD Public Use]

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the CF flag is cleared. If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in continuous mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in synchronized mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, the head pointer is not advanced, and the CF flag is set.

LWPINS generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPINS simply clears CF if LWP is not enabled. This allows LWPINS instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPINS when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPINS in these situations is not recommended.

LWPINS can be used by a program to mark significant events in the ring buffer as they occur. For instance, a program might capture information on changes in the process' address space such as library loads and unloads, or changes in the execution environment such as a change in the state of a user-mode thread of control.

Note that when the LWPINS instruction finishes writing a event record in the event ring buffer, it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store another event record with the same instruction address (but different EventId values).

13.4.6 LWP Control Block

An application uses the LWP Control Block (LWPCB) to specify the details of Lightweight Profiling operation. It is an interactive region of memory in which some fields are controlled and modified by the LWP hardware and others are controlled and modified by the software that processes the LWP event records.

Most of the fields in the LWPCB are constant for the duration of a LWP session (the time between enabling LWP and disabling it). This means that they are loaded into the LWP hardware when it is enabled, and may be periodically reloaded from the same location as needed. The contents of the constant fields must not be changed during a LWP run or results will be unpredictable. Changing the LWPCB memory to read-only or unmapped will cause an exception the next time LWP attempts to access it. To change values in the LWPCB, disable LWP, change the LWPCB (or create a new one), and re-enable LWP.

[AMD Public Use]

A few fields are modified by the LWP hardware to communicate progress to the software that is emptying the event ring buffer. Software may read them but should never modify them during an LWP session. Other fields are for software to modify to indicate that progress has been made in emptying the ring buffer. Software writes these fields and the LWP hardware reads them as needed.

For efficiency, some of the LWPCB fields may be shadowed internally in the LWP hardware unit when profiling is enabled. LWP refreshes these fields from (or flushes them to) memory as needed to allow software to make progress. For more information, refer to “LWPCB Access” on page 499.

The BufferTailOffset field is at offset 64 in the LWPCB in order to place it in a separate cache line on most implementations, assuming that the LWPCB itself is aligned properly. This allows the software thread that is emptying the ring buffer to retain write ownership of that cache line without colliding with the changes made by LWP when writing BufferHeadOffset. In addition, most implementations will use a value of 128 as the offset to the EventInterval1 field, since that places the event information in a separate cache line.

All fields in the LWPCB (as shown in Figure 13-42) that are marked as “Reserved” (or “Rsvd”) should be zero.

[AMD Public Use]

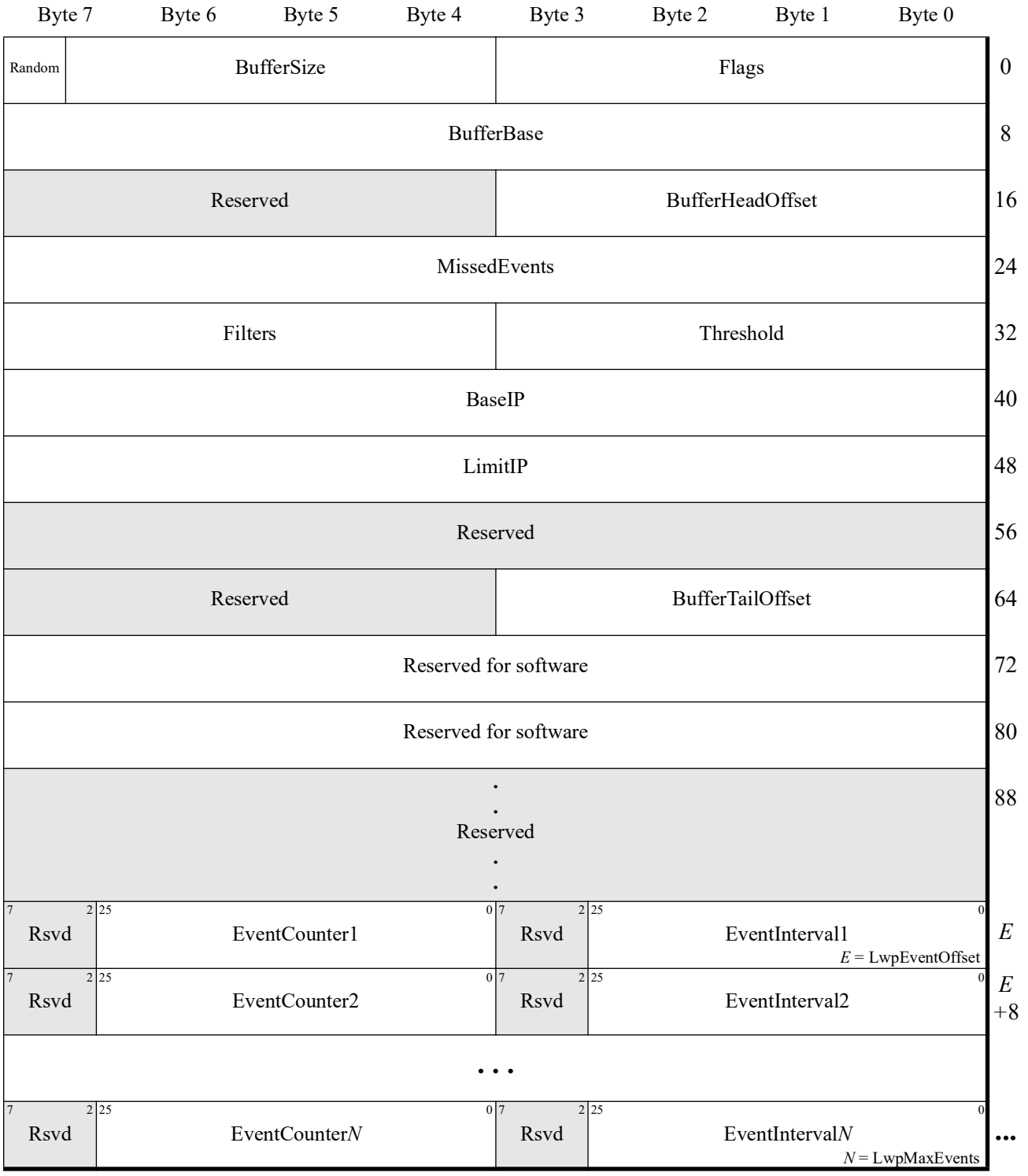


Figure 13-42. LWPCB—Lightweight Profiling Control Block

The R/W column in Table 13-14 below indicates how a field is used while LWP is enabled:

[AMD Public Use]

- LWP—hardware modifies the field; software may read it, but must not change it
- Init—hardware reads and modifies the field while executing LLWPCB; the field must then remain unchanged as long as the LWPCB is in use
- SW—software may modify the field; hardware may read it, but does not change it
- No—field must remain unchanged as long as the LWPCB is in use

Table 13-14. LWPCB—Lightweight Profiling Control Block Fields

Bytes	Bits	Field	Description	R/W
3–0		Flags	Flags indicating which events should be or are being counted (see Figure 13-43, “LWPCB Flags”) and whether threshold interrupts should be enabled. Before executing LLWPCB, the application sets Flags to a bit mask of the events (and interrupt) that should be enabled. LLWPCB does a logical “and” of this mask with the available feature bits in LWP_CFG and rewrites Flags with the mask of features actually enabled.	Init
7–4	27:0	BufferSize	Total size of the event ring buffer (in bytes). Must be a multiple of the event record size LwpEventSize (the value used internally will be rounded down if not). BufferSize must be at least (32 * LwpMinBufferSize * LwpEventSize).	No
7	7:4	Random	Number of bits of randomness to use in counters. Each time a counter is loaded from an interval to start counting down to the next event to record, the bottom Random bits are set to a random value. This avoids fixed patterns in events.	No
15–8		BufferBase	The Effective Address of the event ring buffer. Should be aligned on a 64-byte boundary for reasonable performance. Software is encouraged to align the ring buffer to a page boundary for best performance. If the default address size is less than 64 bits, the upper bits of BufferBase must be zero. LLWPCB converts BufferBase to a linear address and stores it internally. LWPCB.BufferBase is not modified.	No
19–16		BufferHeadOffset	Unsigned offset from BufferBase specifying where the LWP hardware will store the next event record. When BufferHeadOffset == BufferTailOffset, the ring buffer is empty. BufferHeadOffset must always be less than BufferSize; LWP will use a value of 0 if BufferHeadOffset is too large. Also, it must always be a multiple of LwpEventSize; LWP will round it down if not.	LWP
23–20			Reserved	
31–24		MissedEvents	The 64-bit count of the number of events that were missed. A missed event occurs when LWP stores an event record, attempts to advance BufferHeadOffset, and discovers that it would be equal to BufferTailOffset. In this case, LWP leaves BufferHeadOffset unchanged and instead increments the MissedEvents counter. Thus, when the ring buffer is full, the last event record is overwritten.	LWP

[AMD Public Use]

Table 13-14. LWPCB—Lightweight Profiling Control Block Fields (continued)

Bytes	Bits	Field	Description	R/W
35–32		Threshold	Threshold for signaling an interrupt to indicate that the ring buffer is filling up. If threshold interrupts are enabled in Flags, then when LWP advances BufferHeadOffset, it computes the space used as ((BufferHeadOffset – BufferTailOffset) % BufferSize). If the space used equals or exceeds Threshold, LWP causes an interrupt. If Threshold is greater than BufferSize, no interrupt will ever be taken. If Threshold is zero, an interrupt will be taken every time an event record is stored in the ring buffer. Threshold is an unsigned integer multiple of LwpEventSize (the value used internally will be rounded down if not). Ignored if threshold interrupts are not available in LWP_CFG or if they are not enabled in Flags	No
39–36		Filters	Filters to qualify which events are eligible to be counted. This field includes bits to filter branch events by type and prediction status, and bits and values to filter cache events by type and latency. See Figure 13-44, “LWPCB Filters” for details.	
47–40		BaseIP	Low limit of the IP filtering range. An instruction must start at a location greater than or equal to BaseIP to be in range. Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.	No
55–48		LimitIP	High limit of the IP filtering range. An instruction must start at a location less than or equal to LimitIP to be in range. Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.	No
63–56			Reserved	
67–64		BufferTailOffset	Unsigned offset from BufferBase to the oldest event record in the ring buffer. BufferTailOffset is maintained by software and must always be less than BufferSize and a multiple of LwpEventSize. If software stores a value of BufferTailOffset into the LWPCB that violates these rules, the LWP hardware might not detect ring buffer overflow or threshold conditions properly.	SW
71–68			Reserved	
72–87			Reserved for software use. These bytes are never read or written by the LWP hardware	SW
(*E*-1) – 88			Reserved area between the fixed portion of the LWPCB and the event specifiers. Should be zero. The EventInterval1 field is at offset *E* = LwpEventOffset.	

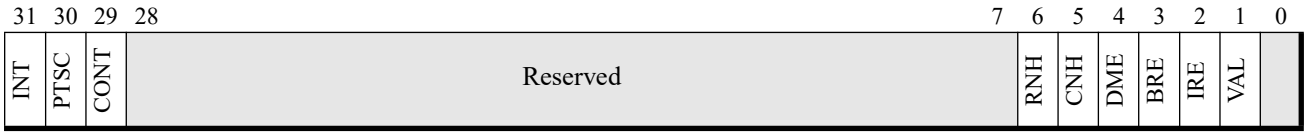
[AMD Public Use]

Table 13-14. LWPCB—Lightweight Profiling Control Block Fields (continued)

Bytes	Bits	Field	Description	R/W
(*E*+3)–*E*	25:0	EventInterval1	Reset value for counting events of type EventId = 1 (Programmed Value Sample). A value of *n* specifies that after *n+1* (modified by Random) LWPVAL instructions, LWP will store an event record in the ring buffer. EventInterval1 is a signed value. If it is negative, LLWPCB will use zero and will store zero into EventInterval1 in the LWPCB. The Programmed Value Sample event is the only one which allows an interval to be below the implementation minimum interval value.	Init
E+3	7:2		Reserved	
(*E*+7)–(*E*+4)	25:0	EventCounter1	Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.	LWP
E+7	7:2		Reserved	
(*E*+11)–(*E+8*)	25:0	EventInterval2	Reset value for counting events of type EventId = 2 (Instructions Retired). A value of *n* specifies that after *n+1* (modified by Random) instructions are retired, LWP will store an event record in the ring buffer. EventInterval2 is a signed value. If it is negative or is below the implementation minimum, LLWPCB will use the minimum and will store that value into EventInterval2 in the LWPCB.	Init
E+11	7:2		Reserved	
(*E*+15)–(*E*+12)	57:32	EventCounter2	Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.	LWP
E+15	7:2		Reserved	
		Event3…	Repeat event configuration similar to EventInterval2 and EventCounter2 for EventId values from 3 to LwpMaxEvents.	

The LLWPCB instruction reads the Flags word from the LWPCB to determine which events to profile and whether threshold interrupts should be enabled. LLWPCB writes the Flags word after turning off bits corresponding to features which are not currently available.

[AMD Public Use]

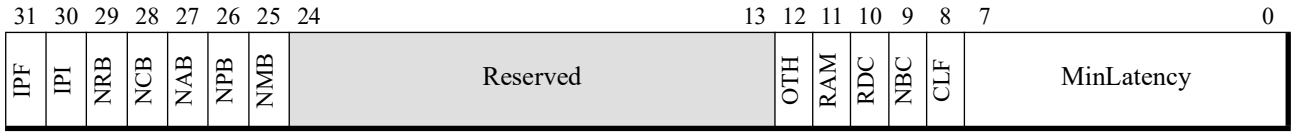


Bit	Field	Input to LLWPCB	Value after LLWPCB
0		Reserved	
1	VAL	Enable LWPVAL instruction	LWPVAL instruction enabled
2	IRE	Enable Instructions Retired event	Instructions Retired event enabled
3	BRE	Enable Branches Retired event	Branches Retired event enabled
4	DME	Enable DCache miss event	DCache Miss event enabled
5	CNH	Enable CPU clocks not halted event	CPU Clocks Not Halted event enabled
6	RNH	Enable CPU reference clocks not halted event	CPU Reference Clocks Not Halted event enabled
28:7		Reserved	
29	CONT	1—Use continuous mode. If the ring buffer overflows, LWP continues to store events and advance BufferHead. Software must stop LWP in order to empty the ring buffer. 0—Use synchronized mode.	LWP operates in continuous mode if input bit is set and continuous mode is available. Otherwise, LWP operates in synchronous mode.
30	PTSC	1—Store the Performance Time Stamp Counter (PTSC) in the TimeStamp field of each event record, if PTSC is available. 0—Store 0 in the TimeStamp field.	Performance Time Stamp Counter value will be stored if input bit is set and PTSC feature is available. Otherwise 0 is stored.
31	INT	Enable threshold interrupts.	Threshold interrupts are enabled.

Figure 13-43. LWPCB Flags

Event counting can be filtered by a number of conditions which are specified in the Filters word of the LLWPCB. The IP filtering applies to all events. Cache level filtering applies to all events that interact with the caches. Branch filtering applies to the Branches Retired event.

[AMD Public Use]



Bits	Field	Description
7:0	MinLatency	Minimum latency for a cache-related event
8	CLF	Cache level filtering
9	NBC	Northbridge cache events
10	RDC	Remote data cache events
11	RAM	DRAM cache events
12	OTH	Other cache events
24:13		Reserved
25	NMB	No mispredicted branches
26	NPB	No predicted branches
27	NAB	No absolute branches
28	NCB	No conditional branches
29	NRB	No unconditional relative branches
30	IPI	IP filtering invert
31	IPF	IP filtering

Figure 13-44. LWPCB Filters

The following table provides detailed descriptions of the fields in the Filters word.

[AMD Public Use]

Table 13-15. LWPCB Filters Fields

Bits	Field	Description
7:0	MinLatency	Minimum latency for a cache-related event to be eligible for LWP counting. Applies to all cache-related events being monitored. MinLatency is multiplied by 16 to get the actual latency in cycles, providing less resolution but a larger range for filtering. An implementation may have a maximum for the latency value. If MinLatency*16 exceeds this maximum value, the maximum is used instead. A value of 0 disables filtering by latency. Ignored if no cache latency event is enabled or if the CPUID LwpCacheLatency bit is 0 to indicate that the implementation does not filter by latency (use the CLF bits to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.
8	CLF	Cache level filtering. 1—Enables filtering cache-related events by the cache level or memory level that returned the data. It enables the next 4 bits. Cache-related events are only eligible for counting if the bit describing the memory level is on. 0—Disables cache level filtering. The next 4 bits are ignored, and any cache or memory level is eligible. Ignored if no cache latency event is enabled or if the CPUID LwpCacheLevels bit is 0 to indicate that the implementation does not filter by cache level (use the MinLatency field to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.
9	NBC	Northbridge cache events. 1—Count cache-related events that are satisfied from data held in a cache that resides on the Northbridge. 0—Ignore Northbridge cache events Ignored if CLF is 0.
10	RDC	Remote data cache events. 1—Count cache-related events that are satisfied from data held in a remote data cache. 0—Ignore remote cache events. Ignored if CLF is 0.
11	RAM	DRAM cache events. 1—Count cache-related events that are satisfied from DRAM. 0—Ignore DRAM cache events. Ignored if CLF is 0.
12	OTH	Other cache events. 1—Count cache-related events that are satisfied from other sources, such as MMIO, Config space, PCI space, or APIC. 0—Ignore such cache events Ignored if CLF is 0.
24:13		Reserved

[AMD Public Use]

Table 13-15. LWPCB Filters Fields (continued)

Bits	Field	Description
25	NMB	No mispredicted branches. 1—Mispredicted branches will not be counted. 0—Mispredicted branches will be counted if not suppressed by other filter conditions. Caution: If NMB and NPB are both set, no branches will be counted. Ignored if the Branches Retired event is not enabled or if the CPUID LwpBranchPrediction bit is 0 to indicate that the implementation does not filter by prediction.
26	NPB	No predicted branches. 1—Correctly predicted branches will not be counted. Note that since direct branches are always predicted correctly, this is a superset of the NDB filter. 0—Correctly predicted branches will be counted if not suppressed by other filter conditions. Caution: If NMB and NPB are both set, no branches will be counted. Ignored if the Branches Retired event is not enabled or if the CPUID LwpBranchPrediction bit is 0 to indicate that the implementation does not filter by prediction.
27	NAB	No absolute branches. 1—Absolute branches will not be counted. This only applies to jumps through a register or memory (JMP opcode FF /4) and calls through a register or memory (CALL opcode FF /2). Relative branches (both conditional and unconditional) are counted normally if not disabled via the NRB or NCB bits. 0—Absolute branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the Branches Retired event is not enabled.
28	NCB	No conditional branches. 1—Conditional branches will not be counted. This only applies to conditional jumps (Jcc) and loops (LOOPcc). Unconditional relative branches, indirect jumps through a register or memory, and returns are counted normally if not disabled via the NRB or NAB bits. 0—Conditional branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the Branches Retired event is not enabled.

[AMD Public Use]

Table 13-15. LWPCB Filters Fields (continued)

Bits	Field	Description
29	NRB	No unconditional relative branches. 1—Unconditional relative branches will not be counted. This applies to unconditional jumps (JMP), calls (CALL), and returns (RET). Conditional branches and indirect jumps or calls through a register or memory are counted normally if not disabled via the NCB or NAB bits. 0—Direct branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the Branches Retired event is not enabled.
30	IPI	IP filtering invert. 1—IP filtering inverted. Only instructions outside the range from BaseIP to LimitIP are eligible for LWP counting. 0—IP filtering normal. Only instructions inside the range from BaseIP to LimitIP are eligible for LWP counting. Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.
31	IPF	IP filtering. 1—IP filtering enabled. The values of the BaseIP and LimitIP fields specify a range of instruction addresses that are eligible for LWP event counting and reporting. The range is inclusive if IPI is 0 and exclusive if IPI is 1. 0—IP filtering disabled; instructions at every address are eligible for LWP counting. Ignored if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.

13.4.7 XSAVE/XRSTOR

LWP requires that the processor support the XSAVE/XRSTOR instructions for managing extended processor state components.

13.4.7.1 Configuration

The processor uses bit 62 of XCR0 register to indicate whether LWP state can be saved and restored, and thus whether LWP is available to applications. The LWP XSAVE area length and offset from the beginning of the XSAVE area are available from the CPUID instruction (see “Detecting LWP XSAVE Area” on page 474). In Version 1 of LWP, the LWP XSAVE area is 128 (080h) bytes long and the offset is 832 (340h) bytes.

13.4.7.2 XSAVE Area

Figure 13-45 below shows the layout of the XSAVE area for LWP. It is large enough to allow for future expansion of the number of event counters. Details of the fields are in Table 13-16.

All fields in the XSAVE area that are marked as “Reserved” (or “Rsvd”) must be zero.

[AMD Public Use]

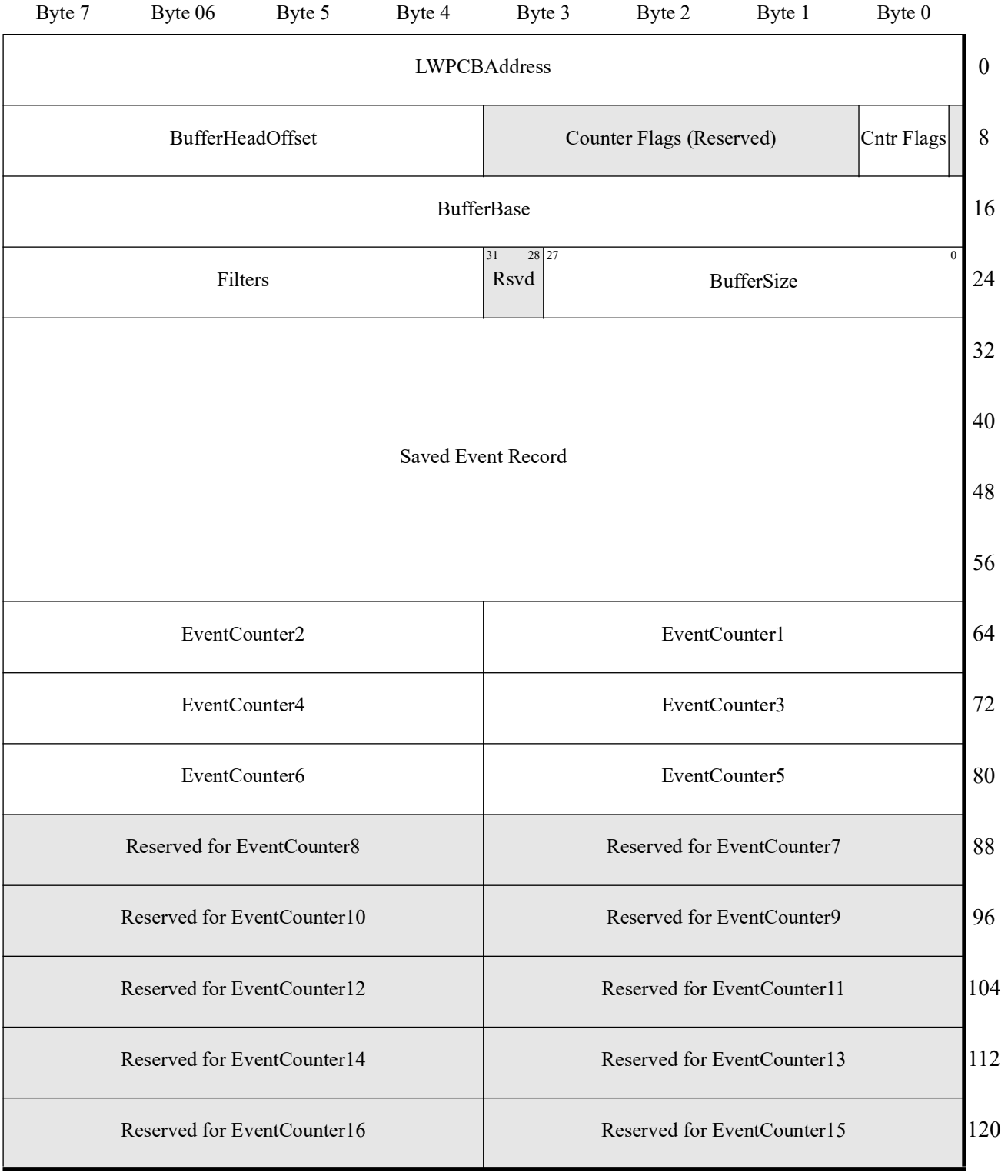


Figure 13-45. XSAVE Area for LWP

[AMD Public Use]

Table 13-16. XSAVE Area for LWP Fields

Bytes	Bits	Field	Description
7–0		LWPCBAddress	Address of LWPCB. 0 if LWP is disabled, in which case the rest of the save area is ignored. This is a linear address.
9–8	0	—	Reserved
9–8	1	CntrFlags.Counter1	1—Event with EventId 1 is active. XRSTOR will make the event active and restore its counter from EventCounter1. 0—Event 1 is not active. XRSTOR will make the event inactive.
9–8	6:2	CntrFlags.Counter*n*	Bit flags defined as above for EventCounter2–6.
9–8	15:7	—	Reserved for counter flags
11–10	15:0	—	Reserved for counter flags
15–12		BufferHeadOffset	BufferHeadOffset value
23–16		BufferBase	Address of the event ring buffer. This is a linear address.
27–24	27:0	BufferSize	Size of the event ring buffer
27–24	31:28	—	Reserved
31–28		Filters	Profiling filters (same as the Filters field in the LWPCB)
63–32		SavedEventRecord	If an event record is pending, the data to write. May be sparse. Zero in the EventId field means no record pending.
67–64		EventCounter1	Counter for event 1 (valid if CntrFlags.Counter1 bit is set)
87–68		EventCounter*n*	Counters for events 2–6 (valid if the respective Counter*n* bit is set)
127–88		—	Reserved for future event counters

13.4.7.3 XSAVE operation

If LWP is not currently enabled (i.e., if LWP_CBADDR = 0), no state needs to be stored. XSAVE sets bit 62 in XSAVE.HEADER.XSTATE_BV to 0 so that an attempt to restore state from this save area will use the processor supplied values. See “Processor supplied values” on page 497.

If LWP is enabled, XSAVE stores the various internal LWP values into the XSAVE area with no checking or conversion and sets bit 62 in XSAVE.HEADER.XSTATE_BV to 1.

13.4.7.4 XRSTOR operation

If bit 62 in XCR0 is 0 or if bit 62 of EDX:EAX (EDX[30]) is 0, XRSTOR does not alter the LWP state.

If the above bits are 1 but bit 62 in XSAVE.HEADER.XSTATE_BV is 0, XRSTOR writes the LWP state using the processor supplied values, disabling LWP. See “Processor supplied values” on page 497.

If all of the above bits are 1, XRSTOR loads LWP state from the XSAVE area as follows:

1. The internal pointers and sizes are loaded.
   - If BufferSize is below the implementation minimum, LWP is disabled and XRSTOR of LWP state terminates.

[AMD Public Use]

- If BufferSize is not a multiple of the event record size, it is rounded down.
- If BufferHeadOffset is greater than (BufferSize - LwpEventSize), a value of 0 is used instead.
- If BufferHeadOffset is not a multiple of the event record size, it is rounded down.

2. For each bit that is set in the Flags field that corresponds to an available event (as currently set in the LWP_CFG MSR), the corresponding event is enabled and the event counter is loaded from the EventCounter*n* field. All other events are disabled.
3. If the EventId field in the SavedEventRecord is non-zero, there was a pending event when XSAVE was executed. XRSTOR loads the event record into hardware. LWP will store it into the event ring buffer as soon as possible once the CPL is 3.

   Software should not alter the SavedEventRecord field. An implementation may ignore a saved event record if it was not constructed by XSAVE. Storing an event into SavedEventRecord and then executing XRSTOR is not a reliable way of injecting an event into the ring buffer.

Note that if LWP is already enabled when executing XRSTOR, the old LWP state is overwritten without being saved.

No interrupt is generated by XRSTOR if the restored value of BufferHeadOffset results in a buffer that is filled beyond the threshold. The interrupt will occur the next time an event record is stored.

XRSTOR may not restore all of the state necessary for LWP to operate. The LWP hardware will read additional state from the LWPCB when it stores then next event record.

If the CPL = 0, XRSTOR simply reloads the LWPCB address and the ring buffer address from the XSAVE area. Kernel software is trusted not to alter the area in such a way as to allow access to memory that the application could not otherwise read or write. The linear addresses in the XSAVE area were validated when the application executed LLWPCB.

If the CPL ≠ 0, XRSTOR first validates the LWPCB and ring buffer pointers. This prevents an application from altering the XSAVE area in order to gain access to memory that it could not otherwise read or write (based on the current values in the DS segment register). Note that if a program's DS value changes after doing a successful LLWPCB, it might be incapable of doing an XSAVE and then an XRSTOR of LWP state. The XRSTOR will fail if the new DS value no longer allows access to the linear addresses corresponding to the LWPCB or the ring buffer. Programs should avoid this behavior.

If XRSTOR is executed when the CPL ≠ 0, the system performs additional checks on the LWPCB and ring buffer addresses according to the pseudo-code below. A "Store-type Segment_check" fails if the limit check fails (address is beyond the segment limit) or if the segment is read-only.

```
bool Check(uint64 addr, uint32 size) { // Utility function
      if (!64bit_Mode)
            addr = truncate32(addr - DS.BASE)
      uint64 top = addr + size - 1;
      if (! Store-type Segment_check on DS:[addr] || // Check lower bound
          ! Store-type Segment_check on DS:[top])    // and upper bound
            return false;
      return true;
```

[AMD Public Use]


```
}

if (! Check(XSAVE.LWPCBAddress, sizeof(LWPCB)) ||
    ! Check(XSAVE.BufferAddress, XSAVE.BufferSize))
      Disable LWP
```

If any of the address checks fails, LWP is disabled. No fault is generated. A program that executes XRSTOR when the CPL ≠ 0 and DS has changed can use SLWPCB to check whether LWP is running.

As with all features that use XSAVE and XRSTOR, if bit 62 of XCR0 is 0 but bit 62 of XSAVE.HEADER.XSTATE_BV is 1, XRSTOR will cause a #GP(0) exception.

13.4.7.5 Processor supplied values

If XRSTOR is executed when bit 62 of XCR0 and EDX:EAX are both 1, but the corresponding bit in XSAVE.HEADER.XSTATE_BV is 0, it indicates that there is no LWP state to restore. In this case, LWP_CBADDR is set to 0 and LWP is disabled. Other processor internal state for LWP is set to 0 as necessary to avoid security issues.

13.4.8 Implementation Notes

The following subsections describe other LWP considerations.

13.4.8.1 Multiple Simultaneous Events

Multiple events are possible when an instruction retires. For instance, an indirect jump through a pointer in memory can trigger the instructions retired, branches retired, and DCache miss events simultaneously. LWP counts all events that apply to the instruction, but might not store event records for all events whose event counters became negative. It is implementation dependent as to how many event records are stored when multiple event counters simultaneously become negative. If not all events cause event records to be stored, the choice of which event(s) to report is implementation dependent and may vary from run to run on the same processor.

13.4.8.2 Processor State for Context Switch, SVM, and SMM

Implementations of LWP have internal state to hold information such as the current values of the counters for the various events, a pointer into the event ring buffer, and a copy of the tail pointer for quick detection of threshold and overflow states.

There are times when the system must preserve the volatile LWP state. When the operating system context switches from one user thread to another, the old user state must be saved with the thread's context and the new state must be loaded. When a hypervisor decides to switch from one guest OS to another, the same must be done for the guest systems' states. Finally, state must be stored and reloaded when the system enters and exits SMM, since the SMM code may decide to shut off power to the core.

Hardware does not maintain the LWP state in the active LWPCB. This is because the counters change with every event (not just every reported event), so keeping them in memory would generate a large

[AMD Public Use]

amount of unnecessary memory traffic. Also, the LWPCB is in user memory and may be paged out to disk at any time, so the memory may not be available when needed.

Saving State at Thread Context Switches

LWP requires that an operating system use the XSAVE and XRSTOR instructions to save and restore LWP state across context switches.

XRSTOR restores the LWP volatile state when restoring other system state. Some additional LWP state will be restored from the LWPCB when operations in ring 3 require that information.

LWP does not support the "lazy" state save and restore that is possible for floating point and SSE state. It does not interact with the CR0[TS] bit. Operating systems that support LWP must always do an XSAVE to preserve the old thread's LWP context and an XRSTOR to set up the new LWP context. The OS can continue to do a lazy switch of the FP and SSE state by ensuring that the corresponding bits in EDX:EAX are clear when it executes the XSAVE and XRSTOR to handle the LWP context.

Saving State at SVM Worldswitch to a Different Guest

Hypervisors that allow guests to use LWP must save and restore LWP state when the guest OS changes. In addition to the usual information in the VMCB, the hypervisor must use XSAVE/XRSTOR to maintain the volatile LWP state and must also save and restore LWP_CFG. When switching between a guest that uses LWP and one that does not, the hypervisor changes the value of XCR0, which ensures that LWP is only enabled in the appropriate guest.

A hypervisor need not modify the LWP state if the guest OS is not changed.

Enabling SVM Live Migration

Some hypervisors support live migration of a guest virtual machine. Live migration is when a hypervisor preserves the entire state of the guest running on one physical machine, copies that state to another physical machine, and then resumes execution of the guest on the new hardware.

To allow live migration among machines which may have different internal implementations of LWP, the hypervisor must present the common subset of features among all the hosts in the pool of machines that can be used. Furthermore, since the hypervisor may XSAVE LWP state on one machine and XRSTOR it on another machine, the contents of the XSAVE area must be consistent across all implementations.

This means that an implementation of LWP keeps all event counters internally, not in the LWPCB. If implementations were permitted to differ in this detail, a counter might not get properly restored after migrating the guest machine.

Saving State at SMM Entry and Exit

SMM entry and exit must save and restore LWP state when the processor is going to change power state. SMM must use XSAVE/XRSTOR and must also save and restore LWP_CFG. Since LWP is ring 3 only and is inactive in System Management Mode, its state should not need to be saved and restored otherwise.

[AMD Public Use]

Notes on Restoring LWP State

The LWPCB may not be in memory at all times. Therefore, the LWP hardware does not attempt to access it while still in the OS kernel/VMM/SMM, since that access might fault. Some LWP state is restored once the processor is in ring 3 and can take a #PF exception without crashing. This usually happens the next time LWP needs to store an event record into the ring buffer.

13.4.8.3 LWPCB Access

Several LWPCB fields are written asynchronously by the LWP hardware and by the user software. This section discusses techniques for reducing the associated memory traffic. This is interesting to software because it influences what state is kept internally in LWP, and it explains the protocol between the hardware filling the event ring buffer and the software emptying it.

The hardware keeps an internal copy of the event ring buffer head pointer. It need not flush the head pointer to the LWPCB every time it stores an event record. The flush can be done periodically or it can be deferred until a threshold or buffer full condition happens or until the application executes LLWPCB or SLWPCB. Exceeding the buffer threshold always forces the head pointer to memory so that the interrupt handler emptying the ring buffer sees the threshold condition.

The hardware may keep an internal copy of the event ring buffer tail pointer. It need not read the software-maintained tail pointer unless it detects a threshold or buffer full condition. At that point, it rereads the tail pointer to see if software has emptied some records from the ring buffer. If so, it recomputes the condition and acts accordingly. This implies that software polling the ring buffer should begin processing event records when it detects a threshold condition itself. To avoid a race condition with software, the hardware rereads the tail pointer every time it stores an event record while the threshold condition appears to be true. (An implementation can relax this to “every n^{th} time” for some small value of n.) It also rereads it whenever the ring buffer appears to be full.

The interval values used to reset the counters can be cached in the hardware when the LLWPCB instruction is executed, or they can be read from the LWPCB each time the counter overflows.

The ring buffer base and size are cached in the hardware.

The MissedEvents value is a counter for an exceptional condition and is kept in memory.

The cached LWP state is refreshed from the LWPCB when LWP is enabled either explicitly via LLWPCB or implicitly when needed in ring 3 after LWP state is restored via XRSTOR.

Caching implies that software cannot reliably change sampling intervals or other cached state by modifying the LWPCB. The change might not be noticed by the LWP hardware. On the other hand, changing state in the LWPCB while LWP is running may change the operation at an unpredictable moment in the future if LWP context is saved and restored due to context switching. Software must stop and restart LWP to ensure that any changes reliably take effect.

[AMD Public Use]

13.4.8.4 Security

The operating system must ensure that information does not leak from one process to another or from the kernel to a user process. Hence, if it supports LWP at all, the operating system must ensure that the state of the LWP hardware is set appropriately when a context switch occurs and when a new process or thread is created. LWP state for a new thread can be initialized by executing XRSTOR with bit 62 of XSAVE.HEADER.XSTATE_BV set to 0 and the corresponding bit in EDX:EAX set to 1.

13.4.8.5 Interrupts

The LWP threshold interrupt vector number is specified in the LWP_CFG MSR. The operating system must assign a vector for LWP threshold interrupts and fill in the corresponding entry in the interrupt-descriptor table. Note that the LWP interrupt is not shared with the performance counter interrupt, since the system allows concurrent and independent use of those two mechanisms.

13.4.8.6 Memory Access During LWP Operation

When LWP needs to save an event record in the event ring buffer, it accesses the user memory containing the ring buffer and sometimes the memory containing the LWPCB. This causes a Page Fault (#PF) exception if those pages are not in memory.

A particular implementation of LWP has several ways to deal with page faults when storing an event record. These may include saving the event record in the XSAVE area and retrying the store later, re-executing the instruction, or discarding the event and reporting the next event of the appropriate type.

Note that this reinforces the notion that LWP is a sampling mechanism. Programs cannot rely on it to precisely capture every n^{th} instance of an event. It captures *approximately* every n^{th} instance.

13.4.8.7 Guidelines for Operating Systems

To support LWP, an operating system should follow the following guidelines. Most of these operations should be done on each core of a multi-core system.

System initialization

1. Use CPUID Fn0000_0000 to ensure that the system is running on an “Authentic AMD” processor, and then check CPUID Fn8000_0001_ECX[LWP] to ensure that the processor supports LWP.

   Alternatively, check CPUID Fn0000_000D_EDX_x0[30] to ensure that the system supports the LWP XSAVE area, indicating that the processor supports LWP.
2. Enable XSAVE operations by setting CR4[OSXSAVE].
3. Enable LWP by executing XSETBV to set bit 62 of XCR0.
4. Assign a unique interrupt vector number for LWP threshold interrupts and load the corresponding entry in the interrupt-descriptor table with the address of the interrupt handler. This handler should use some system-specific method to forward any threshold interrupts to the application.

[AMD Public Use]

5. Make LWP available by setting LWP_CFG. To enable all supported LWP features, set LWP_CFG[31:0] to the value returned by CPUID Fn8000_001C_EDX. Set LWP_CFG[COREID] to the APIC core number (or some other value unique to the core) and LWP_CFG[VECTOR] to the assigned interrupt vector number.

Thread support

- For each thread, allocate an XSAVE area that is at least as big as the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0). This is good practice for any system that supports XSAVE.
- When creating a new process or thread, execute XRSTOR with bit 62 of EDX:EAX set to 1 and bit 62 of XSAVE.HEADER.XSTATE_BV set to 0. This ensures that LWP is turned off for any new thread. Alternatively, use WRMSR to write 0 into LWP_CBADDR before starting the thread.
- When saving a running thread's context, execute XSAVE with bit 62 of EDX:EAX set to 1 to save the thread's LWP state. It takes almost no time or resources if the thread is not using LWP.
- When restoring a thread's context, execute XRSTOR with bit 62 of EDX:EAX set to 1. This restores the LWP state for the thread or disables LWP if the thread is not using it.
- When a thread exits or aborts, use WRMSR to store 0 into LWP_CBADDR. This ensures that LWP is turned off.

13.4.8.8 Summary of LWP State

LWP adds the following visible state to the AMD64 architecture:

- CPUID Fn8000_0001_ECX[LWP] (bit 15) to indicate LWP support.
- CPUID Fn8000_001C to indicate LWP features.
- Two new MSRs: LWP_CFG, LWP_CBADDR,.
- Four new instructions: LLWPCB, SLWPCB, LWPINS, and LWPVAL.
- Bit 62 in XCR0
- A new XSAVE area for LWP state.
- New fields for LWP state in the SVM and SMM context, whether in the VMCB and SMM save area or elsewhere.

See Section 3.3, "Processor Feature Identification," on page 73 for information on using the CPUID instruction to obtain information about processor capabilities.

[AMD Public Use]

14 Processor Initialization and Long Mode Activation

This chapter describes the hardware actions taken following a processor reset and the steps that must be taken to initialize processor resources and activate long mode. In some cases the actions required are implementation-specific with references made to the appropriate implementation-specific documentation.

14.1 Processor Initialization

System logic can initialize the processor in either of two ways. One method, called RESET, is usually initiated by the assertion of an external signal (typically designated RESET#). The other method, called INIT, is typically initiated by another processor by means of an INIT interprocessor interrupt (IPI). See “Interprocessor Interrupts (IPI)” on page 673 for more information.

Both initialization techniques place the processor in real mode and initialize processor resources to a known, consistent state from which software can begin execution. The processor begins execution when the RESET# pin is deasserted or the INIT state is exited.

The RESET method places the processor in a known state and prepares it to begin execution in real mode. The INIT method is similar except it does not modify the state of certain registers. See Section 14.1.3 on page 503 for a comparison of these initialization methods.

System logic ensures that the processor transitions through the RESET state whenever power is reapplied after a planned or unplanned interruption. A RESET can also be performed when power is stable. An INIT can be performed at any time after the processor is powered up.

14.1.1 Built-In Self Test (BIST)

An optional built-in self-test can be performed after the processor is reset. The mechanism for triggering the BIST is implementation-specific, and can be found in the hardware documentation for the implementation. The number of processor cycles BIST can consume before completing is also implementation-specific but typically consumes several million cycles.

BIST can be used by system implementations to assist in verifying system integrity, thereby improving system reliability, availability, and serviceability. The internal BIST hardware generally tests all internal array structures for errors. These structures can include (but are not limited to):

- All internal caches, including the tag arrays as well as the data arrays.
- All TLBs.
- Internal ROMs, such as the microcode ROM and floating-point constant ROM.
- Branch-prediction structures.

[AMD Public Use]

EAX is loaded with zero if BIST completes without detecting errors. If any hardware faults are detected during BIST, a non-zero value is loaded into EAX.

14.1.2 Clock Multiplier Selection

The internal processor clock runs at some multiple of the system clock. The processor-to-system clock multiple does not have to be fixed by a processor implementation but instead can be programmable through hardware or software, or some combination of the two. For information on selecting the processor-clock multiplier, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

14.1.3 Processor Initialization State

Table 14-1 shows the initial processor state following either RESET or INIT. Except as indicated, processor resources generally are set to the same value after either RESET or INIT.

Table 14-1. Initial Processor State

Processor Resource	Value After RESET	Value After INIT
CR0	0000_0000_6000_0010h	CD and NW are unchanged Bit 4 (reserved) = 1 All others = 0
CR2, CR3, CR4	0	
CR8	0	0
RFLAGS	0000_0000_0000_0002h	
EFER	0	
RIP	0000_0000_0000_FFF0h	
CS	Selector = F000h Base = 0000_0000_FFFF_0000h Limit = FFFFh Attributes = See Table 14-2 on page 505	
DS, ES, FS, GS, SS	Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2 on page 505	
GDTR, IDTR	Base = 0 Limit = FFFFh	
LDTR, TR	Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2 on page 505	
RAX	0 (non-zero if BIST is run and fails)	0

[AMD Public Use]

Table 14-1. Initial Processor State (continued)

Processor Resource	Value After RESET	Value After INIT
RDX	Family/Model/Stepping, including extended family and extended model—see "Processor Implementation Information" on page 506	
RBX, RCX, RBP, RSP, RDI, RSI, R8, R9, R10, R11, R12, R13, R14, R15	0	
x87 Floating-Point State	FPR0–FPR7 = 0 Control Word = 0040h Status Word = 0000h Tag Word = 5555h Instruction CS = 0000h Instruction Offset = 0 x87 Instruction Opcode = 0 Data-Operand DS = 0000h Data-Operand Offset = 0	Not modified
64-Bit Media State	MMX0–MMX7 = 0	Not modified
SSE State	XMM0–XMM15 = 0 MXCSR = 1F80h	Not modified
Memory-Type Range Registers	See "Memory-Typing MSRs" on page 768	Not modified
Machine-Check Registers	See "Machine-Check MSRs" on page 770	Not modified
DR0, DR1, DR2, DR3	0	
DR6	0000_0000_FFFF_0FF0h	
DR7	0000_0000_0000_0400h	
Time-Stamp Counter	0	Not modified
Performance-Monitor Resources	See "Performance-Monitoring MSRs" on page 772	Not modified
Other Model-Specific Registers	See "MSR Cross-Reference" on page 757	Not modified, except IbsBufferBase[IbsBufferEn] = 0
Instruction and Data Caches	Invalidated	Not modified
Instruction and Data TLBs		
APIC	Disabled, see Table 16-2 on page 663.	Enabled, see Table 16-2 on page 663.
SMRAM Base Address (SMBASE)	0003_0000h	Not modified
XCR0	0000_0000_0000_0001h	Not modified
PKRU	0000_0000h	Not modified

Table 14-2 on page 505 shows the initial state of the segment-register attributes (located in the hidden portion of the segment registers) following either RESET or INIT.

[AMD Public Use]

Table 14-2. Initial State of Segment-Register Attributes

Attribute		Value (Binary)	Description
G		0	Byte Granularity
D/B		0	16-Bit Segment
L (CS Only)		0	Legacy-Mode Segment
P		1	Segment is Present
DPL		00	Privilege-Level 0
S and Type	Code Segment	S = 1 Type = 1010	Executable/Readable Code Segment
	Data Segment	S = 1 Type = 0010	Read/Write Data Segment
	LDTR	S = 0 Type = 0010	LDT
	TR	S = 0 Type = 0011	Busy 16-Bit TSS

14.1.4 Multiple Processor Initialization

Following reset in multiprocessor configurations, the processors use a multiple-processor initialization protocol to negotiate which processor becomes the *bootstrap* processor. This bootstrap processor then executes the system initialization code while the remaining processors wait for software initialization to complete. For further information, see the documentation for particular implementations of the architecture.

14.1.5 Fetching the First Instruction

After a RESET or INIT, the processor is operating in 16-bit real mode. Normally within real mode, the code-segment base-address is formed by shifting the CS-selector value left four bits. The base address is then added to the value in EIP to form the physical address into memory. As a result, the processor can only address the first 1 Mbyte of memory when in real mode.

However, immediately following RESET or INIT, the CS-selector register is loaded with F000h, but the CS base-address is *not* formed by left-shifting the selector. Instead, the CS base-address is initialized to FFFF_0000h. EIP is initialized to FFF0h. Therefore, the first instruction fetched from memory is located at physical-address FFFF_FFF0h (FFFF_0000h + 0000_FFF0h).

The CS base-address remains at this initial value until the CS-selector register is loaded by software. This can occur as a result of executing a far jump instruction or call instruction, for example. When CS is loaded by software, the new base-address value is established as defined for real mode (by left shifting the selector value four bits).

[AMD Public Use]

14.2 Hardware Configuration

14.2.1 Processor Implementation Information

Software can read processor-identification information from the EDX register immediately following RESET or INIT. This information can be used to initialize software to perform processor-specific functions. The information stored in EDX is defined as follows:

- *Stepping ID (bits 3:0)*—This field identifies the processor-revision level.
- *Extended Model (bits 19:16) and Model (bits 7:4)*—These fields combine to differentiate processor models within a instruction family. For example, two processors may share the same microarchitecture but differ in their feature set. Such processors are considered different models within the same instruction family. This is a split field, comprising an extended-model portion in bits 19:16 with a legacy portion in bits 7:4
- *Extended Family (bits 27:20) and Family (bits 11:8)*—These fields combine to differentiate processors by their microarchitecture.

The CPUID instruction can be used to obtain the same information. This is done by executing CPUID with either function 1 or function 8000_0001h. Additional information about the processor and the features supported can be gathered using CPUID with other feature codes. See Section 3.3, “Processor Feature Identification,” on page 73 for additional information.

14.2.2 Enabling Internal Caches

Following a RESET (but not an INIT), all instruction and data caches are disabled, and their contents are invalidated (the MOESI state is set to the invalid state). Software can enable these caches by clearing the cache-disable bit (CR0.CD) to zero (RESET sets this bit to 1). Software can further refine caching based on individual pages and memory regions. Refer to “Cache Control Mechanisms” on page 209 for more information on cache control.

Memory-Type Range Registers (MTRRs). Following a RESET (but not an INIT), the MTRRdefType register is cleared to 0, which disables the MTRR mechanism. The variable-range and fixed-range MTRR registers are not initialized and are therefore in an undefined state. Before enabling the MTRR mechanism, the initialization software (usually platform firmware) must load these registers with a known value to prevent unexpected results. Clearing these registers, for example, sets memory to the uncacheable (UC) type.

14.2.3 Initializing Media and x87 Processor State

Some resources used by x87 floating-point instructions and media instructions must be initialized by software before being used. Initialization software can use the CPUID instruction to determine whether the processor supports these instructions, and then initialize their resources as appropriate.

x87 Floating-Point State Initialization. Table 14-3 on page 507 shows the differences between the initial x87 floating-point state following a RESET and the state established by the FINIT/FNINIT instruction. An INIT does not modify the x87 floating-point state. The initialization software can

[AMD Public Use]

execute an FINIT or FNINIT instruction to prepare the x87 floating-point unit for use by application software. The FINIT and FNINIT instructions have no effect on the 64-bit media state.

Table 14-3. x87 Floating-Point State Initialization

x87 Floating-Point Resource	RESET	FINIT/FNINIT Instructions
FPR0–FPR7	0	Not modified
Control Word	0040h • Round to nearest • Single precision • Unmask all exceptions	037Fh • Round to nearest • Extended precision • Mask all exceptions
Status Word	0000h	
Tag Word	5555h (FPR*n* contain zero)	FFFFh (FPR*n* are empty)
Instruction CS	0000h	
Instruction Offset	0	
x87 Instruction Opcode	0	
Data-Operand DS	0000h	
Data-Operand Offset	0	

Initialization software should also load the MP, EM, and NE bits in the CR0 register as appropriate for the operating system. The recommended settings are:

- *MP=1*—Setting MP to 1 causes a device-not-available exception (#NM) to occur when the FWAIT/WAIT instruction is executed and the task-switched bit (CR0.TS) is set to 1. This supports operating systems that perform lazy context-switching of x87 floating-point state.
- *EM=0*—Clearing EM to 0 allows the x87 floating-point unit to execute instructions rather than causing a #NM exception (CR0.EM=1). System software sets EM to 1 only when software emulation of x87 instructions is desired.
- *NE=1*—Setting NE to 1 causes x87 floating-point exceptions to be handled by the floating-point exception-pending exception (#MF) handler. Clearing this bit causes the processor to externally indicate the exception occurred, and an external device can then cause an external interrupt to occur in response.

Refer to “CR0 Register” on page 42 for additional information on these control bits.

64-Bit Media State Initialization. There are no special requirements placed on software to initialize the processor state used by 64-bit media instructions. This state is initialized completely by the processor following a RESET. System software should leave CR0.EM cleared to 0 to allow execution of the 64-bit media instructions. If CR0.EM is set to 1, attempted execution of the 64-bit media instructions causes an invalid-opcode exception (#UD).

The 64-bit media state is not modified by an INIT.

[AMD Public Use]

SSE State Initialization. Platform firmware or system software must also prepare the processor to allow execution of SSE instructions. The required preparations include:

- Leaving CR0.EM cleared to 0 to allow execution of the SSE instructions. If CR0.EM is set to 1, attempted execution of the SSE instructions except FXSAVE/FXRSTOR causes an invalid-opcode exception (#UD). An attempt to execute either of these instructions when CR0.EM is set results in a #NM exception.
- Enabling the SSE instructions by setting CR4.OSFXSR to 1. Software cannot execute the SSE instructions unless this bit is set. Setting this bit also indicates that system software uses the FXSAVE and FXRSTOR instructions to save and restore, respectively, the SSE state. These instructions also save and restore the 64-bit media state and x87 floating-point state.
- Indicating that system software uses the SIMD floating-point exception (#XF) for handling SSE floating-point exceptions. This is done by setting CR4.OSXMMEXCPT to 1.
- Setting (optionally) the MXCSR mask bits to mask or unmask SSE floating-point exceptions as desired. Because this register can be read and written by application software, it is not absolutely necessary for system software to initialize it.

Refer to “CR4 Register” on page 47 for additional information on these CR4 control bits.

14.2.4 Model-Specific Initialization

Implementations of the AMD64 architecture can contain model-specific features and registers that are not initialized by the processor and therefore require system-software initialization. System software must use the CPUID instruction to determine which features are supported. Model-specific features are generally configured using model-specific registers (MSRs), which can be read and written using the RDMSR and WRMSR instructions, respectively.

Some of the model-specific features are pervasive across many processor implementations of the AMD64 architecture and are therefore described within this volume. These include:

- System-call extensions, which must be enabled in the EFER register before using the SYSCALL and SYSRET instructions. See “System-Call Extension (SCE) Bit” on page 57 for information on enabling these instructions.
- Memory-typing MSRs. See “Memory-Type Range Registers (MTRRs)” on page 506 for information on initializing and using these registers.
- The machine-check mechanism. See “Initializing the Machine-Check Mechanism” on page 321 for information on enabling and using this capability.
- Extensions to the debug mechanism. See “Software-Debug Resources” on page 392 for information on initializing and using these extensions.
- The performance-monitoring resources. See “Performance Monitoring Counters” on page 413 for information on initializing and using these resources.

Initialization of other model-specific features used by the page-translation mechanism and long mode are described throughout the remainder of this section.

[AMD Public Use]

Some model-specific features are not pervasive across processor implementations and are therefore not described in this volume. For more information on these features and their initialization requirements, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

14.3 Initializing Real Mode

A basic real-mode (real-address-mode) operating environment must be initialized so that system software can initialize the protected-mode operating environment. This real-mode environment must include:

- A real-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in real mode. The IDT base-address value in the IDTR initialized by the processor can be used, or system software can relocate the IDT by loading a new base-address into the IDTR.
- The real-mode interrupt and exception handlers. These must be loaded before enabling external interrupts.

  Because the processor can always accept a non-maskable interrupt (NMI), it is possible an NMI can occur before initializing the IDT or the NMI handler. System hardware must provide a mechanism for disabling NMIs to allow time for the IDT and NMI handler to be properly initialized. Alternatively, the IDT and NMI handler can be stored in non-volatile memory that is referenced by the initial values loaded into the IDTR.

  Maskable interrupts can be enabled by setting EFLAGS.IF after the real-mode IDT and interrupt handlers are initialized.
- A valid stack pointer (SS:SP) to be used by the interrupt mechanism should interrupts or exceptions occur. The values of SS:SP initialized by the processor can be used.
- One or more data-segment selectors for storing the protected-mode data structures that are created in real mode.

Once the real-mode environment is established, software can begin initializing the protected-mode environment.

14.4 Initializing Protected Mode

Protected mode must be entered before activating long mode. A minimal protected-mode environment must be established to allow long-mode initialization to take place. This environment must include the following:

- A protected-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in protected mode.
- The protected-mode interrupt and exception handlers referenced by the IDT. Gate descriptors for each handler must be loaded in the IDT.

[AMD Public Use]

- A GDT which contains:
  - A code descriptor for the code segment that is executed in protected mode.
  - A read/write data segment that can be used as a protected-mode stack. This stack can be used by the interrupt mechanism if interrupts or exceptions occur.

Software can optionally load the GDT with one or more data segment descriptors, a TSS descriptor, and an LDT descriptor for use by long-mode initialization software.

After the protected-mode data structures are initialized, system software must load the IDTR and GDTR with pointers to those data structures. Once these registers are initialized, protected mode can be enabled by setting CR0.PE to 1.

If legacy paging is used during the long-mode initialization process, the page-translation tables must be initialized before enabling paging. At a minimum, one page directory and one page table are required to support page translation. The CR3 register must be loaded with the starting physical address of the highest-level table supported in the page-translation hierarchy. After these structures are initialized and protected mode is enabled, paging can be enabled by setting CR0.PG to 1.

14.5 Initializing Long Mode

From protected mode, system software can initialize the data structures required by long mode and store them anywhere in the first 4 Gbytes of physical memory. These data structures can be relocated above 4 Gbytes once long mode is activated. The data structures required by long mode include the following:

- An IDT with 64-bit interrupt-gate descriptors. Long-mode interrupts are always taken in 64-bit mode, and the 64-bit gate descriptors are used to transfer control to interrupt handlers running in 64-bit mode. See “Long-Mode Interrupt Control Transfers” on page 283 for more information.
- The 64-bit mode interrupt and exception handlers to be used in 64-bit mode. Gate descriptors for each handler must be loaded in the 64-bit IDT.
- A GDT containing segment descriptors for software running in 64-bit mode and compatibility mode, including:
  - Any LDT descriptors required by the operating system or application software.
  - A TSS descriptor for the single 64-bit TSS required by long mode.
  - Code descriptors for the code segments that are executed in long mode. The code-segment descriptors are used to specify whether the processor is operating in 64-bit mode or compatibility mode. See “Code-Segment Descriptors” on page 99, “Long (L) Attribute Bit” on page 100, and “CS Register” on page 81 for more information.
  - Data-segment descriptors for software running in compatibility mode. The DS, ES, and SS segments are ignored in 64-bit mode. See “Data-Segment Descriptors” on page 100 for more information.

[AMD Public Use]

- FS and GS data-segment descriptors for 64-bit mode, if required by the operating system. If these segments are used in 64-bit mode, system software can also initialize the full 64-bit base addresses using the WRMSR instruction. See “FS and GS Registers in 64-Bit Mode” on page 82 for more information.

The existing protected-mode GDT can be used to hold the long-mode descriptors described above.

- A single 64-bit TSS for holding the privilege-level 0, 1, and 2 stack pointers, the interrupt-stack-table pointers, and the I/O-redirection-bitmap base address (if required). This is the only TSS required, because hardware task-switching is not supported in long mode. See “64-Bit Task State Segment” on page 378 for more information.
- The 4-level page-translation tables required by long mode. Long mode also requires the use of physical-address extensions (PAE) to support physical-address sizes greater than 32 bits. See “Long-Mode Page Translation” on page 141 for more information.

If paging is enabled during the initialization process, it *must* be disabled before enabling long mode. After the long-mode data structures are initialized, and paging is disabled, software can enable and activate long mode.

14.6 Enabling and Activating Long Mode

Long mode is *enabled* by setting the long-mode enable control bit (EFER.LME) to 1. However, long mode is not *activated* until software also enables paging. When software enables paging while long mode is enabled, the processor activates long mode, which the processor indicates by setting the long-mode-active status bit (EFER.LMA) to 1. The processor behaves as a 32-bit x86 processor in all respects until long mode is activated, even if long mode is enabled. None of the new 64-bit data sizes, addressing, or system aspects available in long mode can be used until EFER.LMA=1.

Table 14-4 shows the control-bit settings for enabling and activating the various operating modes of the AMD64 architecture. The default address and data sizes are shown for each mode. For the methods of overriding these default address and data sizes, see “Instruction Prefixes” in Volume 1.

[AMD Public Use]

Table 14-4. Processor Operating Modes

Mode		Encoding			Default Address Size (bits)[2]	Default Data Size (bits)[2]
		EFER.LMA[1]	CS.L	CS.D		
Long Mode	64-Bit Mode	1	1	0	64	32
	Compatibility Mode		0	1	32	32
				0	16	16
Legacy Mode		0	x	1	32	32
				0	16	16

Note:
1. *EFER.LMA is set by the processor when software sets EFER.LME and CR0.PG according to the sequence described in "Activating Long Mode" on page 512.*
2. *See "Instruction Prefixes" in Volume 1 for overrides to default sizes.*

Long mode uses two code-segment-descriptor bits, CS.L and CS.D, to control the operating submodes. If long mode is active, CS.L = 1, and CS.D = 0, the processor is running in 64-bit mode, as shown in Table 14-4 on page 512. With this encoding (CS.L=1, CS.D=0), default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, the default operand size can be overridden to 64 bits or 16 bits, and the default address size can be overridden to 32 bits.

The final encoding of CS.L and CS.D in long mode (CS.L=1, CS.D=1) is reserved for future use.

When long mode is active and CS.L is cleared to 0, the processor is in compatibility mode, as shown in Table 14-4 on page 512. In compatibility mode, CS.D controls default operand and address sizes exactly as it does in the legacy x86 architecture. Setting CS.D to 1 specifies default operand and address sizes as 32 bits. Clearing CS.D to 0 specifies default operand and address sizes as 16 bits.

14.6.1 Activating Long Mode

Switching the processor to long mode requires several steps. In general, the sequence involves disabling paging (CR0.PG=0), enabling physical-address extensions (CR4.PAE=1), loading CR3, enabling long mode (EFER.LME=1), and finally enabling paging (CR0.PG=1).

Specifically, software must follow this sequence to activate long mode:

1. If starting from page-enabled protected mode, disable paging by clearing CR0.PG to 0. This requires that the MOV CR0 instruction used to disable paging be located in an identity-mapped page (virtual address equals physical address).

[AMD Public Use]

2. In any order:
   - Enable physical-address extensions by setting CR4.PAE to 1. Long mode requires the use of physical-address extensions (PAE) in order to support physical-address sizes greater than 32 bits. Physical-address extensions must be enabled before enabling paging.
   - Load CR3 with the physical base-address of the level-4 page-map-table (PML4) or level-5 page-map-table (PML5). See “Long-Mode Page Translation” on page 141 for details on creating the 4-level or 5-level page translation tables required by long mode.
   - Enable long mode by setting EFER.LME to 1.
3. Enable paging by setting CR0.PG to 1. This causes the processor to set the EFER.LMA bit to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

14.6.2 Consistency Checks

The processor performs long-mode consistency checks whenever software attempts to modify any of the control bits directly involved in activating long mode (EFER.LME, CR0.PG, and CR4.PAE). A general-protection exception (#GP) occurs when a consistency check fails. Long-mode consistency checks ensure that the processor does not enter an undefined mode or state that results in unpredictable behavior.

Long-mode consistency checks cause a general-protection exception (#GP) to occur if:

- An attempt is made to enable or disable long mode while paging is enabled.
- Long mode is enabled, and an attempt is made to enable paging before enabling physical-address extensions (PAE).
- Long mode is enabled, and an attempt is made to enable paging while CS.L=1.
- Long mode is active and an attempt is made to disable physical-address extensions (PAE).

Table 14-5 summarizes the long-mode consistency checks made during control-bit transitions.

Table 14-5. Long-Mode Consistency Checks

Control Bit	Transition	Check
EFER.LME	0 → 1	If (CR0.PG=1) then #GP(0)
	1 → 0	If (CR0.PG=1) then #GP(0)
CR0.PG	0 → 1	If ((EFER.LME=1) & (CR4.PAE=0) then #GP(0) If ((EFER.LME=1) & (CS.L=1)) then #GP(0)
CR4.PAE	1 → 0	If (EFER.LMA=1) then #GP(0)

[AMD Public Use]

14.6.3 Updating System Descriptor Table References

Immediately after activating long mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy descriptor tables. The tables referenced by these descriptors all reside in the lower 4 Gbytes of virtual-address space. After activating long mode, 64-bit operating-system software should use the LGDT, LLDT, LIDT, and LTR instructions to load the system descriptor-table registers with references to the 64-bit versions of the descriptor tables. See “Descriptor Tables” on page 84 for details on descriptor tables in long mode.

Long mode requires 64-bit interrupt-gate descriptors to be stored in the interrupt-descriptor table (IDT). Software must not allow exceptions or interrupts to occur between the time long mode is activated and the subsequent update of the interrupt-descriptor-table register (IDTR) that establishes a reference to the 64-bit IDT. This is because the IDTR continues to reference a 32-bit IDT immediately after long mode is activated. If an interrupt or exception occurred before updating the IDTR, a legacy 32-bit interrupt gate would be referenced and interpreted as a 64-bit interrupt gate, with unpredictable results.

External interrupts can be disabled using the CLI instruction. Non-maskable interrupts (NMI) and system-management interrupts (SMI) must be disabled using external hardware. See “Long-Mode Interrupt Control Transfers” on page 283 for more information on long mode interrupts.

14.6.4 Relocating Page-Translation Tables

The long-mode page-translation tables must be located in the first 4 Gbytes of physical-address space before activating long mode. This is necessary because the MOV CR3 instruction used to initialize the page-map level-4 base address must be executed in legacy mode before activating long mode. Because the MOV CR3 is executed in legacy mode, only the low 32 bits of the register are written, which limits the location of the page-map level-4 translation table to the low 4 Gbytes of memory. Software can relocate the page tables anywhere in physical memory, and re-initialize the CR3 register, after long mode is activated.

14.7 Leaving Long Mode

To return from long mode to legacy protected mode with paging enabled, software must deactivate and disable long mode using the following sequence:

1. Switch to compatibility mode and place the processor at the highest privilege level (CPL=0).
2. Deactivate long mode by clearing CR0.PG to 0. This causes the processor to clear the LMA bit to 0. The MOV CR0 instruction used to disable paging must be located in an identity-mapped page. Once paging is disabled, the processor behaves as a standard 32-bit x86 processor.
3. Load CR3 with the physical base-address of the legacy page tables.
4. Disable long mode by clearing EFER.LME to 0.

[AMD Public Use]

5. Enable legacy page-translation by setting CR0.PG to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

14.8 Long-Mode Initialization Example

Following is sample code that outlines the steps required to place the processor in long mode.

```
mydata  segment para
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This generic data-segment holds pseudo-descriptors used
; by the LGDT and LIDT instructions.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Establish a temporary 32-bit GDT and IDT.
;
pGDT32  label   fword                   ; Used by LGDT.
                dw      gdt32_limit     ; GDT limit ...
                dd      gdt32_base      ; and 32-bit GDT base
pIDT32  label   fword                   ; Used by LIDT.
                dw      idt32_limit     ; IDT limit ...
                dd      idt32_base      ; and 32-bit IDT base
;
; Establish a 64-bit GDT and IDT (64-bit linear base-
; address)
;
pGDT64  label   tbyte                   ; Used by LGDT.
                dw      gdt64_limit     ; GDT limit ...
                dq      gdt64_base      ; and 64-bit GDT base
pIDT64  label   tbyte                   ; Used by LIDT.
                dw      idt64_limit     ; IDT limit ...
                dq      idt64_base      ; and 64-bit IDT base
mydata  ends                ; end of data segment
code16  segment para use16 ; 16-bit code segment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 16-bit code, real mode
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Initialize DS to point to the data segment containing
; pGDT32 and PIDT32. Set up a real-mode stack pointer, SS:SP,
; in case of interrupts and exceptions.
;
   cli
   mov   ax, seg mydata
   mov   ds, ax
   mov   ax, seg mystack
   mov   ss, ax
```

[AMD Public Use]


```
   mov   sp, esp0
;
; Use CPUID to determine if the processor supports long mode. ;

   mov   eax, 80000000h ; Extended-function 8000000h.
   cpuid                ; Is largest extended function
   cmp   eax, 80000000h ; any function > 80000000h?
   jbe   no_long_mode   ; If not, no long mode.
   mov   eax, 80000001h ; Extended-function 8000001h.
   cpuid                ; Now EDX = extended-features flags.
   bt    edx, 29        ; Test if long mode is supported.
   jnc   no_long_mode   ; Exit if not supported.
;
; Load the 32-bit GDT before entering protected mode.
; This GDT must contain, at a minimum, the following
; descriptors:
;  1) a CPL=0 16-bit code descriptor for this code segment.
;  2) a CPL=0 32/64-bit code descriptor for the 64-bit code.
;  3) a CPL=0 read/write data segment, usable as a stack
;  (referenced by SS).
;
; Load the 32-bit IDT, in case any interrupts or exceptions
; occur after entering protected mode, but before enabling
; long mode).
;
; Initialize the GDTR and IDTR to point to the temporary
; 32-bit GDT and IDT, respectively.
;
   lgdt  ds:[pGDT32]
   lidt  ds:[pIDT32]
;
; Enable protected mode (CR0.PE=1).
;
   mov   eax, 000000011h
   mov   cr0, eax
;
; Execute a far jump to turn protected mode on.
; code16_sel must point to the previously-established 16-bit
; code descriptor located in the GDT (for the code currently
; being executed).
;
   db    0eah              ;Far jump...
   dw    offset now_in_prot;to offset...
   dw    code16_sel        ;in current code segment.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; At this point we are in 16-bit protected mode, but long
; mode is still disabled.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

[AMD Public Use]


```
now_in_prot:
;
; Set up the protected-mode stack pointer, SS:ESP.
; Stack_sel must point to the previously-established stack
; descriptor (read/write data segment), located in the GDT.
; Skip setting DS/ES/FS/GS, because we are jumping right to
; 64-bit code.
;
   mov   ax, stack_sel
   mov   ss, ax
   mov   esp, esp0
;
; Enable the 64-bit page-translation-table entries by
; setting CR4.PAE=1 (this is _required_ before activating
; long mode). Paging is not enabled until after long mode
; is enabled.
;
   mov   eax, cr4
   bts   eax, 5
   mov   cr4, eax
;
; Create the long-mode page tables, and initialize the
; 64-bit CR3 (page-table base address) to point to the base
; of the PML4 page table. The PML4 page table must be located
; below 4 Gbytes because only 32 bits of CR3 are loaded when
; the processor is not in 64-bit mode.
;
   mov   eax, pml4_base ; Pointer to PML4 table (<4GB).
   mov   cr3, eax       ; Initialize CR3 with PML4 base.
;
; Enable long mode (set EFER.LME=1).
;
   mov   ecx, 0c0000080h   ; EFER MSR number.
   rdmsr                   ; Read EFER.
   bts   eax, 8            ; Set LME=1.
   wrmsr                   ; Write EFER.
;
; Enable paging to activate long mode (set CR0.PG=1)
;
   mov   eax, cr0          ; Read CR0.
   bts   eax, 31           ; Set PE=1.
   mov   cr0, eax          ; Write CR0.
;
; At this point, we are in 16-bit compatibility mode
; ( LMA=1, CS.L=0, CS.D=0 ).
; Now, jump to the 64-bit code segment. The offset must be
; equal to the linear address of the 64-bit entry point,
; because 64-bit code is in an unsegmented address space.
; The selector points to the 32/64-bit code selector in the
; current GDT.
;
```

[AMD Public Use]


```
   db    066h
   db    0eah
   dd    start64_linear
   dw    code64_sel
code16ends     ; End of the 16-bit code segment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;;    Start of 64-bit code
;;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
code64  para use64
start64:       ; At this point, we're using 64-bit code
;
; Point the 64-bit RSP register to the stack's _linear_
; address. There is no need to set SS here, because the SS
; register is not used in 64-bit mode.
;
   mov   rsp, stack0_linear
;
; This LGDT is only needed if the long-mode GDT is to be
; located at a linear address above 4 Gbytes.  If the long
; mode GDT is located at a 32-bit linear address, putting
; 64-bit descriptors in the GDT pointed to by [pGDT32] is
; just fine. pGDT64_linear is the _linear_ address of the
; 10-byte GDT pseudo-descriptor.
;
; The new GDT should have a valid CPL0 64-bit code segment
; descriptor at the entry-point corresponding to the current
; CS selector. Alternatively, a far transfer to a valid CPL0
; 64-bit code segment descriptor in the new GDT must be done
; before enabling interrupts.
;
   lgdt  [pGDT64_linear]
;
; Load the 64-bit IDT. This is _required_, because the 64-bit
; IDT uses 64-bit interrupt descriptors, while the 32-bit
; IDT used 32-bit interrupt descriptors. pIDT64_linear is
; the _linear_ address of the 10-byte IDT pseudo-descriptor.
;
   lidt  [pIDT64_linear]
;
; Set the current TSS. tss_sel should point to a 64-bit TSS
; descriptor in the current GDT. The TSS is used for
; inner-level stack pointers and the IO bit-map.
;
   mov   ax, tss_sel
   ltr   ax
;
; Set the current LDT. ldt_sel should point to a 64-bit LDT
; descriptor in the current GDT.
```

[AMD Public Use]


```
;
   mov   ax, ldt_sel
   lldt  ax
;
; Using fs: and gs: prefixes on memory accesses still uses
; the 32-bit fs.base and gs.base.  Reload these 2 registers
; before using the fs: and gs: prefixes. FS and GS can be
; loaded from the GDT using a normal "mov fs,foo" type
; instructions, which loads a 32-bit base into FS or GS.
; Alternatively, use WRMSR to assign 64-bit base values to
; MSR_FS_base or MSR_GS_base.
;
   mov   ecx, MSR_FS_base
   mov   eax, FsbaseLow
   mov   edx, FsbaseHi
   wrmsr
;
; Reload CR3 if long-mode page tables are to be located above
; 4 Gbytes. Because the original CR3 load was done in 32-bit
; legacy mode, it could only load 32 bits into CR3. Thus, the
; current page tables are located in the lower 4 Gbytes of
; physical memory. This MOV to CR3 is only needed if the
; actual long-mode page tables should be located at a linear
; address above 4 Gbytes.
;
   mov   rax, final_pml4_base  ; Point to PML4
   mov   cr3, rax              ; Load 64-bit CR3
;
; Enable interrupts.
;
   sti                         ; Enabled INTR
   <insert 64-bit code here>
```

[AMD Public Use]

15 Secure Virtual Machine

The AMD Virtualization™ (AMD-V™) architecture is designed to support enterprise-class server virtualization software technology and facilitate virtualization development and deployment on any type of system, through the Secure Virtual Machine (SVM) extension. An SVM-enabled virtual machine architecture provides hardware resources that allow a single physical machine to run multiple operating systems efficiently, while maintaining secure, hardware-enforced isolation.

15.1 The Virtual Machine Monitor

A *virtual machine monitor* (VMM), also known as a *hypervisor*, consists of software that controls the execution of multiple *guest* operating systems on a single physical machine. The VMM provides each guest the appearance of full control over a complete computer system (memory, CPU, and all peripheral devices). The use of the term *host* refers to the execution context of the VMM. *World switch* refers to the operation of switching between the host and guest. A guest may have one or more virtual CPUs (vCPUs) managed by the guest OS, just as on a non-virtualized system, and a VMM may run any mix of vCPUs from the same or different guests on different logical processors simultaneously with no hardware-imposed constraints.

Fundamentally, VMMs work by *intercepting* and emulating in a safe manner sensitive operations in the guest (such as changing the page tables, which could give a guest access to memory it is not allowed to access, or accessing peripheral devices that are shared among multiple guests). The AMD SVM architecture provides hardware assists to improve performance and facilitate implementation of virtualization.

15.2 SVM Hardware Overview

SVM processor support provides a set of hardware extensions designed to enable economical and efficient implementation of virtual machine systems. Generally speaking, hardware support falls into two complementary categories: *virtualization* support and *security* support.

15.2.1 Virtualization Support

The AMD virtual machine architecture is designed to provide:

- A guest/host tagged TLB to reduce virtualization overhead
- External (DMA) access protection for memory
- Assists for interrupt handling, virtual interrupt support, and enhanced pause filter
- The ability to intercept selected instructions or events in the guest
- Mechanisms for fast world switch between VMM and guest

[AMD Public Use]

15.2.2 Guest Mode

This new processor mode is entered through the VMRUN instruction. When in guest mode, the behavior of some x86 instructions changes to facilitate virtualization.

The CPUID function numbers 4000_0000h–4000_00FFh have been reserved for software use. Hypervisors can use these function numbers to provide an interface to pass information from the hypervisor to the guest. This is similar to extracting information about a physical CPU by using CPUID. Hypervisors use the CPUID Fn 400000[FF:00] bit to denote a virtual platform.

Feature bit CPUID Fn0000_0001_ECX[31] has been reserved for use by hypervisors to indicate the presence of a hypervisor. Hypervisors set this bit to 1 and physical CPUs set this bit to zero. This bit can be probed by the guest software to detect whether they are running inside a virtual machine.

15.2.3 External Access Protection

Guests may be granted direct access to selected I/O devices. Hardware support is designed to prevent devices owned by one guest from accessing memory owned by another guest (or the VMM).

15.2.4 Interrupt Support

To facilitate efficient virtualization of interrupts, the following support is provided under control of VMCB flags:

Intercepting physical interrupt delivery. The VMM can request that physical interrupts cause a running guest to exit, allowing the VMM to process the interrupt.

Virtual interrupts. The VMM can inject virtual interrupts into the guest. Under control of the VMM, a virtual copy of the EFLAGS.IF interrupt mask bit, and a virtual copy of the APIC's task priority register are used transparently by the guest instead of the physical resources.

Sharing a physical APIC. SVM allows multiple guests to share a physical APIC with isolation of each guest's manipulation of APIC state from the other guests' views of their own APIC state, so that no guest can interfere with delivery of interrupts to another guest.

Direct interrupt delivery. On models that support it, the Advanced Virtual Interrupt Controller (AVIC) extension virtualizes the APIC's interrupt delivery functions. This provides for delivery of device or inter-processor interrupts directly to a target vCPU or vCPUs, which avoids the overhead of having the VMM determine interrupt routing and speeds up interrupt delivery. (see Section 15.29).

15.2.5 Restartable Instructions

SVM is designed to safely restart, with the exception of task switches, any intercepted instruction (either atomic or idempotent) after the intercept.

[AMD Public Use]

15.2.6 Security Support

To further support secure initialization and execution, SVM provides additional system support through a variety of extensions.

Attestation. The SKINIT instruction and associated system support (the Trusted Platform Module, or TPM) allow for verifiable startup of trusted software (such as a hypervisor, or a native operating system), based on secure hash comparison. (Section 15.27).

Encrypted memory. On models that support it, the Secure Encrypted Virtualization (SEV) and SEV Encrypted State (SEV-ES) extensions guard against inspection of guest memory and (for SEV-ES) guest register state by malicious hypervisor code, memory bus tracing or memory device removal through encryption of guest memory and register contents (Section 15.34 and Section 15.35).

Secure Nested Paging. On models that support it, the SEV-SNP extension provides additional protection for guest memory against malicious manipulation of address translation mechanisms by hypervisor code. (Section 15.36).

15.3 SVM Processor and Platform Extensions

SVM hardware extensions can be grouped into the following categories:

- State switch—VMRUN, VMSAVE, VMLOAD instructions, global interrupt flag (GIF), and instructions to manipulate the latter (STGI, CLGI). (Section 15.5, Section 15.5.2, Section 15.17)
- Intercepts—allow the VMM to intercept sensitive operations in the guest. (Section 15.7 through Section 15.14)
- Interrupt and APIC assists—physical interrupt intercepts, virtual interrupt support, APIC.TPR virtualization. (Section 15.17 and Section 15.21)
- SMM intercepts and assists (Section 15.22)
- External (DMA) access protection (Section 15.24)
- Nested paging support for two levels of address translation. (Section 15.25)
- Security—SKINIT instruction. (Section 15.27)

15.4 Enabling SVM

The VMRUN, VMLOAD, VMSAVE, CLGI, VMMCALL, and INVLPGA instructions can be used when the EFER.SVME is set to 1; otherwise, these instructions generate a #UD exception. The SKINIT and STGI instructions can be used when either the EFER.SVME bit is set to 1 or the feature flag CPUID Fn8000_0001_ECX[SKINIT] is set to 1; otherwise, these instructions generate a #UD exception.

Before enabling SVM, software should detect whether SVM can be enabled using the following algorithm:

[AMD Public Use]


```
if (CPUID Fn8000_0001_ECX[SVM] == 0)
  return SVM_NOT_AVAIL;

if (VM_CR.SVMDIS == 0)
  return SVM_ALLOWED;

if (CPUID Fn8000_000A_EDX[SVML]==0)
  return SVM_DISABLED_AT_BIOS_NOT_UNLOCKABLE
  // the user must change a platform firmware setting to enable SVM
else return SVM_DISABLED_WITH_KEY;
  // SVMLock may be unlockable; consult platform firmware or TPM to obtain the
key.
```

For more information on using the CPUID instruction to obtain processor capability information, see Section 3.3, “Processor Feature Identification,” on page 73.

15.5 VMRUN Instruction

The VMRUN instruction is the cornerstone of SVM. VMRUN takes, as a single argument, the physical address of a 4KB-aligned page, the *virtual machine control block* (VMCB), which describes a virtual machine (guest) to be executed. The VMCB contains:

- a list of instructions or events in the guest (e.g., write to CR3) to intercept,
- various control bits that specify the execution environment of the guest or that indicate special actions to be taken before running guest code, and
- guest processor state (such as control registers, etc.).

Note that VMRUN is not supported inside the SMM handler and the behavior is undefined.

15.5.1 Basic Operation

The VMRUN instruction has an implicit addressing mode of [rAX]. Software must load RAX (EAX in 32-bit mode) with the physical address of the VMCB, a 4-Kbyte-aligned page that describes a virtual machine to be executed. The portion of RAX used in forming the address is determined by the current effective address size.

The VMCB is accessed by physical address and should be mapped as writeback (WB) memory.

VMRUN is available only at CPL 0. A #GP(0) exception is raised if the CPL is greater than 0. Furthermore, the processor must be in protected mode and EFER.SVME must be set to 1, otherwise, a #UD exception is raised.

The VMRUN instruction saves some host processor state information in the host state-save area in main memory at the physical address specified in the VM_HSAVE_PA MSR; it then loads corresponding guest state from the VMCB state-save area. VMRUN also reads additional control bits from the VMCB that allow the VMM to flush the guest TLB, inject virtual interrupts into the guest, etc.

[AMD Public Use]

The VMRUN instruction then checks the guest state just loaded. If an illegal state has been loaded, the processor exits back to the host (Section 15.6).

Otherwise, the processor now runs the guest code until an intercept event occurs, at which point the processor suspends guest execution and resumes host execution at the instruction following the VMRUN. This is called a #VMEXIT and is described in detail in (Section 15.6).

VMRUN saves or restores a minimal amount of state information to allow the VMM to resume execution after a guest has exited. This allows the VMM to handle simple intercept conditions quickly. If additional guest state information must be saved or restored (e.g., to handle more complex intercepts or to switch to a different guest), the VMM must use the VMLOAD and VMSAVE instructions to handle the additional guest state (Section 15.5.2).

Saving Host State. To ensure that the host can resume operation after #VMEXIT, VMRUN saves at least the following host state information:

- CS.SEL, NEXT_RIP—The CS selector and rIP of the instruction following the VMRUN. On #VMEXIT the host resumes running at this address.
- RFLAGS, RAX—Host processor mode and the register used by VMRUN to address the VMCB.
- SS.SEL, RSP—Stack pointer for host.
- CR0, CR3, CR4, EFER—Paging/operating mode for host.
- IDTR, GDTR—The pseudo-descriptors. VMRUN does not save or restore the host LDTR.
- ES.SEL and DS.SEL.

Processor implementations may store only part or none of host state in the memory area pointed to by VM_HSAVE_PA MSR and may store some or all host state in hidden on-chip memory. Different implementations may choose to save the hidden parts of the host's segment registers as well as the selectors. For these reasons, software must not rely on the format or contents of the host state save area, nor attempt to change host state by modifying the contents of the host save area.

Loading Guest State. After saving host state, VMRUN loads the following guest state from the VMCB:

- CS, rIP—Guest begins execution at this address. The hidden state of the CS segment register is also loaded from the VMCB.
- RFLAGS, RAX.
- SS, RSP—Includes the hidden state of the SS segment register.
- CR0, CR2, CR3, CR4, EFER—Guest paging mode. Writing paging-related control registers with VMRUN does *not* flush the TLB since address spaces are switched. (Section 15.16.)
- INTERRUPT_SHADOW—This flag indicates whether the guest is currently in an interrupt lockout shadow; (Section 15.21.5).
- IDTR, GDTR.
- ES and DS—Includes the hidden state of the segment registers.

[AMD Public Use]

- DR6 and DR7—The guest's breakpoint state.
- V_TPR—The guest's virtual TPR.
- V_IRQ—The flag indicating whether a virtual interrupt is pending in the guest.
- CPL—If the guest is in real mode, the CPL is forced to 0; if the guest is in v86 mode, the CPL is forced to 3. Otherwise, the CPL saved in the VMCB is used.

Additional virtualization features may load and save additional guest state beyond what is listed above; see the section documenting each specific feature for details.

The processor checks the loaded guest state for consistency. If a consistency check fails while loading guest state, the processor performs a #VMEXIT. For additional information, see "Canonicalization and Consistency Checks" on page 526.

If the guest is in PAE paging mode according to the registers just loaded and nested paging is not enabled, the processor will also read the four PDPEs pointed to by the newly loaded CR3 value; setting any reserved bits in the PDPEs also causes a #VMEXIT.

It is possible for the VMRUN instruction to load a guest rIP that is outside the limit of the guest code segment or that is non-canonical (if running in long mode). If this occurs, a #GP fault is delivered inside the guest; the rIP falling outside the limit of the guest code segment is not considered illegal guest state.

After all guest state is loaded, and intercepts and other control bits are set up, the processor reenables interrupts by setting GIF to 1. It is assumed that VMM software cleared GIF some time before executing the VMRUN instruction, to ensure an atomic state switch.

Some processor models allow the VMM to designate certain guest VMCB fields as "clean," meaning that they haven't been modified relative to the current state of hardware. This allows the hardware to optimize execution of VMRUN. See Section 15.15 for details on which fields may be affected by this. The descriptions below assume all fields are loaded.

Control Bits. Besides loading guest state, the VMRUN instruction reads various control fields from the VMCB; most of these fields are not written back to the VMCB on #VMEXIT, since they cannot change during guest execution:

- TSC_OFFSET—an offset to add when the guest reads the TSC (time stamp counter). Guest writes to the TSC can be intercepted and emulated by changing the offset (without writing the physical TSC). This offset is cleared when the guest exits back to the host.
- V_INTR_PRIO, V_INTR_VECTOR, V_IGN_TPR—fields used to describe a virtual interrupt for the guest (see "Injecting Virtual (INTR) Interrupts" on page 556).
- V_INTR_MASKING—controls whether masking of interrupts (in EFLAGS.IF and TPR) is to be virtualized (Section 15.21).
- The address space ID (ASID) to use while running the guest.
- A field to control flushing of the TLB during a VMRUN (see Section 15.16).

[AMD Public Use]

- The intercept vectors describing the active intercepts for the guest. On exit from the guest, the internal intercept registers are cleared so no host operations will be intercepted.

The maximum ASID value supported by a processor is implementation specific. The value returned in EBX after executing CPUID Fn8000_000A is the number of ASIDs supported by the processor.

See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

Segment State in the VMCB. The segment registers are stored in the VMCB in a format similar to that for SMM: both base and limit are fully expanded; segment attributes are stored as 12-bit values formed by the concatenation of bits 55:52 and 47:40 from the original 64-bit (in-memory) segment descriptors; the descriptor “P” bit is used to signal NULL segments (P=0) where permissible and/or relevant. The loading of segment attributes from the VMCB (which may have been overwritten by software) may result in attribute bit values that are otherwise not allowed. However, only some of the attribute bits are actually observed by hardware, depending on the segment register in question:

- CS—D, L, P, and R.
- SS—B, P, E, W, and Code/Data
- DS, ES, FS, GS —D, P, DPL, E, W, and Code/Data.
- LDTR—P, S, and Type (LDT)
- TR—P, S, and Type (32- or 16-bit TSS)

NOTE: For the Stack Segment attributes, P is observed in legacy and compatibility mode. In 64-bit mode, P is ignored because all stack segments are treated as present.

The VMM should follow these rules when storing segment attributes into the VMCB:

- For NULL segments, set all attribute bits to zero; otherwise, write the concatenation of bits 55:52 and 47:40 from the original 64-bit (in-memory) segment descriptors.
- The processor reads the current privilege level from the CPL field in the VMCB. The CS.DPL will match the CPL field.
- When in virtual x86 or real mode, the processor ignores the CPL field in the VMCB and forces the values of 3 and 0, respectively.

When examining segment attributes after a #VMEXIT:

- Test the Present (P) bit to check whether a segment is NULL; note that CS and TR never contain NULL segments and so their P bit is ignored;
- Retrieve the CPL from the CPL field in the VMCB, not from any segment DPL.

Canonicalization and Consistency Checks. The VMRUN instruction performs consistency checks on guest state and #VMEXIT performs the appropriate subset of these consistency checks on host state. Illegal guest state combinations cause a #VMEXIT with error code VMEXIT_INVALID. The following conditions are considered illegal state combinations (note that some checks may be subject to VMCB Clean field settings, see below):

[AMD Public Use]

- EFER.SVME is zero.
- CR0.CD is zero and CR0.NW is set.
- CR0[63:32] are not zero.
- Any MBZ bit of CR3 is set.
- Any MBZ bit of CR4 is set.
- DR6[63:32] are not zero.
- DR7[63:32] are not zero.
- Any MBZ bit of EFER is set.
- EFER.LMA or EFER.LME is non-zero and this processor does not support long mode.
- EFER.LME and CR0.PG are both set and CR4.PAE is zero.
- EFER.LME and CR0.PG are both non-zero and CR0.PE is zero.
- EFER.LME, CR0.PG, CR4.PAE, CS.L, and CS.D are all non-zero.
- The VMRUN intercept bit is clear.
- The MSR or IOIO intercept tables extend to a physical address that is greater than or equal to the maximum supported physical address.
- Illegal event injection (Section 15.20).
- ASID is equal to zero.
- Any reserved bit is set in S_CET
- CR4.CET=1 when CR0.WP=0
- CR4.CET=1 and U_CET.SS=1 when EFLAGS.VM=1
- Any reserved bit set in U_CET (SEV-ES only):
  - VMRUN results in VMEXIT(INVALID)
  - VMEXIT forces reserved bits to 0

VMRUN can load a guest value of CR0 with PE = 0 but PG = 1, a combination that is otherwise illegal (see Section 15.19).

In addition to consistency checks, VMRUN and #VMEXIT canonicalize (i.e., sign-extend to bit 63):

- All base addresses in the segment registers that have been loaded.
- SSP
- ISST_ADDR
- PL0_SSP, PL1_SSP, PL2_SSP, PL3_SSP

Additional virtualization features may introduce additional consistency checks beyond what is listed above; see the section documenting each specific feature for details.

[AMD Public Use]

VMCB Clean field behavior: On processor models that support designation of clean fields, the final merged hardware state is used for consistency checks. This may include state from fields marked as clean, if the processor chooses to ignore the indication.

VMRUN and TF/RF Bits in EFLAGS. When considering interactions of VMRUN with the TF and RF bits in EFLAGS, one must distinguish between the behavior of host as opposed to that of the guest.

From the host point of view, VMRUN acts like a single instruction, even though an arbitrary number of guest instructions may execute before a #VMEXIT effectively completes the VMRUN. As a single host instruction, VMRUN interacts with EFLAGS.RF and EFLAGS.TF like ordinary instructions. EFLAGS.RF suppresses any potential instruction breakpoint match on the VMRUN, and EFLAGS.TF causes a #DB trap after the VMRUN completes on the host side (i.e., after the #VMEXIT from the guest). As with any normal instruction, completion of the VMRUN instruction clears the host EFLAGS.RF bit.

The value of EFLAGS.RF from the VMCB affects the first guest instruction. When VMRUN loads a guest value of 1 for EFLAGS.RF, that value takes effect and suppresses any potential (guest) instruction breakpoint on the first guest instruction. When VMRUN loads a guest value of 1 in EFLAGS.TF, that value does *not* cause a trace trap between the VMRUN and the first guest instruction, but rather *after* completion of the first guest instruction.

Host values of EFLAGS have no effect on the guest and guest values of EFLAGS have no effect on the host.

See also Section 15.7.1 regarding the value of EFLAGS.RF saved on #VMEXIT.

15.5.2 VMSAVE and VMLOAD Instructions

These instructions transfer additional guest register context, including hidden context that is not otherwise accessible, between the processor and a guest's VMCB for a more complete context switch than VMRUN and #VMEXIT perform. The system physical address of the VMCB is specified in rAX. When these operations are needed, VMLOAD would be executed as desired prior to executing a VMRUN, and VMSAVE at any desired point after a #VMEXIT.

The VMSAVE and VMLOAD instructions take the physical address of a VMCB in rAX. These instructions complement the state save/restore abilities of VMRUN instruction and #VMEXIT. They provide access to hidden processor state that software cannot otherwise access, as well as additional privileged state.

These instructions handle the following register state:

- FS, GS, TR, LDTR (including all hidden state)
- KernelGsBase
- STAR, LSTAR, CSTAR, SFMASK
- SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

[AMD Public Use]

Like VMRUN, these instructions are only available at CPL0 (otherwise causing a #GP(0) exception) and are only valid in protected mode with SVM enabled via EFER.SVME (otherwise causing a #UD exception).

15.6 #VMEXIT

When an intercept triggers, the processor performs a #VMEXIT (i.e., an exit from the guest to the host context).

On #VMEXIT, the processor:

- Disables interrupts by clearing the GIF, so that after the #VMEXIT, VMM software can complete the state switch atomically.
- Writes back to the VMCB the current guest state—the same subset of processor state as is loaded by the VMRUN instruction, including the V_IRQ, V_TPR, and the INTERRUPT_SHADOW bits.
- Saves the reason for exiting the guest in the VMCB's EXITCODE field; additional information may be saved in the EXITINFO1 or EXITINFO2 fields, depending on the intercept. Note that the contents of the EXITINFO1 and EXITINFO2 fields are undefined for intercepts where their use is not indicated.
- Clears all intercepts.
- Resets the current ASID register to zero (host ASID).
- Clears the V_IRQ and V_INTR_MASKING bits inside the processor.
- Clears the TSC_OFFSET inside the processor.
- Reloads the host state previously saved by the VMRUN instruction. The processor reloads the host's CS, SS, DS, and ES segment registers and, if required, re-reads the descriptors from the host's segment descriptor tables, depending on the implementation. The segment descriptor tables must be mapped as present and writable by the host's page tables. Software should keep the host's segment descriptor tables consistent with the segment registers when executing VMRUN instructions. Immediately after #VMEXIT, the processor still contains the guest value for LDTR. So for CS, SS, DS, and ES, the VMM must only use segment descriptors from the global descriptor table. (The VMSAVE instruction can be used for a more complete context switch, allowing the VMM to then load LDTR and other registers not saved by #VMEXIT with desired values; see Section 15.5.2 for details.) Any exception encountered while reloading the host segments causes a shutdown.
- If the host is in PAE mode, the processor reloads the host's PDPEs from the page table indicated by the host's CR3. If the PDPEs contain illegal state, the processor causes a shutdown.
- Forces CR0.PE = 1, RFLAGS.VM = 0.
- Sets the host CPL to zero.
- Disables all breakpoints in the host DR7 register.

[AMD Public Use]

- Checks the reloaded host state for consistency; any error causes the processor to shutdown. If the host's rIP reloaded by #VMEXIT is outside the limit of the host's code segment or non-canonical (in the case of long mode), a #GP fault is delivered inside the host.

15.7 Intercept Operation

Various instructions and events (such as exceptions) in the guest can be intercepted by means of control bits in the VMCB "VMCB Layout" on page 780. The two primary classes of intercepts supported by SVM are instruction and exception intercepts.

Exception intercepts. Exception intercepts are checked when normal instruction processing must raise an exception before resolving possible double-fault conditions and before attempting delivery of the exception (which includes pushing an exception frame, accessing the IDT, etc.).

For some exceptions, the processor still writes certain exception-specific registers even if the exception is intercepted. (See the descriptions in Section 15.12 and following for details.) When an external or virtual interrupt is intercepted, the interrupt is left pending.

When an intercept occurs while the guest is in the process of delivering a non-intercepted interrupt or exception using the IDT, SVM provides additional information on #VMEXIT (See Section 15.7.2).

Instruction intercepts. These occur at well-defined points in instruction execution—before the results of the instruction are committed, but ordered in an intercept-specific priority relative to the instruction's exception checks. Generally, instruction intercepts are checked after simple exceptions (such as #GP—when CPL is incorrect—or #UD) have been checked, but before exceptions related to memory accesses (such as page faults) and exceptions based on specific operand values. There are several exceptions to this guideline, e.g., the RSM instruction. Instruction breakpoints for the current instruction and pending data breakpoint traps from the previous instruction are designed to be checked before instruction intercepts.

15.7.1 State Saved on Exit

When triggered, intercepts write an EXITCODE into the VMCB identifying the cause of the intercept. The EXITINTINFO field signals whether the intercept occurred while the guest was attempting to deliver an interrupt or exception through the IDT; a VMM can use this information to transparently complete the delivery (Section 15.20). Some intercepts provide additional information in the EXITINFO1 and EXITINFO2 fields in the VMCB; see the individual intercept descriptions for details.

The guest state saved in the VMCB is the processor state as of the moment the intercept triggers. In the x86 architecture, traps (as opposed to faults) are detected and delivered after the instruction that triggered them has completed execution. Accordingly, a trap intercept takes place after the execution of the instruction that triggered the trap in the first place. The saved guest state thus includes the effects of executing that instruction.

[AMD Public Use]

Example: Assume a guest instruction triggers a data breakpoint (#DB) trap which is in turn intercepted. The VMCB records the guest state after execution of that instruction, so that the saved CS:rIP points to the following instruction, and the saved DR7 includes the effects of matching the data breakpoint.

The next sequential instruction pointer (nRIP) is saved in the guest VMCB control area at location C8h on all #VMEXITs that are due to instruction intercepts, as defined in Section 15.9, as well as MSR and IOIO intercepts and exceptions caused by the INT3, INTO, and BOUND instructions. For all other intercepts, nRIP is reset to zero.

The nRIP is the RIP that would be pushed on the stack if the current instruction were subject to a trap-style debug exception, if the intercepted instruction were to cause no change in control flow. If the intercepted instruction would have caused a change in control flow, the nRIP points to the next sequential instruction rather than the target instruction.

Some exceptions write special registers even when they are intercepted; see the individual descriptions in Section 15.12 for details.

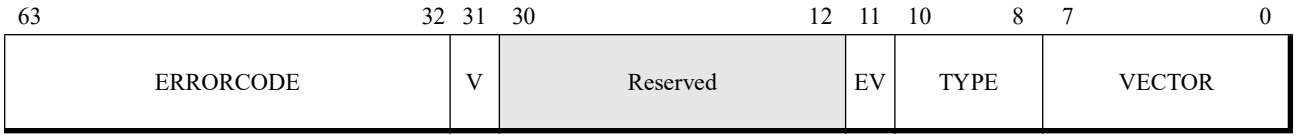
Support for the NRIP save on #VMEXIT is indicated by CPUID Fn8000_000A_EDX[NRIPS]. See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

15.7.2 Intercepts During IDT Interrupt Delivery

It is possible for an intercept to occur while the guest is attempting to deliver an exception or interrupt through the IDT (e.g., #PF because the VMM has paged out the guest’s exception stack). In some cases, such an intercept can result in the loss of information necessary for transparent resumption of the guest. In the case of an external interrupt, for example, the processor will already have performed an interrupt acknowledge cycle with the PIC or APIC to obtain the interrupt type and vector, and the interrupt is thus no longer pending.

To recover from such situations, all intercepts indicate (in the EXITINTINFO field in the VMCB) whether they occurred during exception or interrupt delivery though the IDT. This mechanism allows the VMM to complete the intercepted interrupt delivery, even when it is no longer possible to recreate the event in question.

[AMD Public Use]



Bits	Mnemonic	Description
63:32	ERRORCODE	Error Code
31	V	Valid
30:12	—	Reserved
11	EV	Error Code Valid
10:8	TYPE	Qualifies the guest exception or interrupt. Table 15-1 shows possible values returned and their corresponding interrupt or exception types. Values not indicated are unused and reserved.
7:0	VECTOR	8-bit IDT vector of the interrupt or exception.

Figure 15-1. EXITINTINFO

Table 15-1. Guest Exception or Interrupt Types

Value	Type
0	External or virtual interrupt (INTR)
2	NMI
3	Exception (fault or trap)
4	Software interrupt (caused by INT*n* instruction)

Despite the instruction name, the events raised by the INT1 (also known as ICEBP), INT3 and INTO instructions (opcodes F1h, CCh and CEh) are considered exceptions for the purposes of EXITINTINFO, not software interrupts. Only events raised by the INT*n* instruction (opcode CDh) are considered software interrupts.

- Error Code Valid—Bit 11. Set to 1 if the guest exception would have pushed an error code; otherwise cleared to zero.
- Valid—Bit 31. Set to 1 if the intercept occurred while the guest attempted to deliver an exception through the IDT; otherwise cleared to zero.
- Errorcode—Bits 63:32. If EV is set to 1, holds the error code that the guest exception would have pushed; otherwise is undefined.

In the case of multiple exceptions, EXITINTINFO records the aggregate information on all exceptions but the last (intercepted) one.

Example: A guest raises a #GP during delivery of which a #NP is raised (a scenario that, according to x86 rules, resolves to a #DF), and an intercepted #PF occurs during the attempt to deliver the #DF. Upon intercept of the #PF, EXITINTINFO indicates that the guest was in the process of delivering a #DF when the #PF occurred. The information about the intercepted page fault itself is encoded in the

[AMD Public Use]

EXITCODE, EXITINFO1 and EXITINFO2 fields. If the VMM decides to repair and dismiss the #PF, it can resume guest execution by re-injecting (see Section 15.20) the fault recorded in EXITINTINFO. If the VMM decides that the #PF should be reflected back to the guest, it must combine the event in EXITINTINFO with the intercepted exception according to x86 rules. In this case, a #DF plus a #PF would result in a triple fault or shutdown.

15.7.3 EXITINTINFO Pseudo-Code

When delivering exceptions or interrupts in a guest, the processor checks for exception intercepts and updates the value of EXITINTINFO should an intercept occur during exception delivery. The following pseudo-code outlines how the processor delivers an event (exception or interrupt) E.

```
if E is an exception and is intercepted:
    #VMEXIT(E)
E = (result of combining E with any prior events)

if (result was #DF and #DF is intercepted):
    #VMEXIT(#DF)
if (result was shutdown and shutdown is intercepted):
    #VMEXIT(#shutdown)
EXITINTINFO = E // Record the event the guest is delivering.

Attempt delivery of E through the IDT
Note that this may cause secondary exceptions

Once an exception has been successfully taken in the guest:

EXITINTINFO.V = 0 // Delivery succeeded; no #VMEXIT.
Dispatch to first instruction of handler
```

When an exception triggers an intercept, the EXITCODE, and optionally EXITINFO1 and EXITINFO2, fields always reflect the intercepted exception, while EXITINTINFO, if marked valid, indicates the prior exception the guest was attempting to deliver when the intercept occurred.

15.8 Decode Assists

Decode assists are provided to allow hypervisors to decode guest instructions more efficiently. CPUID Fn8000_000A_EDX[DecodeAssists] = 1 indicates support for this feature. See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

15.8.1 MOV CR*x*/DR*x* Intercepts

The EXITINFO1 field holds a flag indicating whether the instruction was a MOV CRx and the number of the GPR operand. MOV-to-CR instructions always set bit 63 and provide the GPR number, except for CR0 as specified below.

[AMD Public Use]

Table 15-2. EXITINFO1 for MOV CR*x*

Bit Offsets	Field Contents
3:0	GPR number
62:4	0
63	Instruction was MOV CR*x*—set to1 if the instruction was a MOV CR*x* instruction; cleared to 0 otherwise.

Table 15-3. EXITINFO1 for MOV DR*x*

Bit Offsets	Field Contents
3:0	GPR number
63:4	0

MOV-to-CR0 Special Case. If the instruction is MOV-to-CR, the GPR number is provided. If the instruction is LMSW or CLTS, no additional information is provided and bit 63 is not set.

MOV-from-CR0 Special Case. If the instruction is MOV-from-CR, the GPR number is provided and bit 63 is set. If the instruction is SMSW, no information is provided and bit 63 is not set.

15.8.2 INT*n* Intercepts

EXITINFO1 records the immediate value of the interrupt number for INT *n* instructions. See Table 15-4.

Table 15-4. EXITINFO1 for INT*n*

Bit Offsets	Field Contents
7:0	Software interrupt number
63:8	0

15.8.3 INVLPG and INVLPGA Intercepts

For an INVLPG intercept, EXITINFO1 provides the linear address after segment base addition and address size masking produce the effective address size. See Table 15-5. For an INVLPGA intercept, the linear address is available directly from the guest rAX register and is not provided in EXITINFO1.

Table 15-5. EXITINFO1 for INVLPG

Bit Offsets	Field Contents
63:0	Linear address

[AMD Public Use]

15.8.4 Nested and Intercepted #PF

In the case of a Nested Page Fault or intercepted #PF, guest instruction bytes at guest CS:RIP are stored into the 16-byte wide field Guest Instruction Bytes located at offset 0D0h in the VMCB. The format of this field is summarized in Table 15-6 below. Up to 15 bytes are recorded, read from guest CS:RIP. If a faulting condition occurs, such as not-present page or exceeding the CS limit, then the Guest Instruction Bytes field records as many bytes as could be fetched. The number of bytes fetched is put into the first byte of this field. Zero indicates that no bytes were fetched. The default number of bytes is always 15. Fewer bytes are returned only if a fault occurs while fetching.

This field is filled in only during data page faults. Instruction-fetch page faults provide no additional information.

All other intercepts clear bits 7:0 in this field to zero (to indicate an invalid condition); implementations may leave the other bytes untouched.

Table 15-6. Guest Instruction Bytes

Bit Offsets	Field Contents
3:0	Number of bytes fetched
7:4	0
127:8	Instruction bytes

15.9 Instruction Intercepts

Table 15-7 specifies the instructions that check a given intercept and, where relevant, how the intercept is prioritized relative to exceptions.

[AMD Public Use]

Table 15-7. Instruction Intercepts

Instruction Intercept	Checked By	Priority
Read/Write of CR0	MOV TO/FROM CR0, LMSW, SMSW, CLTS	Checks non-memory exceptions (CPL, illegal bit combinations, etc.) before the intercept. For LMSW and SMSW, checks SVM intercepts before checking memory exceptions.
Read/Write of CR3 (excluding task switch)	MOV TO/FROM CR3 (not checked by task switch operations)	Checks non-memory exceptions first, then the intercept. If the intercept triggers on a write, the intercept happens *before* the TLB is flushed. If PAE is enabled, the loading of the four PDPEs can cause a #GP; that exception is checked *after* the intercept check, so the VMM handling a CR3 intercept cannot rely on the PDPEs being legal; it must examine them in software if necessary. The reads and writes of CR3 that occur in VMRUN, #VMEXIT or task switches are *not* subject to this intercept check.
Read/Write of other CRs	MOV TO/FROM CR*n*	All normal exception checks take precedence over the SVM intercepts.
Read/Write of Debug Registers, DR*n*	MOV TO/FROM DR*n*. (Not checked by implicit DR6/DR7 writes.)	All normal exception checks take precedence over the SVM intercepts.
Selective CR0 Write Intercept	MOV TO CR0, LMSW	Checks non-memory exceptions (CPL, illegal bit combinations, etc.) before the intercept. For LMSW and SMSW, checks SVM intercepts before checking memory exceptions. The selective write intercept on CR0 triggers only if a bit other than CR0.TS or CR0.MP is being changed by the write. In particular, this means that CLTS does not check this intercept. When both selective and non-selective CR0-write intercepts are active at the same time, the non-selective intercept takes priority. With respect to exceptions, the priority of this intercept is the same as the generic CR0-write intercept. The LMSW instruction treats the selective CR0-write intercept as a non-selective intercept (i.e., it intercepts regardless of the value being written).
Reading or Writing IDTR, GDTR, LDTR, TR	LIDT, SIDT, LGDT, SGDT, LLDT, SLDT, LTR, STR	The SVM intercept is checked after #UD and #GP exception checks, but before any memory access is performed.
RDTSC	RDTSC	Checks all exceptions before the SVM intercept.
RDPMC	RDPMC	Checks all exceptions before the SVM intercept.
PUSHF	PUSHF	The intercept takes priority over any exceptions.
POPF	POPF	The intercept takes priority over any exceptions.
CPUID	CPUID	The intercept takes priority over any exceptions.

[AMD Public Use]

Table 15-7. Instruction Intercepts (continued)

Instruction Intercept	Checked By	Priority
RSM	RSM	The intercept takes priority over any exceptions.
IRET	IRET	The intercept takes priority over any exceptions.
Software Interrupt	INT*n*	The intercept occurs before any exceptions are checked. The CS:rIP reported on #VMEXIT are those of the intercepted INTn instruction. Though the INT*n* instruction may dispatch through IDT vectors in the range of 0–31, those events cannot be intercepted by means of exception intercepts (see “Exception Intercepts” on page 541).
INVD	INVD	Exceptions (#GP) are checked before the intercept.
PAUSE	PAUSE	No exceptions to check. VMRUN copies the VMCB.PauseFilterCount into an internal counter. Each PAUSE instruction decrements the counter, and the PAUSE intercept only occurs if the counter goes below zero while the PAUSE intercept is enabled. The VMCB.PauseFilterCount field is not written by the processor. Certain events, including SMI, can cause the internal count to be reloaded from the VMCB. VMCB.PauseFilterCount support is indicated by EDX[10] as returned by CPUID extended function 8000_000A. If This feature is not supported or VMCB.PauseFilterCount = 0, then the first PAUSE instruction can be intercepted.
HLT	HLT	Checks all exceptions before checking for this intercept.
IDLE_HLT	HLT	This intercept occurs only if a virtual interrupt is not pending (V_INTR or V_NMI). When both HLT and Idle HLT intercepts are active at the same time, the HLT intercept takes priority.
INVLPG	INVLPG	Checks all exceptions (#GP) before the intercept.
INVLPGA	INVLPGA	Checks all exceptions (#GP) before the intercept.
VMRUN	VMRUN	Checks exceptions (#GP) before the intercept. *The current implementation requires that the VMRUN intercept always be set in the VMCB.*
VMLOAD	VMLOAD	Checks exceptions (#GP) before the intercept.
VMSAVE	VMSAVE	Checks exceptions (#GP) before the intercept.
VMMCALL	VMMCALL	The intercept takes priority over exceptions. VMMCALL causes #UD in the guest if it is not intercepted.
STGI	STGI	Checks exceptions (#GP) before the intercept.
CLGI	CLGI	Checks exceptions (#GP) before the intercept.
SKINIT	SKINIT	Checks exceptions (#GP) before the intercept.

[AMD Public Use]

Table 15-7. Instruction Intercepts (continued)

Instruction Intercept	Checked By	Priority
RDTSCP	RDTSCP	Checks all exceptions before the SVM intercept.
ICEBP	ICEBP(opcode F1h).	Although the ICEBP instruction dispatches through IDT vector 1, that event is not interceptable by means of the #DB exception intercept.
WBINVD	WBINVD, WBNOINVD	Checks exceptions (#GP) before the intercept.
MONITOR	MONITOR, MONITORX	Checks all exceptions before the intercept.
MWAIT	MWAIT, MWAITX	Checks all exceptions before the intercept. There are conditional and unconditional MWAIT intercepts. The conditional MWAIT intercept is checked before the unconditional MWAIT intercept. When both conditional and unconditional MWAIT intercepts are active, the conditional intercept is checked first. A hypervisor that sets both intercepts will receive the conditional MWAIT intercept exit code for a guest MWAIT instruction that would have entered a low-power state, and will receive the unconditional MWAIT intercept exit code for a guest MWAIT instruction that would not have entered the low-power state. These checks also apply to MWAITX.
XSETBV	XSETBV	Checks intercept before exceptions (#GP).
RDPRU	RDPRU	Check all exceptions before the intercept.
INVLPGB	INVLPGB	Intercept takes priority over all exceptions except #GP for CPL<>0.
INVLPGB_ILLEGAL	INVLPGB exception cases	Intercept takes priority over all exceptions except #GP for CPL<>0.
INVPCID	INVPCID	Intercept takes priority over all exceptions except #GP for CPL<>0.
TLBSYNC	TLBSYNC	Checks exceptions (#GP) before the intercept.

15.10 IOIO Intercepts

The VMM can intercept IOIO instructions (IN, OUT, INS, OUTS) on a port-by-port basis by means of the SVM I/O permissions map.

15.10.1 I/O Permissions Map

The I/O Permissions Map (IOPM) occupies 12 Kbytes of contiguous physical memory. The map is structured as a linear array of 64K+3 bits (two 4-Kbyte pages, and the first three bits of a third 4-Kbyte page) and must be aligned on a 4-Kbyte boundary; the physical base address of the IOPM is specified in the IOPM_BASE_PA field in the VMCB and loaded into the processor by the VMRUN instruction. The VMRUN instruction ignores the lower 12 bits of the address specified in the VMCB. If the

[AMD Public Use]

address of the last byte in the IOPM is greater than or equal to the maximum supported physical address, this is treated as illegal VMCB state and causes a #VMEXIT(VMEXIT_INVALID).

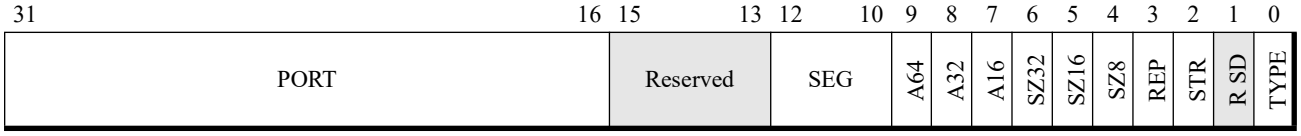
Each bit in the IOPM corresponds to an 8-bit I/O port. Bit 0 in the table corresponds to I/O port 0, bit 1 to I/O port 1 and so on. A bit set to 1 indicates that accesses to the corresponding port should be intercepted. The IOPM is accessed by physical address, and should reside in memory that is mapped as writeback (WB).

15.10.2 IN and OUT Behavior

If the IOIO_PROT intercept bit is set, the IOPM controls port access. For IN/OUT instructions that access more than a single byte, the permission bits for all bytes are checked; if any bit is set to 1, the I/O operation is intercepted.

Exceptions related to virtual x86 mode, IOPL, or the TSS-bitmap are checked *before* the SVM intercept check. All other exceptions are checked *after* the SVM intercept check.

I/O Intercept Information. When an IOIO intercept triggers, the following information (describing the intercepted operation in order to facilitate emulation) is saved in the VMCB's EXITINFO1 field:



Bits	Mnemonic	Description
31:16	PORT	Intercepted I/O port
15-13	—	Reserved
12:10	SEG	Effective segment number
9	A64	64-bit address
8	A32	32-bit address
7	A16	16-bit address
6	SZ32	32-bit operand size
5	SZ16	16-bit operand size
4	SZ8	8-bit operand size
3	REP	Repeated port access
2	STR	String based port access (INS, OUTS)
1	—	Reserved
0	TYPE	Access Type (0 = OUT instruction, 1 = IN instruction)

Figure 15-2. EXITINFO1 for IOIO Intercept

The rIP of the instruction *following* the IN/OUT is saved in EXITINFO2, so that the VMM can easily resume the guest after I/O emulation.

[AMD Public Use]

15.10.3 (REP) OUTS and INS

Bits 12:10 of the EXITINFO1 field provide the effective segment number (the default segment is DS). (For segment register encodings, see Table A-32, “16-Bit Register and Memory References” on page 478, in *AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions.*)

INS provides the effective segment (always ES, encoded as 0).

On intercepted SMI-on-I/O, bits 12:10 of EXITINFO1 encode the segment. For definitions of the remaining bits of this field, (Section 15.13.3).

15.11 MSR Intercepts

The VMM can intercept RDMSR and WRMSR instructions by means of the *SVM MSR permissions map* (MSRPM) on a per-MSR basis.

MSR Permissions Map. The MSR permissions bitmap consists of four separate bit vectors of 16 Kbits (2 Kbytes) each. Each 16 Kbit vector controls guest access to a defined range of 8K MSRs. Each MSR is covered by two bits defining the guest read and write access permissions. The lsb of the two bits controls read access to the MSR and the msb controls write access. A value of 1 indicates that the operation is intercepted. The four separate bit vectors must be packed together and located in two contiguous physical pages of memory. If the MSR_PROT intercept is active, any attempt to read or write an MSR not covered by the MSRPM will automatically cause an intercept.

The following table defines the ranges of MSRs covered by the MSR permissions map. Note that the MSR ranges are not contiguous.

Table 15-8. MSR Ranges Covered by MSRPM

MSRPM Byte Offset	MSR Range
000h–7FFh	0000_0000h–0000_1FFFh
800h–FFFh	C000_0000h–C000_1FFFh
1000h–17FFh	C001_0000h–C001_1FFFh
1800h–1FFFh	Reserved

The MSRPM is accessed by physical address and should reside in memory that is mapped as writeback (WB). The MSRPM must be aligned on a 4KB boundary. The physical base address of the MSRPM is specified in MSRPM_BASE_PA field in the VMCB and is loaded into the processor by the VMRUN instruction. The VMRUN instruction ignores the lower 12 bits of the address specified in the VMCB, and if the address of the last byte in the table is greater than or equal to the maximum supported physical address, this is treated as illegal VMCB state and causes a #VMEXIT(VMEXIT_INVALID).

[AMD Public Use]

RDMSR and WRMSR Behavior. If the MSR_PROT bit in the VMCB's intercept vector is clear, RDMSR/WRMSR instructions are not intercepted.

RDMSR and WRMSR instructions check for exceptions and intercepts in the following order:

- Exceptions common to all MSRs (e.g., #GP if not at CPL 0)
- Check SVM intercepts in the MSR permission map, if the MSR_PROT intercept is requested.
- Exceptions specific to a given MSR (including password protection, unimplemented MSRs, reserved bits, etc.)

MSR Intercept Information. On #VMEXIT, the processor indicates in the VMCB's EXITINFO1 whether a RDMSR (EXITINFO1 = 0) or WRMSR (EXITINFO1 = 1) was intercepted.

15.12 Exception Intercepts

When intercepting exceptions that define an error code (normally pushed onto the exception stack), the SVM hardware delivers that error code in the VMCB's EXITINFO1 field; the exception vector number can be derived from the EXITCODE. The CS.SEL and rIP saved in the VMCB on an exception-intercept match those that would otherwise have been pushed onto the exception stack frame, except that when an interrupt-based instruction causes an intercept, the rIP of the instruction is stored in the VMCB, rather than the rIP of the next instruction. The interrupt-based instructions are INT3 (opcode CC), INTO, and BOUND.

Unless otherwise noted below, no special registers are written before an exception is intercepted. For details on guest state saved in the VMCB, see Section 15.7.1.

External interrupts and software interrupts (INT*n* instruction) do not check the exception intercepts, even when they use a vector in the range 0 to 31.

Exceptions that occur during the handling of a prior exception are checked for intercepts *before* being combined with the prior exception (e.g., into a double-fault). If the result of combining exceptions is a double-fault or shutdown, the processor checks whether those are intercepted before attempting delivery.

Example: Assume that the VMM intercepts #GP and #DF exceptions, and the guest raises a (non-intercepted) #NP, during the delivery of which it also gets a #GP (e.g., due to an illegal IDT entry)—a situation that, according to x86 semantics, results in a #DF. In this case, #VMEXIT signals an intercepted #GP, *not* an intercepted #DF and fills EXITINTINFO with the #NP fault. On the other hand, if only the #DF intercept were active in this scenario, #VMEXIT would signal an intercepted #DF.

The following subsections detail the individual intercepts.

15.12.1 #DE (Divide By Zero)

The EXITINFO1 and EXITINFO2 fields are undefined.

[AMD Public Use]

15.12.2 #DB (Debug)

The #DB exception can have fault-type (e.g., instruction breakpoint) or trap-type (e.g., data breakpoint) behavior; accordingly the intercept differs in what state is saved in the VMCB (see Section 15.7.1). In either case, however, the value saved for DR6 and DR7 matches what would be visible to a #DB exception handler (i.e., both #DB faults and traps are permitted to write DR6 and DR7 before the intercept). The EXITINFO1 and EXITINFO2 fields are undefined.

Fault-type #DB exceptions, whether indicated in EXITCODE or EXITINTINFO, cause the CS:rIP saved in the VMCB to indicate the instruction that caused the #DB exception. Trap-type #DB exceptions cause the VMCB's CS:rIP to indicate the instruction following the instruction that caused the exception. A vector 1 exception generated by the single byte INT1 instruction (also known as ICEBP) does not trigger the #DB intercept. Software should use the dedicated ICEBP intercept to intercept ICEBP (see Section 15.9).

15.12.3 Vector 2 (Reserved)

This intercept bit is not implemented; use the NMI intercept (Section 15.13.2) instead. The effect of setting this bit is undefined.

15.12.4 #BP (Breakpoint)

This intercept applies to the trap raised by the single byte INT3 (opcode CCh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined. The CS:rIP reported on #VMEXIT are those of the INT3 instruction.

15.12.5 #OF (Overflow)

This intercept applies to the trap raised by the INTO (opcode CEh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.6 #BR (Bound-Range)

This intercept applies to the fault raised by the BOUND instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.7 #UD (Invalid Opcode)

The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.8 #NM (Device-Not-Available)

The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.9 #DF (Double Fault)

The EXITINFO1 and EXITINFO2 fields are undefined. The rIP value saved in the VMCB is undefined (as is the case for the rIP value pushed on the stack for #DF exceptions). If a double fault is

[AMD Public Use]

intercepted, the exceptions leading up to the double fault will have written any status registers normally written by those exceptions.

15.12.10 Vector 9 (Reserved)

This intercept is not implemented. The effect of setting this bit is undefined.

15.12.11 #TS (Invalid TSS)

The EXITINFO1 and EXITINFO2 fields are undefined. The rIP value saved in the VMCB may point to either the instruction causing the task switch, or to the first instruction of the incoming task. See Section 15.14.1 for information on the EXITINFO1 and EXITINFO2 fields.

15.12.12 #NP (Segment Not Present)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #NP exception. The EXITINFO2 field is undefined.

15.12.13 #SS (Stack Fault)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #SS exception. The EXITINFO2 field is undefined.

15.12.14 #GP (General Protection)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #GP exception.

15.12.15 #PF (Page Fault)

This intercept is tested *before* CR2 is written by the exception. The error code saved in EXITINFO1 is the same as would be pushed onto the stack by a non-intercepted #PF exception in protected mode. The faulting address is saved in the EXITINFO2 field in the VMCB. Even when the guest is running in paged real mode, the processor will deliver the (protected-mode) page-fault error code in EXITINFO1, for the VMM to use in analyzing the intercepted #PF. The processor may provide additional instruction decode assist information. (See Section 15.8.4.)

15.12.16 #MF (X87 Floating Point)

This intercept is tested *after* the floating point status word has been written, as is the case for a normal FP exception. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.17 #AC (Alignment Check)

The EXITINFO1 field contains the error code that would be pushed on the stack by an #AC exception. The EXITINFO2 field is undefined.

[AMD Public Use]

15.12.18 #MC (Machine Check)

The SVM intercept is checked after all #MC-specific registers have been written, but before other guest state is modified. When #MC is being intercepted, a machine-check exits to the VMM, whenever possible, and shuts down the processor only when this is not a reasonable option. The EXITINFO1 and EXITINFO2 fields are undefined.

Note that in some processors, if the guest VM has disabled machine check handling (CR4.MCE=0) then all machine check errors that occur in the guest will result in a shutdown event. However in processors where CPUID Fn8000_000A_EDX[HOST_MCE_OVERRIDE] (bit 23) = 1 the VMM may override this behavior by setting CR4.MCE=1 in the host. In this scenario, machine check errors that occur in the guest and which can be contained by the processor will always result in a #VMEXIT(MC).

15.12.19 #XF (SIMD Floating Point)

This intercept is tested after the SIMD status word (MXCSR) has been written, as is the case for a normal FP exception. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.20 #SX (Security Exception)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #SX exception. The EXITINFO2 field is undefined.

15.12.21 #CP (Control Protection)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #CP exception. The EXITINFO2 field is undefined.

15.13 Interrupt Intercepts

External interrupts, when intercepted, cause a #VMEXIT; the interrupt is held pending so that the interrupt can eventually be taken in the VMM. Exception intercepts do not apply to external or software interrupts, so it is not possible to intercept an interrupt by means of the exception intercepts, even if the interrupt should happen to use a vector in the range from 0 to 31.

15.13.1 INTR Intercept

This intercept affects physical, as opposed to virtual, maskable interrupts. See “Virtual Interrupt Intercept” on page 557 for virtualization of maskable interrupts.

15.13.2 NMI Intercept

This intercept affects non-maskable interrupts. NMI interrupts (and SMIs) may be blocked for one instruction following an STI.

[AMD Public Use]

15.13.3 SMI Intercept

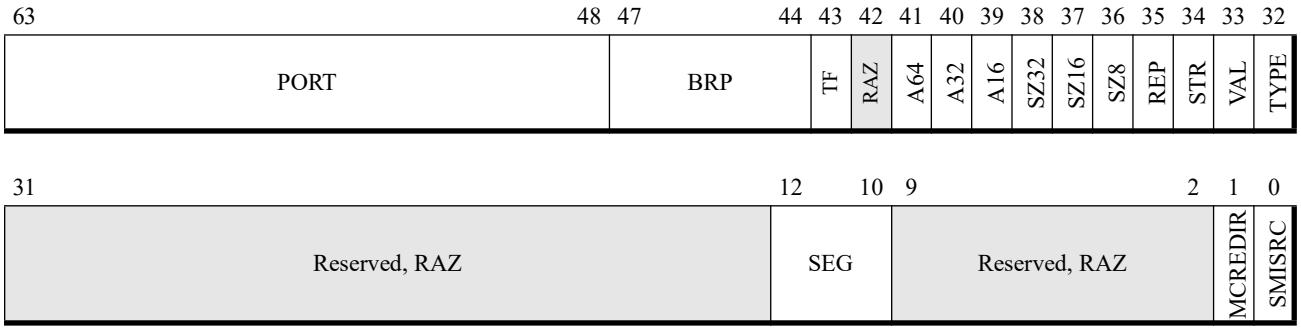
This intercept affects System Management Mode Interrupts (SMIs); see “SMM Support” on page 560 for details on SMI handling.

When this intercept triggers, bit 0 of the EXITINFO1 field distinguishes whether the SMI was caused internally by I/O Trapping (bit 0 = 0), or asserted externally (bit 0 = 1).

If the SMI was asserted while the guest was executing an I/O instruction, extra information (describing the I/O instruction) is saved in the upper 32 bits of EXITINFO1, and the rIP of the I/O instruction is saved in EXITINFO2. EXITINFO1 indicates that SMI was asserted during an I/O instruction when the VALID bit is set.

If the SMI wasn't asserted during an I/O instruction, the extra EXITINFO1 and EXITINFO2 bits are undefined.

The SMI intercept is ignored when HWCR[SMMLOCK] is set.



[AMD Public Use]

Bits	Mnemonic	Description
63:48	PORT	Intercepted I/O port
47:44	BRP	I/O breakpoint matches
43	TF	EFLAGS TF value
42	—	Reserved, RAZ
41	A64	64-bit address
40	A32	32-bit address
39	A16	16-bit address
38	SZ32	32-bit operand size
37	SZ16	16-bit operand size
36	SZ8	8-bit operand size
35	REP	Repeated port access
34	STR	String based port access (INS, OUTS)
33	VAL	Valid (SMI was detected during an I/O instruction)
32	TYPE	Access Type (0 = OUT instruction, 1 = IN instruction)
31:13	—	Reserved, RAZ
12:10	SEG	Effective segment number (see Section 15.9)
9:2	—	Reserved, RAZ
1	MCREDIR	SMI was due to a redirect machine check error (See “Interaction with SMI and #MC” on page 627)
0	SMISRC	SMI source (0 = internal, 1 = external)

Figure 15-3. EXITINFO1 for SMI Intercept

15.13.4 INIT Intercept

The INIT intercept allows the VMM to intercept the assertion of INIT while a guest is running. An intercepted INIT remains pending until the VMM sets GIF (see “Global Interrupt Flag, STGI and CLGI Instructions” on page 553), at which point it either takes effect or is redirected. See Section 15.21.8 for a discussion of the INIT redirection feature.

15.13.5 Virtual Interrupt Intercept

This intercept is taken just before a guest takes a virtual interrupt. When the intercept triggers, the virtual interrupt has not been taken, and remains pending in the guest's VMCB V_IRQ field. This intercept is not required for handling fixed local APIC interrupts, but may be used for emulating ExtINT interrupt delivery mode (which is not masked by the TPR), or legacy PICs in auto-EOI mode.

15.14 Miscellaneous Intercepts

The SVM architecture includes intercepts to handle task switches, processor freezes due to FERR, and shutdown operations.

[AMD Public Use]

15.14.1 Task Switch Intercept

Checked by—Any instruction or event that causes a task switch (e.g., JMP, CALL, exceptions, interrupts, software interrupts).

Priority—The intercept is checked before the task switch takes place but *after* the incoming TSS and task gate (if one was involved) have been checked for correctness.

Task switches can modify several resources that a VMM may want to protect (CR3, EFLAGS, LDT). However, instead of checking various intercepts (e.g., CR3 Write, LDTR Write) individually, task switches check only a single intercept bit.

On #VMEXIT, the following information is delivered in the VMCB:

- EXITINFO1[15:0] holds the segment selector identifying the incoming TSS.
- EXITINFO2[31:0] holds the error code to push in the new task, if applicable; otherwise, this field is undefined.
- EXITINFO2[63:32] holds auxiliary information for the VMM:
  - EXITINFO2[36]—Set to 1 if the task switch was caused by an IRET; else cleared to 0.
  - EXITINFO2[38]—Set to 1 if the task switch was caused by a far jump; else cleared to 0.
  - EXITINFO2[44]—Set to 1 if the task switch has an error code; else cleared to 0.
  - EXITINFO2[48]—The value of EFLAGS.RF that would be saved in the outgoing TSS if the task switch were not intercepted.

15.14.2 Ferr_Freeze Intercept

Checked when the processor freezes due to assertion of FERR (while IGNNE is deasserted, and legacy handling of FERR is selected in CR0.NE), i.e., while the processor is waiting to be unfrozen by an external interrupt.

15.14.3 Shutdown Intercept

When this intercept occurs, any condition that normally causes a shutdown causes a #VMEXIT to the VMM instead. After an intercepted shutdown, the VMCB control area is valid (with the exception of offsets 60h, 61h, and 68h) and the VMCB state save area is undefined.

15.14.4 Pause Intercept Filtering

On processors that support Pause filtering (indicated by CPUID Fn8000_000A_EDX[PauseFilter] = 1), the VMCB provides a 16 bit PAUSE Filter Count value. On VMRUN this value is loaded into an internal counter. Each time a PAUSE instruction is executed, this counter is decremented until it reaches zero at which time a #VMEXIT is generated if PAUSE intercept is enabled. If the PAUSE Filter Count is set to zero and PAUSE Intercept is enabled, every PAUSE instruction will cause a #VMEXIT.

[AMD Public Use]

In addition, some processor families support Advanced Pause Filtering (indicated by CPUID Fn8000_000A_EDX[PauseFilterThreshold] = 1). In this mode, a 16-bit PAUSE Filter Threshold field is added in the VMCB. The threshold value is a cycle count that is used to reset the pause counter.

As with simple Pause filtering, VMRUN loads the PAUSE count VMCB value into an internal counter. Then, on each PAUSE instruction the processor checks the elapsed number of cycles since the most recent PAUSE instruction against the PAUSE Filter Threshold. If the elapsed cycle count is greater than the PAUSE Filter Threshold, then the internal pause count is reloaded from the VMCB and execution continues. If the elapsed cycle count is less than the PAUSE Filter Threshold, then the internal pause count is decremented. If the count value is less than zero and PAUSE intercept is enabled, a #VMEXIT is triggered.

If Advanced Pause Filtering is supported and PAUSE Filter Threshold field is set to zero, the filter will operate in the simpler, count only mode.

See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

15.14.5 Bus Lock Threshold

On processors that support Bus Lock Threshold (indicated by CPUID Fn8000_000A_EDX[29] BusLockThreshold=1), the VMCB provides a Bus Lock Threshold enable bit and an unsigned 16-bit Bus Lock Threshold count. On VMRUN, this value is loaded into an internal count register. Before the processor executes a bus lock in the guest, it checks the value of this register. If the value is greater than 0, the processor executes the bus lock successfully and decrements the count. If the value is 0, the bus lock is not executed and a #VMEXIT to the VMM is taken. A Bus Lock Threshold #VMEXIT is reported to the VMM with VMEXIT code A5h, VMEXIT_BUSLOCK. EXITINFO1 and EXITINFO2 are set to 0 on a VMEXIT_BUSLOCK.

On a #VMEXIT, the processor writes the current value of the Bus Lock Threshold Counter to the VMCB.

Section 7.3.3 on page 197 describes the conditions under which a bus lock is executed.

The bus lock threshold #VMEXIT may occur due to a table walk A/D bit update if a page table memory type is not WB. In this case, the bus lock threshold #VMEXIT is still reported as a VMEXIT_BUSLOCK, not a Nested Page Fault.

15.15 VMCB State Caching

VMCB state caching allows the processor to cache certain guest register values in hardware between a #VMEXIT and subsequent VMRUN instructions and use the cached values to improve context-switch performance. Depending on the particular processor implementation, VMRUN loads each guest register value either from the VMCB or from the VMCB state cache, as specified by the value of the VMCB Clean field in the VMCB. Support for VMCB state caching is indicated by CPUID Fn8000_000A_EDX[VmcbClean] = 1.

[AMD Public Use]

The SVM architecture uses the physical address of the VMCB as a unique identifier for the guest virtual CPU for the purposes of deciding whether the cached copy belongs to the guest. For the purposes of VMCB state caching, the ASID is not a unique identifier for a guest virtual CPU.

15.15.1 VMCB Clean Bits

The VMCB Clean field (VMCB offset 0C0h, bits 31:0) controls which guest register values are loaded from the VMCB state cache on VMRUN. Each set bit in the VMCB Clean field allows the processor to load one guest register or group of registers from the hardware cache; each clear bit requires that the processor load the guest register from the VMCB. The clean bits are a hint, since any given processor implementation may ignore bits that are set to 1 on any given VMRUN, unconditionally loading the associated register value(s) from the VMCB. Clean bits that are set to zero are always honored.

This field is backward-compatible to CPUs that do not support VMCB state caching; older CPUs neither cache VMCB state nor read the VMCB Clean field.

Older hypervisors that are not aware of VMCB state caching and respect the SBZ property of undefined VMCB fields will not enable VMCB state caching.

15.15.2 Guidelines for Clearing VMCB Clean Bits

The hypervisor must clear specific bits in the VMCB Clean field every time it explicitly modifies the associated guest state in the VMCB. The guest's execution can cause cached state to be updated, but the hypervisor is not responsible for setting VMCB Clean bits corresponding to any state changes caused by guest execution.

The hypervisor must clear the entire VMCB field to 0 for a guest, under the following circumstances:

- This is the first time a particular guest is run.
- The hypervisor executes the guest on a different CPU core than one used the last time that guest was executed.
- The hypervisor has moved the guest's VMCB to a different physical page since the last time that guest was executed.

Failure to clear the VMCB Clean bits to zero in these cases may result in undefined behavior.

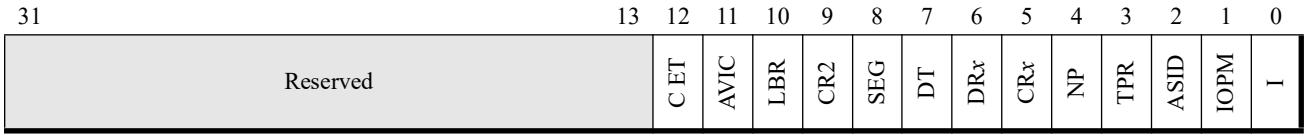
The CPU automatically treats the VMCB Clean field as zero on the current VMRUN when the hypervisor executes a guest that is not currently cached. The CPU compares the VMCB physical address against all cached VMCB physical addresses and treats the VMCB Clean field as zero, if no cached VMCB address matches.

SMM software (or any other agent external to the hypervisor that has access to VMCBs) that changes the contents of a VMCB needs to comprehend the clean bits and adjust them accordingly; otherwise the guest may not operate as intended.

[AMD Public Use]

15.15.3 VMCB Clean Field

The VMCB Clean field layout is illustrated in Figure 15-4 below.



Bits	Mnemonic	Description
31:13	—	Reserved
12	CET	S_CET, SSP, ISST_ADDR
11	AVIC	AVIC APIC_BAR; AVIC APIC_BACKING_PAGE, AVIC PHYSICAL_TABLE and AVIC LOGICAL_TABLE Pointers
10	LBR	DebugCtl MSR, br_from/to, lastint_from/to
9	CR2	CR2
8	SEG	CS/DS/SS/ES Sel/Base/Limit/Attr, CPL
7	DT	GDT/IDT Limit and Base
6	DR*x*	DR6, DR7
5	CR*x*	CR0, CR3, CR4, EFER
4	NP	Nested Paging: NCR3, G_PAT
3	TPR	V_TPR, V_IRQ, V_INTR_PRIO, V_IGN_TPR, V_INTR_MASKING, V_INTR_VECTOR, V_NMI, V_NMI_MASK (Offset 60h–67h)
2	ASID	ASID
1	IOPM	IOMSRPM: IOPM_BASE, MSRPM_BASE
0	I	Intercepts: all the intercept vectors, TSC offset, Pause Filter Count

Figure 15-4. VMCB Clean Field

For forward compatibility, if the hypervisor has not modified the VMCB, the hypervisor may write FFFF_FFFFh to the VMCB Clean Field to indicate that it has not changed any VMCB contents other than the fields described below as explicitly uncached. The hypervisor should write 0h to indicate that the VMCB is new or potentially inconsistent with the CPU's cached copy, as occurs when the hypervisor has allocated a new location for an existing VMCB from a list of free pages and does not track whether that page had recently been used as a VMCB for another guest. If any VMCB fields (excluding explicitly uncached fields) have been modified, all clean bits that are undefined (within the scope of the hypervisor) must be cleared to zero.

Bit 31 is a special bit reserved for the host that the CPU will never use for a clean bit.

The following are explicitly not cached and not represented by Clean bits:

- TLB_Control
- Interrupt shadow
- VMCB status fields (Exitcode, EXITINFO1, EXITINFO2, EXITINTINFO, Decode Assist, etc.)

[AMD Public Use]

- Event injection
- RFLAGS, RIP, RSP, RAX

15.16 TLB Control

TLB entries are tagged with *Address Space Identifier* (ASID) bits to distinguish different guest virtual address spaces when shadow page tables are used, or different guest physical address spaces when nested page tables are used. The VMM can choose a software strategy in which it keeps multiple shadow page tables, and/or multiple nested page tables in processors that support nested paging, up-to-date; the VMM can allocate a different ASID for each shadow or nested page table. This allows switching to a new process in a guest under shadow paging (changing CR3 contents), or to a new guest under nested paging (changing nCR3 contents), without flushing the TLBs. (See Section 15.25 for a complete explanation of nested paging operation.)

With shadow paging, the VMM is responsible for setting up a shadow page table for each guest linear address space that maps it to system physical addresses. These are used as the active page tables in place of the guest OS's page tables. The VMM sets the CR3 field in the guest VMCB to point to the system physical address of the desired shadow page table. The VMM is responsible for updating the shadow page table when the guest changes its page table or paging control state, and the VMM updates the access and dirty bits of the guest page table.

The VMRUN instruction and #VMEXIT write the CR0, CR3, CR4 and EFER registers, but these writes do *not* flush the TLB. The VMM is responsible for explicitly invalidating any guest translations that may be affected by its actions. There are two mechanisms available for this described in the next two sections.

When running with SVM enabled, global page table entries (PTEs) are global only *within* an ASID, not across ASIDs.

Software Rule. When the VMM changes a guest’s paging mode by changing entries in the guest’s VMCB, the VMM must ensure that the guest’s TLB entries are flushed from the TLB. The relevant VMCB state includes:

- CR0—PG, WP, CD, NW.
- CR3—Any bit.
- CR4—PGE, PAE, PSE.
- EFER—NXE, LMA, LME.

15.16.1 TLB Flush

TLB flush operations function identically whether or not SVM is enabled (e.g., MOV CR3 instruction flushes non-global mappings, whereas MOV CR4 instruction flushes global and non-global mappings). TLB flush operations must not be assumed to affect all ASIDs. If a VMM sets the intercept bit for any guest action that would have flushed the TLB, the #VMEXIT intercept occurs and the TLB is not flushed; it is the VMM's responsibility to flush the TLB appropriately. In implementations that

[AMD Public Use]

do not provide a way to selectively flush all translations of a single specified ASID, software may effectively flush the guest's TLB entries by allocating a new ASID for the guest and not reusing the old ASID until the entire TLB has been flushed at least once.

The TLB_CONTROL field in the VMCB provides the commands specified by the control byte encodings shown in Table 15-9. The first two commands are available on all processors that support SVM; support for the other commands is optional and is indicated by CPUID Fn8000_000A_EDX[FlushByAsid] = 1.

Table 15-9. TLB Control Byte Encodings

Encoding	Function Definition
00h	Do not flush
01h	Flush entire TLB (Should be used only on legacy hardware.)
03h	Flush this guest's TLB entries
07h	Flush this guest's non-global TLB entries
Note: Encodings not defined in this table are reserved.	

When the VMM sets the TLB_CONTROL field to 1, the VMRUN instruction flushes the TLB for all ASIDs, for both global and non-global pages. The VMRUN instruction reads, but does not change, the value of the TLB_CONTROL field.

A MOV CR3 instruction, a task switch that changes CR3, or clearing or setting CR0.PG or bits PGE, PAE, PSE of CR4 affects only the TLB entries belonging to the current ASID, regardless of whether the operation occurred in host or guest mode. The current ASID is 0 when the CPU is not inside a guest context.

All TLB entries belonging to all ASIDs are flushed by SMI, RSM, MTRR modifications, IORR modifications, and access to other system MSRs that affect address translation.

If a hypervisor modifies a nested page table by decreasing permission levels, clearing present bits, or changing address translations and intends to return to the same ASID, it should use either TLB command 011b or 001b.

15.16.2 Invalidate Page, Alternate ASID

The INVLPGA instruction allows the VMM to selectively invalidate the TLB mapping for a given guest virtual page within a given ASID. The linear address is specified in the implicit register operand rAX; the ASID is specified in ECX. The input address is always interpreted as a guest virtual address, so INVLPGA is typically meaningful only when used with shadow page tables; it does not provide a means to invalidate a nested translation by guest physical address.

[AMD Public Use]

15.17 Global Interrupt Flag, STGI and CLGI Instructions

The global interrupt flag (GIF) is a bit that controls whether interrupts and other events can be taken by the processor. The STGI and CLGI instructions set and clear, respectively, the GIF. Table 15-10 shows how the value of the GIF affects how interrupts and exceptions are handled. Implementations may provide hardware support for virtualizing the GIF in nested virtualization scenarios; see Section 15.33, for details.

Table 15-10. Effect of the GIF on Interrupt Handling

Interrupt source	GIF==0	GIF ==1
Debug exception or trap, due to breakpoint register match	Ignored and discarded	Normal operation
Debug trace trap due to EFLAGS.TF	Normal operation	Normal operation
RESET	Normal operation	Normal operation
INIT	Held pending until GIF==1	Normal operation, see Table 15-12
NMI	Held pending until GIF==1	Normal operation, see Table 15-13
External SMI	Held pending until GIF==1	Normal operation, see Table 15-14
Internal SMI (I/O Trapping)	Ignored and discarded	Normal operation, see Table 15-14
INTR and vINTR	Held pending until GIF==1	Normal operation
#SX (Security Exception)	n/a[1]	Normal operation
Machine Check	If possible (implementation-dependent), held pending until GIF==1, otherwise shutdown.	Normal operation
A20M	Normal operation	Normal operation
	(VM_CR.DIS_A20M controls A20 masking)	
Other implementation-specific but non-architecturally-visible interrupts (STPCLK, IGNNE toggle, ECC scrub)	Normal operation	Normal operation

Note:

1. *#SX is caused only by an INIT signal that has been "redirected" (i.e., converted to an #SX; see* Section 15.28*); the conversion only happens when GIF==1, as the INIT is simply held pending otherwise.*

15.18 VMMCALL Instruction

This instruction is meant as a way for a guest to explicitly call the VMM. No CPL checks are performed, so the VMM can decide whether to make this instruction legal at the user-level or not.

If VMMCALL instruction is not intercepted, the instruction raises a #UD exception.

[AMD Public Use]

15.19 Paged Real Mode

To facilitate virtualization of real mode, the VMRUN instruction may legally load a guest CR0 value with PE = 0 but PG = 1. Likewise, the RSM instruction is permitted to return to paged real mode. This processor mode behaves in every way like real mode, with the exception that paging is applied. The intent is that the VMM run the guest in paged-real mode at CPL0, and with page faults intercepted. The VMM is responsible for setting up a shadow page table that maps guest *physical* memory to the appropriate system physical addresses.

The behavior of running a guest in paged real mode without intercepting page faults to the VMM is undefined.

15.20 Event Injection

The VMM can inject exceptions or interrupts (collectively referred to as events) into the guest by setting bits in the VMCB's EVENTINJ field prior to executing the VMRUN instruction. The format of the field is shown in Figure 15-5. The encoding matches that of the EXITINTINFO field. When an event is injected by means of this mechanism, the VMRUN instruction causes the guest to take the specified exception or interrupt unconditionally before executing the first guest instruction.

Injected events are treated in every way as though they had occurred normally in the guest (in particular, they are recorded in EXITINTINFO) with the following exceptions:

- Injected events are not subject to intercept checks. (Note, however, that if secondary exceptions occur during delivery of an injected event, those exceptions *are* subject to exception intercepts.)
- An injected NMI does not block delivery of further NMIs.
- If the VMM attempts to inject an event that is impossible for the guest mode (e.g., a #BR exception when the guest is in 64-bit mode), the event injection will fail and no guest state instructions will be executed; VMRUN will immediately exit with an error code of VMEXIT_INVALID.
- Injecting an exception (TYPE = 3) with vectors 3 or 4 behaves like a trap raised by INT3 and INTO instructions, respectively, in which case the processor checks the DPL of the IDT descriptor before dispatching to the handler.
- Software interrupts cannot be properly injected if the processor does not support the NextRIP field. Support is indicated by CPUID Fn8000_000A_EDX[NRIPS] = 1. Hypervisor software should emulate the event injection of software interrupts if NextRIP is not supported.
- Event injection does not support injection of intercepted #DB faults that are the result of a guest ICEBP instruction. ICEBP does not perform DPL checks, as does INT*n* injection. Hypervisor software should emulate the injection of ICEBP.

[AMD Public Use]

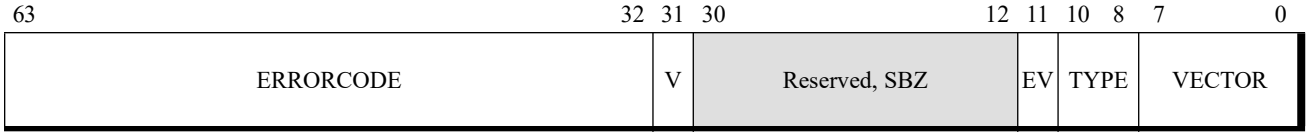


Figure 15-5. EVENTINJ Field in the VMCB

The fields in EVENTINJ are as follows:

- *VECTOR*—Bits 7:0. The 8-bit IDT vector of the interrupt or exception. If TYPE is 2 (NMI), the VECTOR field is ignored.
- *TYPE*—Bits 10:8. Qualifies the guest exception or interrupt to generate. Table 15-11 shows possible values and their corresponding interrupt or exception types. Values not indicated are unused and reserved.

Table 15-11. Guest Exception or Interrupt Types

Value	Type
0	External or virtual interrupt (INTR)
2	NMI
3	Exception (fault or trap)
4	Software interrupt (INT*n* instruction)

- *EV (Error Code Valid)*—Bit 11. Set to 1 if the exception should push an error code onto the stack; clear to 0 otherwise.
- *V (Valid)*—Bit 31. Set to 1 if an event is to be injected into the guest; clear to 0 otherwise.
- *ERRORCODE*—Bits 63:32. If EV is set to 1, the error code to be pushed onto the stack, ignored otherwise.

VMRUN exits with VMEXIT_INVALID error code if either:

- Reserved values of TYPE have been specified, or
- TYPE = 3 (exception) has been specified with a vector that does not correspond to an exception (this includes vector 2, which is an NMI, not an exception).

15.21 Interrupt and Local APIC Support

SVM hardware support is designed to ensure efficient virtualization of interrupts.

15.21.1 Physical (INTR) Interrupt Masking in EFLAGS

To prevent the guest from blocking maskable interrupts (INTR), SVM provides a VMCB control bit, V_INTR_MASKING, which changes the operation of EFLAGS.IF and accesses to the TPR by means of the CR8 register. While running a guest with V_INTR_MASKING cleared to zero:

[AMD Public Use]

- EFLAGS.IF controls both virtual and physical interrupts.

While running a guest with V_INTR_MASKING set to 1:

- The host EFLAGS.IF at the time of the VMRUN is saved and controls physical interrupts while the guest is running.
- The guest value of EFLAGS.IF controls virtual interrupts only.

15.21.2 Virtualizing APIC TPR

SVM provides a virtual TPR register, V_TPR, for use by the guest; its value is loaded from the VMCB by VMRUN and written back to the VMCB by #VMEXIT. The APIC's TPR always controls the task priority for physical interrupts, and the V_TPR always controls virtual interrupts.

While running a guest with V_INTR_MASKING cleared to 0:

- Writes to CR8 affect both the APIC's TPR and the V_TPR register.
- Reads from CR8 operate as they would without SVM.

While running a guest with V_INTR_MASKING set to 1:

- Writes to CR8 affect only the V_TPR register.
- Reads from CR8 return V_TPR.

15.21.3 TPR Access in 32-Bit Mode

The mechanism for TPR virtualization described in Section 15.21.2 applies only to accesses that are performed using the CR8 register. However, in 32-bit mode, the TPR is traditionally accessible only by using a memory-mapped register. Typically, a VMM virtualizes such TPR accesses by not mapping the APIC page addresses in the guest. A guest access to that region then causes a #PF intercept to the VMM, which inspects the guest page tables to determine the physical address and, after recognizing the physical address as belonging to the APIC, finally invokes software emulation code.

To improve the efficiency of TPR accesses in 32-bit mode, SVM makes CR8 available to 32-bit code by means of an alternate encoding of MOV TO/FROM CR8 (namely, MOV TO/FROM CR0 with a LOCK prefix). To achieve better performance, 32-bit guests should be modified to use this access method, instead of the memory-mapped TPR. (For details, see “MOV CRn” on page 377 of the *AMD64 Programmer’s Reference Volume 3: General Purpose and System Instructions*, order# 24594.)

The alternate encodings of the MOV TO/FROM CR8 instructions are available even if SVM is disabled in EFER.SVME. They are available in both 64-bit and 32-bit mode.

15.21.4 Injecting Virtual (INTR) Interrupts

Virtual Interrupts allow the host to pass an interrupt (#INTR) to a guest. While inside a guest, the virtual interrupt follows the same rules that a real interrupt follows (virtual #INTR is not taken until EFLAGS.IF is 1, the guest's TPR has enabled interrupts at the same priority as that of the pending virtual interrupt).

[AMD Public Use]

SVM provides an efficient mechanism by which the VMM can inject virtual interrupts into a guest:

- As described in Section 15.13.1, the VMM can intercept physical interrupts that arrive while a guest is running, by activating the INTR intercept in the VMCB.
- As described in Section 15.21.4, the VMM can virtualize the interrupt masking logic by setting the V_INTR_MASKING bit in the VMCB.
- The three VMCB fields V_IRQ, V_INTR_PRIO, and V_INTR_VECTOR indicate whether there is a virtual interrupt pending, and, if so, what its vector number and priority are. The VMRUN instruction loads this information into corresponding on-chip registers.
- The processor takes a virtual INTR interrupt if:
  - V_IRQ and V_INTR_PRIO indicate that there is a virtual interrupt pending whose priority is greater than the value in V_TPR,
  - interrupts are enabled in EFLAGS.IF,
  - virtual interrupts are enabled with VGIF, and
  - the processor is not in an interrupt shadow (see Section 15.21.5).

  The only other difference between virtual INTR handling and normal interrupt handling is that, in the latter case, the interrupt vector is obtained from the V_INTR_VECTOR register (as opposed to running an INTACK cycle to the local APIC).
- The V_IGN_TPR field in the VMCB can be set to indicate that the currently pending virtual interrupt is not subject to masking by TPR. The priority comparison against V_TPR is omitted in this case. This mechanism can be used to inject ExtINT-type interrupts into the guest.
- When the processor dispatches a virtual interrupt (through the IDT), V_IRQ is cleared after checking for intercepts of virtual interrupts and before the IDT is accessed.
- On #VMEXIT, V_IRQ is written back to the VMCB, allowing the VMM to track whether a virtual interrupt has been taken.
- Physical interrupts take priority over virtual interrupts, whether they are taken directly or through a #VMEXIT.
- On #VMEXIT, the processor clears its internal copies of V_IRQ and V_INTR_MASKING, so virtual interrupts do not remain pending in the VMM, and interrupt control reverts to normal.

15.21.5 Interrupt Shadows

The x86 architecture defines the notion of an *interrupt shadow*—a single-instruction window during which interrupts are not recognized. For example, the instruction after an STI instruction that sets EFLAGS.IF (from zero to one) does not recognize interrupts or certain debug traps. The VMCB INTERRUPT_SHADOW field indicates whether the guest is currently in an interrupt shadow. This information is saved on #VMEXIT and loaded on VMRUN.

15.21.6 Virtual Interrupt Intercept

When virtualizing interrupt handling, a VMM typically needs only gain control when new interrupts for a guest arrive or are generated, and when the guest issues an EOI (end-of-interrupt). In some

[AMD Public Use]

circumstances, it may also be necessary for the VMM to gain control at the moment interrupts become enabled in the guest (i.e., just before the guest takes a virtual interrupt). The VMM can do so by enabling the VINTR intercept.

15.21.7 Interrupt Masking in Local APIC

When guests have direct access to devices, interrupts arriving at the local APIC can usually be dismissed only by the guest that owns the device causing the interrupt. To prevent one guest from blocking other guests' interrupts (by never processing their own), the VMM can mask pending interrupts in the local APIC, so they do not participate in the prioritization of other interrupts.

SVM introduces the following APIC features:

- A 256-bit IER (interrupt enable) register is added to the local APIC. This register resets to all ones (enabling all 256 vectors). Software can read and write the IER by means of the memory-mapped APIC page.
- Only vectors that are enabled in the IER participate in the APIC computation of the highest-priority pending interrupt.
- The VMM can issue specific end-of-interrupt (EOI) commands to the local APIC, allowing the VMM to clear pending interrupts in any order, rather than always targeting the interrupt with highest-priority.

15.21.8 INIT Support

The INIT signal interrupts the processor at the next instruction boundary and causes an unconditional control transfer. INIT reinitializes the control registers, segment registers and GP registers in a manner similar to RESET, but does not alter the contents of most MSRs, caches or numeric coprocessor (x87 or SSE) state, and then transfers control to the same instruction address as RESET (physical address FFFFFFF0h). Unlike RESET, INIT is not expected to be visible to the memory controller, and hence will not trigger automatic clearing of trusted memory pages by memory controller hardware.

To maintain the security of such pages, the VMM can request that INITs be redirected and turned into #SX exceptions by setting the R_INIT bit in the VM_CR MSR (see Section 15.30.1). This allows the VMM to gain control when an INIT is requested and scrub any sensitive context. The VMM may then disable the redirection of INIT and cause the platform to reassert INIT (see the relevant *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details), at which point the processor will respond in the normal manner. The actions initiated by the INIT pin may also be initiated by an incoming APIC INIT interrupt; the mechanisms described here apply in either case. Table 15-12 summarizes the handling of INITs.

[AMD Public Use]

Table 15-12. INIT Handling in Different Operating Modes

GIF	INIT Intercept	INIT Redirect	Processor Response to INIT
0	X	X	Hold pending until GIF = 1.
1	1	X	#VMEXIT(INIT), INIT is still pending.
	0	0	Taken normally.
		1	#SX, INIT is no longer pending.

If redirection is enabled without the INIT intercept being enabled, an INIT that asserts during guest execution will result in #SX being asserted within the guest, with the INIT being cleared. The VMM may intercept the assertion of #SX as described in “Intercept Operation” on page 530. Note that when a VMM has intercepted an INIT assertion, it may modify R_INIT any time before setting GIF to control the behavior when GIF is ultimately set, or the VMM may instead return to a guest VM with the intercept disabled and redirection enabled to effectively hand off the INIT to the guest as a #SX exception.

15.21.9 NMI Support

The VMM can intercept non-maskable interrupts (NMI) using a VMCB control bit (see Table 15-13). When intercepted, NMIs cause an exit from the guest and are held pending.

Table 15-13. NMI Handling in Different Operating Modes

GIF	NMI Intercept	Processor Response to NMI
0	X	Hold pending until GIF=1.
1	1	#VMEXIT(NMI), NMI is still pending.
	0	Taken normally.

15.21.10 NMI Virtualization

NMI Virtualization allows the host to inject an NMI (#NMI) into a guest. While inside a guest, the virtual NMI follows the same rules as a physical NMI. With NMI Virtualization, the processor virtualizes the masking state of NMIs which prevents a guest from taking a second virtual NMI until the execution of an IRET instruction.

NMI Virtualization support is indicated by CPUID Fn8000_000A_EDX[VNMI] = 1.

NMI Virtualization is enabled by setting V_NMI_ENABLE (bit 26 in offset 60h of the VMCB). Enabling NMI Virtualization requires the NMI intercept bit to be set. An attempt to run a guest with V_NMI_ENABLE without the NMI intercept bit set results in #VMEXIT(INVALID). When NMI Virtualization is enabled, NMI intercepts only apply to physical NMIs, not virtual NMIs.

Three new bits are added to the VMCB field at offset 60h to provide NMI virtualization hardware support:

[AMD Public Use]

V_NMI: Indicates whether a virtual NMI is pending in the guest. The processor will clear V_NMI once it takes the virtual NMI.

V_NMI_MASK: Indicates whether virtual NMIs are masked. The processor will set V_NMI_MASK once it takes the virtual NMI. V_NMI_MASK is cleared when the guest successfully completes an IRET instruction or #VMEXIT occurs while delivering the virtual NMI.

V_NMI_ENABLE: Enables NMI virtualization.

SVM provides an efficient mechanism by which the VMM can inject virtual NMI into a guest:

- The VMM can inject NMI into the guest by setting V_NMI_ENABLE and V_NMI in the VMCB.
- When V_NMI_ENABLE is set, VMRUN loads V_NMI and V_NMI_MASK from the VMCB into internal registers.
- The processor takes a virtual NMI if:
  - virtual NMIs are not masked,
  - virtual interrupts are enabled with VGIF, and
  - the processor is not in an interrupt shadow.
- When the processor recognizes a virtual NMI, V_NMI is cleared and V_NMI_MASK is set before the IDT is accessed.
- On #VMEXIT, V_NMI and V_NMI_MASK are written back to the VMCB.
- If a #VMEXIT occurs during the virtual NMI delivery, EXITINTINFO is set appropriately and V_NMI_MASK is saved as 0.
- If Event Injection is used to inject an NMI when NMI Virtualization is enabled, VMRUN sets V_NMI_MASK in the guest state.
- Physical NMI takes priority over virtual NMI.

15.22 SMM Support

This section describes SVM support for virtualization of System Management Mode (SMM).

15.22.1 Sources of SMI

Various events can cause an assertion of a system management interrupt (SMI); these are classified into three categories

- Internal, synchronous (also known as I/O Trapping)—implementation-specific IOIO or config space trapping in the CPU itself; always synchronous in response to an IN or OUT instruction. I/O Trapping is set up by means of MSRs and can be brought under the control of the VMM by intercepting guest access to those MSRs.
- External, synchronous—IOIO trapping in response to (and synchronous with) IN or OUT instructions, but generated by an external agent (typically the Southbridge).

[AMD Public Use]

- External, asynchronous—generated externally in response to an external, physical event, e.g., closing a laptop lid, temperature sensor triggering, etc.

15.22.2 Response to SMI

How hardware responds to SMIs is a function of whether SMM interrupts are being intercepted and whether interrupts are enabled globally, as shown in Table 15-14.

Table 15-14. SMI Handling in Different Operating Modes

GIF	Intercept SMI	Internal SMI	External SMI
0	x	Lost.	Hold pending until GIF=1.
1	1	Exit guest, code #VMEXIT(SMI), SMI is not pending.	#VMEXIT(SMI), SMI is still pending.
	0	Taken normally.	Taken normally.

By intercepting SMIs, the VMM can gain control before the processor enters SMM.

15.22.3 Containerizing Platform SMM

In some usage scenarios, the VMM may not trust the existing platform SMM code, or may otherwise want to ensure that the SMM does not operate in the context of certain guests or the hypervisor. To address these cases, SVM provides the ability to *containerize* SMM code, i.e., run it inside a guest, with the full protection mechanisms of the VMM in place. In other scenarios, the VMM may not want to exert control over SMM.

There are three solutions for the VMM to control SMM handlers:

- The simplest solution is to not intercept SMI signals. SMIs encountered while in a guest context are taken from within the guest context. In this case, the SMM handler is not subject to any intercepts set up by the VMM and consequently runs outside of the virtualization controls. The state saved in the SMM State-Save area as seen by the SMM handler reflects the state of the guest that had been running at the time the SMI was encountered. When the SMM handler executes the RSM instruction, the processor returns to executing in the guest context, and any modifications to the SMM State-Save area made by the SMM handler are reflected in the guest state.
- A hypervisor may want to emulate all SMI-based I/O interceptions for a guest and to take SMI signals only in the hypervisor context. The hypervisor should set all IOIO intercept bits and the SMI intercept bit for the guest to ensure that there is no possibility of encountering synchronous (internal or external) SMI signals while running the guest. Any #VMEXIT(SMI) encountered is then known to be due to an external, asynchronous SMI. The hypervisor may respond to the #VMEXIT(SMI) by executing the STGI instruction, which causes the pending SMI to be taken immediately. When an SMI due to an I/O instruction is pending, the effect of executing STGI in the hypervisor is undefined. To handle a pending SMI due to an I/O instruction, the hypervisor must either containerize SMM or not intercept SMI.

[AMD Public Use]

- The most involved solution is to containerize SMM by placing it in a guest. Containerizing gives the VMM full control over the state that the SMM handler can access.

Containerizing Platform SMM. A VMM can containerize SMM by creating its own trusted SMM hypervisor and use that handler to run the platform SMM code in a container. The SMM hypervisor may be the same code as the VMM itself, or may be an entirely different set of code. The trusted SMM hypervisor sets up a guest context to run the platform SMM as a guest. The guest context consists of a VMCB and related state and the guest's (real or virtual) SMM save area. The SMM hypervisor emulates SMM entry, including setup of the SMM save area, and emulates RSM at the end of SMM operation. The guest executes the platform SMM code in paged real mode with appropriate SVM intercepts in place, thus ensuring security.

For this approach to work, the VMM may need to write the SMM_BASE MSR, as well as related SMM control registers. As part of the emulation of SMM entry and RSM, the VMM needs to access the SMM_CTL MSR (see Section 15.30.3). However, these actions conflict with any platform firmware that locks SMM control registers.

A VMM can determine if it is running with a compatible firmware setup by checking the SMMLOCK bit in the HWCR MSR (described in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product). If the bit is 1, firmware has locked the SMM control registers and the VMM is unable to move them or insert its own SMM hypervisor.

As the processor physically enters SMM, the SMRAM regions are remapped. The VMM design must ensure that none of its code or data disappears when the SMRAM areas are mapped or unmapped. Also note that the ASEG region of the SMRAM overlaps with a portion of video memory, so the SMM hypervisor should not attempt to write diagnostic messages to the screen. Any attempt by guests to relocate any of the SMRAM areas (by means of certain MSR writes) must also be intercepted to prevent malicious SMM code from interfering with VMM operation.

Writes to the SMM_CTL MSR cause a #GP if firmware has locked the SMM control registers.

15.23 Last Branch Record Virtualization

The debug control MSR (DebugCtl) provides control of control-transfer recording and other debug facilities. (See Chapter 13, "Software Debug and Performance Resources," on page 392, for more information on using the debug control MSR.) Software sets the last-branch record (DebugCtl[LBR]) bit to 1 to cause the processor to record the source and target addresses of the last control transfer taken before a debug exception. These control transfers include branch instructions, interrupts, and exceptions. Recorded information is stored in four MSRs:

- LastBranchFromIP
- LastBranchToIP
- LastIntFromIP
- LastIntToIP

[AMD Public Use]

Under SVM, to virtualize the function of these MSRs, the VMM must save the contents of the control-transfer recording MSRs on #VMEXIT and restore them prior to the VMRUN for each guest. If control-transfer recording is to be used in host state as well the values of these registers must be exchanged between values tracked by host and guest.

15.23.1 Hardware Acceleration for LBR Virtualization

Processors optionally support hardware acceleration for LBR virtualization. The following fields are allocated in the VMCB state save area to hold the contents of the DebugCtl and control-transfer recording MSRs:

- DBGCTL—Holds the guest value of the DebugCtl MSR.
- BR_FROM—Holds the guest value of the LastBranchFromIP MSR.
- BR_TO—Holds the guest value of the LastBranchToIP MSR.
- LASTEXCPFROM—Holds the guest value of the LastIntFromIP MSR.
- LASTEXCPTO—Holds the guest value of the LastIntToIPLastIntToIP MSR.

When VMCB.LBR_VIRTUALIZATION_ENABLE is set, VMRUN saves all five host control-transfer MSRs in the host save area, and then loads the same five MSRs for the guest from the VMCB save area. Similarly, #VMEXIT saves the guest's MSRs and loads the host's MSRs to and from their respective save areas.

On processors supporting LBR Stack, VMCB.LBR_VIRTUALIZATION_ENABLE also controls save and restore of the following fields in the VMCB state save area:

- DBGEXTNCTL - Holds the guest value of the DebugExtnCtl MSR.
- LBR_STACK_FROM - Holds the guest value of the LastBranchStackFromIp MSRs
- LBR_STACK_TO - Holds the guest value of the LastBranchStackToIp MSRs
- LBR_SELECT - Holds the guest value of the LastBranchStackSelect MSR.

VMRUN saves the host state and restores the guest state and #VMEXIT saves the guest state and restores the host state of the DebugExtnCtl MSR. VMRUN only restores the guest state and #VMEXIT only saves the guest state of the LastBranchStackFromIp, LastBranchStackToIp and LastBranchStackSelect MSRs. For SEV-ES guests #VMEXIT clears the contents of the LastBranchStackFromIp, LastBranchStackToIp and LastBranchStackSelect MSRs after saving the guest state.

15.23.2 LBR Virtualization CPUID Feature Detection

CPUID Fn8000_000A_EDX[LbrVirt] = 1 indicates support for the LBR virtualization acceleration feature on AMD64 processors. See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

[AMD Public Use]

15.24 External Access Protection

By securing the virtual address translation mechanism, the VMM can restrict guest CPU accesses to memory. However, should the guest have direct access to DMA-capable devices, an additional protection mechanism is required. SVM provides multiple protection domains which can restrict device access to physical memory on a per-page basis. This is accomplished via control logic in the Northbridge's host bridge which governs any external access port (e.g., PCI or HyperTransport™ technology interfaces).

15.24.1 Device IDs and Protection Domains

The Northbridge's host bridge provides a number of protection domains. Each protection domain has associated with it a device exclusion vector (DEV) that specifies the per-page access rights of devices in that domain. Devices are identified by a HyperTransport™ bus/unitID (device ID) and the host bridge contains a lookup table of fixed size that maps device IDs to a protection domain.

15.24.2 Device Exclusion Vector (DEV)

A DEV is a contiguous array of bits in physical memory; each bit in the DEV (in little-endian order) corresponds to one 4-Kbyte page in physical memory.

The physical address of the base of a DEV must be 4-Kbyte-aligned and stored in one of the DEVBASE registers, which are accessed through an indirection mechanism in the DEVCTL PCI Configuration Space function block in the host bridge (see "DEV Control and Status Registers" on page 568). The DEV protection hardware is not operational until enabled by setting a control bit in the DEV Control Register, also in the DEVCTL function block.

The DEV may have to cover part of MMIO space beyond the DRAM. Especially in 64-bit systems, the operating system should map MMIO space starting immediately after the DRAM area and building up, as opposed to starting down from the maximum physical address.

Host Bridge and Processor DEV Caching. For improved performance, the host bridge may cache portions of the DEV. Any such cached information can be invalidated by setting the DEV_FLUSH flag in the DEV control register to 1. Software must set this flag after modifying DEV contents to ensure that the protection logic uses the updated values. The host bridge automatically clears this flag when the flush operation completes. After setting this flag, software should monitor it until it has cleared, in order to synchronize DEV updates with subsequent activity.

By default, the host bridge probes the processor caches for the latest data when it accesses the DEV in DRAM. However, it is possible to disable probing by means of the DEV_CR register ("DEV_CR Register" on page 568); this is recommended in the case of unified memory architecture (UMA) graphics systems. If cache probing is disabled, host bridge reads of the DEV will not check processor caches for more recent copies. This requires software on the CPU to map the memory containing the DEV as uncacheable (UC) or write-through (WT). Alternatively, software must perform a CLFLUSH before it can expect a change to the DEV to be visible by the Northbridge (and before software flushes the DEV cache in the host controller).

[AMD Public Use]

Multiprocessor Issues. Device-originated memory requests are checked against the DEV at the point of entry to the system—the Northbridge to which the device is physically attached. Each Northbridge can have its own set of domains, device-to-domain mappings, and DEV tables (e.g., domain #2 on one node can encompass different devices, and can have different access rights than domain #2 on another node). Thus, the number of protection domains available to software can scale with the number of Northbridges in the system.

15.24.3 Access Checking

Memory Space Accesses. When a memory-space read or write request is received on an external host bridge port, the host bridge maps the HyperTransport bus device ID to a protection domain number, which in turn selects the DEV defining the access permissions for the device (see Figure 15-6). The host bridge then checks the memory address against the DEV contents by indexing into the DEV with the PFN portion of the address (bits 39:12). The PFN is used as a bit index within the DEV. If the bit read from the DEV is set to 1, the host bridge inhibits the access by returning all ones for the data for a read request, or suppressing the store operation on a write request. A Master Abort error response will be returned to the requesting device.

Peer-to-peer memory accesses routed up to the host bridge are also subjected to checks against the DEV. Peer-to-peer transfers that may be occurring behind bridges are not checked.

DEV checks are applied before addresses are translated by the GART. The DEV table is never consulted by accesses originating in the CPU.

I/O Space Accesses. The host bridge can be configured to reject all I/O space accesses from devices, by setting the IOSPE bit in the DEV_CR control register (see “DEV_CR Register” on page 568). I/O space peer-to-peer transfers behind bridges are not checked.

Config Space Accesses. Major aspects of host bridge functionality are configured by means of control registers that are accessed through PCI configuration space. Because this is potentially accessible by means of device peer-to-peer transfers, the host bridge always blocks access to this space from anything other than the CPU.

[AMD Public Use]

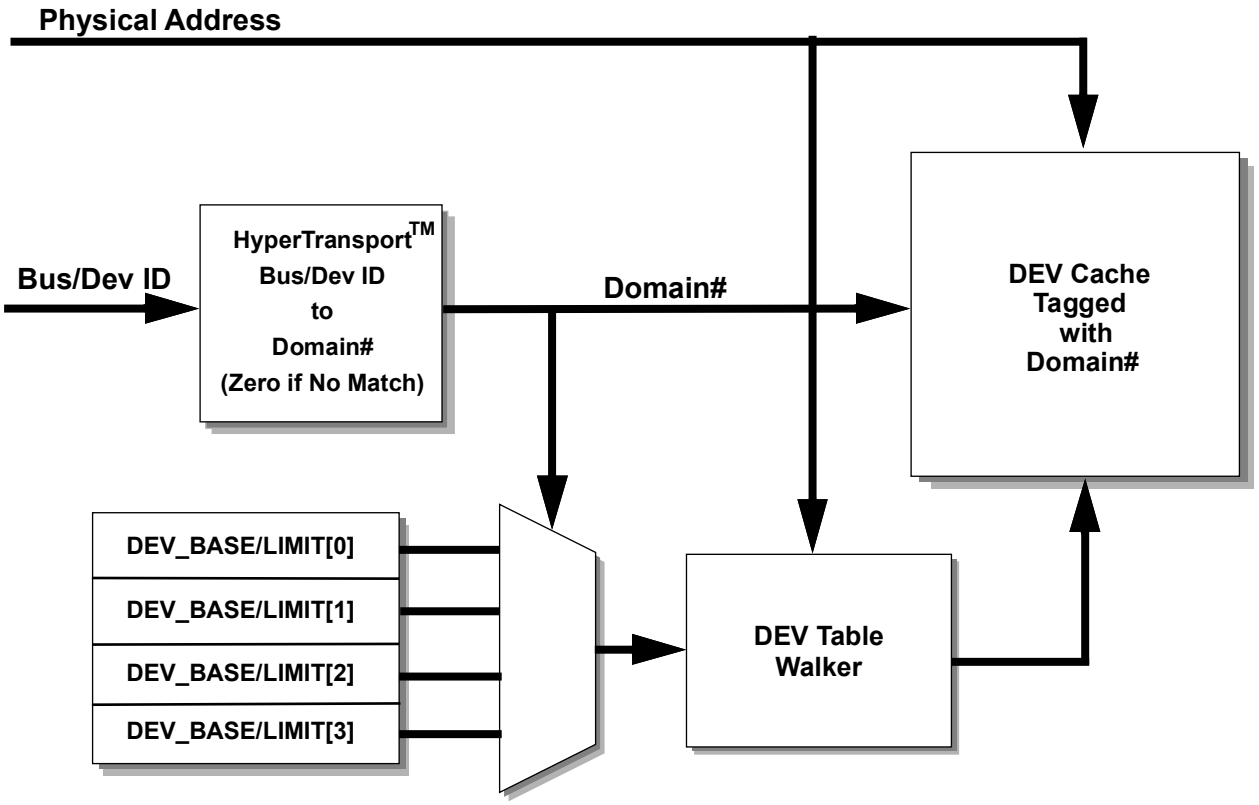


Figure 15-6. Host Bridge DMA Checking

15.24.4 DEV Capability Block

The presence of DEV support is indicated through a new PCI capability block. The capability block also provides access to the registers that control operation of the DEV feature.

The DEV capability block in PCI space contains three 32-bit words: the capability header (DEV_HDR), and two registers (DEV_OP and DEV_DATA) which serve as an indirection mechanism for accessing the actual DEV control and status registers.

Table 15-15. DEV Capability Block, Overall Layout

Byte Offset	Register	Comments
0	DEV_HDR	Capability block header
4	DEV_OP	Selects control/status register to access
8	DEV_DATA	Read/write to access register selected in DEV_OP

DEV Capability Header. The DEV capability header (DEV_HDR) is defined in Table 15-16.

[AMD Public Use]

Table 15-16. DEV Capability Header (DEV_HDR) (in PCI Config Space)

Bit(s)	Definition
31:22	Reserved, MBZ
21	Interrupt Reporting Capability
20	Machine Check Exception Reporting Capability
19	Reserved, MBZ
18:16	DEV Capability Block Type; hardwired to 000b.
15:8	PCI Capability pointer; points to next capability in list
7:0	PCI Capability ID; hardwired to 0x0F

15.24.5 DEV Register Access Mechanism

The Northbridge's DEV control and status registers are accessed through an indirection mechanism: writing the DEV_OP register selects which internal register is to be accessed, and the DEV_DATA register can be read or written to access the selected register.

Figure 15-7 shows the format of the DEV_OP register. The DEV_DATA register reflects the format of the DEV register selected in DEV_OP.

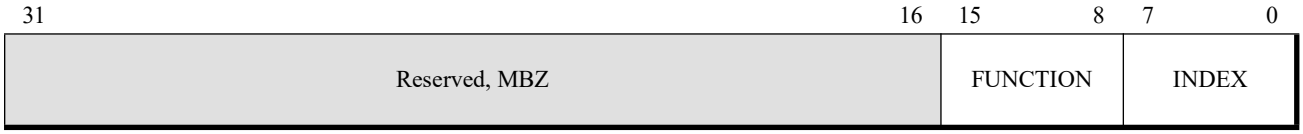


Figure 15-7. Format of DEV_OP Register (in PCI Config Space)

The FUNCTION field in the DEV_OP register selects the function/register to read or write according to the encoding in Table 15-17; for blocks of registers that have multiple instances (e.g., multiple DEV_BASE_HI/LO registers), the INDEX field selects the instance; otherwise it is ignored.

Table 15-17. Encoding of Function Field in DEV_OP Register

Function Code	Register Type	Number of Instances
0	DEV_BASE_LO	multiple
1	DEV_BASE_HI	multiple
2	DEV_MAP	multiple
3	DEV_CAP	single
4	DEV_CR	single
5	DEV_ERR_STATUS	single
6	DEV_ERR_ADDR_LO	single
7	DEV_ERR_ADDR_HI	single

[AMD Public Use]

For example, to write the DEV_BASE_HI register for protection domain number 2, software sets DEV_OP.FUNCTION to 1, and DEV_OP.INDEX to 2, and then writes the desired 32-bit value into DEV_DATA. As the DEV_OP and DEV_DATA registers are accessed through PCI config space (ports 0CF8h–0CFFh), they may be secured from unauthorized access by software executing on the processor by appropriate settings in the SVM I/O protection bitmap. These registers are also protected by the host bridge from external access as described in “Config Space Accesses” on page 565.

15.24.6 DEV Control and Status Registers

The DEV control and status registers are accessible by means of the indirection mechanism; these registers are *not* directly visible in PCI config space.

DEV_CAP Register. Read-only register; holds implementation specific information: the number of protection domains supported, the number of DEV_MAP registers (which map device/unit IDs to domain numbers), and the revision ID.

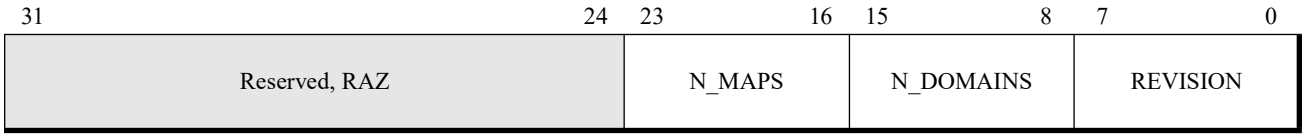


Figure 15-8. Format of DEV_CAP Register (in PCI Config Space)

The initial implementation provide four domains and three map registers.

DEV_CR Register. This is the main control register for the DEV mechanism; it is cleared to zero by RESET.

Table 15-18. DEV_CR Control Register

Bit(s)	Definition
31:7	Reserved, MBZ
6	DEV table walk probe disable. 0 = Use probe on DEV walk; 1 = Do not use probe
5	SL_DEV_EN. Enable bit for limited memory protection, see Section 15.24.8 on page 570. Set to “1” by SKINIT instruction, can be cleared by software.
4	Invalidate DEV cache. Software must set this bit to 1 to invalidate the DEV cache; cleared by hardware when invalidation is complete.
3	Enable MCE reporting. 0 = Do not generate MCE; 1 = Generate MCE on errors.
2	I/O space protection enable (IOSPEN) 0 = Allow upstream I/O cycles; 1 = Block.
1	Memory clear disable. If non-zero, memory-clearing on reset is disabled. This bit is not writable until the memory is enabled.
0	DEV global enable bit. If zero, DEV protection is turned off.

[AMD Public Use]

DEV_BASE Address/Limit Registers. The DEV base address registers (one set per domain) each point to the physical address of a DEV table corresponding to a protection domain. The address and size are encoded in a pair (high/low) of 32-bit registers. The N_DOMAINS field in DEV_CAP indicates how many (pairs of) DEV_BASE registers are implemented. The register format is as shown in Figures 15-9 and 15-10.

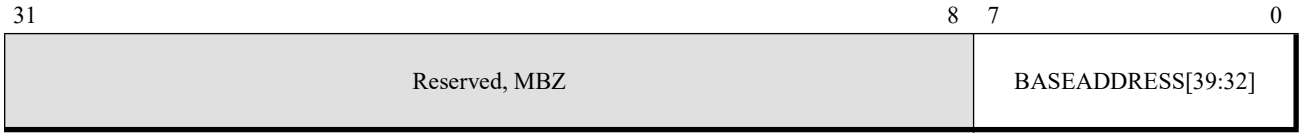


Figure 15-9. Format of DEV_BASE_HI[n] Registers

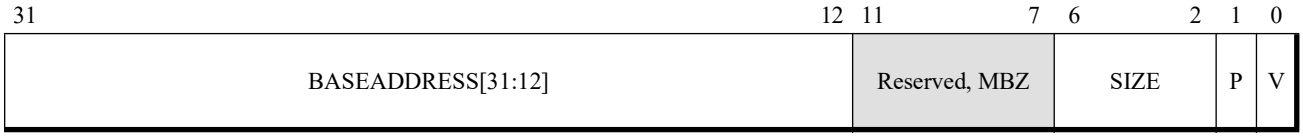


Figure 15-10. Format of DEV_BASE_LO[n] Registers

Fields of the DEV_BASE_HI and DEV_BASE_LO registers are defined as follows:

- *Valid (V)*—Bit 0. Indicates whether a DEV table has been defined for the given protection domain; if this bit is clear, software can leave the other fields undefined, and no protection checks are performed for memory references in this domain.
- *Protect (P)*—Bit 1. Indicates whether accesses to addresses beyond the address range covered by the DEV are legal (P=0) or illegal (P=1).
- *SIZE*—Bits 6:2. Specifies how much memory the DEV covers, expressed increments of 4GB * 2^{size}. In other words, a DEV table covers a minimum of 4GB, and can expand by powers of two.

DEV_MAP Registers. The DEV_MAP registers assign protection domain numbers to device-originated requests by matching the device ID (HT bus and unit number) associated with the request against bus and unit numbers in the registers. If no match is found in any of the registers, a domain number of zero is returned. The number of DEV_MAP registers implemented by the chip is indicated by the N_MAPS field in DEV_CAP.

The format of the DEV_MAP registers is shown in Figure 15-11.

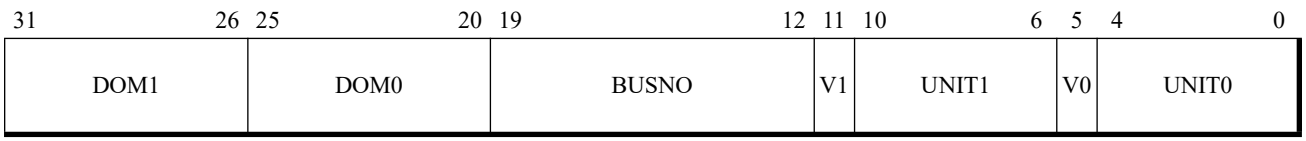


Figure 15-11. Format of DEV_MAP[n] Registers

[AMD Public Use]

The fields of the DEV_MAP[*n*] registers are defined as follows:

- UNIT0—Bits 4:0. Specifies the first of two HyperTransport link unit numbers on the bus number specified by the BUSNO field.
- V0—Bit 5. Indicates whether UNIT0 is valid (no matches occur on invalid entries).
- UNIT1—Bits 10:6. Specifies the second of two HyperTransport link unit numbers on the bus number specified by the BUSNO field.
- V1—Bit 11. Indicates whether UNIT1 is valid (no matches occur on invalid entries).
- BUSNO—Bits 19:12. Specifies a HyperTransport link bus number.
- DOM0—Bits 25:20. Specifies the protection domain for the first HyperTransport link unit.
- DOM1—Bits 31:26. Specifies the protection domain for the second HyperTransport link unit.

15.24.7 Unauthorized Access Logging

Any attempted unauthorized access by devices to DEV-protected memory is logged by the host bridge in the DEV_Error_Status and DEV_Error_Address registers for possible inspection by the VMM.

15.24.8 Secure Initialization Support

The host bridge contains additional logic that operates in conjunction with the SKINIT instruction to provide a limited form of memory protection during the secure startup protocol. This provides protection for a Secure Loader image in memory, allowing it to, among other things, set up full DEV protection. (See Section 15.27 for detailed operation of SKINIT.)

The host bridge logic includes a hidden (not accessible to software) SL_DEV_BASE address register. SL_DEV_BASE points to a 64KB-aligned 64KB region of physical memory. When SL_DEV_EN is 1, the 64KB region defined by SL_DEV_BASE is protected from external access (as if it were protected by the DEV), as well as from any access (both CPU and external accesses) via GART-translated addresses. Additionally, the SL_DEV mechanism, when enabled, blocks all device accesses to PCI Configuration space.

15.25 Nested Paging

The optional SVM nested paging feature provides for two levels of address translation, thus eliminating the need for the VMM to maintain shadow page tables.

15.25.1 Nested Paging Overview

Figure 15-12 shows how a page in the linear address space is mapped to a page in the physical address space in single-level address translation. Control register CR3 contains the physical address of the base of the page tables (PT, represented by the shaded box in the figure), which governs the address translation.

[AMD Public Use]

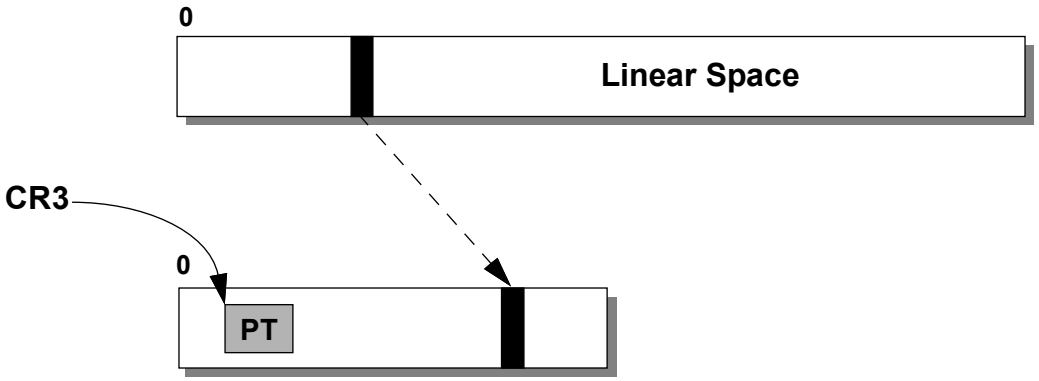


Figure 15-12. Address Translation with Single-Level Paging

With nested paging enabled, *two* levels of address translation are applied; refer to Figure 15-13 below.

- Both guest and host levels have their own copy of CR3, referred to as gCR3 and nCR3, respectively.
- Guest page tables (gPT) map guest linear addresses to guest physical addresses. The guest page tables are in guest physical memory, and are pointed to by gCR3.
- Nested page tables (nPT) map guest physical addresses to system physical addresses. The nested page tables are in system physical memory, and are pointed to by nCR3.
- The most-recently used translations from guest linear to system physical address are cached in the TLB and used on subsequent guest accesses.

It is important to note that gCR3 and the guest page table entries contain guest physical addresses, not system physical addresses. Hence, before accessing a guest page table entry, the table walker first translates that entry's guest physical address into a system physical address.

[AMD Public Use]

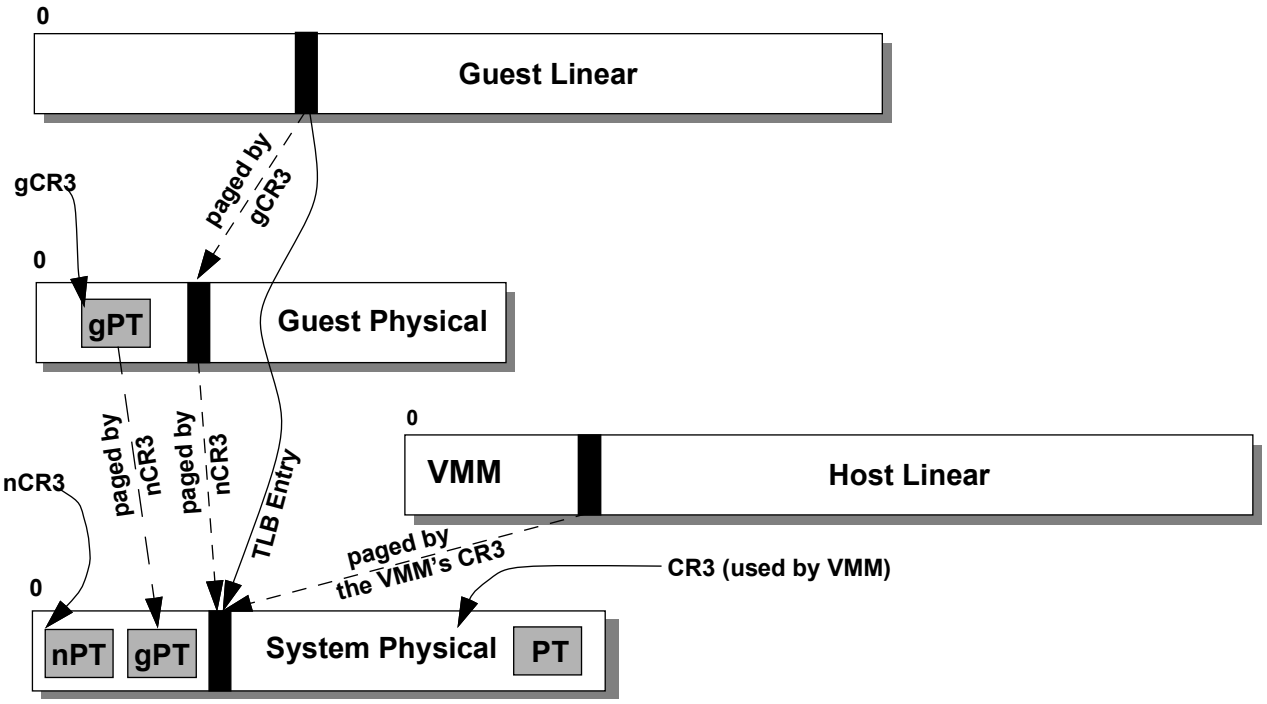


Figure 15-13. Address Translation with Nested Paging

The VMM can give each guest a different ASID, so that TLB entries from different guests can coexist in the TLB. The ASID value of zero is reserved for the host; if the VMM attempts to execute VMRUN with a guest ASID of zero, the result is #VMEXIT(VMEXIT_INVALID). Note that because an ASID is associated with the guest's physical address space, it is common across all of the guest's virtual address spaces within a processor. This differs from shadow page tables where ASIDs tag individual guest virtual address spaces. Note also that the same ASID may or may not be associated with the same address space across all processors in a multiprocessor system, for either nested tables or shadow tables; this depends on how the VMM manages ASID assignment.

15.25.2 Replicated State

Most processor state affecting paging is replicated for host and guest. This includes the paging registers CR0, CR3, CR4, EFER and PAT. CR2 is not replicated but is loaded by VMRUN. The MTRRs are not replicated.

While nested paging is enabled, all (guest) references to the state of the paging registers by x86 code (MOV to/from CR*n*, etc.) read and write the guest copy of the registers; the VMM's versions of the registers are untouched and continue to control the second level translations from guest physical to system physical addresses. In contrast, when nested paging is disabled, the VMM's paging control registers are stored in the host state save area and the paging control registers from the guest VMCB are the only active versions of those registers.

[AMD Public Use]

15.25.3 Enabling Nested Paging

The VMRUN instruction enables nested paging when the NP_ENABLE bit in the VMCB is set to 1. The VMCB contains the hCR3 value for the page tables for the extra translation. The extra translation uses the same paging mode as the VMM used when it executed the most recent VMRUN.

Nested paging is automatically disabled by #VMEXIT.

Nested paging is allowed only if the host has paging enabled. Support for nested paging is indicated by CPUID Fn8000_000A_EDX[NP] = 1. If VMRUN is executed with hCR0.PG cleared to zero and NP_ENABLE set to 1, VMRUN terminates with #VMEXIT(VMEXIT_INVALID). See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

15.25.4 Nested Paging and VMRUN/#VMEXIT

When VMRUN is executed with nested paging enabled (NP_ENABLE = 1), the paging registers are affected as follows:

- VMRUN saves the VMM’s CR3 in the host save area.
- VMRUN loads the guest paging state from the guest VMCB into the guest registers (i.e., VMRUN loads CR3 with the VMCB CR3 field, etc.). The guest PAT register is loaded from G_PAT field in the VMCB.
- VMRUN loads nCR3, the version of CR3 to be used while the nested-paging guest is running, from the N_CR3 field in the VMCB. The other host paging-control bits (hCR4.PAE, etc.) remain the same as they were in the VMM at the time VMRUN was executed.

When VMRUN is executed with nested paging enabled (NP_ENABLE = 1), the following conditions are considered illegal state combinations, in addition to those mentioned in “Canonicalization and Consistency Checks” on page 526:

- Any MBZ bit of nCR3 is set.
- Any G_PAT.PA field has an unsupported type encoding or any reserved field in G_PAT has a non-zero value. (See Section 7.8.1, “PAT Register,” on page 229.)

When #VMEXIT occurs with nested paging enabled:

- #VMEXIT writes the guest paging state (gCR3, gCR0, etc.) back into the VMCB. nCR3 is not saved back into the VMCB.
- #VMEXIT need not reload any host paging state other than CR3 from the host save area, though an implementation is free to do so.

15.25.5 Nested Table Walk

When the guest is running with nested paging enabled, a TLB miss causes several nested table walks:

- Guest Page Tables—The gCR3 register specifies a guest physical address, as do the entries in the guest's page tables. These guest physical addresses must be translated to system physical addresses

[AMD Public Use]

using the nested page tables. Nested page table level faults can occur on these accesses, including write faults due to setting of accessed and dirty bits in the guest page table.

- Final Guest-Physical Page—Once a guest linear to guest physical mapping is known, guest permissions can be checked. If the guest page tables allow the access, the guest physical address is translated through the nested page tables to find the system physical address.

Table walks for guest page tables are by default treated as user writes at the nested page table level. For this reason,

- The page must be writable by user at the nested page table level, or else a #VMEXIT(NPF) is raised, and
- The dirty and accessed bits are always set in the nested page table entries that were touched during nested page table walks for guest page table entries.

A table walk for the guest page itself is always treated as a user access at the nested page table level, but is treated as a data read, data write, or code read, depending on the guest access.

If the guest has paging disabled (gCR0.PG = 0), there are no guest page table entries to be translated in the nested page tables. In this case, the final guest-physical address is equal to the guest-linear address, and is still translated in the nested page tables.

15.25.6 Guest Page Table Protection

Guest page table protection features restrict guest page table accesses at the nested page table level.

15.25.6.1 Read-Only Guest Page Tables

Support for the Read-Only Guest Page Tables (ROGPT) feature is indicated by CPUID Fn8000_000A_EDX[ROGPT] (bit 21) = 1.

ROGPT is enabled by setting bit 6 at offset 90h (RogptEn) in the VMCB. When ROGPT is enabled, guest page table accesses are treated as writes at the nested page table level when the accessed or dirty bit update in a guest page table is required, allowing the hypervisor to map guest page tables into read-only nested pages.

15.25.6.2 Paging-Write

Support for the Paging-Write feature is indicated by CPUID Fn8000_000A_ECX[PW] (bit 1) = 1. Paging-Write is enabled by setting bit 9 at offset 90h (PagingWriteEn) in the VMCB. If PagingWriteEn is set in the VMCB, nested paging must be enabled, and the hypervisor must be in 64-bit mode. If not, the VMRUN will fail with a VMEXIT_INVALID error code.

Paging writes are processor writes to guest paging entries to update accessed and dirty bits. When Paging-Write is enabled, writes for guest page table entries are treated as the paging-write access type at the nested page table level, allowing accessed or dirty bit update in a guest page table if:

- R/W is 1 in nested non-leaf page table entries, and
- PW (Paging-Write, bit 58) is 1 in the nested leaf page table entry.

[AMD Public Use]

Paging-write feature allows the hypervisor to permit accessed and dirty bit updates in a guest page table entry and to prevent guest software from modifying it by setting R/W bit to 0 and PW bit to 1 in the nested leaf page table entry. If the Paging-Write access check fails, a #VMEXIT(NPF) is generated and bit 1 (RW) in EXITINFO1 is set.

The following table defines guest page table access type at the nested page table level.

RogptEn	PagingWriteEn	GPT Read	GPT Write (A/D Bit)
0	0	Write	Write
0	1	Paging-Write	Paging-Write
1	0	Read	Write
1	1	Read	Paging-Write

15.25.6.3 Verify Paging-Write

Support for the Verify Paging-Write feature is indicated by CPUID Fn8000_000A_ECX[VPW] (bit 2) = 1. Verify Paging-Write is enabled by setting bit 10 at offset 90h (VerifyPagingWriteEn) in the VMCB. If VerifyPagingWriteEn is set in the VMCB, nested paging must be enabled, and the hypervisor must be in 64-bit mode. If not, the VMRUN will fail with a VMEXIT_INVALID error code.

When Verify Paging-Write is enabled, the processor checks that the final guest physical address with VPW (Verify Paging-Write, bit 57) set to 1 in the nested leaf is translated with guest pages that have the PW bit set in their nested leaf entries.

If the Verify Paging-Write check fails, a #VMEXIT(NPF) is generated and bit 38 (VPW) in EXITINFO1 is set.

15.25.7 Nested Page Faults and Fault Ordering

In nested paging, page faults can be raised at either the guest or nested page table level. Nested walks proceed in the following order; faults are generated in the same order:

1. Walk the guest page table entries in the nested page table. Dirty/Accessed bits are set as needed in the nested page table. Any nested page table faults result in #VMEXIT(NPF).
2. As the guest page table walk proceeds from the top of the page table to the last entry, any not-present entries or reserved bits in the guest page table entries at each level of the guest walk cause #PF in the guest. Guest dirty and accessed bits are set as needed in the guest page tables during the walk. Steps 1 and 2 are repeated for each level of the guest page table that is traversed.
3. Once the guest physical address for the guest access has been determined, check the guest permissions; any fault at this point causes a #PF in the guest.
4. Perform the final translation from guest physical to system physical using the nested page table; any fault during this translation results in a #VMEXIT(NPF).

[AMD Public Use]

Nested page faults are entirely a function of the nested page table and VMM processor mode. Nested faults cause a #VMEXIT(NPF) to the VMM. The faulting guest physical address is saved in the VMCB's EXITINFO2 field; EXITINFO1 delivers an error code similar to a #PF error code:

- Bit 0 (P)—Cleared to 0 if the nested page was not present, 1 otherwise.
- Bit 1 (RW)—Set to 1 if the nested page table level access was a write.
- Bit 2 (US)—Set to 1 if the nested page table level access was a user access. Note that nested page table accesses performed by the MMU are treated as user accesses unless there are features enabled that override this.
- Bit 3 (RSV)—Set to 1 if reserved bits were set in the corresponding nested page table entry.
- Bit 4 (ID)—Set to 1 if the nested page table level access was a code read. Note that nested table walks for guest page tables are always treated as data writes, even if the access itself is a code read.
- Bit 6 (SS) - Set to 1 if the fault was caused by a shadow stack access.

EXITINFO1 for nested page faults reports the following additional information:

- Bit 32—Set to 1 if nested page fault occurred while translating the final guest physical address.
- Bit 33—Set to 1 if nested page fault occurred while translating the guest page tables.
- Bit 37—Set to 1 if the page was marked as a supervisor shadow stack page in the leaf node of the nested page table and the supervisor shadow stack check feature is enabled.
- Bit 38—Set to 1 if nested page fault was caused by verify paging-write check.

Guest faults are entirely a function of the guest page tables and processor mode. They are delivered to the guest as normal #PF exceptions without any VMM intervention, unless the VMM is intercepting guest #PF exceptions.

See Section 15.36.10, “RMP and VMPL Access Checks,” on page 634 for SNP related #VMEXIT(NPF) EXITINFO1 field definitions.

The processor may provide additional instruction decode assist information. See Section 15.10.

15.25.8 Combining Nested and Guest Attributes

Any access to guest physical memory is subjected to a permission check by examining the mapping of the guest physical address in the nested page table.

A page is considered writable by the guest only if it is marked writable at both the guest and nested page table levels. Note that the guest’s gCR0.WP affects only the interpretation of the guest page table entry; setting gCR0.WP cannot make a page writable at any CPL in the guest, if the page is marked read-only in the nested page table. The host hCR0.WP bit is ignored under nested paging.

A page is considered executable by the guest only if it is marked executable at both the guest and nested page table levels. If the EFER.NXE bit is cleared for the guest, all guest pages are executable at the guest level. Similarly, if the EFER.NXE bit is cleared for the host, all nested page table mappings are executable at the underlying nested level.

[AMD Public Use]

Some attributes are taken from the guest page tables and operating modes only. A page is considered global within the guest only if is marked global in the guest page tables; the nested page table entry and host hCR4.PGE are irrelevant. Global pages are only global within their ASID.

A page is considered user in the guest only if it is marked as user at the guest level. The page must be marked user in the nested page table to allow any guest access at all.

15.25.9 Combining Memory Types, MTRRs

When nested paging is disabled, the processor behaves as though there is no gPAT register.

The host PAT MSR determines memory type attributes for the current VM, and guest writes to the PAT MSR that aren't intercepted by the VMM will alter the host PAT MSR. The hypervisor is responsible for context-switching the PAT MSR contents on world switches between VMs.

When nested paging is enabled, the processor combines guest and nested page table memory types. Registers that affect memory types include:

- The PCD/PWT/PAT*i* bits in the nested and guest page table entries.
- The PCD/PWT bits in the nested CR3 and guest CR3 registers.
- The guest PAT type (obtained by appropriately indexing the gPAT register).
- The host PAT type (obtained by appropriately indexing the host’s PAT register).
- The MTRRs (which are referenced based only on system physical address).
- gCR0.CD and hCR0.CD.

Note that there is no hardware support for guest MTRRs; the VMM can simulate their effect by altering the memory types in the nested page tables. Note that the MTRRs are only applied to system physical addresses.

The rules for combining memory types when constructing a guest TLB entry are:

- Nested and guest PAT types are combined according to Table 15-19, producing a “combined PAT type.”
- Combined PAT type is further combined with the MTRR type according to Table 15-20, where the relevant MTRRs are determined by the system physical address.
- Either gCR0.CD or hCR0.CD can disable caching.

Memory Consistency Issues. Because the guest uses extra fields to determine the memory type, the VMM may use a different memory type to access a given piece of memory than does the guest. If one access is cacheable and the other is not, the VMM and guest could observe different memory images, which is undesirable. (MP systems are particularly sensitive to this problem when the VMM desires to migrate a virtual processor from one physical processor to another.)

To address this issue, the following mechanisms are provided:

[AMD Public Use]

- VMRUN and #VMEXIT flush the write combiners. This ensures that all writes to WC memory by the guest are visible to the host (or vice-versa) regardless of memory type. (It does not ensure that cacheable writes by one agent are properly observed by WC reads or writes by the other agent.)
- A new memory type *WC+* is introduced. WC+ is an uncacheable memory type, and combines writes in write-combining buffers like WC. Unlike WC (but like the CD memory type), accesses to WC+ memory also snoop the caches on all processors (including self-snooping the caches of the processor issuing the request) to maintain coherency. This ensures that cacheable writes are observed by WC+ accesses.
- When combining nested and guest memory types that are incompatible with respect to caching, the WC+ memory type is used instead of WC (and Table 15-20 ensures that the snooping behavior is retained regardless of the host MTRR settings). Refer to Table 15-19 or details.

Table 15-19 shows how guest and host PAT types are combined into an effective PAT type. When interpreting this table, recall (a) that guest and host PAT types are not combined when nested paging is disabled and (b) that the intent is for the VMM to use its PAT type to simulate guest MTRRs.

Table 15-19. Combining Guest and Host PAT Types

		Host PAT Type					
		UC	**UC–**	**WC**	**WP**	**WT**	**WB**
Guest PAT Type	**UC**	UC	UC	UC	UC	UC	UC
	UC–	UC	UC–	WC	UC	UC	UC
	WC	WC	WC	WC	WC+	WC+	WC+
	WP	UC	UC	UC	WP	UC	WP
	WT	UC	UC	UC	UC	WT	WT
	WB	UC	UC	WC	WP	WT	WB

The existing AMD64 table that defines how PAT types are combined with the physical MTRRs is extended to handle CD and WC+ PAT types as shown in Table 15-20.

Table 15-20. Combining PAT and MTRR Types

		MTRR Type				
		UC	**WC**	**WP**	**WT**	**WB**
Effective PAT Type	**UC**	UC	CD	CD	CD	CD
	UC–	UC	WC	CD	CD	CD
	WC	WC	WC	WC	WC	WC
	WC+	WC	WC	WC+	WC+	WC+
	WP	UC	CD	WP	CD	WP
	WT	UC	CD	CD	WT	WT
	WB	UC	WC	WP	WT	WB

[AMD Public Use]

15.25.10 Page Splintering

When an address is mapped by guest and nested page table entries with different page sizes, the TLB entry that is created matches the size of the smaller page.

15.25.11 Legacy PAE Mode

The behavior of PAE mode in a nested-paging guest differs slightly from the behavior of (host-only) legacy PAE mode, in that the guest's four PDPEs are not loaded into the processor at the time CR3 is written. Instead, the PDPEs are accessed on demand as part of a table walk. This has the side-effect that illegal bit combinations in the PDPEs are not signaled at the time that CR3 is written, but instead when the faulty PDPE is accessed as part of a table walk.

This means that an operating system cannot rely on the behavior when the in-memory PDPEs are different than the in-processor copy.

15.25.12 A20 Masking

There is no provision for applying A20 masking to guest physical addresses; the VMM can emulate A20 masking by changing the nested page mappings accordingly.

15.25.13 Detecting Nested Paging Support

Nested Paging is an optional feature of SVM and is not available in all implementations of SVM-capable processors. The CPUID instruction should be used to detect nested paging support on a particular processor. See Section 3.3, "Processor Feature Identification," on page 73 for more information on using the CPUID instruction.

15.25.14 Guest Mode Execute Trap Extension

The Guest Mode Execute Trap (GMET) extension allows a hypervisor to cause nested page faults on attempts by a guest to execute code at CPL0, 1 or 2 from pages designated by the hypervisor. The presence of the GMET extension is indicated by CPUID Fn8000_000A EDX[17]=1. The GMET mode is selected for a targeted guest by setting bit 3 of VMCB offset 090h to 1. For processors that don't support GMET this bit is ignored.

On GMET capable processors, when this bit is set to 1 on a VMRUN, the processor changes how the U/S bit in the nested page table is interpreted. The NX bit still prohibits execution of code at any privilege level when set to 1. However, with GMET enabled and the effective NX bit =0, if the effective U/S bit =1 and the page is being accessed for execution at CPL0, 1 or 2, a nested page fault #VMEXIT(NPF) is generated. If the effective NX bit =0 and the effective U/S bit =0 then the

[AMD Public Use]

translation is allowed for the code page. The following table summarizes the behavior when GMET is enabled.

Table 15-21. GMET Page Configuration

nPT NX Bit	nPT U/S Bit	Guest User-Mode Code	Guest Supervisor-Mode Code
1	X	No Execute	No Execute
0	1	Execute	No Execute
0	0	Execute	Execute

The EXITINFO1 field for the nested page fault contains the page fault error code describing attributes of the attempted translation that caused the fault. A GMET violation is not explicitly indicated with a separate bit. It is up to software to determine if it was NX based or GMET based by inspecting this error code along with the faulting page's effective NX and U/S settings.[1]

15.25.15 Supervisor Shadow Stacks

The Supervisor Shadow Stack (SSS) feature is an extension to nested paging which allows a hypervisor to restrict which guest physical addresses may be used for a guest supervisor shadow stack. Supervisor shadow stack accesses made by the guest to pages not designated as SSS pages in the nested page tables result in a #VMEXIT(NPF).

Determining Support for SSS. Support for the SSS feature is indicated by CPUID Fn8000_000A_EDX[19](SupervisorShadowStack)=1.

Enabling SSS. The SSS feature is enabled by setting bit 4 in VMCB offset 90h. The SSS feature can be enabled only if nested paging is enabled in the VMCB and the PAE and No Execute paging modes (EFER.NXE=1) are enabled in the host.

Attempting to execute a VMRUN with SSS enabled and nested paging disable result in a VMEXIT(INVALID). If the host is not legacy non-PAE mode or EFER.NXE=0, attempts to enable the SSS feature are silently ignored.

The SSS feature can be enabled regardless of whether the guest has enabled shadow stacks or not.

Designating SSS Pages. When the SSS feature is enabled, the hypervisor indicates a page may be used for a supervisor shadow stack using following combination of nested page table bits:

- NX=1 and U/S=0 in the final nested page table entry used to translate the address.
- R/W=1 in all other nested non-leaf page table entries leading to the final nested page table entry.

Although is not enforced by the SSS feature, R/W should be 0 in the final nested page table entry in order to achieve the desired security functionality.

[1] The guest user/supervisor indication is normally provided in ExitInfo1, however on some implementations a GMET erratum may require CPL to be read from the guest VMCB.

[AMD Public Use]

SSS Access Checking. When the SSS feature is enabled, guest supervisor shadow stack accesses are allowed only to physical pages designated as SSS pages in the nested page tables. Note that supervisor shadow stack writes to SSS pages are allowed to complete even though R/W=0 in the final nested page table entry.

The following accesses to SSS pages are not allowed:

- Supervisor shadow accesses made to non-SSS pages. These result in a #VMEXIT(NPF) with the SS bit set in the EXITINFO1 error code.
- Attempting to execute code from an SSS page. This results in a #VMEXIT(NPF) the same as any page with NX=1.

See Chapter 15.25.7, “Nested Page Faults and Fault Ordering,” on page 575 for more information on EXITINFO1 error codes for nested page faults.

15.25.16 Page Modification Logging

Processors that set CPUID Fn8000_0000A_ECX[PML] (bit 4) = 1 support Page Modification Logging (PML). With PML, the processor logs guest physical addresses that set the Dirty bit of a nested page-table entry.

PML is enabled on VMRUN when bit 11 in VMCB control area at offset 90h is set and nested paging is enabled. When a guest memory write access sets the Dirty bit of a nested page table entry, the processor writes the 4-Kbyte aligned guest physical address (bits 11:0 cleared) to the PML buffer. The PML buffer is a 4-Kbyte memory region located at the PML_BASE address configured at offset 1C8h in the VMCB control area. The processor maintains a PML buffer index which points to the next PML buffer entry to be written. Before the write access is allowed and the nested page table Dirty bit is set, the processor checks if the PML buffer index is in the range between 0 and 1FFh. If it is, the processor allows the write access, sets the Dirty bit and writes the guest physical address (as specified above) into the PML buffer. Specifically, to address PML_BASE + PML_INDEX * 8. The PML index is then decremented. On #VMEXIT the current PML index is written to PML_INDEX (offset 1D0h) in the VMCB control area.

If the PML buffer index is outside the range between 0 and 1FFh, the write access is not performed, and the Dirty bit is not set. Instead, the processor performs a #VMEXIT with exit code VMEXIT_PML_FULL (407h). VMEXIT_PML_FULL does not advance the rIP in the guest and is AE (Automatic Exit) for SEV-ES and SEV-SNP guests.

After handling the VMEXIT_PML_FULL event and when configuring PML, hypervisor software should write 1FFh to PML_INDEX (offset 1D0h in the VMCB control area) to empty the buffer.

Since PML also tracks writes to guest page table entries, a single guest write access may result in multiple guest physical addresses being logged.

[AMD Public Use]

15.26 Security

SVM provides additional hardware support that is designed to facilitate the construction of trusted software systems. While the security features described in this section are orthogonal to SVM's virtualization support (and are not required for processor virtualization), the two form building blocks for trusted systems.

SKINIT Instruction. The SKINIT instruction and associated system support (the Trusted Platform Module or TPM) are designed to allow for verifiable startup of trusted software (such as a VMM), based on secure hash comparison.

Security Exception. A security exception (#SX) is used to signal certain security-critical events.

15.27 Secure Startup with SKINIT

The SKINIT instruction is one of the keys to creating a "root of trust" starting with an initially untrusted operating mode. SKINIT reinitializes the processor to establish a secure execution environment for a software component called the secure loader (SL) and starts execution of the SL in a way that cannot be tampered with. SKINIT also copies the secure loader executable image to an external device, such as a Trusted Platform Module (TPM) for verification using unique bus transactions that preclude SKINIT operation from being emulated by software in a way that the TPM could not readily detect. (Detailed operation is described in Section 15.27.4.)

15.27.1 Secure Loader

A secure loader (SL) typically initializes SVM hardware mechanisms and related data structures, and initiates execution of a trusted piece of software such as a VMM (referred to as a Security Kernel, or SK, in this document), after first having validated the identity of that software.

SKINIT allows SVM protections to be reliably enabled after the system is already up and running in a non-trusted mode — there is no requirement to change the typical x86 platform boot process.

Exact details of the hand-off from the SL to an SK are dependent on characteristics of the SL, SK and the initial untrusted operating environment. However, there are specific requirements for the SL image, as described in Section 15.27.2.

15.27.2 Secure Loader Image

The secure loader (SL) image contains all code and initialized data sections of a secure loader. This code and initial data are used to initialize and start a security kernel in a completely safe manner, including setting up DEV protection for memory allocated for use by SL and SK. The SL image is loaded into a region of memory called the secure loader block (SLB) and can be no larger than 64Kbyte (see Section 15.27.3). The SL image is defined to start at byte offset 0 in the SLB.

The first word (16 bits) of the SL image must specify the SL entry point as an unsigned offset into the SL image. The second word must contain the length of the image in bytes; the maximum length

[AMD Public Use]

allowed is 65535 bytes. These two values are used by the SKINIT instruction. The layout of the rest of the image is determined by software conventions. The image typically includes a digital signature for validation purposes. The digital signature hash must include the entry point and length fields. SKINIT transfers the SL image to the TPM for validation prior to starting SL execution (see Section 15.27.6 for further details of this transfer). The SL image for which the hash is computed must be ready to execute without prior manipulation.

15.27.3 Secure Loader Block

The secure loader block is a 64Kbyte range of physical memory which may be located at any 64Kbyte-aligned address below 4Gbyte. The SL image must have been loaded into the SLB starting at offset 0 before executing SKINIT. The physical address of the SLB is provided as an input operand (in the EAX register) to SKINIT, which sets up special protection for the SLB against device accesses (i.e., the DEV need not be activated yet).

The SL must be written to execute initially in flat 32-bit protected mode with paging disabled. A base address can be derived from the value in EAX to access data areas within the SL image using base+displacement addressing, to make the SL code position-independent.

Memory between the end of the SL image and the end of the SLB may be used immediately upon entry by the SL as secure scratch space, such as for an initial stack, before DEV protections are set up for the rest of memory. The amount of space required for this will limit the maximum size of the SL image, and will depend on SL implementation. SKINIT sets the ESP register to the appropriate top-of-stack value (EAX + 10000h).

Figure 15-14 illustrates the layout of the SLB, showing where EAX and ESP point after SKINIT execution. Labels in italics indicate suggested uses; other labels reflect required items.

[AMD Public Use]

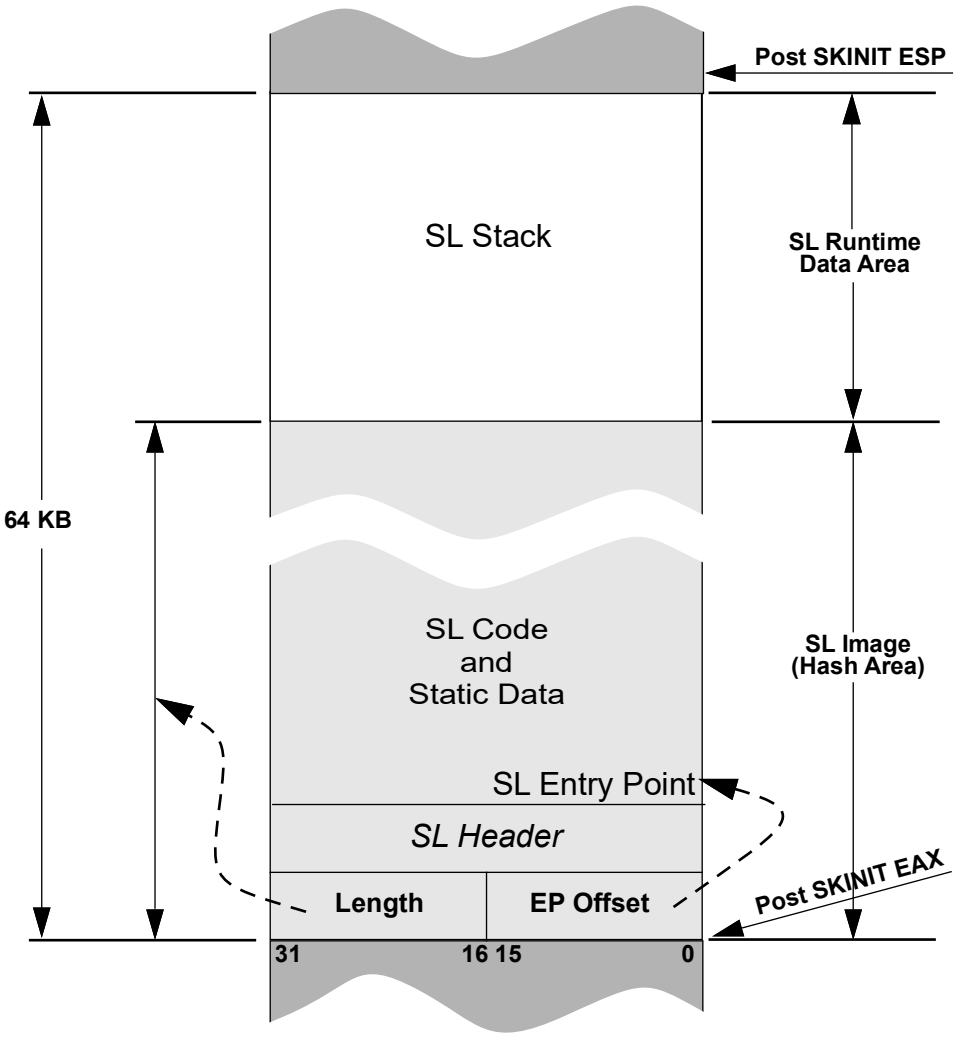


Figure 15-14. SLB Example Layout

15.27.4 Trusted Platform Module

The trusted platform module, or TPM, is an essential part of full trusted system initialization. This device is attached to an LPC link off the system I/O hub. It recognizes special SKINIT transactions, receives the SL image sent by SKINIT and verifies the signature. Based on the outcome, the device decides whether or not to cooperate with the SL or subsequent SK. The TPM typically contains sealed storage containing cryptographic keys and other high-security information that may be specific to the platform.

[AMD Public Use]

15.27.5 System Interface, Memory Controller and I/O Hub Logic

SKINIT uses special support logic in the processor's system interface unit, the internal controller and the I/O hub to which the TPM is attached. SKINIT uses special transactions that are unique to SKINIT, along with this support logic, designed to securely transmit the SL Image to the TPM for validation.

The use of this special protocol is intended to allow the TPM to detect true execution, as opposed to emulation, of a trusted Secure Loader, which in turn provides a means for verifying the subsequent loading and startup of a trusted Security Kernel.

15.27.6 SKINIT Operation

The SKINIT instruction is intended to be used primarily in normal mode prior to the VMM taking control.

SKINIT takes the physical base address of the SLB as its only input operand in EAX, and performs the following steps:

1. Reinitialize processor state in the same manner as for the INIT signal, then enter flat 32-bit protected mode with paging off. The CS selector is set to 8h and CS is read only. The SS selector is set to 10h and SS is read/write and expand-up. The CS and SS bases are cleared to 0 and limits are set to 4G. DS, ES, FS and GS are left as 16-bit real mode segments and the SL must reload these with protected mode selectors having appropriate GDT entries before using them. Initialized data in the SLB may be referenced using the SS segment override prefix until DS is reloaded. The general purpose registers are cleared except for EAX, which points to the start of the secure loader, EDX, which contains model, family and stepping information, and ESP, which contains the initial stack pointer for the secure loader. Cache contents remain intact, as do the x87 and SSE control registers. Most MSRs also retain their values, except those which might compromise SVM protections. The EFER MSR, however, is cleared. The DPD, R_INIT and DIS_A20M flags in the VM_CR register are unconditionally set to 1.
2. Form the SLB base address by clearing bits 15:0 of EAX (EAX is updated), and enable the SL_DEV protection mechanism (see Section 15.24.8) to protect the 64-Kbyte region of physical memory starting at the SLB base address from any device access.
3. In multiprocessor operation, perform an interprocessor handshake as described in Section 15.27.8.
4. Read the SL image from memory and transmit it to the TPM in a manner that cannot be emulated by software.
5. Signal the TPM to complete the hash and verify the signature. If any failures have occurred along the way, the TPM will conclude that no valid SL was started.
6. Clear the Global Interrupt Flag. This disables all interrupts, including NMI, SMI and INIT and ensures that the subsequent code can execute atomically. If the processor enters the shutdown state (due to a triple fault for instance) while GIF is clear, it can only be restarted by means of a RESET.

[AMD Public Use]

7. Update the ESP register to point to the first byte beyond the end of the SLB (SLB base + 65536), so that the first item pushed onto the stack by the SL will be at the top of the SLB.
8. Add the unsigned 16-bit entry point offset value from the SLB to the SLB base address to form the SL entry point address, and jump to it.

The validation of the SL image by the TPM is a one-way transaction as far as SKINIT is concerned. It does not depend on any response from the TPM after transferring the SL image before jumping to the SL entry point, and initiates execution of the Secure Loader unconditionally. Because of the processor initialization performed, SKINIT does not honor instruction or data breakpoint traps, or trace traps due to EFLAGS.TF.

Pending interrupts. Device interrupts that may be pending prior to SKINIT execution due to EFLAGS.IF being clear, or that assert during the execution of SKINIT, will be held pending until software subsequently sets GIF to 1. Similarly, SMI, INIT and NMI interrupts that assert after the start of SKINIT execution will also be held pending until GIF is set to 1.

Debug Considerations. SKINIT automatically disables various implementation-specific hardware debug features. A debug version of the SL can reenable those features by clearing the VM_CR.DPD flag immediately upon entry.

15.27.7 SL Abort

If the SL determines that it cannot properly initialize a valid SK, it must cause GIF to be set to 1 and clear the VM_CR MSR to re-enable normal processor operation.

15.27.8 Secure Multiprocessor Initialization

The following standard APIC features are used for secure MP initialization:

- The concept of a single Bootstrap Processor (BSP) and multiple Application Processors (APs).
- The INIT interprocessor interrupt (IPI), which puts the target processors into a halted state (INIT state) which is responsive only to a subsequent Startup IPI.
- The Startup IPI causes target processors to begin execution at a location in memory that is specified by the Boot Processor and conveyed along with the Startup IPI. The operation of the processor in response to a Startup IPI is slightly modified to support secure initialization, as described below.

A Startup IPI normally causes an AP to start execution at a location provided by the IPI. To support secure MP startup, each AP responds to a startup IPI by additionally clearing its GIF and setting the DPD, R_INIT and DIS_A20M flags in the VM_CR register if, and only if, the BSP has indicated that it has executed an SKINIT. All other aspects of Startup IPI behavior remain unchanged.

Software Requirements for Secure MP initialization. The driver that starts the SL must execute on the BSP. Prior to executing the SKINIT instruction, the driver must save any processor-specific system register contents to memory for restoration after reinitialization of the APs. The driver should also put all APs in an idle state. The driver must first confirmed that all APs are idle and then it must issue an

[AMD Public Use]

INIT IPI to all APs and wait for its local APIC busy indication to clear. This places the APs into a halted state which is responsive only to a subsequent Startup IPI. APs will still respond to snoops for cache coherency. The driver may execute SKINIT at any time after this point. Depending on processor implementation, a fixed delay of no more than 1000 processor cycles may be necessary before executing SKINIT to ensure reliable sensing of APIC INIT state by the SKINIT.

AP Startup Sequence. While the SL starts executing on the BSP, the APs remain halted in APIC INIT state. Either the SL or the SK may issue the Startup IPI for the APs at whatever point is deemed appropriate. The Startup IPI conveys an 8-bit vector specified by the software that issues the IPI to the APs. This vector provides the upper 8 bits of a 20-bit physical address. Therefore, the AP startup code must reside in the lower 1 Mbyte of physical memory—with the entry point at offset 0 on that particular page.

In response to the Startup IPI, the APs start executing at the specified location in 16-bit real mode. This AP startup code must set up protections on each processor as determined by the SL or SK. It must also set GIF to re-enable interrupts, and restore the pre-SKINIT system context (as directed by the SL or SK executing on the BSP), before resuming normal system operation.

The SL must ensure the integrity of the AP startup sequence, for example by including the startup code in the hashed SL image and setting up DEV protection for it before copying it to the desired area. The AP startup code does not need to (and should not) execute SKINIT. Care must also be taken to avoid issuing another INIT IPI from any processor after the BSP executes SKINIT and before all APs have received a Startup IPI, as this could compromise the integrity of AP initialization.

Pending interrupts. Device interrupts that may be pending on an AP prior to the APIC INIT IPI due to EFLAGS.IF being clear, or that assert any time after the processor has accepted the INIT IPI, will be held pending through the subsequent Startup IPI, and remain pending until software sets GIF to 1 on that AP. Similarly, SMI, INIT, and NMI interrupts that assert after the processor has accepted the INIT IPI will also be held pending until GIF is set to 1.

Aborting MP initialization. In the event that the SL or SK on the BSP decides to abort SVM system initialization for any reason, the following clean-up actions must be performed by SL code executing on each processor before returning control to the original operating environment:

- The BSP and all APs that responded to the Startup IPI must restore GIF and clear VM_CR on each processor for normal operation.
- For each processor that has a distinct memory controller associated with it, the SL_DEV_EN flag in the DEV control register must be cleared in order to restore normal device accessibility to the 64KB SL memory range.

Any secure context created by the SL that should not be exposed to untrusted code should be cleaned up as appropriate before these steps are taken.

[AMD Public Use]

15.28 Security Exception (#SX)

The Security Exception fault signals security-sensitive events that occur while executing the VMM, in the form of an exception so that the VMM may take appropriate action. (A VMM would typically intercept comparable sensitive events in the guest.) Currently, the only use of the #SX is to redirect external INITs into an exception so that the VMM may — among other possibilities — destroy sensitive information before re-issuing the INIT, this time without redirection. The INIT redirection is controlled by the VM_CR.R_INIT bit. (See “INIT Support” on page 558 for more details on INIT and #SX behavior). Note that INIT is gated by the Global Interrupt Flag (GIF), and so will be held pending if asserted while GIF is 0. When GIF transitions to 1, INIT will either take effect or be redirected to #SX, depending on the state of R_INIT.

The #SX exception dispatches to vector 30, and behaves like other fault-class exceptions such as General Protection Fault (#GP). The #SX exception pushes an error code. The only error code currently defined is 1, and indicates redirection of INIT has occurred.

The #SX exception is a contributory fault.

15.29 Advanced Virtual Interrupt Controller

The AMD Advanced Virtual Interrupt Controller (AVIC) is an important enhancement to AMD Virtualization™ Technology (AMD-V). In a virtualized environment, AVIC presents to each guest a virtual interrupt controller that is compliant with the local Advanced Programmable Interrupt Controller (APIC) architecture. See Chapter 16, “Advanced Programmable Interrupt Controller (APIC),” on page 659 for a detailed description of APIC.

15.29.1 Introduction

In a virtualized computer system, each guest operating system needs access to an interrupt controller to send and receive device and interprocessor interrupts. When there is no hardware acceleration, it falls to the virtual machine monitor (VMM) to intercept guest-initiated attempts to access the interrupt controller registers and provide direct emulation of the controller system programming interface allowing the guest to initiate and process interrupts. The VMM uses the underlying physical and virtual interrupt delivery mechanisms of the system to deliver interrupts from I/O devices and virtual processors to the target guest virtual processor and to handle any required end of interrupt processing.

Given the high rate of device and interprocessor interrupt generation in certain scenarios, in particular on server-class systems, the emulation of a local APIC can be a significant burden for the VMM. The AVIC architecture addresses the overhead of guest interrupt processing in a virtualized environment by applying hardware acceleration to the following components of interrupt processing:

- Providing a guest operating system access to performance-critical interrupt controller registers
- Initiating intra- and inter-processor interrupts (IPIs) in and between virtual processors in a guest

Software-initiated Interrupts. Modern operating systems use software interrupts (self-IPIs) to implement software event signaling, inter-process communication and the scheduling of deferred

[AMD Public Use]

processing. System software sets up and initiates these interrupts by writing to control registers of the local APIC. AVIC hardware reduces VMM overhead by providing hardware assist for many of these operations.

Inter-processor Interrupts. Inter-processor interrupts (IPIs) are used extensively by modern operating systems to handle communication between processor cores within a machine (or, in a virtualized environment, between virtual processors within a virtual machine). IPIs are also employed to provide signaling and synchronization for operations such as cross-processor TLB invalidations (also known as TLB shootdowns). AVIC provides hardware mechanisms that deliver the interrupt to the virtual interrupt controller of the target virtual processor without VMM intervention.

Device Interrupts. Acceleration of the delivery of virtual interrupts from I/O devices to virtual processors is not addressed directly by AVIC hardware. This acceleration would be provided by an I/O memory management unit (IOMMU). The AVIC architecture is compatible with the AMD I/O Memory Management Unit (IOMMU). For more information on the IOMMU architecture, see *AMD I/O Virtualization Technology (IOMMU) Specification* (order #48882). See “Device Interrupts” on page 602 for further details of device interrupt handling under the AVIC extension.

The following subsections describe the AVIC architecture in detail.

15.29.2 Local APIC Register Virtualization

The system programming interface for the local APIC comprises a set of memory-mapped registers. In a non-virtualized environment, system software directly reads and writes these registers to configure the interrupt controller and initiate and process interrupts. In a virtualized environment, each guest operating system still requires access to this system programming interface but does not own the underlying interrupt processing hardware. To provide this facility to the guest operating system, VMM-level software emulates the local APIC for each guest virtual processor.

The AVIC architecture provides an image of the local APIC called the guest virtual APIC (guest vAPIC) in the guest physical address (GPA) space of each virtual processor when the virtual machine for the guest is instantiated. This image is backed by a page in the system physical address (SPA) space called a vAPIC backing page. The backing page remains pinned in system memory as long as the virtual machine persists, even when the specific virtual processor associated with the backing page is not running. Accesses to the memory-mapped register set by the guest are redirected by AVIC hardware to this backing page.

The VMM reads configuration, control, and command information written by the guest from the backing page and writes status information to this page for the guest to read. The guest is allowed to read most registers directly without the need for VMM intervention. Most writes are intercepted allowing the VMM to process and act on the configuration, control, and command data from the guest. However, for certain frequently used command and control operations, specific hardware support allows the guest to directly initiate interrupts and complete end of interrupt processing, eliminating the need for VMM intervention in the execution of performance-critical operations.

[AMD Public Use]

15.29.3 AVIC Backing Page

AVIC hardware detects attempted accesses by the guest to its local APIC register set and redirects these accesses to the vAPIC backing page. This is illustrated in the figure below.

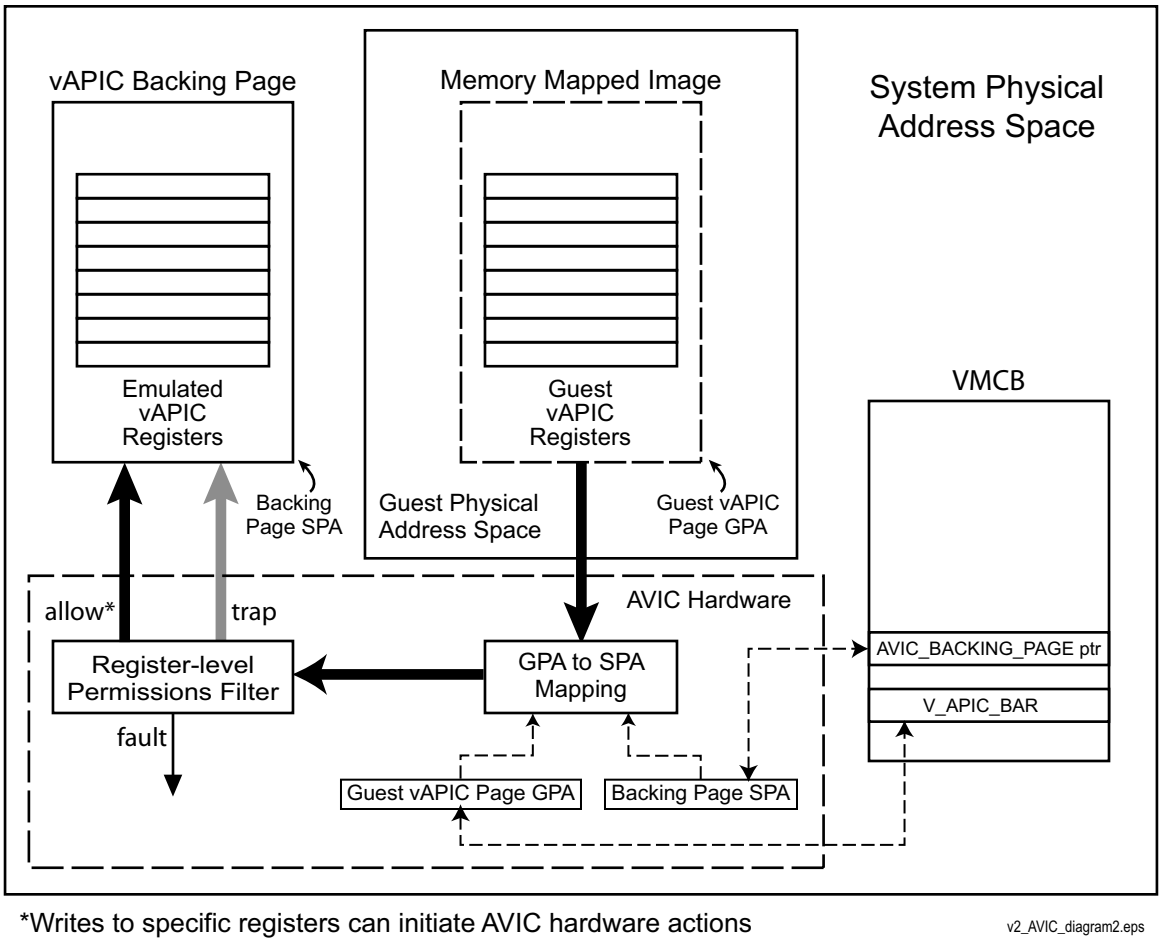


Figure 15-15. vAPIC Backing Page Access

To correctly redirect guest accesses of the guest vAPIC registers to the vAPIC backing page, the hardware needs two addresses. These are:

- vAPIC backing page address in the SPA space
- Guest vAPIC base address (APIC BAR) in the GPA space

System software is responsible for setting up a translation in the nested page table granting guest read and write permissions for accesses to the vAPIC Backing Page in SPA space. AVIC hardware walks the nested page table to check permissions, but does not use the SPA address specified in the leaf page table entry. Instead, AVIC hardware finds this address in the AVIC_BACKING_PAGE pointer field of the VMCB.

[AMD Public Use]

The VMM initializes the backing page with appropriate default APIC register values including items such as APIC version number. The vAPIC backing page address and the guest vAPIC base address are stored in the VMCB fields AVIC_BACKING_PAGE pointer and V_APIC_BAR respectively.

System firmware initializes the value of guest vAPIC base address (and VMCB.V_APIC_BAR) to FEE0_0000h. This is the address where the guest operating system expects to find the local APIC register set when it boots. If the guest attempts to relocate the local APIC register base address in GPA space by writing to the APIC Base Address Register (MSR 0000_001Bh), the VMM should intercept the write to update the V_APIC_BAR field of the guest's VMCB(s) and the GPA part of translation in the virtual machine's nested page tables.

The vAPIC backing page must be present in system physical memory for the life of the guest VM because some fields are updated even when the guest is not running.

15.29.3.1 Virtual APIC Register Accesses

AVIC hardware detects attempted guest accesses to the vAPIC registers in the backing page. These attempted accesses are handled by the register-level permissions filter in one of three ways:

- Allow—The access to the backing page is allowed to complete. Writes update the backing page value, while reads return the current value. In certain cases, a write results in specific hardware-based acceleration actions (summarized in Table 15-22 and described below).
- Fault—The processor performs an SVM intercept before the access. Causes a #VMEXIT.
- Trap— The processor performs an SVM intercept immediately after the access completes. Causes a #VMEXIT.

The details of this behavior for each of these registers are summarized in the following table.

Table 15-22. Guest vAPIC Register Access Behavior

xAPIC Register Offset	x2APIC MSR Address	Register Name	xAVIC and x2AVIC Register Access Behavior
20h	802h	APIC ID Register	Read: Allowed Write: #VMEXIT (trap)
30h	803h	APIC Version Register	Read: Allowed Write: #VMEXIT (fault)
80h	808h	Task Priority Register (TPR)	Read: Allowed Write: Accelerated by AVIC
90h	809h	Arbitration Priority Register (APR)	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
A0h	80Ah	Processor Priority Register (PPR)	Read: Allowed Write: #VMEXIT (fault)
B0h	80Bh	End of Interrupt Register (EOI)	Read: Allowed Write: Accelerated by AVIC for edge-triggered interrupts or #VMEXIT (trap) for level triggered interrupts

[AMD Public Use]

Table 15-22. Guest vAPIC Register Access Behavior (continued)

xAPIC Register Offset	x2APIC MSR Address	Register Name	xAVIC and x2AVIC Register Access Behavior
C0h	-	Remote Read Register	Read: Allowed Write: #VMEXIT (trap)
D0h	80Dh	Logical Destination Register	Read: Allowed Write: #VMEXIT (trap)
E0h	-	Destination Format Register	Read: Allowed Write: #VMEXIT (trap)
F0h	80Fh	Spurious Interrupt Vector Register	Read: Allowed Write: #VMEXIT (trap)
100h–170h	810h-817h	In-Service Register (ISR)	Read: Allowed Write: #VMEXIT (fault)
180h–1F0h	818h-81Fh	Trigger Mode Register (TMR)	Read: Allowed Write: #VMEXIT (fault)
200h–270h	820h-827h	Interrupt Request Register (IRR)	Read: Allowed Write: #VMEXIT (fault)
280h	828h	Error Status Register (ESR)	Read: Allowed Write: #VMEXIT (trap)
300h	830h	Interrupt Command Register Low (ICRL)	Read: Allowed Write: Accelerated by AVIC or #VMEXIT (trap) for advanced functions.
310h	-	Interrupt Command Register High (ICRH)	Read: Allowed (xAVIC) Write: Allowed (xAVIC)
320h	832h	Timer Local Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
330h	833h	Thermal Local Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
340h	834h	Performance Counter Local Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
350h	835h	Local Interrupt 0 Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
360h	836h	Local Interrupt 1 Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
370h	837h	Error Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
380h	838h	Timer Initial Count Register	Read: Allowed Write: #VMEXIT (trap)
390h	839h	Timer Current Count Register	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
3E0h	83Eh	Timer Divide Configuration Register	Read: Allowed Write: #VMEXIT (trap)
-	83Fh	Self IPI Register (x2APIC only)	Write: Allowed (x2AVIC)

[AMD Public Use]

Table 15-22. Guest vAPIC Register Access Behavior (continued)

xAPIC Register Offset	x2APIC MSR Address	Register Name	xAVIC and x2AVIC Register Access Behavior
400h	840h	Extended APIC Feature Register	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
410h	841h	Extended APIC Control Register	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
420h	842h	Specific End of Interrupt Register (SEOI)	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
480h–4F0h	848h-84Fh	Interrupt Enable Registers (IER)	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
500h-570h	850h-857h	Extended Interrupt [7:0] Local Vector Table Registers	CPUID Fn8000_000A_EDX[27]=1: Read: Allowed Write: #VMEXIT (trap) CPUID Fn8000_000A_EDX[27]=0: Read: #VMEXIT (fault) Write: #VMEXIT(fault)
580h-FFFh	-	Reserved	Read: #VMEXIT (fault) Write: #VMEXIT (fault)

Accesses to any other register locations not explicitly defined in this table are allowed to read and write the backing page.

All vAPIC registers are 32-bits wide and are located at 16-byte aligned offsets. The results of an attempted read or write of any bytes in the range [register_offset + 4:register_offset + 15] are undefined.

Guest writes to the Task Priority Register (TPR) and specific usage cases of writes to the End of Interrupt (EOI) Register and the Interrupt Command Register Low (ICRL) cause specific hardware actions. AVIC hardware allows guest writes to the Interrupt Command Register High (ICRH) since the writing of this register has no immediate hardware side-effect. AVIC hardware maintains and uses the value in the Processor Priority Register (PPR) to control the delivery of interrupts to guest virtual processors. The following sections discuss the handling of accesses by the guest to these registers in the vAPIC backing page.

Task Priority Register (TPR). When the guest operating system writes to the TPR, the value is updated in the backing page and the upper 4 bits of the value are automatically copied by the hardware to the V_TPR value in the VMCB. All reads from the TPR location return the value from the vAPIC backing page. Also, any TPR accesses using the MOV CR8 semantics update the backing page and V_TPR values.

The priority value stored in CR8 and V_TPR are not the same format as the APIC TPR register. Only the Task Priority bits of are maintained in the lower 4 bits of CR8 and V_TPR. The Task Priority Sub-

[AMD Public Use]

class value is not stored. Writes to the memory-mapped TPR register update bits 3:0 of CR8 and V_TPR and writes to CR8 update the TPR backing page value bits 7:4 while bits 3:0 are set to zero.

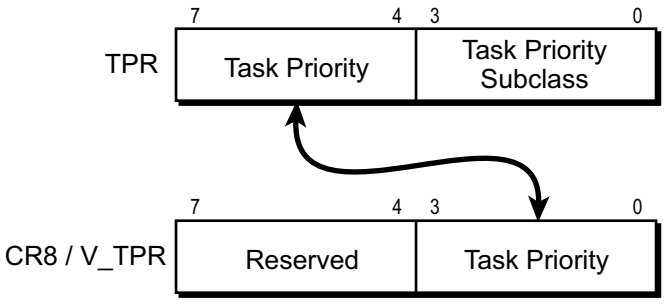


Figure 15-16. Virtual APIC Task Priority Register Synchronization

The synchronization between the Task Priority field of the TPR and the Task Priority field of CR8 is normal local APIC behavior which is emulated by AVIC. For more information on the APIC, see Chapter 16, “Advanced Programmable Interrupt Controller (APIC),” on page 659.

Processor Priority Register (PPR). Writes to the processor priority register by the guest cause a #VMEXIT without updating the value in the backing page. AVIC hardware maintains the PPR value in the backing page. AVIC hardware updates the PPR value in the backing page when either the TPR value or the highest in-service interrupt changes. This value is used to control the delivery of virtual interrupts to the guest. PPR reads by the guest are allowed.

End of Interrupt (EOI) Register. When the guest writes to the EOI register address, AVIC hardware clears the highest priority in-service interrupt (ISR) bit in the backing page and re-evaluates the interrupt state to determine if another pending interrupt should be delivered. If the highest priority in-service interrupt is set to level mode (in the corresponding TMR bit), the EOI write causes a #VMEXIT to allow the VMM to emulate the level-triggered behavior.

Interrupt Command Register Low (ICRL). Writes to the ICRL register have the side-effect of initiating the generation of an interprocessor interrupt (IPI) based on the values written to the fields in both the ICRL and ICRH registers. AVIC hardware handles the generation of IPIs when the specified Message Type is Fixed (also known as fixed delivery mode) and the Trigger Mode is edge-triggered. The hardware also supports self and broadcast delivery modes specified via the Destination Shorthand (DSH) field of the ICRL. Logical and physical APIC ID formats are supported. All other IPI types cause a #VMEXIT. For more information on AVIC’s handling of IPI commands, see “Inter-processor Interrupts” on page 589.

15.29.4 VMCB Changes for AVIC

The following paragraphs provide an overview of new VMCB fields defined as part of the AVIC architecture.

[AMD Public Use]

15.29.4.1 VMCB Virtual Interrupt Control Word

AVIC adds the AVIC Enable bit to the VMCB virtual interrupt control word at offset 60h. x2AVIC mode, described in Section 15.29.10 on page 607, adds x2AVIC Mode Enable.

AVIC Enable—Virtual Interrupt Control, Bit 31. The AVIC hardware support may be enabled on a per virtual processor basis. This bit determines whether or not AVIC is enabled for a particular virtual processor. Any guest configured to use AVIC must also enable nested paging. Enabling AVIC implicitly disables the V_IRQ, V_INTR_PRIO, V_IGN_TPR, and V_INTR_VECTOR fields in the VMCB Control Word. Enabling AVIC also affects CR8 behavior independent of V_INTR_MASKING enable (bit 24): writes to CR8 affect the V_TPR and update the backing page and reads from CR8 return V_TPR.

x2AVIC Mode Enable—Virtual Interrupt Control, Bit 30. The x2APIC MSR interface virtualization enable. When this bit is set to 1 on a VMRUN, AVIC Enable also has to be set to 1. If not the VMRUN fails with a VMEXIT_INVALID error code.

When AVIC is enabled (bit 31 set to 1), x2AVIC Mode Enable (bit 30) determines AVIC mode. If the x2AVIC bit is cleared to 0, xAVIC virtualization mode is enabled (used for MMIO local APIC register interface). If the x2AVIC bit is set to 1, x2AVIC virtualization mode is enabled (used for MSR local APIC register interface).

15.29.4.2 AVIC VMCB Fields

AVIC utilizes a number of formerly reserved locations in the VMCB.

V_APIC_BAR—VMCB, Offset 098h. This entry is used to hold a copy of guest physical base address of its local APIC register block. The guest can change the GPA of its local APIC register block by writing to the guest version of the APIC Base Address Register (MSR 0000_001Bh). Writes to this MSR are intercepted by the VMM and the value is used to update the GPA in the nested page table entry for the vAPIC backing page and the value to be saved in this field of the VMCB.

APIC_BACKING_Page Pointer—VMCB, Offset 0E0h. This is a 52-bit HPA pointer to the vAPIC backing page for this virtual processor. The vAPIC backing page is described in more detail in the following section.

Logical APIC Table Pointer—VMCB, Offset 0F0h. This is a 52-bit HPA pointer to the Logical APIC ID Table for the virtual machine containing this virtual processor. This table is described in more detail in the following section.

Physical APIC Table Pointer—VMCB, Offset 0F8h. This is a 52-bit HPA pointer to the Physical APIC ID Table for the virtual machine containing this virtual processor. This table is described in more detail in the following section.

AVIC_PHYSICAL_MAX_INDEX—VMCB, Offset 0F8h. Bits 11:0. This value provides the index of the last guest physical core ID for this guest.

[AMD Public Use]

15.29.4.3 Physical Address Pointer Restrictions

All of the physical addresses in the previous sections must point to legal, implementation-supported physical address ranges. These pointers are evaluated on VMRUN and cause a #VMEXIT if they are outside of the legal range. These memory ranges must be mapped as write-back cacheable memory type.

All the addresses point to 4-Kbyte aligned data structures. Bits 11:0 are reserved (except for offset 0F8h) and should be set to zero. The lower 12 bits of offset 0F8h are used for the AVIC_PHYSICAL_MAX_INDEX field. VMRUN fails with #VMEXIT(VMEXIT_INVALID) if AVIC_PHYSICAL_MAX_INDEX is greater than 255 in xAVIC mode or greater than 511 in x2AVIC mode when CPUID Fn8000_000A_ECX[x2AVIC_EXT] (bit 6)=0.

Multiprocessor VM requirements. When running a VM which has multiple virtual CPUs with xAVIC mode enabled, and the VMM runs a virtual CPU on a core which had last run a different virtual CPU from the same VM, regardless of the respective ASID values, care must be taken to flush the TLB on the VMRUN using a TLB_CONTROL value of 3h. Failure to do so may result in stale mappings misdirecting virtual APIC accesses to the previous virtual CPU's APIC backing page.

15.29.5 AVIC Memory Data Structures

The AVIC architecture defines three new memory-resident data structures. Each of these structures is defined to fit exactly in one 4-Kbyte page. Future implementations may expand the size.

15.29.5.1 Virtual APIC Backing Page

Each virtual processor in the system is assigned a virtual APIC backing page (vAPIC backing page). Accesses by the guest to the local APIC register block in the guest physical address space are redirected to the vAPIC backing page in system memory. The vAPIC backing page is used by AVIC hardware and the VMM to emulate the local APIC. See “Virtual APIC Register Accesses” on page 591 for a detailed description.

15.29.5.2 Physical APIC ID Table

The physical APIC ID table is set up and maintained by the VMM and is used by the hardware to locate the proper vAPIC backing page to be used to deliver interrupts based on the guest physical APIC ID. One physical APIC ID table must be provided per virtual machine.

The guest physical APIC ID is used as an index into this table. Each entry contains a pointer to the virtual processor’s vAPIC backing page, a bit to indicate whether the virtual processor is currently scheduled on a physical core, and if so, the physical APIC ID of that core.

If CPUID Fn8000_000A_ECX[x2AVIC_EXT] (bit 6) = 0, the length of this table is fixed at 4 Kbytes allowing a maximum of 512 virtual processors per virtual machine. If CPUID Fn8000_000A_ECX[x2AVIC_EXT] (bit 6) = 1, the length of this table is up to eight consecutive 4-Kbyte pages, where the number of pages is equal to AVIC_PHYSICAL_MAX_INDEX[11:9] plus 1. The physical ID table can be populated in a sparse manner using the valid bit to indicate assigned IDs. The index of the last valid entry is stored in the VMCB AVIC_PHYSICAL_MAX_INDEX field.

[AMD Public Use]

A pointer to this table is maintained in the VMCB. Because there is a single Physical APIC ID Table per virtual machine, the value of this pointer is the same for every virtual processor within the virtual machine.

Each entry in the table has the following format:

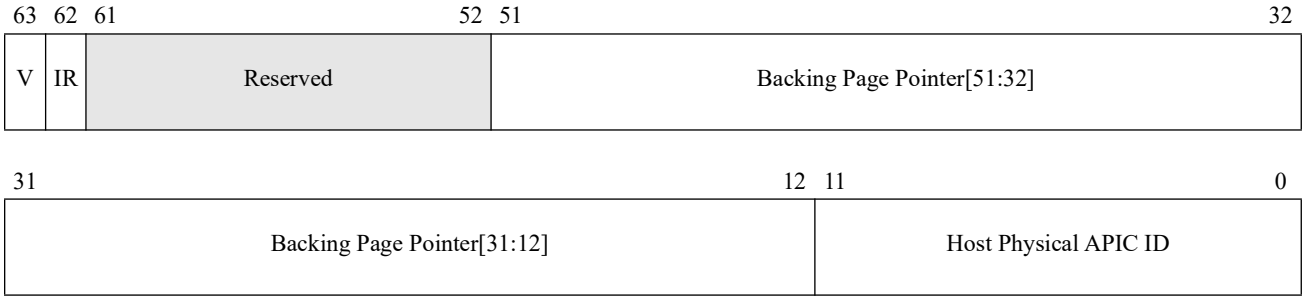


Figure 15-17. Physical APIC ID Table Entry

Table 15-23. Physical APIC ID Table Entry Fields

Bit(s)	Field Name	Description
63	V	Valid bit. When set, indicates that this entry contains a valid vAPIC backing page pointer. If cleared, this table entry contains no information.
62	IR	IsRunning. This bit indicates that the corresponding guest virtual processor is currently scheduled by the VMM to run on a physical core.
61:52	—	Reserved, SBZ. Should always be set to zero.
51:12	Backing Page Pointer	4-Kbyte aligned HPA of the vAPIC backing page for this virtual processor.
11:0	Host Physical APIC ID	Physical APIC ID of the physical core allocated by the VMM to host the guest virtual processor. This field is not valid unless the IsRunning bit is set.

Note that the IR bit, when set, indicates that the VMM has assigned a physical core to host this virtual processor. The bit does not differentiate between a physical processor running in guest mode (actively executing guest software) or in host mode (having suspended the execution of guest software).

[AMD Public Use]

In xAVIC mode, the Physical APIC ID table occupies the lower half of a single 4-Kbyte memory page, formatted as follows:

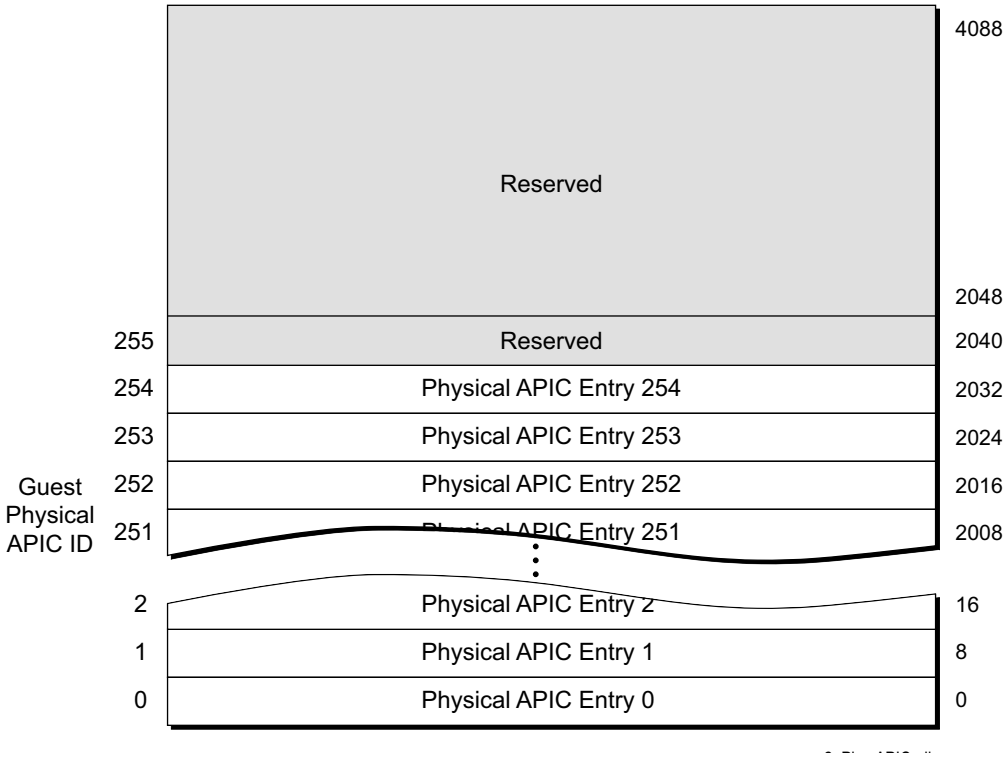


Figure 15-18. Physical APIC Table in Memory

Since a destination of FFh is used to specify a broadcast, physical APIC ID FFh is reserved. The upper 2048 bytes of the table are reserved and should be set to zero.

15.29.5.3 Logical APIC ID Table

In addition to the Physical APIC ID Table, each guest VM is assigned a Logical APIC ID Table. This table is used to lookup the guest physical APIC ID for logically addressed interrupt requests. Each entry of this table provides the guest physical APIC ID corresponding to a single logically addressed APIC. Note that this implies that the logical ID of each vAPIC must be unique. The entries of this table are selected using the logical ID and interpreted differently depending upon logical APIC addressing mode of the guest.

If the guest attempts to change the logical ID of its APIC, the VMM must reflect this change in the Logical APIC ID Table. AVIC hardware supports the fixed interrupt message type targeting one or more logical destinations. The hardware also supports self and broadcast delivery modes specified via the Destination Shorthand (DSH) field of the ICRL. Any other message types must be supported through emulation by the VMM.

[AMD Public Use]

A pointer to this table is maintained in the VMCB. Because there is a single Logical APIC ID Table per virtual machine, the value of this pointer is the same for every virtual processor within the virtual machine.

For all logical destination modes, the table entries have the following format:

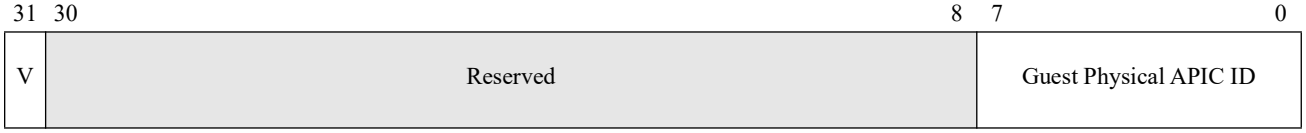


Figure 15-19. Logical APIC ID Table Entry

Table 15-24. Logical APIC ID Table Entry Fields

Bit(s)	Field Name	Description
31	V	Valid Bit. When set, indicates that this table entry contains a valid physical APIC ID. If cleared, this table entry contains no information.
30:8	—	Reserved, SBZ. Should always be set to zero.
7:0	Guest Physical APIC ID	Guest physical APIC ID corresponding to the local APIC selected when logically addressed.

Logical APIC ID Table Format for Flat Mode. When running in flat mode, AVIC expects the logical APIC ID table to be formatted as shown in Figure 15-20 below. This mode uses only the first 8 entries of the table. Although the logical APIC ID is an eight bit value, supported encodings must be of the form 2^i, where i = 0 to 7. In the figure the value i is used and represents the index into the table. The actual byte offset into the table for a given logical APIC ID *l_apic_id* is 4 * $\log_2$(l_apic_id).

The Logical APIC ID Table is not used by AVIC hardware in x2AVIC mode. Instead, the destination logical ID is derived from the target x2APIC ID as follows: Logical x2APICID = (X2APICID[19:4] << 16) | (1 << x2APICID[3:0]).

[AMD Public Use]

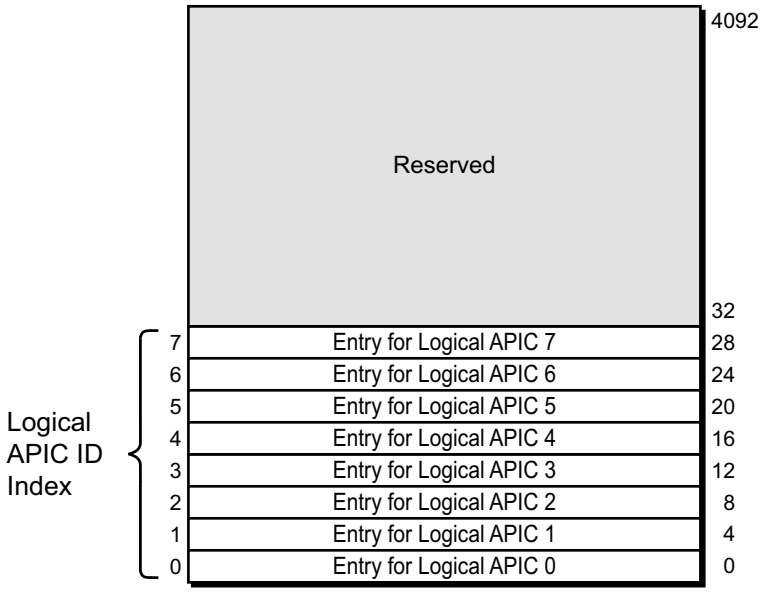


Figure 15-20. Logical APIC ID Table Format, Flat Mode

Logical APIC ID Table Format for Cluster Mode. In cluster mode, bits 7:4 of the logical APIC ID represent the cluster number and bits 3:0 represent the APIC index (bit encoded). The cluster number Fh (15) is reserved. Since the APIC index field is four bits, four encodings are supported for the APIC index value.

The actual byte offset into the table for a given cluster *c* and an APIC index *apic_ix* is (16 * c) + 4 * $\log_2$(apic_ix).

When running in cluster mode, AVIC expects the logical APIC ID table to be formatted as shown in Figure 15-21 below.

[AMD Public Use]

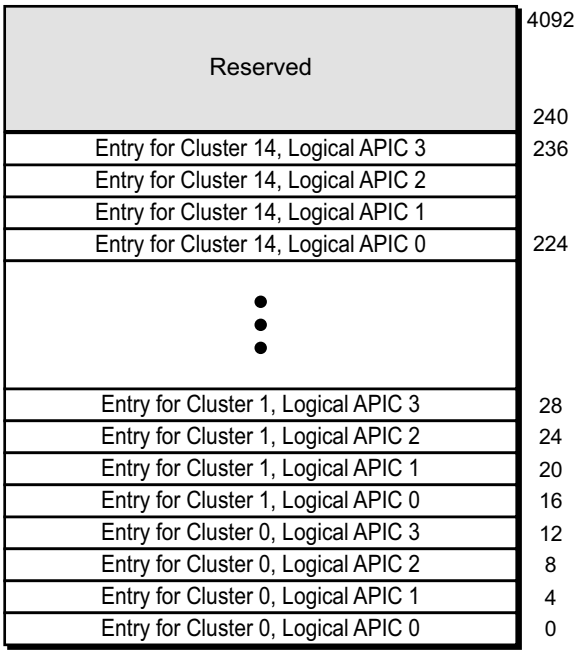


Figure 15-21. Logical APIC ID Table Format, Cluster Mode

15.29.6 Interrupt Delivery

There are two fundamental types of virtual interrupts—interprocessor interrupts (IPIs) and I/O device interrupts (device interrupts). An IPI is initiated when guest system software writes the ICRL register. A device interrupt is initiated by a I/O device that has been programmed by guest system software (usually a device driver) to send a message signaling an event to a specific guest physical processor. This message usually includes an interrupt vector number indicating the nature of the event.

The following sections discuss the actions taken by AVIC hardware when a virtual processor signals an IPI and the actions taken by I/O virtualization hardware when a device signals a virtual interrupt.

15.29.6.1 Interprocessor Interrupts

To process an IPI, AVIC hardware executes the following steps:

1. If the destination-shorthand coded in the command is 01b (i.e. self), update the IRR in the backing page, signal doorbell to self and skip remaining steps.
2. If destination-shorthand is non-zero, or if the destination field is FFh (i.e. broadcast), jump to step 4.
3. If the destination(s) is (are) logically addressed, lookup the guest physical APIC IDs for each logical ID using the Logical APIC ID table.

   If the entry is not valid (V bit is cleared), cause a #VMEXIT.

[AMD Public Use]

If the entry is valid, but the Guest Physical APIC ID is greater than 255, cause a #VMEXIT (xAVIC).

If the entry is valid, but contains an invalid backing page pointer, cause a #VMEXIT.

4. Lookup the vAPIC backing page address in the Physical APIC table using the guest physical APIC ID as an index into the table.

   For directed interrupts, if the selected table entry is not valid, cause a #VMEXIT. For broadcast IPIs, invalid entries are ignored.

5. For every valid destination:
   - Atomically set the appropriate IRR bit in each of the destinations' vAPIC backing page.
   - Check the IsRunning status of each destination.
   - If the destination IsRunning bit is set, send a doorbell message using the host physical core number from the Physical APIC ID table.

6. If any destinations are identified as not currently scheduled on a physical core (i.e., the IsRunning bit for that virtual processor is not set), cause a #VMEXIT.

Refer to Section 15.29.9.1, "AVIC IPI Delivery Not Completed," on page 605 for new exit codes associated with the #VMEXIT exceptions listed above.

15.29.6.2 Device Interrupts

The delivery of a I/O device interrupt to a virtual processor is handled by an IOMMU with virtual interrupt capability. To deliver a virtual interrupt, I/O virtualization hardware executes the following steps:

1. An interrupt message arrives from the I/O device identifying the source device and interrupt vector number.
2. I/O virtualization hardware uses the device ID to determine the guest physical APIC ID of the core that is the target of the device interrupt.
3. I/O virtualization hardware uses the guest physical APID ID to index into the Physical APIC ID Table to find the SPA of the vAPIC backing page. If the I/O virtualization hardware accesses an entry in the Physical APIC ID Table that is not valid (V bit is cleared), the I/O virtualization hardware aborts the virtual interrupt delivery and logs an error.
4. I/O virtualization hardware performs any required vector number translation.
5. I/O virtualization hardware atomically sets the bit in the IRR in the vAPIC backing page that corresponds to the vector.
6. If the virtual processor that is the target of the interrupt is not currently running on its assigned physical core, the virtual interrupt will be presented when the virtual processor is made active again. I/O virtualization hardware may provide additional information to the VMM about the device interrupt to aid in virtual processor scheduling decisions.

   If the virtual processor that is the target of the interrupt is scheduled on a physical processor (indicated by the IsRunning bit of the Physical APIC ID table entry being set), I/O virtualization

[AMD Public Use]

hardware uses the host physical APIC ID in the table entry to send a doorbell signal to the corresponding processor core to signal that an interrupt needs to be processed.

15.29.7 AVIC CPUID Feature Flags

A CPUID feature bit is used indicate support for AVIC on a specific hardware implementation. CPUID Fn8000_000A_EDX[AVIC] (bit 13) = 1 indicates support for the AVIC architecture on that hardware. Additionally, CPUID Fn8000_000A_EDX[x2AVIC] (bit 18) = 1 indicates support for the AVIC architecture when the guest is using x2APIC MSR interface.

See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

15.29.8 New Processor Mechanisms

In order to support the direct injection of interrupts into the guest and to accelerate critical vAPIC functions, new hardware mechanisms are implemented in the processor.

15.29.8.1 Special Trap/Fault Handling for vAPIC Accesses

To virtualize the local APIC utilized by the guest to generate and process interrupts, all read and write accesses by the guest virtual processor to its local APIC registers are redirected to the vAPIC backing page. Most reads and many writes to this guest physical address range read or write the contents of memory locations within the vAPIC backing page at the corresponding offset.

To support proper handling and emulation of the guest local APIC, the processor provides permissions filtering hardware (Refer to Figure 15-15.) that detects and intercepts accesses to specific offsets (representing APIC registers) within the vAPIC backing page. This hardware either allows the access, blocks the access and causes a #VMEXIT (fault behavior), or allows the access and then causes a #VMEXIT (trap behavior).

Hardware directly handles the side effects of guest writes to the TPR and EOI registers. Writes to the ICRL register with simple functional side effects such as the generation of a directed IPI or a self-IPI request are handled directly. Values written to the ICRL defined to initiate more complex behavior cause a #VMEXIT to allow the VMM to emulate the function. A guest write to the ICRH register has no immediate hardware side effect and is allowed.

Most other write access attempts within the vAPIC register address range cause a #VMEXIT with trap or fault behavior allowing the VMM to emulate the function of that register. See Table 15-22 for more detail.

Reads and writes to locations within the vAPIC backing page, but outside the offset range of defined vAPIC registers are allowed to complete.

15.29.8.2 Doorbell Mechanism

Each core provides a doorbell mechanism that is used by other cores (for IPIs) and the IOMMU (for device interrupts) to signal to the VMM of the target physical core that a virtual interrupt requires

[AMD Public Use]

processing. The exact mechanism is implementation-specific, but must be protected from access from non-privileged software running on other cores and from direct access by an external device.

When the doorbell is received in guest mode, hardware on the receiving core evaluates the vAPIC state in the vAPIC backing page for the currently running virtual processor and injects the interrupt into the guest as appropriate.

Doorbell Register. The system programming interface to the doorbell mechanism is provided via an MSR. Sending a doorbell signal to a another core is initiated by writing the physical APIC ID corresponding to that core to the Doorbell Register (MSR C001_011Bh). The format of this register is shown in Figure 15-22 below.

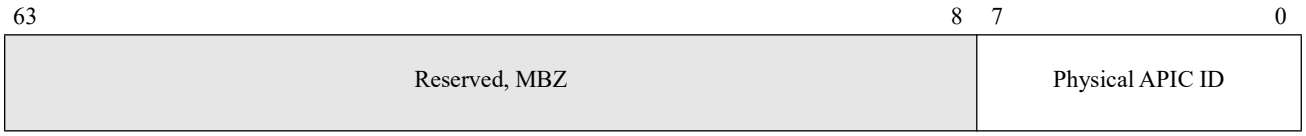


Figure 15-22. Doorbell Register, MSR C001_011Bh

Writing to this register causes a doorbell signal to be sent to the specified physical core. The serializing semantics of WRMSR are relaxed when writing to the Doorbell Register. Any attempt to read from this register results in a #GP.

Processing of Doorbell Signals. A doorbell signal delivered to a running guest is recognized by the hardware regardless of whether it can be immediately injected into the guest as a virtual interrupt. On the next VMRUN, the virtual interrupt delivery mechanism evaluates the state of the IRR register of the guest's vAPIC backing page to find the highest priority pending interrupt and injects it if interrupt masking and priority allow.

15.29.8.3 Additional VMRUN Handling

In addition to the normal VMRUN operations, the core re-evaluates the APIC state in the vAPIC backing page upon entry into the guest and processes pending interrupts as necessary. Specifically:

- On VMRUN the interrupt state is evaluated and the highest priority pending interrupt indicated in the IRR is delivered if interrupt masking and priority allow
- Any doorbell signals received during VMRUN processing are recognized immediately after entering the guest
- When AVIC mode is enabled for a virtual processor, the V_IRQ, V_INTR_PRIO, V_INTR_VECTOR, and V_IGN_TPR fields in the VMCB are ignored.

15.29.9 AVIC Exit Codes

The AVIC architecture defines two new AVIC-related #VMEXIT events. These cases are described in the following sections. Assigned EXITCODE values are given in Table C-1 on page 801.

[AMD Public Use]

15.29.9.1 AVIC IPI Delivery Not Completed

An IPI could not be delivered to all targeted guest virtual processors because at least one guest virtual processor was not allocated to a physical core at the time. This results in a #VMEXIT with an exit code of AVIC_INCOMPLETE_IPI. Additional data associated with this #VMEXIT event is returned in the EXITINFO1 and EXITINFO2 fields.

EXITINFO1. This field contains the values written to the vAPIC ICRH and ICRL registers.

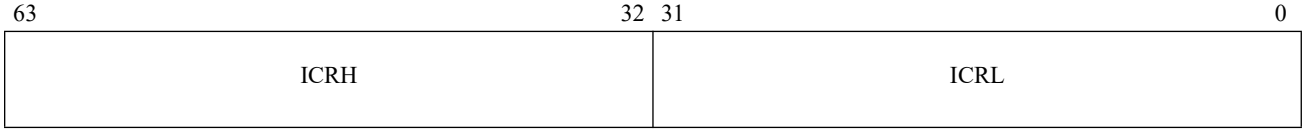


Figure 15-23. EXITINFO1 for AVIC_INCOMPLETE_IPI

Table 15-25. EXITINFO1 Fields for AVIC_INCOMPLETE_IPI

Bit(s)	Field Name	Description
63:32	ICRH	Value written to the vAPIC ICRH register.
31:0	ICRL	Value written to the vAPIC ICRL register.

EXITINFO2. This field contains information describing the specific reason for the IPI delivery failure.

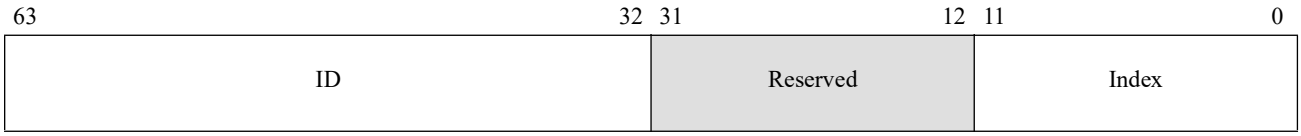


Figure 15-24. EXITINFO2 for AVIC_INCOMPLETE_IPI

Table 15-26. EXITINFO2 Fields for AVIC_INCOMPLETE_IPI

Bit(s)	Field Name	Description
63:32	ID	Specific reason for the delivery failure. See Table 15-27 for defined values.
31:12	—	Reserved
11:0	Index	For ID = 1 – 3, this field provides the index of a logical or physical table entry. Reserved for all other ID values.

The ID field identifies the reason for the IPI delivery failure:

[AMD Public Use]

Table 15-27. ID Field—IPI Delivery Failure Cause

ID	Cause	Description	Index
0	Invalid Interrupt type	The trigger mode for the specified IPI was set to level or the destination type is unsupported.	Reserved.
1	IPI Target Not Running	IsRunning bit of the target for a Singlecast/Broadcast/Multicast IPI is not set in the physical APIC ID table.	Index of the physical or logical APIC ID table entry for the target virtual processor that was not scheduled on a physical core.
2	Invalid IPI Target	Target ID invalid. Target is not covered by the physical or logical ID table.	Index of the physical or logical table entry for the invalid target.
3	Invalid Backing Page Pointer	The vAPIC Backing Page Pointer field of the Physical APIC ID Table contained an invalid physical address.	For shorthand or broadcast delivery modes, index of the physical APIC ID Table containing the invalid address. For directed IPIs, index of the logical or physical APIC ID table depending on the destination mode.
4	Invalid IPI Vector	The vector for the specified IPI was set to an illegal value (VEC < 16).	Reserved
5	Un-accelerated IPI	Destination Shorthand is not set to Self (Secure AVIC).	Reserved
> 5	Reserved	—	Reserved

15.29.9.2 AVIC Access to Un-accelerated vAPIC register

A guest access to an APIC register that is not accelerated by AVIC results in a #VMEXIT with the exit code of AVIC_NOACCEL. This fault is also generated if an EOI is attempted when the highest priority in-service interrupt is set for level-triggered mode. Additional data associated with this #VMEXIT event is returned in the EXITINFO1 and EXITINFO2 fields.

[AMD Public Use]

EXITINFO1. This field contains the offset of the un-accelerated virtual APIC register and a bit indicating whether a read or write operation was attempted.

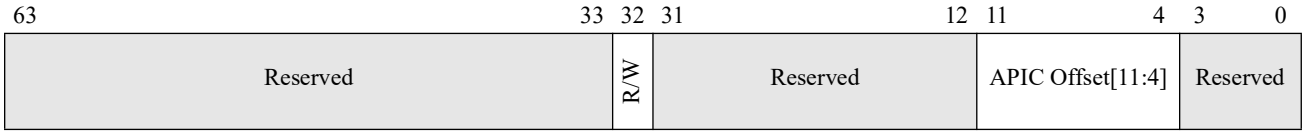


Figure 15-25. EXITINFO1 for AVIC_NOACCEL

Table 15-28. EXITINFO1 Fields for AVIC_NOACCEL

Bits	Field Name	Description
63:33	—	Reserved.
32	R/W	If set, write was attempted. If clear, read was attempted.
31:12	—	Reserved.
11:4	APIC_Offset[11:4]	Offset within virtual vAPIC backing page at which read or write was attempted. APIC_Offset[3:0] = 0, since all registers are aligned on 16-byte boundaries.
3:0	—	Reserved.

EXITINFO2. This field contains extra information for the un-accelerated operation. If the EXITINFO1 fields indicate a write to the vAPIC EOI register (offset = B0h), bits 7:0 of this value contain the number of the highest in-service vector found in the virtual APIC ISR.

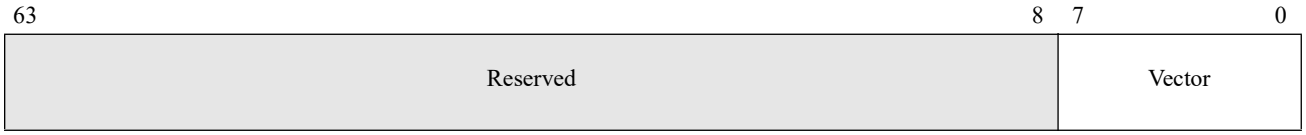


Figure 15-26. EXITINFO2 for AVIC_NOACCEL

Table 15-29. EXITINFO2 Fields for AVIC_NOACCEL

Bit(s)	Field Name	Description
63:8	—	Reserved
7:0	Vector	Vector for attempted EOI; otherwise undefined.

15.29.10 x2AVIC

The x2AVIC virtualization feature provides hardware acceleration for performance-sensitive APIC accesses when the x2APIC MSR interface is used by the guest operating system. x2AVIC support is reported by Fn8000_000A_EDX[X2AVIC] (bit 18) = 1.

[AMD Public Use]

x2AVIC mode is enabled by setting AVIC Enable (bit 31) and x2AVIC Mode Enable (bit 30) in VMCB offset 60h to 1. When x2AVIC mode is enabled, x2APIC MSR accesses are virtualized in a similar manner to MMIO accesses in xAVIC mode. x2APIC MSR intercept checks and access checks have higher priority than AVIC access permission checks. See Section 15.29.3.1 for x2APIC register access behavior when x2AVIC is enabled.

In x2APIC mode Interrupt Command Register Low (ICRL) and Interrupt Command Register High (ICRH) registers are combined into a 64-bit register and accessed through the ICR MSR (830h). In x2AVIC mode ICR MSR bits 31:0 and 63:32 are mapped to ICRL (offset 300h) and ICRH (offset 310h) in the backing page. SELF_IPI MSR (83Fh) acceleration is handled the same way as ICR MSR acceleration.

V_APIC_BAR and Logical Destination Table are not used in x2AVIC mode.

New x2AVIC mode error conditions are documented in Section 15.29.4.1, Section 15.29.4.3 and Table 15-28.

15.30 SVM Related MSRs

SVM uses the following MSRs for various control purposes. These MSRs are available regardless of whether SVM is enabled in EFER.SVME. For details on implementation-specific features, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

15.30.1 VM_CR MSR (C001_0114h)

The VM_CR MSR controls certain global aspects of SVM. The VM_CR MSR layout is shown in Figure 15-27.

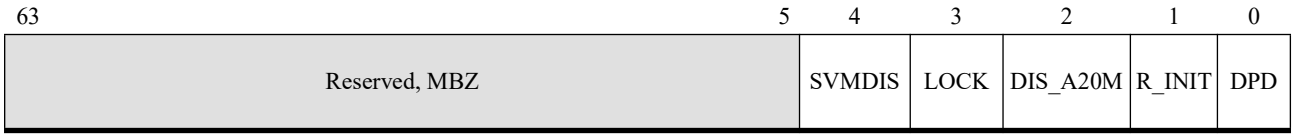


Figure 15-27. VM_CR MSR (C001_0114h)

The individual fields are as follows:

- DPD—Bit 0. If set, disables the external hardware debug port and certain internal debug features.
- R_INIT—Bit 1. If set, non-intercepted INIT signals are converted into an #SX exception.
- DIS_A20M—Bit 2. If set, disables A20 masking.
- LOCK—Bit 3. When this bit is set, writes to LOCK and SVMDIS are silently ignored. When this bit is clear, VM_CR bits 3 and 4 can be written. Once set, LOCK can only be cleared using the SVM_KEY MSR (See Section 15.31.) This bit is not affected by INIT or SKINIT.

[AMD Public Use]

- SVMDIS—Bit 4. When this bit is set, writes to EFER treat the SVME bit as MBZ. When this bit is clear, EFER.SVME can be written normally. This bit does not prevent CPUID from reporting that SVM is available. Setting SVMDIS while EFER.SVME is 1 generates a #GP fault, regardless of the current state of VM_CR.LOCK. This bit is not affected by SKINIT. It is cleared by INIT when LOCK is cleared to 0; otherwise, it is not affected.

15.30.2 IGNNE MSR (C001_0115h)

The read/write IGNNE MSR is used to set the state of the processor-internal IGNNE signal directly. This is only useful if IGNNE emulation has been enabled in the HW_CR MSR (and thus the external signal is being ignored). Bit 0 specifies the current value of IGNNE; all other bits are MBZ.

15.30.3 SMM_CTL MSR (C001_0116h)

The write-only SMM_CTL MSR provides software control over SMM signals. SMM_CTL MSR is not supported when CPUID Fn8000_0021[NoSmmCtlMSR] is set.



Figure 15-28. SMM_CTL MSR (C001_0116h)

Writing individual bits causes the following actions:

- DISMISS—Bit 0. Clear the processor-internal “SMI pending” flag.
- ENTER—Bit 1. Enter SMM: map the SMRAM memory areas, record whether NMI was currently blocked and block further NMI and SMI interrupts.
- SMI_CYCLE—Bit 2. Send SMI special cycle.
- EXIT—Bit 3. Exit SMM: unmap the SMRAM memory areas, restore the previous masking status of NMI and unconditionally reenable SMI.
- RSM_CYCLE—Bit 4. Send RSM special cycle.

Writes to the SMM_CTL MSR cause a #GP if platform firmware has locked the SMM control registers by setting HWCR[SMMLOCK].

Conceptually, the bits are processed in the order of ENTER, SMI_CYCLE, DISMISS, RSM_CYCLE, EXIT, though only the following bit combinations may be set together in a single write (for all other combinations of more than one bit, behavior is undefined):

- ENTER + SMI_CYCLE
- DISMISS + ENTER
- DISMISS + ENTER + SMI_CYCLE
- EXIT + RSM_CYCLE

[AMD Public Use]

The VMM must ensure that ENTER and EXIT operations are properly matched, and *not* nested, otherwise processor behavior is undefined. Also undefined are ENTER when the processor is already in SMM, and EXIT when the processor is not in SMM.

15.30.4 VM_HSAVE_PA MSR (C001_0117h)

The 64-bit read/write VM_HSAVE_PA MSR holds the physical address of a 4KB block of memory where VMRUN saves host state, and from which #VMEXIT reloads host state. The VMM software is expected to set up this register before issuing the first VMRUN instruction.

Writing this MSR causes a #GP if:

- any of the low 12 bits of the address written are nonzero, or
- the address written is greater than or equal to the maximum supported physical address for this implementation.

15.30.5 TSC Ratio MSR (C000_0104h)

Writing to the TSC Ratio MSR allows the hypervisor to control the guest's view of the Time Stamp Counter. The contents of TSC Ratio MSR sets the value of the TSCRatio. This constant scales the timestamp value returned when the TSC is read by a guest via the RDTSC or RDTSCP instructions or when the TSC, MPERF, or MPerfReadOnly MSRs are read via the RDMSR instruction by a guest running under virtualization.

This facility allows the hypervisor to provide a consistent TSC, MPERF, and MPerfReadOnly rate for a guest process when moving that process between cores that have a differing P0 rate. The TSCRatio does not affect the value read from the TSC, MPERF, and MPerfReadOnly MSRs when in host mode or when virtualization is disabled. System Management Mode (SMM) code sees unscaled TSC, MPERF and MPerfReadOnly values unless the SMM code is executed within a guest container. The TSCRatio value does not affect the rate of the underlying TSC, MPERF, and MPerfReadOnly counters, nor the value that gets written to the TSC, MPERF, and MPerfReadOnly MSRs counters on a write by either the host or the guest.

The TSC Ratio MSR specifies the TSCRatio value as a fixed-point binary number in 8.32 format, which is composed of 8 bits of integer and 32 bits of fraction. This number is the ratio of the desired P0 frequency to be presented to the guest relative to the P0 frequency of the core (See Section 17.1, “P-State Control,” on page 696). The reset value of the TSCRatio is 1.0, which sets the guest P0 frequency to match the core P0 frequency.

Note that:

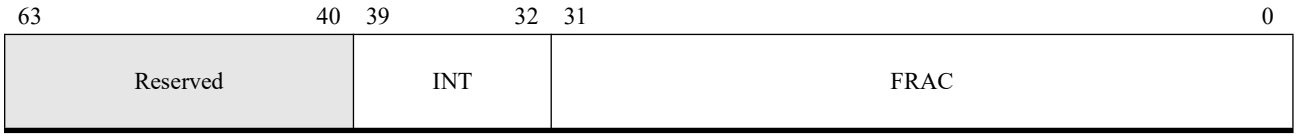
TSCFreq = Core P0 frequency * TSCRatio, so TSCRatio = (Desired TSCFreq) / Core P0 frequency.

The TSC value read by the guest is computed using the TSC Ratio MSR along with the TSC_OFFSET field from the VMCB so that the actual value returned is:

TSC Value (in guest) = (P0 frequency * TSCRatio * t) + VMCB.TSC_OFFSET + (Last Value Written to TSC) * TSCRatio
Where **t** is time since the TSC was last written via the TSC MSR (or since reset if not written)

[AMD Public Use]

The TSC Ratio MSR layout is illustrated in the figure below.



Bits	Mnemonic	Description	Access Type
63:40	Reserved		MBZ
39:32	INT	Integer Part	R/W
31:0	FRAC	Fractional Part	R/W

Figure 15-29. TSC Ratio MSR (C000_0104h)

INT. Integer Part. Bits 39:32. Integer part of TSCRatio.

FRAC. Fractional Part. Bits 39:32. Fractional part of TSCRatio.

$TSCRatio = INT + FRAC \times 2^{-32}$

CPUID Fn8000_000A_EDX[TscRateMsr] =1 indicates support for the TSC Ratio MSR. See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

15.31 SVM-Lock

The SVM-Lock feature allows software to prevent EFER.SVME from being set, either unconditionally or with a 64-bit key to re-enable SVM functionality.

Support for SVM-Lock is indicated by CPUID Fn8000_000A_EDX[SVML] = 1. On processors that support the SVM-Lock feature, SKINIT and STGI can be executed even if EFER.SVME=0. See descriptions of LOCK and SVMDIS bits in Section 15.30.1. When the SVM-Lock feature is not available, hypervisors can use the read-only VM_CR.SVMDIS bit to detect SVM (see Section 15.4).

15.31.1 SVM_KEY MSR (C001_0118h)

The write-only SVM_KEY MSR is used to create a password-protected mechanism to clear VM_CR.LOCK.

When VM_CR.LOCK is zero, writes to SVM_KEY MSR set the 64-bit SVM Key value.

When VM_CR.LOCK is one, writes to SVM_KEY MSR compare the written value to the SVM Key value; if the values match and are non-zero, the VM_CR.LOCK bit is cleared. If the values mismatch or the SVM Key value is zero, the write to SVM_KEY is ignored, and VM_CR.LOCK is unmodified. Software should read VM_CR.LOCK after writing SVM_KEY to determine whether the unlock succeeded.

[AMD Public Use]

If SVM Key is zero when VM_CR.LOCK is one, VM_CR.LOCK can only be cleared by a processor reset.

To preserve the security of the SVM key, reading the SVM_KEY MSR always returns zero.

15.32 SMM-Lock

The SMM-Lock feature allows platform firmware to prevent System Management Interrupts (SMI) from being intercepted in SVM. The SmmLock bit is located in the HWCR MSR register.

15.32.1 SmmLock Bit — HWCR[0]

The SmmLock bit (bit 0) is located in the HWCR MSR (C001_0015h). When SmmLock is clear, it can be set to one. Once set, the bit cannot be cleared by software and writes to it are ignored. SmmLock can only be cleared using the SMM_KEY MSR (see Section 15.32.2), or by a processor reset. This bit is not affected by INIT or SKINIT. When SmmLock is set, other SMM configuration registers cannot be written. For complete information on the HWCR register, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

15.32.2 SMM_KEY MSR (C001_0119h)

The write-only SMM_KEY MSR is used to create a password-protected mechanism to clear SmmLock.

When SmmLock is zero, writes to SMM_KEY MSR set the 64-bit SMM Key value.

When SmmLock is one, writes to SMM_KEY MSR compare the written value to the SMM Key value; if the values match and are non-zero, the SmmLock bit is cleared. If the values mismatch or the SMM Key value is zero, the write to SMM_KEY is ignored, and SmmLock is unmodified. Software should read SmmLock after writing SMM_KEY to determine whether the unlock succeeded.

If SMM_KEY MSR is equal to zero when SmmLock is one, SmmLock can only be cleared by a processor reset.

To preserve the security of the SMM key, reading SMM_KEY MSR always returns zero.

15.33 Nested Virtualization

Hardware support for improved performance of nested virtualization, which is the act of running a hypervisor as a guest under a higher-level hypervisor, is provided through the features described here. These relieve the top-level hypervisor from performing certain common, high-overhead operations that can occur with nested virtualization.

[AMD Public Use]

15.33.1 VMSAVE and VMLOAD Virtualization

VMSAVE and VMLOAD virtualization feature allows these instructions to execute in guest mode, where the guest specifies the VMCB address in RAX as a guest physical address. The VMCB address is translated to a host physical address, which is used to save or load the register state. VMSAVE and VMLOAD virtualization support is indicated by CPUID Fn8000_000A_EDX[15]=1.

VMSAVE and VMLOAD virtualization is enabled by setting VMCB offset 0B8h, bit 1 to 1. When VMSAVE and VMLOAD virtualization is enabled and the corresponding instruction intercept bits are not set, the following checks are performed when VMSAVE or VMLOAD are executed by the guest:

- Nested paging must be enabled
- The hypervisor must be in 64-bit mode
- Secure Encrypted Virtualization must be disabled

If any of the above checks fail, VMEXIT_VMSAVE or VMEXIT_VMLOAD is generated. If the VMSAVE or VMLOAD intercept bits are set, the corresponding instructions are intercepted regardless of VMSAVE and VMLOAD virtualization enablement.

The VMCB address is translated to a host physical address during virtualized VMSAVE and VMLOAD execution. If a page fault occurs during the translation, a #VMEXIT(NPF) is generated.

15.33.2 Virtual GIF

Virtual GIF feature allows STGI and CLGI instructions to execute in guest mode and control virtual interrupts. Virtual GIF support is indicated by CPUID Fn8000_000A_EDX[16]=1.

Virtual GIF is enabled by setting VMCB offset 60h, bit 25 (VGIF_EN). Virtual GIF (VGIF) is loaded on VMRUN from VMCB offset 60h, bit 9, and saved on #VMEXIT. Virtual interrupts are masked when VGIF is 0 and allowed to be taken if VGIF is 1. When Virtual GIF is enabled and not intercepted STGI and CLGI instructions are executed by the guest, VGIF is set or cleared, respectively.

If the STGI or CLGI intercept bits are set, the corresponding instructions are intercepted regardless of Virtual GIF enablement.

15.34 Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is available when the CPU is running in guest mode utilizing AMD-V virtualization features. SEV enables running encrypted virtual machines (VMs) in which the code and data of the virtual machine are secured so that the decrypted version is available only within the VM itself. Each virtual machine may be associated with a unique encryption key so if data is accessed by a different entity using a different key, the SEV encrypted VM's data will be decrypted with an incorrect key, leading to unintelligible data.

It is important to note that SEV mode therefore *represents a departure from the standard x86 virtualization security model,* as the hypervisor is no longer able to inspect or alter all guest code or data. The guest page tables, managed by the guest, may mark data memory pages as either private or

[AMD Public Use]

shared, thus allowing selected pages to be shared outside the guest. Private memory is encrypted using a guest-specific key, while shared memory is accessible to the hypervisor.

15.34.1 Determining Support for SEV

Support for memory encryption features is reported in CPUID 8000_001F[EAX] as described in Section 7.10.1, “Determining Support for Secure Memory Encryption,” on page 240. Bit 1 indicates support for Secure Encrypted Virtualization.

When memory encryption features are present, CPUID 8000_001F[EBX] and 8000_001F[ECX] supply additional information regarding the use of memory encryption, such as the number of keys supported simultaneously and which page table bit is used to mark pages as encrypted. Additionally, in some implementations, the physical address size of the processor may be reduced when memory encryption features are enabled, for example from 48 to 43 bits. In this example, physical address bits 47:43 would be treated as reserved except where otherwise indicated. When memory encryption is supported in an implementation, CPUID 8000_001F[EBX] reports any physical address size reduction present. Bits reserved in this mode are treated the same as other page table reserved bits, and will generate a page fault if found to be non-zero when used for address translation.

Full CPUID details for memory encryption features may be found in Volume 3, section E.4.17.

15.34.2 Key Management

Under the memory encryption extensions defined here, each SEV-enabled guest virtual machine is associated with a memory encryption key, and the SME mode (if used, see Section 7.10 on page 240) is associated with a separate key. Key management for the SEV feature is not handled by the CPU but rather by a separate processor known as the AMD Secure Processor (AMD-SP) which is present on AMD SOCs. A detailed discussion of AMD-SP operation is beyond the scope of this manual.

CPU software is not aware of the values of these keys but the hypervisor should coordinate the loading of virtual machine keys through the AMD-SP driver. This coordination will also determine which ASID the hypervisor should use for a particular guest. Under SEV, the ASID is used as the key index that identifies which encryption key is used to encrypt/decrypt memory traffic associated with that SEV-enabled guest. Encryption keys themselves are never visible to CPU software and are never stored off-chip in the clear.

15.34.3 Enabling SEV

Prior to starting an encrypted VM, software must set MemEncryptionModEn to 1 in the SYSCFG MSR as described in Section 7.10.2, “Enabling Memory Encryption Extensions,” on page 240. SEV may then be enabled on a specific virtual machine during the VMRUN instruction if the hypervisor sets the SEV enable (bit 1) in VMCB offset 090h.

When SEV is enabled in VMCB, the following additional consistency checks are performed during VMRUN:

- Nested paging must be enabled

[AMD Public Use]

- SmmLock bit in HWCR MSR must be set
- ASID must not be greater than the maximum value defined by CPUID Fn8000_001F_ECX[NumEncryptedGuests]

If any of the above consistency checks fail, the VMRUN instruction will terminate with a VMEXIT_INVALID error code. If MemEncryptionModEn is 0, SEV cannot be enabled and the VMCB control bit for SEV is ignored.

Note that on systems where CPUID Fn8000_001F_EAX[64BitHost] is set to 1, the hypervisor must be in 64-bit mode in order to execute a VMRUN to an SEV-enabled guest. If not, the VMRUN fails with a VMEXIT_INVALID error code.

15.34.4 Supported Operating Modes

Secure Encrypted Virtualization may be enabled on guests running in any operating mode. However the guest is only able to control memory encryption when operating in long mode or legacy PAE mode. In all other modes, all guest memory accesses are unconditionally considered private and are encrypted with the guest-specific key.

15.34.5 SEV Encryption Behavior

When a guest is executed with SEV enabled, the guest page tables are used to determine the C-bit for a memory page and hence the encryption status of that memory page. This allows a guest to determine which pages are private or shared, but this control is available only for data pages. Memory accesses on behalf of instruction fetches and guest page table walks are always treated as private, regardless of the software value of the C-bit. This behavior ensures non-guest entities (such as the hypervisor) cannot inject their own code or data into an SEV-enabled guest. If a guest does wish to make data in instruction pages or page tables accessible to code outside of the guest, this data must be explicitly copied into a shared data page.

Note that while the guest may choose to set the C-bit explicitly on instruction pages and page table addresses, the value of this bit is a don't-care in such situations as hardware always performs these as private accesses.

15.34.6 Page Table Support

An SEV-enabled guest controls encryption in its own guest page tables using the C-bit defined by CPUID 8000_001F[EBX]. This location is the same C-bit location as defined under SME (Section 7.10, “Secure Memory Encryption,” on page 240) in non-virtualized mode. If the C-bit is an address bit, this bit is masked from the guest physical address when it is translated through the nested page tables. Consequently, the hypervisor does not need to be aware of which pages the guest has chosen to mark private.

For example if the C-bit is address bit 47, when a guest accesses virtual address 0x54321, it might be translated to guest physical address 0x8000_00AB_C321, indicating the page should be encrypted with the private guest key. When this guest physical address is translated through the nested page tables, host virtual address 0xAB_C321 is used for translation. The C-bit value from the guest physical

[AMD Public Use]

address is saved and used on the final system physical address after the nested table translation as shown in Figure 15-30.

Note that because guest physical addresses are always translated through the nested page tables, the size of the guest physical address space is not impacted by any physical address space reduction indicated in CPUID 8000_001F[EBX]. If the C-bit is a physical address bit however, the guest physical address space is effectively reduced by 1 bit.

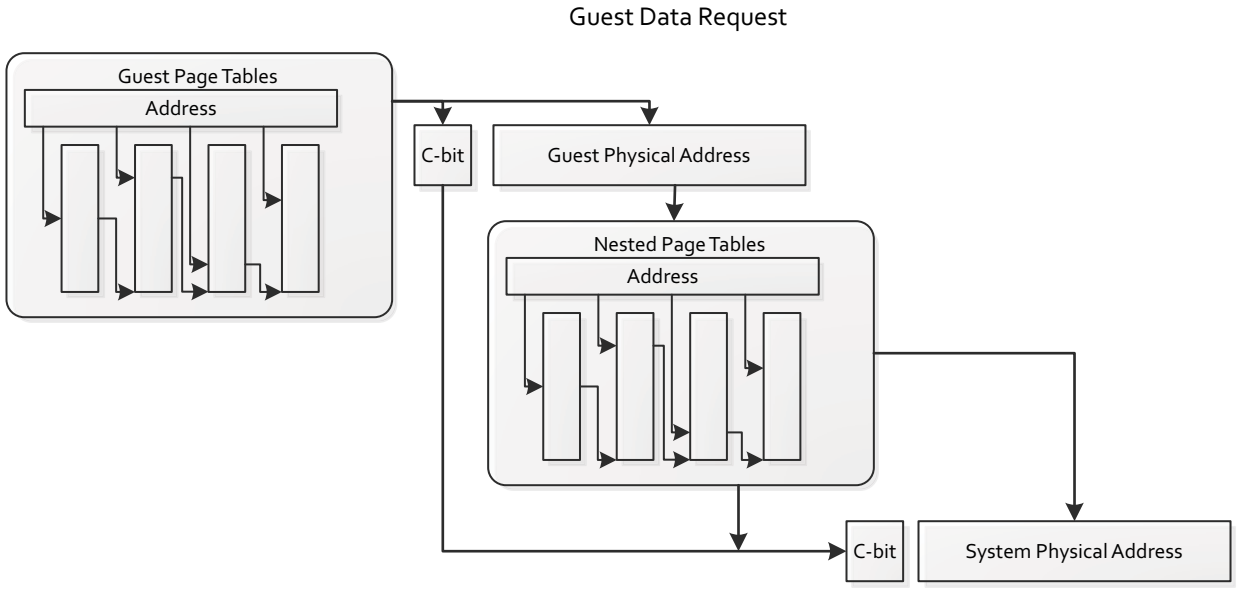


Figure 15-30. Guest Data Request

15.34.7 Restrictions

As with SME, some hardware implementations may not enforce coherency between mappings of the same physical page with different encryption enablement or keys. In such a system, when the encryption enablement or key for a particular memory page is to be changed, software must first ensure the page is flushed from all CPU caches. Certain conventional cache flushing techniques may not work however; see Section 15.34.9 for further details.

Note that if the hardware implementation enforces coherency across encryption domains as indicated by CPUID Fn8000_001F_EAX[10] then this flush is not required.

15.34.8 SEV Interaction with SME

SEV may be used in conjunction with SME mode. In this scenario, the guest page tables control encryption for guest memory, and the host (nested) page tables control encryption for shared memory. This behavior is summarized in Table 15-30. SEV is considered active when the CPU is in guest mode and the guest has SEV enabled in the VMCB.

[AMD Public Use]

Table 15-30. Encryption Control

Type of Access	MemEncryptionModEn	Guest Mode	SEV Mode Active	Encrypted	Encryption Key	Notes
Legacy Mode (memory encryption disabled)						
All	0	X	X	No	N/A	
Secure Memory Encryption Mode						
All	1	0	X	Optional	Host Key	Determined by page tables (CR3)
All	1	1	0	Optional	Host Key	Determined by nested page tables (hCR3)
Secure Encrypted Virtualization Mode						
Instruction Fetch	1	1	1	Yes	Guest Key	
Guest Page Table Access	1	1	1	Yes	Guest Key	
Nested Page Table Access	1	1	1	Optional	Host Key	Determined by nested page tables (hCR3)
Data Access	1	1	1	Optional[1]	See Table 15-31: SEV/SME Interaction	Determined by guest page tables (gCR3) and nested page tables (hCR3)

Note:

1. Encryption is guest-controlled in long mode and legacy PAE mode only. In all other modes, these accesses are always considered private and are encrypted with the guest key

Note that during a nested page table walk, it is possible for both the guest page tables to be encrypted and the nested page tables to be encrypted. In this scenario, the guest page tables are decrypted using the guest private encryption key, and the nested page tables are decrypted using the host (SME) encryption key.

Guest data accesses that are marked shared (C=0) by the guest may still be optionally encrypted using the host (SME) key if the pages are marked encrypted in the nested tables. If a page is marked encrypted in both the guest and nested tables, the guest tables have priority and the page will be encrypted using the guest key. This behavior is summarized in Table 15-31.

[AMD Public Use]

Table 15-31. SEV/SME Interaction

		Nested Page Table	
		C=0	C=1
Guest Page Table	C=0	Unencrypted	Encrypted with host key
	C=1	Encrypted with guest key	Encrypted with guest key

15.34.9 Page Flush MSR

If coherency across encryption domains is not supported (see "Restrictions" on page 616), and the hypervisor wishes to read an encrypted page, it must first flush the guest view of that page from all CPU caches to ensure it is able to view the most recent copy of that data. This may be accomplished by issuing a WBINVD instruction on all cores on which the guest has run, or by using the VMPAGE_FLUSH MSR (C001_011E). Support for the VMPAGE_FLUSH MSR is indicated in CPUID 8000_001F[EAX] bit 2.

The VMPAGE_FLUSH MSR is a write-only register that may be used to flush 4KB of data on behalf of a guest. The hypervisor writes the host linear address of the page and guest ASID to the MSR, and hardware will then perform a write-back invalidation of the page causing any dirty data present in any CPU caches throughout the system to be encrypted and written to DRAM. Note that the VMPAGE_FLUSH MSR uses the standard host page tables to perform the page translation. The Page Flush MSR operation will hit on and evict guest-cached instances of the memory, whereas CLFLUSH instructions using this same translation will not.

Bit[s]	Description
63:12	**VirtualAddr:** Write-only. Host virtual address of page to flush
11:0	**ASID:** Write-only. Guest ASID to use for the flush

The VMPAGE_FLUSH MSR will only flush memory pages marked private by the guest. If the hypervisor does not know if the memory page was marked private but wishes to evict the page from the cache, it should perform a standard CLFLUSH in addition to using the VMPAGE_FLUSH MSR.

Attempts to flush a host virtual address that is not mapped into a physical address or use of an ASID=0 will cause a #GP(0) fault. If SMAP is enabled, the input address needs to be mapped in page tables as a supervisor address.

If coherency across encryption domains is supported, the CLFLUSH instruction may be used to evict guest pages from the cache regardless of whether they are marked private or not by the guest.

15.34.10 SEV_STATUS MSR

Guests can determine what SEV features are currently active by reading the SEV_STATUS MSR (C001_0131). This MSR indicates which SEV features were enabled in the last VMRUN for that guest as shown in Table 15-32. The SEV_STATUS MSR is read-only and accesses to the SEV_STATUS

[AMD Public Use]

MSR cannot be intercepted by the hypervisor. The SEV_STATUS MSR is available on processors that support SEV.

Table 15-32. SEV_STATUS MSR Fields

Bit[s]	Description
63:24	**Reserved**
23	**IbpbOnEntry_Active:** IBPB on Entry feature is enabled in SEV_FEATURES[21]
22-20	**Reserved**
19	**EnhSmtProtection_Active:** Enhanced SMT Protection feature is enabled in SEV_FEATURES[17]
18	**SecureAVIC_Active:** Secure AVIC feature is enabled in SEV_FEATURES[16]
17	**SmtProtection_Active:** SMT Protection feature is enabled in SEV_FEATURES[15]
16	**VmsaRegProt_Active:** VMSA Register Protection feature is enabled in SEV_FEATURES[14]
15	**GuestInterceptCtl_Active:** Guest Intercept Control feature is enabled in SEV_FEATURES[13]
14	**IbsVirtualization_Active:** IBS Virtualization feature is enabled in SEV_FEATURES[12]
13	**PmcVirtualization_Active:** PMC Virtualization feature is enabled in SEV_FEATURES[11]
12	**VmgexitParameter_Active:** VMGEXIT Parameter feature is enabled in SEV_FEATURES[10]
11	**SecureTsc_Active:** Secure TSC feature is enabled in SEV_FEATURES[9]
10	**VmplSSS_Active:** VMPL SSS feature is enabled in SEV_FEATURES[8]
9	**SNPBTBIsolation_Active:** BTB isolation feature is enabled in SEV_FEATURES[7]
8	**PreventHostIBS_Active:** PreventHostIBS feature is enabled in SEV_FEATURES[6]
7	**DebugVirtualization_Active:** Debug Virtualization feature is enabled in SEV_FEATURES[5]
6	**AlternateInjection_Active:** Alternate Injection feature is enabled in SEV_FEATURES[4]
5	**RestrictedInjection_Active:** Restricted Injection feature is enabled in SEV_FEATURES[3]
4	**ReflectVC_Active:** ReflectVC feature is enabled in SEV_FEATURES[2]
3	**vTOM_Active:** Virtual TOM feature is enabled in SEV_FEATURES[1]
2	**SNP_Active:** SNP-Active mode is selected by SEV_FEATURES[0]
1	**SEV_ES_Enabled:** SEV-ES feature is enabled in VMCB offset 90h
0	**SEV_Enabled:** SEV feature is enabled in VMCB offset 90h

15.34.11 Virtual Transparent Encryption (VTE)

The Virtual Transparent Encryption feature can be enabled to force all memory accesses within an SEV guest to be encrypted with the guest's key. Support for this feature is indicated in CPUID Fn8000_001F[EAX] bit 16.

To enable this feature, the hypervisor must set VMCB offset 90h bit 5. Bit 5 is only observed when SEV (bit 1) is also set to 1 and SEV-ES (bit 2) is cleared to 0. In all other configurations of these bits (namely SEV disabled or SEV-ES enabled), bit 5 is ignored by hardware.

When this feature is enabled, CPU hardware treats the guest C-bit as 1 for all guest memory references. The actual C-bit in the guest page tables is ignored by hardware.

[AMD Public Use]

Guest address translation is unchanged, so the guest physical address (without the C-bit) is used for translation in the nested page tables.

15.35 Encrypted State (SEV-ES)

Encrypted VMs that use the SEV feature described in Section 15.34 may additionally use the SEV-ES feature to protect guest register state from the hypervisor. An SEV-ES VM's CPU register state is encrypted during world switches and cannot be directly accessed or modified by the hypervisor. This is designed to protect against attacks such as exfiltration (unauthorized reading of VM state) and control flow attacks (modifying VM state) including rollback attacks (restoring an earlier VM register state).

SEV-ES includes architectural support for notifying a VM's operating system when certain types of world switches are about to occur, allowing the VM to selectively share information with the hypervisor when needed for functionality.

15.35.1 Determining Support for SEV-ES

SEV-ES support can be determined by reading CPUID Fn8000_001F[EAX] as described in Section 15.34.1. Bit 3 of EAX indicates support for SEV-ES.

15.35.2 Enabling SEV-ES

SEV-ES may be enabled on a per-VM basis by setting bit 2 in offset 90h of the VMCB. When enabling SEV-ES, the hypervisor must also enable SEV (offset 90h bit 1) and LBR virtualization (offset B8h bit 0). Additionally, all other programming requirements related to enabling SEV (see Section 15.34.3) must be satisfied when running an SEV-ES guest.

On some systems, there is a limitation on which ASID values can be used on SEV guests that are run with SEV-ES disabled. While SEV-ES may be enabled on any valid SEV ASID (as defined by CPUID Fn8000_001F[ECX]), there are restrictions on which ASIDs may be used for SEV guests with SEV-ES disabled. CPUID Fn8000_001F[EDX] indicates the minimum ASID value that must be used for an SEV-enabled, SEV-ES-disabled guest. For example, if CPUID Fn8000_001F[EDX] returns the value 5, then any VMs which use ASIDs 1-4 and which enable SEV must also enable SEV-ES.

Note that prior to running an SEV-ES VM for the first time, the hypervisor must coordinate with the AMD Secure Processor to create the initial encrypted state image for the guest VM.

15.35.3 SEV-ES Overview

The SEV-ES architecture is designed to protect guest VM register state by default, and only allow the guest VM itself to grant selective access as required. This additional security protection functionality is accomplished in two ways. First, all VM register state is saved and encrypted when a VM exit event (#VMEXIT) occurs. This state is decrypted and restored on a VMRUN only. Second, certain types of #VMEXIT events cause a new exception to be taken within the guest VM. This new exception (#VC, see Section 15.35.5) indicates that the guest VM performed some action which requires hypervisor involvement, an example of which would be an I/O access by the VM. The guest #VC handler is

[AMD Public Use]

responsible for determining what register state is necessary to expose to the hypervisor for the purpose of emulating this operation. The #VC handler also inspects the returned values from the hypervisor and updates the guest state if the output is deemed acceptable.

Register state that needs to be exposed utilizes a new structure called the Guest-Hypervisor Communication Block (GHCB). The GHCB location is chosen by the guest who maps the page as a shared memory page, thus allowing direct hypervisor access. Only state located in the GHCB can be read by the hypervisor as all state stored in the traditional VMCB save state structure is encrypted using the guest memory encryption key and integrity protected.

In the #VC handler, the guest may utilize a new instruction (Section 15.35.6) to perform a world switch and invoke the hypervisor. In response to this, the hypervisor can inspect the GHCB and determine the services requested by the guest.

15.35.4 Types of Exits

When SEV-ES is enabled, all #VMEXIT events are classified as either Automatic Exits (AE) or Non-Automatic Exits (NAE). AE events are generally events that occur asynchronously with respect to the guest execution (e.g. interrupts) or events that need not involve exposing any guest register state. All other #VMEXIT events are classified as NAE events, and with NAE events the guest is allowed to determine what register state (if any) to expose in the GHCB. During guest execution, #VMEXIT events (both AE and NAE) are only taken if the corresponding intercept bit in the VMCB control area is set.

The hypervisor is informed of specific AE events exclusively via the #VMEXIT codes within the EXITCODE field of the VMCB control area. NAE events result in a #VC exception which is handled by the guest. Table 15-33 lists the possible AE events, all other events are considered NAE events.

Table 15-33. AE Exitcodes

Code	Name	Notes	HW Advances RIP
52h	VMEXIT_MC	Machine check exception	No
60h	VMEXIT_INTR	Physical INTR	No
61h	VMEXIT_NMI	Physical NMI	No
62h	VMEXIT_SMI	Physical SMI	No
63h	VMEXIT_INIT	Physical INIT	No
64h	VMEXIT_VINTR	Virtual INTR	No
77h	VMEXIT_PAUSE	PAUSE instruction	Yes
78h	VMEXIT_HLT	HLT instruction	Yes
7Fh	VMEXIT_SHUTDOWN	Shutdown	No
8Fh	VMEXIT_EFER_WRITE_TRAP	See Section 15.35.10	Yes
90h	VMEXIT_CR0_WRITE_TRAP	See Section 15.35.10	Yes
93h	VMEXIT_CR3_WRITE_TRAP	See Section 15.35.10	Yes
94h	VMEXIT_CR4_WRITE_TRAP	See Section 15.35.10	Yes

[AMD Public Use]

Table 15-33. AE Exitcodes (continued)

Code	Name	Notes	HW Advances RIP
A5h	VMEXIT_BUSLOCK	Bus Lock Threshold	No
A6h	VMEXIT_IDLE_HLT	HLT instruction if idle	Yes
400h	VMEXIT_NPF	Only if PFCODE[3]=0	No
403h	VMEXIT_VMGEXIT	VMGEXIT instruction	Yes
407h	VMEXIT_PML_FULL	PML Buffer overflow	No
–1	VMEXIT_INVALID	Invalid guest state	–
–2	VMEXIT_BUSY	BUSY bit was set in VMSA	–
-3	VMEXIT_IDLE_REQUIRED	The sibling thread is not idle	–
-4	VMEXIT_INVALID_PMC	Invalid PMC state	–
-5	VMEXIT_ILLSIB	Illegal Sibling	–
-6	VMEXIT_ESMTP_TIMEOUT	ESMTP Timeout	–

In the case of exits due to specific instructions, the CPU will automatically advance the guest RIP in response to the AE so that execution will resume at the next instruction on a subsequent VMRUN.

In the case of nested page faults, these are treated as AEs only if there was no reserved bit error. This is intended to be used to help distinguish nested page faults due to demand misses (hypervisor needs to allocate a page) vs MMIO emulation (hypervisor needs to emulate a device). Consequently, the hypervisor should set a reserved page table bit, such as a reserved address bit, on all MMIO pages that it intends to emulate. (This can include address bits that may become reserved when SEV is enabled; see Section 15.34.1.) This will ensure that MMIO page faults become NAE events, so the guest #VC handler can be invoked to assist in the MMIO emulation. Nested page faults that are AE events do not invoke any guest handler and the hypervisor is intended to allocate memory as needed and then resume the guest.

Note that when a guest is running with SEV-ES enabled, instruction bytes (VMCB offset D0h) are never saved to the VMCB on a nested page fault.

15.35.5 #VC Exception

The VMM Communication Exception (#VC) is always generated by hardware when an SEV-ES enabled guest is running and an NAE event occurs. The #VC exception is a precise, contributory, fault-type exception utilizing exception vector 29. This exception cannot be masked. The error code of the #VC exception is equal to the #VMEXIT code (see Appendix C) of the event that caused the NAE.

In response to a #VC exception, a typical flow would involve the guest handler inspecting the error code to determine the cause of the exception and deciding what register state must be copied to the GHCB for the event to be handled. The handler should then execute the VMGEXIT instruction to create an AE and invoke the hypervisor. After a later VMRUN, guest execution will resume after the VMGEXIT instruction where the handler can view the results from the hypervisor and copy state from the GHCB back to its internal state as needed. This flow is shown in Figure 15-31.

[AMD Public Use]

Note that it is inadvisable for the hypervisor to set the VMCB intercept bit for the #VC exception as this would prevent proper handling of NAEs by the guest. Similarly, the hypervisor should avoid setting intercept bits for events that would occur in the #VC handler (such as IRET).

[AMD Public Use]

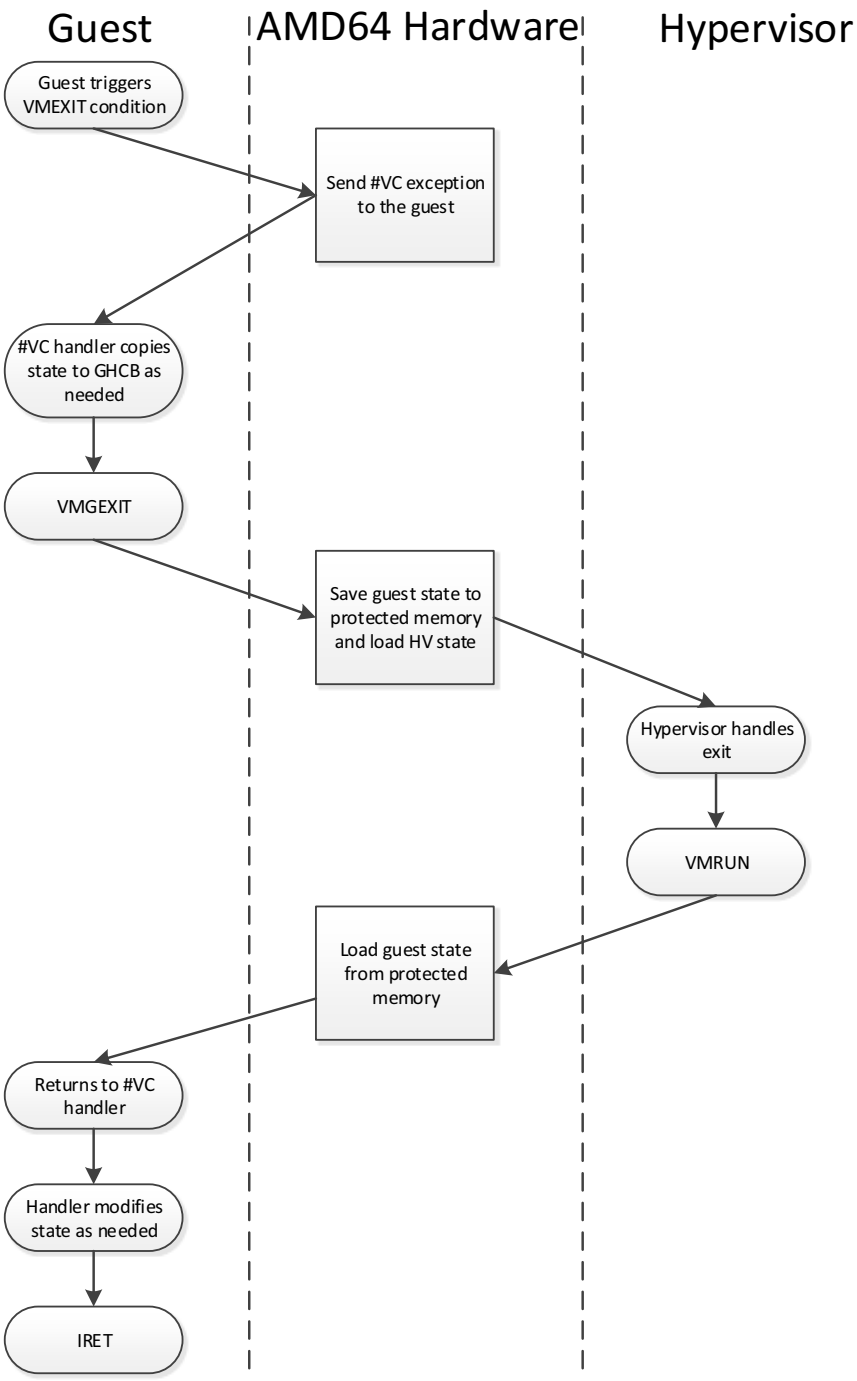


Figure 15-31. EXAMPLE #VC FLOW

[AMD Public Use]

15.35.6 VMGEXIT

The VMGEXIT instruction creates an AE and is intended to allow a guest #VC handler to invoke the hypervisor when needed. VMGEXIT causes an AE with the VMEXIT_VMGEXIT code and behaves like a trap so that upon a subsequent VMRUN, execution resumes following the VMGEXIT. There is no hypervisor intercept bit for VMGEXIT as the instruction unconditionally causes an AE when executed in an SEV-ES guest.

The VMGEXIT opcode is only valid within a guest when run with SEV-ES mode active. If the guest is not run with SEV-ES mode active, the VMGEXIT opcode will be treated as a VMMCALL opcode and will behave exactly like a VMMCALL.

The VMGEXIT Parameter feature may be used by a guest #VC handler to atomically pass one parameter to the hypervisor. Support for the VMGEXIT Parameter feature is indicated by CPUID Fn8000_001F_EAX[VmgexitParameter] (bit 17) = 1. The VMGEXIT Parameter feature is enabled by setting SEV_FEATURES bit 10 (VmgexitParameter) in VMSA. When this feature is enabled, the RAX and the CPL values are written to the VMCB control area on VMGEXIT at offsets 110h (VMGEXIT_RAX) and 118h (VMGEXIT_CPL).

15.35.7 GHCB

The GHCB is an unencrypted memory page used to communicate register state between the SEV-ES guest and the hypervisor. The guest VM is able to set the location of the GHCB via the GHCB MSR (C001_0130). This value is also included in the VMCB and is saved/restored on VMRUN/#VMEXIT respectively.

The GHCB MSR is used to set up the location of the GHCB memory page. The format of this MSR is defined below:

Bit	Function
63:0	Guest physical address of GHCB

The value of this MSR is saved/restored from the VMCB offset 0A0h. It is recommended software write this MSR with a page-aligned address. The GHCB MSR can be read/written only in guest mode, attempts to access this MSR in host mode will result in a #GP.

Hardware never accesses the GHCB directly, and as a result the format of the GHCB is not fixed.

15.35.8 VMRUN

When SEV-ES is enabled, the VM save state area does not reside at offset 400h in the VMCB page. Instead it resides starting at offset 0h in a separate page called the VM Save Area (VMSA) as indicated by the VMSA Pointer at offset 108h. The VMSA Pointer value is stored as a host physical address. Hardware always accesses the VMSA save state area using encrypted memory accesses utilizing the guest's memory encryption key.

When hardware executes a VMRUN instruction and the VMCB indicates SEV-ES is enabled for the guest, the hardware loads guest state from the encrypted save state area indicated by the VMSA

[AMD Public Use]

Pointer. Also, the VMRUN instruction will perform the following actions in addition to the standard VMRUN behavior:

- Calculate a checksum over guest state to verify integrity
- Perform a VMLOAD to load additional guest register state
- Load guest GPR state
- Load guest FPU state

When a guest has SEV-ES enabled, the encrypted VM state save area definition is expanded to include all GPR and FPU state (see Appendix B). If any part of the VMRUN flow faults or if the integrity checksum fails to match, a #VMEXIT(VMEXIT_INVALID) is generated.

If SEV-ES is enabled, the VMRUN instruction ignores bits 10:5 of the VMCB clean bits and always reloads the full guest state.

For SEV-ES guests, while the full guest state is loaded on VMRUN only the minimal hypervisor state defined by the legacy VMRUN instruction (see Section 15.5.1) is saved to the host save area. The hypervisor itself should save its desired additional segment state and GPR values to the host save area since these values will be restored by hardware on a subsequent VMEXIT. Hardware does not automatically save host state such as FS, STAR, or GPR values from the hypervisor on a VMRUN. See Appendix B for a detailed breakdown of each piece of VMCB state.

Software interrupts and exception vectors 3 and 4 may not be injected into SEV-ES guests. If this is attempted, the VMRUN will fail with a VMEXIT_INVALID error code.

15.35.9 Automatic Exits

When an automatic exit event occurs while an SEV-ES enabled guest is executing, hardware automatically saves guest state to the encrypted save state area and restores hypervisor state from the host save area. Specifically, in addition to the standard state saved/restored by the VMEXIT flow, hardware will also perform the following steps:

- Perform a VMSAVE to save additional guest register state
- Save guest GPR state
- Save guest FPU state
- Calculate and store a checksum over the guest state for use in a subsequent VMRUN
- Perform a VMLOAD to load additional host register state
- Load host GPR state
- Re-initialize FPU state to their reset values

The loading of host GPR state from the host save area is done using the format of the expanded VMCB described in Appendix B. All register state is either loaded from this location or re-initialized to default values so no guest register state is visible to the hypervisor.

[AMD Public Use]

15.35.10 Control Register Write Traps

CR0, CR3, CR4 and EFER MSR write trap intercepts (VMCB offset 10h, bits 16, 19, 20, and 15 respectively) allow the hypervisor to track the guest mode and verify if desired features are being enabled by the guest. These intercepts cause an AE after a control register has been modified. The new value of the control register is saved in EXITINFO1 in the VMCB. Control register write traps are only supported for SEV-ES guests.

15.35.11 Interaction with SMI and #MC

If an SMI occurs while an SEV-ES guest is executing, the platform SMI handler is not immediately executed. Instead, the SMI will remain pending and a #VMEXIT(SMI) is generated. The SMI will then be taken in hypervisor context after STGI is executed. Note that this behavior occurs regardless of the value of the SMI intercept bit in the VMCB.

In some systems, machine check errors are first delivered as an SMI. If this occurs while an SEV-ES guest is executing, #VMEXIT(SMI) will be generated and EXITINFO1[MCREDIR] will be set to 1 (“SMI Intercept” on page 545). As described above, the SMI will be held pending until STGI is executed. After the platform SMI handler executes following STGI, the hypervisor should check the MCREDIR bit to determine if the #VMEXIT(SMI) was due to a machine check error in the guest and handle it appropriately.

15.36 Secure Nested Paging (SEV-SNP)

The SEV-SNP features enable additional protection for encrypted VMs designed to achieve stronger isolation from the hypervisor. SEV-SNP is used with the SEV and SEV-ES features described in Section 15.34 and Section 15.35 respectively and requires the enablement and use of these features.

Primarily, SEV-SNP provides integrity protection of VM memory to help prevent hypervisor-based attacks that rely on guest data corruption, aliasing, replay, and various other attack vectors. To achieve this, a new system-wide data structure called the Reverse Map Table (RMP) is used to perform additional security checks on memory access as described in Section 15.36.3.

In addition to memory protection, SEV-SNP also includes several security features including a new Virtual Machine Privilege Level (VMPL) architecture, interrupt injection restrictions, and side-channel protection. These features are designed to enable additional use models and enhanced security protections.

While this chapter describes the CPU hardware behavior of SEV-SNP, the technology also requires the use of the AMD Secure Processor (AMD-SP) SEV-SNP Application Binary Interface (ABI) to manage the lifecycle events of SEV-SNP VMs. See the SEV-SNP ABI specification (PID#56860) on AMD’s website for more details.

[AMD Public Use]

15.36.1 Determining Support for SEV-SNP

Support for SEV-SNP can be determined by reading CPUID Fn8000_001F[EAX] as described in Section 15.34.1. Bit 4 indicates support for SEV-SNP, while bit 5 indicates support for VMPLs. The number of VMPLs available in an implementation is indicated in bits 15:12 of CPUID Fn8000_001F[EBX].

CPUID Fn8000_001F[EAX] also indicates support for additional security features used with SEV-SNP guests, which are described in the following sections.

15.36.2 Enabling SEV-SNP

SEV-SNP depends on SEV for confidentiality protection. Before enabling SEV-SNP, the MemEncryptionModEn bit in MSR C001_0010 (SYSCFG) must be set, and all programming requirements described in Section 15.34.3 must be satisfied. After SecureNestedPagingEn is set to 1 in MSR C001_0010, certain MSRs may no longer be modified. This includes the fixed range MTRR registers (see Section 7.7.2), the IORR registers (see Section 7.9.2), the TOP_MEM and TOP_MEM2 registers (see Section 7.9.4), the SMM_KEY register (see section 15.32.2), as well as the SYSCFG MSR. Attempts to write the SYSCFG MSR after SecureNestedPagingEn is set to 1 will be ignored while attempts to write the other MSRs mentioned will result in #GP(0).

Enabling SEV-SNP requires a two-step initialization procedure:

1. Construct the Reverse Map Table (RMP) as described in Section 15.36.4.
2. Set VMPLEn and SecureNestedPagingEn in MSR C001_0010 (SYSCFG) on every core in the system.

After the SEV-SNP feature has been globally enabled, SEV-SNP can be activated on a per-VM basis by setting bit 0 of the SEV_FEATURES field at offset 3B0h of the VMSA during VM creation. SEV-SNP activated VMs must also enable SEV-ES as described in Section 15.35.2 and SEV as described in Section 15.34.3.

In this chapter, the term *SNP-enabled* indicates that SEV-SNP is globally enabled in the SYSCFG MSR. The term *SNP-active* indicates that SEV-SNP is enabled for a specific VM in the SEV_FEATURES field of its VMSA. While SNP-enabled systems support both SNP-active and non-SNP-active VMs, SNP-active VMs can only run on SNP-enabled systems.

15.36.3 Reverse Map Table

The Reverse Map Table (RMP) is a structure shared globally by all logical processors that resides in system memory and is used to ensure a one-to-one mapping between system physical addresses and guest physical addresses. Each page of physical memory that is potentially assignable to guests has one entry within the RMP. RMP entries contain the security attributes of the system physical page as described in Table 15-34.

[AMD Public Use]

Table 15-34. Fields of an RMP Entry

Name	Notes
Assigned	Flag indicating that the system physical page is assigned to a guest or to the AMD-SP. 0: Owned by the hypervisor 1: Owned by a guest or the AMD-SP
Page_Size	Encoding of the page size. 0: 4KB page 1: 2MB page
Immutable	Flag indicating that software can alter the entry via x86 RMP manipulation instructions. 0: RMP entry can be altered by software 1: RMP entry cannot be altered by software
Guest_Physical_Address	Guest physical address associated with the page
ASID	ASID of guest to which page is assigned
VMSA	Flag indicating that the page is a VMSA page. 0: Non-VMSA page 1: VMSA page
Validated	Flag indicating that the guest has validated the page. See Section 15.36.6 for details. 0: The guest has not yet validated the page 1: The guest validated the page with PVALIDATE
Not-Dirty	Flag indicating whether a private page has been written. 0: Page has been written 1: Page has not been written See RMPADJUST, RMPQUERY and RMPCHKD instructions in Volume 3 for details.
Permissions[0]	VMPL permission masks for the page. See Section 15.36.7 for details.
...	
Permissions[n-1]	

The integrity of the RMP is maintained by restricting software manipulation of it to the following special-purpose instructions:

- RMPUPDATE: Available to the hypervisor to alter the Guest_Physical_Address, Assigned, Page_Size, Immutable, and ASID fields of an RMP entry. See Section 15.36.5 for details.
- PSMASH: Allows the hypervisor to split a 2MB entry in the RMP into 512 4KB entries in the RMP. See Section 15.36.11 for details.
- RMPADJUST: Allows a guest to alter the VMPL permission masks of the RMP entry. See Section 15.36.7 for details.

[AMD Public Use]

- PVALIDATE: Allows a guest to write to the Validated flag in the RMP entry. See Section 15.36.6 for details.

When SEV-SNP is globally enabled, it adds more restrictions to page access controls. The hypervisor and the guests use the above instructions to enforce these restrictions on memory accesses. A violation of memory access restrictions indicated by the RMP will result in an exception. See Section 15.36.10 for details.

15.36.4 Initializing the RMP

This section describes the single-level RMP initialization. The two-level RMP is described in Section 15.36.22, “Segmented RMP,” on page 649.

MSR C001_0132 (RMP_BASE) defines the system physical address of the first byte of the RMP. The MSR C001_0133 (RMP_END) defines the system physical address of last byte of the RMP. Software must program RMP_BASE and RMP_END identically for each core in the system and before enabling SEV-SNP globally.

RMP_BASE and (RMP_END+1) must be 8KB aligned. The AMD-SP may place further alignment requirements on these registers. Refer to the latest AMD-SP specifications to determine the required alignment.

The region of memory between RMP_BASE and RMP_END contains a 16KB region used for processor bookkeeping followed by the RMP entries, which are each 16B in size. The size of the RMP determines the range of physical memory that the hypervisor can assign to SNP-active virtual machines at runtime. The RMP covers the system physical address space from address 0h to the address calculated by:

((RMP_END + 1 – RMP_BASE – 16KB) / 16B) x 4KB

For example, if the RMP_BASE is equal to 10_0000h, then to cover the first 4GB of physical memory, RMP_END must be set to 110_3FFFh, which makes the RMP just over 16MB.

Once SEV-SNP is globally enabled, memory accesses are restricted by RMP checks. To ensure that the RMP starts in a known and non-restrictive state, software should write zeros to all memory from RMP_BASE to RMP_END before setting the SecureNestedPagingEn bit in the SYSCFG MSR. The hypervisor then requests the AMD-SP to finalize the initialization of the RMP. The AMD-SP initializes the RMP to prevent all software from directly writing to the memory between RMP_BASE and RMP_END. All subsequent RMP entry manipulation must occur either via the x86 RMP manipulation instructions or through interactions with the AMD-SP.

15.36.5 Hypervisor RMP Management

The hypervisor manages the SEV-SNP security attributes of pages assigned to SNP-active guests by altering the RMP entries of those pages. Because the RMP is initialized by the AMD-SP to prevent direct access to the RMP, the hypervisor must use the RMPUPDATE instruction to alter the entries of the RMP. RMPUPDATE allows the hypervisor to alter the Guest_Physical_Address, Assigned, Page_Size, Immutable, and ASID fields of an RMP entry.

[AMD Public Use]

SEV-SNP associates an owner with each system physical page through settings of the Assigned, ASID, and Immutable fields of the page's RMP entry according to Table 15-35. A page can be owned by the hypervisor, a guest, or the AMD-SP.

Table 15-35. RMP Page Assignment Settings

Owner	Assigned	ASID	Immutable
Hypervisor	0	0	-
Guest	1	ASID of the guest	-
AMD-SP	1	0	1

When the hypervisor assigns a page to a guest, it must also set the Guest_Physical_Address and Page_Size to match the nested page table mapping for the guest. If not, access to the page by the guest will result in a fault. See Section 15.36.10 for details on the RMP access checks.

The hypervisor may transition any page that has Immutable set to 0 into a hypervisor-owned page by using RMPUPDATE to set Assigned to 0 and ASID to 0. To transition a page that has Immutable set to 1, the hypervisor must request the AMD-SP to transition the page.

The RMP initialization requirement to write zeros to the RMP (see Section 15.36.4) results in all pages in the system initially belonging to the hypervisor. Any memory pages which are not covered by the RMP are considered permanent hypervisor pages. For example, if the RMP is configured to only cover the first 4GB of memory then all memory above 4GB is considered hypervisor memory for the purpose of RMP access checks.

15.36.6 Page Validation

Each page assigned to a VM is either validated or unvalidated, as indicated by the Validated flag in the page's RMP entry. Memory accesses by the VM to private pages that are unvalidated generate a #VC. All pages are initially assigned as unvalidated.

The VM may use the PVALIDATE instruction to either set or clear the Validated flag of a page. It is expected that VMs would use PVALIDATE to set the Validated flag during VM startup to gain access to the memory the hypervisor has assigned. The VM may later use PVALIDATE to clear the Validated flag if its memory space is being reduced, such as after a memory hot-plug event.

Page validation allows a VM to detect an unexpected remapping of its pages by the hypervisor. Before accessing a page, the VM must validate the page. Once validated, any use of RMPUPDATE by the hypervisor to unassign, reassign, or remap the page will cause the page to become unvalidated. The VM can then detect tampering with the page mapping via the #VC that occurs from accessing unvalidated pages.

PVALIDATE takes a page size as an input parameter indicating that either a 4KB or 2MB page should be validated. If the VM attempts to use PVALIDATE on a 4KB page that is mapped to a 2MB or 1GB page in the nested page table, PVALIDATE generates an #VMEXIT(NPF). In this case, the hypervisor can smash the larger page into 4KB pages using the PSMASH instruction as described in

[AMD Public Use]

Section 15.36.11. If the VM attempts to use PVALIDATE on a 2MB guest page that is mapped to 4KB nested pages, PVALIDATE returns an error indication to the VM. The VM can instead attempt to execute PVALIDATE for each of the 4KB pages individually.

15.36.7 Virtual Machine Privilege Levels

A typical guest VM may consist of multiple vCPUs. SEV-SNP expands on this capability by enabling the vCPUs to run at distinct Virtual Machine Privilege Levels (VMPLs). Within a vCPU, the different VMPLs are represented by unique VMSAs and are expected to be run in a mutually exclusive manner. Each VMSA is assigned a VMPL as indicated by the VMPL field in the VMSA.

VMPLs are identified numerically starting at 0 with VMPL0 being the most privileged. The number of VMPLs available in an implementation is indicated in bits 15:12 of CPUID Fn8000_001F[EBX]. The VMPL feature enables a guest to sub-divide its address space and implement vCPU-specific access controls on a page-by-page basis.

The processor restricts guest memory accesses based on VMPL permission masks in RMP entries. Each RMP entry contains a set of permission masks, one mask for each implemented VMPL. On memory accesses, the processor checks the current VMPL permission mask of the page to determine whether the access is allowed. The permission mask bits are defined in Table 15-36.

Table 15-36. VMPL Permission Mask Definition

Bit	Name	Settings
0	Read (RD)	0: Reads cause #VMEXIT(NPF) 1: Reads are allowed
1	Write (WR)	0: Writes cause #VMEXIT(NPF) 1: Writes are allowed
2	Execute-User (XU)	0: Execution at CPL 3 causes #VMEXIT(NPF) 1: Execution at CPL 3 is allowed
3	Execute-Supervisor (XS)	0: Execution at CPL < 3 causes #VMEXIT(NPF) 1: Execution at CPL < 3 is allowed
4	Supervisor-Shadow-Stack (SSS)	0: SSS accesses cause #VMEXIT(NPF) 1: SSS accesses are allowed
5-7	Reserved	SBZ

When a guest access results in a #VMEXIT(NPF) due to a VMPL permission violation, an error code bit in EXITINFO1 is set as described in Section 15.36.10.

When the hypervisor assigns a page to a guest using RMPUPDATE, full permissions are enabled for VMPL0 and are disabled for all other VMPLs. A VM can then use the RMPADJUST instruction to modify the permissions of VMPLs numerically higher than its own. For example, a vCPU executing at VMPL0 could use RMPADJUST to restrict a page of memory to be only read-write but not executable at VMPL1. However, the vCPU executing at VMPL1 could not alter its own permissions or the permissions of VMPL0.

[AMD Public Use]

RMPADJUST cannot be used to grant greater permissions than what is allowed by the permission mask for the current VMPL. For example, if VMPL1 attempts to grant write permission to a page to VMPL2, but VMPL1 does not have write permission to the page, RMPADJUST will fail.

VMPL permission misconfiguration occurs when an attempt is made to grant permissions not supported by the architecture and causes the RMPADJUST instruction to fail. VMPL misconfiguration conditions are defined in Table 15-37.

Table 15-37. VMPL Misconfiguration Conditions

VMPL Permissions	Description
(R==0) && (Permissions[7:1] !=0)	Non-readable pages must not have any other permissions.
(SSS==1) && ((XU==1) || (XS==1))	Supervisor shadow stack pages must not be executable.
(XS==1) && (XU==0)	Supervisor executable pages must also be user executable.

VMPL Supervisor Shadow Stack. The VMPL Supervisor Shadow Stack (VMPL SSS) feature allows an SNP-active guest instead of the hypervisor (see Section 15.25.15) to restrict which guest physical addresses may be used for a guest supervisor shadow stack by designating SSS pages with VMPL SSS permission bit. Supervisor shadow stack accesses made by the guest to pages not designated as SSS pages in the VMPL Permissions result in a #VMEXIT(NPF).

Support for the VMPL SSS feature is indicated by CPUID Fn8000_001F_EAX[VmplSSS] (bit 7) = 1. The VMPL SSS feature is enabled by setting SEV_FEATURES bit 8 (VmplSSS) in VMSA. VMPL SSS and nested page controlled SSS feature are mutually exclusive. If SEV_FEATURES[VmplSSS] = 1 and VMCB[SSS] = 1, VMRUN fails with #VMEXIT(VMEXIT_INVALID).

If the VMPL SSS feature is enabled, than both supervisor and user shadow stack accesses must be to private memory pages. If not, a #PF exception with the reserved (RSV) error code bit set is generated in the guest.

15.36.8 Virtual Top-of-Memory

In the VMSA of an SNP-active guest, the VIRTUAL_TOM field designates a 2MB aligned guest physical address called the virtual top of memory. When bit 1 (vTOM) of SEV_FEATURES is set in the VMSA of an SNP-active VM, the VIRTUAL_TOM field is used to determine the C-bit for data accesses instead of the guest page table contents. All data accesses below VIRTUAL_TOM are accessed with an effective C-bit of 1 and all addresses at or above VIRTUAL_TOM are accessed with an effective C-bit of 0. Note that page table accesses and instruction fetches always have an effective C-bit of 1, regardless of the value of VIRTUAL_TOM or whether the feature is enabled.

The Virtual TOM MSR (C001_0135) may be used to change the VIRTUAL_TOM value. CPUID Fn8000_001F_EAX[VirtualTom] (bit 18) = 1 indicates support for the Virtual TOM MSR. This MSR register is read-write when Virtual TOM is active, and an attempt to access it when Virtual TOM is not active will result in a #GP(0) exception. Virtual TOM MSR bits 63:52 and 20:0 are Reserved, RAZ. A WRMSR to this register flushes TLB entries belonging to the current ASID.

[AMD Public Use]

When virtual top of memory is enabled in SEV_FEATURES, the C-bit in the guest page table entries must be zero for all accesses. Any guest memory accesses with the C-bit set to 1 in the guest page tables will result in a #PF due to a reserved bit error.

15.36.9 Reflect #VC

When running an SEV-SNP VM, the CPU generates #VC exceptions in response to events that may require hypervisor interaction. #VC exceptions and the events that may lead to them are discussed in Section 15.35.5. SEV-SNP VMs may either chose to handle #VC exceptions directly in their current guest context or turn #VC exceptions into Automatic Exits. This behavior is controlled by bit 2 (ReflectVC) of SEV_FEATURES. If this bit is set to 1, then any event that would otherwise lead to a #VC exception is instead turned into an Automatic Exit.

When a #VC is turned into an Automatic Exit, the guest VM terminates with an exit code of VMEXIT_VC. The error code for the #VC, which reflects the event which led to the #VC (e.g., VMEXIT_CPUID), is saved to the GUEST_EXITCODE field in the VMSA. Additional information about the event that caused the #VC is saved to the GUEST_EXITINFO1, GUEST_EXITINFO2, GUEST_EXITINTINFO, and GUEST_NRIP fields in the VMSA. The information saved to these fields is the same as the standard exit information provided for the event that occurred. For example, if the VM performs a port I/O instruction which is marked for interception, the GUEST_EXITCODE field will be set to VMEXIT_IOIO and the GUEST_EXITINFO1 field will contain information about the I/O port access, as defined in Section 15.10.2.

The Reflect #VC feature enables #VC events to be handled by a vCPU at a VMPL other than the one that initiated them. For example, a guest may contain a vCPU which consists of two different VMSAs. One VMSA is defined to execute at VMPL0, while the other has ReflectVC enabled and is defined to execute at VMPL3. When the vCPU running at VMPL3 encounters a #VC condition, the information is saved to its VMSA and control is returned to the hypervisor. The hypervisor may then run the vCPU at VMPL0 which can read the exit information saved to the VMPL3 VMSA, interact with the hypervisor as required, write appropriate response data back into the VMPL3 VMSA, and instruct the hypervisor to resume execution of the vCPU at VMPL3.

If the #VC event occurred during the processing of an interrupt or exception, the GUEST_EXITINTINFO.V bit will be set. If the Alternate Injection feature is enabled (see Section 15.36.15), hardware will automatically set the VINTR_CTRL[BUSY] bit in the VMSA. This enables a higher privileged VMPL to re-inject the event that caused the #VC.

The GUEST_EXITCODE, GUEST_EXITINFO1, GUEST_EXITINFO2, GUEST_EXITINTINFO, and GUEST_NRIP fields are populated by hardware on every Automatic Exit, regardless of the ReflectVC feature. For Automatic Exits other than reflected #VCs, these fields are set to the same values that are set in the VMCB.

15.36.10 RMP and VMPL Access Checks

When SEV-SNP is enabled globally, the processor places restrictions on all memory accesses based on the contents of the RMP, whether the accesses are performed by the hypervisor, a legacy guest VM, a

[AMD Public Use]

non-SNP guest VM or an SNP-active guest VM. The processor may perform one or more of the following checks depending on the context of the access:

- RMP-Covered: Checks that the target page is covered by the RMP. A page is covered by the RMP if its corresponding RMP entry is below RMP_END. Any page not covered by the RMP is considered a Hypervisor-Owned page.
- Hypervisor-Owned: Checks that if the target page is covered by the RMP then the Assigned bit of the target page is 0. If the page table entry that specifies the sPA indicates that the target page size is 2MB, then all RMP entries for the 4KB constituent pages of the target page must have the Assigned bit set to 0. Accesses to 1GB pages only install 2MB TLB entries when SEV-SNP is enabled, therefore this check treats 1GB accesses as 2MB accesses for purposes of this check.
- Guest-Owned: Checks that the ASID field of the RMP entry of the target page matches the ASID of the current VM.
- Reverse-Map: Checks that the Guest_Physical_Address of the RMP entry of the target page matches the guest physical address of the translation.
- Validated: Checks that the Validated field of the RMP entry of the target page is 1.
- Mutable: Checks that the Immutable field of the RMP entry of the target page is 0.
- Page-Size: Checks that the following conditions are met:
  - If the nested page table indicates a 2MB or 1GB page size, the Page_Size field of the RMP entry of the target page is 1.
  - If the nested page table indicates a 4KB page size, the Page_Size field of the RMP entry of the target page is 0.
- VMPL: Checks that the VMPL permission mask allows access. See Section 15.36.7 for details.

Table 15-38 describes under which conditions each check is performed and what fault is produced on failure.

Table 15-38. RMP Memory Access Checks

Host/Guest	SNP-Active	Type of Access	C-Bit	Check	Fault
Host	-	Data write, Page Table Access	-	Hypervisor-Owned	#PF
Guest	No	Data write, Page Table Access	-	Hypervisor-Owned	#VMEXIT(NPF)
Guest	Yes	Instruction Fetch, Page Table Access	-	RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size	#VMEXIT(NPF)
				Validated	#VC
				VMPL	#VMEXIT(NPF)

[AMD Public Use]

Table 15-38. RMP Memory Access Checks (continued)

Host/Guest	SNP-Active	Type of Access	C-Bit	Check	Fault
Guest	Yes	Data write	0	Hypervisor-Owned	#VMEXIT(NPF)
Guest	Yes	Data write, Data read	1	RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size	#VMEXIT(NPF)
				Validated	#VC
				VMPL	#VMEXIT(NPF)

In addition, any memory access that results in an RMP check may result in an RMP violation (#PF or #VMEXIT(NPF)) if the accessed RMP entries are in use by other logical processors. In this case, software should retry the access.

If a memory access results in a modification of the Accessed or Dirty bits in a page table entry, this page table modification is treated similarly to data write accesses by SEV-SNP. For any such page table modification access, the page size of the access is inherently 4KB.

If the virtual TOM feature (see Section 15.36.8) is enabled, then the Virtual TOM setting is used to determine the C-bit for a given guest access. Guest physical addresses below Virtual TOM are considered to have a C-bit set to 1.

The following page-fault error bits are set on an RMP check related #PF:

- Bit 31 (RMP): Set to 1 if the fault was caused due to an RMP check or a VMPL check failure, 0 otherwise. All RMP violations described in this section will set this bit to 1.

Additionally, the following page-fault error bits may be set on a #VMEXIT(NPF) in EXITINFO1:

- Bit 34 (ENC): Set to 1 if the guest's effective C-bit was 1, 0 otherwise.
- Bit 35 (SIZEM): Set to 1 if the fault was caused by a size mismatch between PVALIDATE or RMPADJUST and the RMP, 0 otherwise.
- Bit 36 (VMPL): Set to 1 if the fault was caused by a VMPL permission check failure, 0 otherwise.
- Bit 37 (SSS): Set to VMPL permission mask SSS (bit 4) value if VmplSSS is enabled.

The effective C-bit is always a 1 on any guest instruction fetch, page table access, or data write to private (C=1) memory.

All RMP checks described in this section occur after page table and nested page table access checks and have lower priority than existing paging checks. Table 15-38 reflects the relative priority of RMP checks. Namely, VMPL checks have the lowest priority, preceded by page validation checks. For example, if a guest access fails the Page-Size check and the Validated check, a #VMEXIT(NPF) will occur instead of a #VC since the Page-Size check has priority over the page validation check.

[AMD Public Use]

A failure of the page validation check results in a #VC with error code PAGE_NOT_VALIDATED (0x404). The faulting guest virtual address is saved to CR2 when this error occurs.

15.36.11 Large Page Management

The hypervisor may need to convert a 2MB page assigned to a guest into 4KB pages. This conversion is called page smashing and requires the hypervisor to alter the RMP. The hypervisor can use RMPUPDATE to alter the size of the page in the RMP, but this will clear the validated bit.

To convert a 2MB page into 4KB pages without altering the validated status of the region, the hypervisor may use the PSMASH instruction. PSMASH takes a 2MB aligned system physical address and smashes the page while preserving the Validated bit in the RMP. After PSMASH successfully completes, the RMP entries of the resulting 4KB pages have the following contents:

- Consecutive values in the Guest_Physical_Address fields
- Page_Size set to 0 indicating 4KB pages
- All other RMP fields copied from the original 2MB page RMP entry

One reason the hypervisor may need to smash a 2MB page is if the guest executes PVALIDATE or RMPADJUST on a 4KB page that is backed by a 2MB page. In that case, the instructions generate a #VMEXIT(NPF) with the SIZEM bit set in EXITINFO1. To resolve this, the hypervisor can smash the page and then have the guest restart the instruction.

If the guest wishes to validate a 2MB aligned region, the guest should first attempt to execute PVALIDATE with a size of 2MB. If the page is backed by 4KB pages, PVALIDATE terminates with a FAIL_SIZEMISMATCH error. In this case, the guest should then execute PVALIDATE on each 4KB page individually. This allows the guest to take advantage of the more efficient 2MB mappings and avoid having the hypervisor unnecessarily smash the page.

Table 15-39 summarizes the potential page size mismatches and how to resolve them.

Table 15-39. PVALIDATE/RMPADJUST Page Size Mismatch Combinations

Requested Page Size	Page Size in RMP	Error Condition	Recommended Handling
4KB	2MB	#VMEXIT(NPF)	PSMASH
2MB	4KB	FAIL_SIZEMISMATCH	Guest retries on each 4KB constituent page

The reverse operation of converting a set of consecutive 4KB pages into a single 2MB page requires assistance from either the guest or the AMD-SP to ensure that the operation is safe to perform.

15.36.12 Running SNP-Active Virtual Machines

As with SEV-ES guests, SNP-active guests are described by a hypervisor controlled VMCB and a guest encrypted VMSA. The initial VMSA for an SNP-active guest must be set up through coordination with the AMD-SP, the details of which are beyond the scope of this manual. This includes the initial configuration of the SEV_FEATURES field in the VMSA which indicates which

[AMD Public Use]

guest security features are enabled for that particular VM instance. VMRUN to an SNP-active guest will fail with a VMEXIT_INVALID error code if SEV-SNP is not globally enabled.

VMRUN Checks. When SEV-SNP is globally enabled on a system, the VMRUN instruction performs additional security checks on various memory pages. These checks are similar to the ones described in Section 15.36.10. Note that where a check depends on page size, a page size of 4KB is used. In addition to the checks described in that section, an additional check exists:

- VMSA: Checks that the VMSA field in the RMP entry equals 1.

The VMSA field in an RMP entry may be set by the AMD-SP, or by a vCPU running at VMPL0 using the RMPADJUST instruction.

The checks performed on VMRUN are as follows:

Table 15-40. VMRUN Page Checks

Page Type	SNP-Active	Check	Fault
VMCB	-	Hypervisor-Owned	#GP(0)
AVIC Backing Page	-	Hypervisor-Owned	#VMEXIT(VMEXIT_INVALID)
PML Buffer	-	Hypervisor-Owned	#VMEXIT(VMEXIT_INVALID)
IBS Buffer	-	Hypervisor-Owned	#VMEXIT(VMEXIT_INVALID)
VMSA	No	Hypervisor-Owned	#VMEXIT(VMEXIT_INVALID)
VMSA	Yes	RMP-Covered Guest-Owned Reverse-Map Mutable VMSA	#VMEXIT(VMEXIT_INVALID)

The AVIC Logical Table, AVIC Physical Table, IOPM_BASE_PA, and MSRPM_BASE_PA are not checked by VMRUN as these structures are only read by the hardware.

After a successful VMRUN, the VMCB page, as well as any page listed in Table 15-39 which is used by the processor between VMRUN and VMEXIT is marked in-use by hardware, and any attempt to modify the RMP entries for these pages via instructions like RMPUPDATE will result in a FAIL_INUSE response. The in-use marking is automatically cleared by hardware after a #VMEXIT event.

Other Checks. In addition to the RMP checks performed by VMRUN, a few other VM-related operations perform special RMP checks.

The address written to the VM_HSAVE_PA MSR, which holds the address of the page used to save the host state on a VMRUN, must point to a hypervisor-owned page. If this check fails, the WRMSR will fail with a #GP(0) exception. Note that a value of 0 is not considered valid for the VM_HSAVE_PA MSR and a VMRUN that is attempted while the HSAVE_PA is 0 will fail with a #GP(0) exception.

[AMD Public Use]

The VMSAVE instruction also performs checks to ensure that the target page is hypervisor-owned. The VMSAVE instruction is not expected to be used with SEV-ES and SNP-active guests, as described in Section 15.36.8, but may be used with other guests.

If VMSAVE is executed in host mode and the target page fails the RMP check, a #GP(0) exception is generated. If VMSAVE is executed in a guest when the VMSAVE instruction is virtualized (see Section 15.33.1) and the target page fails the RMP check, then a #VMEXIT(NPF) is generated indicating an RMP permission error. In processors that support SEV-SNP, the execution of the VMSAVE instruction inside an SEV-ES or SNP-active guest is not supported and will result in a #VMEXIT(VMSAVE).

Intercept Behavior. All port I/O (IN, INS, OUT, OUTS) and CPUID instructions executed by an SNP-Active guest are treated as intercepted regardless of the intercept bits set in the VMCB and IOPM. Execution of these instructions in an SNP-Active guest will unconditionally generate a Non-Automatic Exit.

15.36.13 Debug Registers

SEV-ES and SNP-active guests may choose to enable full virtualization of CPU debug registers through SEV_FEATURES bit 5 (DebugVirtualization).

When enabled, the DR[0-3] registers and DR[0-3]_ADDR_MASK registers are swapped as type ‘B’ state (see Appendix B).

15.36.14 Memory Types

When an SNP-active guest accesses memory, the hardware forces the use of coherent memory types. This prevents the hypervisor from attempting to corrupt guest memory by the use of non-coherent memory types for accesses by the guest.

If a guest memory access is determined to be non-coherent after the memory type determination logic described in Section 15.25.9, the hardware forces a coherent type as described in Table 15-41.

Table 15-41. Non-Coherent Memory Type Conversion

Non-Coherent Memory Type	Forced Coherent Memory Type
UC	CD
WC	WC+

15.36.15 TLB management

For non-SNP-active guests, when a hypervisor moves a VMSA to a new logical processor it must ensure that the VMSA cannot use any stale (incorrect) TLB translations to prevent corruption of the guest. For SNP-active guests, to avoid any dependency on the hypervisor for correctly managing guest TLB contents, the hardware detects when the VMSA is moved and manages the TLB for that guest automatically. The hardware uses two VMSA fields to track this information: the TLB_ID (byte offset 3D0h) and the PCPU_ID (byte offset 3D8h).

[AMD Public Use]

During guest creation, software should initialize the TLB_ID and PCPU_ID by setting both to zero. The hardware subsequently manages the values in both fields throughout the lifetime of that VMSA. During operation, software may explicitly write PCPU_ID to 0 to force a TLB flush on the next VMRUN to that VMSA if desired. If this occurs, the hardware will set the PCPU_ID field to a non-zero value when it flushes the TLB.

For example, when guest software performs an RMPADJUST to alter the permissions of a VMPL, it may need to ensure that the existing TLB entries of all vCPUs executing at the targeted VMPL are not used anymore. The guest software can do this by writing zero to PCPU_ID of the affected VMSAs. When these VMSAs are re-entered with VMRUN, the hardware will ensure existing TLB entries are no longer used and set PCPU_ID to a non-zero value. This value can be checked for non-zero by the guest software to ensure the operation has completed before proceeding.

As with any guest, the hypervisor may use the TLB_CONTROL field in the VMCB to force TLB flushes when desired. When the hypervisor writes 3h or 7h to TLB_CONTROL, both global and non-global TLB entries of the guest are invalidated.

15.36.16 Interrupt Injection Restrictions

SNP-active guests may choose to enable the Restricted Injection or Alternate Injection features through SEV_FEATURES bits 3 and 4 respectively. These features enforce additional interrupt and event injection security protections designed to help protect against malicious injection attacks. The two are mutually exclusive for a specific VMSA and an attempt to enable both will result in a #VMEXIT(VMEXIT_INVALID) when the VMRUN instruction is executed.

Restricted Injection. This feature disables all hypervisor-based interrupt queuing and event injection of all vectors except a new exception vector, #HV (28), which is reserved for SNP guest use, but never generated by hardware. #HV is only allowed to be injected into VMSAs that execute with Restricted Injection. #HV is a benign exception and can only be injected as an exception (VMCB.EVENTINJ[Type]=3) and without an error code. Guests running with Restricted Injection are expected to communicate with the hypervisor about events via a software-managed para-virtualization interface. This interface can use #HV injection as a doorbell to inform the guest that new events have been added.

The VMRUN instruction with Restricted Injection enabled will fail with a VMEXIT_INVALID error code if the hypervisor attempts the injection of any unsupported event or attempts to run the guest with AVIC enabled.

Alternate Injection. This feature replaces all hypervisor-based interrupt queuing and event injection with guest-controlled queuing and injection. When Alternate Injection is enabled in a VMSA, event injection information on VMRUN is read from the EventInjCtrl field in the VMSA (offset 3E0h) and interrupt queuing information is read from the VIntrCtrl field in the VMSA (offset 3B8h). This feature is intended to be used in a multi-VMPL architecture where a high privilege VMSA injects events and interrupts directly into a low privilege VMSA.

When Alternate Injection is enabled, the EventInjCtlr field in the VMCB (offset A8h) is ignored on VMRUN. The VIntrCtrl field in the VMCB (offset 60h) is processed, but only the

[AMD Public Use]

V_INTR_MASKING, Virtual GIF Mode, and AVIC Enable bits are used. The AVIC Enable bit must be 0 if the guest is running with Alternate Injection enabled, otherwise the VMRUN will fail with a VMEXIT_INVALID error code.

The remaining fields of VIntrCtrl (V_TPR, V_IRQ, VGIF, V_INTR_PRIO, V_IGN_TPR, V_INTR_VECTOR, V_NMI, V_NMI_MASK, V_NMI_EN) are read from the VMSA. Additionally, bit 10 of the encrypted VIntrCtrl field is defined as the INT_SHADOW bit and the unencrypted INT_SHADOW bit in VMCB offset 68h bit 0 is ignored. On a VMEXIT, the V_TPR, V_IRQ, V_NMI, V_NMI_MASK, and INT_SHADOW values are written back to the encrypted VIntrCtrl only.

In guests that run with Alternate Injection, bit 63 of the encrypted VIntrCtrl field is defined as a BUSY bit. On VMRUN, if VIntrCtrl[BUSY] is set to 1, then the VMRUN fails with a VMEXIT_BUSY error code. The BUSY bit enables a VMSA to be temporarily marked non-runnable while software modifications are in progress.

Additional Intercept Behavior. Additional hardware-forced intercept behavior exists in guests that run with either of these features enabled:

- For either feature, hardware treats physical INTR, NMI, INIT, and #MC events as intercepted regardless of the intercept bit set in the VMCB.
- Under Alternate Injection, any MSR access to the x2APIC MSR range (MSR 0x800-0x8FF) by the guest is intercepted regardless of the MSR_PROT intercept and MSR protection bitmap. In this case, the interception behavior is the same as what would occur if the MSR bitmap indicated an interception of the corresponding MSR.

15.36.17 Side-Channel Protection

SEV-SNP provides optional protections against certain side channel attacks.

15.36.17.1 Branch Target Buffer Isolation

SNP-active guests may choose to enable the Branch Target Buffer Isolation mode through SEV_FEATURES bit 7 (BTBIsolation). The Branch Target Buffer (BTB) is an internal CPU structure that is used when predicting indirect branches, and SNP-active guests may choose to impose additional restrictions on it in order to help prevent certain types of speculative execution-based side channels.

When executing an SNP-active guest when BTB Isolation is enabled, CPU hardware will ensure that no code outside of that guest context is able to influence the BTB-based predictions performed by hardware within the guest. Hardware tracks the source of prediction information in the BTB and may flush BTB contents when required to maintain this isolation.

In hardware that supports BTB Isolation, new BTB prediction information is never written if SPEC_CTRL[IBRS] is enabled in the current context. Therefore, it is recommended that non-guest software that executes temporarily (e.g., hypervisor exit handling code) run with SPEC_CTRL[IBRS]

[AMD Public Use]

set to 1. This ensures that indirect branch information from that context is not stored in the BTB and may avoid the need for a BTB flush when guest execution is resumed.

15.36.17.2 Indirect Branch Prediction Barrier on Entry

SNP-active guests may choose to enable Indirect Branch Prediction Barrier (IBPB) on Entry through SEV_FEATURES bit 21 (IbpbOnEntry). Support for IBPB on Entry is indicated by CPUID Fn8000_001F[IbpbOnEntry], bit 31. When entering a guest context with IbpbOnEntry enabled, CPU hardware writes PRED_CMD[IBPB]=1 before executing guest instructions.

15.36.17.3 Instruction Based Sampling

SEV-ES and SNP-active guests may choose to disallow the use of Instruction Based Sampling (IBS) by the hypervisor in order to limit the information that may be gathered about their execution. Guests may enable this restriction through SEV_FEATURES bit 6 (PreventHostIBS). On a VMRUN to a guest that has enabled this protection, IBS Fetch Sampling, IBS Execution Sampling, and IBS Memory Profiler sampling must all be disabled: IbsFetchCtl[IbsFetchEn] = 0 or IbsFetchCtl2[IbsFetchDis] = 1; IbsOpCtl[IbsOpEn] = 0 or IbsOpCtl2[IbsOpDis] = 1; and IbsMemCtl[IbsMemEn] = 0 or IbsMemCtl2[IbsMemDis] = 1. Not all of these IBS features and enable/disable bits may be supported by the processor. If any of these IBS features are enabled, then the VMRUN will fail with a VMEXIT_INVALID error code.

15.36.17.4 VMSA Register Protection

SNP-active guests may choose to enable the VMSA Register Protection feature through SEV_FEATURES bit 14 (VmsaRegProt). When an Automatic Exit occurs and this feature is enabled, CPU hardware will obfuscate the values of certain registers before they are written to the VMSA. The hardware will then deobfuscate these values on a subsequent VMRUN. This obfuscation may help prevent certain types of side channel attacks on the encrypted VMSA ciphertext.

The obfuscation is performed with a bit-wise XOR operation between the register value and an 8B nonce. The nonce value is stored in the VMSA and is updated in a pseudo-random manner by the CPU hardware on every Automatic Exit. When initializing a new VMSA, it is recommended that the nonce is set to a random value.

The specific VMSA fields which are obfuscated when this feature is enabled may vary by implementation. More information about this may be found in the AMD-SP SEV-SNP ABI specification.

15.36.17.5 SMT Protection

The SMT Protection feature allows SEV-SNP VMs to require the sibling thread to be idle while the VM is being run. This ensures that malicious code is not executed by the sibling thread and therefore provides mitigation for potential side channel attacks related to shared core resources. The hypervisor must execute the HLT instruction or request an I/O C-state on the sibling thread prior to executing the VMRUN instruction to run an SEV-SNP vCPU with the SMT Protection feature enabled.

[AMD Public Use]

SNP-active guests may choose to enable the SMT Protection feature through SEV_FEATURES bit 15 (SmtProtection). Support for the SMT Protection feature is indicated by CPUID Fn8000_001F_EAX[25](SmtProtection)=1.

When the hypervisor executes VMRUN to run an SNP-active guest with SMT Protection enabled, the processor checks if the sibling thread is in an idle state and in host mode. If not, the VMRUN fails with a VMEXIT_IDLE_REQUIRED error code.

When a thread is in the idle state and the sibling thread is in guest mode with SMT Protection active, the thread in the idle state does not immediately exit the idle state upon receiving a wake up event, such as an interrupt. Instead, the processor writes the APIC ICR register with the IDLE_WAKEUP_ICR MSR value and remains in the idle state until the sibling thread enters host mode. The write is done once and only if x2APIC mode is enabled. It is recommended that hypervisor software program the IDLE_WAKEUP_ICR value to send an IPI to the sibling thread to force it to enter host mode.

The IDLE_WAKEUP_ICR MSR (C001_0137h) has the same layout and access properties as the x2APIC Interrupt Command Register (ICR) MSR (see Section 16.13).

15.36.17.6 Enhanced SMT Protection

The Enhanced SMT Protection (ESMTP) feature extends SMT Protection functionality by allowing the sibling thread(s) to either be in an idle state in host mode or execute a vCPU from the same VM, which allows more efficient use of hardware multi-threading resources. Additionally, ESMTP adds the ability for guests to define a relationship between vCPUs by specifying which vCPUs may run on the same CPU core.

Support for ESMTP is indicated by CPUID Fn8000_0025_EDX[EnhSmtProtection] (bit 1) = 1. SEV_FEATURES bit 17 (EnhSmtProtection) may be used to enable the ESMTP feature for SNP-active guests. Enhanced SMT Protection and SMT Protection (SEV_FEATURES bit 15) are mutually exclusive. VMRUN fails with VMEXIT_INVALID if both features are enabled.

The VCPU_ID field (VMSA offset 8A0h, 4 bytes) identifies the vCPU, and the VCPU_SIBLING_MASK field (VMSA offset 8A4h, 4 bytes) defines a mask that determines VCPU_ID values that are allowed to run on the same core simultaneously. VCPU_ID and VCPU_SIBLING_MASK fields should be set during guest creation. An SEV-SNP guest may read the current value of VCPU_ID (returned in EAX) and VCPU_SIBLING_MASK (returned in EDX) through the VCPU_ID MSR (C001_013Ah). This MSR is read-only. An attempt to write the VCPU_ID MSR will result in a #GP(0) exception. The VCPU_ID MSR read will return zero when ESMTP is not enabled.

The ESMTP_TIMEOUT_CTL field (VMCB offset 148h, 8 bytes) specifies the maximum wait time expressed in the number of clocks at P0 frequency for thread synchronization during VMRUN when ESMTP is enabled. ESMTP timeout is disabled if ESMTP_TIMEOUT_CTL is equal to 0.

[AMD Public Use]

VMRUN ESMTP Behavior

When the hypervisor executes a VMRUN to an SNP-Active vCPU with ESMTP enabled, the processor waits until all sibling threads have done a VMRUN to a legal ESMTP sibling, as defined below, or are in an idle state in host mode before entering guest mode. VMRUN to an ESMTP-enabled vCPU fails if any of the following events occurs:

- Any sibling thread does a VMRUN to an illegal ESMTP sibling
- A physical INTR, NMI, SMI, or INIT
- The timeout specified by ESMTP_TIMEOUT_CTL

If a physical interrupt occurs, the VMRUN instruction finishes with the exit code for the physical interrupt (e.g., VMEXIT_INTR). Physical INTR, NMI, SMI, and INIT interrupts detected during VMRUN are unconditionally intercepted. If an internal event is detected during VMRUN, the VMRUN instruction exits with VMEXIT_RETRY exit code (AE, exit code -7). The hypervisor should re-execute VMRUN if it exits with VMEXIT_RETRY.

If the ESMTP timer expires, the VMRUN instruction exits with VMEXIT_ESMTP_TIMEOUT exit code (AE, exit code -6). A legal ESMTP sibling is defined as follows:

- SEV-SNP VM with ESMTP enabled
- ASID values match
- VCPU_SIBLING_MASK values match
- VCPU_ID & ~VCPU_SIBLING_MASK values match

An illegal ESMTP sibling is an SEV-SNP vCPU with ESMTP enabled that does not meet all of the above conditions. If any of the above checks fails, VMRUN exits with VMEXIT_ILLSIB exit code (AE, exit code -5).

Once all sibling threads are either in an idle state in host mode (HLT, MWAIT, MWAITX at CPL0, or an I/O C-state) or have executed VMRUN to a legal ESMTP sibling, any pending VMRUN operations on that core are allowed to finish and enter guest mode.

VMEXIT ESMTP Behavior

When an automatic exit event occurs while a vCPU with ESMTP enabled is running, the VMEXIT cannot complete until all sibling threads not in an idle state in host mode stop executing guest code. During the VMEXIT flow, the processor checks the status of the sibling thread(s). If a sibling thread is executing guest code, the processor writes the APIC ICR register with the IDLE_WAKEUP_ICR MSR value and waits until the sibling thread enters host mode. The IPI is not sent if all other sibling threads are in an idle state or are attempting to complete VMEXIT.

If the VMEXIT occurs during the VMRUN instruction, the processor writes the APIC ICR register with the IDLE_WAKEUP_ICR MSR value if any sibling thread with ESMTP enabled is attempting to complete VMRUN or is executing guest code, and waits until the sibling thread enters host mode.

[AMD Public Use]

15.36.18 Secure TSC

SNP-active guests may choose to enable the Secure TSC feature through SEV_FEATURES bit 9 (SecureTscEn). When enabled, Secure TSC changes the guest view of the Time Stamp Counter when read by the guest via either the TSC MSR, RDTSC, or RDTSCP instructions. The TSC value is first scaled with the GUEST_TSC_SCALE value from the VMSA and then is added to the VMSA GUEST_TSC_OFFSET value. The P0 frequency, TSC_RATIO (C001_0104h) and TSC_OFFSET (VMCB offset 50h) values are not used in the calculation.

The GUEST_TSC_SCALE is an 8.32 fixed point binary number which is composed of 8 bits of integer and 32 bits of fraction. The AMD-SP SEV-SNP ABI specification provides additional information about the Secure TSC feature and initialization of GUEST_TSC_SCALE.

Guests that run with Secure TSC enabled may read the GUEST_TSC_FREQ MSR (C001_0134h) which returns the effective frequency in MHz of the guest view of TSC. This MSR is read-only and attempting to write the MSR or read it when outside of a guest with Secure TSC enabled causes a #GP(0) exception.

Guests that run with Secure TSC enabled are not expected to perform writes to the TSC MSR (10h). If such a write occurs, subsequent TSC values read are undefined.

15.36.19 SEV-SNP Instruction Virtualization

The hypervisor uses RMPUPDATE and PSMASH instructions to modify the RMP when SEV-SNP is enabled. In a nested virtualization use case, when the hypervisor is running as a guest, these instructions should be replaced with WRMSR VIRT_RMPUPDATE MSR (C001_F001h) and WRMSR VIRT_PSMASH MSR (C001_F002h), respectively. The VIRT_RMPUPDATE MSR, VIRT_PSMASH MSR and CPUID Fn8000_001F_EAX[NestedVirtSnpMsr] (bit 29), which reports VIRT_RMPUPDATE and VIRT_PSMASH MSR support, are not implemented in the processor and are expected to be emulated by the top-level hypervisor.

VIRT_RMPUPDATE MSR input convention:

RAX: 4KB aligned GPA

RDX: New RMP entry, bytes 7:0

R8: New RMP entry, bytes 15:8

VIRT_RMPUPDATE MSR output convention:

RAX: RMPUPDATE return code

VIRT_PSMASH MSR input convention:

RAX: 2MB aligned GPA

VIRT_PSMASH MSR output convention:

RAX: PSMASH return code

[AMD Public Use]

The hypervisor uses the RMPREAD instruction to read an RMP entry when SEV-SNP is enabled. In a nested virtualization use case, RMPREAD should be replaced with WRMSR VIRT_RMPREAD MSR (C001_F003h) if CPUID Fn8000_001F_EAX[NestedVirtSnpMsr] = 1 and CPUID Fn8000_001F_EAX[RMPREAD] = 1.

VIRT_RMPREAD input convention:

RAX: 4KB aligned GPA

VIRT_RMPREAD output convention:

RAX: RMPREAD return code

RDX: RMP entry, bytes 7:0

R8: RMP entry, bytes 15:8

15.36.20 Allowed SEV Features

A hypervisor may enforce which SEV features may be enabled in an SEV-SNP VM with Allowed SEV Features. Allowed SEV Features support is indicated by CPUID Fn8000_001F_EAX[AllowedSevFeatures] (bit 27) = 1. The Allowed SEV Features Mask is enabled when bit 63 at offset 138h in the VMCB is set to 1. The hypervisor may allow a specific feature by setting the corresponding ALLOWED_SEV_FEATURES_MASK bit to 1 and may disallow a specific feature by setting the corresponding bit to 0, where ALLOWED_SEV_FEATURES_MASK bits 61:0 correspond to VMSA SEV_FEATURES bits 61:0.

Some SEV features can only be used if the Allowed SEV Features Mask is enabled, and the mask is configured to permit the corresponding feature. If the Allowed SEV Features Mask is not enabled, these features are not available (see SEV_FEATURES in Appendix B, Table B-4).

When the Allowed SEV Features Mask is enabled, the VMRUN instruction checks that all SEV_FEATURES bits which are set in VMSA offset 3B0h are also set in ALLOWED_SEV_FEATURES_MASK. If not, VMRUN fails with a VMEXIT_INVALID error code. SEV_FEATURES is saved to VMCB field GUEST_SEV_FEATURES at offset 140h on a #VMEXIT.

15.36.21 Secure AVIC

Secure AVIC feature provides hardware acceleration of performance sensitive APIC accesses and support for managing guest-owned APIC state for SEV-SNP guests. Secure AVIC additionally offers security protections designed to help protect against malicious injection attacks by limiting events which may be injected into an SEV-SNP guest.

15.36.21.1 Enabling Secure AVIC

Hardware support for Secure AVIC is indicated by CPUID Fn8000_001F_EAX[SecureAvic] (bit 26) = 1. Secure AVIC mode is selected when SecureAvic (bit 16 in SEV_FEATURES) is set. While in Secure AVIC mode, the guest may set SecureAvicEn (bit 0 in Secure AVIC Control MSR) to enable the guest APIC backing page and full Secure AVIC capabilities. Enablement of this feature is

[AMD Public Use]

additionally dependent on the value of ALLOWED_SEV_FEATURES_MASK in VMCB (see Section 15.36.20, “Allowed SEV Features,” on page 646)

The Secure AVIC feature only supports the x2APIC MSR interface.

15.36.21.2 VMRUN and #VMEXIT

Secure AVIC mode is mutually exclusive with Restricted Injection, Alternate Injection, and hypervisor controlled AVIC modes. If the SecureAvic bit is set to 1, and the AVIC Enable bit in the VMCB is set to 1 or the RestrictedInjection or AlternateInjection bits in SEV_FEATURES are set to 1, VMRUN will fail with #VMEXIT(VMEXIT_INVALID).

The VMRUN instruction loads the Secure AVIC Control MSR from VMSA offset 320h. If SecureAvicEn is set to 1 and GuestApicBackingPagePtr is not a valid guest physical address, VMRUN will fail with #VMEXIT(VMEXIT_INVALID).

The interrupt control information loaded from the VMCB and VMSA for Secure AVIC mode operation is the same as the information loaded in Alternate Injection mode. When the SecureAvicEn bit is set to 1, virtual INTR and virtual VNMI information loaded from the VMSA is ignored, and instead determined as follows:

- The IRR field in the Guest APIC Backing page is updated with IRRs hypervisor wishes to inject. If the UpdateIRR bit is set, the guest-controlled AllowedIRR mask is logically ANDed with the host-controlled RequestedIRR and then is logically ORed into the IRR field in the Guest APIC Backing page. The 256-bit AllowedIRR vector is specified in the guest backing page with eight 32-bit registers, where Guest Allowed IRR bit n is at bit position (n modulo 32) at offset (204h + n / 32). RequestedIRR is specified at VMCB offset 150h.
- The Guest AVIC Backing page is evaluated and V_IRQ, V_INTR_PRIO, V_IGN_TPR, and V_INTR_VECTOR fields are updated in hardware.
- The V_NMI bit is set in hardware if the guest-controlled V_NMI bit in VMSA is set, both the guest-controlled AllowedNMI and the host-controlled V_NMI in VMCB are set, or if NmiReq (offset 278h, bit 0) in the Guest APIC Backing page is set.

When SecureAvicEn is set, NmiReq is cleared in the Guest APIC Backing page, and the UpdateIRR and RequestedIRR fields are cleared to 0 in the VMCB by the VMRUN instruction.

On processors that indicate support for Secure AVIC, if the BUSY bit in the VIntrCtrl VMSA field is set to 1, VMRUN to an SEV-ES or SEV-SNP guest will fail with a VMEXIT_BUSY error code.

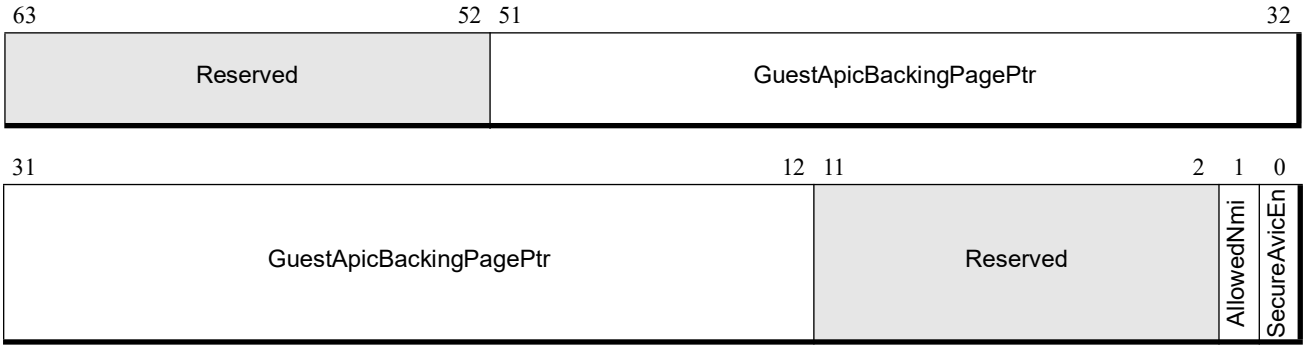
On a VMEXIT in Secure AVIC mode, the guest virtual interrupt state (VIntrCtrl bits 15:0) and the Secure AVIC Control MSR are saved in the VMSA. Additionally, EXITINTINFO is written to the VMSA EVENTINJ field for software interrupts and exceptions with vector 3, 4 or 29. This causes the processor to re-inject the interrupted event automatically on the next VMRUN.

In Secure AVIC mode hardware treats physical INTR, NMI, INIT, and #MC events as intercepted regardless of the corresponding intercept bit values in the VMCB.

[AMD Public Use]

15.36.21.3 Secure AVIC Control MSR

The Secure AVIC Control MSR is used to configure Secure AVIC feature in a guest. The Secure AVIC Control MSR (C001_0138) fields are defined as follows:



Bits	Mnemonic	Description	Access type
63:52	Reserved		MBZ
51:12	GuestApicBackingPagePtr	Guest APIC Backing Page Pointer	R/W
11:2	Reserved		MBZ
1	AllowedNmi	Host injection of NMI is allowed	R/W
0	SecureAvicEn	Secure Avic Enable	R/W

Figure 15-32. Secure AVIC Control MSR

The Secure AVIC Control MSR can only be accessed in Secure AVIC mode. An attempt to access it when not in Secure AVIC mode will result in a #GP(0) exception.

If the SecureAvicEn is set to 1 and Guest APIC Backing Page Pointer is not a valid guest physical address when this MSR is written, a #GP(0) exception is generated.

15.36.21.4 Guest APIC Backing Page

Guest accesses to local APIC registers are redirected to the guest APIC backing page in system memory. The GPA of the guest backing page is saved in the VMSA and may be controlled by the guest with the Secure AVIC Control MSR.

It is required that the guest APIC backing page for a vCPU is pinned in system memory between VMRUN and VMEXIT because some AVIC hardware acceleration sequences may not be restartable when secure AVIC is enabled. If an access to the guest's own backing page by AVIC hardware results in a nested page fault, EXITINFO1 bit 63 (Not Restartable) is set (this is an Automatic Exit) and the BUSY bit in the VMSA is set. If the guest APIC backing page is not validated (Validated bit in the RMP entry is 0), a #VC with error code NOT_RESTARTABLE (0x406) is generated. Additionally, if ReflectVC is enabled, the BUSY bit in the VMSA is set.

For security, the guest should ensure that the Guest Backing Page Pointer maps to an encrypted guest page.

[AMD Public Use]

15.36.21.5 Guest APIC Accesses

Guest APIC accesses are handled by Secure AVIC hardware as follows:

- Allow: The backing page access is performed.
- Fault: The backing page is not accessed and #VC is generated (NAE).
- Trap: The backing page access is performed and #VC is generated (NAE).

APIC register access behavior is the same for Secure AVIC and x2AVIC except for Interrupt Command Register (ICR) accesses (see Table 15-22).

When a VMEXIT_AVIC_INCOMPLETE_IPI or VMEXIT_AVIC_NOACCEL is generated and ReflectVC is enabled, the BUSY bit in the VMSA is set.

x2APIC MSR intercepts in the MSRPM are ignored in Secure AVIC mode. x2APIC MSR accesses are not intercepted when Secure AVIC is enabled and are always intercepted when Secure AVIC is disabled.

15.36.21.6 ICR, TPR and EOI Accesses

Secure AVIC hardware accelerates Self IPIs, specifically ICR MSR (830h) write with Destination Shorthand (DSH) field equal to self (01b) and SELF_IPI MSR (83Fh) write. It updates the IRR in the backing page, evaluates the new IRR, injects VINTR if allowed by interrupt masking and priority, and continues with guest code execution. ICR writes with DSH field equal to all including self (10b), all excluding self (11b), or target will trap with a non-automatic exit and an AVIC_INCOMPLETE_IPI exit code with Reason ID Unaccelerated IPI (5). In Secure AVIC mode, the Physical APIC ID and Logical APIC ID tables are not used.

Secure AVIC hardware additionally accelerates accesses to TPR and EOI APIC registers as described in APM Volume 2. Secure AVIC acceleration of these registers is identical to the legacy AVIC acceleration.

15.36.22 Segmented RMP

The segmented RMP feature provides a way to allocate non-contiguous RMP memory for a more efficient RMP layout and reduces RMP access latency in NUMA systems.

An RMP segment corresponds to a range of system physical addresses and an RMP that covers some or all addresses in that segment. RMP segment size is programmable, which provides flexibility for various system configurations. The RMP Segment Table is used to specify the RMP base, and the size of memory covered by RMP for each segment.

15.36.22.1 Determining Support for Segmented RMP

Hardware support for Segmented RMP is indicated by CPUID Fn8000_001F_EAX[SegmentedRmp] (bit 23) = 1. When Segmented RMP is supported, CPUID Fn8000_0025_EAX and CPUID Fn8000_0025_EBX provide additional Segmented RMP information.

[AMD Public Use]

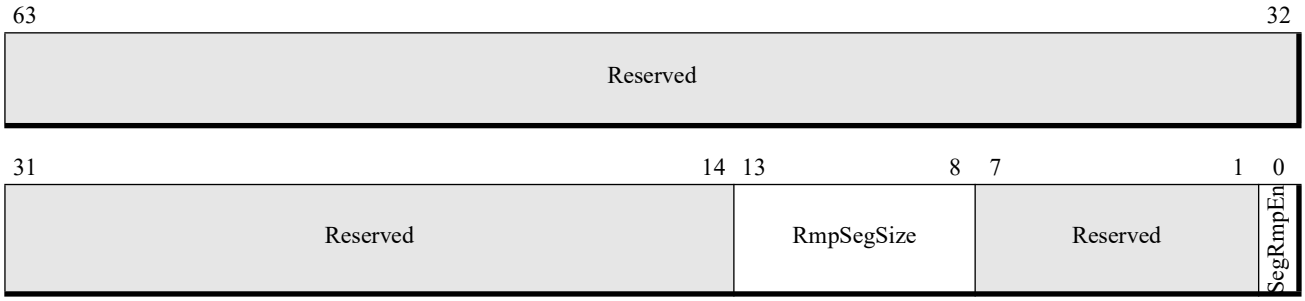
The segment size refers to the amount of system physical addresses mapped by one entry in the RMP Segment Table. (See Section 15.36.22.3, “RMP Segment Table,” on page 651.) The minimum and maximum RMP segment sizes are calculated from CPUID Fn8000_0025_EAX[MinRmpSegSize] (bits 5:0) and CPUID Fn8000_0025_EAX[MaxRmpSegSize] (bits 11:6), as 2^(MinRmpSegSize) and 2^(MaxRmpSegSize) Mbytes, respectively.

CPUID Fn8000_0025_EBX[NumCachedSegments] (bits 9:0) indicates the number of RMP segment definitions that are cached by the hardware. For best performance, the number of RMP segments should be less than or equal to the NumCachedSegments.

CPUID Fn8000_0025_EBX[NumSegReduction] (bit 10) indicates that the number of RMP segments defined by the RMP Segment Table is reduced. When NumSegReduction is equal to 0, up to 512 RMP segments can be defined. When NumSegReduction is equal to 1, up to NumCachedSegments segments can be defined.

15.36.22.2 Enabling Segmented RMP

The Segmented RMP Configuration MSR (C001_0136) is used to configure Segmented RMP. The Segmented RMP Configuration MSR fields are defined as follows:



Bits	Mnemonic	Description	Access type
63:14	Reserved		MBZ
13:8	RmpSegSize	RMP Segment Size	R/W
7:1	Reserved		MBZ
0	SegRmpEn	Segmented RMP Enable	R/W

Figure 15-33. Segmented RMP Configuration Register

This register may be written when SecureNestedPagingEn in SYSCFG MSR is 0. An attempt to write it when SecureNestedPagingEn is 1 will result in a #GP(0) exception. When SecureNestedPagingEn is 1, this MSR is read-only.

RmpSegSize is used to determine RMP segment size, which is equal to 2^(RmpSegSize). When SegRmpEn is 1, RmpSegSize must be between MinRmpSegSize and MaxRmpSegSize, inclusive. An attempt to write RmpSegSize to a value not within the allowed range will result in a #GP(0) exception.

[AMD Public Use]

When Segmented RMP is enabled, the RMP_BASE MSR points to a 1 MB aligned memory region. The first 16 KB of this region is used for processor bookkeeping. The next 4 KB of this region contains the RMP Segment Table. This table must be populated before SecureNestedPagingEn is set to 1. After the AMD-SP has initialized the system for SNP, the RMP Segment Table can no longer be modified by software.

The RMP_BASE alignment is checked when SecureNestedPagingEn is set to 1. If SegRmpEn is 1 and the value of RMP_BASE is not 1MB aligned, a #GP(0) is generated.

15.36.22.3 RMP Segment Table

The RMP Segment Table (RST) specifies the RMP, and the amount of system memory covered by the RMP for each segment.

Each RST entry has the following format:

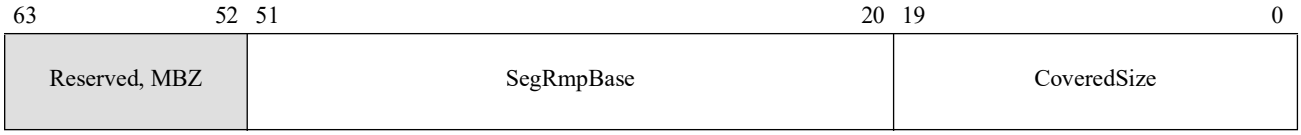


Figure 15-34. RMP Segment Table Entry

The SegRmpBase specifies the pointer to the RMP for the RMP covered memory in this segment.

The CoveredSize filed indicates segment size covered by the RMP. Segment size covered by the RMP is expressed in Gbytes. If CoveredSize is equal to 0, then memory that corresponds to this segment is not covered by the RMP. If the covered segment size is greater than the RMP segment size, the entire segment is covered by the RMP. The size of the RMP for the segment is equal to 4MB × CoveredSize.

When NumSegReduction is equal to 0, if during the table walk the reserved bits in the corresponding RST entry are set, a #PF or VMEXIT_NPF reserved bit error is generated.

When NumSegReduction is equal to 1 and SecureNestedPagingEn changes from 0 to 1, a #GP(0) is generated if the RMPs for defined RST entries do not reside in valid physical address space.

When Segmented RMP is enabled, a page in system memory has an RMP entry if:

1. There is an RST entry for the page address.
2. CoveredSize in the RST entry is not equal to 0.
3. The page address is covered by the RMP defined by the SegRmpBase and CoveredSize.

If a page does not have an RMP entry, it is a Hypervisor-Owned page. See GET_RMP_ENTRY_ADDR function in APM Volume 3 for more details on how RMP entry address is determined when Segmented RMP is enabled.

[AMD Public Use]

15.36.23 Guest Intercept Control

The Guest Intercept Control feature allows a vCPU executing at a higher privilege VMPL to intercept and emulate specific events for a vCPU running at a lower privilege VMPL.

Guest Intercept Control support is indicated by CPUID Fn8000_001F[GuestInterceptCtl] (bit 22) = 1. SEV_FEATURES[GuestInterceptCtl], bit 13, may be used to enable the Guest Intercept Control feature in an SEV-SNP guest.

Guest-controlled intercepts for instructions, exceptions and interrupt events are defined by eight 4-byte intercept vectors in the VMSA, starting at offset 900h. These fields have the same layout as the host-controlled intercept vectors in the VMCB, starting at offset 0h. When the GuestInterceptCtl bit in SEV_FEATURES is set, an instruction, exception or interrupt event will be intecepted if either the corresponding host-controlled intercept or guest-controlled intercept is set. Note that the CPUID instruction and all IO port accesses are unconditionally intercepted in SEV-ES and SEV-SNP guests.

Guest-controlled MSR intercepts are defined for specific MSRs in four 8-byte intercept vectors in the VMSA, starting at offset 920h. (See Table B-2 on page 786.) For each MSR, two bits are defined for read and write access, where the lsb of the two bits is the read MSR intercept and the msb is the write MSR intercept. (See Appendix B.1, “Guest MSR Intercepts,” on page 797.) A RDMSR or WRMSR instruction will be intercepted if the corresponding intercept bit is set for the given MSR. Except where explicitly stated otherwise, if the MSR does not have defined intercept bits in these VMSA vectors, a RDMSR or WRMSR instruction will be unconditionally intercepted. Guest-controlled MSR intercepts are independent of host-controlled MSR intercepts defined in Section 15.11, “MSR Intercepts,” on page 540. A RDMSR or WRMSR instruction will be intercepted as indicated by either the corresponding host-controlled or guest-controlled intercept. Any undefined guest-controlled MSR intercept bits should be initialized to 1 during VM creation.

x2APIC MSRs are not subject to the guest intercept control feature in Secure AVIC mode.

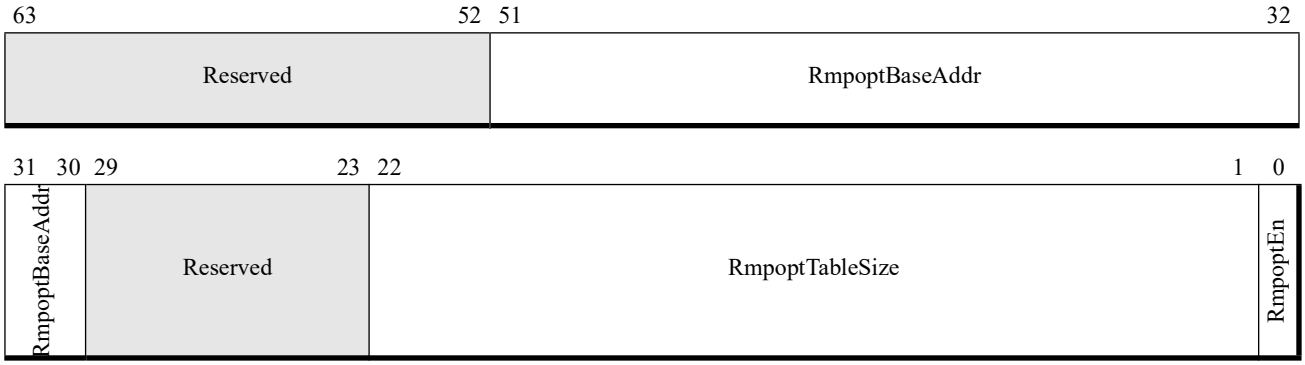
15.36.24 RMPOPT

The RMPOPT feature minimizes the performance overhead of RMP checks for the hypervisor and non-SEV-SNP guests. In the SEV-SNP architecture, the hypervisor and non-SEV-SNP guests are subject to RMP checks on writes to protect against corruption of SEV-SNP guest memory. The RMPOPT architecture enables optimizations by which these checks can be skipped if large 1GB regions of memory are known to not contain any SEV-SNP guest memory.

[AMD Public Use]

15.36.24.1 Determining Support and Enabling RMPOPT

Support for RMPOPT is indicated by CPUID Fn8000_0025_EDX[RmpOpt] (bit 0) = 1. The RMPOPT feature is enabled through the per-core RMPOPT_BASE MSR (C001_0139h).



Bits	Mnemonic	Description	Access type
63:52	Reserved	MBZ	R/W
51:30	RmpoptBaseAddr	Start address of the memory covered by the CPUs RMPOPT table expressed in Gbytes	R/W
29:23	Reserved	MBZ	R/W
22:1	RmpoptTableSize	Maximum address space size supported by the processor for RMPOPT expressed in Gbytes	RO
0	RmpoptEn	RMPOPT feature enable	R/W

Figure 15-35. RMPOPT_BASE MSR

The RmpoptEn bit may only be written to 1 if SYSCFG[SNPE] and SEGMENTED_RMP_CFG[SegRmpEn] are 1. The RmpoptEn bit cannot be cleared to 0 when SYSCFG[SNPE] is 1. The RmpoptBaseAddr field is read-only when RmpoptEn is 1. An attempt to write the RMPOPT Base MSR when requirements are not met will result in a #GP(0) exception.

15.36.24.2 RMPOPT Table

Each core has an RMPOPT Table which indicates if specific 1 Gbyte system memory regions are entirely hypervisor-owned. Each 1Gbyte of the system address space starting at RmpoptBaseAddr is represented by a single bit, indicating if the entire region is hypervisor-owned.

Software may choose to optimize different regions of the system address space for different cores. The combination of RmpoptBaseAddr and RmpoptTableSize describes which portion of the system address space may be optimized for each core.

When performing memory accesses other than to private memory by an SEV-SNP guest, the processor may consult the RMPOPT table to potentially skip RMP access. If the address of the memory access resides within an address region covered by the local RMPOPT table, and the table indicates the entire region is hypervisor-owned, then the RMP check may be skipped, thus improving performance. If the address of the memory access does not reside within the memory covered by the local RMPOPT table,

[AMD Public Use]

or if the table indicates the region may not be entirely hypervisor-owned, RMP checks must be performed on write requests.

The RMPOPT table is managed by a combination of software and hardware. Software uses the RMPOPT instruction to set bits in the table for regions of memory that are entirely hypervisor-owned (see RMPOPT instruction description in Volume 3 for more details). Hardware automatically clears bits in the RMPOPT table when RMP contents are changed during the RMPUPDATE instruction.

15.37 Speculation Control Virtualization

This section describes SVM support for speculation control virtualization.

15.37.1 SPEC_CTRL Virtualization

A hypervisor may impose speculation controls on guest execution or a guest may impose its own speculation controls. Therefore, the processor implements the host and guest SPEC_CTRL register. Support for SPEC_CTRL virtualization is indicated by CPUID Fn8000_000A_EDX[SpecCtrl] (bit 20) = 1.

When in host mode, the host SPEC_CTRL value is in effect and writes update the host SPEC_CTRL register. On a VMRUN, the processor loads the guest SPEC_CTRL register value from the VMCB or VMSA. For most guests, processor behavior is controlled by the logical OR of the two registers. When the guest writes SPEC_CTRL, the guest register is updated. On a VMEXIT, the guest value is saved into the VMCB or VMSA and only the host SPEC_CTRL is in effect.

For SNP-active guests with BTB isolation enabled (see Section 15.36.17, “Side-Channel Protection,” on page 641), the host IBRS value, controlled by SPEC_CTRL[IBRS] or EFER[AIBRSE], does not affect the guest IBRS.

15.37.2 ERAPS Virtualization

A hypervisor may control the return address predictor configuration in the guest if CPUID Fn8000_0021_EAX[ERAPS] (bit 24) = 1.

The ALLOW_LARGER_RAP bit in the VMCB (offset 58h, bit 40) controls the return address predictor size in the guest. If set to 0, the return address predictor size in the guest is 32. If ALLOW_LARGER_RAP is set to 1, the return address predictor size in the guest has the default value which is equal to CPUID Fn8000_0021_EBX[RapSize]. RapSize is always equal or greater than 32.

The CLEAR_RAP bit in the VMCB (offset 58h, bit 41) controls return address predictor clearing. If set to 1 on VMRUN, the processor clears the return address predictor before entering guest mode.

15.38 Instruction-Based Sampling Virtualization

Hardware support for Instruction-Based Sampling (IBS) virtualization is reported by CPUID Fn8000_001F_EAX[IbsVirtGuestCtl] (bit 19) = 1 for SEV-ES and SEV-SNP guests, and by CPUID

[AMD Public Use]

Fn8000_000A_EDX[IbsVirt] (bit 26) = 1 for all other guests. IBS virtualization is enabled when bit 12 in SEV_FEATURES in VMSA is set to 1 for SEV-ES and SEV-SNP guests, and when bit 2 at offset B8h in the VMCB is set to 1 for all other guests.

When a VMRUN is executed to an SEV-ES or SEV-SNP guest with IBS virtualization enabled, Fetch IBS and Execution IBS must be disabled: IbsFetchCtl[IbsFetchEn] = 0 or IbsFetchCtl2[IbsFetchDis] = 1, and IbsOpCtl[IbsOpEn] = 0 or IbsOpCtl2[IbsOpDis] = 1. If either of these is enabled, the VMRUN will fail with a VMEXIT_INVALID error code. The same requirement, extended to the IBS Memory Profiler, applies when PreventHostIBS is set, whether IBS virtualization is enabled or not; see Section 15.36.17, “Side-Channel Protection,” on page 641. For all other guests, Fetch IBS and Execution IBS should be disabled in the same way if IBS virtualization is enabled, to prevent host IBS interrupts from leaking across a world switch.

When IBS virtualization is enabled, the following fields in the VMCB and VMSA state save area hold the guest fetch and op IBS register values:

- IBS_FETCH_CTL — IbsFetchCtl MSR
- IBS_FETCH_LINADDR — IbsFetchLinAd MSR
- IBS_OP_CTL — IbsOpCtl MSR
- IBS_OP_RIP — IbsOpRip MSR
- IBS_OP_DATA — IbsOpData1 MSR
- IBS_OP_DATA2 — IbsOpData2 MSR
- IBS_OP_DATA3 — IbsOpData3 MSR
- IBS_DC_LINADDR — IbsDcLinAd MSR
- IBS_BRTGT_RIP — IbsBrTarget MSR
- IBS_FETCH_EXTD_CTL — IbsFetchExtdCtl MSR
- IBS_FETCH_CTL2 — IbsFetchCtl2 MSR
- IBS_OP_CTL2 — IbsOpCtl2 MSR
- IBS_BUFFER_BASE — IbsBufferBase MSR
- IBS_BUFFER_SIZE — IbsBufferSize MSR
- IBS_BUFFER_TAIL — IbsBufferTail MSR

With IBS virtualization enabled, a guest is not able to read a physical address from either IbsFetchPhysAd MSR or IbsDcPhysAd MSR. Both registers return zero when read by the guest and the associated valid bits IbsFetchCtl[IbsPhyAddrValid] and IbsOpData3[IbsDcPhyAddrValid] respectively, return zero. This also applies to the Fetch and Execution IBS samples written by IBS Buffering. Writes to IbsFetchPhysAd MSR and IbsDcPhysAd MSR within the guest are ignored.

IBS virtualization requires the use of AVIC (see Section 15.29, “Advanced Virtual Interrupt Controller,” on page 588) or NMI virtualization (see Section 15.21.10, “NMI Virtualization,” on page 559) for delivery of a virtualized interrupt from the IBS hardware in the guest. Without virtualized interrupt delivery, an IBS interrupt occurring in the guest will not be delivered to either the

[AMD Public Use]

guest or the hypervisor. When AVIC is enabled, the IBS LVT entry (Extended Interrupt 0 LVT) message type should be programmed to INTR or NMI.

15.39 Performance Monitoring Counter Virtualization

Hardware support for Performance Monitoring Counter (PMC) virtualization is reported by CPUID Fn8000_001F_EAX[PmcVirtGuestCtl] (bit 20) = 1 for SEV-ES and SEV-SNP guests and by CPUID Fn8000_000A_EDX[PmcVirt] (bit 8) = 1 for all other guests. PMC virtualization is enabled when bit 11 in SEV_FEATURES in VMSA is set to 1 for SEV-ES and SEV-SNP guests, and when bit 3 at offset B8h in the VMCB is set to 1 for all other guests. Enablement of this feature is additionally dependent on the value of ALLOWED_SEV_FEATURES_MASK in VMCB (see Section 15.36.20, “Allowed SEV Features,” on page 646).

When PMC virtualization is enabled, the following fields are allocated in the VMCB and VMSA state save area hold the guest PMC register values:

- PERF_CTL*n* — PerfEvtSel*n* MSR, n = 0 to 5
- PERF_CTR*n* — PerfCtr*n* MSR, n = 0 to 5
- INSTR_RETIRED_CTR — IRPerfCount MSR
- PERF_CTR_GLOBAL_STS — PerfCntrGlobalStatus MSR
- PERF_CNT_GLOBAL_CTL — PerfCntrGlobalCtl MSR

If PMC virtualization is not enabled by the hypervisor in the VMCB and is enabled by an SEV-ES or SEV-SNP guest in VMSA, each PERF_CTL*n* bit 22 (PMC enable) and PERF_CNT_GLOBAL_CTL bits 5:0 (global PMC enable) in the VMSA must be 0. If any of these bits are not 0, the VMRUN will fail with a VMEXIT_INVALID_PMC error code.

PMC virtualization requires the use of AVIC (see Section 15.29, “Advanced Virtual Interrupt Controller,” on page 588) or NMI virtualization (see Section 15.21.10, “NMI Virtualization,” on page 559) for delivery of a virtualized interrupt from the PMC hardware in the guest. Without virtualized interrupt delivery, a PMC interrupt occurring in the guest will not be delivered to either the guest or the hypervisor. When AVIC is enabled, the Performance Counter LVT entry message type should be programmed to INTR or NMI.

15.39.1 Guest PMC Event Filtering

Hardware support for Guest PMC Event Filtering is reported by CPUID Fn8000_000A_ECX[GuestPmcFiltering] (bit 3) = 1. Guest PMC Event Filtering is enabled when bit 5 at offset B8h in the VMCB is set to 1 and PMC Virtualization is enabled for the guest. When Guest PMC Event Filtering is enabled, enabled performance counter event-select register values are subject to additional checks. A core performance-event select register value X is allowed if the following is satisfied for any of the ALLOWED_PERF_CTLi values:

(X & ALLOWED_PERF_CTL_MASK) == (ALLOWED_PERF_CTLi & ALLOWED_PERF_CTL_MASK)

[AMD Public Use]

Event-select register values are checked when the guest writes the PerfEvtSeln MSR, the PerfCntrGlobalCtl MSR, and during VMRUN.

During VMRUN, in addition to existing PMC virtualization checks the processor checks if each PERF_CTLn guest value is allowed when PERF_CTLn[22] and PERF_CNT_GLOBAL_CTL[n] guest values are equal to 1. If any value fails this check, VMRUN fails with #VMEXIT(VMEXIT_INVALID_PMC) (exit code -4). When a guest writes to a PerfEvtSeln MSR, EAX[22] is 1 and MSR PerfCntrGlobalCtl[n] is 1, the processor checks that the new value (EDX:EAX) is allowed. Otherwise, a #VC exception is generated for SEV-ES and SEV-SNP guests, and VMEXIT_WRMSR is generated for all other guests.

When a guest writes to the PerfCntrGlobalCtl MSR, the processor checks if every counter being enabled contains an allowed value. If any counter value fails this check, a #VC exception is generated for SEV-ES and SEV-SNP guests, and VMEXIT_WRMSR is generated for all other guests.

15.40 Guest Alternate Address Translation

Guest Alternate Address Translation (GAAT) is a hypervisor-controlled feature that changes the way in which guest linear addresses are translated. Instead of normal paging, translation uses GAAT paging.

15.40.1 GAAT Paging

GAAT paging is a form of paging only available in guest mode that uses GAAT Pointer (GAATP, VMCB offset 128h) as the base address of the highest-level page-translation table instead of CR3 used for normal paging.

GAAT is active in long mode. GAATP has the same format as CR3 but does not contain a PCID. If CR4.PCIDE is 1, GAAT paging uses the PCID value in CR3[11:0].

If GAAT paging is enabled, translation of a linear address depends on the linear address and on the value of the GAAT prefix width (GAAT_PREFIX_WIDTH, VMCB offset 130h) as follows:

- If the GAAT prefix width is zero, every linear address is translated using GAAT paging.
- If the GAAT prefix width is not zero, a linear address is translated using GAAT paging if bit 63 of the address is 1; otherwise, it is translated using normal paging.

GAAT paging may specify that a translation of a linear address must be restarted. This occurs when a page table entry encountered during GAAT page walk has bit 11 set (Restart bit). When the Restart bit is set, translation of the linear address is restarted using normal paging, starting with CR3. The linear address is translated again, including parts that had been translated by GAAT. Permissions for the restarted translation are determined by normal page tables.

The page size of the restarted translation associated with a TLB entry is the smaller of the page size at the GAAT page table level at which the restart occurred and the page size of the restarted translation.

[AMD Public Use]

If the Restart bit is set in a GAAT page table entry, the processor sets the Accessed bit in that entry. If the entry is at the lowest level of the GAAT page-translation hierarchy, the processor does not set the Dirty bit.

15.40.2 GAAT CPUID

Support for GAAT is indicated by CPUID Fn8000_000A_ECX[GAAT] (bit 0) = 1. The maximum GAAT prefix width is reported by CPUID Fn8000_000A_EAX[MAX_GAAT_PREFIX_WIDTH] (bits 13:8).

15.40.3 GAAT Enablement

GAAT is enabled by setting bit 8 at offset 90h (GaatEn) in the VMCB. When VMRUN is executed and GaatEn is set in the VMCB, the following consistency checks are performed:

- Nested paging must be enabled.
- The host must be in 64-bit mode.
- SEV must be disabled.
- GAAT Pointer (VMCB offset 128h) MBZ bits must be zero.

If any of the above consistency checks fail, a #VMEXIT(VMEXIT_INVALID) is generated.

[AMD Public Use]

16 Advanced Programmable Interrupt Controller (APIC)

The Advanced Programmable Interrupt Controller (APIC) provides interrupt support on AMD64 architecture processors. The local APIC accepts interrupts from the system and delivers them to the local CPU core interrupt handler.

Support for APIC is indicated by CPUID Fn0000_0001_EDX[APIC] = 1. For information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 73.

The APIC block diagram is provided in Figure 16-1.

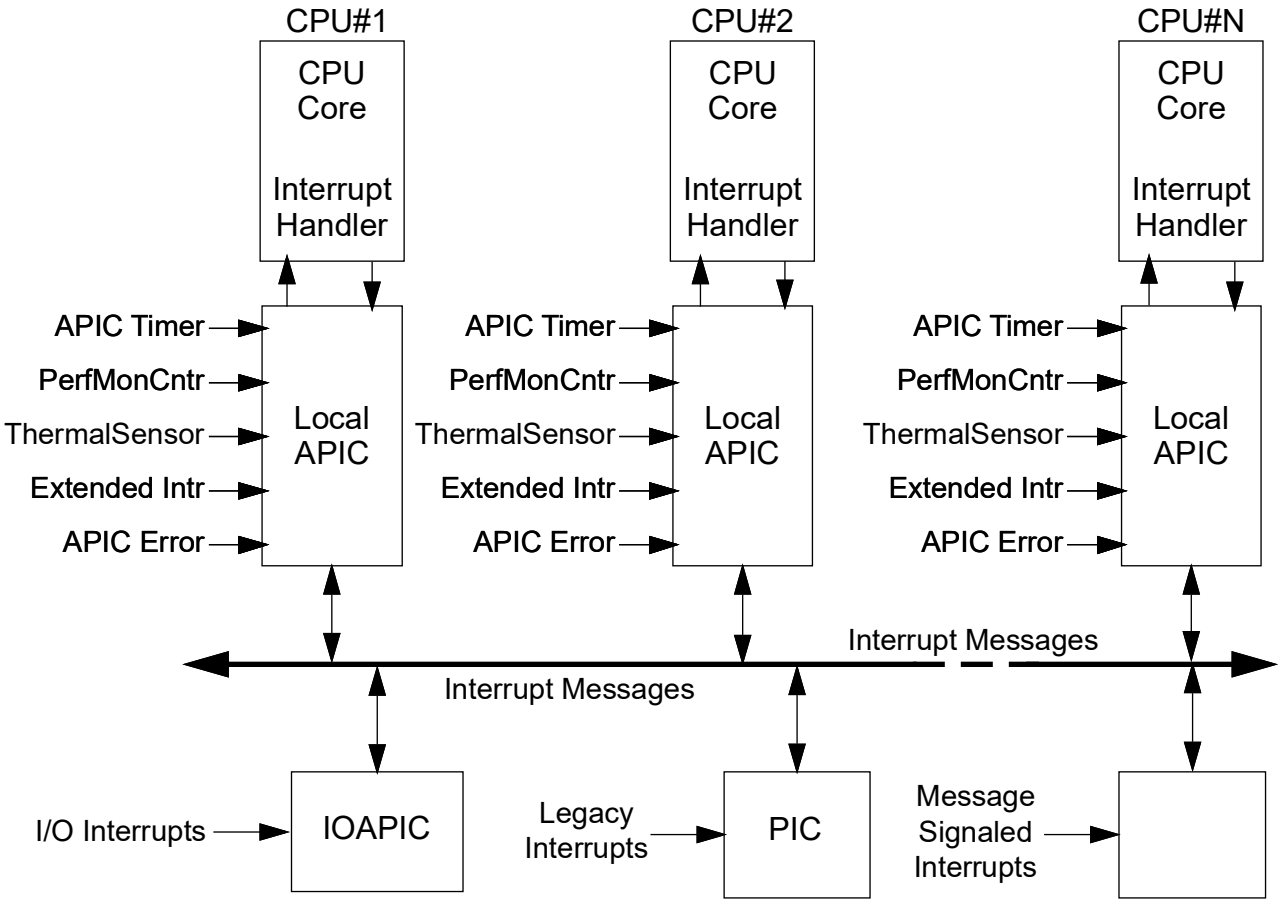


Figure 16-1. Block Diagram of a Typical APIC Implementation

[AMD Public Use]

16.1 Sources of Interrupts to the Local APIC

Each CPU core has an associated local APIC which receives interrupts from the following sources:

- I/O interrupts from the IOAPIC interrupt controller (including LINT0 and LINT1)
- Legacy interrupts (INTR and NMI) from the legacy interrupt controller
- Message Signalled Interrupts
- Interprocessor Interrupts (IPIs) from other local APICs. Interprocessor Interrupts are used to send interrupts or to execute system wide functions between CPU cores in the system, including the originating CPU core (self-interrupt).
- Locally generated interrupts within the local APIC. The local APIC receives local interrupts from the APIC timer, Performance Monitor Counters, thermal sensors, APIC errors and extended interrupts from implementation specific sources.

The sources of interrupts for the local APIC are provided in Table 16-1.

Table 16-1. Interrupt Sources for Local APIC

Source	Description	Message Type to Local APIC
I/O interrupts	System interrupts from I/O devices or system hardware received through the I/O APIC and sent to the local APIC as interrupt messages. They may be edge-triggered or level-sensitive.	Fixed, Lowest Priority, SMI, NMI, INIT, STARTUP, External interrupt, LINT0, LINT1
Legacy Interrupts	Legacy interrupts (INT and NMI) from the PIC and sent to the local APIC as interrupt messages.	NMI, INT
Interprocessor (IPI)	Interprocessor interrupts. Used for interrupt forwarding, system-wide functions, or software self-interrupts.	Fixed, lowest priority, SMI, read request, NMI, INIT, STARTUP, External interrupt
APIC Timer	Local interrupt from the programmed APIC timer reaches zero, under control of TIMER_LVT.	Fixed
Performance Monitor Counter	Local interrupt from the performance monitoring counter when it overflows, under control of PERF_CNT_LVT.	Fixed, SMI, or NMI
Thermal Sensor	Local interrupt from internal thermal sensors when it has tripped, under control of THERMAL_LVT.	Fixed, SMI, or NMI
Extended Interrupt[7:0]	Local Interrupts from programmable internal CPU core sources, under the control of the EXTENDED_INTERRUPT[7:0]_LVT.	Fixed, SMI, NMI, or External interrupt
APIC Internal Error	Local interrupt when an error is detected within the local APIC, under control of ERROR_LVT.	Fixed, SMI, or NMI

16.2 Interrupt Control

I/O, legacy, and interprocessor interrupts are sent via interrupt messages. The interrupt messages contain the following information:

[AMD Public Use]

- Destination address of the local APIC.
- VECTOR[7:0] indicating interrupt priority of up to 256 interrupt vectors. This information is captured in the IRR register for Fixed and Lowest Priority interrupt message types.
- Trigger Mode indicating edge triggered or level-sensitive (which requires an EOI response to the source).
- Message Type[3:0] indicating the type of interrupt to be presented to the local APIC. For Fixed and Lowest Priority message types, the interrupt is processed through the target local APIC. For all other message types, the interrupt is sent directly to the destination CPU core. There is a 5-line interrupt interface to the CPU core for INTR, SMI, NMI, INIT and STARTUP interrupts. For locally-generated interrupts, control is provided by local vector tables or LVTs. Separate LVTs are provided for each interrupt source, allowing for a unique entry point for each source. The LVT contains the VECTOR[7:0], trigger mode and message type as well as other fields associated with the specific interrupt. The message type may be Fixed, SMI, NMI, or External interrupt. A Mask bit is also provided to mask the interrupt.

16.3 Local APIC

16.3.1 Local APIC Enable

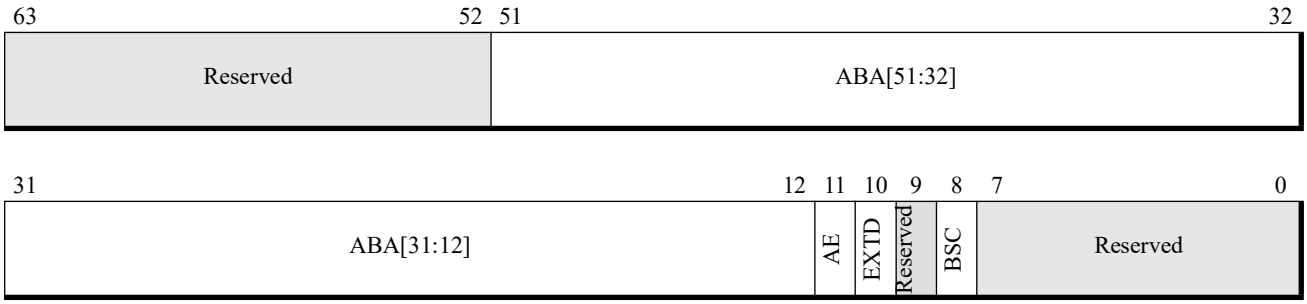
The local APIC is controlled by the APIC enable bit (AE) in the APIC Base Address Register (MSR 0000_001Bh). See Figure 16-2 on page 662.

When AE is set to 1, the local APIC is enabled and all interrupt types are accepted. When AE is cleared to 0, the local APIC is disabled, including all local vector table interrupts.

Software can disable the local APIC, using the APIC_SW_EN bit in the Spurious Interrupt Vector Register (APIC_F0). When this bit is cleared to zero, the local APIC is temporarily disabled:

- SMI, NMI, INIT, Startup, and Remote Read interrupts may be accepted.
- Pending interrupts in the ISR and IRR are held.
- Further fixed, lowest-priority, and ExtInt interrupts are not accepted.
- All LVT entry mask bits are set and cannot be cleared.

[AMD Public Use]



Bits	Mnemonic	Description	Access Type
63:52	Reserved		MBZ
51:12	ABA	APIC Base Address	R/W
11	AE	APIC Enable	R/W
10	EXTD	x2APIC Mode Enable	R/W
9	Reserved		MBZ
8	BSC	Boot Strap CPU Core	RO
7:0	Reserved		MBZ

Figure 16-2. APIC Base Address Register (MSR 0000_001Bh)

The fields within the APIC Base Address register are as follows:

- *Boot Strap CPU Core (BSC)*—Bit 8. The BSC bit indicates that this CPU core is the boot core of the BSP. Each CPU core that is not the boot core of the boot processor is an AP (Application Processor).
- *APIC Enable (AE)*—Bit 11. This is the APIC enable bit. The local APIC is enabled and all interruption types are accepted when AE is set to 1. Clearing AE to 0 disables the local APIC, and no local vector table interrupts are supported.
- *APIC Base Address (ABA)*—Bits 51:12. Specifies the base physical address for the APIC register set. The address is extended by 12 bits at the least-significant end to form the 52-bit physical base address. The reset value of the APIC base address is 0_0000_FEE0_0000h. This address is not affected by INIT.

Note that a given processor may implement a physical address less than 52 bits in length.

16.3.2 APIC Registers

The system programming interface of the local APIC is made up of the registers listed in Table 16-2 below. All APIC registers are memory-mapped into the 4-Kbyte APIC register space, and are accessed with memory reads and writes. The memory address is indicated as:

APIC Register address = APIC Base Address + Offset

where the APIC Base Address must point to an uncacheable memory region, and is located in APIC Base Address Register, MSR 0000_001Bh. See Figure 16-2.

[AMD Public Use]

APIC registers are aligned to 16-byte offsets and must be accessed using naturally-aligned DWORD size read and writes. All other accesses cause undefined behavior.

The table includes the value of each register after reset and INIT.

Table 16-2. APIC Registers

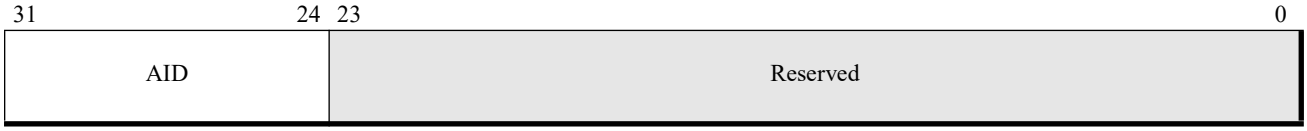
Offset	Name	Reset
20h	APIC ID Register	??000000h
30h	APIC Version Register	80??0010h
80h	Task Priority Register (TPR)	00000000h
90h	Arbitration Priority Register (APR)	00000000h
A0h	Processor Priority Register (PPR)	00000000h
B0h	End of Interrupt Register (EOI)	–
C0h	Remote Read Register	00000000h
D0h	Logical Destination Register (LDR)	00000000h
E0h	Destination Format Register (DFR)	FFFFFFFF
F0h	Spurious Interrupt Vector Register	000000FFh
100-170h	In-Service Register (ISR)	00000000h
180-1F0h	Trigger Mode Register (TMR)	00000000h
200-270h	Interrupt Request Register (IRR)	00000000h
280h	Error Status Register (ESR)	00000000h
300h	Interrupt Command Register Low (bits 31:0)	00000000h
310h	Interrupt Command Register High (bits 63:32)	00000000h
320h	Timer Local Vector Table Entry	00010000h
330h	Thermal Local Vector Table Entry	00010000h
340h	Performance Counter Local Vector Table Entry	00010000h
350h	Local Interrupt 0 Vector Table Entry	00010000h
360h	Local Interrupt 1 Vector Table Entry	00010000h
370h	Error Vector Table Entry	00010000h
380h	Timer Initial Count Register	00000000h
390h	Timer Current Count Register	00000000h
3E0h	Timer Divide Configuration Register	00000000h
400h	Extended APIC Feature Register	00040007h
410h	Extended APIC Control Register	00000000h
420h	Specific End of Interrupt Register (SEOI)	–
480-4F0h	Interrupt Enable Registers (IER)	FFFFFFFFh
500-570h	Extended Interrupt [7:0] Local Vector Table Registers	00000000h

[AMD Public Use]

16.3.3 Local APIC ID

Unique local APIC IDs are assigned to each CPU core in the system. The value is determined by hardware, based on the number of CPU cores on the processor and the node ID of the processor.

The APIC ID is located in the APIC ID register at APIC offset 20h. See Figure 16-3. It is model dependent, whether software can modify the APIC ID Register. The initial value of the APIC ID (after a reset) is the value returned in CPUID function 0000_0001h_EBX[31:24].



Bits	Mnemonic	Description	Access type
31:24	AID	APIC ID	R/W
23:0	Reserved		MBZ

Figure 16-3. APIC ID Register (APIC Offset 20h)

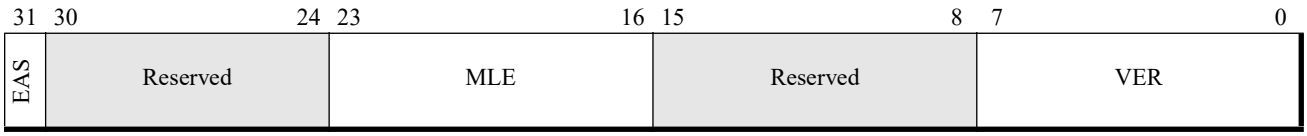
- *APIC ID (AID)*—Bits 31:24. The APIC ID field contains the unique APIC ID value assigned to this specific CPU core. A given implementation may use some bits to represent the CPU core and other bits represent the processor.

16.3.4 APIC Version Register

A version register is provided to allow software to identify which APIC version is used. Bits 7:0 of the APIC Version Register indicate the version number of the APIC implementation.

The number of entries in the local vector table are specified in bits 23:16 of the register as the maximum number minus one.

Bit 31 indicates the presence of extended APIC registers which have an offset starting at 400h.



Bits	Mnemonic	Description	Access type
31	EAS	Extended APIC Register Space Present	RO
30:24	Reserved		MBZ
23:16	MLE	Max LVT Entries	RO
15:8	Reserved		MBZ
7:0	VER	Version	RO

Figure 16-4. APIC Version Register (APIC Offset 30h)

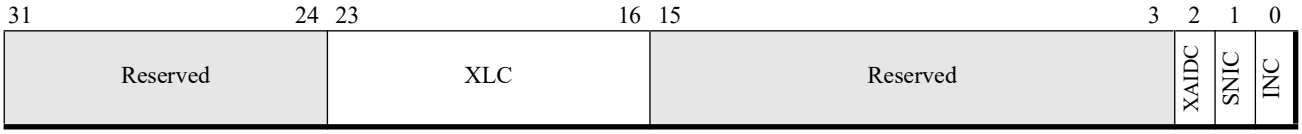
[AMD Public Use]

The fields within the APIC Version register are as follows:

- *Version (VER)*—Bits 7:0. The VER field indicates the version number of the APIC implementation. The local APIC implementation is identified with a value=1Xh (20h-FFh are reserved).
- *Max LVT Entries (MLE)*—Bits 23:16. The MLE field specifies the number of entries in the local vector table minus one.
- *Extended APIC Register Space Present (EAS)*—Bit 31. The EAS bit when set to 1 indicates the presence of an extended APIC register space, starting at offset 400h.

16.3.5 Extended APIC Feature Register

The Extended APIC Feature Register indicates the number of extended Local Vector Table registers in the local APIC, whether the Interrupt Enable Registers are present, and whether the 8-bit Extended APIC ID and Specific End Of Interrupt (SEOI) Register are supported.



Bits	Mnemonic	Description	Access type
31:24	Reserved		MBZ
23:16	XLC	Extended LVT Count	RO
15:3	Reserved		MBZ
2	XAIDC	Extended APIC ID Capable	RO
1	SNIC	Specific End of Interrupt Capable	RO
0	INC	Interrupt Enable Register Capable	RO

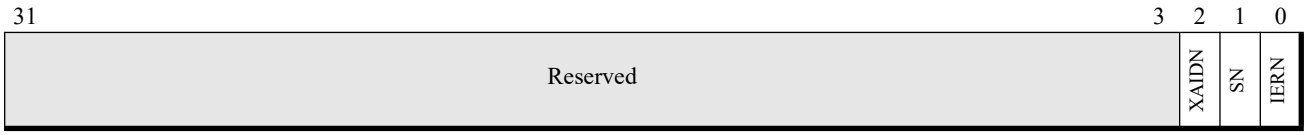
Figure 16-5. Extended APIC Feature Register (APIC Offset 400h)

- Extended LVT Count (XLC)—(Bits 23:16) Specifies the number of extended local vector table registers in the local APIC.
- Extended APIC ID Capability (XAIDC)—(Bit 2) Indicates that the processor is capable of supporting an 8-bit APIC ID.
- Specific End of Interrupt Capable—(Bit 1) Indicates that the Specific End Of Interrupt Register is present.
- Interrupt Enable Register Capable—(Bit 0) Read-only. Indicates that the Interrupt Enable Registers are present.

16.3.6 Extended APIC Control Register

This bit enables writes to the interrupt enable registers.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
31:3	Reserved		MBZ
2	XAIDN	Extended APIC ID Enable.	R/W
1	SN	Enable SEOI Generation	R/W
0	IERN	Enable Interrupt Enable Registers	R/W

Figure 16-6. Extended APIC Control Register (APIC Offset 410h)

- Extended APIC ID Enable (XAIDN)—Bit 2. Setting XAIDN to 1 enables the upper four bits of the APIC ID field described in "APIC ID Register (APIC Offset 20h)" on page 664. Clearing this bit, specifies a 4-bit APIC ID using only the lower four bits of the APIC ID field of the APIC ID register.
- Enable SEOI Generation (SN)—Bit 1. Read-write. This bit enables Specific End of Interrupt (SEOI) generation when a write to the specific end of interrupt register is received.
- Enable Interrupt Enable Registers (IERN)—Bit 0. This bit enables writes to the interrupt enable registers.

16.4 Local Interrupts

The local APIC handles the following local interrupts:

- APIC Timer
- Local Interrupt 0 (LINT0)
- Local Interrupt 1 (LINT1)
- Performance Monitor Counters
- Thermal Sensors
- APIC internal error
- Extended (Implementation dependent)

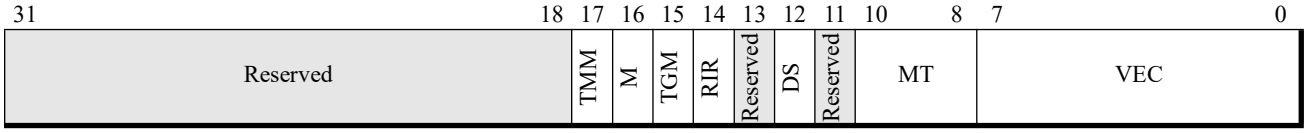
A separate entry in the local vector table is provided for each interrupt to allow software to specify:

- Whether the interrupt is masked or not.
- The delivery status of the interrupt.
- The message type.
- The unique address vector.
- For LINT0 and LINT1 interrupts, the trigger mode, remote IRR, and input pin polarity.

[AMD Public Use]

- For the APIC timer interrupt, the timer mode.

The general format of a Local Vector Table Register is shown in Figure 16-7.



Bits	Mnemonic	Description	Access type
31:18	Reserved		MBZ
17	TMM	Timer Mode	R/W
16	M	Mask	R/W
15	TGM	Trigger Mode	R/W
14	RIR	Remote IRR	RO
13	Reserved		MBZ
12	DS	Delivery Status	RO
11	Reserved		MBZ
10:8	MT	Message Type	R/W
7:0	VEC	Vector	R/W

Figure 16-7. General Local Vector Table Register Format

The fields within the General Local Vector Table register are as follows:

- *Vector (VEC)*—Bits 7:0. The VEC field contains the vector that is sent for this interrupt source when the message type is fixed. It is ignored when the message type is NMI and is set to 00h when the message type is SMI. Valid values for the vector field are from 16 to 255. A value of 0 to 15 when the message type is fixed results in an illegal vector APIC error.
- *Message Type (MT)*—Bits 10:8. The MT field specifies the delivery mode sent to the CPU core interrupt handler. The legal values are:
  - 000b = Fixed - The vector field specifies the interrupt delivered.
  - 010b = SMI - An SMI interrupt is delivered. In this case, the vector field should be set to 00h.
  - 100b = NMI - A NMI interrupt is delivered with the vector field being ignored.
  - 111b = External interrupt is delivered.
- *Delivery Status (DS)*—Bit 12. The DS bit indicates the interrupt delivery status. The DS bit is set to 1 when the interrupt is pending at the CPU core interrupt handler. After a successful delivery of the interrupt, the associated bit in the IRR is set and this bit is cleared to zero. See Section 16.6.2, “Lowest Priority Messages and Arbitration,” on page 678 for details. The bit is cleared to 0 when the interrupt is idle.
- *Remote IRR (RIR)*—Bit 14. The RIR bit is set to 1 when the local APIC accepts an LINT0 or LINT1 interrupt with the trigger mode=1 (level sensitive). The bit is cleared to 0 when the interrupt completes, as indicated when an EOI is received.

[AMD Public Use]

- *Trigger Mode (TGM)*—Bit 15. Specifies how interrupts to the local APIC are triggered. The TGM bit is set to 1 when the interrupt is level-sensitive. It is cleared to 0 when the interrupt is edge-triggered. When the message type is SMI or NMI, the trigger mode is edge triggered.
- *Mask (M)*—Bit 16. When the M bit is set to 1, reception of the interrupt is disabled. When the M bit is cleared to 0, reception of the interrupt is enabled.
- *Timer Mode (TMM)*—Bit 17. Specifies the timer mode for the APIC Timer interrupt. The TMM bit set to 1 indicates periodic timer interrupts. The TMM bit cleared to 0 indicates one-shot operation.

16.4.1 APIC Timer Interrupt

The APIC timer is a programmable 32-bit counter used by software to time operations or events. The timer can operate in two modes, periodic and one-shot, under the control of bit 17 (Timer Mode) in APIC Timer Local Vector Table Register (see Figure 16-8). In both modes, the APIC timer is set to a programmable initial value and starts to decrement at a programmable clock rate. When the Initial Count Register is written to a non-zero value, the APIC timer is initialized to the value just written and starts decrementing. When the Initial Count Register is written to zero, the APIC timer is initialized to zero and stops decrementing. In one-shot mode, the APIC timer stops counting when the timer reaches zero. In periodic mode, the APIC timer is initialized again when it reaches zero, and it starts to decrement again. Whenever the timer value is decremented to zero, an APIC timer interrupt is generated under the control of bit 16 (Mask) in the APIC Timer Local Vector Table Register.

To avoid race conditions, software should initialize the Divide Configuration Register and the Timer Local Vector Table Register prior to writing the Initial Count Register to start the timer.

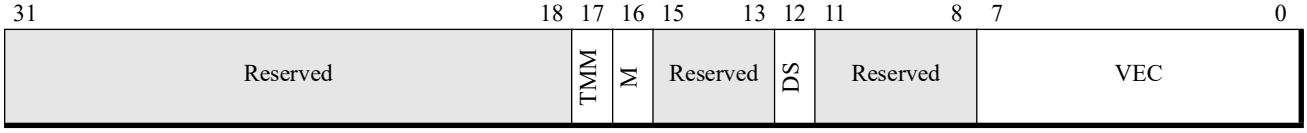


Figure 16-8. APIC Timer Local Vector Table Register (APIC Offset 320h)

Three APIC registers are defined for the APIC timer function:

- Current Count Register (CCR) is the actual APIC timer. Whenever the ICR is written to a non-zero value, or when the CCR reaches zero while in periodic mode, it is initialized to a start count loaded from the ICR and then decrements. The APIC timer interrupt is generated when the CCR value reaches zero. The counting rate is controlled by the DCR. See Figure 16-9.
- Initial Count Register (ICR) provides the initial value for the APIC timer. See Table 16-10.
- Divide Configuration Register (DCR) controls the counting rate of the APIC timer by dividing the CPU core clock by a programmable amount. See Figure 16-11. For the specific details on the implementation of the APIC timer base clock rate, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
31:0	APICTCC	APIC Timer Current Count	RO

Figure 16-9. Timer Current Count Register (APIC Offset 390h)

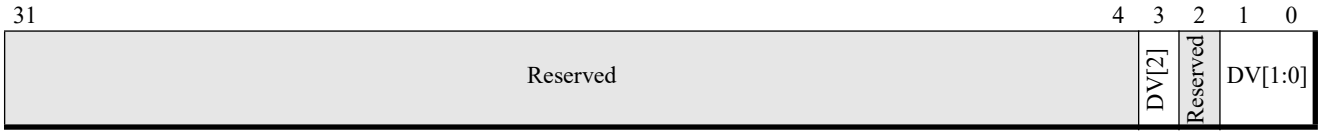
- *APIC Timer Current Count (APICTCC)*—Bits 31:0. The APICTCC field contains the current value of the APIC timer.



Bits	Mnemonic	Description	Access type
31:0	APICTIC	APIC Timer Initial Count	R/W

Figure 16-10. Timer Initial Count Register (APIC Offset 380h)

- *APIC Timer Initial Count (APICTIC)*—Bits 31:0. The APICTIC field contains the value that is loaded into the APIC Timer Current Count Register when the APIC timer is initialized.



Bits	Mnemonic	Description	Access type
31:4	Reserved		MBZ
3	DV[2]	Divide Value[2]	R/W
2	Reserved	,	MBZ
1:0	DV[1:0]	Divide Value[1:0]	R/W

Figure 16-11. Divide Configuration Register (APIC Offset 3E0h)

- *Divide Value (DV)*—Bits 3, 1:0. The DV field specifies the value of the CPU core clock divisor. Table 16-3 lists the allowable values.

[AMD Public Use]

Table 16-3. Divide Values

Bits 3, 1:0	Resulting Timer Divide
000b	Divide by 2
001b	Divide by 4
010b	Divide by 8
011b	Divide by 16
100b	Divide by 32
101b	Divide by 64
110b	Divide by 128
111b	Divide by 1

16.4.2 Local Interrupts LINT0 and LINT1

When the target local APIC receives an interrupt message from an IOAPIC with the LINT0 or LINT1 message type, the appropriate local interrupt is generated under the control of bit 16 (Mask) in the APIC LINT0 or LINT1 Local Vector Table Register. See Figure 16-12.

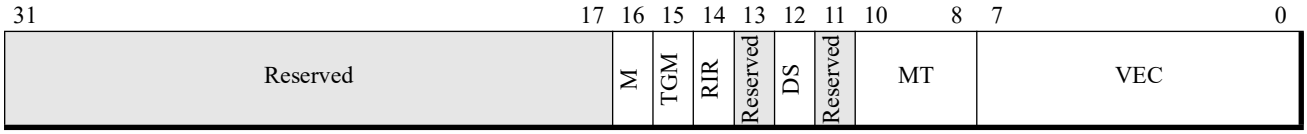


Figure 16-12. Local Interrupt 0/1 (LINT0/1) Local Vector Table Register (APIC Offset 350h/360h)

In addition to the normal LVT control bits (mask, delivery status and vector offset), the LINT0/LINT1 interrupts provide the following controls:

- Trigger Mode - indicates whether the interrupt pin is edge triggered or level sensitive when the message type is fixed.
- Remote IRR - When the trigger mode indicates level, this flag is set when the local APIC accepts the interrupt, and is reset when the local APIC receives an EOI. When the flag is set, no additional local interrupt requests are sent to the local APIC, and they remain pending.

16.4.3 Performance Monitor Counter Interrupts

When a performance monitor counter overflows, an APIC interrupt is generated under the control of bit 16 (Mask) in the APIC Performance Monitor Counter Local Vector Table Register. See Figure 16-13 on page 670.

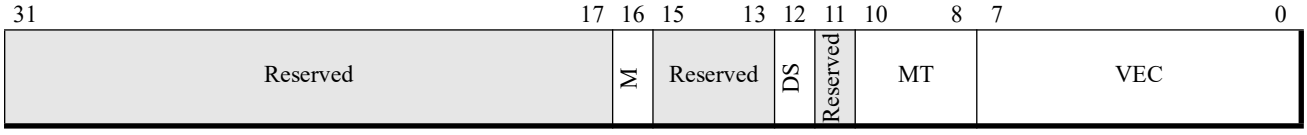


Figure 16-13. Performance Monitor Counter Local Vector Table Register (APIC Offset 340h)

[AMD Public Use]

16.4.4 Thermal Sensor Interrupts

When a thermal event occurs, an APIC interrupt is generated under the control of bit 16 (Mask) in the APIC Thermal Sensor Local Vector Table Register. See Figure 16-14. See the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product for more information on thermal events. This interrupt may not be supported in all implementations.

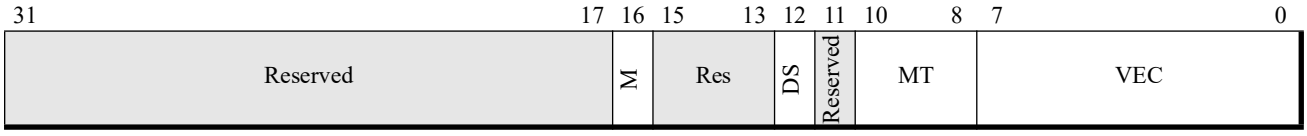


Figure 16-14. Thermal Sensor Local Vector Table Register (APIC Offset 330h)

16.4.5 Extended Interrupts

The local interrupts are extended to include more LVT registers, to allow additional interrupt sources. The additional sources are model dependent and can include:

- Counter overflow from the Machine Check Miscellaneous Threshold Register. See "Machine-Check Miscellaneous-Error Information Register 0 (MCi_MISC0)" on page 313 for details.
- ECC Error Count Threshold in memory system.
- Instruction Sampling.

The LVT register used for each interrupt source is specified by the control register associated with the source.

The Extended LVT Count field (bits 23:16) of the Extended APIC Feature Register specifies the number of extended LVT registers. Currently there are eight additional LVT registers defined, Extended Interrupt [7:0], Local Vector Table Register, located at APIC offsets 500h–570h. (See Section 16.7.1, "Specific End of Interrupt Register," on page 684 and Figure 16-5 on page 665.)

16.4.6 APIC Error Interrupts

Errors that are detected while handling interrupts cause an APIC error interrupt to be generated under the control of bit 16 (Mask) in the APIC Error Local Vector Table Register. See Figure 16-15 on page 671.

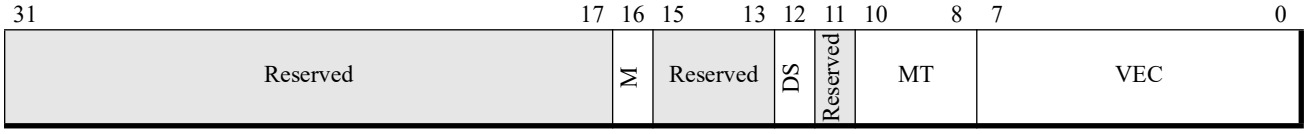
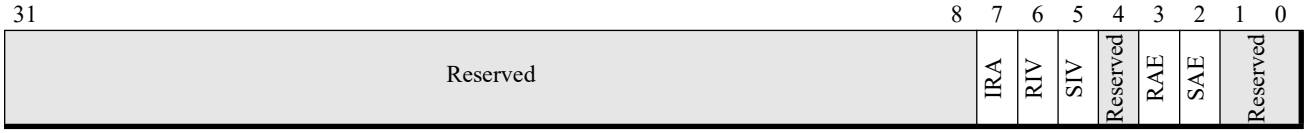


Figure 16-15. APIC Error Local Vector Table Register (APIC Offset 370h)

[AMD Public Use]

The error information is recorded in the APIC Error Status Registers. The APIC Error Status Register is a read-write register. Writes to the register cause the internal error state to be recorded in the register, clearing the original error. See Figure 16-16.



Bits	Mnemonic	Description	Access type
31:8	Reserved		MBZ
7	IRA	Illegal Register Address	R/W
6	RIV	Received Illegal Vector	R/W
5	SIV	Sent Illegal Vector	R/W
4	Reserved		MBZ
3	RAE	Receive Accept Error	R/W
2	SAE	Sent Accept Error	R/W
1:0	Reserved		MBZ

Figure 16-16. APIC Error Status Register (APIC Offset 280h)

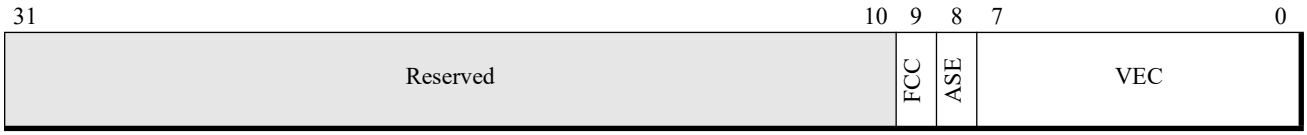
The fields within the APIC Error Status register are as follows:

- *Sent Accept Error (SAE)*—Bit 2. The SAE bit when set to 1 indicates that a message sent by the local APIC was not accepted by any other APIC.
- *Receive Accept Error (RAE)*—Bit 3. The RAE bit when set to 1 indicates that a message received by the local APIC was not accepted by this or any other APIC
- *Sent Illegal Vector (SIV)*—Bit 5. The SIV bit when set to 1 indicates that the local APIC attempted to send a message with an illegal vector value.
- *Receive Illegal Vector (RIV)*—Bit 6. The RIV bit when set to 1 indicates that the local APIC has received a message with an illegal vector value.
- *Illegal Register Address (IRA)*—Bit 7. The IRA bit when set to 1 indicates that an access to an unimplemented register location within the local APIC register range (APIC Base Address + 4 Kbytes) was attempted.

16.4.7 Spurious Interrupts

A timing issue exists between software and hardware that, though rare, results in spurious interrupts. In the event that the task priority is set to or above the level of the interrupt to be serviced while the interrupt is being acknowledged, the local APIC delivers a spurious interrupt to the CPU core instead, with the vector number specified by the Vector field of the Spurious Interrupt Register. The ISR is unaffected by the spurious interrupt, so the interrupt handler completes without sending an EOI back to the issuing local APIC.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
31:10	Reserved		MBZ
9	FCC	Focus CPU Core Checking	R/W
8	ASE	APIC Software Enable	R/W
7:0	VEC	Vector	R/W

Figure 16-17. Spurious Interrupt Register (APIC Offset F0h)

The fields within the Spurious Interrupt register are as follows:

- *Vector (VEC)*—Bits 7:0. The VEC field contains the vector that is sent to the CPU core in the event of a spurious interrupt.
- *APIC Software Enable (ASE)*—Bit 8. The ASE bit when set to 0 disables the local APIC temporarily. When the local APIC is disabled, SMI, NMI, INIT, Startup, and Remote Read may be accepted; pending interrupts in the ISR and IRR are held, but further fixed, lowest-priority, and ExtInt interrupts are not accepted. All LVT entry mask bits are set and cannot be cleared. Setting the ASE bit to 1, enables the local APIC.
- *Focus CPU Core Checking (FCC)*—Bit 9. The FCC bit when set to 1 disables focus CPU core checking when the lowest-priority message type is used. A CPU core is the focus of an interrupt if it is already servicing that interrupt (ISR=1) or if it has a pending request for that interrupt (IRR=1). Clearing the FCC bit to 0 disables focus CPU core checking.

16.5 Interprocessor Interrupts (IPI)

A local APIC can send interrupts to other local APICs (or itself) using software-initiated Interprocessor Interrupts (IPIs) using the Interrupt Command Register (ICR). Writing into the low order doubleword of the ICR causes the IPI to be sent.

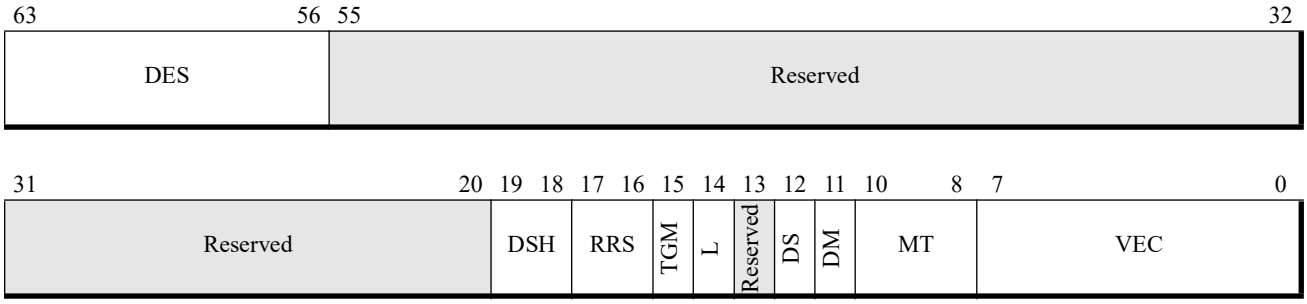
The ICR can issue the following types of interrupt messages:

- basic interrupt message to another local APIC, including forwarding an interrupt that was received but not serviced
- basic interrupt message to the same local APIC (self-interrupt)
- system management interrupt (SMI)
- remote read message to another local APIC to read one of its APIC registers.
- non-maskable interrupt (NMI) delivered to another local APIC

[AMD Public Use]

- initialization message (INIT) to a target local APIC to be reset to their INIT state and await a STARTUP IPI.
- startup message (SIPI) to the target local APICs, pointing to a start-up routine.

The format of the Interrupt Command Register is shown in Figure 16-18.



Bits	Mnemonic	Description	Access type
63:56	DES	Destination	R/W
55:20	Reserved		MBZ
19:18	DSH	Destination Shorthand	R/W
17:16	RRS	Remote Read Status	RO
15	TGM	Trigger Mode	R/W
14	L	Level	R/W
13	Reserved		MBZ
12	DS	Delivery Status	RO
11	DM	Destination Mode	R/W
10:8	MT	Message Type	R/W
7:0	VEC	Vector	R/W

Figure 16-18. Interrupt Command Register (APIC Offset 300h–310h)

The fields within the Interrupt Command register are as follows:

- *Vector (VEC)*—Bits 7:0. The function of this field varies with the Message Type field. The VEC field contains the vector that is sent for this interrupt source for fixed and lowest priority message types.
- *Message Type (MT)*—Bits 10:8. The MT field specifies the message type sent to the CPU core interrupt handler. The legal values are:
  - 000b = Fixed - The IPI delivers an interrupt to the target local APIC specified in Destination field.
  - 001b = Lowest Priority - The IPI delivers an interrupt to the local APIC executing at the lowest priority of all local APICs that match the destination logical ID specified in the Destination field. See Section 16.6.1, “Receiving System and IPI Interrupts,” on page 677.
  - 010b = SMI - The IPI delivers an SMI interrupt to target local APIC(s). The trigger mode is edge-triggered and the Vector field must = 00h.

[AMD Public Use]

  - 011b = Remote read - The IPI delivers a read request to read an APIC register in the target local APIC specified in Destination field. The trigger mode is edge triggered and the Vector field specifies the APIC offset of the APIC register to be read. The Remote Status field provides the current status of the remote read access after it has been issued. Data is returned from the target local APIC and captured in the Remote Read Register of the issuing local APIC. See Figure 16-19 on page 676.
  - 100b = NMI - The IPI delivers a non-maskable interrupt to the target local APIC specified in the Destination field. The Vector field is ignored.
  - 101b = INIT - The IPI delivers an INIT request to the target local APIC(s) specified in the Destination field, causing the CPU core to assume the INIT state. The trigger mode is edge-triggered, and the Vector field must =00h. In the INIT state, the target APIC is responsive only to the STARTUP IPI. All other interrupts (including SMI and NMI) are held pending until the STARTUP IPI has been accepted.
  - 110b = STARTUP - The IPI delivers a start-up request (SIPI) to the target local APIC(s) specified in Destination field, causing the CPU core to start processing the platform firmware boot-strap routine whose address is specified by the Vector field.
  - 111b = External interrupt - The IPI delivers an external interrupt to the target local APIC specified in Destination field. The interrupt can be delivered even if the APIC is disabled.

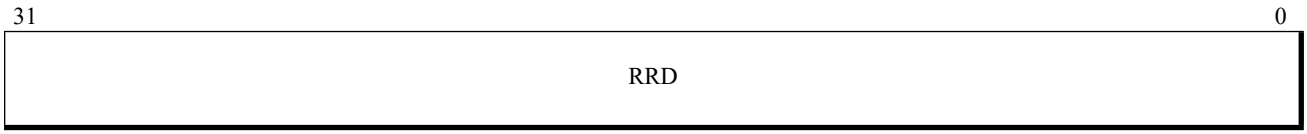
- *Destination Mode (DM)*—Bit 11. The DM bit when set to 1 specifies a logical destination which may be one or more local APICs with a common destination logical ID. When cleared to 0, the DM bit specifies a physical destination which indicates a single local APIC ID.
- *Delivery Status (DS)*—Bit 12. The DS bit indicates the interrupt delivery status. The DS bit is set to 1 when the local APIC has sent the IPI and is waiting for it to be accepted by another local APIC (the ICR is not idle). Clearing the DS bit indicates that the target local APIC is idle. Code may repeatedly write ICRL without polling the DS bit; all requested IPIs will be delivered.
- *Level (L)*—Bit 14. The L bit when set to 1 indicates assert. Clearing the L bit to 0 indicates deassert.
- *Trigger Mode (TGM)*—Bit 15. Specifies how IPIs to the local APIC are triggered. The TGM bit is set to 1 when the interrupt is level-sensitive. It is cleared to 0 when the interrupt is edge-triggered.
- *Remote Read Status (RRS)*—Bits 17:16. The RRS field indicates the current read status of a Remote Read from another local APIC. The encoding for this field is as follows:
  - 00b = Read was invalid
  - 01b = Delivery pending
  - 10b = Delivery done and access was valid. Data available in Remote Read Register.
  - 11b = Reserved
- *Destination Shorthand (DSH)*—Bits 19:18. The DSH field indicates whether a shorthand notation is used, and provides a quick way to specify a destination for a message. It replaces the Destination field, when the destination field is not required (DSH > 00b), allowing software to use a single write to the low order ICR. The encoding are as follows:

[AMD Public Use]

- 00b = Destination - The Destination field is required to specify the destination.
- 01b = Self - The issuing APIC is the only destination.
- 10b = All including self - The IPI is sent to all local APICs including itself (destination field=FFh).
- 11b = All excluding self - The IPI is sent to all local APICs except itself (destination field=FFh).

  Note that if the lowest priority is used, the message could end up being reflected back to this local APIC. If DS=1xb, the destination mode is ignored and physical is automatically used.

- *Destination (DES)*—Bits 63:56. The DES field identifies the target local APIC(s) for the IPI and contains the destination encoding used when the Destination Shorthand field=00b. The field indicates the target local APIC when the destination mode=0 (physical), and the destination logical ID (as indicated by LDR and DFR) when the destination mode=1 (logical).



Bits	Mnemonic	Description	Access type
31:0	RRD	Remote Read Data	RO

Figure 16-19. Remote Read Register (APIC Offset C0h)

- *Remote Read Data (RRD)*—Bits 31:0. The RRD field contains the data resulting from a valid completion of a remote read interprocessor interrupt.

Not all combinations of ICR fields are valid. Only the combinations indicated in Table 16-4 are valid.

Table 16-4. Valid ICR Field Combinations

Message Type	Trigger Mode	Level	Destination Shorthand
Fixed	Edge	x	x
	Level	Assert	x
Lowest Priority, SMI, NMI, INIT	Edge	x	Destination or all excluding self
	Level	Assert	Destination or all excluding self
Startup	x	x	Destination or all excluding self

Note: *x indicates a don't care.*

[AMD Public Use]

16.6 Local APIC Handling of Interrupts

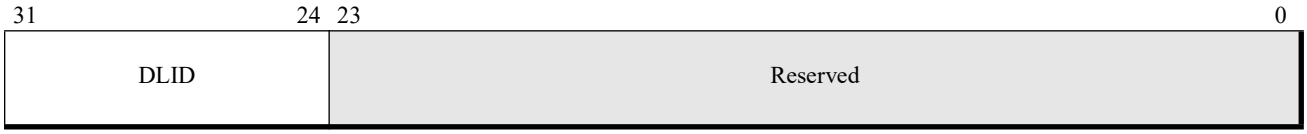
16.6.1 Receiving System and IPI Interrupts

Each local APIC verifies the destination ID, the destination mode and the message type of an APIC interrupt to determine if it is the target of the interrupt.

The destination mode is either physical or logical. In physical destination mode, the value of the interrupt message destination field is compared with the unique APIC ID value of each local APIC to select the target local APIC. If the destination field of the Interrupt Command Register is set to FFh, the interrupt is broadcasted and accepted by all local APICs. In physical destination mode, the lowest priority message type is not supported.

In logical destination mode, all local APICs use the Logical Destination Register and the Destination Format Register to determine if the interrupt is directed to them. The value of the interrupt message destination field is compared with the value in the Logical Destination Register (see Figure 16-20) of all local APICs.

The logical APIC ID must be unique. Since the comparison with the interrupt message destination field is on a bit-basis, there are only 8 unique logical IDs (01h, 02h, 04h, 08h, 10h, 20h, 40h, and 80h). For flat mode, the logical ID must be one of these values (for a total of eight local APICs supported). In cluster mode, the value of the logical ID is constrained to be *xy*h, where $0 \le x \le$ Eh and $y =$ either 1,2,4, or 8, for a total of (15 × 4) possible unique logical IDs.



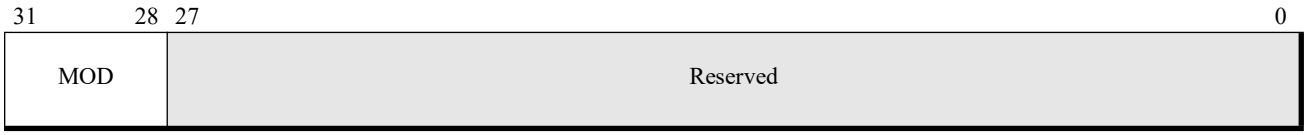
Bits	Mnemonic	Description	Access type
31:24	DLID	Destination Logical ID	R/W
23:0	Reserved		MBZ

Figure 16-20. Logical Destination Register (APIC Offset D0h)

- *Destination Logical ID (DLID)*—Bits 31:24. The DLID field contains the logical APIC ID assigned to this specific CPU core. The logical APIC ID must be unique.

Two interrupt models are defined for the logical destination mode, the flat model and the cluster model, under the control of the Destination Format Register. See Figure 16-21.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
31:28	MOD	Model	R/W
27:0	Reserved	All ones	RO

Figure 16-21. Destination Format Register (APIC Offset E0h)

- *Model (MOD)*—Bits 31:28. The MOD field controls which format to use when accepting interrupts in logical destination mode. The allowable values are 0h = cluster model and Fh = flat model.

With the flat model, up to eight unique logical APIC ID values can be provided by software by setting a different bit in the LDR. When the logical ID of the destination is compared with the LDR, if any bit position is set in both fields, this local APIC is a valid destination. A broadcast to all local APICs occurs when the LDR is set to all ones.

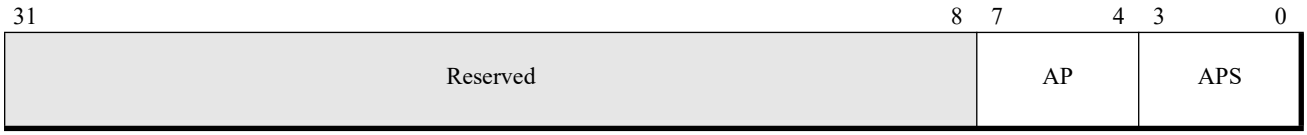
In the cluster model, bits 31:28 of the logical ID of the destination are compared with bits 31:28 of the LDR. If there is a match, then bits 27:24 are tested for matching ones, similar to the flat model. If bits 31:28 match, and any of bits 27:24 are set in both fields, this local APIC is a valid destination. The cluster model allows for 15 unique clusters to be defined, with each cluster having four unique logical APIC values to be addressed. In cluster logical destination mode, lowest priority message type is not supported.

In both the flat model and the cluster model, if the destination field = FFh, the interrupt is accepted by all local APICs.

16.6.2 Lowest Priority Messages and Arbitration

In the case where the interrupt is valid for several local APICs in logical destination mode with a lowest priority message type, the interrupt is accepted by the local APIC with the lowest arbitration priority, as indicated by the *Arbitration Priority* field in the Arbitration Priority Register (APR). The value in the *Arbitration Priority* field indicates the current priority for a pending interrupt or task, or an interrupt being serviced by the CPU core. See Figure 16-22.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
31:8	Reserved		MBZ
7:4	AP	Arbitration Priority	RO
3:0	APS	Arbitration Priority Sub-class	RO

Figure 16-22. Arbitration Priority Register (APIC Offset 90h)

The fields within the Arbitration Priority register are as follows:

- *Arbitration Priority Sub-class (APS)*—Bits 3:0. The APS field indicates the current sub-priority to handle arbitrated interrupts to be serviced by the CPU core.
- *Arbitration Priority (AP)*—Bits 7:4. The AP field indicates the current priority to handle arbitrated interrupts to be serviced by the CPU core. The priority is used to arbitrate between CPU cores to determine which core accepts a lowest-priority interrupt request.

The value in the Arbitration Priority field is equal to the highest priority of the Task Priority field of the Task Priority Register (TPR), the highest bit set in the In-Service Register (ISR) vector, or the highest bit set in the Interrupt Request Register (IRR) vector. The value in the Arbitration Priority Sub-class field is equal to the Task Priority Sub-class if the APR is equal to the TPR, and zero otherwise.

If focus CPU core checking is enabled (Spurious Interrupt Register bit 9=0), the focus CPU core for an interrupt can always accept the interrupt. A CPU core is the focus of an interrupt if it is already servicing that interrupt (corresponding ISR bit is set) or if it already has a pending request for that interrupt (corresponding IRR bit is set). If there is no focus CPU core for an interrupt or if focus CPU core checking is disabled (Spurious Interrupt Register bit 9=1), all target local APICs identified as candidates for the interrupt arbitrate to determine which is executing with the lowest arbitration priority. If there is a tie for lowest priority, the local APIC with the highest APIC ID is selected.

16.6.3 Accepting System and IPI Interrupts

If the local APIC accepting the interrupt determines that the message type for the interrupt request indicates SMI, NMI, INIT, STARTUP or ExtINT, it sends the interrupt directly to the CPU core for handling. If the message type is fixed or lowest priority, the accepting local APIC places the interrupt into an open slot in either the IRR or ISR registers. If there is no free slot, the interrupt is rejected and sent back to the sender with a retry request.

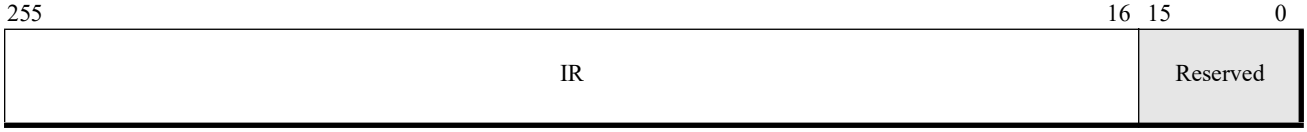
Three 256-bit acceptance registers support interrupts accepted by the local APIC. Bits 255:16 correspond to interrupt vectors 255:16 with 255 being the highest priority; bits 15:0 are reserved.

- Interrupt Request Register (IRR), which contains interrupt requests that have been accepted but have not been sent to the CPU core for interrupt handling. When a system interrupt is accepted, the associated bit corresponding to the interrupt vector is set in the IRR. When the CPU core requests a

[AMD Public Use]

new interrupt, the local APIC selects the highest priority IRR interrupt and sends it to the CPU core. The local APIC then sets the corresponding bit in the ISR and resets the associated IRR bit. See Figure 16-23 on page 680.

- In-Service Register (ISR) contains the bit map of the interrupts that have been sent to the CPU core and are still being serviced. When the CPU core writes to the EOI register indicating completion of the interrupt processing, the associated ISR bit is reset and a new interrupt is selected from the IRR register. If a higher priority interrupt is accepted by the local APIC while the CPU core is servicing another interrupt, the higher priority interrupt is sent directly to the CPU core (before the current interrupt finishes processing) and the associated IRR bit is set. The CPU core interrupts the current interrupt handler to service the higher priority interrupt. When the interrupt handler for the higher priority interrupt completes, the associated IRR bit is reset and the interrupt handler returns to complete the previous interrupt handler routine. If a second interrupt with the same interrupt vector number is received by the local APIC while the ISR bit is set, the local APIC sets the IRR bit. No more than two interrupts can be pending for the same interrupt vector number. Subsequent interrupt requests to the same interrupt vector number will be rejected. See Figure 16-24 on page 681.
- Trigger Mode Register (TMR) indicates the trigger mode of the interrupt and determines whether an EOI message is sent to the I/O APIC for level-sensitive interrupts. When the interrupt is accepted by the local APIC and the IRR bit is set, the associated TMR bit is set for level-sensitive interrupts or reset for edge-triggered interrupts. At the end of the interrupt handler routine, when the EOI is received at the local APIC, an EOI message is sent to the I/O APIC if the associated TMR bit is set for a system interrupt. See Figure 16-25 on page 681.



Bits	Mnemonic	Description	Access type
255:16	IR	Interrupt Request bits	RO
15:0	Reserved		MBZ

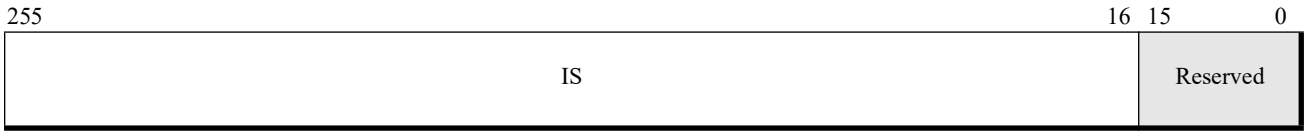
Figure 16-23. Interrupt Request Register (APIC Offset 200h–270h)

- *Interrupt Request bits (IR)*—Bits 255:16. The corresponding request bit is set when an interrupt is accepted by the local APIC. The interrupt request registers provide a bit per interrupt to indicate that the corresponding interrupt has been accepted by the local APIC. Interrupts are mapped as follows:

Register	Interrupt Number
IRR (APIC offset 200h)	31–16
IRR (APIC offset 210h)	63–32
IRR (APIC offset 220h)	95–64
IRR (APIC offset 230h)	127–96

[AMD Public Use]

Register	Interrupt Number
IRR (APIC offset 240h)	159–128
IRR (APIC offset 250h)	191–160
IRR (APIC offset 260h)	223–192
IRR (APIC offset 270h)	255–224

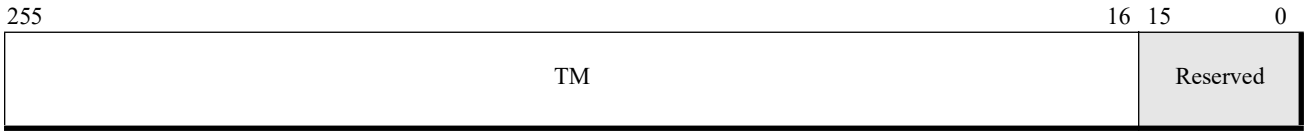


Bits	Mnemonic	Description	Access type
255:16	IS	In Service bits	RO
15:0	Reserved		MBZ

Figure 16-24. In Service Register (APIC Offset 100h–170h)

- *In Service bits (IS)*—Bits 255:16. These bits are set when the corresponding interrupt is being serviced by the CPU core. The in-service registers provide a bit per interrupt to indicate that the corresponding interrupt is being serviced by the CPU core. Interrupts are mapped as follows:

Register	Interrupt Number
ISR (APIC offset 100h)	31–16
ISR (APIC offset 110h)	63–32
ISR (APIC offset 120h)	95–64
ISR (APIC offset 130h)	127–96
ISR (APIC offset 140h)	159–128
ISR (APIC offset 150h)	191–160
ISR (APIC offset 160h)	223–192
ISR (APIC offset 170h)	255–224



Bits	Mnemonic	Description	Access type
255:16	TM	Trigger Mode bits	RO
15:0	Reserved		MBZ

Figure 16-25. Trigger Mode Register (APIC Offset 180h–1F0h)

[AMD Public Use]

- *Trigger Mode bits (TM)*—Bits 255:16. These bits provide a bit per interrupt to indicate the assertion mode of each interrupt. Interrupts are mapped as follows:

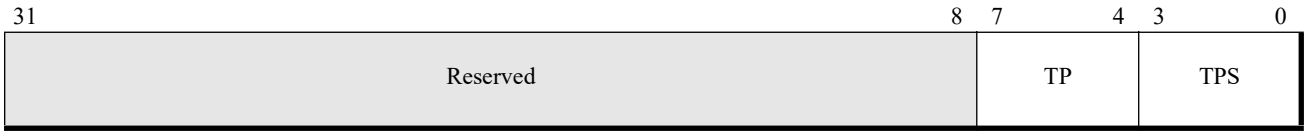
Register	Interrupt Number
TMR (APIC offset 180h)	31–16
TMR (APIC offset 190h)	63–32
TMR (APIC offset 1A0h)	95–64
TMR (APIC offset 1B0h)	127–96
TMR (APIC offset 1C0h)	159–128
TMR (APIC offset 1D0h)	191–160
TMR (APIC offset 1E0h)	223–192
TMR (APIC offset 1F0h)	255–224

16.6.4 Selecting and Handling Interrupts

Interrupts are selected by the local APIC for delivery to the CPU core interrupt handler on a priority determined by the interrupt vector number. Of the 15 priority levels, 15 is the highest and 1 is the lowest. The priority level for an interrupt is equal to the interrupt vector number divided by 16, rounded down to the nearest integer, with vectors 0Fh–00h reserved. Therefore, interrupt vectors 79h and 70h have the same priority level. The high-order hex digit indicates the priority level while the low-order hex digit indicates the priority within the same priority level.

Two registers are used to determine the priority threshold for selecting interrupts to be delivered to the CPU core, the Task Priority Register (TPR) and the Processor Priority Register (PPR). Software uses the TPR to set a priority threshold for interrupts to the CPU core, allowing the OS to block specific interrupts. See Figure 16-26 on page 682 for more details on the TPR.

The value in the *Task Priority* field is set by software to set a threshold priority at which the processor is to be interrupted. The value varies from 0 (all interrupts are allowed) to 15 (all interrupts with fixed delivery mode are inhibited). See Figure 16-26.



Bits	Mnemonic	Description	Access type
31:8	Reserved		MBZ
7:4	TP	Task Priority	R/W
3:0	TPS	Task Priority Sub-class	R/W

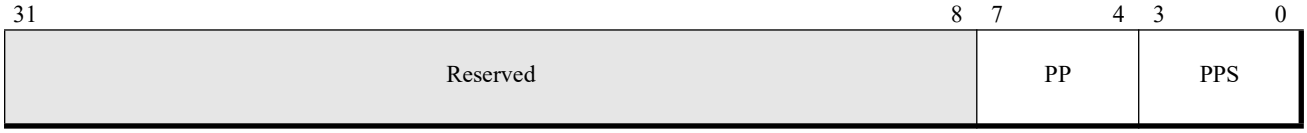
Figure 16-26. Task Priority Register (APIC Offset 80h)

The fields within the Task Priority register are as follows:

[AMD Public Use]

- *Task Priority Sub-class (TPS)*—Bits 3:0. The TPS field indicates the current sub-priority to be used when arbitrating lowest-priority messages. This field is written with zero when TPR is written using the architectural CR8 register.
- *Task Priority (TP)*—Bits 7:4. The TP field indicates the current priority to be used when a core is deciding when to handle interrupts. A value of zero allows all interrupts; a value of Fh disables all interrupts. TP is also used to arbitrate between CPU cores to determine which core accepts a lowest-priority interrupt request. This field can also be written using the architectural CR8 register.

The PPR is set by the CPU core and represents the current priority level at which the CPU core is executing. The PPR determines whether a pending interrupt in the local APIC can be selected for interrupt handling in the CPU core. The value set by hardware is either the interrupt priority level of the highest priority ISR bit set or the value in the TPR, whichever is higher. The PPR is equal to the TPR when the CPU core is not servicing a higher priority interrupt. See Figure 16-27 on page 683.



Bits	Mnemonic	Description	Access type
31:8	Reserved		MBZ
7:4	PP	Processor Priority	RO
3:0	PPS	Processor Priority Sub-class	RO

Figure 16-27. Processor Priority Register (APIC Offset A0h)

The fields within the Processor Priority register are as follows:

- *Processor Priority Sub-class (PPS)*—Bits 3:0. The PPS field is set to the Task Priority sub-class field of the Task Priority Register (TPR) if the PP field is equal to the Task Priority field of the TPR.
- *Processor Priority (PP)*—Bits 7:4. The PP field indicates the CPU core's current priority for servicing a task or interrupt, and is used to determine if any pending interrupts should be serviced. It is the higher value of either the interrupt priority level of the highest priority ISR bit set or the value in the TPR.

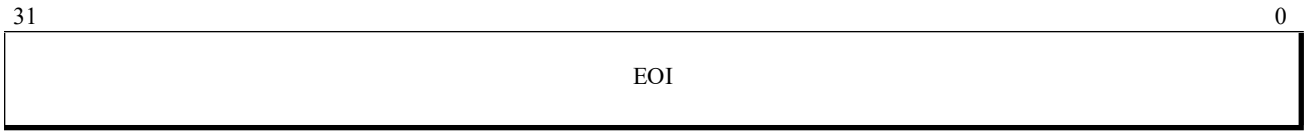
Pending interrupts must have a higher priority level than the value in the PPR to be selected by the local APIC for interrupt handling in the core; otherwise, they remain pending in the IRR until the PPR is lowered below the pending interrupt priority level. No pending interrupts are selected by the local APIC when the TPR=15.

The local APIC selects the highest priority pending interrupt (highest priority IRR) when the CPU core is ready, and sends the interrupt (with the IRR vector) to the CPU core. The local APIC resets the highest priority IRR bit and sets the associated ISR bit.

[AMD Public Use]

As part of the completion of the interrupt handling routine, software writes a value of zero to the End-of-Interrupt Register (EOI) in the local APIC, which causes the local APIC to reset the associated ISR bit. The EOI register is a write-only register.

If a higher priority interrupt is accepted by the local APIC while the CPU core is servicing another interrupt, the higher priority interrupt is sent directly to the CPU core (before the current interrupt finishes processing) and the associated ISR bit is set. The CPU core interrupts the current interrupt handler to service the higher priority interrupt. When the interrupt handler for the higher priority interrupt completes, the associated ISR bit is reset and the interrupt handler returns to complete the previous interrupt handler routine.



Bits	Mnemonic	Description	Access type
31:0	EOI	End of Interrupt	WO

Figure 16-28. End of Interrupt (APIC Offset B0h)

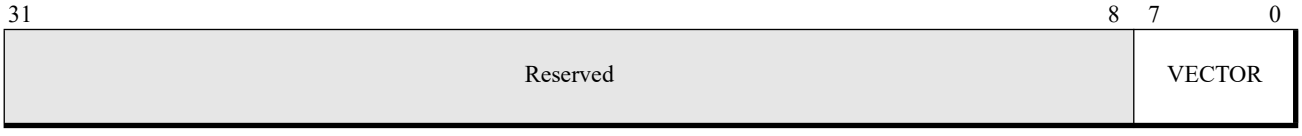
- *End of Interrupt (EOI)*—Bits 31:0. Write-only operation signals end of interrupt processing to source of interrupt.

16.7 SVM Support for Interrupts and the Local APIC

The SVM hypervisor uses the Extended APIC Feature Register, Extended APIC Control Register, Specific End of Interrupt Register (SEOI), and Interrupt Enable Register (IER) to control virtualized interrupts. When guests have direct access to devices, interrupts arriving at the local APIC can usually be dismissed only by the guest that owns the device causing the interrupt. To prevent one guest from blocking other guests' interrupts (by never processing their own), the VMM can mask pending interrupts in the local APIC, so they do not participate in the prioritization of other interrupts.

16.7.1 Specific End of Interrupt Register

Software issues a specific EOI (SEOI) by writing the vector number of the interrupt to the SEOI register in the local APIC. The SEOI register is located at offset 420h in the APIC space. The SEOI register format is shown in Figure 16-29.



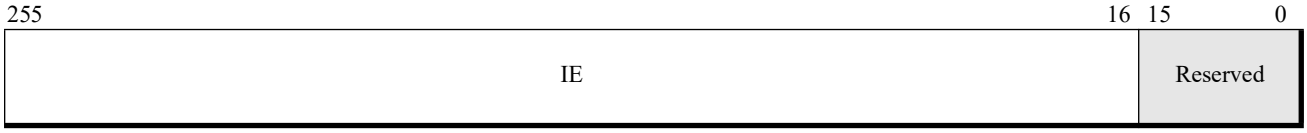
[AMD Public Use]

Bits	Mnemonic	Description	Access type
31:8	Reserved		MBZ
7:0	VECTOR	Vector Number of Interrupt	R/W

Figure 16-29. Specific End of Interrupt (APIC Offset 420h)

16.7.2 Interrupt Enable Register

The IER is made available to software by means of eight 32-bit registers in the local APIC; bit *i* of the 256-bit IER is located at bit position (*i* mod 32) in the local APIC register IER[*i* / 32]. The eight IER registers are located at offsets 480h, 490h, ...,4F0h in APIC space. The IER format is shown in Figure 16-30.



Bits	Mnemonic	Description	Access type
255:16	IE	Interrupt Enable	R/W
15:0	Reserved		MBZ

Figure 16-30. Interrupt Enable Register (APIC Offset 480h–4F0h)

Interrupt Enable (IE)—Bits 255:16. Interrupts are mapped as follows:

Register	Interrupt Number
IER (APIC offset 480h)	31–16
IER (APIC offset 490h)	63–32
IER (APIC offset 4A0h)	95–64
IER (APIC offset 4B0h)	127–96
IER (APIC offset 4C0h)	159–128
IER (APIC offset 4D0h)	191–160
IER (APIC offset 4E0h)	223–192
IER (APIC offset 4F0h)	255–224

The IER and SEOI registers are located in the APIC Extended Space area. The presence of the APIC Extended Space area is indicated by bit 31 of the APIC Version Register (at offset 30h in APIC space).

The presence of the IER and SEOI functionality is identified by bits 0 and 1, respectively, of the APIC Extended Feature Register (located at offset 400h in APIC space). IER and SEOI are enabled by setting bits 0 and 1, respectively, of the APIC Extended Control Register (located at offset 410h).

[AMD Public Use]

Only vectors that are enabled in IER participate in APIC's computation of the highest-priority pending interrupt. The reset value of IER is all ones.

16.8 x2APIC Mode

x2APIC mode is an extension to the local APIC architecture designed to support larger CPU topologies and to enhance delivery of interrupts. When enabled, x2APIC mode adds the following features:

- All APIC and x2APIC registers are accessed via an MSR-based interface.
- The Local APIC ID and Logical APIC ID registers are expanded to 32 bits.
- Logical and physical destinations are extended to 32 bits.
- A new self-IPI register simplifies sending IPIs to self.

In general, x2APIC mode maintains backwards compatibility with the key elements of the local APIC functionality previously described in Section 16. However, the following local APIC functionality is changed in x2APIC mode:

- The Destination Format Register (DFR) is no longer needed and is not supported.
- The two 32-bit Interrupt Command Registers (ICRs) are merged into a single 64-bit ICR.
- Local APIC Base (MSR 01Bh) bit 10, previously reserved, is used to enable x2APIC mode.

16.8.1 x2APIC Terminology

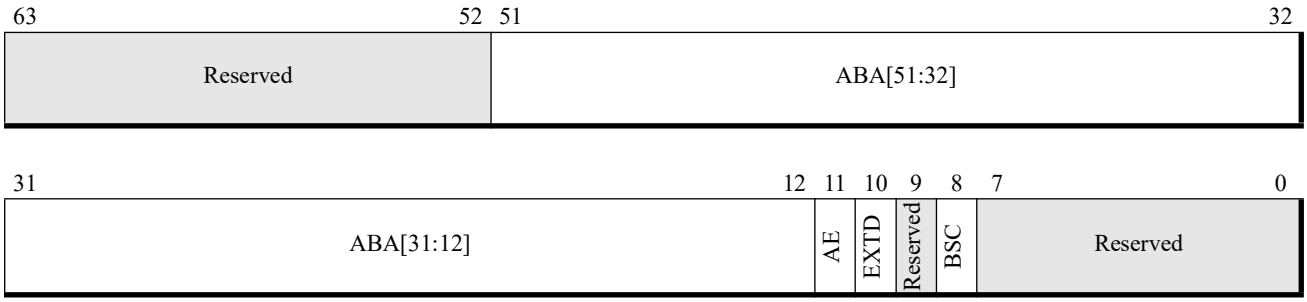
Although x2APIC is an operational mode of the local APIC, the following sections use the term ‘x2APIC’ as a qualifier to more conveniently identify a given aspect of the local APIC when that mode is enabled. For example, the term ‘x2APIC programming interface’ may be used instead of ‘the programming interface when the local APIC is in x2APIC mode’.

16.9 Detecting and Enabling x2APIC Mode

System software can detect the presence of the x2APIC feature by executing the CPUID instruction. Support for x2APIC mode is indicated by CPUID Fn0000_0001_ECX[x2APIC] (bit 21) = 1.

If the feature is present, the local APIC is placed into x2APIC mode by setting bit 10 in the Local APIC Base register (MSR 01Bh). Before entering x2APIC mode, the local APIC must first be enabled (AE=1, EXTD=0). System software can then place the local APIC into x2APIC mode by executing a WRMSR with both AE=1 and EXTD=1. The format of the Local APIC Base register for x2APIC-capable processors is shown in Figure 16-31.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:52	Reserved		MBZ
51:12	ABA	APIC Base Address	R/W
11	AE	APIC Enable (xAPIC mode)	R/W
10	EXTD	X2APIC Mode Enable	R/W
9	Reserved		MBZ
8	BSC	Boot Strap CPU Core	RO
[7:0]	Reserved		MBZ

Figure 16-31. APIC Base Address Register (MSR 01Bh) support for x2APIC

Not all combinations of the local APIC mode bits are valid. See Table 16-5. Attempting to set the combination of AE=0 and EXTD=1 is invalid and causes the WRMSR instruction to generate a #GP(0) exception.

Table 16-5. Local APIC Operating Modes

AE (bit 11)	EXTD (bit 10)	Local APIC Mode
0	0	local APIC disabled
0	1	Invalid
1	0	local APIC enabled in xAPIC mode
1	1	local APIC enabled in x2APIC mode

Similarly, not all transitions between local APIC states are valid. The valid state transitions are illustrated in Figure 16-2. Once the local APIC has been placed into x2APIC mode, the only valid transition (other than reset) is to “APIC Disabled” mode by simultaneously clearing AE and EXTD to zero. Executing a WRMSR instruction to attempt a transition other than those specified in Figure 16-32 results in a #GP(0) exception.

[AMD Public Use]

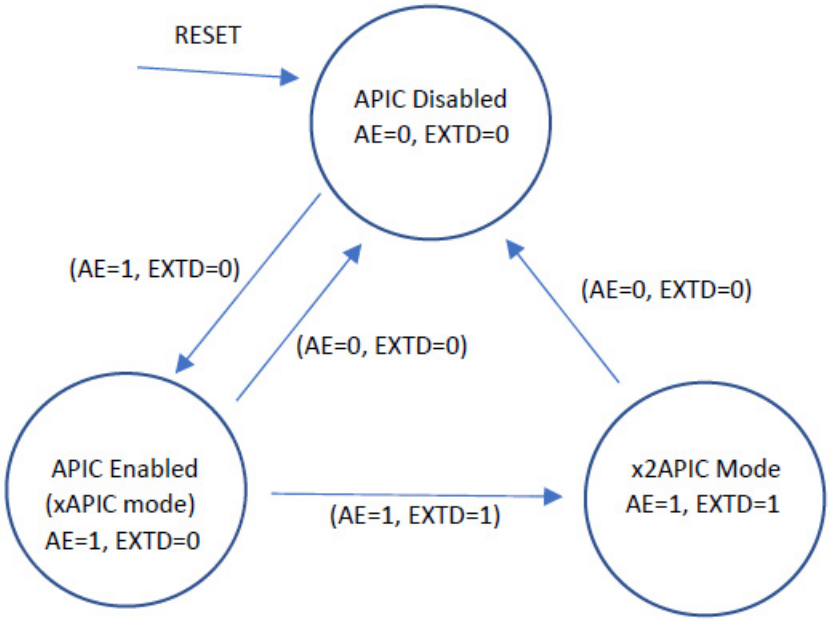


Figure 16-32. APIC State Transitions

16.9.1 Enabling x2APIC Mode

Starting from APIC Disabled mode, enabling x2APIC mode is a two-step process. First, a transition is made to APIC_Enabled mode by setting APIC_Base bit 11 (AE) to 1 with bit 10 (EXTD) left as zero. Next, the transition to x2APIC mode is made by setting both AE and EXTD to 1 simultaneously.

Most APIC registers previously written by software while in APIC_Enabled mode are not affected by the transition to x2APIC mode. The exceptions are:

- The Logical Destination Register (LDR) is not preserved.
- The upper half of the Interrupt Command Register (ICR High) is not preserved.
- A value previously written by software to the 8-bit APIC_ID register (MMIO offset 30h) is converted by hardware into the appropriate format and reflected into the 32-bit x2APIC_ID register (MSR 802h).

Leaving x2APIC mode. Once in x2APIC mode, the only valid mode transition (other than a reset) is to APIC_Disabled mode by using WRMSR to clear APIC_Base bit 11 (AE) bit 10 (EXTD) to 0 at the same time.

[AMD Public Use]

16.10 x2APIC Initialization

A RESET clears the APIC Base Address Register AE bit and EXTD bit, disabling the local APIC. All local APIC registers are initialized to their reset values as described in section 16.3.2 “APIC Registers”.

An INIT does not modify the APIC Base Address Register AE and EXTD bits, thus the local APIC mode is not changed. All other APIC registers are initialized to their values as described in “Reset in x2APIC mode” above.

16.11 Accessing x2APIC Register

The x2APIC system programming interface consists of the Model Specific Registers listed in Table 16-6 below. All APIC registers except the APIC Base Address Register are mapped into the architecturally dedicated MSR range 800h to 8FFh, and accessed using WRMSR and RDMSR instructions.

The RDMSR/WRMSR instructions read and write the MSR specified in the ECX register using the EDX:EAX register pair as the destination and source operand. Bits 31:0 of the APIC register are mapped into EAX[31:0]. For 64-bit x2APIC registers, the high-order bits (bits 63:32) are mapped to EDX[31:0]. A #GP(0) exception is generated if an unimplemented APIC register is specified in ECX. When not in x2APIC mode, attempts to access the APIC register set using the MSR interface results in the WRMSR or RDMSR instruction generating a #GP(0) exception. See APM Volume 3 for more information on the WRMSR and RDMSR instructions.

In x2APIC mode, the legacy MMIO access to the APIC register set is disabled. Attempts to use the legacy MMIO access mechanism may result in an unintended memory access or a memory-related exception such as #GP or #PF.

16.11.1 x2APIC Register Address Space

The x2APIC registers are mapped into the MSR address range 800h through 8FFh. This address range is reserved for accessing the APIC register set in x2APIC mode.

The legacy APIC registers are mapped to this MSR address space using on the following formula:

x2APIC MSR address = 800h + ((APIC MMIO offset) >> 4)

The following registers are exceptions to the above formula:

- The two 32-bit Interrupt Command Registers in APIC mode (MMIO offsets 300h and 310h) are merged into a single 64-bit x2APIC register at MSR address 830h.
- The x2APIC Self_IPI register is added at MSR address 83Fh
- The Destination Format Register (DFR) at MMIO offset E0h and the Remote Read Register (RRR) at MMIO offset C0h are not supported in x2APIC mode. Accordingly, MSR addresses 80Eh and 80Ch are not used and are reserved.

[AMD Public Use]

The system programming interface of the local APIC in x2APIC mode is made up of the MSRs listed in Table 16-6 below.

Table 16-6. x2APIC Register

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	Read/ Write	Notes
802h	20h	x2APIC ID Register [31:0]	RO	Expanded to 32-bits in x2APIC mode
803h	30h	APIC Version Register	RO	
808h	80h	Task Priority Register (TPR)	R/W	
809h	90h	Arbitration Priority Register (APR)	RO	
80Ah	A0h	Processor Priority Register (PPR)	RO	
80Bh	B0h	End of Interrupt Register (EOI)	WO	#GP(0) if non-zero value is written
-	C0h	Remote Read Register	-	Eliminated in x2APIC mode
80Dh	D0h	Logical Destination Register (LDR)	RO	Expanded to 32-bits in x2APIC mode
-	E0h	Destination Format Register	-	Eliminated in x2APIC mode
80Fh	F0h	Spurious Interrupt Vector Register	R/W	
810-817h	100-170h	In-Service Register (ISR)	RO	
818-81Fh	180-1F0h	Trigger Mode Register (TMR)	RO	
820-827h	200-270h	Interrupt Request Register (IRR)	RO	
828h	280h	Error Status Register (ESR)	R/W	#GP(0) if non-zero value is written
830h	300h	Interrupt Command Register (bits 63:0)	R/W	
832h	320h	Timer Local Vector Table Entry	R/W	
833h	330h	Thermal Local Vector Table Entry	R/W	
834h	340h	Perf Counter Local Vector Table Entry	R/W	
835h	350h	Local Interrupt 0 Vector Table Entry	R/W	
836h	360h	Local Interrupt 1 Vector Table Entry	R/W	
837h	370h	Error Vector Table Entry	R/W	
838h	380h	Timer Initial Count Register	R/W	
839h	390h	Timer Current Count Register	RO	
83Eh	3E0h	Timer Divide Configuration Register	R/W	
83Fh	—	Self IPI Register	WO	See Figure 16-36
840h	400h	Extended APIC Feature Register	RO	
841h	410h	Extended APIC Control Register	R/W	
842h	420h	Specific End of Interrupt Register (SEOI)	R/W	
848-84Fh	480-4F0h	Interrupt Enable Registers (IER)	R/W	
850-857h	500-570h	Extended Interrupt [7:0] Local Vector Table Registers	R/W	

[AMD Public Use]

MSR addresses in the range 800h through 8FFh that are not listed in Table 16-2 are unimplemented and reserved. A #GP(0) exception is generated if a WRMSR or an RDMSR instruction attempts to access an unimplemented MSR in the x2APIC address range.

16.11.2 WRMSR / RDMSR serialization for x2APIC Register

The WRMSR instruction is used to write the APIC register set in x2APIC mode. Normally WRMSR is a serializing instruction, however when accessing x2APIC registers, the serializing aspect of WRMSR is relaxed to allow for more efficient access to those registers. Consequently, a WRMSR write to an x2APIC register may complete before older store operations are complete and have become globally visible. When strong ordering of an x2APIC write access is required with respect to preceding memory operations, software can insert a serializing instruction (such as MFENCE) before the WRMSR instruction.

The RDMSR instruction is not a serializing instruction and remains non-serializing when reading x2APIC MSRs. However, WRMSR and RDMSR instructions targeting the x2APIC MSRs are always executed in program order with respect to each other.

16.11.3 Reserved Bit Checking in x2APIC Mode

When writing x2APIC MSRs, the WRMSR instruction checks for reserved bits. Attempting to write a ‘1’ to a reserved bit causes a #GP(0) exception. For x2APIC MSRs, WRMSR reserved bit checks are summarized as follows:

Legacy APIC registers. Reserved bit checks for existing APIC registers are the same as described for each register in non-x2APIC mode. For details, see the APIC register descriptions in sections 16.3 through section 16.6 above. Except for the Interrupt Command Register, attempting to write a ‘1’ into bits 63:32 of the legacy APIC registers causes a #GP(0) exception.

Interrupt Command Register (ICR). See the description of the 64-bit ICR register in section 16.13.

Error Status Register (ESR). A WRMSR of a non-zero value causes a #GP(0) exception.

SELF IPI register. See the description of the 32-bit SELF IPI register in Section 16.15 on page 694.

The RDMSR instruction returns a zero for any reserved bit.

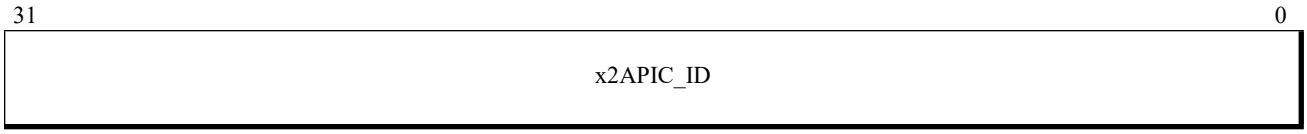
16.12 x2APIC_ID

Unique local APIC IDs are assigned to each logical processor in the system. In x2APIC mode, the APIC ID is expanded to 32 bits and is referred to as the ‘x2APIC_ID’. It is assigned by hardware at reset time based on the processor topology of the system. The x2APIC_ID is a concatenation of several fields such as socket ID, core ID and thread ID.

Because the number of sockets, cores and threads may differ for each SOC, the format of x2APIC ID is model-dependent. Some fields may not be present, depending on the processor model and the

[AMD Public Use]

processor topology. The presence, size and position of each field is discoverable using the CPUID instruction (see “Cache and Processor Topology” on page 214).



Bits	Mnemonic	Description	Access type
31:0	x2APIC_ID	x2APIC ID	RO

Figure 16-33. x2APIC_ID Register (MSR 802h)

System software can read x2APIC_ID using either of the following mechanisms:

RDMSR. An RDMSR of MSR 0802h returns the x2APIC_ID in EAX[31:0]. The x2APIC_ID is a read-only register. Attempting to write MSR 802h or attempting to read this MSR when not in x2APIC mode causes a #GP(0) exception. See 16.11 “Accessing x2APIC Registers”.

CPUID. The x2APIC ID is reported by CPUID functions Fn0000_000B (Extended Topology Enumeration) and CPUID Fn8000_001E (Extended APIC ID) as follows:

- Fn0000_000B_EDX[31:0]_x0 reports the full 32-bit ID, independent of APIC mode (i.e. even with APIC disabled)
- Fn8000_001E_EAX[31:0] conditionally reports APIC ID. There are 3 cases:
  - 32-bit x2APIC_ID, in x2APIC mode.
  - 8-bit APIC ID (upper 24 bits are 0), in xAPIC mode.
  - 0, if the APIC is disabled.

The above CPUID functions also report the presence, width and location of the sub-fields comprising the x2APIC ID. See APM Volume 3 Appendix E, “Obtaining Processor Information Via the CPUID Instruction” for detailed information.

16.13 x2APIC Interrupt Command Register (ICR) Operations

In legacy APIC mode, two 32-bit registers (ICR Low and ICR High) are used by system software to send Inter-Processor Interrupts (IPIs) to other local APICs. The x2APIC architecture combines these two registers into a single 64-bit Interrupt Command Register located at MSR address 830h. Thus in x2APIC mode sending an IPI requires only a single WRMSR to the ICR as opposed to two MMIO accesses in xAPIC mode.

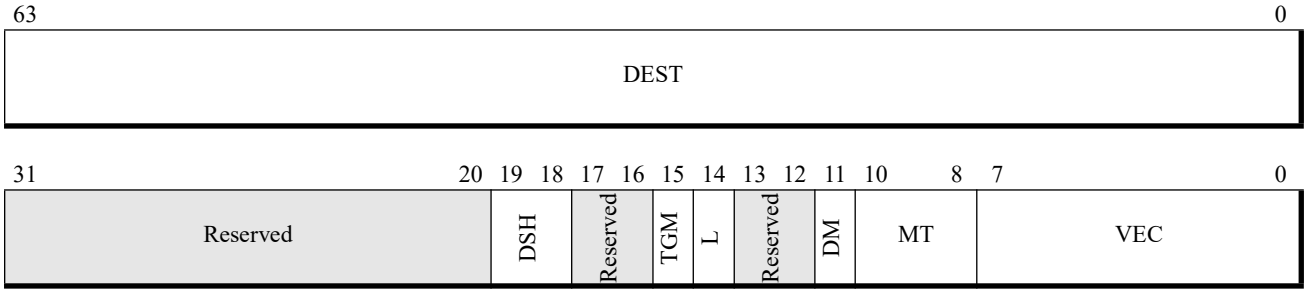
The upper half of the x2APIC ICR (bits 63:32) contains the Destination ID (DEST) field, which is expanded to 32 bits in x2APIC mode. A DEST value of FFFF_FFFFh is used to broadcast IPIs to all local APICs.

[AMD Public Use]

The lower half of the x2APIC ICR (bits 31:0) is identical to the APIC Interrupt Command Register Low[31:0] (see Fig 16-18 on page 582), with the following exceptions:

- The Remote Read Status field (bits 17:16) is eliminated and must be zero.
- Message Type field (bits 10:8). Encodings 1, 3 and 7 are eliminated and the encodings are reserved.
- The Delivery Status field (bit 12) is eliminated and must be zero.

The format of the x2APIC ICR register is shown in Figure 16-34.



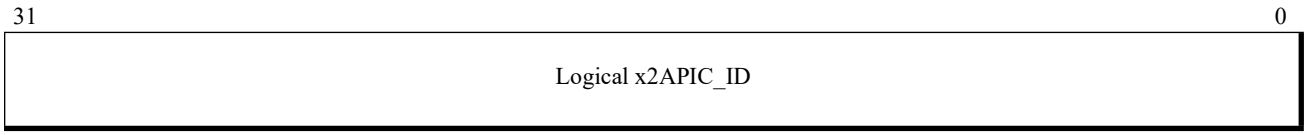
Bits	Mnemonic	Description	Access type
63:32	DEST	Destination	R/W
55:20	Reserved		MBZ
19:18	DSH	Destination Shorthand	R/W
17:16	Reserved		MBZ
15	TMG	Trigger Mode	R/W
14	L	Level	R/W
13:12	Reserved		MBZ
11	DM	Destination Mode	R/W
10:8	MT	Message Type	R/W
7:0	VEC	Vector	R/W

Figure 16-34. Interrupt Command Register (MSR 830h)

16.14 Logical Destination Register

When an IPI is sent using logical destination mode, all local APICs in the system use the Logical Destination Register to determine if the interrupt message is directed to them (see 16.6.1 Receiving System and IPI Interrupts). In x2APIC mode, the Logical Destination Register (LDR) is expanded to 32 bits and contains the ‘logical x2APIC_ID’. System hardware initializes LDR with the 32-bit logical x2APIC_ID whenever x2APIC mode is enabled. The LDR is a read-only register located at MSR address 080Dh. The format of the Logical Destination Registers is shown in Figure 16-35.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
31:0	x2Logical_ID	Logical x2APICID Identifier	RO

Figure 16-35. Logical Destination (MSR 80Dh)

The logical x2APIC_ID consists of two 16-bit sub-fields: cluster_ID and logical_ID.

- LDR[31:16] cluster_ id. Identifies the cluster of which this processor is a member.
- LDR[15:0] logical_id. A bit vector uniquely identifying this processor within the cluster.

In logical destination mode, a given logical processor is addressed by its unique cluster ID and logical_ID combination. The use of a bit vector for logical_ID allows an interrupt message to be routed to multiple processors within the addressed cluster.

The partitioning of logical x2APIC_ID provides for a possible 65,535 (216-1) clusters, with each cluster having up to 16 logical processors. The legacy “flat logical” addressing is not supported in x2APIC mode.

Upon receiving an interrupt message in logical destination format, each x2APIC compares bits 31:16 of the message destination with LDR[31:16] (cluster_id). If there is a match, then bits 15:0 of the destination and LDR[15:0] are tested for matching ones. If bits[31:16] (cluster_id) match and any bit in 15:0 (logical_id) match, this x2APIC is a valid destination.

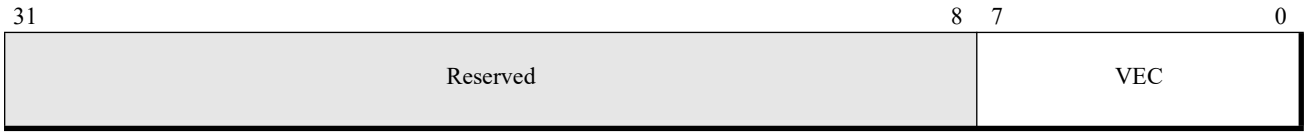
A DEST value of FFFF_FFFFh in the ICR is used to broadcast IPIs to all local APICs.

The two sub-fields comprising logical x2APIC_ID are derived from the value of local x2APIC_ID. The 16-bit logical_ID sub-field is initialized by setting a single bit, ‘n’, where n= the 4 least significant bits of local x2APIC_ID. The 16-bit cluster_id is derived from the remaining bits of the x2APIC_ID. Specifically, logical_id[15:0] = 1 << x2APIC_ID[3:0] and cluster_id[15:0] = x2APIC_ID[19:4].

16.15 Self_IPI Register

The Self_IPI register (MSR 83Fh) provides a performance-optimized interface for system software to send interrupt messages to the local APIC. This register is write-only and attempts to read it cause a #GP(0) exception. The format of the Self_IPI registers is shown in Figure 16-36.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
31:8	Reserved		MBZ
7:0	VEC	Interrupt Vector	WO

Figure 16-36. Self_IPI Register (MSR 83Fh)

The Self_IPI register contains a single field: an interrupt vector. Writing to this register causes a to-self IPI to be generated, equivalent to a to-self IPI generated by writing the Interrupt Command Register (ICR, MSR 830h) with the following settings:

- Destination shorthand = self
- Trigger Mode = edge-triggered
- Message Type = fixed
- Vector = interrupt vector as specified in the Self_IPI register

The x2APIC's response to a self-IPI sent via the Self_IPI register is architecturally identical to one sent via the ICR. In particular, the operation of the Interrupt Response Register (IRR), In-Service Register (ISR) and Trigger Mode Register (TMR) is the same. See "Accepting System and IPI Interrupts" on page 679 for IRR, ISR and TMR details.

The Interrupt Request Register (IRR) contains interrupt requests that have been accepted by the processor core. Completion of the WRMSR to the Self_IPI register ensures that the resulting IPI has been entered into the IRR, and that the associated TMR bit is cleared (as expected for edge-triggered interrupts).

[AMD Public Use]

17 Hardware Performance Monitoring and Control

The AMD64 architecture provides several mechanisms by which software can monitor and control processor performance to optimize power use. The following lists the facilities that are described in the sections that follow:

- The P-state control interface allows dynamic control of performance states. See Section 17.1 which follows immediately below.
- Core performance boost (CPB) dynamically increases core clock rate beyond that defined for the P0 power state to achieve higher performance while maintaining power consumption below a preset level. See Section 17.2 on page 698.
- The effective frequency interface provides a measure of the actual core clock rate over a specified period of time. See Section 17.3 on page 699.
- The processor power reporting interface allows system software to measure average processor core power over a given time period. See Section 17.5 on page 701.
- Collaborative Processor Performance Control (CPPC) provides a mechanism for improving performance and power efficiency in collaboration with Operating System Power Management software. See Section 17.6 on page 703.

17.1 P-State Control

P-states are operational performance states (states in which the processor is executing instructions, that is, running software) characterized by a unique frequency of operation for a CPU core. The P-state control interface supports dynamic P-state changes in up to 16 P-states, called P-states 0 through 15 or P0 though P15. P0 is the highest power, highest performance P-state; each ascending P-state number represents a lower-power, lower-performance state.

Core P-states are controlled by software. Each CPU core contains one set of P-state control registers. Software controls the P-states of each CPU core independently; however, hardware may include interdependencies that affect the P-state achieved by each core.

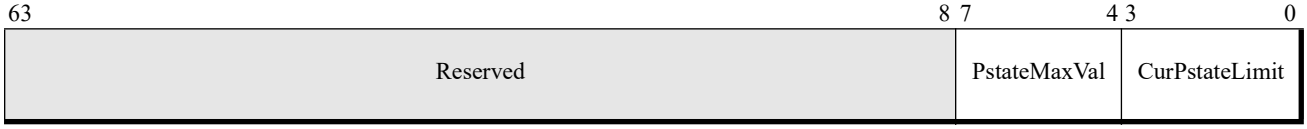
Hardware provides the highest P-state value in the PstateMaxVal field of the P-State Current Limit Register. P-states may be limited to a lower performance value under certain conditions. The current P-state limit is dynamic and is specified in the CurPstateLimit field of the P-State Current Limit Register.

Software requests a core P-state change by writing a 4-bit index corresponding to the desired core P-state number to the P-State Control Register of the appropriate core. For example, to request the P3 state for core 0, software writes 3h to the core 0's PstateCmd field in MSR C001_0062h. If the P-state value is greater than the value in PstateMaxVal, the value written is clipped to that value.

[AMD Public Use]

As the current P-state limit changes, the P-state for the CPU core is either set to the software-requested P-state value or the new current P-state limit, whichever is the higher P-state value.

The current P-state value can be read using the P-State Status Register. The P-State Current Limit Register and the P-State Status Register are read-only registers. Writes to these registers cause a #GP exception. Support for hardware P-state control is indicated by CPUID Fn8000_0007_EDX[HwPstate] = 1. Figure 17-1 below shows the format of the P-State Current Limit register.

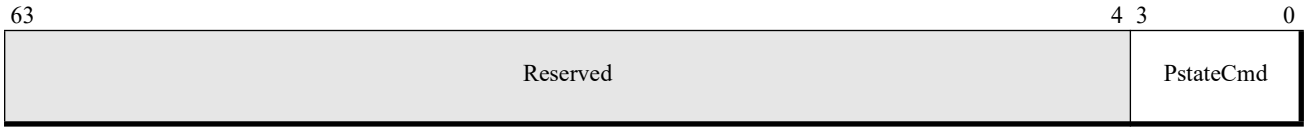


Bits	Mnemonic	Description	Access type
63:8	Reserved		MBZ
7:4	PstateMaxVal	P-state maximum value	RO
3:0	CurPstateLimit	Current P-state limit	RO

Figure 17-1. P-State Current Limit Register (MSR C001_0061h)

The fields within the P-State Current Limit register are:

- *Current P-State Limit (CurPstateLimit)*—Bits 3:0. Provides the current P-state limit, which is the lowest P-state value (highest-performance state) that is currently supported by the hardware. This is a dynamic value controlled by hardware. Reset value is implementation specific.
- *P-State Maximum Value (PstateMaxVal)*—Bits 7:4. Specifies the highest P-state value (lowest performance state) supported by the hardware. Attempts to change the current P-state number to a higher value by writes to the P-State Control Register are clipped to the value of this field. Reset value is implementation specific.

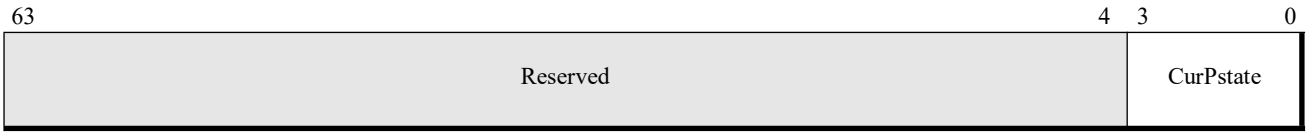


Bits	Mnemonic	Description	Access type
63:4	Reserved		MBZ
3:0	PstateCmd	P-state change command	R/W

Figure 17-2. P-State Control Register (MSR C001_0062h)

P-State Change Command (PstateCmd)—Bits 3:0. Writes to this field cause the CPU core to change to the indicated P-state number, which may be clipped by the PstateMaxVal field of the P-State Current Limit Register. Reset value is implementation specific.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:4	Reserved		MBZ
3:0	CurPstate	Current P-state	RO

Figure 17-3. P-State Status Register (MSR C001_0063h)

Current P-State (CurPstate)—Bits 3:0. This field provides the current P-state of the CPU core regardless of the source of the P-state change, including writes to the P-State Control Register: 0 = P-state 0, 1 = P-state 1, etc. The value of this field is updated when the frequency transitions to a new value associated with the P-state. Reset value is implementation specific.

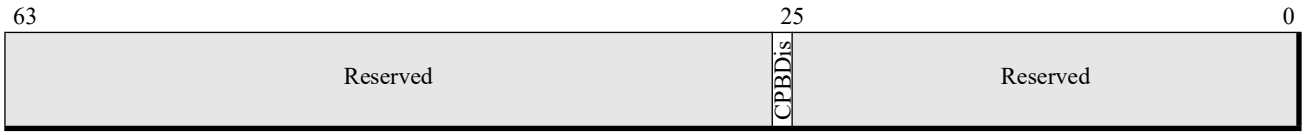
17.2 Core Performance Boost

Core performance boost (CPB) dynamically monitors processor activity to create an estimate of power consumption. If the estimated processor consumption is below an internally defined power limit and software has requested P0 on a given core, hardware may transition the core to a frequency and voltage beyond those defined for P0. If the estimated power consumption exceeds the defined power limit, some or all cores are limited to the frequency and voltage defined by P0. CPB ensures that average power consumption over a thermally significant time period remains at or below the defined power limit.

CPB can be disabled using the CPBDis field of the Hardware Configuration Register (HWCR MSR) on the appropriate core. When CPB is disabled, hardware limits the frequency and voltage of the core to those defined by P0.

Support for core performance boost is indicated by CPUID Fn8000_0007_EDX[CPB] = 1. See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

[AMD Public Use]



Bits	Mnemonic	Description	Access type
63:26	Reserved		MBZ
25	CPBDis	Core Performance Boost Disable	R/W
24:0	Reserved		MBZ

Figure 17-4. Core Performance Boost (MSR C001_0015h)

Core Performance Boost Disable (CpbDis)—Bit 25. Specifies whether core performance boost is enabled or disabled. 0 = Enabled. 1 = Disabled.

17.3 Determining Processor Effective Frequency

The Max Performance Frequency Clock Count (MPERF) and the Actual Performance Frequency Clock Count (APERF) registers constitute the effective frequency interface. This interface provides a means for software to calculate an average, or effective, frequency of a core over a known window of time. This provides software a measure of actual performance rather than forcing software to assume that the current frequency of the core is the frequency of the last P-state requested.

To calculate an effective clock frequency of a given processor core, on that processor do the following:

1. Read both MPERF and APERF and save their initial values.
   - MPERF_INIT = MPERF and APERF_INIT = APERF
2. Wait an appropriate amount of time.
3. Read both MPERF and APERF again.
4. Effective frequency = {(APERF − APERF_INIT) / (MPERF − MPERF_INIT)} * P0 frequency.

The amount of time that elapses between steps 1 and 3 is determined by software. This allows software to define the time window over which the processor frequency is averaged. Software should disable interrupts or any other events that may occur between the read of MPERF and the read of APERF in step 1 and again when the two MSRs are read in step 3. Step 4 provides the equation for the calculation of the effective frequency value. Software determines the P0 frequency using ACPI defined data structures.

The effective frequency interface only counts clock cycles while the core is in the ACPI defined C0 state.

Only the ratio between MPERF and APERF is architecturally defined. Software should not assume any specific definition of the MPERF or APERF registers. If an overflow of either the MPERF or

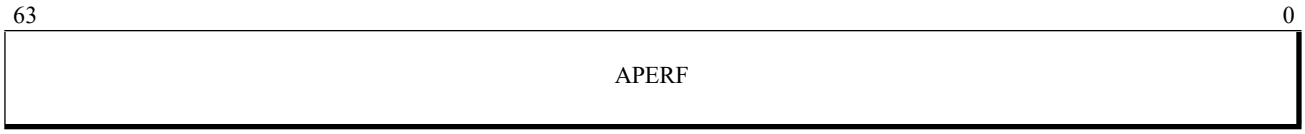
[AMD Public Use]

APERF register occurs between the read of MPERF in step 1 and the read of APERF in step 3, the effective frequency calculated in step 4 is invalid.

Hardware support for the effective frequency interface is indicated by CPUID Fn0000_0006_ECX[EffFreq]. See Section 3.3, “Processor Feature Identification,” on page 73 for more information on using the CPUID instruction.

17.3.1 Actual Performance Frequency Clock Count (APERF)

Specifies the numerator of the effective frequency ratio.

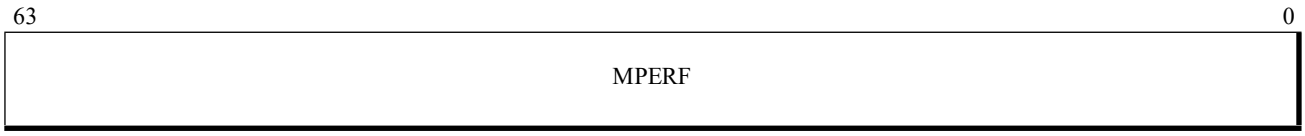


Bits	Mnemonic	Description	Access Type
63:0	APERF	Actual Performance Frequency Clock Count	R/W

Figure 17-5. Actual Performance Frequency Count (MSR0000_00E8h)

17.3.2 Maximum Performance Frequency Clock Count (MPERF)

Specifies the denominator of the effective frequency ratio. The value read is scaled by the TSCRatio value (MSR C000_0104h) for guest reads, but the underlying counters are not affected. Reads in host mode or writes to MPERF are not affected.



Bits	Mnemonic	Description	Access Type
63:0	MPERF	Max Performance Frequency Clock Count	R/W

Figure 17-6. Max Performance Frequency Count (MSR0000_00E7h)

[AMD Public Use]

17.3.3 APERF Read-only (AperfReadOnly)

Specifies the numerator of the effective frequency ratio.



Bits	Mnemonic	Description	Access Type
63:0	APERF_RD_ONLY	APREF Read Only	RO

Figure 17-7. APREF Read Only (MSR C000_00E8h)

17.3.4 MPERF Read-only (MperfReadOnly)

Read-only version of MPERF. The value read is scaled by the TSCRatio value (MSR C000_0104h) for guest reads.



Bits	Mnemonic	Description	Access Type
63:0	MPERF_RD_ONLY	MPERF Read Only	RO

Figure 17-8. MPERF Read Only (MSR C000_00E7h)

17.4 Processor Feedback Interface

The Processor Feedback Interface is deprecated. Some processor products may support this feature. To determine support on a given processor, software can test the feature bit CPUID Fn8000_0007_EDX[ProcFeedbackInterface]. For more information, consult the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

17.5 Processor Core Power Reporting

The processor power reporting interface allows system software to estimate the average power consumed by a processor core over a software-determined time period. Computing the average power involves reading a "core power accumulator" register at the beginning and end of the measurement interval, taking the difference and then dividing by the length of the time interval.

Support for the processor power reporting interface is indicated by CPUID Fn8000_0007_EDX[ProcPowerReporting] = 1.

[AMD Public Use]

17.5.1 Processor Facilities

Estimating core average power involves the use of several processor facilities. Processors that support the processor power reporting interface define the following three facilities:

- CpuSwPwrAcc MSR
- MaxCpuSwPwrAcc MSR
- CpuPwrSampleTimeRatio (CPUID Fn8000_0007_ECX)

A fourth facility, available on all processors, is the time-stamp counter (TSC). The TSC is a free-running counter that increments on every processor clock cycle. The current value o f this counter is read using the RDTSC instruction.

The contents of the CpuSwPwrAcc register represents the cumulative *energy* consumed by the core. Each hardware-determined sample period (Tsample) a value that represents the energy consumed since the previous sample is added to the contents of this register. Tsample is on the order of a few microseconds. The exact value is immaterial because the CpuPwrSampleTimeRatio register provides the ratio of Tsample to the TSC period.

CpuSwPwrAcc is cleared to zero at power-on and is never reset. Therefore, it is possible for this counter to overflow and roll over to zero. To account for this, the interface provides the MaxCpuSwPwrAcc register. When read, this register provides a value that represents the maximum energy that the CpuSwPwrAcc register can report.

17.5.2 Software Algorithm

The following algorithm should be used to calculate the average power consumed by a processor core during the measurement interval T_M. To obtain a stable average power value, T_M should be on the order of several milliseconds.

- Determine the value of the ratio of Tsample to the TSC period (CpuPwrSampleTimeRatio) by executing CPUID Fn8000_0007. Call this value *N*.

  N = CPUID Fn8000_0007_ECX[31:0].
- Read the full range of the cumulative energy value from the register MaxCpuSwPwrAcc.

  Jmax = value returned from RDMSR MaxCpuSwPwrAcc.
- At time *x*, read CpuSwPwrAcc and the TSC

  J_x = value returned by RDMSR CpuSwPwrAcc

  T_x = value returned by RDTSC
- At time *y*, read CpuSwPwrAcc and the TSC again

  J_y = value returned by RDMSR CpuSwPwrAcc

  T_y = value returned by RDTSC

Calculate the average power consumption for the processor core over the measurement interval $T_M = (Ty - Tx)$.

[AMD Public Use]

- If ($J_y < J_x$), rollover has occurred; set $Jdelta = (J_y + Jmax) - J_x$
  else $Jdelta = J_y - J_x$
- PwrCPUave = $N * Jdelta / (T_y - T_x)$

Units of result is milliwatts.

17.6 Collaborative Processor Performance Control

AMD's Collaborative Processor Performance Control (CPPC) feature is an implementation of the Collaborative Processor Performance Control standard, first introduced in the Advanced Configuration and Power Interface (ACPI) Specification Revision 5.0. CPPC provides a framework for managing the performance and power efficiency of SOCs in collaboration with Operating System Power Management (OSPM) software.

The CPPC feature provides a mechanism for the OSPM to manage the performance of a logical processor utilizing a continuous and abstract performance scale. CPPC implements a set of MSRs to enumerate the abstract performance scale, to request processor performance levels using that scale, and to provide feedback to the OSPM.

17.6.1 Detecting Support for CPPC

Support for CPPC is indicated by CPUID Fn8000_0008_EBX[CPPC](bit 27)=1.

17.6.2 CPPC Model Specific Registers

Various aspects of the CPPC are managed though the model-specific registers listed in Table 17-1 below. These MSRs are used by the OSPM to obtain CPPC capabilities, control CPPC operations and obtain feedback on delivered performance.

Table 17-1. CPPC-related MSRs

Register Name	MSR Address	Reference	Description
CPPC_CAPABILITY_1	C001_02B0h	Figure 17-10	Specifies the CPPC performance ranges and thresholds.
CPPC_ENABLE	C001_02B1h	Figure 17-9	Enables CPPC.
CPPC_CAPABILITY_2	C001_02B2h	Figure 17-11	Reports the constrained maximum performance level.
CPPC_REQUEST	C001_02B3h	Figure 17-12	Conveys OSPM requests and hints to the CPPC hardware.
CPPC_STATUS	C001_02B4h	Figure 17-14	Reports excursions to the minimum performance level.
CPPC_REQUEST_2	C001_02B5h	Figure 17-13	Conveys Performance Priority requests to the CPPC hardware. Supported when CPUID Fn8000_0007_EDX[CppcPerfPriority] (bit 16) = 1

[AMD Public Use]

17.6.3 Enabling CPPC

Software may enable Collaborative Processor Performance Control by setting CPPC_ENABLE[CPPC_En] (bit 0) = 1. Enabling CPPC on any one logical processor in a package enables it for all processors within that package.

When CPPC is enabled, the hardware calculates the processor's performance capabilities and initializes the performance level fields in the CPPC capability registers.

The processor ignores the P-state control interface (described in "P-State Control" on page 696) when CPPC is enabled.

17.6.3.1 CPPC_ENABLE



Bits	Mnemonic	Description	Access Type
63:1	Reserved		MBZ
0	CPPC_En	CPPC enable	R/W1

Figure 17-9. CPPC_ENABLE

The CPPC_ENABLE register fields are further described below:

- *CPPC_En (CPPC Enable)*—Bit 0, read/write once. Software sets this bit to enable Collaborative Processor Performance Control. Once CPPC is enabled the processor ignores further writes to this register. CPPC_En can only be cleared by reset.

17.6.4 CPPC Performance Levels and Capabilities

The OSPM can read the CPPC_CAPABILITY_1 and CPPC_CAPABILITY_2 MSRs to determine the performance levels and thresholds supported by the processor. These values are reported on a continuous scale of abstract values ranging from 00h (minimum) to FFh (maximum).

Prior to enabling CPPC, the CPPC_CAPABILTY_1 and CPPC_CAPABILITY_2 registers are initialized to zero. When CPPC is enabled, the hardware calculates the processors' performance capabilities and stores the appropriate values in both CPPC capability registers.

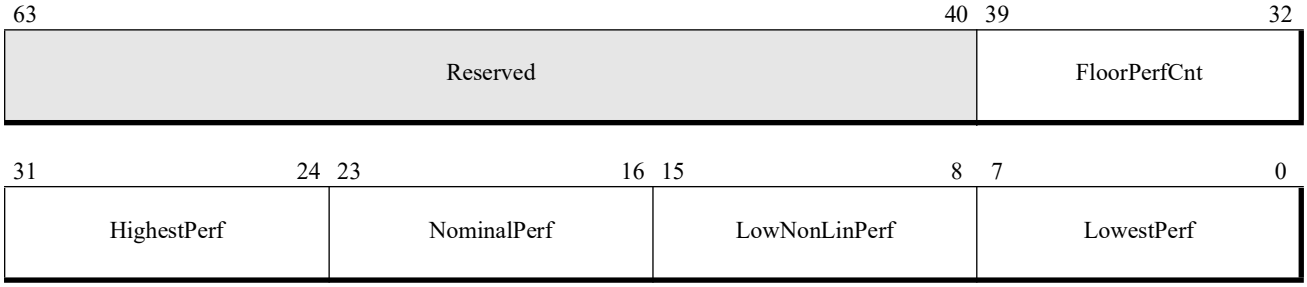
The reported performance levels directly map to processor frequency; however, this mapping is specific to a processor family and implementation.

The CPPC_CAPABILITY_1 and CPPC_CAPABILITY_2 registers are implemented on a per-logical processor basis. Each logical processor contains an independent set of CPPC capability registers.

[AMD Public Use]

17.6.4.1 CPPC_CAPABILITY_1

The CPPC_CAPABILITY_1 register reports the processor performance levels and thresholds.



Bits	Mnemonic	Description	Access Type
63:40	Reserved		RAZ
39:32	FloorPerfCnt	Floor Performance Count	RO
31:24	HighestPerf	Highest Performance - The absolute maximum performance an individual processor may reach.	RO
23:16	NominalPerf	Nominal (Guaranteed) Performance - The maximum sustained performance level of the processor.	RO
15:8	LowNonLinPerf	Lowest Non-Linear Performance is the lowest performance level at which nonlinear power savings are achieved.	RO
7:0	LowestPerf	Lowest performance is the absolute lowest performance level of the processor.	RO

Figure 17-10. CPPC_CAPABILITY_1

The CPPC_CAPABILITY_1 register fields are further described below:

- *FloorPerfCnt (Floor Performance Count)*—Bits 39:32. This field reports the number of distinct Floor Perf values supported by the platform. For example, if this value is 2, then the number of unique values of CPPC_REQUEST_2[FloorPerf] across the system should be at most 2 for the throttling behavior described for CPPC_REQUEST_2[FloorPerf] to take effect. A zero value implies that the platform supports unlimited Floor Perf levels. When the Floor Performance Count is non-zero, then the throttling behavior is undefined if the number of unique values of Floor Perf on the system exceeds Floor Perf Count. This field is supported when CPUID Fn8000_0007_EDX[CppcPerfPriority] (bit 16) = 1.
- *HighestPerf[7:0] (Highest Performance)*—Bits 31:24. Reports the maximum performance an individual processor can reach under ideal conditions. This performance level is not guaranteed due to system thermal and power constraints and may not be sustainable for long durations. The Highest Performance level may require that other processors be in a lower performance state.
- *NominalPerf[7:0] (Nominal Performance)*—Bits 23:16. Reports the maximum sustained performance level of the processor, assuming ideal operating conditions. In the absence of an unexpected system power or thermal constraint, the processor can maintain the nominal performance level continuously. All processors can sustain the Nominal Performance state simultaneously.

[AMD Public Use]

- *LowNonLinPerf[7:0] (Lowest Non-Linear Performance)*—Bits 15:8. Reports the most energy efficient performance level (in terms of performance per watt). Above this threshold, lower performance levels generally result in increased energy efficiency. Reducing performance below this threshold does not result in total energy savings for a given computation, although it reduces instantaneous power consumption.
- *LowestPerf[7:0] (Lowest Performance)*—Bits 7:0. Reports the lowest performance level of the processor.

17.6.4.2 CPPC_CAPABILITY_2

The CPPC_CAPABILITY_2 register reports the constrained maximum performance level of the processor.



Bits	Mnemonic	Description	Access Type
63:8	Reserved		RAZ
7:0	MaxPerf	Constrained Maximum Performance	RO

Figure 17-11. CPPC_CAPABILITY_2

The CPPC_CAPABILITY_2 register fields are further described below:

- *MaxPerf[7:0] (Constrained Maximum Performance)*—Bits 7:0. Reports the current maximum performance level considering all known external constraints (i.e., power limits, thermal limits, AC/DC power source, etc.).

17.6.5 Using CPPC to Manage Processor Performance

The OSPM controls CPPC operation by specifying processor performance goals and hints using the CPPC_REQUEST register. The values written to CPPC_REQUEST use the abstract, continuous performance scale described in Section 17.6.4 on page 704.

CPPC supports two modes of operation – *directed* and *autonomous*. In the directed mode of operation, the OSPM requests explicit performance targets by writing the CPPC_REQUEST[DesPerf] field. The CPPC hardware translates these requests into actual performance and power states (core frequency, data fabric and memory clocks, etc). DesPerf may be set to any performance value between CPPC_REQUEST[MinPerf,MaxPerf], inclusive.

[AMD Public Use]

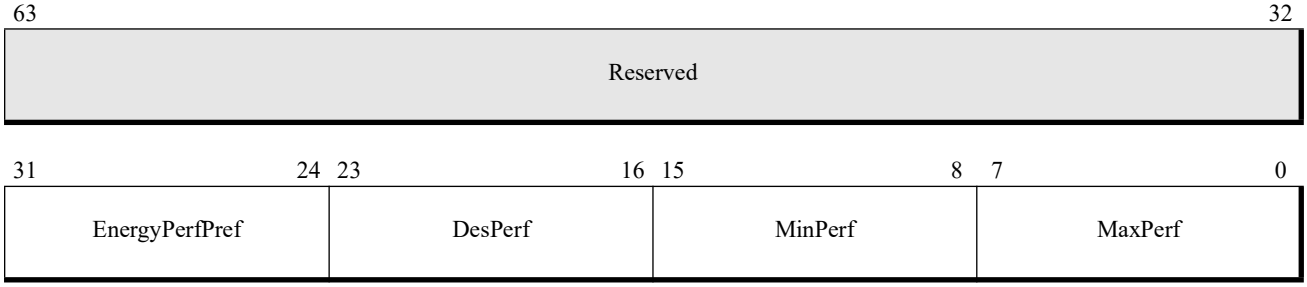
The effective performance delivered by the CPPC hardware is guided by hints provided by the OSPM. These hints include minimum and maximum performance limits, plus a preference towards energy efficiency or performance. The OSPM programs these hints into the MinPerf, MaxPerf, and EnergyPerfPref fields of the CPPC_REQUEST register.

Autonomous mode is selected by writing zero to CPPC_REQUEST[DesPerf]. In autonomous mode, the hardware independently selects a performance level appropriate to the current workload. As with the directed mode, the OSPM provides hints to guide the hardware's performance level selection, including minimum and maximum performance limits and a preference for optimizing for energy efficiency versus performance.

Processors that set CPUID Fn8000_0007_EDX[CppcPerfPriority] (bit 16) = 1 support CPPC Performance Priority. With CPPC Performance Priority software can specify different performance floor frequencies using CPPC_REQUEST_2[FloorPerf] on different cores which the platform firmware considers when throttling CPU frequency due to power or thermal constraints. Platform firmware reports the number of unique CPPC_REQUEST_2[FloorPerf] values across the system in CPPC_CAPABILITY_1[FloorPerfCnt].

17.6.5.1 CPPC_REQUEST

The CPPC_REQUEST register is used by the OSPM to convey performance requests and hints to the CPPC hardware.



Bits	Mnemonic	Description	Access Type
63:32	Reserved		MBZ
31:24	EnergyPerfPref	Energy Performance Preference	RW
23:16	DesPerf	Desired Performance	RW
15:8	MinPerf	Minimum Performance	RW
7:0	MaxPerf	Maximum Performance	RW

Figure 17-12. CPPC_REQUEST

The CPPC_REQUEST register fields are further described below:

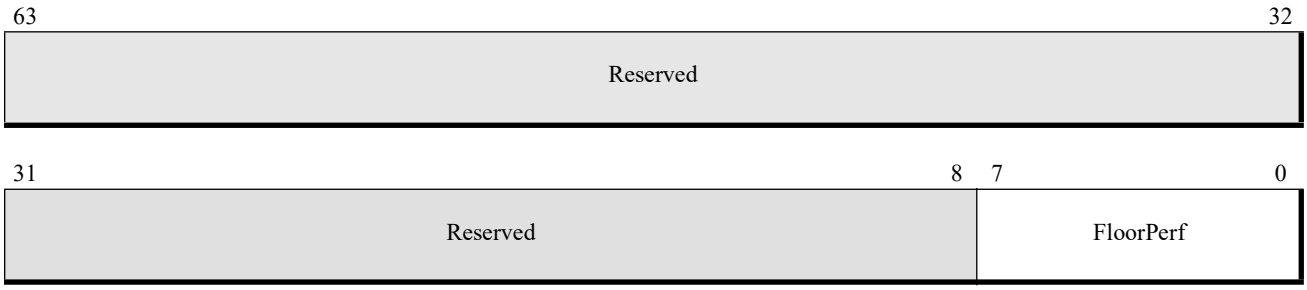
- *EnergyPerfPref[7:0] (Energy Performance Preference hint)*—Bits 31:24. Provides a hint to the hardware if software wants to bias the processor towards performance (00h) or towards energy efficiency (FFh).

[AMD Public Use]

- *DesPerf[7:0] (Desired Performance hint)*—Bits 23:16. The OSPM programs this field with a non-zero value to convey a desired performance level to the hardware. The Desired Performance hint may be set to any performance value in the range [MinPerf, MaxPerf], inclusive. When programmed to zero, autonomous mode is selected, and hardware autonomously determines the performance level appropriate for the current workload.
- *MinPerf[7:0] (Minimum Performance hint)*—Bits 15:8. The OSPM programs this field to limit the minimum performance that is expected to be supplied by the hardware. Excursions below the minimum are possible due to unexpected system thermal or power constraints. MinPerf may be set to any performance value in the range CPPC_CAPABILITY_1[LowestPerf, HighestPerf], inclusive but must be set to a value less than or equal to MaxPerf.
- *MaxPerf[7:0] (Maximum Performance hint)*—Bits 7:0. The OSPM programs this field to limit the maximum performance for the purpose of energy efficiency or thermal control. Excursions above the maximum are possible due to hardware coordination between processor cores and other components in the package. Maximum performance may be set to any performance value in the range CPPC_CAPABILITY_1[LowestPerf, HighestPerf] inclusive.

17.6.5.2 CPPC_REQUEST_2

The CPPC_REQUEST_2 register is supported on processors that set CPUID Fn8000_0007_EDX[CppcPerfPriority] (bit 16) = 1 and conveys Performance Priority requests to the CPPC hardware.



Bits	Mnemonic	Description	Access Type
63:8	Reserved		MBZ
7:0	FloorPerf	Floor Performance	RW

Figure 17-13. CPPC_REQUEST_2

The fields of CPPC_REQUEST_2 are further described below:

- *FloorPerf (Floor Performance)*—Bits 7:0. This field contains the floor performance level associated with that core. Software can write any value between CPPC_CAPABILITY_1[LowestPerf] and CPPC_CAPABILITY_1[HighestPerf] to this field. When the system is under power/thermal constraints, the platform firmware will attempt to throttle the CPU frequency to the value corresponding to Floor Performance before throttling further. This allows software to specify different floor frequencies for different cores.

[AMD Public Use]

17.6.6 CPPC Feedback Mechanisms

The CPPC hardware provides performance feedback to the OSPM via the CPPC_STATUS MSR. The bits within the CPPC_STATUS register are sticky, and when set will remain set until software clears them. Software must clear these bits to enable notifications for subsequent events.

The processor sets CPPC_STATUS[MinEx] to signal that an excursion to less than minimum performance has occurred. This indicates that an unpredicted power or thermal event has occurred, and the processor has limited delivered performance to less than CPPC_REQUEST[MinPerf].

17.6.6.1 CPPC_STATUS

The CPPC_STATUS register conveys performance feedback to the OSPM.



Bits	Mnemonic	Description	Access Type
63:2	Reserved		MBZ
1	MinEx	Minimum Excursion has occurred	R/W
0	Reserved		MBZ

Figure 17-14. CPPC_STATUS

The CPPC_STATUS register fields are further described below:

- *MinEx (Minimum Excursion)*—Bit 1. The processor sets this bit when Delivered Performance has been constrained to less than CPPC_REQUEST[MinPerf]. Software must clear this bit to enable further notifications of minimum performance excursions.

17.6.6.2 Frequency Metrics

The AMD64 architecture provides a mechanism by which the OSPM can determine the effective frequency of a logical processor.

The Maximum Performance Frequency Clock Count (MPERF) and the Actual Performance Frequency Clock Count (APERF) registers constitute the effective frequency interface. This interface provides a means for software to calculate an average, or effective, frequency of a core over a known window of time. See Section 17.3, “Determining Processor Effective Frequency,” on page 699 for more information.

[AMD Public Use]

18 Shadow Stacks

The shadow stack mechanism facilitates protection against a common form of computer exploit known as Return Oriented Programming (ROP). ROP exploits utilize intentionally corrupted stack frames to divert normal processor control flow into short fragments of existing executable code, which ultimately end with a RET instruction. These fragments are then chained together using return addresses previously written to the stack by the attacker.

The shadow stack mechanism protects against ROP exploits by ensuring that return addresses read from the stack by RET and IRET instructions originated from a CALL instruction or similar control transfer.

In the following sections, the term ‘program stack’ is used to distinguish the stack pointed to by the SP register and manipulated by instructions (such as PUSH, POP, ENTER, LEAVE) from the shadow stack.

18.1 Shadow Stack Overview

A shadow stack is a separate, protected stack that is conceptually parallel to the program stack and used only by control-transfer and return instructions. When shadow stacks are enabled, most control transfers that save a return address (such as CALL, INTn, exceptions and interrupts) write the return address to the shadow stack in addition to the program stack. Upon the subsequent RET or IRET operation, the processor reads the return address from both stacks and checks that they match. A control-protection exception (#CP) is raised if the return addresses do not match.

The shadow stack is implemented in regions of memory marked with the “shadow stack” attribute in the page tables (see “Page-Protection Checks” on page 163). Pages with the shadow stack attribute are writable only by control transfer operations that save a return address and by the shadow stack management instructions. Shadow stack pages are not writeable by ordinary data-access instructions and thus protected from tampering.

18.1.1 Detecting and Enabling Shadow Stack Support

Support for the shadow stack feature is indicated by CPUID Fn0000_0007_ECX_x0[CET_SS](bit 7)=1. This bit also indicates the shadow stack MSRs are present.

The shadow stack feature is enabled by setting CR4.CET (bit 23) = 1 (see “CR4 Register” on page 47). The shadow stack feature is operational when in protected mode (CR0.PE=1) with paging enabled (CR0.PG=1). Shadow stacks are disabled in virtual x86 mode (rFLAGS.VM=1).

Once the feature is enabled, shadow stack operation can be separately enabled in user mode and in supervisor mode using control bits in the S_CET and U_CET MSRs (see “Shadow Stack MSRs” on page 722).

[AMD Public Use]

18.2 The Shadow Stack Pointer

On processors implementing the shadow stack feature, the Shadow Stack Pointer (SSP) register contains the address of the current top of the shadow stack. The width of the shadow stack is 64 bits in 64-bit mode and 32 bits in legacy and compatibility modes. The address size of SSP is 64 bits in 64-bit mode and 32 bits in legacy and compatibility modes.

The SSP register cannot be encoded as a source or destination register by regular instructions and thus is not directly accessible to software as a general operand, nor as an address operand. The SSP register is directly accessible only by using the shadow stack management instructions (see “Shadow Stack Management Instructions” on page 721).

18.3 Shadow Stack Operation for CALL (near) and RET (near)

When shadow stacks are enabled, the CALL (near) instruction pushes the return address onto the shadow stack, in addition to pushing it onto the program stack. On the subsequent near RET/RETn instruction, the return addresses are read from both stacks and compared. If the return addresses do not match, a control-protection (#CP) exception is generated.

A CALL (near) with a displacement of 0 (CALL +0) does not push the return address onto the shadow stack since the “CALL +0” idiom does not actually branch to another code sequence, and is typically not followed by a return instruction.

One form of the return instruction, RETn, pops ‘n’ additional parameters from the program stack before returning to the caller. Since the shadow stack does not store any data items, RETn does not pop additional parameters from the shadow stack.

18.4 Shadow Stack Operation for Far Transfers

The term ‘far transfer’ in the following discussion encompasses the following types of control transfers:

- Explicit CALL (far) to a procedure in another code segment
- Exceptions and interrupts that transfer to a handler in another code segment
- RET (far) and IRET instructions

A far transfer may also change the CPL. The exact operation of the shadow stack depends on the type of far transfer involved, the associated CPL change (if any) and whether the shadow stack capability is enabled for the target CPL. Shadow stack operations for far transfers are described in the following sections and summarized in Table 18-1. (See AMD Architecture Programmers Manual Volume 3 for detailed instruction algorithms for CALL (far), RET (far), IRET and INTn.)

[AMD Public Use]

Table 18-1. Shadow Stack Operations for Far Transfers

CALLF/int/excp From CPL	To CPL	Stack switch?	New SSP loaded from: (where n=new CPL)	CS, LIP saved to and restored from:	SSP saved to and restored from:	RETF/IRET checks returns address?
0,1,2,3	Same	No	Uses current SSP	Current Shadow Stack	Current Shadow Stack	Yes
3	0,1,2	Yes	PLn_SSP	<not saved/restored>	PL3_SSP	No
2	0,1	Yes	PLn_SSP	New shadow stack	New shadow stack	Yes
1	0	Yes	PLn_SSP	New shadow stack	New shadow stack	Yes

18.5 Far Transfer to the Same Privilege Level

When shadow stacks are enabled, a far transfer to the same privilege level pushes the CS and LIP (linear address of the return IP) onto the current shadow stack in addition to pushing the return IP onto the program stack. Upon the subsequent RET (far) or IRET operation, the return addresses are popped from both stacks and the linear forms of the two addresses are compared. If the return addresses do not match, a control-protection (#CP) exception is generated.

18.6 Far Transfer to Different Privilege Level

In addition to changing code segments, far transfers can also be used to change to a different privilege level (CPL). When changing the CPL, inter-privilege far transfers switch to a new program stack. In a similar manner, inter-privilege far transfers also switch to a new shadow stack, providing that shadow stacks are enabled for the new privilege level.

18.6.1 Shadow Stack Switching

When an inter-privilege far transfer switches to a new program stack, the new program stack pointer is selected from the either the inner-level stack pointers in the TSS or the Interrupt Stack Table (IST), depending on the type of far transfer. (See “Interrupt To Higher Privilege” on page 275). Similarly, when shadow stacks are enabled, inter-privilege far transfers also switch to a new shadow stack. The new shadow stack pointer is selected from shadow stack pointer MSRs (PLn_SSP) or the Interrupt Shadow Stack Table (ISST) as described in the following sections.

18.6.1.1 Shadow Stack Switching for Inter-Privilege CALL (far)

When a CALL (far) changes privilege level, a shadow stack switch occurs to an inner-level shadow stack. The new SSP is loaded from one of the following MSRs:

- PL2_SSP for transitions to CPL 2.
- PL1_SSP for transitions to CPL 1.
- PL0_SSP for transitions to CPL 0.

[AMD Public Use]

When switching shadow stacks, the processor validates the new shadow stack using a special value called a *shadow stack token* as described in Section 18.6.3 “Supervisor Shadow Stack Token” on page 714.

18.6.1.2 Shadow Stack Switching for Interrupts and Exceptions

In long mode (EFER.LMA=1), the processor provides an additional mechanism, the Interrupt Stack Table (IST) to switch program stacks for interrupts and exceptions. The IST mechanism uses the IST field in the Interrupt Descriptor Table (IDT) entry as an index into a table of inner-level program stack pointers (see “Interrupt-Stack Table” on page 287). The shadow stack feature provides a similar mechanism, the Interrupt Shadow Stack Table (ISST), for switching shadow stacks.

The ISST is an 8-entry table containing supervisor shadow stack pointers. The base of the ISST is specified by the ISST_ADDR MSR. If the IST field in the IDT entry is non-zero, it is used as an index into the ISST to select a new SSP as shown in Figure 18-1.

If the IST field is zero or if the processor is not in long mode, the ISST mechanism is not used and the new inner-level SSP is selected from the PLn_SSP MSRs as previously described in Section 18.6.4 “Shadow Stack Switching for Inter-Privilege CALL (far)” on page 712.

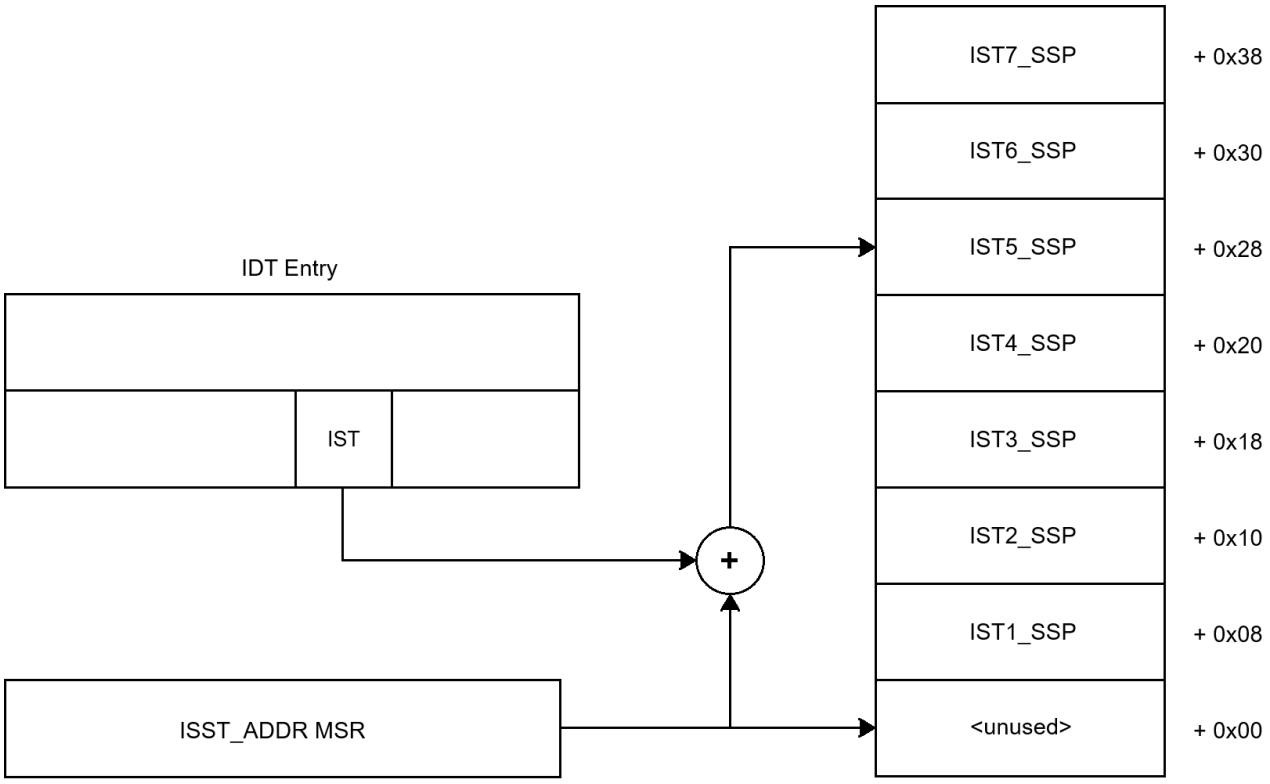


Figure 18-1. Interrupt Shadow Stack Table (ISST)

[AMD Public Use]

18.6.2 Handling CS, LIP and SSP on Privilege Transistions

When changing to a new code segment, the shadow stack behavior for saving and restoring CS, LIP and SSP depends on the CPL of the originating far transfer.

Transitions from CPL=3

Inter-privilege far transfers originating at CPL=3 save the current user-level SSP to PL3_SSP rather than to the supervisor shadow stack. Upon the subsequent RETF/IRET back to CPL=3, the user-level SSP is restored from PL3_SSP and the return CS and LIP are not verified.

Transitions from CPL=1 and CPL=2

Far transfers originating at CPL=1 or CPL=2 to an inner (more privileged) level save the current CS, LIP and SSP to the inner level shadow stack after switching shadow stacks as described in section Section 18.6.1, “Shadow Stack Switching,” on page 712. Upon the subsequent RET(far)/IRET, the SSP is restored (Section 18.6.1, “Shadow Stack Switching,” on page 712) from the inner-level shadow stack and the CS and LIP popped from the shadow stack are verified against the values read from the program stack. If the return addresses do not match, a control-protection (#CP) exception is generated.

18.6.3 Supervisor Shadow Stack Token

When switching shadow stacks, the processor validates the new shadow stack by checking a *supervisor shadow stack token* located at the base of the stack.

When initially creating a shadow stack for use at privilege levels 0, 1 and 2, system software must place a supervisor shadow stack token at the base of each stack. System software must also store a pointer to each shadow stack in the appropriate shadow stack base register, PLn_SSP (n=0,1,2,3) using the WRMSR instruction.

The relationship of the shadow stacks, supervisor shadow stack tokens, and the PLn_SSP registers are shown in Figure 18-2 on page 715. The three supervisor shadow stacks (PL0, PL1 and PL2) have shadow stack tokens stored at the base of the stack. Tokens are not used for PL3 shadow stacks.

[AMD Public Use]

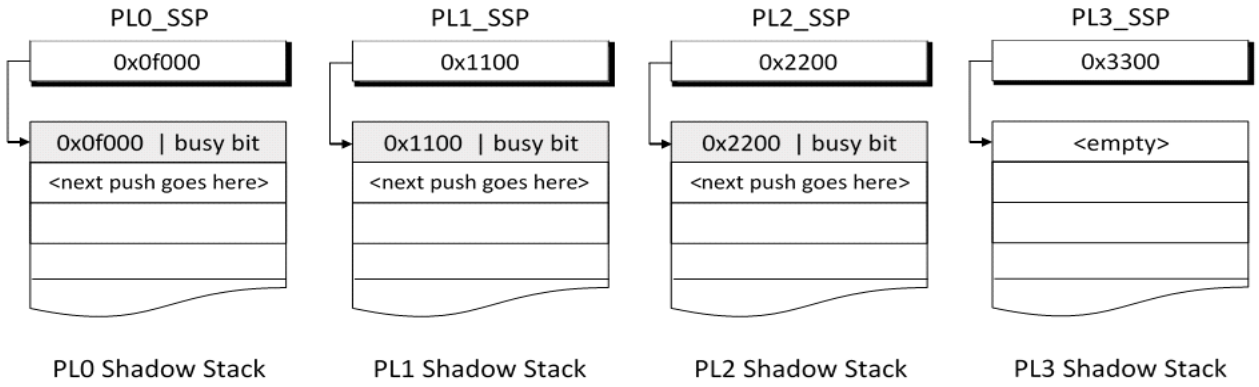


Figure 18-2. Shadow Stacks and Supervisor Shadow Stack Tokens

The supervisor shadow stack token is a 64-bit value and is formatted as shown in Figure 18-3:

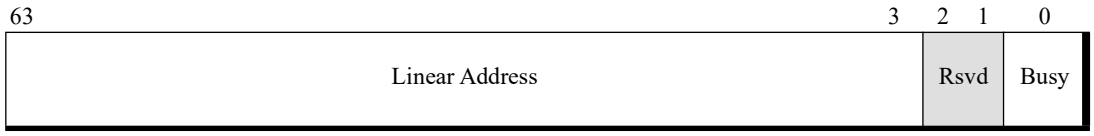


Figure 18-3. Supervisor Shadow Stack Token

The supervisor shadow stack token fields are defined as follows:

Linear Address. Bits 63:3 of the linear address of this token. The linear address is required to be 8-byte aligned.

Rsvd. Reserved, must be zero.

Busy Bit. If 0, this indicates this supervisor shadow stack is not in use by any logical processor. If 1, it indicates this supervisor shadow stack is currently in use by one of the logical processors. The processor sets the busy bit when switching to a supervisor shadow stack and clears it when switching away from that stack.

18.6.4 Shadow Stack Token Validation for Inter-privilege CALL (far) and Interrupts/Exceptions

Before switching shadow stacks, a CALL (far) or an interrupt/exception that changes to a more privileged level validates the supervisor shadow stack token located at the base of the new stack. If the validation checks pass, the supervisor shadow stack token is marked busy. The following steps are performed to validate the supervisor shadow stack token and set the busy bit:

[AMD Public Use]

1. Check that the incoming SSP is 8-byte aligned, otherwise a #GP exception is generated. The incoming SSP is located in PLn_SSP (where 'n' is the new privilege level: 0, 1, 2), or the ISST for interrupts/exceptions (see Section 18.6.1, "Shadow Stack Switching," on page 712).
   - If the far transfer also saves current CS, LIP and current SSP onto the shadow stack, the check also requires that the supervisor shadow stack token and additional information saved on the stack are fully contained within a 32-byte region that is aligned to 32-bytes. If not, a #GP exception is generated.

<Steps 2-6 are performed atomically.>

2. Fetch the 8-byte shadow stack token using the address specified by the incoming SSP (using a locked load with shadow stack access rights).
3. Check that the shadow stack token reserved bits and the busy bit are 0.
4. Check that the address specified in the shadow stack token matches the incoming SSP.
5. If checks 3 and 4 pass, set the supervisor shadow stack token busy bit (using a store, unlock).
6. If checks 3 or 4 fail, the shadow stack token is not modified and a #GP exception is generated.

18.6.5 Shadow Stack Token Validation for Inter-privilege RET and IRET

RET (far) and IRET to lower privilege levels validate the shadow stack token located on the current supervisor shadow stack before switching shadow stacks. If the token is valid, the token's busy bit is cleared. The processor performs the following steps to validate the token:

1. Check that the return SSP (located on the current shadow stack) is 4-byte aligned, else a #CP exception is generated.

<Steps 2-7 are performed atomically.>

2. Fetch the 8-byte shadow stack token from the current shadow stack at SSP+24 (using a locked load with shadow stack access rights).
3. Check that the busy bit is 1.
4. Check that the reserved bits are 0.
5. Check that the address specified in the shadow stack token matches the SSP.
6. If checks 3 through 5 pass, the shadow stack token busy bit is cleared.
7. If any of checks 3 through 5 fail, the shadow stack token is not modified. A fault is not generated.

18.7 Shadow Stack Operation for SYSCALL and SYSRET

SYSCALL and SYSRET are low-latency system call and return instructions designed for use by system and application software implementing a flat-memory model. These instructions do not use the program stack to store return addresses, and therefore do not use the shadow stack to validate return addresses. However, SYSCALL and SYSRET modify SSP as described below. The Shadow stack

[AMD Public Use]

operations described for SYSCALL and SYSRET also apply to SYSENTER and SYSEXIT respectively, although the latter are available only in legacy mode.

The SYSCALL instruction is used by application software executing at CPL=3. When shadow stacks are enabled at CPL=3, SYSCALL saves the current SSP to PL3_SSP. SYSCALL changes to CPL=0 before entering the operating system and if shadow stacks are enabled at CPL=0, then SSP is cleared to 0.

Unlike other inter-privilege far transfers, SYSCALL does not automatically perform a switch to a supervisor shadow stack. If shadow stacks are enabled at CPL=0, prior to executing a CALL instruction or similar transfer of control that pushes a return address to the shadow stack, software at the OS entry point must ensure that a supervisor shadow stack is available for use. System software can use SETSSBSY to set up a supervisor shadow stack.

The operating system uses the SYSRET instruction to return to the application running at CPL=3. If shadow stacks are enabled at CPL=3, SYSRET restores SSP from PL3_SSP. Prior to using SYSRET to return to the application, system software can use the CLRSSBSY to tear down the supervisor shadow stack. Because CLRSSYBSY clears the SSP to 0, system software must ensure that any subsequent interrupt or exception that may occur in CPL=0 prior to the SYSRET is configured to use the ISST stack-switching mechanism. Otherwise taking an interrupt or exception with SSP=0 will likely result in a fault due to SSP wrap-around.

18.8 Shadow Stack Operation for Task Switches

The legacy x86 task-switch mechanism transfers program control to a new task when any of the following control transfers occur:

- A CALL or JMP instruction references a task gate or TSS descriptor.
- A software-interrupt instruction (INTn), exception or external interrupt references a task gate.
- An IRET is executed when the EFLAGS.NT bit is set to 1.

Shadow stack operations for legacy x86 task switches are summarized in this section. Because the x86 task management feature is supported by the AMD64 architecture only in legacy mode (EFER.LMA=0), the shadow stack operations described below only apply to legacy mode. (See “Switching Tasks” on page 382 for more information on task switching).

When switching to a new task with shadow stacks enabled, the new task must use a 32-bit TSS. The SSP for the new task is located at TSS offset 104. Since the SSP is 4 bytes in legacy mode, the TSS must be at least 108 bytes in size. The SSP must be aligned to an 8-byte boundary and point to a supervisor shadow stack token.

If the task switch is initiated by a CALL/JMP/INTn instruction, or an interrupt or exception:

- For task switches originating at CPL=3, and if shadow stacks are enabled at that CPL, the current SSP is saved to PL3_SSP. Otherwise, for task switches originating at supervisor-level (CPL=0,1,2) the current SSP is saved onto the new shadow stack along with current CS and LIP. If shadow

[AMD Public Use]

stacks are enabled at the CPL of the new task, the busy bit is set in the supervisor shadow stack token pointed to by the SSP of the incoming task.

If the task switch is initiated by an IRET instruction:

- For task switches originating at supervisor-level (CPL=0,1,2) and returning to CPL=3, the SSP is restored from PL3_SSP, otherwise the SSP is restored from the current shadow stack. The return CS and LIP are read from the current shadow stack and compared to the CS and linear form of the EIP in the TSS of the incoming task. A control-protection (#CP) exception is generated if the return addresses do not match.

18.9 Restricting Speculative Execution of RET targets

When shadow stacks are enabled, the processor will not speculatively execute instructions from a RET address on the program stack unless the address matches the corresponding address on the shadow stack, or the target of the RET is predicted by a return address branch prediction mechanism.

18.10 Shadow Stack Switching Using RSTORSSP

As previously described in Section 18.6.1, “Shadow Stack Switching,” on page 712, the processor automatically switches shadow stacks as part of the inter-privilege far transfer mechanism. In order to allow programs to switch stacks at other times (when performing operations such as initializing a shadow stack or recovering from a #CP fault) the RSTORSSP and SAVEPREVSSP instructions are provided.

The RSTORSSP instruction is used to switch shadow stacks. The instruction expects to find a shadow stack restore token at the top of the proposed new shadow stack. Upon validating this token, RSTORSSP points the SSP to the top of the new shadow stack and sets the token’s busy bit, making the new shadow stack ready for use.

Upon successful completion, the RSTORSSP instruction modifies the restore token by saving the old SSP into the token’s SSP Restore Value field in order to facilitate a later return to the old shadow stack.

The shadow stack restore token used by RSTORSSP is formatted as follows:



Figure 18-4. Shadow Stack Restore Token

The shadow stack restore token fields are defined as follows:

SSP Restore value. Bits 63:2 of the linear address of this token. Replaced by the previous SSP upon successful completion of the RSTORSSP instruction.

[AMD Public Use]

Busy. Initially must be 0. Set to 1 upon successful completion of the RSTORSSP instruction.

Mode. If 0, this indicates the shadow stack restore token is for use in legacy or compatibility mode. If 1, this indicates the restore token is for use in 64-bit mode.

After RSTORSSP switches to the new stack the modified shadow stack token is at the top of the new in-use shadow stack. If a return to the old stack is not required, the modified token (to which the previous SSP has been saved) can be popped from the stack using the INCSSP instruction.

If a return to the old stack is desired, the SAVEPREVSSP instruction can be used to copy the token back to the previous stack for later use by an RSTORSSP instruction. The SAVEPREVSSP expects to find a *previous SSP token* on the top of the current shadow stack. After moving this token to the previous stack, SAVEPREVSSP pops it off the new stack by incrementing SSP.

The previous SSP token used by SAVEPREVSSP is formatted as follows:

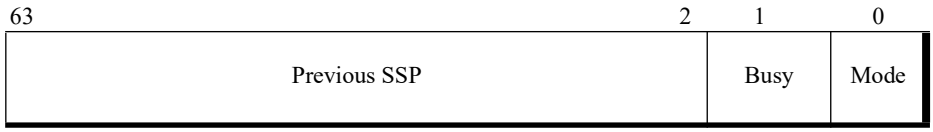


Figure 18-5. Previous SSP Token

The previous SSP token fields are defined as follows:

Previous SSP. Bits 63:2 of the previous SSP. Upon successful completion, the SAVEPREVSSP instruction copies the token to this address.

Busy. Must be set to 1 initially. Cleared to 0 upon successful completion of the SAVEPREVSSP instruction.

Mode. If 0, this indicates the previous SSP token is for use in legacy or compatibility mode. If 1, this indicates the token is for use in 64-bit mode.

Figure 18-6 “RSTORSSP and SAVEPREVSSP Operation” on page 720 illustrates the operation of the RSTORSSP instruction when switching to a new shadow stack, followed by a SAVEPREVSSP instruction to save a previous token back to the original stack.

[AMD Public Use]

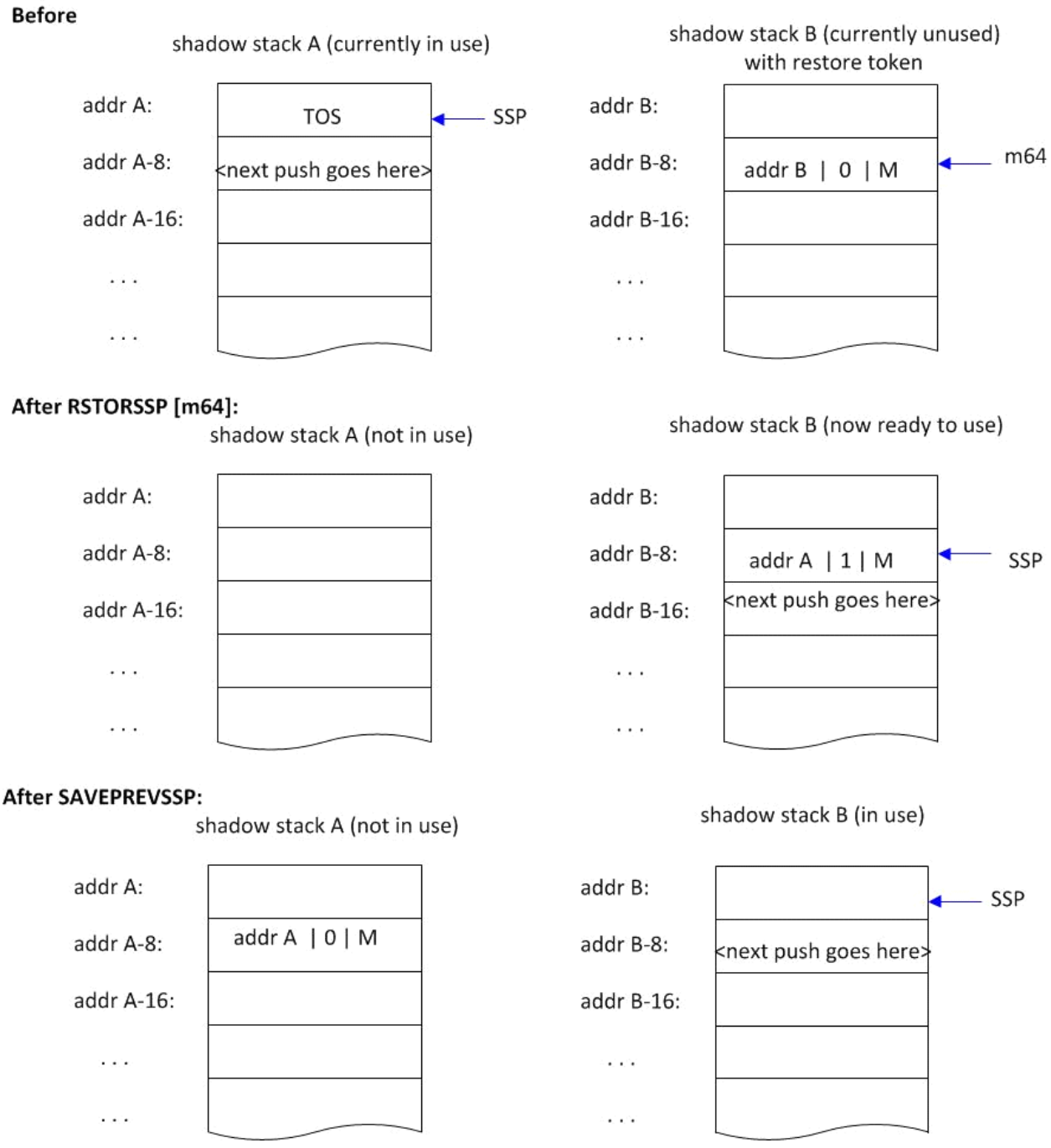


Figure 18-6. RSTORSSP and SAVEPREVSSP Operation

[AMD Public Use]

In this example (Figure 18-6 “RSTORSSP and SAVEPREVSSP Operation” on page 720), RSTORSSP is used to switch from shadow stack A to shadow stack B. Subsequently, a SAVEPREVSSP instruction is used to save a shadow stack restore token.

Initially, shadow stack A is in-use, and the proposed shadow stack B contains a shadow stack restore token at address B - 8. An RSTORSSP instruction is then executed with the operand pointing to the restore token.

The RSTORSSP instruction checks the restore token for validity, copies the previous SSP to the token, sets bit 1 of the token and sets the new value of SSP to address B. Shadow stack B is now ready for use.

Next, a SAVEPREVSSP instruction is executed to facilitate a later return to shadow stack A (using a subsequent RSTORSSP instruction, not shown). The SAVEPREVSSP instruction copies the token to address A - 8 and clears token bit 1, then pops the old token from shadow stack B.

For detailed RSTORSSP and SAVEPREVSSP algorithms refer to the instruction descriptions in APM Volume 3.

18.11 Shadow Stack Management Instructions

When shadow stacks are enabled, the following instructions are available to software for use in managing shadow stacks. Except for RDSSP, attempting to execute these instructions when shadow stacks are disabled results in a #UD exception. RDSSP is treated as a NOP when shadow stacks are disabled. For more information refer to the detailed instruction descriptions in APM Volume 3.

Table 18-2. Shadow Stack Management Instructions

Mnemonic	Name	Description
CLRSSBSY	Clear Shadow Stack Busy	Validates a shadow stack token and clears its busy bit. This is a privileged instruction.
INCSSP	Increment Shadow Stack Pointer	Increment SSP by ‘n’ stack frames. Used to pop unneeded items from a shadow stack.
RDSSP	Read Shadow Stack Pointer	Read the SSP into a GPR. Treated as a NOP if shadow stacks are disabled.
RSTORSSP	Restore Shadow Stack Pointer	Used to switch shadow stacks. Expects a ‘shadow stack restore token’ at the top of the new shadow stack. Upon validating this token, sets the token’s busy bit and sets SSP to the top of the new shadow stack.
SAVEPREVSSP	Save Previous Shadow Stack Pointer	Copies a ‘previous SSP token’ from the current shadow stack back to the previous stack for later use by an RSTORSSP instruction.

[AMD Public Use]

Table 18-2. Shadow Stack Management Instructions (continued)

Mnemonic	Name	Description
SETSSBSY	Set Shadow Stack Busy	Validates the shadow stack token pointed to by the PL0_SSP MSR. If valid, clears the busy bit and sets SSP = PL0_SSP. This is a privileged instruction.
WRSS	Write Shadow Stack	Writes the source operand to a shadow stack. This instruction has associated enable bits in the U_CET and S_CET MSRs
WRUSS	Write User Shadow Stack	Writes the source operand to a user shadow stack. This is a privileged instruction.

18.12 Shadow Stack MSRs

The following MSRs are defined if the shadow stack feature is supported as indicated by CPUID Fn 0000_0007_0 ECX[CET_SS] (bit 7) = 1:

U_CET. MSR 0x6A0. Specifies the user mode shadow stack controls. The individual fields are as follows:

- Bit 0 - SH_STK_EN. When set to 1, enables the shadow stacks in user mode.
- Bit 1 - WR_SHSTK_EN. When set to 1, enables the WRSS instruction is user mode.
- Bits 63:2 – reserved, MBZ

S_CET. MSR 0x6A2. Specifies the supervisor mode shadow stack controls.

- Bit 0 - SH_STK_EN. When set to 1, enables the shadow stacks in supervisor mode.
- Bit 1 - WR_SHSTK_EN. When set to 1, enables the WRSS instruction is supervisor mode.
- Bits 63:2 – reserved, MBZ

PL0_SSP. MSR 0x6A4. Specifies the linear address to be loaded into SSP on the next transition to CPL=0. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

PL1_SSP. MSR 0x6A5. Specifies the linear address to be loaded into SSP on the next transition to CPL=1. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

PL2_SSP. MSR 0x6A6. Specifies the linear address to be loaded into SSP on the next transition to CPL=2. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

PL3_SSP. MSR 0x6A7. The user mode SSP is saved to and restored from this register. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

ISST_ADDR. MSR 0x6A8. This register specifies the linear address of the Interrupt Shadow Stack Table (ISST). The linear address must be in canonical format.

[AMD Public Use]

18.13 XSAVES/XRSTORS

The XSAVES/XRSTORS instructions can be used to manage the shadow stack registers as processor extended state components 11 (CET_U state) and 12 (CET_S state) as described below:

- **CET_U state.** The shadow stack user controls are identified as state component 11 in XSS MSR and consist of the two 64-bit MSRs U_CET and PL3_SSP. Bytes 7:0 in the XSAVE area are used for U_CET and bytes 15:8 are used for PL3_SSP. The XSAVE area size and offset for the CET_U state are available using CPUID Fn0000_000D_ECX_x0B (ECX=11).
- **CET_S state.** The shadow stack supervisor controls are identified as state component 12 in XSS MSR and consist of the three 64-bit MSRs PL0_SSP, PL1_SSP and PL2_SSP. Bytes 7:0 in the XSAVE area are used for PL0_SSP, bytes 15:8 are used for PL2_SSP and bytes 23:16 are used for PL3_SSP. The XSAVE area size and offset for the CET_S state are available using CPUID Fn0000_000D_ECX_x0C (ECX=12).

When restoring CET_U and CET_S state, XRSTORS checks for reserved bits and canonicality as described in Section 18.11 “Shadow Stack Management Instructions” on page 721.

[AMD Public Use]

19 Platform Quality of Service (PQOS) Extension

The AMD PQOS Extension provides system software (operating system, hypervisor, or similar software) with mechanisms to monitor usage of shared platform resources such as L3 cache by a logical processor, thread, application, or guest OS. The PQOS extension also provides system software with the ability to set and enforce limits on usage of certain shared platform resources.

In shared environments, such as those found in cloud computing installations, multiple threads, or applications contending for a shared resource can cause significant impact to collocated workloads. For example, contention for a resource such as the last-level cache may cause violation of service-level objectives. Contention for shared resources may also cause latency spikes that violate runtime objectives of latency-sensitive tasks. The PQOS extension can be used to minimize the effects of contention for shared resources from co-located workloads.

19.1 PQOS Extension Overview

The PQOS extension provides for the *monitoring* of the usage of certain shared resources and, separately, for the *enforcement* of limits on the use of shared resources.

The monitoring and enforcement functions of PQOS are not necessarily applied across the entire system, but in general apply to a QOS Domain, which corresponds to a level in the processor hardware hierarchy containing a shared resource. In some implementations, the QOS Domain is a Core-Complex and the shared resource whose usage is monitored or regulated by PQOS is the last-level cache.

Platform QOS features are implemented on a logical processor basis. Therefore, multiple threads (logical processors) in a single CPU core have independent PQOS resource monitoring and enforcement configurations.

The set of resources that may be monitored, and the set for which limits can be enforced, are implementation-dependent and enumerated by a CPUID function on a feature and sub-feature basis, as described in the following sections.

The PQOS extension consists of two separate but related features—PQOS Monitoring (PQM) and PQOS Enforcement (PQE).

The PQM feature provides mechanisms for monitoring the usage of shared resources. The set of resources that can be monitored and the specific usage metrics that may be tracked for those resources are implementation-dependent and enumerated by CPUID (see Section 19.2.1).

[AMD Public Use]

PQM can be used to monitor usage of the following resources:

- L3 cache occupancy (see Section 19.3.2)
- L3 cache bandwidth (see Section 19.3.3)

The ability to set and enforce limits on usage of shared resources is provided by the PQE feature. The set of resources for which limits can be enforced is implementation-dependent and enumerated by CPUID (see Section 19.4.1). PQE can be used to enforce limits on the usage of the following resources:

- L3 cache allocation (see Section 19.4.2 and Section 19.4.3)
- L3 cache bandwidth (see Section 19.4.4, Section 19.4.5, Section 19.4.6, and Section 19.4.7)

19.1.1 Detecting Support for PQOS

Overall support for the PQOS extension is indicated by CPUID Fn0000_0007_EBX[PQE, PQM] as described in Table 19-1.

Table 19-1. CPUID Fn000_0007_EBX_x0 Extended Feature Identifiers

EBX Bit	Name	Description
12	PQM	PQOS Monitoring Feature
15	PQE	PQOS Enforcement Feature

- PQM, EBX bit 12. This bit also indicates the PQM-related MSRs are present. (See “PQM MSRs” on page 728).
- PQE, EBX bit 15. This bit also indicates that certain PQE-related MSRs are present. (See “PQOS Enforcement (PQE)” on page 742).

19.1.1.1 PQOS Versions

Identifying the attributes and capabilities of the PQOS extension is generally performed using CPUID functions. However, some early implementations also use a *PQOS version* number in addition to CPUID. Table 19-2 identifies the PQOS Version associated with a product of a specific Family/Model.

Table 19-2. PQOS Versions

Family	Model	Version
17h	30h–90h	V1.0
19h	00h–0Fh	V2.0
19h	20h–5Fh	V2.0
All others		In later implementations, only CPUID is needed to identify PQOS capabilities and PQOS Version is not used.

[AMD Public Use]

19.1.2 Identifying PQM and PQE Capabilities

The PQM and PQE features operate on specific shared platform resources, such as the L3 cache. The set of shared resources supported by PQM and PQE are implementation-dependent and identified by CPUID functions Fn0000_000F and Fn0000_0010 respectively. Each of these CPUID functions provide a base leaf (ECX=0) which identifies the shared resource types supported by PQM and PQE and one or more sub-leaves (ECX=1,2,3…) that report additional attributes and capabilities for each shared resource as described in Section 19.2.1 and Section 19.4.1. A complete description of these CPUID functions can be found in Appendix E of APM Volume 3.

19.2 PQOS Monitoring (PQM) Overview

The ability to monitor resource usage of shared resources is provided by the PQM feature. Monitoring is accomplished by assigning a software-defined Resource Management Identifier (RMID) to a logical processor and at some later time reading the usage metrics of a particular monitored resource by that RMID.

19.2.1 Detecting PQM Monitoring Resources and Capabilities

Once overall support for PQM has been established, support for specific PQM sub-features may be determined using CPUID functions Fn00000_000F and Fn8000_0020.

The PQM feature operates on specific shared platform resources, such as the L3 cache. The set of shared resources supported by PQM is implementation dependent and identified by CPUID functions Fn0000_000F. The base leaf (ECX=0) identifies the overall resource types supported by PQM. Each supported resource is identified by a bit in EDX as shown in Table 19-3. This CPUID leaf also reports the largest RMID supported for any resource monitored by PQM.

Table 19-3. CPUID Fn0000_000F_x0 PQM Capabilities

Bits	Name	Description
EDX Bits		
0	–	Reserved
1	L3CacheMon	L3 monitoring capability
31:2	–	Reserved
EBX Bits		
31:0	Max_RMID	Largest RMID supported for any PQM resource

Additional PQM capabilities associated with each resource are returned by executing CPUID Fn0000_000F with ECX equal to that resource's bit number. For example, CPUID Fn0000_000F_x1 (ECX=1) returns L3-specific monitoring capabilities. See Table 19-4 for details.

CPUID Fn0000_000F_x1 returns the following information about the L3 Cache Monitoring feature:

[AMD Public Use]

- If EAX is non-zero, it indicates the size of the L3 monitoring counter, offset from 24 bits. If EAX is 0, the counter size is determined by the PQOS version number as described in Table 19-7 on page 732.
- EBX identifies the counter scaling factor. The QM_CTR value must be multiplied by this scaling factor to obtain the cache occupancy or bandwidth used in bytes.
- ECX identifies the largest RMID for L3 monitoring. Writing a larger value to the RMID field in the QM_EVTSEL register results in a #GP(0) exception.
- EDX identifies the specific L3 usage events that can be monitored. See Table 19-4. This bit vector also identifies the valid EvtID selections for use in the QM_EVTSEL registers.

Table 19-4. CPUID Fn0000_000F_EDX_x1 PQM Event Identifiers

EDX bits	PQM event identifier	Description
0	L3 Occupancy	L3 cache occupancy
1	L3 total bandwidth	L3 Cache Bandwidth Monitoring Event 0 At reset, this event tracks Total L3 Bandwidth. However, if BMEC is supported, the specific types of L3 bandwidth sources to be tracked are configurable. See Section 19.3.3.2 below.
2	L3 local bandwidth	L3 Cache Bandwidth Monitoring Event 1 At reset, this event tracks Local L3 Bandwidth. However, if BMEC is supported, the specific types of L3 bandwidth sources to be tracked are configurable. See Section 19.3.3.2 below.
31:3	–	Reserved

Extended L3 Cache Monitoring capabilities are enumerated by Fn8000_0020_EBX_x0 as described in Table 19-5.

Table 19-5. Fn8000_0020_EBX_x0 PQM Extended Feature Identifiers

EBX Bits	Name	Description
1	L3MBE	Memory Bandwidth Enforcement. See Section 19.4.5.
2	L3SMBE	Slow Memory Bandwidth Enforcement. See Section 19.4.6.
3	BMEC	Bandwidth Monitor Event Configuration (BMEC).See Section 19.3.3.2.
5	ABMC	Assignable Bandwidth Monitoring Counters (ABMC). See Section 19.3.3.3.
6	SDCIAE	SDCI Allocation Enforcement. See Section 19.4.7.
7	GLBE	Global Bandwidth Enforcement. See Section 19.4.8.
8	GLSBE	Global Slow Bandwidth Enforcement. See Section 19.4.9.
9	PLZA	Privilege Level Zero Association. See Section 19.3.1.4.

[AMD Public Use]

19.3 L3 Cache Monitoring

System software obtains the usage metrics for a particular resource by writing the RMID and Event Identifier (EvtID) to the QM_EVTSEL register and then reading the usage count from the QM_CTR register. See Section 19.3.1 for details on the PQM MSRs.

Each PQM usage metric is assigned an EvtID. The EvtID selects which of the monitored metrics is reported for the specified RMID. The EvtID is one more than the event identifier as reported in EDX by CPUID Fn0000_000F_x1. For example, the event identifier for L3 Cache Occupancy Monitoring is bit 0, thus its EvtID is 1.

The association of an RMID with a processor is performed by writing an RMID to the PQR_ASSOC MSR. Each logical processor can be assigned a unique RMID, or the same RMID can be assigned to multiple logical processors to track a multi-threaded application's shared resource usage. The monitoring hardware tracks and reports usage of shared resources (such as the L3 cache occupancy) on a per-RMID basis.

Software may associate multiple processors with the same RMID at the same time. If the processors are located within the same QOS Domain, the resource usage by all these processors is accumulated and the total is reported by the monitoring hardware.

Processors which set CPUID Fn8000_0020_EBX_x0[PLZA] (bit 9) = 1 support Privilege Level Zero Association (PLZA). PLZA allows the hardware to override the RMID and/or COS association specified in PQR_ASSOC MSRs when the logical processor executes at privilege level zero (CPL=0). PLZA is controlled by the PQR_PLZA_ASSOC MSR. The format of this MSR is shown in Section 19.3.1.4 "PQR_PLZA_ASSOC" on page 732. There is one instance of this MSR for each logical processor.

For PQOS Version 1.0 and 2.0, as identified by Family/Model, the QM_EVTSEL register is shared by all the processors in a QOS Domain. It is software's responsibility to ensure that no other process changes the QM_EVTSEL register between the time it is written and the time the QM_CTR read occurs. For later implementations, each processor in the QOS Domain has a private QM_EVTSEL and QM_CTR register pair, therefore software does not require synchronization between processors in the QOS Domain to use the register pair.

Writing a value larger than the MAX_RMID (CPUID Fn0000_000F_EBX_x0[MAX_RMID] to the QM_EVTSEL register will result in a #GP(0) exception. Writing to the QM_EVTSEL register an undefined EvtID or an RMID value greater than the largest RMID supported for that specific resource will cause the subsequent read of the QM_CTR to return 1 in the E (error) field.

19.3.1 PQM MSRs

The following MSRs are used by the PQM feature:

- PQR_ASSOC, MSR C8Fh. This MSR is used to assign an RMID to a logical processor.
- QM_EVTSEL, MSR C8Dh. This MSR is used to request a selected usage metric for a given RMID be returned to the QM_CTR MSR.

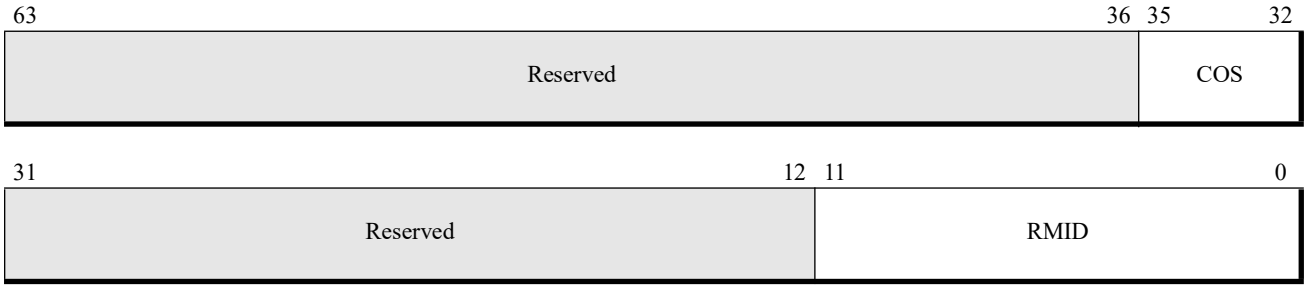
[AMD Public Use]

- QM_CTR, MSR C8Eh. This MSR is used to report the requested usage metric.
- PQR_PLZA_ASSOC, MSR C000_03FCh. This MSR is used to override the RMID and/or COS association specified in PQR_ASSOC when the logical processor executes at privilege level zero (CPL=0).

Each of these MSRs are described in detail below.

19.3.1.1 PQR_ASSOC

This MSR is used by system software to assign an RMID or a COS to a logical processor.



Bits	Mnemonic	Description	Access type
63:36	–	Reserved	MBZ
35:32	COS	Class of Service	R/W
31:12	–	Reserved	MBZ
11:0	RMID	Resource Monitor Identifier	R/W

Figure 19-1. PQR_ASSOC

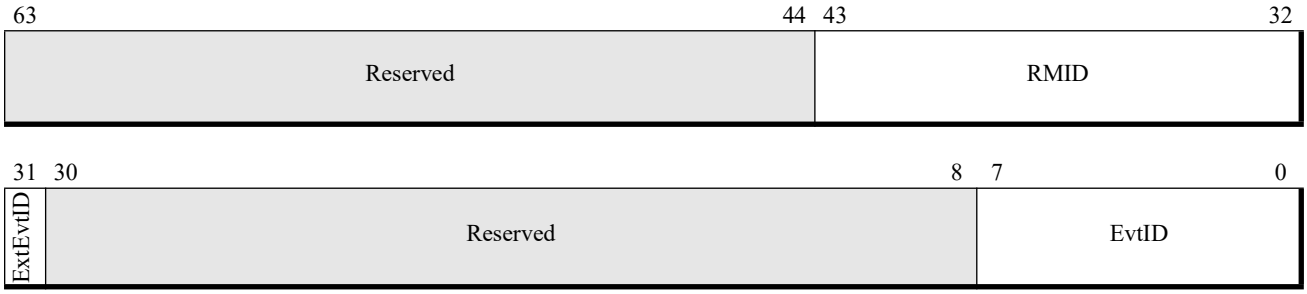
- COS[4:0] (Class of Service) – bits 35:32, read/write. System software uses the COS field to assign an arbitrary, numeric Class of Service value to this logical processor. Based on the COS, the PQOS hardware can enforce limits on usage of shared resources. At reset, the COS value is cleared to 0. The size of the COS field is implementation dependent. Software must use CPUID Fn0000_0010_EDX_x1[MAX_COS](bits 15:0) to determine the largest COS value supported by the processor. An attempt to write a larger value than MAX_COS results in a #GP(0) exception.
- RMID[11:0] (Resource Monitoring Identifier) – bits 11:0, read/write. The RMID field specifies an identifier that the PQOS hardware will use to tag usage of monitored events. Software subsequently uses the RMID value to retrieve the usage data for a particular monitored event. At reset, the RMID value is cleared to 0.

  The size of the RMID field is implementation dependent. Software must use CPUID Fn0000_000F_EBX_x0[MAX_RMID](bits 31:0) to determine the maximum RMID value supported by the processor. An attempt to write a larger value than MAX_RMID results in a #GP(0) exception.

[AMD Public Use]

19.3.1.2 QM_EVTSEL

This MSR is used to request a selected usage metric for a given RMID be returned to the QM_CTR MSR.



Bits	Mnemonic	Description	Access type
63:44	–	Reserved	MBZ
43:32	RMID	Resource Monitoring Identifier	R/W
31	ExtEvtID	Extended Event Identifier	R/W (*see note*)
30:8	–	Reserved	MBZ
7:0	EvtID	Event Identifier	R/W

Figure 19-2. QM_EVTSEL

The fields within the QM_EVTSEL register are:

- RMID[11:0] (Resource Monitoring Identifier) – bits 43:32, read/write. To read the usage of a particular monitored resource/event by an RMID, the corresponding EvtID and RMID are written to the QM_EVTSEL register, and the resulting usage is then read from the QM_CTR register. The width of the RMID field matches the width of the QM_ASSOC.RMID field.
- ExtEvtID (Extended Event Identifier) – bit 31. When set, the EvtID field refers to Extended Event Identifiers as shown in Table 19-6.

  Note: *This bit is MBZ unless L3_QOS_EXT_CFG.ABMC_En is set.*
- EvtID[7:0] (Event Identifier) – bits 7:0, read/write. This field specifies the PQM resource or event for which usage will be returned in the QM_CTR register. EvtID encodings are defined in Table 19-6.

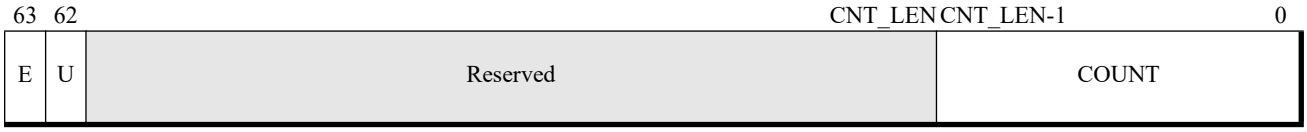
[AMD Public Use]

Table 19-6. QM_EVTSEL.EvtID Encodings

ExtendedEvtID	Value	Name	Description
0	1	L3OccMon	L3 cache monitoring event.
	2	L3BWMonEVT0	L3 cache bandwidth monitoring event 0. At reset, this event tracks TOTAL L3 memory bandwidth. However, if BMEC is supported the types of bandwidth tracked by this event can be configured.
	3	L3BWMonEVT1	L3 cache bandwidth monitoring event 1. At reset, this event tracks LOCAL L3 memory bandwidth. However, if BMEC is supported, the types of bandwidth tracked by this event can be configured.
1	1	L3CacheABMC	Assignable Bandwidth Monitoring Counter (ABMC) access.

19.3.1.3 QM_CTR

QM_CTR is used to report the usage metric selected by the QM_EVTSEL register.



Bits	Mnemonic	Description	Access type
63	E	Error on access to counter	RO
62	U	Count for this event is currently unavailable	RO
61:CNT_LEN	–	Reserved, read as zero	RAZ
CNT_LEN-1:0	CNT	Count of monitored resource or event	RO

Figure 19-3. QM_CTR

The fields within the QM_CTR register are:

- E (Error) – bit 63, read only. The E bit is set if an error occurs upon access to the counter, such as an illegal EvtID or RMID.
- U (Undefined) – bit 62, read only. The U bit is set if the count for this EvtID is not currently available. The monitoring hardware may set the U bit at any time to indicate that the requested event count for the specified RMID is not currently available. This is generally a temporary condition and subsequent reads may succeed.
- CNT (Count) – bits CNT_LEN-1:0, read only. The CNT field reports the usage data for monitored resources and events. To read the usage of a particular monitored resource/event by an RMID, the corresponding EvtID and RMID are written to the QM_EVTSEL register, and the resulting usage data is then read from the QM_CTR register. The value in the count field must be

[AMD Public Use]

scaled by multiplying by the scale factor reported by CPUID Fn0000_000F_EBX_x1[Scaling-Factor].

- The size of the CNT field (CNT_LEN) is implementation-dependent and is identified by CPUID Fn0000_000F_EAX_x1[CounterSize]. If CounterSize is non-zero, the width of the CNT field in bits is (CounterSize + 24). For example, if the value of CounterSize is 20, then the CNT field is 44 bits wide. If CounterSize is 0, use the PQOS Version (see Table 19-2 on page 725) and the following table to identify the counter size.

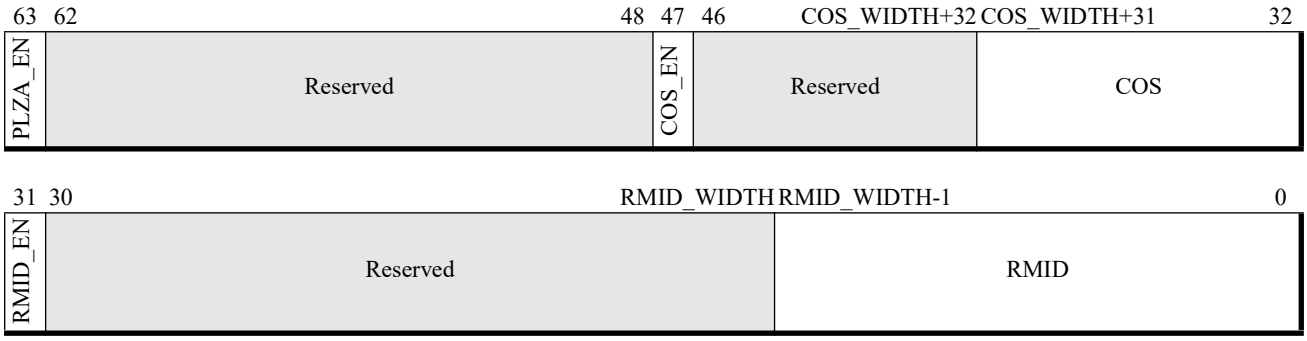
Table 19-7. Counter Size when Fn0000_000F_EAX_x1[CounterSize] = 0

PQOS Version	Counter Size (Bits)
V1.0	62
V2.0	44

19.3.1.4 PQR_PLZA_ASSOC

PQR_PLZA_ASSOC is used by system software to override the RMID and/or COS association specified in PQR_ASSOC (see Section 19.3.1.1 “PQR_ASSOC” on page 729) when the logical processor executes at privilege level zero (CPL=0).

The format of this MSR is shown in Figure 19-4.



Bits	Mnemonic	Description	Access type
63	PLZA_EN	PLZA Enable	R/W
62:48	–	Reserved	MBZ
47	COS_EN	PLZA Class-of-Service Association Enable	R/W
46:COS_WIDTH+32	–	Reserved	MBZ
COS_WIDTH+31:32	COS	PLZA Class-of-Service	R/W
31	RMID_EN	Resource-Monitoring-Identifier Association Enable	R/W
30:RMID_WIDTH	–	Reserved	MBZ
RMID_WIDTH-1:0	RMID	PLZA Resource-Monitoring-Identifier	R/W

Figure 19-4. PQR_PLZA_ASSOC

[AMD Public Use]

The fields within the PQR_PLZA_ASSOC register are:

- PLZA_EN (PLZA Enable) – bit 63. When set, PLZA is enabled for this logical processor. Each logical processor in the QOS Domain may have a unique value for PLZA_EN. When cleared for a logical processor, the COS and RMID association is specified in the PQR_ASSOC MSR independent of the logical processor's privilege level.
- COS_EN (PLZA Class-of-Service Association Enable) – bit 47. When set and PLZA_EN is set, the Class of Service (COS) for this logical processor executing at CPL=0 is specified in the COS field instead of PQR_ASSOC[COS].
- COS (PLZA Class-of-Service) – bits COS_WIDTH+31:32. The COS for this logical processor executing at CPL=0 when PLZA_EN and COS_EN are set. The size of the COS field is implementation dependent. Software must use CPUID Fn0000_0010_EDX_x1[MAX_COS](bits 15:0) to determine the largest COS value supported by the processor. An attempt to write a larger value than MAX_COS results in a #GP(0) exception.
- RMID_EN (PLZA Resource Monitoring Identifier Association Enable) – bit 31. When set and PLZA_EN is set, the Resource Monitoring Identifier (RMID) for this logical processor executing at CPL=0 is specified in the RMID field instead of PQR_ASSOC[RMID].
- RMID (PLZA Resource Monitoring Identifier) – bits RMID_WIDTH-1:0. The RMID for this logical processor executing at CPL=0 when PLZA_EN and RMID_EN are set. The size of the RMID field is implementation dependent. Software must use CPUID Fn0000_000F_EBX_x0[MAX_RMID](bits 31:0) to determine the maximum RMID value supported by the processor. An attempt to write a larger value than MAX_RMID results in a #GP(0) exception.

PLZA Usage

Before using PLZA, the PQR_PLZA_ASSOC MSR must be configured on all logical processors in the QOS Domain. Only one COS/RMID can be associated with PLZA in a given QOS Domain and all of the fields in PQR_PLZA_ASSOC except for PLZA_EN must be set to the same value for all logical processors in the QOS Domain for consistent operation. System software must observe the following configuration sequence to ensure expected PLZA behavior.

1. Disable PLZA for all logical processors in the QOS Domain by setting PQR_PLZA_ASSOC[PLZA_EN] = 0.
2. Configure the COS_EN, COS, RMID_EN, and RMID fields of PQR_PLZA_ASSOC to the desired configuration on all logical processors in the QOS Domain while PLZA remains disabled.
3. Enable PLZA for all logical processors in the QOS Domain where PLZA should be enabled by setting PQR_PLZA_ASSOC[PLZA_EN] = 1. System software should perform this as a read-modify-write to avoid changing the value of COS_EN, COS, RMID_EN, and RMID fields of PQR_PLZA_ASSOC.

[AMD Public Use]

When a PLZA enabled processor is executing at CPL=0, that logical processor will use the PQOS association specified in PQR_PLZA_ASSOC instead of the RMID and COS specified in PQR_ASSOC. Transitions between PQR_PLZA_ASSOC and PQR_ASSOC, as triggered by CPL changes, are not guaranteed to be precise at CPL boundaries.

When running with SVM enabled, PQOS will use the association specified by PQR_PLZA_ASSOC if the VMM is running with CPL=0 or if the guest is running with CPL=0.

19.3.2 Monitoring L3 Occupancy

Monitoring L3 cache occupancy is accomplished by first assigning a RMID to a given logical processor or group of processors. Then, at some later time, the L3 cache occupancy metric may be read by writing the RMID and the L3 Occupancy identifier (EvtID =1) to the QM_EVTSEL register and reading the resulting value from the QM_CTR register. The monitoring hardware may not keep a precise count of the L3 cache occupancy associated with a given RMID; the L3 cache occupancy value returned is an approximation.

The value returned in QM_CTR must be multiplied by the L3 Cache Scaling Factor to obtain the approximate number of bytes in the L3 cache currently associated with the supplied RMID. The L3 Cache Scaling Factor is returned by CPUID Fn0000_000F_EBX_x1[ScalingFactor](bits 31:0).

The QOS Domain for L3 occupancy monitoring is an L3 cache and all the processors which share that cache. When using L3 occupancy monitoring, L3 cache lines are tagged with the current RMID value in the PQR_ASSOC register of the processor which installed the cache line. When the QM_CTR is read for that RMID, the monitoring hardware reports an approximation of the amount of the L3 cache contents currently associated with that RMID in the L3 QOS Domain of the processor performing the read.

If two processors in the same L3 QOS Domain are running with different RMIDs and both processors access a given cache line, that line may be associated with either of the RMIDs. Subsequently, the line may remain associated with the RMID with which it was first associated, or its RMID association may change over time, depending on cache access patterns.

Similarly, if the RMID of a processor is changed but that processor continues to access cache lines which have been associated with the previous RMID, those cache lines may continue to be reported as belonging to the first RMID for an indefinite period.

If software changes the of association of RMIDs with processors such that an RMID is no longer in use by any processor, the L3 occupancy count for that unused RMID will not immediately drop to zero. Lines which were installed in the L3 cache and associated with that RMID may remain in the cache until they are replaced through normal cache replacement mechanisms or software explicitly flushes the lines using CLFLUSH, WBINVD, or INVD instructions.

Note: *L3 cache occupancy as reported by the monitoring hardware is an approximation. Software must not assume that an L3 occupancy measurement of zero is an indication that no lines remain in the cache which were installed by processors running for a given RMID. The L3 cache occupancy monitor should never be used to determine whether cache flushing may be skipped in situations which would ordinarily require cache flushing for functional*

[AMD Public Use]

correctness.

After writing the corresponding EvtID and RMID into the QM_EVTSEL register, L3 occupancy data is reported via the QM_CTR register. The QM_CTR E bit will be set if system software has specified an illegal RMID or an undefined or unsupported EvtID in QM_EVTSEL. If the E bit is set, the rest of the QM_CTR value should be ignored.

The U bit may be set on any QM_CTR read. Subsequent reads issued to the same RMID and EvtID may have the U bit cleared. The U bit is set when the monitoring hardware does not have an accurate count for the selected event.

19.3.3 Monitoring L3 Memory Bandwidth (MBM)

The L3 MB mechanism counts requests made by the L3 cache to the lower-level caches and memory (not including I/O). Processors supporting L3 memory bandwidth monitoring are identified by Fn0000_000F_EDX_x1 bit 1 [L3BWMonEvt0] and bit 2 [L3BWMonEvt1] being set. The corresponding Event IDs are EvtID=2 and EvtID=3 respectively.

By default, L3BWMonEvt0 (EvtID 2) tracks total L3 memory bandwidth and L3BWMonEvt1 (EvtId 3) tracks local L3 memory bandwidth. If Bandwidth Monitoring Event Configuration (BMEC) is not implemented, the types of L3 transactions counted for EvtID2 and EvtID3 are fixed. See Section 19.3.3.1. However, if BMEC is supported, the type of bandwidth transactions tracked by these events is configurable. See Section 19.3.3.2.

If Assignable Bandwidth Monitoring Counters (ABMC) are implemented, software can specify which RMIDs are tracked by the implemented MBM hardware counters. See Section 19.3.3.3.

System software reads MBM bandwidth counters using the QM_EVTSEL and QM_CTR registers as described in the following sections.

For all bandwidth monitoring modes, the value returned in QM_CTR must be multiplied by the scaling factor, CPUID Fn0000_000F_EBX_x1[ScalingFactor](bits 31:0), to obtain the number of bytes used by the selected bandwidth source.

To obtain a bandwidth, software should read the Time Stamp Counter (TSC) and the L3 Cache Bandwidth count, wait some period of time, and then read the TSC and the L3 Cache Bandwidth count again. To obtain the rate, software should then divide the difference between the count values by the sampling period.

In PQOS Version 2.0 or higher, the MBM hardware will set the U bit on the first QM_CTR read when it begins tracking an RMID that it was not previously tracking. The U bit will be zero for all subsequent reads from that RMID while it is still tracked by the hardware. Therefore, a QM_CTR read with the U bit set when that RMID is in use by a processor can be considered 0 when calculating the difference with a subsequent read. The U bit will not be set when a counter rolls over. For these implementations, if the U bit is clear on QM_CTR read for a given RMID and event, the MBM hardware has been counting the event for that RMID continuously without interruption since the previous QM_CTR read for that RMID and that event.

A given implementation may have insufficient hardware to simultaneously track the bandwidth for all

[AMD Public Use]

RMID values that the hardware supports. If an attempt is made to read a Bandwidth Count for an RMID that has been impacted by these hardware limitations, the U bit of the QM_CTR will be set when the counter is read. Subsequent QM_CTR reads for that RMID and Event may return a value with the U bit clear. Potential causes of the U bit being set include (but are not limited to):

- RMID is not currently tracked by the hardware.
- RMID was not tracked by the hardware at some time since it was last read.
- RMID has not been read since it started being tracked by the hardware.

For bandwidth monitoring events which continually increment the counter, when the maximum count is reached the counter will roll over and continue counting. It is the software's responsibility to read the count often enough to avoid having the count roll over twice between reads. The initial counter size is 24 bits and retrieving the value at 1Hz or faster is sufficient to ensure at most one rollover per sampling period.

19.3.3.1 Legacy L3 Cache Bandwidth Monitoring

In early implementations where BMEC is not supported, the types of L3 transactions counted by the hardware for EvtID 2 and EvtID 3 are fixed. EvtID 2 tracks total L3 memory bandwidth and EvtID 3 tracks local L3 memory bandwidth. The precise transactions counted by each EvtID depends on the PQOS version. Table 19-8specifies which transactions are counted by a given EvtID for implementations that do not support BMEC. These implementations are identified by the PQOS Version as determined by the Family/Model of the product (see Table 19-2 on page 725).

Table 19-8. Transaction Types and EvtIDs

Transaction Type	V1.0 EvtIDs	V2.0 EvtIDs
Read to Local NUMA Domain	2,3	2,3
Read to Non-Local NUMA Domain	2	2
Non-Temporal Write to Local NUMA Domain	N/A	2,3
Non-Temporal Write to Non-Local NUMA Domain	N/A	2

System software reads the total and local bandwidth using the QM_EVTSEL and QM_CTR registers. The bandwidth metric can be read by setting the following fields in QM_EVTSEL: [ExtendedEvtID]=0, [EvtID]={2,3}, and setting [RMID] to the desired RMID. Reading QM_CTR will then return the contents of the specified event. The E bit will be set if an illegal RMID was specified in QM_EVTSEL.

19.3.3.2 Bandwidth Monitoring Event Configuration (BMEC)

The BMEC feature allows software to configure the specific types of L3 transactions that are counted for L3 bandwidth monitoring.

The BMEC feature implements a set of *n* MSRs (QOS_EVT_CFG_n) which specify the transaction types to be counted for each L3 bandwidth monitoring event. QOS_EVT_CFG_n configures the L3CacheBwMonEvt<n> encoding of QM_EVTSEL.EvtID. For example, L3BWMonEVT1

[AMD Public Use]

(corresponding to EvtID=3) may be configured using the QOS_EVT_CFG_1 register. The format of the QOS_EVT_CFG_n MSRs is shown in Figure 19-5 on page 738.

Table 19-9 summarizes the BMEC Events and their associated names, EvtIDs, configuration MSRs, and default settings.

Table 19-9. BMEC Events and Corresponding Attributes

BMEC event	BMEC name	EvtID	Configuration MSR	Default (reset) Configuration
L3BWMonEVT0	L3BW Total	2	QOS_EVT_CFG_0 MSR C000_0400h	Tracks Total L3 Bandwidth
L3BWMonEVT1	L3BWLocal	3	QOS_EVT_CFG_1 MSR C000_0401h	Tracks Local L3 Bandwidth

Detecting Support for BMEC

Support for BMEC is indicated by CPUID Fn8000_0020_EBX_x0[BMEC] (bit 3) =1.

The number of BMEC configuration registers and the types of transactions that can be counted are implementation dependent and identified by CPUID.

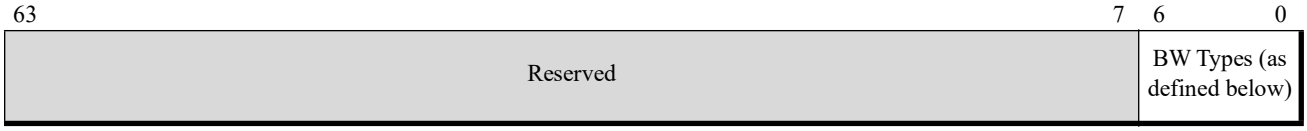
- CPUID Fn8000_0020_EBX_x3[EVT_NUM](bits 7:0) identifies the number of QOS_EVT_CFG_n registers available for BMEC use.
- CPUID Fn8000_0020_ECX_x3 identifies the types of L3 transactions that can be counted.

For implementations that support at least two BMEC configuration registers (EVT_NUM >= 2), QOS_EVT_CFG_0 is reset to a value such that L3CacheBwMonEvt0 reports Total L3 System Memory Bandwidth and QOS_EVT_CFG_1 is reset to a value such that L3CacheBwMonEvt1 reports L3 System Memory Bandwidth to the Local NUMA domain. See Figure 19-5 for the specific reset values.

BMEC MSRs

The BMEC feature implements a set of *n* configuration registers, QOS_EVT_CFG_n, where <n> is the BMEC event number (EvtID). The BMEC configuration registers are used to specify the Bandwidth Types to be counted for each BMEC EvtID. The MSR address of QOS_EVT_CFG_n is C000_0400h + n.

The BMEC configuration registers contain a bit for each Bandwidth (BW) Type that can be used to configure the BMEC EvtID (see Figure 19-5). The Bandwidth Types that can be tracked are implementation dependent and enumerated by CPUID Fn8000_0020_ECX_x3. Bit locations that are 0 in the CPUID field indicate that the corresponding bit positions in QOS_EVT_CFG_n are reserved and MBZ.



[AMD Public Use]

Bits	Mnemonic	Description	Access type
63:7	–	Reserved	MBZ
6	VictimBW	Dirty victim writes to all types of memory	RW
5	RmtSlowFill	Reads issued to remote memory the system identifies as “Slow Memory”	RW
4	LclSlowFill	Reads issued to local memory the system identifies as “Slow Memory”	RW
3	RmtNTWr	Non-temporal writes issued to remote memory	RW
2	LclNTWr	Non-temporal writes issued to local memory	RW
1	RmtFill	Reads issued to remote DRAM memory	RW
0	LclFill	Reads issued to local DRAM memory	RW

Notes:
QOS_EVT_CFG_0 is reset to a value such that L3CacheBwMonEvt0 (EvtID=2) reports Total L3 memory bandwidth.
QOS_EVT_CFG_1 is reset to a value such that L3CacheBwMonEvt1 (EvtID=3) reports Local L3 memory bandwidth.

Figure 19-5. QOS_EVT_CFG_n

The fields in the QOS_EVT_CFG_n registers are further described below:

- BwType (Bandwidth Types) – bits 6:0, read/write. This field is a bit map specifying the types of L3 bandwidth events to be tracked. The Bandwidth Types that can be tracked are implementation dependent and software must use CPUID Fn8000_0020_ECX_x3 to determine if a particular Bandwidth Type is supported. Bit locations that are 0 in the CPUID field indicate that the corresponding bit positions in QOS_EVT_CFG_n are reserved and MBZ.

19.3.3.3 Assignable Bandwidth Monitoring (ABMC)

The ABMC feature implements a set of *n* counters to monitor L3 bandwidth in the QOS Domain. Each of these counters can be assigned by system software to a particular source RMID (or COS, if supported) to track the memory bandwidth used by that source.

ABMC gives system software full control over the bandwidth monitoring resources in the QOS Domain. When enabled, the bandwidth monitoring hardware will only track memory bandwidth used by the sources and Bandwidth Types configured through the L3_QOS_ABMC_CFG_n MSRs. Bandwidth data may still be accessed by specifying an RMID and EvtID, but may also be accessed by specifying an ABMC CtrId. (See “Reading the ABMC Counters” on page 742 for more details)

Detecting Support for ABMC

Support for ABMC is identified by CPUID Fn8000_0020_EBX_x0[ABMC] (bit 5) being set. If ABMC is supported, the feature’s attributes and capabilities are enumerated by CPUID Fn8000_0020_x5 as detailed in Appendix E of APM Volume 3, and summarized below:

- EAX returns the size of the assignable counters and the presence of a counter overflow bit.
- EBX returns the number of ABMC counters supported.
- ECX identifies the capability to associate individual assignable counters with a COS (Class of Service) rather than an RMID.

[AMD Public Use]

The types of L3 transactions that ABMC can track are configurable and identified by CPUID Fn8000_0020_ECX_x3. See “QOS_EVT_CFG_n” on page 738 for a list of transaction types and corresponding encodings.

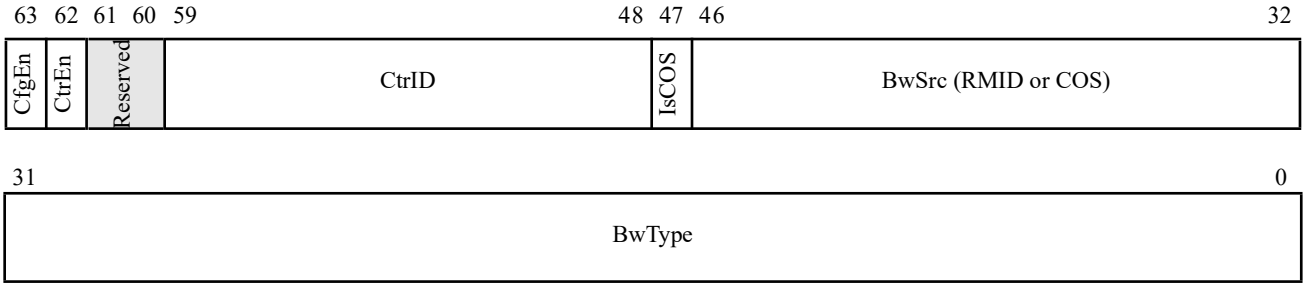
AMBC MSRs

The ABMC feature implements a pair of MSRs, L3_QOS_ABMC_CFG (MSR C000_03FDh) and L3_QOS_ABMC_DSC (MSR C000_3FEh). Each logical processor implements a separate copy of these registers.

Attempts to read or write these MSRs when ABMC is not enabled will result in a #GP(0) exception.

Individual assignable bandwidth counters are configured by writing to L3_QOS_ABMC_CFG MSR and specifying the Counter ID, Bandwidth Source, and Bandwidth Types.

Reading L3_QOS_ABMC_DSC returns the configuration of the counter specified by L3_QOS_ABMC_CFG [CtrID].



Bits	Mnemonic	Description	Access Type	Reset Value
63	CfgEn	Configuration Enable	R/W	0
62	CtrEn	Counter Enable	R/W	0
61:60	–	Reserved	MBZ	0
59:48	CtrID	Counter Identifier	R/W	0
47	IsCOS	BwSrc field is a COS (not an RMID)	R/W	0
46:32	BwSrc	Bandwidth Source (RMID or COS)	R/W	0
31:0	BwType	Bandwidth types to track for this counter	R/W	See BwType below

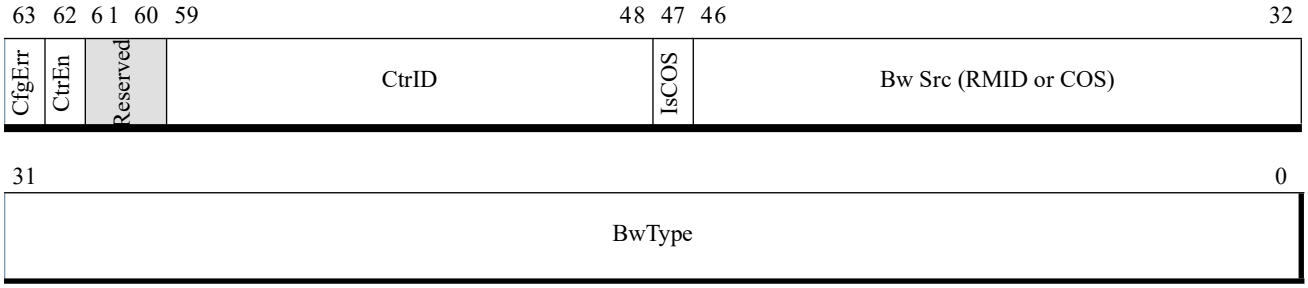
Figure 19-6. L3_QOS_ABMC_CFG

The fields shown in Figure 19-6 are further described below:

- CfgEn (Configuration Enable) – bit 63. When set, indicates a WRMSR to this register should configure the assignable counter selected by the CtrID field. If this bit is not set on a write, no configuration is performed.
- CtrEn (Counter Enable) – bit 62. When set, the selected counter will begin tracking the bandwidth types specified in the BwType field.

[AMD Public Use]

- CtrID (Counter ID) – bits 59:48. Identifies the assignable counter to configure. If the value written to CtrID exceeds CPUID Fn8000_0020_EBX_x5[MAX_ABMC], a #GP(0) exception is signaled.
- IsCOS (BwSrc is a COS) – bit 47. When set, indicates that the BwSrc field is a COS (Class of Service) as opposed to an RMID.
- BwSrc (Bandwidth Source) – bits 46:32. Specifies an RMID (or COS if IsCOS=1) to monitor with this counter. If the value written to BwSrc exceeds CPUID Fn0000_000F_EBX_x0[MAX_RMID], a #GP(0) exception is signaled.
- BwType (Bandwidth Types – bits 31:0. This field is a bit map specifying the types of L3 bandwidth events to be tracked by this assignable counter. This field uses the Bandwidth Type encodings used in CPUID Fn8000_0020_ECX_x3[BandwidthTypes]. Upon reset, the bit for each supported Bandwidth Type enumerated by the above CPUID function is set to 1.



Bits	Mnemonic	Description	Access Type	Reset Value
63	CfgErr	Configuration Error	R	0
62	CtrEn	Counter Enable	R	0
61:60	–	Reserved	R	0
59:48	CtrID	Counter Identifier	R	0
47	IsCOS	BwSrc field is a COS (not an RMID)	R	0
46:32	BwSrc	Bandwidth Source (RMID or COS)	R	0
31:0	BwType	Bandwidth types to track for this counter	R	See BwType below

Figure 19-7. L3_QOS_ABMC_DSC

The fields shown in Figure 19-7 are further described below:

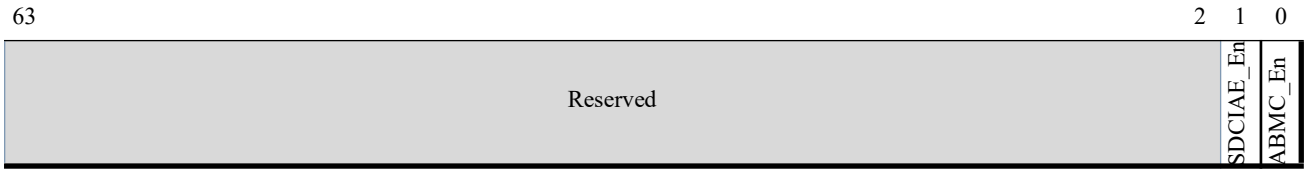
- CfgErr (Configuration Error) – bit 63. When set, indicates that the last configuration specified in L3_QOS_ABMC_CFG was invalid and consequently the specified counter was not enabled.
- CtrEn (Counter Enable) – bit 62. When set, indicates the selected counter is actively tracking the bandwidth types specified in the BwType field. When cleared, the selected counter is also zero.
- CtrID (Counter ID) – bits 59:48. Identifies the assignable counter for which configuration is reported.
- IsCOS (BwSrc is a COS) – bit 47. When set, indicates that the BwSrc field is a COS (Class of Service) as opposed to an RMID.

[AMD Public Use]

- BwSrc (Bandwidth Source) – bits 46:32. Returns the RMID (or COS if IsCOS=1) being monitored with this counter.
- BwType (Bandwidth Types – bits 31:0. This field is a bit map which identifies the types of L3 bandwidth events to be tracked by the specified counter. The BwType field uses the same event encodings used in CPUID Fn8000_0020_ECX_x3[BandwidthMonitoringEvents].

L3 _QOS_EXT_CFG MSR

If support for ABMC is identified (CPUID Fn8000_0020_EBX_x0[ABMC](bit 5)=1), then the L3_QOS_EXT_CFG MSR, (MSR address C000_03FFh) is also supported. The format of this register is shown in Figure 19-8.



Bits	Mnemonic	Description	Access Type
63:2	–	Reserved	RAZ
1	SDCIAE_En	SDCI Allocation Enforcement Enable	R/W
0	ABMC_En	ABMC Enable	R/W

Figure 19-8. L3_QOS_EXT_CFG

Enabling ABMC

ABMC is enabled by setting L3_QOS_EXT_CFG.ABMC_En=1 (see Figure 19-8). When the state of ABMC_En is changed, it must be changed to the updated value on all logical processors in the QOS Domain.

Upon transitions of the ABMC_En the following actions take place:

- All ABMC assignable bandwidth counters are reset to 0.
- The L3 default mode bandwidth counters are reset to 0.
- The L3_QOS_ABMC_CFG MSR is reset to 0.

Configuring ABMC

After enabling ABMC, system software can configure a counter by writing the QOS_ABMC_CFG register and specifying the Counter ID, Bandwidth Source, and Bandwidth Types fields and setting the Configuration Enable bit. The value of the specified Counter ID is cleared to zero upon configuration. Writing QOS_ABMC_CFG with CfgEn clear will not change the configuration of the counter. Reading QOS_ABMC_CFG will return the last value written (see Figure 19-6 on page 739).

Each logical processor has a separate copy of QOS_ABMC_CFG. The contents of this register are not persistent through PC6 events and will be reset in that case. Attempts to access

[AMD Public Use]

L3_QOS_ABMC_CFG when ABMC is not enabled (L3_QOS_EXT_CFG.ABMC_En == 0) result in a #GP(0) exception.

Reading L3_QOS_ABMC_DSC will return the configuration of the counter specified by QOS_ABMC_CFG[CtrID] (see Figure 19-7 on page 740).

Reading the ABMC Counters

System software reads the assignable counters using the QM_EVTSEL and QM_CTR registers.

The contents of a specific counter can be read by setting the following fields in QM_EVTSEL: [ExtendedEvtID]=1, [EvtID]=L3CacheABMC and setting [RMID] to the desired counter ID. Reading QM_CTR will then return the contents of the specified counter. The E bit will be set if the counter configuration was invalid, or if an invalid counter ID was set in the QM_EVTSEL[RMID] field.

Alternatively, the contents of a counter may be read by specifying an RMID and setting the [EvtID] to L3BWMonEvt*n* where *n*= {0,1}. If an assignable bandwidth counter is monitoring that RMID with a BwType bitmask that matches a QOS_EVT_CFG_n, that counter's value will be returned when reading QM_CTR. However, if multiple counters have the same configuration, QM_CTR will return the value of the counter with the lowest CtrID.

The value returned in QM_CTR must be multiplied by the scaling factor, Fn0000_000F_EBX_x1[ScalingFactor](31:0), to obtain the number of bytes used by the selected bandwidth source.

19.4 PQOS Enforcement (PQE)

The PQE feature provides the ability to set and enforce limits on the usage of shared resources by a thread or a group of threads. For example, PQE can be used to specify the amount of L3 cache that is available to a thread.

Enforcement is accomplished by assigning a numerical Class of Service (COS) to a thread and specifying allocations or limits on the usage of a shared resource for that COS. The current COS for a given processor is specified in PQR_ASSOC register, MSR C8Fh (see Figure 19-1 on page 729). If multiple threads share the same COS, those threads will competitively share the controlled resource.

PQE implements several sub-features which are described in detail in the following sections:

- CAT – L3 Cache allocation.
- CDP – L3 Cache allocation, based on code vs. data access.
- L3 bandwidth allocation.
- L3 slow memory bandwidth allocation.
- SDCI allocation enforcement.

For each resource controlled by PQE, a set of Model-Specific Registers is defined which system software can use to specify the allocation limits for each COS.

[AMD Public Use]

19.4.1 Identifying support for PQE

Support for PQE is identified by CPUID Fn0000_0007_EBX_x0[PQE]=1. If PQE is supported, the set of resources for which limits can be set and enforced is reported by CPUID function Fn0000_0010. Each supported resource is identified by a bit in EBX, as shown in Table 19-10.

Table 19-10. CPUID Fn0000_00010_EDX_x0 PQE Resources

EDX Bits	Name	Description
0	–	Reserved
1	L3Alloc	L3 Cache Allocation Enforcement
31:2	–	Reserved

Once support for a PQE resource has been determined as above, CPUID Fn0000_0010 with ECX = {resource bit number} may be used to determine the PQE enforcement capabilities for that resource. For example, Fn0000_00010_x1 (ECX=1) returns the following information about the L3 Cache Monitoring feature:

- EAX identifies CBM_LEN, the length of the L3 cache capacity bitmasks (CBMs).
- EBX identifies the L3 Cache Allocation Sharing Mask.
- ECX identifies the presence of specific L3 Cache Enforcement features, such as Code-Data Prioritization (CDP) support.
- EDX identifies COS_MAX, the maximum COS supported for L3 Cache Allocation Enforcement.

A complete description of these CPUID functions can be found in Appendix E of APM Volume 3.

19.4.2 L3 Cache Allocation Enforcement (CAT)

The L3 CAT feature enables system software to limit the fractional portion of the L3 cache used by threads for each COS.

Limits for L3 cache allocation are specified by a set of L3_MASK_n registers, where *n* is the COS associated with that MSR. There is one L3_MASK register for each COS implemented for the L3 cache resource. This set of registers starts at MSR address C90h and continues through address C90h + (n-1). The maximum COS value supported by the L3 resource is reported by CPUID Fn0000_0010_EDX_x1[MAX_COS](bits 15:0). COS values are zero-based, thus the number of classes supported is (MAX_COS+1).

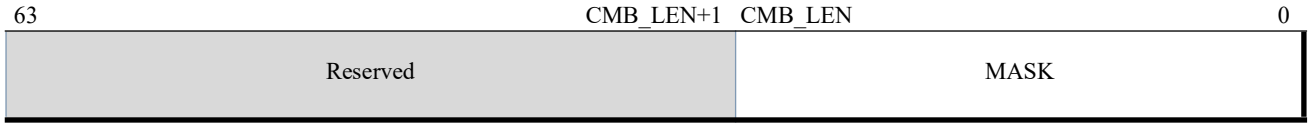
Each of the L3_MASK_n registers contain a bitmask where each bit represents a fixed portion of the L3 cache into which COS <n> is allowed to allocate. The length of the bit mask is implementation-dependent and returned by CPUID Fn0000_0010_EAX_x1[CBM_LEN](bits 4:0). The length is zero based, thus CBM_LEN=15 would indicate that the cache is partitioned into 16 sections for L3 allocation purposes.

Once the L3_MASK_n registers are configured with the desired L3 cache allocations, threads can be assigned a COS using the PQR_ASSOC register as described in Section 19.3.1 “PQM MSRs” on page 728.

[AMD Public Use]

19.4.2.1 L3_MASK_n MSRs

The format of the L3_MASK_n registers is shown in Figure 19-9 below.



Bits	Mnemonic	Description	Access Type
63:CBM_LEN+1	–	Reserved, read as zero	RAZ
CBM_LEN:0	MASK	L3 Cache Allocation Mask	R/W

Figure 19-9. L3_MASK_n

19.4.2.2 Using L3 CAT

System software sets bits in the L3_MASK_n registers to indicate the portions of the L3 cache that may be used by a given COS <n>. For example, if CBM_LEN for a given implementation is 15, then each set bit in the L3_MASK_n represents 1/16 of the cache which may be used by processors running with COS = <n>.

If two or more different Classes of Service have set bits in the same position in their respective L3_MASK_n registers, that portion of the cache is competitively shared by logical processors running with those COS values.

After configuring the L3_Mask_n registers with the desired L3 allocations, system software can set PQR_ASSOC[COS] to specify the Class of Service for the currently active thread. If the value written to PQR_ASSOC[COS] exceeds CPUID Fn0000_00010_EDX_x1[MAX_COS], a #GP(0) exception is signaled.

Some products may implement L3 Cache Allocation Enforcement by allocating some number of ways of the L3 cache for each set bit in the L3_MASK register, but this will not necessarily be true for all implementations and software should not rely on that interpretation. However, because this is one possible implementation, it is possible that use of CAT will reduce the effective associativity available to processes running using an L3_MASK which does not have all the bits set.

The bits which are set in the various L3_MASK_n registers do not have to be contiguous and may overlap in any desired combination. If an L3_MASK_n register is programmed with all bits clear, that COS will be prevented from allocating any lines in the L3 cache. At reset, the L3_MASK register bitmap bits are all set, allowing all processors to use the entire L3 cache accessible to them.

The L3 Cache Allocation mechanism controls only the allocations into the L3 cache. Once a line is installed into the cache, any COS is allowed to “hit” on that line.

When a given cache line is accessed by processors running with different COS values in the same QOS Domain, only a single copy of the line will be allocated in the shared cache at a given time. Depending on the access pattern, this line may be installed in the L3 cache partition belonging to

[AMD Public Use]

either COS. Depending on subsequent access patterns, that cache line may later be evicted and then installed into the partition belonging to the other COS. Software should not assume that the line will always be associated with the last COS that accessed the line.

Depending on the implementation, some portions of the L3 Cache may be shared by other system functions or used for some other purpose not under the control of the PQOS feature set. The L3 Cache Allocation Sharing Mask returned by CPUID Fn0000_0010_EBX_x1[L3ShareAllocMask] is a bitmask that represents portions of the L3 that may be shared by those functions. When software sets a bit in an L3_MASK_n register at the same position as a bit in the L3ShareAllocMask, processors executing with the corresponding COS will competitively share that portion of the cache with the other function. An L3ShareAllocMask with all bits cleared indicates that no other entity in the system is competing with the processors for use of the L3 cache.

19.4.3 Code and Data Prioritization (CDP)

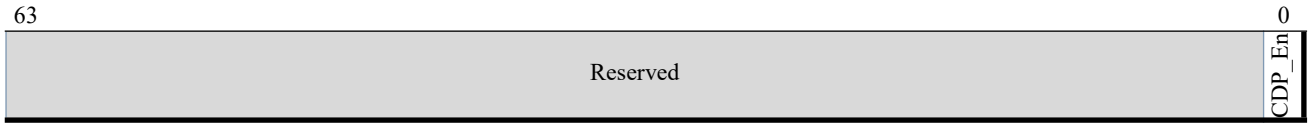
The L3 CDP feature is an extension of the previously-described L3 CAT feature (See Section 19.4.2). When enabled, CDP allows system software to specify different L3 cache allocations for instructions versus data.

19.4.3.1 Detecting Support for CDP

Presence of the L3 CDP feature is indicated by CPUID Fn0000_0010_ECX_x1[CDP](bit 2) being set. The L3 CAT information returned by Fn0000_0010_{EAX, EBX, EDX}_x1 also applies to the L3 CDP feature, except for MAX_COS. When CDP is enabled, the maximum COS value is half the value given by CPUID Fn0000_0010_EDX_x1[MAX_COS] + 1.

19.4.3.2 CDP Model-Specific Registers

If CDP is supported, software can enable the CDP feature by setting the CDP_En bit in L3_QOS_CFG1 (MSR C81h). See Figure 19-10.



Bits	Mnemonic	Description	Access Type
63:1	–	Reserved, read as zero	RAZ
0	CDP_En	CDP Enable	R/W

Figure 19-10. L3_QOS_CFG1

When CDP is enabled, the L3_MASK_n registers operate in pairs, and each pair is associated with a single COS. The lower-numbered register in the pair is used to specify the cache allocation mask for data accesses and the upper register of the pair is used to specify the mask for instruction fetches. For a given COS, the data allocation mask will be specified in MSR C90 + (2*COS) while the instruction allocation mask will be specified by MSR C90 + (2*COS+1). See Table 19-11 below.

[AMD Public Use]

Note that enabling CDP reduces the effective number of unique COS values by half. Specifying a COS outside of the valid range will result in undefined behavior.

The interpretation of the L3_MASK registers is the same with CAT; the difference is that with CDP enabled, the mask to be applied is selected based on whether the line is accessed as an instruction or data fetch. Similar to the behavior with CDP disabled, if a line is accessed as both code and data, then the line may be allocated using either mask register and be counted against that allocation limit. Also similar to the case with CDP disabled, software may not assume that cache lines will necessarily be associated with the code or data mask register with which they were most recently accessed.

Table 19-11. CAT vs. CDP L3 Mask Register Usage

Mask Register	MSR Address	CAT Usage	CDP Usage	
L3_Mask_0	C90h	COS 0	COS 0	data
L3_Mask_1	C91h	COS 1		code
L3_Mask_2	C92h	COS 2	COS 1	data
L3_Mask_3	C93h	COS 3		code
.	.	.	.	.
.	.	.	.	.
L3_Mask_n	C90+n	COS n	COS n/2	data
L3_Mask_n+1	C90+n+1	COS n+1		code

Software should observe the following sequence when enabling or disabling CDP mode:

1. Ensure that all bits of the QOS Allocation mask for each COS (L3_MASK_<COS>[MASK] are set.
2. Ensure that the QOS Bandwidth enforcement limit for each COS is set to “No Limit”. (L3QOS_BW_CONTROL_<COS>[U] = 1)
3. Ensure that all logical processors in the QOS Domain are associated (PQR_ASSOC[COS] with valid COS numbers for when CDP is enabled. Specifically, the COS number must be less than half of COS_MAX. Using COS numbers outside that range will result in undefined behavior.
4. Write the desired state of CDP_En to the L3_QOS_CFG1 MSR on each logical processor in the QOS Domain. In some implementations, a single L3_QOS_CFG1 MSR may be shared between multiple logical processors.

For optimal QOS behavior in the new operating mode, software should flush the caches in the QOS Domain once the new configuration has been enabled to clear out any residual allocations from the previous configuration.

19.4.4 AMD Bandwidth Enforcement

The AMD Bandwidth Enforcement feature provides a set of sub-features to control threads which may be over-utilizing bandwidth relative to their priority with respect to other workloads co-located in the QOS Domain. When support for PQE is identified (see Section 19.4.1 “Identifying support for

[AMD Public Use]

PQE” on page 743) CPUID Fn8000_0008_EBX_x0[BE](bit 6) returns an indication that AMD Bandwidth Enforcement is supported.

The sub-features supported under AMD Bandwidth Enforcement are further enumerated by CPUID function 0x8000_0020_EBX_x0. The supported sub-features include:

- L3 BE - L3 External Bandwidth Allocation Enforcement (see Section 19.4.5 “L3 External Bandwidth Enforcement (L3BE)” on page 747)
- L3 SMBE - L3 External Slow Memory Bandwidth Allocation Enforcement (see Section 19.4.6 “L3 Slow Memory Bandwidth Enforcement (L3SMBE)” on page 749)
- L3 SDCIAE – L3 Smart Data Cache Injection Allocation Enforcement (see Section 19.4.7 “L3 Smart Data Cache Injection Allocation Enforcement (SDCIAE)” on page 750)
- GLBE - Global Bandwidth Enforcement (see Section 19.4.8 “Global Bandwidth Enforcement (GLBE)” on page 751)
- GLSBE - Global Slow Bandwidth Enforcement (see Section 19.4.9 “Global Slow Bandwidth Enforcement (GLSBE)” on page 754)

19.4.5 L3 External Bandwidth Enforcement (L3BE)

L3BE allows system software to specify memory bandwidth limits used by a thread (or a group of threads) assigned to a given Class of Service.

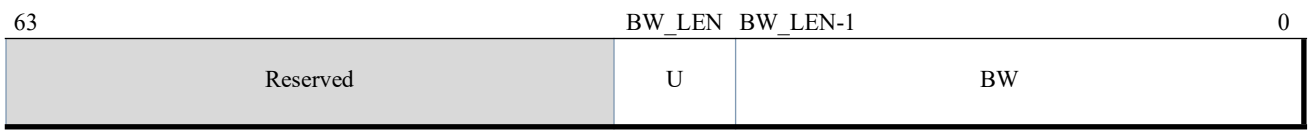
Support for L3BE is indicated by CPUID Fn8000_0020_EBX_x0[L3BE](bit 1)=1. If L3BE is supported, the feature’s attributes and capabilities are enumerated by ID Fn8000_0020_x1 as summarized below:

- EAX[31:0](BW_LEN) returns the size of the bandwidth specifier field in the L3QOS_BW_Control_n MSRs.
- EDX[31:0](COS_MAX) returns the maximum COS number supported by the L3BE feature.

19.4.5.1 L3BE MSRs

Bandwidth limits are specified using the L3QOS_BW_CONTROL_n, MSRs, where <n> is the corresponding COS number. This set of registers starts at MSR address C000_0200h and continues through C000_0200h + n. The format of the L3QOS_BW_CONTROL_n registers is shown in Figure 19-11.

[AMD Public Use]



Bits	Mnemonic	Description	Access Type
63:BW_LEN+1	–	Reserved, read as zero	RAZ
BW_LEN	U	Unlimited bandwidth	R/W
BW_LEN-1:0	BW	L3 cache external bandwidth limit	R/W

Figure 19-11. L3QOS_BW_CONTROL_n

The fields within a L3QOS_BW_CONTROL_n register are:

- U (Unlimited) – bit <BW_LEN>. When set, indicates that threads belonging to this COS are unlimited in bandwidth and the contents of the BW field are ignored. At reset, the U bit for all L3QOS_BW_CONTROL_n MSRs are set.
- BW (Bandwidth) – bits BW_LEN-1:0. Specifies a limit on the total L3 external bandwidth, expressed in 1/8 GB/s increments, for all threads running under COS <n>.

19.4.5.2 Using L3BE

System software can control usage of L3 external bandwidth using the L3QOS_BW_CONTROL MSRs to specify bandwidth limits for a given COS. There is one L3QOS_BW_CONTROL for each COS as identified by CPUID Fn8000_0020_EDX_x1[COS_MAX](bits 31:0).

At reset, the “U” (Unlimited) bit of all L3QOS_BW_CONTROL registers is set, allowing maximum bandwidth usage by all processors.

The value programmed in the L3QOS_BW_CONTROL_n register specifies a limit on the total L3 external bandwidth consumed by all threads running with COS=<n> within the L3 cache QOS Domain. Software may program a bandwidth limit for COS <n> in L3QOS_BW_CONTROL[BW]_n, or it may specify “no limit” by setting the “U” bit. Unlike cache allocation (which specifies a fraction of the cache), this limit is not a relative bandwidth, but an absolute number expressed in 1/8 GB/s increments. The format of the L3QOS_BW_CONTROL_n registers are described in Figure 19-11 on page 748.

Bandwidth limits are upper bounds on the bandwidth the processors in a given domain and COS may consume, but do not ensure that the specified bandwidth will necessarily be available to those processors. Cases where the processors in a given COS may not be able to reach their allocated bandwidth include (but are not limited to):

1. The specified limit may be greater than the maximum system bandwidth.
2. The sum of the limits applied to all classes of service in the domain may exceed the maximum bandwidth the system can deliver to that COS domain.

[AMD Public Use]

3. Multiple COS domains which share the same memory channels may demand more total bandwidth than the shared memory can supply.
4. I/O or other system entities may consume a large fraction of system bandwidth and result in less bandwidth being available to the various processor COS domains.
5. Large amounts of write traffic may affect the memory system's ability to deliver read bandwidth.

In the case where total available L3 external bandwidth has been over-subscribed (that is, not all classes of service can reach their bandwidth limits), the hardware makes no attempt to allocate the available L3 bandwidth in proportion to the limits specified for each COS domain or the limits specified for each COS within a given domain. That is (in case 2 above), if COS *x* has limit A, and COS *y* has limit 2*A, and the total bandwidth which the system can deliver to that COS is only 2*A, the system will only limit COS *x* to A; it will not attempt to give (2/3)*A to COS *x* and (4/3)*A to COS *y* (maintaining the ratio of their limits).

19.4.5.3 CDP Interaction with L3BE

When CDP is enabled, the mapping of COS to its associated L3QOS_BW_CONTROL register is changed. For a given COS, the bandwidth limits are specified by MSR C000_0200h + (2*COS). For example, if CDP is enabled, the L3 bandwidth limit for threads associated with COS 3 are specified in L3QOS_BW_CONTROL_6.

Note that enabling CDP reduces the effective number of unique COS values by half. Specifying a COS outside of the valid range will result in undefined behavior.

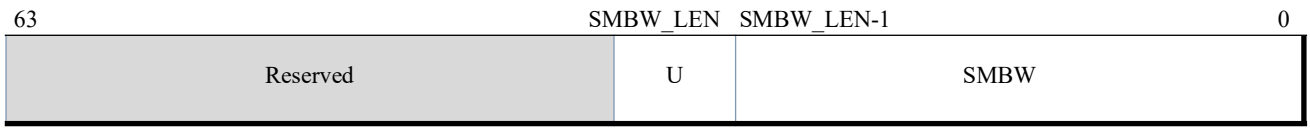
19.4.6 L3 Slow Memory Bandwidth Enforcement (L3SMBE)

L3SMBE allows the system software to limit bandwidth to slow memory available to threads inside the QOS Domain. Support for L3SMBE is indicated by CPUID Fn8000_0020_EBX_x0[L3SMBE](bit 2)=1. If L3SMBE is supported, the feature's attributes and capabilities are enumerated by ID Fn8000_0020_x2 as summarized below:

- EAX[31:0](BW_LEN) returns the size of the bandwidth specifier field in the L3QOS_SMBW_Control_n MSRs.
- EDX[31:0](COS_MAX) returns the maximum COS number for the L3SMBE feature.

19.4.6.1 L3SMBE MSRs

Similar to the L3BE feature (see Section 19.4.5), L3SMBE bandwidth limits are specified in the L3QOS_SMBW_CONTROL_n, MSRs, where <n> is the corresponding COS number. This set of registers starts at MSR address C000_0280h and continues through C000_0280h + n. The format of the L3QOS_SMBW_CONTROL_n registers is shown in Figure 19-12.



[AMD Public Use]

Bits	Mnemonic	Description	Access Type
63:SMBW_LEN+1	–	Reserved, read as zero	RAZ
SMBW_LEN	U	Unlimited bandwidth	R/W
SWBW_LEN-1:0	SMBW	L3 cache slow memory bandwidth limit	R/W

Figure 19-12. L3QOS_SMBW_CONTROL_n

The fields within an L3QOS_SMBW_CONTROL_n register are:

- U (Unlimited) – bit <SMBW_LEN>. When set, indicates that threads belonging to this COS are unlimited in bandwidth and the contents of the SMBW field are ignored. At reset, the U bit for all L3QOS_SMBW_CONTROL_n MSRs are set.
- SMBW (Bandwidth) – bits SMBW_LEN-131. Specifies a limit on the slow memory bandwidth, expressed in 1/8 GB/s increments, for all threads running under COS <n>.

19.4.6.2 Using L3SMBE

System software can control usage of L3 slow memory bandwidth using the L3QOS_SMBW_CONTROL MSRs to specify bandwidth limits for a given COS. There is one such register for each COS as identified by CPUID Fn8000_0020_EDX_x2[COS_MAX](bits 31:0). L3SMBE usage details and operation are the same as L3BE, except that it controls a subset of bandwidth resources related to slow memory. See Section 19.4.5.2 “Using L3BE” on page 748 for usage notes.

19.4.6.3 CDP Interaction with L3SMBE

Similar to the L3BE feature, the mapping of a COS to its associated L3QOS_SMBW_CONTROL register is changed if CDP is enabled. For a given COS, the slow memory bandwidth limits are specified by MSR C000_0280h + (2*COS). For example, if CDP is enabled, the slow memory bandwidth limit for threads associated with COS 3 are specified in L3QOS_SMBW_CONTROL_6. Note that enabling CDP reduces the effective number of unique COS values by half. Specifying a COS outside of the valid range will result in undefined behavior.

19.4.7 L3 Smart Data Cache Injection Allocation Enforcement (SDCIAE)

Smart Data Cache Injection (SDCI) is a mechanism that enables direct insertion of data from I/O devices into the L3 cache. By directly caching data from I/O devices rather than first storing the I/O data in DRAM, SDCI reduces demands on DRAM bandwidth and reduces latency to the processor consuming the I/O data.

The SDCIAE (SDCI Allocation Enforcement) PQE feature allows system software to limit the portion of the L3 cache used for SDCI.

When enabled, SDCIAE forces all SDCI lines to be placed into the L3 cache partitions identified by the highest-supported L3_MASK_n register as reported by CPUID Fn0000_0010_EDX_x1[MAX_COS](bits 15:0). For example, if MAX_COS=15, SDCI lines will be

[AMD Public Use]

allocated into the L3 cache partitions determined by the bitmask in the L3_MASK_15 register.

Support for SDCIAE is indicated by CPUID Fn8000_0020_EBX_x0[SDCIAE](bit 6)=1.

SDCIAE is enabled by setting L3_QOS_EXT_CFG.SDCIAE_En=1 (see Figure 19-8 on page 741). When the state of SDCIAE_En is changed, it must be changed to the updated value on all logical processors in the QOS Domain.

19.4.8 Global Bandwidth Enforcement (GLBE)

GLBE provides a mechanism for software to specify bandwidth limits for groups of logical processors that span multiple QOS Domains. This collection of QOS Domains is referred to as the GLBE Control Domain. The GLBE Ceiling is a bandwidth ceiling for L3 External Bandwidth competitively shared between all logical processors in a given Class of Service (COS) across all QOS Domains within the GLBE Control Domain. This complements L3 External Bandwidth Enforcement (L3BE) which provides L3External Bandwidth control on a per QOS Domain granularity.

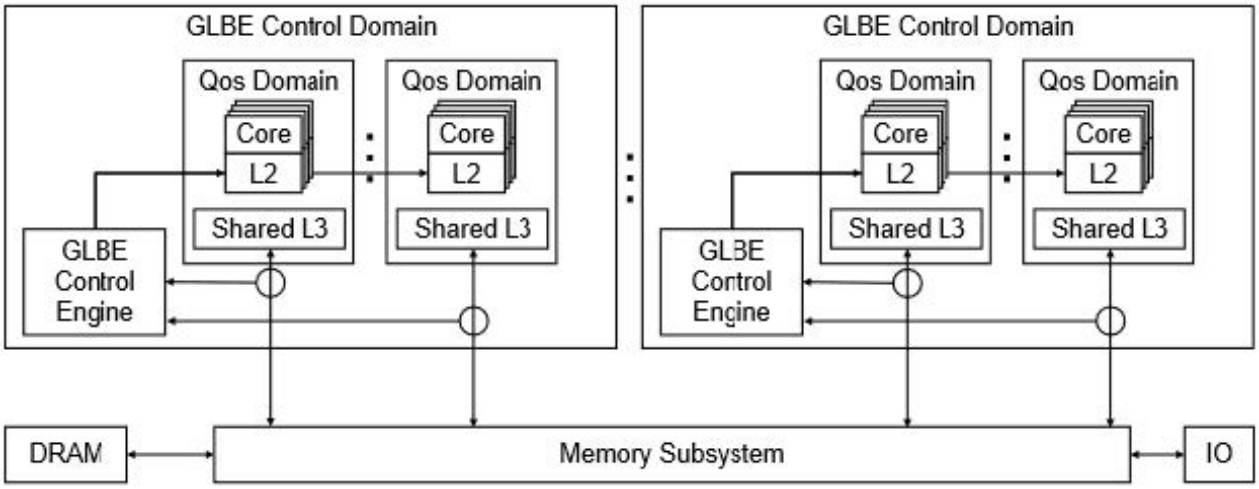


Figure 19-13. GLBE Control Domains within the System

By default, all QOS Domains in the system are included in a single GLBE Control Domain. However, BIOS options may establish several GLBE Control Domains within the system.

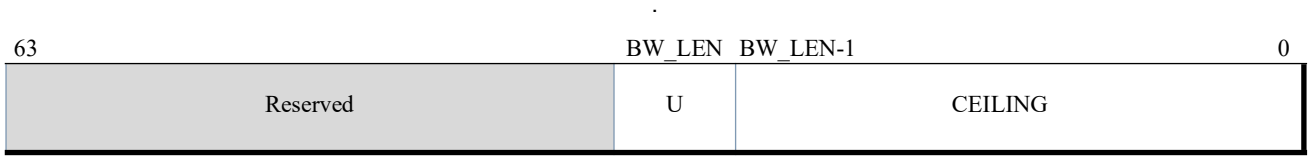
Support for GLBE is indicated by CPUID Fn8000_0020_EBX_x0[GLBE] (bit 7) = 1. If GLBE is supported, the feature's attributes and capabilities are enumerated by CPUID Fn8000_0020_x7 as summarized below:

- EAX[31:0](BW_LEN) returns the size of the bandwidth specifier field in the L3QOS_GL_BW_CONTROL_n MSRs.
- EBX[15:0](BW_MULT) returns the units of the GLBE Ceiling field in the L3QOS_GL_BW_CONTROL_n MSRs; units = 1/8 GB/s * (BW_MULT+1).
- EDX[31:0](COS_MAX) returns the maximum COS number supported by the GLBE feature.

[AMD Public Use]

19.4.8.1 GLBE MSRs

Bandwidth limits are specified per COS using the L3QOS_GL_BW_CONTROL_n MSRs where n is the COS. This set of registers starts at MSR address C000_0600h and continues through C000_0600h + (COS_MAX). The format of the L3QOS_GL_BW_CONTROL_n MSRs is shown below. There is one L3QOS_GL_BW_CONTROL_n register for each COS in each QOS Domain. The maximum COS number supported by GLBE is identified by CPUID Fn8000_0020_EDX_x7[COS_MAX] (bits 31:0).



Bits	Mnemonic	Description	Access Type
63:BW_LEN+1	–	Reserved	MBZ
BW_LEN	U	Unlimited Bandwidth	R/W
BW_LEN-1:0	CEILING	Bandwidth Ceiling	R/W

Figure 19-14. L3QOS_GL_BW_CONTROL_n

The fields within the L3QOS_GL_BW_CONTROL_n register are:

- U (Unlimited Bandwidth) – bit BW_LEN. When cleared, the L3 External Bandwidth of logical processors belonging to this COS counts towards the bandwidth limit configured with CEILING and is limited by GLBE. When set, the L3 External Bandwidth of logical processors belonging to this COS does not count towards the bandwidth limit and is not limited by GLBE. At reset, the U bit for all L3QOS_GL_BW_CONTROL_n MSRs is set. This field may be set to different values in each QOS Domain within a single GLBE Control Domain.
- CEILING (Bandwidth Ceiling) – bits BW_LEN-1:0. Specifies a limit on the combined L3 External Bandwidth consumed by logical processors associated with this COS across all QOS Domains in the GLBE Control Domain. The limit is expressed in units reported by CPUID Fn8000_0020_EBX_x7[BW_MULT] (bits 15:0). This value must be configured to the same value for a given COS in all QOS Domains in the GLBE Control Domain for consistent operation.

19.4.8.2 Using GLBE

To apply a GLBE limit to a set of logical processors, system software should associate all logical processors within that set with the same COS using PQR_ASSOC MSRs as described in Section 19.3.1.1 “PQR_ASSOC” on page 729. Then, system software uses L3QOS_GL_BW_CONTROL_n MSRs to specify the intended ceiling of combined L3 External Bandwidth for this COS and clears the U (Unlimited Bandwidth) bit. The same CEILING value must be configured for a given COS in each QOS Domain within the GLBE Control Domain. Using different CEILING values in different QOS Domains may lead to inconsistent GLBE bandwidth control. The limit specified in CEILING is an absolute value specified in the units reported by CPUID Fn8000_0020_EBX_x7[BW_MULT] (bits

[AMD Public Use]

15:0). The value programmed in this register may exceed the total bandwidth which the system can provide to all logical processors.

At reset, the U bit of all L3QOS_GL_BW_CONTROL MSRs is set to 1, which excludes logical processors belonging to this COS in the QOS Domain from GLBE. System software may clear the U bit and thus enable GLBE for only a subset of QOS Domains within the GLBE Control Domain.

For example, consider a system containing 4 QOS Domains in a single GLBE Control Domain. All the logical processors in all the QOS Domains are associated with COS 0. L3QOS_GL_BW_CONTROL_0[CEILING] is set to 100 GB/s on all QOS Domains. L3QOS_GL_BW_CONTROL_0[U] is set to 0 on QOS Domains 0-2, and to 1 on QOS Domain 3. COS 0 in QOS Domains 0-2 together will be limited to a total of 100 GB/s of L3 External Bandwidth. COS 0 in QOS Domain 3 will not be limited by GLBE and the L3 External Bandwidth it consumes will not be counted towards the 100 GB/s GLBE ceiling.

The GLBE Ceiling does not ensure that the specified bandwidth will be available to the logical processors associated with a given COS. Cases where the logical processors may not be able to reach their allocated bandwidth include (but are not limited to):

- Maximum system bandwidth may be less than the specified limit.
- The sum of the ceilings for all COS within a GLBE Control Domain may exceed the total bandwidth the system can supply to that GLBE Control Domain.
- I/O or other system entities may consume system bandwidth and result in less bandwidth being available to the GLBE Control Domain.
- Large amounts of write traffic may affect the memory system’s ability to deliver read bandwidth.
- Other PQOS Bandwidth Enforcement Features (L3BE, L3SMBE, GLSBE) may limit bandwidth available to the GLBE Control Domain.

GLBE measures L3 External Bandwidth, not DRAM bandwidth. Reasons that bandwidth measured by GLBE may differ from System Memory Bandwidth include (but are not limited to):

- Bandwidth measured by GLBE may originate from a different QOS Domain and not from system memory.
- Bandwidth measured by GLBE may include cache coherency traffic.
- I/O or other system entities outside of a GLBE Control Domain may access system memory without being measured or controlled by a GLBE Control Domain.

19.4.8.3 CDP Interaction with GLBE

Similar to L3 Bandwidth Enforcement (L3BE), the mapping of a COS to its associated L3QOS_GL_BW_CONTROL register is changed if Code and Data Prioritization (CDP) is enabled. GLBE does not support different bandwidth limits for data vs. code access. When CDP is enabled, for a given COS, the bandwidth limits for data and code access are specified by MSR C000_0600h + (2*COS) and bandwidth limits specified by MSR C000_0601h + (2*COS) have no effect. For example, if CDP is enabled, the bandwidth limit for data access by logical processors associated with COS 3 is specified in L3QOS_GL_BW_CONTROL_6. Note that enabling CDP reduces the effective

[AMD Public Use]

number of unique COS values by half. Specifying a COS outside of the valid range will result in undefined behavior.

19.4.9 Global Slow Bandwidth Enforcement (GLSBE)

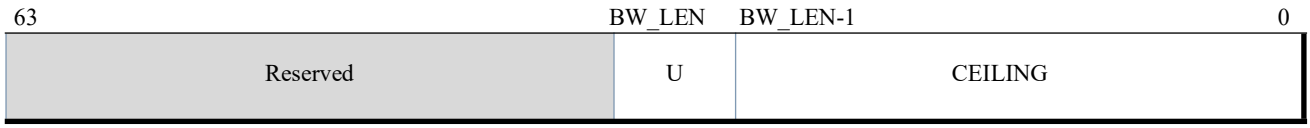
GLSBE is a feature similar to GLBE (Global Bandwidth Enforcement) which provides a mechanism for software to specify bandwidth limits for groups of logical processors that span multiple QOS Domains which access memory designated as *Slow Memory* by the system. The collection of QOS Domains for GLSBE is identical to that for GLBE and is referred to as the GLBE Control Domain. The GLSBE ceiling is a bandwidth ceiling for L3 External Slow Memory Bandwidth which is competitively shared between all logical processors in a given COS across all QOS Domains within the GLBE Control Domain. This complements L3 Slow Memory Bandwidth Enforcement (L3SMBE) which provides L3 External Slow Memory Bandwidth control on a per QOS Domain granularity.

Support for GLSBE is indicated by CPUID Fn8000_0020_EBX_x0[GLSBE] (bit 8) = 1. If GLSBE is supported, the feature's attributes and capabilities are enumerated by CPUID Fn8000_0020_x8 as summarized below:

- EAX[31:0](BW_LEN) returns the size of the bandwidth specifier field in the L3QOS_GL_SLOWBW_CONTROL_n MSRs.
- EBX[15:0](BW_MULT) returns the units of the GLSBE Ceiling field in the L3QOS_GL_SLOWBW_CONTROL_n MSRs; units = 1/8 GB/s * (BW_MULT+1).
- EDX[31:0](COS_MAX) returns the maximum COS number supported by the GLSBE feature.

19.4.9.1 GLSBE MSRs

Bandwidth limits are specified per COS using the L3QOS_GL_SLOWBW_CONTROL_n MSRs where n is the COS. This set of registers starts at MSR address C000_0680h and continues through C000_0680h + (COS_MAX). The format of the L3QOS_GL_SLOWBW_CONTROL_n MSRs is shown below. There is one L3QOS_GL_SLOWBW_CONTROL_n register for each COS in each QOS Domain. The maximum COS number supported by GLSBE is identified by CPUID Fn8000_0020_EDX_x8[COS_MAX] (bits 31:0).



Bits	Mnemonic	Description	Access Type
63:BW_LEN+1	–	Reserved	MBZ
BW_LEN	U	Unlimited Bandwidth	R/W
BW_LEN-1:0	CEILING	Bandwidth Ceiling	R/W

Figure 19-15. L3QOS_GL_SLOWBW_CONTROL_n

[AMD Public Use]

The fields within the L3QOS_GL_SLOWBW_CONTROL_n register are:

- U (Unlimited Bandwidth) – bit BW_LEN. When cleared, the L3 External Slow Memory Bandwidth of logical processors belonging to this COS counts towards the bandwidth limit configured with CEILING and is limited by GLSBE. When set, the L3 External Slow Memory Bandwidth of logical processors belonging to this COS does not count towards the bandwidth limit and is not limited by GLSBE. At reset, the U bit for all L3QOS_GL_SLOWBW_CONTROL_n MSRs is set. This field may be set to different values in each QoS Domain within a single GLBE Control Domain.
- CEILING (Bandwidth Ceiling) – bits BW_LEN-1:0. Specifies a limit on the combined L3 External Slow Memory Bandwidth consumed by logical processors associated with this COS across all QoS Domains in the GLBE Control Domain. The limit is expressed in units reported by CPUID Fn8000_0020_EBX_x8[BW_MULT] (bits 15:0). This value must be configured to the same value for a given COS in all QoS Domains in the GLBE Control Domain for consistent operation.

19.4.9.2 Using GLSBE

Usage of GLSBE is identical to GLBE usage except that it is enumerated by CPUID Fn8000_0020_x8 and controlled by L3QOS_GL_SLOWBW_CONTROL_n MSRs. Refer to “Using GLBE” on page 752 for more details.

The GLSBE Ceiling is an upper bound on the combined L3 External Bandwidth to Slow Memory of all the logical processors in a COS within the QOS Domains included in the GLBE Control Domain. The GLSBE Ceiling does not ensure that the specified bandwidth will be available to the logical processors running with a given COS. For a list of cases where logical processors in a given COS may not be able to reach their allocated bandwidth and how bandwidth measured by GLSBE can be different from system Slow Memory bandwidth, refer to the lists of cases shown in the “Using GLBE” section.

19.4.9.3 CDP Interaction with GLSBE

Similar to L3 Bandwidth Enforcement (L3BE), the mapping of a COS to its associated L3QOS_GL_SLOWBW_CONTROL register is changed if Code and Data Prioritization (CDP) is enabled. GLSBE does not support different bandwidth limits for data vs. code access. When CDP is enabled, for a given COS, the bandwidth limits for data and code access are specified by MSR C000_0680h + (2*COS) and bandwidth limits specified by MSR C000_0681h + (2*COS) have no effect. For example, if CDP is enabled, the bandwidth limit for data access by logical processors associated with COS 3 is specified in L3QOS_GL_SLOWBW_CONTROL_6. Note that enabling CDP reduces the effective number of unique COS values by half. Specifying a COS outside of the valid range will result in undefined behavior.

[AMD Public Use]

Appendix A MSR Cross-Reference

This appendix lists the MSRs that are defined in the AMD64 architecture. The AMD64 architecture supports some of the same MSRs as previous versions of the x86 architecture and implementations thereof. Where possible, the AMD64 architecture supports the same MSRs, for the same functions, as these previous architectures and implementations.

The first section lists the MSRs according to their MSR address, and it gives a cross reference for additional information. The remaining sections list the MSRs by their functional group. Those sections also give a brief description of the register and specify the register reset value.

Some MSRs are implementation-specific For information about these MSRs, see the documentation for specific implementations of the AMD64 architecture.

A.1 MSR Cross-Reference by MSR Address

Table A-1 lists the MSRs in the AMD64 architecture in order of MSR address.

Table A-1. MSRs of the AMD64 Architecture

MSR Address	MSR Name	Functional Group	Cross-Reference
0010h	TSC	Performance	“Time-Stamp Counter” on page 425
001Bh	APIC_BASE	System Software	“Local APIC Enable” on page 661
0048h	SPEC_CTRL	Speculation Control	“Speculation Control Registers” on page 67
0049h	PRED_CMD	Speculation Control	“Speculation Control Registers” on page 67
00E7h	MPERF	Performance	“Determining Processor Effective Frequency” on page 699
00E8h	APERF	Performance	“Determining Processor Effective Frequency” on page 699
00FEh	MTRRcap	Memory Typing	“Identifying MTRR Features” on page 226
010Bh	FLUSH_CMD	Memory Caches	“FLUSH_CMD MSR” on page 184
0174h	SYSENTER_CS	System Software	“SYSENTER and SYSEXIT MSRs” on page 178
0175h	SYSENTER_ESP		
0176h	SYSENTER_EIP		

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
0179h	MCG_CAP	Machine Check	"Machine-Check Global-Capabilities Register" on page 303
017Ah	MCG_STATUS		"Machine-Check Global-Status Register" on page 304
017Bh	MCG_CTL		"Machine-Check Global-Control Register" on page 305
01CCh	FRED_RSP0	System Software	"FRED MSRs"
01CDh	FRED_RSP1		
01CEh	FRED_RSP2		
01CFh	FRED_RSP3		
01D0h	FRED_STKLVLS		
01D1h	FRED_SSP1		
01D2h	FRED_SSP2		
01D3h	FRED_SSP3		
01D4h	FRED_CONFIG		
01D9h	DebugCtl	Software Debug	"Debug-Control MSR (DebugCtl)" on page 399
01DBh	LastBranchFromIP	Software Debug	"Control-Transfer Recording MSRs" on page 401
01DCh	LastBranchToIP		
01DDh	LastIntFromIP		
01DEh	LastIntToIP		
0200h	MTRRphysBase0	Memory Typing	"Variable-Range MTRRs" on page 223
0201h	MTRRphysMask0		
0202h	MTRRphysBase1		
0203h	MTRRphysMask1		
0204h	MTRRphysBase2		
0205h	MTRRphysMask2		
0206h	MTRRphysBase3		
0207h	MTRRphysMask3		
0208h	MTRRphysBase4		
0209h	MTRRphysMask4		
020Ah	MTRRphysBase5		
020Bh	MTRRphysMask5		
020Ch	MTRRphysBase6		
020Dh	MTRRphysMask6		
020Eh	MTRRphysBase7		
020Fh	MTRRphysMask7		

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
0250h	MTRRfix64K_00000	Memory Typing	"Fixed-Range MTRRs" on page 221
0258h	MTRRfix16K_80000		
0259h	MTRRfix16K_A0000		
0268h	MTRRfix4K_C0000		
0269h	MTRRfix4K_C8000		
026Ah	MTRRfix4K_D0000		
026Bh	MTRRfix4K_D8000		
026Ch	MTRRfix4K_E0000		
026Dh	MTRRfix4K_E8000		
026Eh	MTRRfix4K_F0000		
026Fh	MTRRfix4K_F8000		
0277h	PAT	Memory Typing	"PAT Register" on page 229
02FFh	MTRRdefType		"Default-Range MTRRs" on page 225
0400h	MC0_CTL	Machine Check	See the documentation for particular implementations of the architecture.
0404h	MC1_CTL		
0408h	MC2_CTL		
040Ch	MC3_CTL		
0410h	MC4_CTL		
0414h	MC5_CTL		
0401h	MC0_STATUS	Machine Check	"Machine-Check Status Registers" on page 310
0405h	MC1_STATUS		
0409h	MC2_STATUS		
040Dh	MC3_STATUS		
0411h	MC4_STATUS		
0415h	MC5_STATUS		
0402h	MC0_ADDR	Machine Check	"Machine-Check Address Registers" on page 313
0406h	MC1_ADDR		
040Ah	MC2_ADDR		
040Eh	MC3_ADDR		
0412h	MC4_ADDR		
0416h	MC5_ADDR		

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
0403h	MC0_MISC	Machine Check	"Machine-Check Miscellaneous-Error Information Register 0 (MCi_MISC0)" on page 313
0407h	MC1_MISC		
040Bh	MC2_MISC		
040Fh	MC3_MISC		
0413h	MC4_MISC		
0417h	MC5_MISC		
06A0h	U_CET	Shadow Stack	"Shadow Stack MSRs" on page 777
06A2h	S_CET		
06A4h	PL0_SSP		
06A5h	PL1_SSP		
06A6h	PL2_SSP		
06A7h	PL3_SSP		
06A8h	ISST_ADDR		
0C81h	L3_QOS_CFG1	Quality of Service	"Platform Quality of Service (PQOS) Extension" on page 724
0C8Dh	QM_EVTSEL	Quality of Service	
0C8Eh	QM_CTR	Quality of Service	
0C8Fh	PQR_ASSOC	Quality of Service	
0C90h	L3_MASK_0	Quality of Service	
0C90+n	L3_MASK+n	Quality of Service	
0DA0h	XSS	System Software	XSAVES and XSTRORS instructions in APM Volume 4
C000_0080h	EFER	System Software	"Extended Feature Enable Register (EFER)" on page 56
C000_0081h	STAR	System Software	"SYSCALL and SYSRET MSRs" on page 177
C000_0082h	LSTAR	System Software	
C000_0083h	CSTAR	System Software	
C000_0084h	SF_MASK	System Software	
C000_00E7h	MPerfReadOnly	Performance	"MPERF Read-only (MperfReadOnly)" on page 701
C000_00E8h	APerfReadOnly	Performance	"APERF Read-only (AperfReadOnly)" on page 701
C000_00E9h	IRPerfCount	Performance	"Instructions Retired Performance counter" on page 423
C000_0100h	FS.Base	System Software	"FS and GS Registers in 64-Bit Mode" on page 82
C000_0101h	GS.Base	System Software	
C000_0102h	KernelGSbase	System Software	"SWAPGS Instruction" on page 178
C000_0103h	TSC_AUX	System Software	"RDTSCP Instruction" on page 181

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C000_0104h	TSC Ratio	SVM	"TSC Ratio MSR (C000_0104h)" on page 610
C000_0108h	PrefetchControl	—	Controls enabling / disabling hardware prefetchers. See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.
C000_010Eh	LastBranchStackSelect	Software Debug	"Last Branch Record Stack" on page 412
C000_010Fh	DebugExtnCtl	Software Debug	"Debug-Extension-Control MSR (DebugExtnCtl)" on page 401
C000_0200h	L3QOS_BW_CONTROL_0	Quality of Service	"Platform Quality of Service (PQOS) Extension" on page 724
C000_0200h+n	L3QOS_BW_CONTROL_n	Quality of Service	
C000_0280h	L3QOS_SMBW_CONTROL_0	Quality of Service	
C000_0280h+n	L3QOS_SMBW_CONTROL_n	Quality of Service	
C000_0300h	PerfCntrGlobalStatus	Performance	"Core Performance Counter Status Registers" on page 419
C000_0301h	PerfCntrGobalCtl	Performance	"Core Performance Global Control Register" on page 418
C000_0302h	PerfCntrGlobalStatusClr	Performance	"Performance Counter Global Status Clear Register" on page 421
C000_0303h	PerfCntrGlobalStatusSet	Performance	"Performance Counter Global Status Set Register" on page 420
C000_03FCh	PQR_PLZA_ASSOC	Quality of Service	"Platform Quality of Service (PQOS) Extension" on page 724
C000_03FDh	L3_QOS_ABMC_CFG	Quality of Service	
C000_03FEh	L3_QOS_ABMC_DSC	Quality of Service	
C000_03FFh	L3_QOS_EXT_CFG	Quality of Service	
C000_0400h	QOS_EVT_CFG_0	Quality of Service	
C000_0401h	QOS_EVT_CFG_1	Quality of Service	
C000_0408h	MC4_MISC1	Machine Check	"Machine-Check Miscellaneous-Error Information Register 0 (MCi_MISC0)" on page 313
C000_0409h	MC4_MISC2		
C000_040Ah	MC4_MISC3		
C000_0410h	MCA_INTR_CFG		MCA related interrupt configuration. See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.
C000_0420h	MCA_QUERY0		"MCA Query Registers" on page 308
C000_0421h	MCA_QUERY1		
C000_0422h	MCA_QUERY2		
C000_0423h	MCA_QUERY3		

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C000_0600h	L3QOS_GL_BW_CONTROL_0	Quality of Service	"Platform Quality of Service (PQOS) Extension" on page 724
C000_0600h+n	L3QOS_GL_BW_CONTROL_n		
C000_0680h	L3QOS_GL_SLOWBW_CONTROL_0		
C000_0680h+n	L3QOS_GL_SLOWBW_CONTROL_n		
C000_2000h + i*10h	MCA_CTL	Machine Check	See the documentation for particular implementations of the architecture.
C000_2001h + i*10h	MCA_STATUS		"Machine-Check Status Registers" on page 310
C000_2002h + i*10h	MCA_ADDR		"Machine-Check Address Registers" on page 313
C000_2003h + i*10h	MCA_MISC0		"Machine-Check Miscellaneous-Error Information Register 0 (MCi_MISC0)" on page 313
C000_2004h + i*10h	MCA_CONFIG		"MCA Configuration Register" on page 317
C000_2005h + i*10h	MCA_IPID		"MCA IP Identification" on page 319
C000_2006h + i*10h	MCA_SYND		"MCA Syndrome Register" on page 320
C000_2008h + i*10h	MCA_DESTAT		"MCA Deferred Error Status Register" on page 320
C000_2009h + i*10h	MCA_DEADDR		"MCA Deferred Error Address Register" on page 321
(C000_200Ah : C000_200Dh) + i*10h	MCA_MISC[4:1]		"MCA Miscellaneous Registers 1 - 4" on page 321
(C000_200Eh : C000_200Fh)+ i*10h	MCA_SYND[2:1]		"MCA Syndrome Registers 1 - 2" on page 321
C001_0000h	PerfEvtSel0	Performance	"Core Performance Event-Select Registers" on page 415
C001_0001h	PerfEvtSel1		
C001_0002h	PerfEvtSel2		
C001_0003h	PerfEvtSel3		
C001_0004h	PerfCtr0	Performance	"Performance Counter MSRs" on page 413
C001_0005h	PerfCtr1		
C001_0006h	PerfCtr2		
C001_0007h	PerfCtr3		

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C001_0010h	SYSCFG	Memory Typing	"System Configuration Register (SYSCFG)" on page 61
C001_0015h	HWCR	System Software	"Hardware Configuration Register (HWCR)" on page 72
C001_0016h	IORRBase0	Memory Typing	"IORRs" on page 236
C001_0017h	IORRMask0		
C001_0018h	IORRBase1		
C001_0019h	IORRMask1		
C001_001Ah	TOP_MEM	Memory Typing	"Top of Memory" on page 238
C001_001Dh	TOP_MEM2		
C001_0030h	Processor_Name_String	CPUID Name	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.
C001_0031h			
C001_0032h			
C001_0033h			
C001_0034h			
C001_0035h			
C001_0056h	SMI_Trigger_IO_Cycle	SMM	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.
C001_0061h	P-State Current Limit	SMM	"Hardware Performance Monitoring and Control" on page 696
C001_0062h	P-State Control		
C001_0063h	P-State Status		
C001_0074h	CPU_Watchdog_Timer	Machine Check	"CPU Watchdog Timer Register" on page 306
C001_0111h	SMBASE	SMM	"SMBASE Register" on page 328
C001_0112h	SMM_ADDR		"SMRAM Protected Areas" on page 334
C001_0113h	SMM_MASK		
C001_0114h	VM_CR	SVM	"SVM Related MSRs" on page 608
C001_0115h	IGNNE	SVM	"SVM Related MSRs" on page 608
C001_0116h	SMM_CTL	SVM	"SVM Related MSRs" on page 608
C001_0117h	VM_HSAVE_PA	SVM	"SVM Related MSRs" on page 608
C001_0118h	SVM_KEY	SVM	"SVM-Lock" on page 611
C001_0119h	SMM_KEY	SMM	"SMM-Lock" on page 612
C001_011Ah	Local_SMI_Status	SMM	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.
C001_011Bh	Doorbell Register	SVM	"Doorbell Register" on page 604
C001_011Eh	VMPAGE_FLUSH	SVM	"Secure Encrypted Virtualization" on page 613

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C001_011Fh	VIRT_SPEC_CTRL	Speculation Control	“Speculation Control Registers” on page 67
C001_0130h	GHCB	SVM	“GHCB” on page 625
C001_0131h	SEV_STATUS	SVM	“SEV_STATUS MSR” on page 618
C001_0132h	RMP_BASE	SVM	“Initializing the RMP” on page 630
C001_0133h	RMP_END	SVM	“Initializing the RMP” on page 630
C001_0134h	GUEST_TSC_FREQ	SVM	“Secure TSC” on page 645
C001_0135h	VIRTUAL_TOM	SVM	“Virtual Top-of-Memory” on page 633
C001_0136h	SEGMENTED_RMP_CFG	SVM	“Segmented RMP” on page 649
C001_0137h	IDLE_WAKEUP_ICR	SVM	“Side-Channel Protection” on page 641
C001_0138h	SECURE_AVIC_CTRL	SVM	“Secure AVIC Control MSR” on page 648
C001_0139h	RMPOPT_BASE	SVM	“RMPOPT” on page 652
C001_013Ah	VCPU_ID	SVM	“Enhanced SMT Protection” on page 643
C001_0140h	OSVW_ID_Length	System Software	“OS-Visible Workarounds” on page 810
C001_0141h	OSVW Status		

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C001_0200h	PerfEvtSel0	Performance	"Core Performance Event-Select Registers" on page 415
C001_0202h	PerfEvtSel1		
C001_0204h	PerfEvtSel2		
C001_0206h	PerfEvtSel3		
C001_0208h	PerfEvtSel4		
C001_020Ah	PerfEvtSel5		
C001_0201h	PerfCtr0		"Performance Counter MSRs" on page 413
C001_0203h	PerfCtr1		
C001_0205h	PerfCtr2		
C001_0207h	PerfCtr3		
C001_0209h	PerfCtr4		
C001_020Bh	PerfCtr5		
C001_0230h	L2I_PerfEvtSel0		"Performance-Monitoring MSRs" on page 772
C001_0232h	L2I_PerfEvtSel1		
C001_0234h	L2I_PerfEvtSel2		
C001_0236h	L2I_PerfEvtSel3		
C001_0231h	L2I_PerfCtr0		
C001_0233h	L2I_PerfCtr1		
C001_0235h	L2I_PerfCtr2		
C001_0237h	L2I_PerfCtr3		
C001_0240h + 2i	NB_PerfEvtSeli (i=0..15)		
C001_0241h + 2i	NB_PerfCtri (i=0..15)		
C001_02B0h	CPPC_CAPABILITY_1	CPPC	"Collaborative Processor Performance Control" on page 703
C001_02B1h	CPPC_ENABLE	CPPC	
C001_02B2h	CPPC_CAPABILITY_2	CPPC	
C001_02B3h	CPPC_REQUEST	CPPC	
C001_02B4h	CPPC_STATUS	CPPC	
C001_02B5h	CPPC_REQUEST_2	CPPC	
C001_0300 + i*2h	LastBranchStackFromIp	Software Debug	"Control-Transfer Recording MSRs" on page 401
C001_0301 + i*2h	LastBranchStackToIp		

[AMD Public Use]

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C001_0380h	IbsMemCtl	Instruction Based Sampling	"IBS Memory Profiler" on page 453
C001_0381h	IbsMemCtl2		
C001_0382h	IbsMemData2		
C001_0383h	IbsMemData3		
C001_0384h	IbsMemLinAd		
C001_0385h	IbsMemPhysAd		
C001_0386h	IbsMemRip		
C001_0390h	IbsBufferBase	Instruction Based Sampling	"IBS Buffering" on page 448
C001_0391h	IbsBufferSize		
C001_0392h	IbsBufferTail		
C001_1019h	DR1_ADDR_MASK	Software Debug	"Debug Breakpoint Address Masking" on page 412
C001_101Ah	DR2_ADDR_MASK		
C001_101Bh	DR3_ADDR_MASK		
C001_1027h	DR0_ADDR_MASK		
C001_1030h	IbsFetchCtl	Instruction Based Sampling	"IBS Fetch Sampling Registers" on page 427
C001_1031h	IbsFetchLinAd		
C001_1032h	IbsFetchPhysAd		
C001_1033h	IbsOpCtl		"IBS Execution Sampling Registers" on page 435
C001_1034h	IbsOpRip		
C001_1035h	IbsOpData1		
C001_1036h	IbsOpData2		
C001_1037h	IbsOpData3		
C001_1038h	IbsDcLinAd		
C001_1039h	IbsDcPhysAd		
C001_103Ah	IbsCtl		
C001_103Bh	IbsBrTarget		
C001_103Ch	IbsFetchExtdCtl		"IBS Fetch Sampling Registers" on page 427
C001_103Eh	IbsOpCtl2		"IBS Execution Sampling Registers" on page 435
C001_103Fh	IbsFetchCtl2		"IBS Fetch Sampling Registers" on page 427
C001_1095h	L3RangeReserveBaseAddr	Memory Caches	"L3 Cache Range Reservation" on page 215
C001_1096h	L3RangeReserveMaxAddr		
C001_109Ah	L3RangeReserveWayMask		

[AMD Public Use]

A.2 System-Software MSRs

Table A-2 lists the MSRs defined for general use by system software in controlling long mode and in allowing fast control transfers between applications and the operating system.

Table A-2. System-Software MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0000_001Bh	APIC_BASE	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.	0000_0000_FEE0_0x00h
0DA0h	XSS	Contains a bitmap of supervisor-level state components managed by the XSAVES/XRSTORS instructions.	0000_0000_0000_0000h
C000_0080h	EFER	Contains control bits that enable extended features supported by the processor, including long mode.	0000_0000_0000_0000h
C000_0081h	STAR	In legacy mode, used to specify the target address of a SYSCALL instruction, as well as the CS and SS selectors of the called and returned procedures.	undefined
C000_0082h	LSTAR	In 64-bit mode, used to specify the target RIP of a SYSCALL instruction.	undefined
C000_0083h	CSTAR	In compatibility mode, used to specify the target RIP of a SYSCALL instruction.	undefined
C000_0084h	SF_MASK	SYSCALL Flags Mask	undefined
C000_0100h	FS.Base	Contains the 64-bit base address in the hidden portion of the FS register (the base address from the FS descriptor).	0000_0000_0000_0000h
C000_0101h	GS.Base	Contains the 64-bit base address in the hidden portion of the GS register (the base address from the GS descriptor).	0000_0000_0000_0000h
C000_0102h	KernelGSbase	The SWAPGS instruction exchanges the value in KernelGSbase with the value in GS.base, providing a fast method for system software to load a pointer to system data-structures.	undefined
C000_0103h	TSC_AUX	The RDTSCP instruction copies the value of this MSR into the ECX register.	0000_0000_0000_0000h
C001_0015h	HWCR	Contains control bits that affect the functionality of other features.	See the appropriate *BIOS and Kernel Developer's Guide or Processor Programming Reference Manual.*
C001_0140h	OSVW_ID_Length	Indicates the total number of valid OS-Visible Workaround (OSVW) status bits.	0000_0000_0000_0000h
C001_0141h	OSVW_Status	Contains OS-Visible Workaround (OSVW) erratum status bits.	0000_0000_0000_0000h

[AMD Public Use]

Table A-2. System-Software MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
0174h	SYSENTER_CS	In legacy mode, used to specify the CS selector of the procedure called by SYSENTER.	undefined
0175h	SYSENTER_ESP	In legacy mode, used to specify the stack pointer for the procedure called by SYSENTER.	undefined
0176h	SYSENTER_EIP	In legacy mode, used to specify the EIP of the procedure called by SYSENTER.	undefined
01CCh	FRED_RSP0	FRED RSP0	0000_0000_0000_0000h
01CDh	FRED_RSP1	FRED RSP1	0000_0000_0000_0000h
01CEh	FRED_RSP2	FRED RSP2	0000_0000_0000_0000h
01CFh	FRED_RSP3	FRED RSP3	0000_0000_0000_0000h
01D0h	FRED_STKLVLS	FRED Stack Levels	0000_0000_0000_0000h
01D1h	FRED_SSP1	FRED SSP1	0000_0000_0000_0000h
01D2h	FRED_SSP2	FRED SSP2	0000_0000_0000_0000h
01D3h	FRED_SSP3	FRED SSP3	0000_0000_0000_0000h
01D4h	FRED_CONFIG	FRED Configuration	0000_0000_0000_0000h

A.3 Memory-Typing MSRs

Table A-3 lists the MSRs used to control memory-typing and the page-attribute-table mechanism.

Table A-3. Memory-Typing MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
00FEh	MTRRcap	A *read-only* register containing information describing the level of MTRR support provided by the processor.	0000_0000_0000_0508h
0200h	MTRRphysBase0	Specifies the memory-range base address in physical-address space of a variable-range memory region. These registers also specify the memory type used for the memory region.	undefined
0202h	MTRRphysBase1		
0204h	MTRRphysBase2		
0206h	MTRRphysBase3		
0208h	MTRRphysBase4		
020Ah	MTRRphysBase5		
020Ch	MTRRphysBase6		
020Eh	MTRRphysBase7		

[AMD Public Use]

Table A-3. Memory-Typing MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
0201h	MTRRphysMask0	Specifies the size of a variable-range memory region.	Valid (bit 11) = 0 All Other Bits Undefined
0203h	MTRRphysMask1		
0205h	MTRRphysMask2		
0207h	MTRRphysMask3		
0209h	MTRRphysMask4		
020Bh	MTRRphysMask5		
020Dh	MTRRphysMask6		
020Fh	MTRRphysMask7		
0250h	MTRRfix64K_00000	Fixed-range MTRRs used to characterize the first 1 Mbyte of physical memory. Each 64-bit register contains eight type fields for characterizing a total of eight memory ranges. • MTRRfix64K_n characterizes 64 Kbyte ranges. • MTRRfix16K_n characterizes 16 Kbyte ranges. • MTRRfix4K_n characterizes 4 Kbyte ranges.	undefined
0258h	MTRRfix16K_80000		
0259h	MTRRfix16K_A0000		
0268h	MTRRfix4K_C0000		
0269h	MTRRfix4K_C8000		
026Ah	MTRRfix4K_D0000		
026Bh	MTRRfix4K_D8000		
026Ch	MTRRfix4K_E0000		
026Dh	MTRRfix4K_E8000		
026Eh	MTRRfix4K_F0000		
026Fh	MTRRfix4K_F8000		
0277h	PAT	Used to extend the page-table entry format, allowing memory-type characterization on a physical-page basis.	0007_0406_0007_0406h
02FFh	MTRRdefType	Sets the default memory-type for physical addresses not within ranges established by fixed-range and variable-range MTRRs.	0000_0000_0000_0000h
C001_0010h	SYSCFG	Contains control bits for enabling and configuring system bus features.	0000_0000_0002_0601h
C001_0016h	IORRBase0	Specifies the memory-range base address in physical-address space of a variable-range I/O region.	undefined
C001_0018h	IORRBase1		
C001_0017h	IORRMask0	Specifies the size of a variable-range I/O region.	Valid (bit 11) = 0 All Other Bits Undefined
C001_0019h	IORRMask1		
C001_001Ah	TOP_MEM	Sets the boundary between system memory and memory-mapped I/O for addresses below 4 Gbytes.	0000_0000_0400_0000h
C001_001Dh	TOP_MEM2	Sets the boundary between system memory and memory-mapped I/O for addresses above 4 Gbytes.	undefined

[AMD Public Use]

A.4 Machine-Check MSRs

Table A-4 lists the MSRs used in support of the machine-check mechanism.

Table A-4. Machine-Check MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0179h	MCG_CAP	A read-only register that specifies the machine-check mechanism capabilities supported by the processor.	0000_0000_0000_010xh
017Ah	MCG_STATUS	Provides basic information about the processor state immediately after the occurrence of a machine-check error.	undefined
017Bh	MCG_CTL	Controls global reporting of machine-check errors from various sources.	0000_0000_0000_0000h
0400h + 4i for 0 <=i <=31	MCi_CTL	Control for error reporting banks, per implementation.	0000_0000_0000_0000h
0401h + 4i	MCi_STATUS[5:0]	Status registers for each error-reporting register bank, used to report machine-check error information for the specified register bank.	undefined
0402h + 4i	MCi_ADDR[5:0]	Reports the instruction memory-address or data memory-address responsible for the machine-check error for the specified register bank.	undefined
0403h + 4i	MCi_MISC[5:0]	Reports miscellaneous information about the machine-check error for the specified register bank.	c00x_xxxx_xx00_0000
C000_0408h	MC4_MISC1		c00x_xxxx_0000_0000
C000_0409h	MC4_MISC2		
C000_040Ah	MC4_MISC3		
C000_0410h	MCA_INTR_CFG	MCA interrupt configuration.	0000_0000_0000_0000h
C000_0420h	MCA_QUERY0	Reports MCi_STATUS[VAL] for MCA banks 0 through 63.	0000_0000_0000_0000h
C000_0421h	MCA_QUERY1	reserved for future use	
C000_0422h	MCA_QUERY2		
C000_0423h	MCA_QUERY3		
C000_2000h + i*10h	MCA_CTL	Control for error reporting banks per implementation.	0000_0000_0000_0000h
C000_2001h + i*10h	MCA_STATUS	Status registers for each error-reporting register bank, used to report machine-check error information for the specified register bank.	undefined

[AMD Public Use]

Table A-4. Machine-Check MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
C000_2002h + i*10h	MCA_ADDR	Reports the instruction memory-address or data memory-address responsible for the machine-check error for the specified register bank.	undefined
C000_2003h + i*10h	MCA_MISC0	Reports miscellaneous information about the machine-check error for the specified register bank.	c00x_xxxx_xx00_0000
C000_2004h + i*10h	MCA_CONFIG	Controls configuration information of each MCA bank.	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual*.
C000_2005h + i*10h	MCA_IPID	Holds information to identify the MCA bank type.	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual*.
C000_2006h + i*10h	MCA_SYND	Holds syndrome associated with the error.	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual*.
C000_2008h + i*10h	MCA_DESTAT	Holds status information for deferred errors.	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual*.
C000_2009h + i*10h	MCA_DEADDR	Provides the address associated with the deferred error.	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual*.
(C000_200Ah: C000_200Dh) + i*10h	MCA_MISC[4:1]	Reports miscellaneous-error information.	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual*.
(C000_200Eh: C000_200Fh) + i*10h	MCA_SYND[2:1]	Stores information associated with the error in MCA_STATUS or MCA_DESTAT.	See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual*.
***i* - indicates the register bank number**			

A.5 Software-Debug MSRs

Table A-5 lists the MSRs used in support of the software-debug architecture.

[AMD Public Use]

Table A-5. Software-Debug MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
01D9h	DebugCtl	Provides debug controls for control-transfer recording and control-transfer single stepping, and external-breakpoint reporting and trace messages.	0000_0000_0000_0000h
01DBh	LastBranchFromIP	During control-transfer recording, this register is loaded with the segment offset of the control-transfer source.	undefined
01DCh	LastBranchToIP	During control-transfer recording, this register is loaded with the segment offset of the control-transfer target.	undefined
01DDh	LastIntFromIP	When an interrupt occurs during control-transfer recording, this register is loaded with LastBranchFromIP before LastBranchFromIP is updated.	undefined
01DEh	LastIntToIP	When an interrupt occurs during control-transfer recording, this register is loaded with LastBranchToIP before LastBranchToIP is updated.	undefined
C000_010Eh	LastBranchStackSelect	Controls the types of branches suppressed from Last Branch Record Stack recording.	0000_0000_0000_0000h
C000_010Fh	DebugExtnCtl	Provides control over additional debug features.	0000_0000_0000_0000h
C000_1027h	DR0_ADDR_MASK	Address mask for DR0 breakpoint [31:0]	0000_0000_0000_0000h
C000_1019h	DR1_ADDR_MASK	Address mask for DR1 breakpoint [31:0]	0000_0000_0000_0000h
C000_101Ah	DR2_ADDR_MASK	Address mask for DR2 breakpoint [31:0]	0000_0000_0000_0000h
C000_101Bh	DR3_ADDR_MASK	Address mask for DR3 breakpoint [31:0]	0000_0000_0000_0000h
C001_0300h + i*2h	LastBranchStackFromIp	Records the source address of a control-transfer instruction in the Last Branch Record Stack.	0000_0000_0000_0000h
C001_0301h + i*2h	LastBranchStackToIp	Records the target address of a control-transfer instruction in the Last Branch Record Stack.	0000_0000_0000_0000h

A.6 Performance-Monitoring MSRs

Table A-6 lists the MSRs used in support of performance monitoring, including the time-stamp counter.

[AMD Public Use]

Table A-6. Performance-Monitoring MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0010h	TSC	Counts processor-clock cycles. It is incremented once for each processor-clock cycle.	0000_0000_0000_0000h
00E7h	MPERF	Denominator of effective frequency ratio.	0000_0000_0000_0000h
00E8h	APERF	Numerator of effective frequency ratio.	0000_0000_0000_0000h
C000_00E7h	MPerfReadOnly	Read only version of MPERF.	0000_0000_0000_0000h
C000_00E8h	APerfReadOnly	Read only version of APERF.	
C000_00E9h	IRPerfCount	Dedicated instructions retired performance counter.	
C000_0300h	PerfCntrGlobalStatus	A read-only register that contains status bits for the Core Performance Counters.	0000_0000_0000_0000h
C000_0301h	PerfCntrGlobalCtl	Provides additional enable bits for core performance counters.	0000_0000_0000_003Fh
C000_0302h	PerfCntrGlobalStatusClr	A write-only register for clearing bits in PerfCntrGlobalStatus.	undefined
C000_0303h	PerfCntrGlobalStatusSet	A write-only register for setting bits in PerfCntrGlobalStatus.	undefined
C001_0000h	PerfEvtSel0	For the corresponding performance counter, this register specifies the events counted, and controls other aspects of counter operation.	0000_0000_0000_0000h
C001_0001h	PerfEvtSel1		
C001_0002h	PerfEvtSel2		
C001_0003h	PerfEvtSel3		
C001_0004h	PerfCtr0	Used to count specific processor events, or the duration of events, as specified by the corresponding PerfEvtSel*n* register.	undefined
C001_0005h	PerfCtr1		
C001_0006h	PerfCtr2		
C001_0007h	PerfCtr3		
C001_0200h	PerfEvtSel0	These MSR addresses are aliases for the base set of performance event-select registers PerfEvtSel[3:0] and exist when CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1.	0000_0000_0000_0000h
C001_0202h	PerfEvtSel1		
C001_0204h	PerfEvtSel2		
C001_0206h	PerfEvtSel3		
C001_0208h	PerfEvtSel4	Extended core performance event-select registers. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1.	
C001_020Ah	PerfEvtSel5		

[AMD Public Use]

Table A-6. Performance-Monitoring MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
C001_0201h	PerfCtr0	These MSR addresses are aliases for the base set of performance-monitoring counter registers PerfCtr[3:0] and exist when CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1.	undefined
C001_0203h	PerfCtr1		
C001_0205h	PerfCtr2		
C001_0207h	PerfCtr3		
C001_0209h	PerfCtr4	Extended core performance counter registers. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1.	
C001_020Bh	PerfCtr5		
C001_0230h	L2I_PerfEvtSel0	Specifies the L2 cache events to be counted and controls other aspects of counter operation. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtL2I] = 1.	0000_0000_0000_0000h
C001_0232h	L2I_PerfEvtSel1		
C001_0234h	L2I_PerfEvtSel2		
C001_0236h	L2I_PerfEvtSel3		
C001_0231h	L2I_PerfCtr0	Counts specific L2 cache events as specified by the corresponding L2I_PerfEvtSel*n* Register. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtL2I] = 1.	undefined
C001_0233h	L2I_PerfCtr1		
C001_0235h	L2I_PerfCtr2		
C001_0237h	L2I_PerfCtr3		
C001_0240h + 2i	NB_PerfEvtSeli (i=0..15)	Specifies Northbridge events to be counted and controls other aspects of counter operation. Support for NB_PerfEvtSeli (i=0..3) is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1; support for NB_PerfEvtSeli (i=4..15) is indicated by CPUID Fn8000_0022_EBX[NumPerfCtrNB] > i.	0000_0000_0000_0000h
C001_0241h + 2i	NB_PerfCtri (i=0..15)	Counts specific Northbridge events as specified by the corresponding NB_PerfEvtSeli register. Support for NB_PerfCtri (i=0..3) is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1; support for NB_PerfCtri (i=4..15) is indicated by CPUID Fn8000_0022_EBX[NumPerfCtrNB] > i.	undefined

A.7 Secure Virtual Machine MSRs

Table A-7 lists the MSRs used in support of SVM functions.

[AMD Public Use]

Table A-7. Secure Virtual Machine MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
C000_0104h	TSC Ratio	Ratio for scaling TSC, MPERF, and MPerfReadOnly values read by guest.	0000_0001_0000_0000h
C001_0114h	VM_CR	Controls certain global aspects of SVM.	undefined
C001_0115h	IGNNE	Sets the state of the processor-internal IGNNE signal.	
C001_0116h	SMM_CTL	Provides software control over SMM signals.	
C001_0117h	VM_HSAVE_PA	Holds the physical address of a block of memory where VMRUN saves host state, and from which #VMEXIT reloads host state.	
C001_0118h	SVM_KEY	Creates a password-protected mechanism to clear VM_CR.LOCK.	
C001_011Bh	Doorbell Register	Sends a doorbell signal to the specified physical APIC.	
C001_011Eh	VMPAGE_FLUSH	“Secure Encrypted Virtualization” on page 613	
C001_0130h	GHCB	Guest-HV Communication Block address (see section 15.35.7)	
C001_0131h	SEV_STATUS	SEV active features indication (see section 15.35.10)	
C001_0132h	RMP_BASE	Base address of RMP (see 15.36.4)	0000_0000_0000_0000h
C001_0133h	RMP_END	Ending address of RMP (see 15.36.4)	0000_0000_0000_1FFFh
C001_0134h	GUEST_TSC_FREQ	Guest TSC Frequency (see 15.36.18)	0000_0000_0000_0000h
C001_0135h	VIRTUAL_TOM	Virtual Top-of-Memory (see “Virtual Top-of-Memory” on page 633)	0000_0000_0000_0000h
C001_0136h	SEGMENTED_RMP_CFG	Segmented RMP Configuration (see “Segmented RMP” on page 649)	0000_0000_0000_0000h
C001_0137h	IDLE_WAKEUP_ICR	Used to wake a sibling thread when SMT Protection is active (see Section “SMT Protection” on page 642, “Side-Channel Protection” on page 641)	0000_0000_0000_0000h
C001_0138h	SECURE_AVIC_CTRL	Configures the Secure AVIC feature in a guest (see Section 15.36.21.3, “Secure AVIC Control MSR,” on page 648)	0000_0000_0000_0000h
C001_0139h	RMPOPT_BASE	RMPOPT Base	0
C001_013Ah	VCPU_ID	vCPU ID	undefined

[AMD Public Use]

A.8 System Management Mode MSRs

Table A-8 lists the MSRs used in support of SMM functions.

Table A-8. System Management Mode MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
C001_0056h	SMI_Trigger_IO_Cycle	Specifies an I/O cycle that may be generated when a local SMI trigger event occurs. See the appropriate See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.	0000_0000_0000_0000h
C001_0061h	P-State Current Limit		
C001_0062h	P-State Control		
C001_0063h	P-State Status		
C001_0111h	SMBASE	Contains the SMRAM base address.	0000_0000_0003_0000h
C001_0112h	SMM_ADDR	Contains the base address of protected memory for the SMM Handler.	0000_0000_0000_0000h
C001_0113h	SMM_MASK	Contains a mask which determines the size of the protected area for the SMM handler.	0000_0000_0000_0000h
C001_0119h	SMM_KEY		
C001_011Ah	Local_SMI_Status	Contains status associated with SMI sources local to the CPU core. See the appropriate *BIOS and Kernel Developer's Guide* or *Processor Programming Reference Manual* for details.	0000_0000_0000_0000h

A.9 CPUID Name MSR Cross-Reference

Table A-9 lists the MSRs used to support CPUID namestring.

Table A-9. CPUID Namestring MSR Cross Reference

MSR Address	MSR Name	Description	Reset Value
C001_0030h	Processor_Name_String	See appropriate *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* and *Processor Revision Guide*.	0000_0000_0000_0000h
C001_0031h			
C001_0032h			
C001_0033h			
C001_0034h			
C001_0035h			

[AMD Public Use]

A.10 Shadow Stack MSRs

Table A-10 lists the MSRs that support the shadow stack feature. These registers are defined if the shadow stack feature is present as indicated by CPUID Fn0000_0007_x0_ECX[CET_SS] (bit 7) =1.

Table A-10. Shadow Stack MSR Cross Reference

MSR Address	MSR Name	Description	Reset Value
06A0h	U_CET	User-mode Shadow Stack Controls	0000_0000_0000_0000h
06A2h	S_CET	Supervisor-mode Shadow Stack Controls	
06A4h	PL0_SSP	CPL 0 Shadow Stack Pointer	
06A5h	PL1_SSP	CPL 1 Shadow Stack Pointer	
06A6h	PL2_SSP	CPL 2 Shadow Stack Pointer	
06A7h	PL3_SSP	CPL 3 Shadow Stack Pointer	
06A8h	ISST_ADDR	Interrupt Shadow Stack Table Base Address	

A.11 Speculation Control MSRs

Table A-11 lists the MSRs that support speculation control. See “Speculation Control Registers” on page 67 for further details, including how to determine whether these registers are defined.

Table A-11. Speculation Control MSRs

MSR Address	MSR Name	Description	Reset Value
0048h	SPEC_CTRL	Speculation Control	0000_0000_0000_0000h
0049h	PRED_CMD	Prediction Control	
C001_011Fh	VIRT_SPEC_CTRL	Virtual Speculation Control	

A.12 Memory Cache MSRs

Table A-12. Memory Cache MSRs

MSR Address	MSR Name	Description	Reset Value
10Bh	FLUSH_CMD	L1 Cache Flush	
C000_0108h	PrefetchControl	Hardware Prefetcher Control	See the appropriate *BIOS and Kernel Developer's Guide or Processor Programming Reference Manual*
C001_1095h	L3RangeReserveBaseAddr	L3 Range Reserve Base Address Register	
C001_1096h	L3RangeReserveMaxAddr	L3 Range Reserve Maximum Address Register	
C001_109Ah	L3RangeReserveWayMask	L3 Range Reserve Way Mask	

[AMD Public Use]

A.13 Quality of Service MSRs

Table A-13. Quality of Service MSRs

MSR Address	MSR Name	Description	Reset Value
0C81h	L3_QOS_CFG1	CDP Enable	0
0C8Dh	QM_EVTSEL	QOS Event Selection	0
0C8Eh	QM_CTR	QOS Counter	0
0C8Fh	PQR_ASSOC	RMID Association	0
0C90h+n	L3_MASK_n	L3 Allocation Mask	Bits CBM_LEN:0 are set, all others cleared
C000_0200+n	L3QOS_BW_CONTROL_n	L3 Bandwidth Control	U bit set, all other bits cleared
C000_0280+n	L3QOS_SMBW_CONTROL_n	L3 Slow Memory Bandwidth Control	U bit set, all other bits cleared
C000_03FCh	PQR_PLZA_ASSOC	PLZA Association	0
C000_03FDh	L3_QOS_ABMC_CFG	ABMC Configuration	See 19.3.3.3
C000_03FEh	L3_QOS_ABMC_DSC	ABMC Discovery	0
C000_03FFh	L3_QOS_EXT_CFG	ABMC, SDCI Enables	0
C000_0400h	QOS_EVT_CFG_0	BMEC Configuration	See 19.3.3.2
C000_0401h	QOS_EVT_CFG_1	BMEC Configuration	See 19.3.3.2
C000_0600h+n	L3QOS_GL_BW_CONTROL_n	Global Bandwidth Control	U bit set, all other bits cleared
C000_0680h+n	L3QOS_GL_SLOWBW_ CONTROL_n	Global Slow Memory Bandwidth Control	U bit set, all other bits cleared

A.14 Collaborative Processor Performance Control MSRs

Table A-14. Collaborative Processor Performance Control MSRs

MSR Address	MSR Name	Description	Reset Value
C001_02B0h	CPPC_CAPABILITY_1	CPPC Performance Ranges	0
C001_02B1h	CPPC_ENABLE	CPPC Enable	0
C001_02B2h	CPPC_CAPABILITY_2	CPPC Constrained Max Performance Level	0
C001_02B3h	CPPC_REQUEST	CPPC Request	0
C001_02B4h	CPPC_STATUS	CPPC Status	0
C001_02B5h	CPPC_REQUEST_2	CPPC Performance Floor Request	0

[AMD Public Use]

A.15 Instruction-Based Sampling MSRs

Table A.15 lists the MSRs used in support of Instruction-Based Sampling (IBS). For more information on IBS, see Section 13.3, “Instruction-Based Sampling,” on page 426.

Table A-15. Instruction-Based Sampling MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
C001_0380h	IbsMemCtl	Controls Memory Profiler op sampling and op execution status.	0000_0000_0000_0000h
C001_0381h	IbsMemCtl2	Extended Memory Profiler control.	0000_0000_0000_0000h
C001_0382h	IbsMemData2	Northbridge data source information.	0000_0000_0000_0000h
C001_0383h	IbsMemData3	Data cache access status for Memory Profiler load or store ops.	0000_0000_0000_0000h
C001_0384h	IbsMemLinAd	Linear address of the Memory Profiler tagged op's memory operand.	0000_0000_0000_0000h
C001_0385h	IbsMemPhysAd	Physical address of the Memory Profiler tagged op's memory operand.	0000_0000_0000_0000h
C001_0386h	IbsMemRip	Linear address of the Memory Profiler tagged op's instruction.	0000_0000_0000_0000h
C001_0390h	IbsBufferBase	Base address and control bits for the IBS Buffer.	0000_0000_0000_0000h
C001_0391h	IbsBufferSize	Size of the IBS Buffer.	0000_0000_0000_0000h
C001_0392h	IbsBufferTail	Tail pointer and status bits for the IBS Buffer.	0000_0000_0000_0000h
C001_1030h	IbsFetchCtl	Controls fetch sampling and captures fetch status.	0000_0000_0000_0000h
C001_1031h	IbsFetchLinAd	Linear address of the tagged instruction fetch.	0000_0000_0000_0000h
C001_1032h	IbsFetchPhysAd	Physical address of the tagged instruction fetch.	0000_0000_0000_0000h
C001_1033h	IbsOpCtl	Controls op sampling and captures op execution status.	0000_0000_0000_0000h
C001_1034h	IbsOpRip	Linear address of the tagged op's instruction.	0000_0000_0000_0000h
C001_1035h	IbsOpData1	Provides core cycle counts for tagged ops and performance data for branch ops.	0000_0000_0000_0000h
C001_1036h	IbsOpData2	Northbridge data source information for the tagged op.	0000_0000_0000_0000h
C001_1037h	IbsOpData3	Data cache access status load or store ops.	0000_0000_0000_0000h
C001_1038h	IbsDcLinAd	Linear address of the tagged op's memory operand.	0000_0000_0000_0000h
C001_1039h	IbsDcPhysAd	Physical address of the tagged op's memory operand.	0000_0000_0000_0000h
C001_103Ah	IbsCtl	Specifies the extended LVT offset for IBS interrupt delivery. Read-only.	Implementation-specific
C001_103Bh	IbsBrTarget	Branch target linear address for branch ops.	0000_0000_0000_0000h
C001_103Ch	IbsFetchExtdCtl	Extended fetch status.	0000_0000_0000_0000h
C001_103Eh	IbsOpCtl2	Extended Execution IBS control.	0000_0000_0000_0000h
C001_103Fh	IbsFetchCtl2	Extended Fetch IBS control.	0000_0000_0000_0000h

[AMD Public Use]

Appendix B VMCB Layout

The VMCB is divided into two areas—the first one contains various control bits including the intercept vectors and the second one contains saved guest state.

Table B-1 describes the layout of the control area of the VMCB, which starts at offset zero within the VMCB page. The control area is padded to a size of 1024 bytes. All unused bytes must be zero, as they are reserved for future expansion. It is recommended that software zero out any newly allocated VMCB.

Table B-1. VMCB Layout, Control Area

Byte Offset	Bit(s)	Function
000h (vector 0)	15:0	Intercept reads of CR0–15, respectively
	31:16	Intercept writes of CR0–15, respectively
004h (vector 1)	15:0	Intercept reads of DR0–15, respectively
	31:16	Intercept writes of DR0–15, respectively.
008h (vector 2)	31:0	Intercept exception vectors 0–31, respectively
00Ch (vector 3)	0	Intercept INTR (physical maskable interrupt)
	1	Intercept NMI
	2	Intercept SMI
	3	Intercept INIT
	4	Intercept VINTR (virtual maskable interrupt)
	5	Intercept CR0 writes that change bits other than CR0.TS or CR0.MP
	6	Intercept reads of IDTR
	7	Intercept reads of GDTR
	8	Intercept reads of LDTR
	9	Intercept reads of TR
	10	Intercept writes of IDTR
	11	Intercept writes of GDTR
	12	Intercept writes of LDTR
	13	Intercept writes of TR
	14	Intercept RDTSC instruction
	15	Intercept RDPMC instruction

[AMD Public Use]

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
00Ch (continued)	16	Intercept PUSHF instruction
	17	Intercept POPF instruction
	18	Intercept CPUID instruction
	19	Intercept RSM instruction
	20	Intercept IRET instruction
	21	Intercept INT*n* instruction
	22	Intercept INVD instruction
	23	Intercept PAUSE instruction
	24	Intercept HLT instruction
	25	Intercept INVLPG instruction
	26	Intercept INVLPGA instruction
	27	IOIO_PROT—Intercept IN/OUT accesses to selected ports
	28	MSR_PROT—intercept RDMSR or WRMSR accesses to selected MSRs
	29	Intercept task switches
	30	FERR_FREEZE: intercept processor “freezing” during legacy FERR handling
	31	Intercept shutdown events

[AMD Public Use]

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
010h (vector 4)	0	Intercept VMRUN instruction
	1	Intercept VMMCALL instruction
	2	Intercept VMLOAD instruction
	3	Intercept VMSAVE instruction
	4	Intercept STGI instruction
	5	Intercept CLGI instruction
	6	Intercept SKINIT instruction
	7	Intercept RDTSCP instruction
	8	Intercept ICEBP instruction
	9	Intercept WBINVD and WBNOINVD instructions
	10	Intercept MONITOR/MONITORX instruction
	11	Intercept MWAIT/MWAITX instruction unconditionally
	12	Intercept MWAIT/MWAITX instruction if monitor hardware is armed
	13	Intercept XSETBV instruction
	14	Intercept RDPRU instruction
	15	Intercept EFER writes (trap)
	16	Intercept CR0 writes (trap)
	18:17	RESERVED
	19	Intercept CR3 writes (trap)
	20	Intercept CR4 writes (trap)
	31:21	RESERVED
014h (vector 5)	0	Intercept all INVLPGB instructions
	1	Intercept only illegally specified INVLPGB instructions
	2	Intercept INVPCID instruction
	3	Intercept MCOMMIT instruction
	4	Intercept TLBSYNC instruction. Presence of this bit is indicated by CPUID Fn8000_000A, EDX[24] = 1.
	5	Intercept bus lock operations when Bus Lock Threshold Counter is 0 (occurs before guest instruction executes).
	6	Intercept HLT instruction if a virtual interrupt is not pending.
	31:7	RESERVED, SBZ
018h–03Bh	RESERVED, SBZ	
03Ch	15:0	PAUSE Filter Threshold
03Eh	15:0	PAUSE Filter Count
040h	63:0	IOPM_BASE_PA—Physical base address of IOPM
048h	63:0	MSRPM_BASE_PA—Physical base address of MSRPM

[AMD Public Use]

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
050h	63:0	TSC_OFFSET—To be added in RDTSC and RDTSCP
058h	31:0	Guest ASID
	39:32	TLB_CONTROL 00h—Do nothing 01h—Flush entire TLB (all entries, all ASIDs) on VMRUN Should only be used by legacy hypervisors 03h—Flush this guest's TLB entries 07h—Flush this guest's non-global TLB entries
	40	ALLOW_LARGER_RAP 0—RAP size for the guest is 32 1—RAP size for the guest is equal to CPUID Fn8000_0021_EBX[RapSize]
	41	CLEAR_RAP 1—Clear RAP on VMRUN
	63:42	RESERVED, SBZ
060h	7:0	V_TPR—The virtual TPR for the guest. Bits 3:0 are used for a 4-bit virtual TPR value; bits 7:4 are SBZ.
	8	V_IRQ—If nonzero, virtual INTR is pending
	9	VGIF—Virtual GIF
	11	V_NMI—If nonzero, virtual NMI is pending
	12	V_NMI_MASK - if nonzero, virtual NMI is masked
	15:13	RESERVED, SBZ
	19:16	V_INTR_PRIO—Priority for virtual interrupt.
	20	V_IGN_TPR—If nonzero, the current virtual interrupt ignores the virtual TPR.
	23:21	RESERVED, SBZ
	24	V_INTR_MASKING—Virtualize masking of INTR interrupts ("Virtualizing APIC TPR" on page 556)
	25	VGIF_EN—Virtual GIF Enable
	26	V_NMI_ENABLE—NMI Virtualization Enable (see "NMI Virtualization" on page 559)
	29:27	RESERVED, SBZ
	30	x2AVIC Enable (see "x2AVIC" on page 607)
	31	AVIC Enable
	39:32	V_INTR_VECTOR—Vector to use for virtual interrupt
	63:40	RESERVED, SBZ

[AMD Public Use]

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
068h	0	INTERRUPT_SHADOW—Guest is in an interrupt shadow
	1	GUEST_INTERRUPT_MASK—Value of the RFLAGS.IF bit for SEV-ES guest.
	63:2	RESERVED, SBZ
070h	63:0	EXITCODE
078h	63:0	EXITINFO1
080h	63:0	EXITINFO2
088h	63:0	EXITINTINFO
090h	0	NP_ENABLE—Enable nested paging.
	1	Enable Secure Encrypted Virtualization
	2	Enable Encrypted State for Secure Encrypted Virtualization
	3	Guest Mode Execute Trap
	4	SSSCheckEn—Enable supervisor shadow stack restrictions in nested page tables. Support for this feature is indicated by CPUID Fn8000_000A_EDX[19] (SSSCheck)
	5	Virtual Transparent Encryption.
	6	RogptEn—Enable Read-Only Guest Page Tables. See “Read-Only Guest Page Tables” on page 574
	7	Enable INVLPGB/TLBSYNC. 0 - INVLPGB and TLBSYNC will result in #UD. 1 - INVLPGB and TLBSYNC can be executed in guest. Presence of this bit is indicated by CPUID bit 8000_000A, EDX[24] = 1. When in SEV-ES guest or this bit is not present, INVLPGB/TLBSYNC is always enabled in guest if supported by processor.
	8	GaatEn—Enable Guest Alternate Address Translation
	9	PagingWriteEn—Enable Paging-Write
	10	VerifyPagingWriteEn—Enable Verify Paging-Write
	11	PmlEn—Enable Page Modification Logging
	63:12	RESERVED, SBZ
098h	63:52	RESERVED, SBZ
	51:0	AVIC APIC_BAR
0A0h	63:0	Guest physical address of GHCB
0A8h	63:0	EVENTINJ—Event injection (“Event Injection” on page 554 for details)
0B0h	63:0	N_CR3—Nested page table CR3 to use for nested paging

[AMD Public Use]

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
0B8h	0	LBR Virtualization Enable
	1	VMSAVE/VMLOAD Virtualization Enable
	2	IBS Virtualization Enable
	3	PMC Virtualization Enable
	63:4	RESERVED, SBZ
0C0h	31:0	VMCB Clean Bits.
	63:32	RESERVED, SBZ
0C8h	63:0	nRIP—Next sequential instruction pointer
0D0h	7:0	Number of bytes fetched
	127:8	Guest instruction bytes
0E0h	63:52	RESERVED, SBZ
	51:0	AVIC APIC_BACKING_PAGE Pointer
0E8h–0EFh	RESERVED, SBZ	
0F0h	63:52	RESERVED, SBZ
	51:12	AVIC LOGICAL_TABLE Pointer
	11:0	Reserved, SBZ
0F8h	63:52	RESERVED, SBZ
	51:12	AVIC PHYSICAL_TABLE Pointer[51:12]
	11:0	AVIC_PHYSICAL_MAX_INDEX
100h – 107h	RESERVED, SBZ	
108h	63:52	RESERVED, SBZ
	51:12	VMSA Pointer[51:12]
	11:0	RESERVED, SBZ
110h	63:0	VMGEXIT_RAX
118h	7:0	VMGEXIT_CPL
120h	15:0	Bus Lock Threshold Counter
128h	63:52	RESERVED, MBZ
	51:12	GAAT Pointer
	11:5	RESERVED, SBZ
	4	PCD
	3	PWT
	2:0	RESERVED, SBZ
130h	31:6	RESERVED, SBZ
	5:0	GAAT_PREFIX_WIDTH
134h	0	UPDATE_IRR

[AMD Public Use]

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
138h	63	ALLOWED_SEV_FEATURES_EN
	62	RESERVED, SBZ
	61:0	ALLOWED_SEV_FEATURES_MASK
140h	61:0	GUEST_SEV_FEATURES
148h	63:0	ESMTP_TIMEOUT_CTL
150h	255:0	REQUESTED_IRR
170h	63:0	EXITINTDATA
178h	63:0	EVENTINJDATA
180h	63:0	ALLOWED_PERF_CTL0
188h	63:0	ALLOWED_PERF_CTL1
190h	63:0	ALLOWED_PERF_CTL2
198h	63:0	ALLOWED_PERF_CTL3
1A0h	63:0	ALLOWED_PERF_CTL4
1A8h	63:0	ALLOWED_PERF_CTL5
1B0h – 1BFh	RESERVED, SBZ	
1C0h	63:0	ALLOWED_PERF_CTL_MASK
1C8h	63:0	PML_BASE
1D0h	15:0	PML_INDEX
1D4h – 3DFh	RESERVED, SBZ	
3E0h – 3FFh	Reserved for Host usage	

When SEV-ES is not enabled, the state-save area within the VMCB starts at offset 400h into the VMCB page; Table B-2 describes the fields within the state-save area; note that the table lists offsets *relative to the state-save area* (not the VMCB as a whole).

Table B-2. VMCB Layout, State Save Area

Offset	Size	Contents		Notes
000h	word	ES	selector	
002h	word		attrib	
004h	dword		limit	
008h	qword		base	Only lower 32 bits are implemented
010h	word	CS	selector	
012h	word		attrib	
014h	dword		limit	
018h	qword		base	Only lower 32 bits are implemented

[AMD Public Use]

Table B-2. VMCB Layout, State Save Area (continued)

Offset	Size	Contents		Notes
020h	word	SS	selector	
022h	word		attrib	
024h	dword		limit	
028h	qword		base	Only lower 32 bits are implemented
030h	word	DS	selector	
032h	word		attrib	
034h	dword		limit	
038h	qword		base	Only lower 32 bits are implemented
040h	word	FS	selector	
042h	word		attrib	
044h	dword		limit	
048h	qword		base	
050h	word	GS	selector	
052h	word		attrib	
054h	dword		limit	
058h	qword		base	
060h	word	GDTR	selector	RESERVED
062h	word		attrib	RESERVED
064h	dword		limit	Only lower 16 bits are implemented
068h	qword		base	
070h	word	LDTR	selector	
072h	word		attrib	
074h	dword		limit	
078h	qword		base	
080h	word	IDTR	selector	RESERVED
082h	word		attrib	RESERVED
084h	dword		limit	Only lower 16 bits are implemented
088h	qword		base	
090h	word	TR	selector	
092h	word		attrib	
094h	dword		limit	
098h	qword		base	
0A0h–0CAh		RESERVED		
0CBh	byte	CPL		If the guest is real-mode then the CPL is forced to 0; if the guest is virtual-mode then the CPL is forced to 3
0CCh	dword	RESERVED		

[AMD Public Use]

Table B-2. VMCB Layout, State Save Area (continued)

Offset	Size	Contents	Notes
0D0h	qword	EFER	
0D8h–0DFh		RESERVED	
0E0h	qword	PERF_CTL0	
0E8h	qword	PERF_CTR0	
0F0h	qword	PERF_CTL1	
0F8h	qword	PERF_CTR1	
100h	qword	PERF_CTL2	
108h	qword	PERF_CTR2	
110h	qword	PERF_CTL3	
118h	qword	PERF_CTR3	
120h	qword	PERF_CTL4	
128h	qword	PERF_CTR4	
130h	qword	PERF_CTL5	
138h	qword	PERF_CTR5	
148h	qword	CR4	
150h	qword	CR3	
158h	qword	CR0	
160h	qword	DR7	
168h	qword	DR6	
170h	qword	RFLAGS	
178h	qword	RIP	
180h–1BFh		RESERVED	
1C0h	qword	INSTR_RETIRED_CTR	
1C8h	qword	PERF_CTR_GLOBAL_STS	
1D0h	qword	PERF_CTR_GLOBAL_CTL	
1D8h	qword	RSP	
1E0h	qword	S_CET	
1E8h	qword	SSP	
1F0h	qword	ISST_ADDR	
1F8h	qword	RAX	
200h	qword	STAR	
208h	qword	LSTAR	
210h	qword	CSTAR	
218h	qword	SFMASK	
220h	qword	KernelGsBase	
228h	qword	SYSENTER_CS	

[AMD Public Use]

Table B-2. VMCB Layout, State Save Area (continued)

Offset	Size	Contents	Notes
230h	qword	SYSENTER_ESP	
238h	qword	SYSENTER_EIP	
240h	qword	CR2	
248h–267h		RESERVED	
268h	qword	G_PAT	Guest PAT—only used if nested paging enabled
270h	qword	DBGCTL	Guest DebugCtl MSR—only used if hardware acceleration of LBR virtualization is supported and enabled by setting the LBR_VIRTUALIZATION_ENABLE bit of the VMCB control area.
278h	qword	BR_FROM	Guest LastBranchFromIP MSR—only used if hardware acceleration of LBR virtualization is supported and enabled.
280h	qword	BR_TO	Guest LastBranchToIP MSR—only used if hardware acceleration of LBR virtualization is supported and enabled.
288h	qword	LASTEXCPFROM	Guest LastIntFromIP MSR—Only used if hardware acceleration of LBR virtualization is supported and enabled.
290h	qword	LASTEXCPTO	Guest LastIntToIP MSR—Only used if hardware acceleration of LBR virtualization is supported and enabled.
298h	qword	DBGEXTNCTL	Guest DebugExtnCtl MSR—only used if hardware acceleration of LBR Stack virtualization is supported and enabled by setting the LBR_VIRTUALIZATION_ENABLE bit of the VMCB control area.
2A0h–2DFh	72 bytes	RESERVED	
2E0h	qword	SPEC_CTRL	
2E8h–66Fh	904 bytes	RESERVED	
670h–76Fh	256 bytes	LBR_STACK_FROM LBR_STACK_TO	Guest LastBranchStackFromIp and LastBranchStackToIp MSRs in MSR address order — only used if hardware acceleration of LBR Stack virtualization is supported and enabled by setting the LBR_VIRTUALIZATION_ENABLE bit of the VMCB control area.

[AMD Public Use]

Table B-2. VMCB Layout, State Save Area (continued)

Offset	Size	Contents	Notes
770h	qword	LBR_SELECT	Guest LastBranchStackSelect MSR - only used if hardware acceleration of LBR Stack virtualization is supported and enabled by setting the LBR_VIRTUALIZATION_ENABLE bit of the VMCB control area.
778h	qword	IBS_FETCH_CTL	IBS Virtualization state
780h	qword	IBS_FETCH_ LINADDR	
788h	qword	IBS_OP_CTL	
790h	qword	IBS_OP_RIP	
798h	qword	IBS_OP_DATA	
7A0h	qword	IBS_OP_DATA2	
7A8h	qword	IBS_OP_DATA3	
7B0h	qword	IBS_DC_LINADDR	
7B8h	qword	IBS_BRTGT_RIP	
7C0h	qword	IBS_FETCH_EXTD_CTL	
7C8h	qword	IBS_FETCH_CTL2	
7D0h	qword	IBS_OP_CTL2	
7D8h	qword	IBS_BUFFER_BASE	
7E0h	dword	IBS_BUFFER_SIZE	
7E4h	dword	IBS_BUFFER_TAIL	
7E8h-8BFh	216 bytes	RESERVED	
8C0h	qword	FRED_RSP1	
8C8h	qword	FRED_RSP2	
8D0h	qword	FRED_RSP3	
8D8h	qword	FRED_STKLVLS	
8E0h	qword	FRED_SSP1	
8E8h	qword	FRED_SSP2	
8F0h	qword	FRED_SSP3	
8F8h	qword	FRED_CONFIG	
900h–BFFh		RESERVED	

When SEV-ES is enabled (Section 15.35 “Encrypted State (SEV-ES)” on page 620), the VMSA structure starts at offset 0h in the page indicated by the VMSA Pointer. The format of the VM save state for SEV-ES guests is described in the table below.

[AMD Public Use]

All state is categorized into 3 swap types based on how it is handled by hardware during a world switch:

Table B-3. Swap Types

Swap Type	Behavior in VMRUN	Behavior in AE VMEXIT
A	Host state saved to host save area Guest state loaded from VMSA	Guest state saved to VMSA Host state loaded from host save area
B	Guest state loaded from VMSA (Host state **not** saved to host save area)	Guest state saved to VMSA Host state loaded from host save area
C	Guest state loaded from VMSA (Host state **not** saved to host save area)	Guest state saved to VMSA Host state initialized to default (reset) values

The format of the host save area is identical to the guest save area described in the table below, except that it begins at offset 400h in the host save page (For example, the host TR value is stored at offset 490h relative to the start of the host save page.)

Table B-4. VMSA Layout, State Save Area for SEV-ES

Offset	Size	Content	Swap Type	Notes
000h	16 bytes	ES	A	
010h	16 bytes	CS	A	
020h	16 bytes	SS	A	
030h	16 bytes	DS	A	
040h	16 bytes	FS	B	
050h	16 bytes	GS	B	
060h	16 bytes	GDTR	A	
070h	16 bytes	LDTR	B	
080h	16 bytes	IDTR	A	
090h	16 bytes	TR	B	
0A0h	qword	PL0_SSP	B	
0A8h	qword	PL1_SSP	B	
0B0h	qword	PL2_SSP	B	
0B8h	qword	PL3_SSP	B	
0C0h	qword	U_CET	B	
0C8h	dword	RESERVED	–	
0CAh	byte	VMPL	–	Swapped for guest. Not used in host mode.
0CBh	byte	CPL	A	
0CCh	dword	RESERVED	–	
0D0h	qword	EFER	A	

[AMD Public Use]

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
0D8h-0DFh	8 bytes	RESERVED	–	
0E0h	qword	PERF_CTL0	C	
0E8h	qword	PERF_CTR0	C	
0F0h	qword	PERF_CTL1	C	
0F8h	qword	PERF_CTR1	C	
100h	qword	PERF_CTL2	C	
108h	qword	PERF_CTR2	C	
110h	qword	PERF_CTL3	C	
118h	qword	PERF_CTR3	C	
120h	qword	PERF_CTL4	C	
128h	qword	PERF_CTR4	C	
130h	qword	PERF_CTL5	C	
138h	qword	PERF_CTR5	C	
140h	qword	XSS	B	
148h	qword	CR4	A	
150h	qword	CR3	A	
158h	qword	CR0	A	
160h	qword	DR7	C	
168h	qword	DR6	C	
170h	qword	RFLAGS	A	
178h	qword	RIP	A	
180h	qword	DR0	B	
188h	qword	DR1	B	
190h	qword	DR2	B	
198h	qword	DR3	B	
1A0h	qword	DR0_ADDR_MASK	B	
1A8h	qword	DR1_ADDR_MASK	B	
1B0h	qword	DR2_ADDR_MASK	B	
1B8h	qword	DR3_ADDR_MASK	B	
1C0h	qword	INSTR_RETIRED_CTR	A	
1C8h	qword	PERF_CTR_GLOBAL_STS	A	
1D0h	qword	PERF_CTR_GLOBAL_CTL	C	
1D8h	qword	RSP	A	
1E0h	qword	S_CET	A	
1E8h	qword	SSP	A	
1F0h	qword	ISST_ADDR	A	

[AMD Public Use]

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
1F8h	qword	RAX	A	
200h	qword	STAR	B	
208h	qword	LSTAR	B	
210h	qword	CSTAR	B	
218h	qword	SFMASK	B	
220h	qword	KernelGsBase	B	
228h	qword	SYSENTER_CS	B	
230h	qword	SYSENTER_ESP	B	
238h	qword	SYSENTER_EIP	B	
240h	qword	CR2	C	
248h-267h	32 bytes	RESERVED	–	
268h	qword	G_PAT	–	Swapped for guest, not used in host mode.
270h	qword	DBGCTL	A	
278h	qword	BR_FROM	A	
280h	qword	BR_TO	A	
288h	qword	LASTEXCPFROM	A	
290h	qword	LASTEXCPTO	A	
298h	qword	DBGEXTNCTL	A	
2A0-2DFh	72 bytes	RESERVED	–	
2E0h	qword	SPEC_CTRL	A	
2E8h	dword	PKRU	B	
2ECh	dword	TSC_AUX	B	
2F0h	qword	GUEST_TSC_SCALE	–	
2F8h	qword	GUEST_TSC_OFFSET	–	
300h	qword	REG_PROT_NONCE	–	
308h	qword	RCX	B	
310h	qword	RDX	B	
318h	qword	RBX	B	
320h	qword	SECURE_AVIC_CTL	–	
328h	qword	RBP	B	
330h	qword	RSI	B	
338h	qword	RDI	B	
340h	qword	R8	B	
348h	qword	R9	B	
350h	qword	R10	B	

[AMD Public Use]

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
358h	qword	R11	B	
360h	qword	R12	B	
368h	qword	R13	B	
370h	qword	R14	B	
378h	qword	R15	B	
380h	16 bytes	RESERVED	–	
390h	qword	GUEST_EXITINFO1	–	EXITINFO1 for AE exits
398h	qword	GUEST_EXITINFO2	–	EXITINFO2 for AE exits
3A0h	qword	GUEST_EXITINTINFO	–	EXITINTINFO for AE exits
3A8h	qword	GUEST_NRIP	–	Next sequential instruction pointer for AE exits
3B0h	qword	SEV_FEATURES	–	Guest-controlled SEV feature selection • Bit 0: SNPActive • Bit 1: vTOM • Bit 2: ReflectVC • Bit 3: RestrictedInjection • Bit 4: AlternateInjection • Bit 5: DebugVirtualization • Bit 6: PreventHostIBS • Bit 7: BTBIsolation • Bit 8: VmplSSS • Bit 9: SecureTSC • Bit 10: VmgexitParameter • Bit 11: PmcVirtualization * • Bit 12: IbsVirtualization • Bit 13: GuestInterceptCtl * • Bit 14: VmsaRegProt • Bit 15: SmtProtection • Bit 16: SecureAvic * • Bit 17: EnhSmtProtection * • Bit 20:18: Reserved, SBZ • Bit 21: IbpbOnEntry • Bits 63:22: Reserved, SBZ * - This feature may only be used if Allowed SEV Features is enabled and the Allowed SEV Features Mask permits the use of the feature.

[AMD Public Use]

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
3B8h	qword	VINTR_CTRL	–	• Bits 7:0: V_TPR • Bit 8: V_IRQ • Bit 9: VGIF • Bit 10: INT_SHADOW • Bit 11: V_NMI • Bit 12: V_NMI_MASK • Bits 15:13: Reserved, SBZ • Bits 19:16: V_INTR_PRIO • Bit 20: V_IGN_TPR • Bits 25:21: Reserved, SBZ • Bit 26: V_NMI_ENABLE • Bits 31:27: Reserved, SBZ • Bits 39:32: V_INTR_VECTOR • Bits 62:40: Reserved, SBZ • Bit 63: BUSY
3C0h	qword	GUEST_EXITCODE	–	EXITCODE for AE exits
3C8h	qword	VIRTUAL_TOM	–	Swapped for guest, not used in host mode. Only bits 51:21 are observed.
3D0h	qword	TLB_ID	–	
3D8h	qword	PCPU_ID	–	
3E0h	qword	EVENTINJ	–	Same as the EVENTINJ field in the VMCB (Table B-1) at offset 0A8h.
3E8h	qword	XCR0	B	
3F0h-3FFh	16 bytes	Reserved	–	
400h	qword	X87_DP	C	FP x87 data pointer
408h	dword	MXCSR	C	FP MXCSR
40Ch	word	X87_FTW	C	FP x87 tag word
40Eh	word	X87_FSW	C	FP x87 status word
410h	word	X87_FCW	C	FP control word
412h	word	X87_FOP	C	FP x87 opcode
414h	word	X87_DS	C	FP x87 DS
416h	word	X87_CS	C	FP x87 CS
418h	qword	X87_RIP	C	FP x87 RIP
420h-46Fh	80 bytes	FPREG_X87	C	X87 register state (stack order)
470h-56Fh	256 bytes	FPREG_XMM	C	XMM register state
570h-66Fh	256 bytes	FPREG_YMM	C	YMM_HI register state

[AMD Public Use]

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
670h-76Fh	256 bytes	LBR_STACK_FROM LBR_STACK_TO	C	LBR Stack state
770h	qword	LBR_SELECT	C	LastBranchStackSelect state
778h	qword	IBS_FETCH_CTL	C	IBS Virtualization state
780h	qword	IBS_FETCH_LINADDR	C	
788h	qword	IBS_OP_CTL	C	
790h	qword	IBS_OP_RIP	C	
798h	qword	IBS_OP_DATA	C	
7A0h	qword	IBS_OP_DATA2	C	
7A8h	qword	IBS_OP_DATA3	C	
7B0h	qword	IBS_DC_LINADDR	C	
7B8h	qword	IBS_BRTGT_RIP	C	
7C0h	qword	IBS_FETCH_EXTD_CTL	C	
7C8h	qword	IBS_FETCH_CTL2	C	
7D0h	qword	IBS_OP_CTL2	C	
7D8h	qword	IBS_BUFFER_BASE	C	
7E0h	dword	IBS_BUFFER_SIZE	C	
7E4h	dword	IBS_BUFFER_TAIL	C	
7E8h-89Fh	184 bytes	RESERVED	–	
8A0h	dword	VCPU_ID	–	
8A4h	dword	VCPU_SIBLING_MASK	–	
8A8h	qword	GUEST_EXITINTDATA	–	
8B0h	qword	GUEST_EVENTINJDATA	–	
8B8h	qword	FRED_RSP0	A	
8C0h	qword	FRED_RSP1	A	
8C8h	qword	FRED_RSP2	A	
8D0h	qword	FRED_RSP3	A	
8D8h	qword	FRED_STKLVLS	A	
8E0h	qword	FRED_SSP1	A	
8E8h	qword	FRED_SSP2	A	
8F0h	qword	FRED_SSP3	A	
8F8h	qword	FRED_CONFIG	A	

[AMD Public Use]

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
900h	dword	INTERCEPT_VEC_0	–	Guest Intercept Control See Section 15.36.23, “Guest Intercept Control,” on page 652.
904h	dword	INTERCEPT_VEC_1	–	
908h	dword	INTERCEPT_VEC_2	–	
90Ch	dword	INTERCEPT_VEC_3	–	
910h	dword	INTERCEPT_VEC_4	–	
914h	dword	INTERCEPT_VEC_5	–	
918h	dword	INTERCEPT_VEC_6	–	
91Ch	dword	INTERCEPT_VEC_7	–	
920h	qword	INTERCEPT_MSR_VEC_0	–	See Table B-5.
928h	qword	INTERCEPT_MSR_VEC_1	–	See Table B-6.
930h	qword	INTERCEPT_MSR_VEC_2	–	See Table B-7.
938h	qword	INTERCEPT_MSR_VEC_3	–	See Table B-8.
940h-97Fh	64 bytes	RESERVED	–	
980h-9BFh	64 bytes	FPREG_K	C	AVX512 K0-K7 opmask registers
9C0h-BBFh	512 bytes	FPREG_ZMMHI	C	ZMM0-ZMM15 register state, upper half
BC0h-FBFh	1024 bytes	FPREG_HIZMM	C	ZMM16-ZMM31 register state

B.1 Guest MSR Intercepts

The guest-controlled MSR intercept layout for VMSA fields INTERCEPT_MSR_VEC0-3 is shown in Table B-5, Table B-6, Table B-7, and Table B-8. See Section 15.36.23, “Guest Intercept Control,” on page 652.

Table B-5. INTERCEPT_MSR_VEC0 Layout

RDMSR Intercept Bit	WRMSR Intercept Bit	MSR Name	MSR Address
0	1	FS_BASE	C000_0100h
2	3	GS_BASE	C000_0101h
4	5	KERNEL_GS_BASE	C000_0102h
6	7	EFER	C000_0080h
8	9	STAR	C000_0081h
10	11	LSTAR	C000_0082h
12	13	CSTAR	C000_0083h

[AMD Public Use]

Table B-5. INTERCEPT_MSR_VEC0 Layout (continued)

RDMSR Intercept Bit	WRMSR Intercept Bit	MSR Name	MSR Address
14	15	SF_MASK	C000_0084h
16	17	SYSENTER_CS	174h
18	19	SYSENTER_RSP	175h
20	21	SYSENTER_RIP	176h
22	23	CET_U	6A0h
24	25	CET_S	6A2h
26	27	PL0_SSP	6A4h
28	29	PL1_SSP	6A5h
30	31	PL2_SSP	6A6h
32	33	PL3_SSP	6A7h
34	35	ISST_ADDR	6A8h
36	37	XSS	DA0h
38	39	DBG_CTL	1D9h
40	41	LAST_BR_FROM_IP	1DBh
42	43	LAST_BR_TO_IP	1DCh
44	45	LAST_INT_FROM_IP	1DDh
46	47	LAST_INT_TO_IP	1DEh
48	49	LBR_SELECT	C000_010Eh
50	51	DebugExtnCtl	C000_010Fh
52	53	DR0_ADDR_MASK	C001_1027h
54	55	DR1_ADDR_MASK	C001_1019h
56	57	DR2_ADDR_MASK	C001_101Ah
58	59	DR3_ADDR_MASK	C001_101Bh
60	61	SPEC_CTRL	48h
62	63	PRED_CMD	49h

Table B-6. INTERCEPT_MSR_VEC1 Layout

RDMSR Intercept Bit	WRMSR Intercept Bit	MSR Name	MSR Address
0	1	PERF_CTL0	C001_0000h, C001_0200h
2	3	PERF_CTR0	C001_0004h, C001_0201h
4	5	PERF_CTL1	C001_0001h, C001_0202h
6	7	PERF_CTR1	C001_0005h, C001_0203h
8	9	PERF_CTL2	C001_0002h, C001_0204h
10	11	PERF_CTR2	C001_0006h, C001_0205h

[AMD Public Use]

Table B-6. INTERCEPT_MSR_VEC1 Layout (continued)

RDMSR Intercept Bit	WRMSR Intercept Bit	MSR Name	MSR Address
12	13	PERF_CTL3	C001_0003h, C001_0206h
14	15	PERF_CTR3	C001_0007h, C001_0207h
16	17	PERF_CTL4	C001_0208h
18	19	PERF_CTR4	C001_0209h
20	21	PERF_CTL5	C001_020Ah
22	23	PERF_CTR5	C001_020Bh
24	25	Reserved	
26	27	Reserved	
28	29	Reserved	
30	31	Reserved	
32	33	INST_RET_CNT	C000_00E9h
34	35	IBS_FETCH_CTL	C001_1030h
36	37	IBS_FETCH_LIN_ADDR	C001_1031h
38	39	IBS_OP_CTL	C001_1033h
40	41	IBS_OP_RIP	C001_1034h
42	43	IBS_OP_DATA	C001_1035h
44	45	IBS_OP_DATA2	C001_1036h
46	47	IBS_OP_DATA3	C001_1037h
48	49	IBS_DC_LINADDR	C001_1038h
50	51	IBS_BR_TGT_RIP	C001_103Bh
52	53	IBS_FETCH_EXTD_CTL	C001_103Ch
54	55	LAST_BRANCH_STACK_FROM/TO	C001_0300h - C001_031Fh
56	57	PERF_GLOBAL_STS	C000_0300h
58	59	PERF_GLOBAL_CTL	C000_0301h
60	61	PERF_GLOBAL_STS_CLR	C000_0302h
62	63	PERF_GLOBAL_STS_SET	C000_0303h

Table B-7. INTERCEPT_MSR_VEC2 Layout

RDMSR Intercept Bit	WRMSR Intercept Bit	MSR Name	MSR Address
0	1	PAT	277h
2	3	TSC_AUX	C000_0103h
4	5	GHCB	C001_0130h
6	7	GUEST_TSC_FREQ	C001_0134h
8	9	SEV_STATUS	C001_0131h

[AMD Public Use]

Table B-7. INTERCEPT_MSR_VEC2 Layout (continued)

RDMSR Intercept Bit	WRMSR Intercept Bit	MSR Name	MSR Address
10	11	VIRTUAL_TOM	C001_0135h
12	13	FRED_RSP0	1CCh
14	15	FRED_RSP1	1CDh
16	17	FRED_RSP2	1CEh
18	19	FRED_RSP3	1CFh
20	21	FRED_STKLVLS	1D0h
22	23	FRED_SSP1	1D1h
24	25	FRED_SSP2	1D2h
26	27	FRED_SSP3	1D3h
28	29	FRED_CONFIG	1D4h
30	31	Reserved	
32	33	Reserved	
34	35	IBSOPCTL2	C001_103Eh
36	37	IBSFETCHCTL2	C001_103Fh
38	39	IBSBUFFERBASE, IBSBUFFERSIZE, IBSBUFFERTAIL	C001_0390h, C001_0391h, C001_0392h
40 - 63		Reserved	

Table B-8. INTERCEPT_MSR_VEC3 Layout

RDMSR Intercept Bit	WRMSR Intercept Bit	MSR Name	MSR Address
0 - 63		Reserved	

[AMD Public Use]

Appendix C SVM Intercept Exit Codes

On a #VMEXIT, a reason for exit (an exit code) is stored in the EXITCODE field in the VMCB. The exit codes are defined in Table C-1. Intercept exit codes are equal to the bit position of the corresponding flag in the VMCB's intercept vector.

Table C-1. SVM Intercept Codes

Code	Name	Cause
0h–Fh	VMEXIT_CR[0–15]_READ	Read of CR 0 through 15, respectively.
10h–1Fh	VMEXIT_CR[0–15]_WRITE	Write of CR 0 through 15, respectively.
20h–2Fh	VMEXIT_DR[0–15]_READ	Read of DR 0 through 15, respectively.
30h–3Fh	VMEXIT_DR[0–15]_WRITE	Write of DR 0 through 15, respectively.
40h–5Fh	VMEXIT_EXCP[0–31]	Exception vector 0–31, respectively.
60h	VMEXIT_INTR	Physical INTR (maskable interrupt).
61h	VMEXIT_NMI	Physical NMI.
62h	VMEXIT_SMI	Physical SMI (the EXITINFO1 field provides more information).
63h	VMEXIT_INIT	Physical INIT.
64h	VMEXIT_VINTR	Virtual INTR.
65h	VMEXIT_CR0_SEL_WRITE	Write of CR0 changed bits other than CR0.TS or CR0.MP.
66h	VMEXIT_IDTR_READ	Read of IDTR.
67h	VMEXIT_GDTR_READ	Read of GDTR.
68h	VMEXIT_LDTR_READ	Read of LDTR.
69h	VMEXIT_TR_READ	Read of TR.
6Ah	VMEXIT_IDTR_WRITE	Write of IDTR.
6Bh	VMEXIT_GDTR_WRITE	Write of GDTR.
6Ch	VMEXIT_LDTR_WRITE	Write of LDTR.
6Dh	VMEXIT_TR_WRITE	Write of TR.
6Eh	VMEXIT_RDTSC	RDTSC instruction.
6Fh	VMEXIT_RDPMC	RDPMC instruction.
70h	VMEXIT_PUSHF	PUSHF instruction.
71h	VMEXIT_POPF	POPF instruction.
72h	VMEXIT_CPUID	CPUID instruction.
73h	VMEXIT_RSM	RSM instruction.
74h	VMEXIT_IRET	IRET instruction.
75h	VMEXIT_SWINT	Software interrupt (INT*n* instructions).
76h	VMEXIT_INVD	INVD instruction.
77h	VMEXIT_PAUSE	PAUSE instruction.
78h	VMEXIT_HLT	HLT instruction.

[AMD Public Use]

Table C-1. SVM Intercept Codes (continued)

Code	Name	Cause
79h	VMEXIT_INVLPG	INVLPG instructions.
7Ah	VMEXIT_INVLPGA	INVLPGA instruction.
7Bh	VMEXIT_IOIO	IN or OUT accessing protected port (the EXITINFO1 field provides more information).
7Ch	VMEXIT_MSR	RDMSR or WRMSR access to protected MSR.
7Dh	VMEXIT_TASK_SWITCH	Task switch.
7Eh	VMEXIT_FERR_FREEZE	FP legacy handling enabled, and processor is frozen in an x87/mmx instruction waiting for an interrupt.
7Fh	VMEXIT_SHUTDOWN	Shutdown
80h	VMEXIT_VMRUN	VMRUN instruction.
81h	VMEXIT_VMMCALL	VMMCALL instruction.
82h	VMEXIT_VMLOAD	VMLOAD instruction.
83h	VMEXIT_VMSAVE	VMSAVE instruction.
84h	VMEXIT_STGI	STGI instruction.
85h	VMEXIT_CLGI	CLGI instruction.
86h	VMEXIT_SKINIT	SKINIT instruction.
87h	VMEXIT_RDTSCP	RDTSCP instruction.
88h	VMEXIT_ICEBP	ICEBP instruction.
89h	VMEXIT_WBINVD	WBINVD or WBNOINVD instruction.
8Ah	VMEXIT_MONITOR	MONITOR or MONITORX instruction.
8Bh	VMEXIT_MWAIT	MWAIT or MWAITX instruction.
8Ch	VMEXIT_MWAIT_CONDITIONAL	MWAIT or MWAITX instruction, if monitor hardware is armed.
8Eh	VMEXIT_RDPRU	RDPRU instruction.
8Dh	VMEXIT_XSETBV	XSETBV instruction.
8Fh	VMEXIT_EFER_WRITE_TRAP	EFER MSR write trap.
90h	VMEXIT_CR0_WRITE_TRAP	CR0 write trap.
93h	VMEXIT_CR3_WRITE_TRAP	CR3 write trap.
94h	VMEXIT_CR4_WRITE_TRAP	CR4 write trap.
A0h	VMEXIT_INVLPGB	INVLPGB instruction.
A1h	VMEXIT_INVLPGB_ILLEGAL	Illegal INVLPGB instruction.
A2h	VMEXIT_INVPCID	INVPCID instruction.
A3h	VMEXIT_MCOMMIT	MCOMMIT instruction.
A4h	VMEXIT_TLBSYNC	TLBSYNC instruction.
A5h	VMEXIT_BUSLOCK	Bus lock while Bus Lock Threshold Counter value is 0.
A6h	VMEXIT_IDLE_HLT	HLT instruction if a virtual interrupt is not pending.

[AMD Public Use]

Table C-1. SVM Intercept Codes (continued)

Code	Name	Cause
400h	VMEXIT_NPF	Nested paging: host-level page fault occurred (EXITINFO1 contains fault error code; EXITINFO2 contains the guest physical address causing the fault).
401h	AVIC_INCOMPLETE_IPI	AVIC—Virtual IPI delivery not completed. See "AVIC IPI Delivery Not Completed" on page 605 for EXITINFO1–2 definitions.
402h	AVIC_NOACCEL	AVIC—Attempted access by guest to vAPIC register not handled by AVIC hardware. See "AVIC Access to Un-accelerated vAPIC register" on page 606 for EXITINFO1–2 definitions.
403h	VMEXIT_VMGEXIT	VMGEXIT instruction.
407h	VMEXIT_PML_FULL	Page Modification Logging Buffer overflow
F000_0000h	Unused	Reserved for Host.
–1	VMEXIT_INVALID	Invalid guest state in VMCB.
–2	VMEXIT_BUSY	BUSY bit was set in the VMSA (see "Interrupt Injection Restrictions" on page 640).
–3	VMEXIT_IDLE_REQUIRED	The sibling thread is not in an idle state (see "Side-Channel Protection" on page 641).
–4	VMEXIT_INVALID_PMC	Invalid PMC state (see “Performance Monitoring Counter Virtualization” on page 656).
–5	VMEXIT_ILLSIB	Illegal Sibling (See “Enhanced SMT Protection” on page 643).
–6	VMEXIT_ESMTP_TIMEOUT	ESMTP Timeout (See “Enhanced SMT Protection” on page 643).

[AMD Public Use]

Appendix D SMM Containerization

To minimally participate in SMM activity, the VMM can implement simple containerization. This appendix provides example pseudocode to perform this simple containerization. VMMs that do not trust SMM code should implement secure containerization, which requires further extension of the code provided here.

D.1 SMM Containerization Pseudocode

This code emulates transitions to and from SMM:

- The process of entering SMM mode as a result of a system management interrupt (SMI)
- The RSM instruction, which returns the processor from SMM.

A hypervisor that containerizes SMM must set the SMM intercept bit in all guest VMCBs. When the hypervisor encounters a #VMEXIT(SMI), it should then emulate SMM entry and execute the SMM handler by means of VMRUN with the RSM intercept bit set. When the RSM instruction is intercepted, the hypervisor should emulate the RSM instruction and then resume normal execution.

In this code, the hypervisor sets up the `smm_vmcb` from scratch and assigns it the supplied address space identifier (ASID).

This example code sets up a container VMCB for the SMM handler and copies appropriate state information into the SMM save area. After calling `emulate_smm()`, the hypervisor should repeatedly VMRUN the SMM handler VMCB until the hypervisor encounters a #VMEXIT(RSM). Finally, the hypervisor should call `emulate_rsm()`.

```
//emulate_smm( ):
// Inputs:
//   smm_vmcb:   the _virtual address_ of a VMCB that will be configured
//               as an SMM container
//   asid:       the asid to use for the SMM handler; the hypervisor should
//               ensure that no TLB entries for this ASID are present in the TLB
//   smm_regs:   an array of 64-bit values that will be filled with the
//               GPRs (except RSP and RAX) for the SMM handler
//   guest_vmcb:  the _virtual address_ of the VMCB of the guest
//                that was running when the intercepted SMI occurred
//   guest_regs: an array of 64-bit values that contains the GPRs (except RSP
//               and RAX) for the guest that was running when the intercepted
//               SMI occurred

void
emulate_smm(VMCB *smm_vmcb, uint32 asid, uint64 smm_regs[16],
            VMCB *guest_vmcb, uint64 guest_regs[16])
{
   setup_smm_container(*smm_vmcb, asid, smm_regs, *guest_vmcb, guest_regs)
```

[AMD Public Use]


```
   //Enter SMM mode:
   wrmsr(SMM_CTL_MSR, ENTER+DISMISS+SMI_CYCLE)
   setup_smm_save_state(*guest_vmcb, guest_regs)


   do { VMRUN(smm_vmcb) } until we see #VMEXIT(RSM).
       Shadow EFER reads and writes to protect the SVME bit.


   //Emulate RSM:
   copy_smm_save_to_guest_vmcb(guest_vmcb, guest_regs)
   //Leave SMM mode:
   wrmsr(SMM_CTL_MSR, EXIT+RSM_CYCLE)
}

void
setup_smm_container(VMCB &smm_vmcb, uint32 asid, uint64 smm_regs[16],
                    VMCB &g_vmcb, uint64 guest_regs[16])
{
   clear smm_vmcb to all zeros
   set intercepts in smm_vmcb:
      RSM
      VMRUN
      MSR
   smm_vmcb.msrpm = (physical address of msr protection map with
                     efer read and efer write set)
   // Note that the hypervisor should shadow the SVME bit of EFER and
   // return EFER.SVME=0 on reads of EFER.
   //
   // Note also that the IOPM (unused in this example code) and MSRPM for the SMM
   // container can be statically set up and reused on subsequent SMM entries,
   // and can be shared between multiple cores' SMM container VMCBs. Each core
   // must have a separate VMCB for the SMM container, but those cores' VMCBs may
   // be statically or dynamically allocated.

   smm_vmcb.asid = asid

   smmbase = rdmsr(smmbase_msr) // Note: smmbase is a 32 bit value

   Set up the smm handler's segment information: {Selector, Attrib, Limit, Base}

   smm_vmcb.CS = {(smmbase & 0x00ffff00) >> 4, 0x089B, 0xffff_ffff, smmbase}
   smm_vmcb.{ES, SS, DS, FS, GS} = {0x0000, 0x0893, 0xffff_ffff, 0x0000_0000}
   smm_vmcb.GDTR = {unused, unused, g_vmcb.gdtr_limit, g_vmcb.gdtr_base}
   smm_vmcb.LDTR = (copy all from g_vmcb.LDTR)
   smm_vmcb.IDTR = {unused, unused, g_vmcb.idtr_limit, g_vmcb.idtr_base}
   smm_vmcb.TR = (copy all from g_vmcb.TR)

   smm_vmcb.CPL = 0
   smm_vmcb.EFER = 0x1000 (SVME = 1)
```

[AMD Public Use]


```
   smm_vmcb.CR4 = 0
   smm_vmcb.DR7 = 0x0000_0400
   smm_vmcb.RFLAGS = 0x0000_0002
   smm_vmcb.RIP = 0x0000_8000


   Copy the following values from g_vmcb to smm_vmcb
   CR3
   DR6
   RSP
   RAX
   STAR
   LSTAR
   CSTAR
   SFMASK
   KERNELGSBASE
   SYSENTER_CS
   SYSENTER_ESP
   SYSENTER_EIP
   CR2
   CR0: clear bits 0, 2, 3, 31

   copy 14 guest GPRs from guest_regs (all except RAX, RSP) to smm_regs
}


void
setup_smm_save_state(struct VMCB &g_vmcb, uint64 guest_regs[16])
{
   smmbase = rdmsr(smmbase_msr) // Note: smmbase is a 32 bit value
   smmsave_physical_addr = smmbase + 0xfe00
   // smmsave is the physical address of the SMM save area;
   // the hypervisor will need to map this into its virtual memory space.
   smmsave = virtual_to_physical_map(smmsave_physical_addr)

   Copy the following values from g_vmcb to smmsave:
   all defined portions of ES, CS, SS, DS, FS, GS, GDTR, LDTR, IDTR, TR
        (all bytes of each 16-byte segment save area)
   CPL
   EFER
   CR4
   CR3
   CR0
   DR7
   DR6
   RFLAGS
   RIP
   RSP
   RAX

   copy 14 guest GPRs (other than RAX and RSP) from guest_regs
```

[AMD Public Use]


```
                                                to GPR entries in smmsave

iorestart_dword[31:0] = g_vmcb.exitinfo1[63:32]
if ((iorestart_dword & IO_RESTART_VALID) != 0)
{
   Copy iorestart_dword to smmsave.iorestart_dword,
         masking out address size bits
   Copy g_vmcb.exitinfo2 to smmsave.iorestart_rip

   uint64 *guest_indexreg // Point to the index register in the guest context
                          // that is changed by the string instruction...
   uint64 *smm_indexreg   // ...similarly, for the smm save area
   if (iorestart_dword & IO_RESTART_IN != 0) {
      guest_indexreg = &guest_regs[RDI] // type=IN, indexreg=RDI
      smm_indexreg = &smmsave.iorestart_rdi
      smmsave.iorestart_rsi = guest_regs[RSI]
   } else {
      guest_indexreg = &guest_regs[RSI] // type=OUT, indexreg=RSI
      smm_indexreg = &smmsave.iorestart_rsi
      smmsave.iorestart_rdi = guest_regs[RDI]
   }

   // Reconstruct the IORestart values
   if (iorstart_dword & IO_RESTART_STR != 0)
   {
       uint64 mask
       uint64 ecxfix

       operand_size = (iorestart_dword >> 4) & 0x7)

       address_size = (iorestart_dword >> 7) & 0x7)
       if (address_size == 0) // Some SVM implementations do not provide
                              // these bits; we must decode on those CPUs
          address_size = decode_io_size(guest_vmcb)

       mask = (1<<address_size) - 1
       if (g->RFLAGS D-bit is set)
          operand_size = -operand_size

       if (iorestart_dword & IO_RESTART_RIP != 0)
          ecxfix = 1
       else ecxfix = 0

       *smm_indexreg = *guest_indexreg & ~mask  |  (*guest_indexreg -
                                                    operand_size) & mask
       smmsave.iorestart_rcx = mask & (guest_regs[RCX] + ecxfix)
   } else { // not string
       *smm_indexreg = *guest_indexreg
       smmsave.iorestart_rcx = guest_regs[RCX]
   }
} else { // iorestart isn't valid: Put the same values into the restart values.
```

[AMD Public Use]


```
      smmsave.iorestart_dword = 0
      smmsave.iorestart_rip = g.rip
      smmsave.iorestart_rcx = guest_regs[RCX]
      smmsave.iorestart_rsi = guest_regs[RSI]
      smmsave.iorestart_rdi= guest_regs[RDI]
   }

   smmsave.iorestart = 0
   smmsave.hltrestart = 0
   smmsave.nmimask = 0
   smmsave.smm_revision = 0x30064
   smmsave.smm_base = smmbase
}

void
copy_smm_save_to_guest_vmcb(struct VMCB &g_vmcb, uint64 guest_regs[16])
{
   smmbase = rdmsr(smmbase_msr) // Note: smmbase is a 32 bit value
   smmsave_physical_addr = smmbase + 0xfe00
   // smmsave is the physical address of the SMM save area;
   // the hypervisor will need to map this into its virtual memory space.
   smmsave = virtual_to_physical_map(smmsave_physical_addr)

   Copy the following values from smmsave to g_vmcb
   all defined portions of ES, CS, SS, DS, FS, GS, GDTR, LDTR, IDTR, TR
   CPL
   EFER
   CR4
   CR3
   CR0
   DR7
   DR6
   RFLAGS
   RSP
   RAX

   Copy the other 14 GPRs from smmsave into guest_regs.

   If smmsave.iorestart is set, copy RDI,
   RSI, RCX from the smmsave.iorestart_{RDI, RSI, RCX} fields
                instead of the regular {RDI, RSI, RCX} fields.

   if (smmsave.iorestart is zero and smmsave.iorestart_dword is valid)
   {
      modify g_vmcb.DR6:
         clear g_vmcb.DR6[3:0] and copy BRP bits from
                smmsave.iorestart_dword[15:12] into g_vmcb.DR6[3:0]
         // this preserves AMD's behavior that dr6[3:0] is not sticky,
         // but the other bits are sticky
         g_vmcb.DR6.BS |= smmsave.iorestart_dword.TF
      if any bit of smmsave.iorestart_dword.{BRP[3:0], TF} is nonzero,
```

[AMD Public Use]


```
         we have a pending #DB exception,
         so set up a #DB event injection for the guest.
   }


   if (smmsave.iorestart is set) {
      set g_vmcb.RIP = smmsave.iorestart_rip
   } else if (smmsave.hltrestart is set) {
      // (In the event that the guest is allowed to execute HLT and
      // the SMM code wants to use the auto-halt restart function,
      // we need to re-execute the HLT instruction in the guest context.
      // Even if the HLT has prefixes (all of which would be ignored),
      // we know that RIP-1 is the F4 opcode itself.)

      Subtract 1 from the guest RIP under a mask that masks out bits
             above the current default address size:

      mask = (1 << current_address_size) - 1
      g_vmcb.RIP = mask & (g_vmcb.RIP-1)
   } else {
      set g_vmcb.RIP = smmsave.RIP
   }
// Note that it is undefined to have both iorestart and hltrestart set at
// the same time.

// Perform the RSM consistency checks listed in Volume 3 of the
// AMD64 Architecture Programmer's manual, except the check that
// disallows CR0.PG = 1 when CR0.PE = 0.  Note that the expected
// value for the SMM revision field is 0x0003_0064.  If any of the
// checks fail, the native RSM instruction would have caused a
// processor shutdown (which commonly results in a reboot
// triggered by the chipset). The hypervisor may wish to destroy
// the guest or cause its own shutdown.
}
```

D.1.1 Converting Simple Containerization into Secure Containerization

To convert this simple containerization example into secure containerization, the hypervisor must limit the SMM handler's access to I/O ports, MSRs, and memory. Based on security policy decisions, the hypervisor should set appropriate bits in the I/O Protection Map and the MSR Protection map and emulate any accesses the SMM handler makes to those protected resources. The hypervisor should run the SMM handler in paged real mode, with a page table that appropriately limits memory accessible to SMM code. Additionally, the hypervisor may wish to conceal some or all of the contents of a guest's general purpose and floating-point registers from the SMM handler.

[AMD Public Use]

Appendix E OS-Visible Workarounds

Operating system software may provide a workaround for a hardware erratum. These operating system-visible workarounds are provisional and should be removed or disabled when the erratum is corrected in a subsequent hardware release.

The OS-Visible Workaround (OSVW) architecture provides a means by which operating system software may determining the status of a known erratum for the hardware on which the software is running. Support for the OSVW mechanism is indicated by CPUID Fn8000_0001_ECX[OSVW] = 1.

See Section 3.3, "Processor Feature Identification," on page 73 for information on using the CPUID instruction.

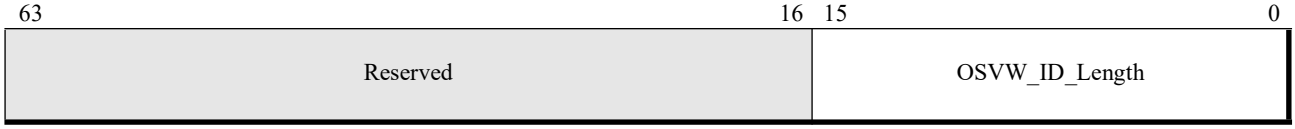
Each hardware erratum is assigned a unique OSVW ID number. OSVW ID numbers start at 0 and are assigned sequentially up to the most recently identified erratum which is assigned the number *m*–1. The OSVW mechanism encodes the status of each erratum for a given hardware system in a bit vector of length *m* accessed through OSVM MSRs 1–N. The state of bit *n* of the vector indicates the status of the erratum with the OSVW ID number *n*. The OSVW ID number for the erratum and the bit position within the erratum status bit vector, once assigned, are global across all AMD processors; the OSVW ID and bit position will not be re-used.

The OSVW MSRs are defined as follows:

- OSVW MSR0 contains the *OSVW_ID_Length* field, used to indicate the total number of valid OSVW ID bits (*m*). The format of this MSR is shown in Figure E-1 below.
- OSVW MSR 1 and following contain the erratum status bit vector of length *m*. Each bit *n* of this vector encodes the status of erratum *n* (OSVW ID = *n*). The format of these MSRs is shown in Figure E-2 on page 811.

The bank of OSVW MSRs is located at address C001_0140h, starting with OSVW MSR0.

The OSVW MSRs should be treated as read-only registers for the OS. The OS should never write into these registers. Hardware allows platform firmware writes to these registers.



Bits	Mnemonic	Description	Access type[1]
63:16	Reserved		
15:0	OSVW_ID_Length	Total length of the status vector OSVW_E in bits.	R/W

Note 1: MSR should be treated as read-only by operating system software.

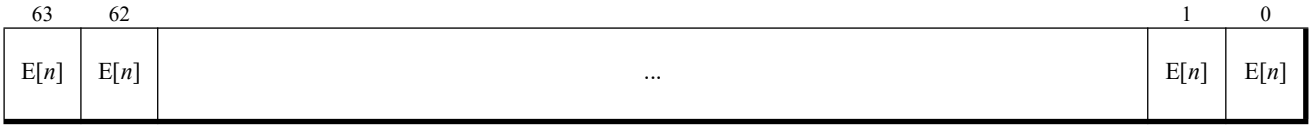
Figure E-1. OSVW MSR0: OSVW_ID_Length

[AMD Public Use]

OSVW_ID_Length—Bits [15:0]. The number of valid bits in the OSVW erratum status vector OSVW_E. If a specific erratum has an OSVW ID that is greater than or equal to the *OSVW_ID_Length*, the erratum is unknown to the latest release. Otherwise, the erratum status bit in the appropriate OSVW MSR can be checked to see if a workaround is required.

The erratum status bit vector (OSVW_E) is accessed through OSVW MSR 1 and following. For MSR *N*, the 64-bit MSR holds erratum status bits (*N*–1)*64+63:(*N*–1)*64. To access the erratum status for OSVW ID number *n* (E[*n*] in the diagram), read MSR *N*, where *N* = *n*/64 + 1, and test bit *i*, where *i* = *n* modulo 64.

Figure E-2 below gives the format of the OSVW MSRs 1–N.



Bit	Mnemonic	Description	Access type[1]
i	OSVW_E[*n*]	OS-visible workaround status bit *n*	R/W

Note 1: MSR should be treated as read-only by operating system software.

Figure E-2. OSVW MSRs 1–N: OSVW Erratum Status Registers

OS-Visible Workaround Erratum Status (OSVW_E[n])—Bits 63:0. Each bit indicates whether platform hardware is affected by OS-visible erratum *n* and whether the OS needs to apply a workaround.

For the status bit:

1 = Hardware contains the erratum; an OS software workaround is required.

0 = Hardware has corrected the erratum; an OS software workaround is unnecessary. If one is installed, it must be disabled.

The location of an OSVW ID status bit within a bank of OSVW MSRs is determined as follows:

- MSR address = OSVW_MSR0 + 1 + floor (OSVW_ID /64)
- Bit offset in MSR = OSVW_ID modulo 64

If a specific erratum has an OSVW_ID that is greater than or equal to the OSVW_ID_LENGTH, hardware does not know about the erratum and the processor model must be used to determine whether the workaround must be applied.

OSVW MSR bits beyond the end of the OSVW_E bit vector are reserved.

[AMD Public Use]

E.1 Erratum Process Overview

Following is an overview of the AMD erratum process:

1. When an OS-visible erratum is discovered, AMD assigns a unique OSVW ID to the erratum and publishes to OS vendors the starting range of affected processor models and suggested workarounds.
2. AMD works with platform firmware vendors and OEMs in parallel to develop a firmware update to add the new erratum status bit to the OSVW_E erratum status bit vector for affected silicon revisions to report the new OSVW ID as requiring a workaround. The OSVW_ID_Length field in OSVW MSR0 is incremented by one.
3. OS vendors schedule the workaround into their release schedules and eventually release it.
4. The OS detection logic for the workaround first checks whether the processor OSVW MSRs 1–N record the erratum by comparing the OSVW ID of the erratum with the OSVW_ID_Length field in OSVW MSR0.
5. If the erratum OSVW ID is greater than or equal to the OSVW_ID_Length, the current firmware does not know about this erratum. In this case, the OS software compares the processor model ID with the starting model ID that AMD supplied with the erratum to determine if the workaround should be applied.
6. If the erratum OSVW ID is less than the OSVW_ID_Length, the firmare is aware of the erratum. In this case, the OS uses the state of the associated OSVW_E status bit to conditionally apply the workaround. If the associated status bit is set, the workaround is applied.
7. Once AMD fixes the erratum in a future release, updated firmware ensures that the OSVW_E status bit associated with the erratum is cleared. When OS workaround detection logic runs on the new hardware, it will see that the bit corresponding to the OSVW ID is cleared and not apply the OS workaround for that erratum.

[AMD Public Use]

Index

Symbols

#AC 260
#BP 251
#BR 252
#D 259, 262
#DB 250
#DE 250
#DF 253
#GP 257
#I 259, 262
#IA 259
#IS 259
#MC 261
#MF 259
#NM 253
#NP 256
#O 259, 262
#OF 252
#P 259, 262
#PF 258
#SS 256
#SX 588
#TS 255
#U 259, 262
#UD 252
#VMEXIT 524, 525
#XF 261, 262
#Z 259, 262

Numerics

13-7LastBranchStackFromIp MSR 402
16-bit mode liii
1-Gbyte page 151
1-Gbyte page translation 151
  long mode 4-level paging 152
  long mode 5-level paging 153
2-Mbyte page translation—long mode 5-level paging 147
32-bit mode liii
4-Kbyte page translation 143
5-Level paging 129
5-Level Paging Enable (LA57) 51
64-bit media instructions
  causing #MF exception 349
  initializing 507, 508
  MMX registers 348
  saving state 351
64-bit mode ..... liii, 13

A

A bit 93, 95, 157
A20 Masking 579
abort 245
AC bit 55
access checking 565
accessed (A)
  code segment 93
  data segment 95
  page-translation tables 157
address space identifier (ASID) 551
address-breakpoint registers (DR0-DR3) 394
addressing
  RIP-relative lviii
address-size prefix 31
ADDRV bit 312
Advanced Programmable Interrupt Controller (APIC) 659
AIBRSE bit 59
alignment check (rFLAGS.AC) 55, 260
alignment mask (CR0.AM) 44, 260
alignment-check exception (#AC) 45, 55, 260
AM bit 44
AP startup sequence 587
APIC 659
  base address 662
  enable 662
  error interrupts 671
  internal error 660
  registers 662
  timer interrupt 668
  version register 664
APIC.TPR 556
APIC.TPR virtualization 522
Application Processors (APs) 586
Arbitration 678
architecture differences 23
ARPL instruction 183
ASID 551
attributes 89
available to software (AVL)
  descriptor 92
  page-translation tables 158
AVL bit 92, 158

B

base address 86, 88, 91, 134, 155, 156
benign exception 253

[AMD Public Use]

BIST 502
bootstrap CPU core (BSC) 662
bootstrap processor (BSP) 505, 586
BOUND instruction 252
bound-range exception (#BR) 252
BR_FROM 563
BR_TO 563
branches 32
Breakpoint
  on bus lock 393
breakpoint
  determining cause 406
  on address match 393, 408
  on any instruction 393
  on I/O 408
  on instruction 407
  on task switch 393, 410
  setting address 404
  specifying address-match length 405
breakpoint exception (#BP) 251
breakpoints 404
built-in self test (BIST) 502

C

C bit 93
cache
  control mechanisms 209
  control precedence 211
  enabling 506
  index 208
  invalidate 213
  line 188
  offset 208
  organization 206
  self-modifying code 208
  set 207
  tag 208
  way 207
  writeback and invalidate 212
cache disable (CD) bit 45, 210
cache disable (CD), memory type 199
cache-coherency protocol
  losing coherency 196
CALL
  See call gate and control transfer.
call gate 97, 115
  count field 99
  count field, long mode 105
  descriptor, long mode 33
  jump through 117
  parameters 119
  privilege checks 116
  stack switch 119
  stack switch, long mode 33, 120
canonical address form 4, 142
CD bit 45, 210
CD memory type 199
CLFLUSH 212, 564
CLGI 537, 553
CLI instruction 180
clock multiplier 503
CLTS 180, 536
code segment 26, 81, 93
  64-bit mode 82
  accessed (A) 93
  conforming (C) 93
  default-operand size (D) 94
  ignored fields in 64-bit mode 99
  long bit (L) 26, 100
  long mode 99
  readable (R) 93
  type field 93
coherency, across encryption domains 618
coherency, cache 188
Combining Memory Types and MTRRs 577
commit liii
commit, instruction results 189
compatibility mode ..... liii, 13
config space accesses 565
conforming (C), code segment 93
consistency checks, long mode 513
containerized SMM code 561
contributory exception 254
control registers 29, 41
control transfer 111
  See also call gate and interrupt.
  call gate 115
  direct 111
  far, conforming code segment 113
  far, nonconforming code segment 111
  interrupt to higher privilege 275
  interrupt to same privilege 274
  parameters 119
  stack switch 119
control-transfer recording MSRs 401
coprocessor-segment-overrun exception 254
core performance counter global control register 418
core performance counter status registers 419
count field 105
CPL 107, 525
  definition 107
  in call gate protection 116
  in data segment protection 108, 322
  in interrupt to higher privilege 276
  in protecting conforming CS 113
  in protecting nonconforming CS 112
  in stack segment protection 110

[AMD Public Use]

privileged instructions ... 172
SYSCALL, SYSRET assumptions ... 176
CPU watchdog timer register ... 306
CPUID ... 56, 73, 179, 536
nested paging ... 579
CR0 ... 42, 524
alignment mask (AM) ... 44, 260
cache disable (CD) ... 45, 210
emulate coprocessor (EM) ... 43
emulate coprocessor (EM) bit ... 346
extension type (ET) ... 44
monitor coprocessor (MP) ... 43
not write-through (NW) ... 45, 210
numeric error (NE) ... 44, 260
paging enable (PG) ... 45, 131
protection enable (PE) ... 43, 76, 84
task switched (TS) ... 44, 180
write protect (WP) ... 44
CR1 ... 52
CR2 ... 45, 258, 524
CR3 ... 25, 45, 46, 133, 142, 376, 524
non-PAE paging ... 133
PAE paging ... 46, 133
PAE paging, long mode ... 142
page-level cache disable (PCD) ... 134, 143
page-level write-through (PWT) ... 134, 143
table-base address ... 134, 142
CR4 ... 47, 524
debugging extensions (DE) ... 49
machine-check enable (MCE) ... 50, 261
OS #XF support (OSXMMEXCPT) ... 261, 346, 347
OS FXSAVE/FXRSTOR support (OSFXSR) ... 171, 346
page-global enable (PGE) ... 50, 160
page-size extensions (PSE) ... 49, 132, 136
performance counter enable (PCE) ... 50, 180, 415
physical-address extensions (PAE) ... 49, 52, 131, 141
protected-mode virtual interrupts (PVI) ... 49
time-stamp disable (TSD) ... 49, 180, 425
virtual-8086 mode extensions (VME) ... 48, 290
CR5–CR7 ... 52
CR8 ... 52, 269
CR9–CR15 ... 52
CS register ... 81, 524
selector ... 524
CSTAR register ... 177, 760, 767

D

D bit ... 94, 100, 157
D/B bit ... 92, 96
Data Limit Checks ... 125
Data limit checks ... 125
data prefetch, cache ... 212
data segment ... 27, 82, 94
64-bit mode ... 82
accessed (A) ... 95
default operand size (D) ... 96
expand down (E) ... 95
FS and GS ... 27, 82
ignored fields in 64-bit mode ... 100
long mode ... 100
privilege checks ... 108
type field ... 95
writable (W) ... 95
DAZ bit ... 365
DBGCTL ... 563
DE bit ... 49
DE exception ... 259, 262
debug ... 21, 586
See breakpoint and single-step.
debug exception (#DB) ... 250, 406
debug registers ... 29, 393
address-breakpoint registers (DR0-DR3) ... 394
control-transfer recording MSRs ... 401
debug-control MSR (DebugCtl) ... 399
debug-control register (DR7) ... 396
debug-extension-control (DebugExtnCtl) ... 401
last branch stack registers ... 402
reserved (DR4, DR5) ... 394
debug-control MSR (DebugCtl) ... 399
debug-control register (DR7) ... 396
DebugCtl register ... 758, 772
debug-extension-control register
last-branch record stack enable ... 402
debug-extension-control register (DebugExtnCtl) ... 401
DebugExtnCtl register ... 401
debugging extensions (CR4.DE) ... 49
DEC instruction ... 34
default operand size
B bit, stack segment ... 96
D bit, code segment ... 94
D bit, data segment ... 96, 123
D/B bit, descriptor ... 92
with expand down ... 124
denormalized-operand exception (DE) ... 259, 262
denormals-are-zeros (DAZ) mode ... 365
descriptor ... 77, 90
available to software (AVL) ... 92
code segment ... 26
data segment ... 27
default operand size (D/B) ... 92
DPL ... 92, 108, 381
gate ... 27
granularity (G) ... 92
long mode ... 99
present (P) ... 92, 381
S field ... 92, 381
segment base ... 91
segment limit ... 91

[AMD Public Use]

system segment ... 27
TSS ... 371
type field ... 92, 381
descriptor table ... 77, 84
global-descriptor table (GDT) ... 79
interrupt-descriptor table (IDT) ... 37
local-descriptor table (LDT) ... 79
descriptor-table registers ... 26, 78
64-bit mode ... 105
GDTR ... 85
IDTR ... 90
LDTR ... 87
DEV base address registers ... 569
DEV caching ... 564
DEV capability block ... 566
DEV register access ... 567
DEV_BASE_HI/LO registers ... 567
DEV_CAP register ... 568
DEV_CR register ... 568
DEV_DATA ... 566
DEV_HDR ... 566
DEV_MAP Registers ... 569
DEV_OP ... 566, 567
DEVBASE registers ... 564
device exclusion vector (DEV) ... 564
device ID ... 564
device-not-available exception (#NM) ... 43, 253
differences (architectural) ... 23
direct referencing ... liii
dirty (D), page-translation tables ... 157
displacement ... 31
displacements ... liv
divide-by-zero-error exception (#DE) ... 250
double quadword ... liv
double-fault exception (#DF) ... 253
doubleword ... liv
DP field ... 365
DPL ... 108
data segment, 64-bit mode ... 101
definition ... 108
in call gate protection ... 116
in data segment protection ... 108, 322
in interrupt stack switch ... 275
in interrupt to higher privilege ... 276
in protecting conforming CS ... 113
in protecting nonconforming CS ... 112
in stack segment protection ... 110
in stack switching ... 119
DPL field ... 92, 381
DR0-DR3 registers ... 394
DR4, DR5 registers ... 394
DR6 register ... 395, 525
DR7 register ... 525
DS field ... 365
DS register ... 82, 524
DS.SEL ... 524

E

E bit ... 95
eAX–eSP register ... lx
EFER register ... 29, 56, 68, 70, 524, 760, 767
fast FXSAVE/FXRSTOR (FFXSR) ... 58
long mode active (LMA) ... 58, 511
long mode enable (LME) ... 57, 511
no-execute enable (NXE) ... 58
system-call extension (SCE) ... 57
EFER.SVME ... 522
effective address ... 2, 25
effective address size ... liv
effective memory type ... 227
effective operand size ... liv
EFLAGS
See rFLAGS.
eFLAGS register ... lx
EIP
See rIP.eIP register ... lx
EIPV bit ... 305
EM bit ... 43, 507
emulate coprocessor (CR0.EM) ... 43
EN bit ... 312
enabling SVM ... 522
endian byte-ordering ... lxii
End-of-Interrupt Register (EOI) ... 684
environment ... 352
error code
page fault ... 265
selector ... 264
ES register ... 82, 524
ES.SEL ... 524
ESP
See rSP.
ET bit ... 44
event handler, definition ... 244
event injection ... 554
EVENTINJ ... 554
exception handler, definition ... 244
exception intercept
#AC ... 543
#BP ... 542
#BR ... 542
#DB ... 542
#DE ... 541
#DF ... 542
#GP ... 543
#MC ... 544
#MF ... 543

[AMD Public Use]

#NM ... 542
#NP ... 543
#OF ... 542
#PF ... 543
#SS ... 543
#TS ... 543
#UD ... 542
#XF ... 544
vector 2 ... 542
vector 9 ... 543
Exception Intercepts ... 541
exceptions ... liv
abort ... 245
benign ... 253
contributory ... 254
definition of ... 244
definition of vector ... 247
differences in long mode ... 37
error code, page fault ... 265
error code, selector ... 264
fault ... 245
floating-point priorities ... 267
imprecise ... 245
maskable SSE floating point ... 246
maskable x87 floating point ... 246
masking during stack switches ... 246
precise ... 244
priorities ... 266
trap ... 245
while in SMM ... 337
EXITINFO1 ... 539
expand down (E)
data segment ... 95
stack segment ... 95, 124
extended family field ... 506
Extended Interrupts ... 671
extended model field ... 506
extended save area ... 359
extended state management ... 357
extended topology ... 215
extensible state management ... 357
extension type (CR0.ET) ... 44

F

family field ... 506
far control transfer ... 111
far return ... 33, 122
fast FXSAVE/FXRSTOR ... 58
fault ... 245
FCW register ... 350, 352, 364
feature identification ... 73
FENCE ... 191
FFXSR bit ... 58
fill, cache-line ... 188
first instruction ... 505
flat segmentation ... 6, 9, 77
FLDENV, FSTENV instructions ... 356
floating-point exception pending (#MF) ... 259
caused by 64-bit media instructions ... 349
floating-point exception priorities ... 267
flush ... liv
FOP register ... 364
FPR registers ... 350, 352
FS and GS ... 27, 82
FS register ... 82
FS.Base register ... 760, 767
FSAVE, FRSTOR instructions ... 352
FSW register ... 349, 350, 352, 364
FTW register ... 349, 350, 352, 364
FXSAVE, FXRSTOR instructions ... 36, 50, 357
32-bit memory image ... 364
64-bit memory image ... 363
x87 tag word format ... 365

G

G bit ... 92, 158
gate descriptors ... 27
call gate ... 97
DPL ... 108
ignored fields in long mode ... 103
illegal types in long mode ... 103
interrupt gate ... 97
long mode ... 103, 105
redefined types in long mode ... 103
target-segment offset ... 98
target-segment selector ... 98
task gate ... 97
trap gate ... 97
GDT ... 84
GDTR ... 85, 524, 536
general detect fault ... 250, 409
general-protection exception (#GP) ... 257
general-purpose registers (GPRs) ... 28
GIF ... 553
global descriptor table (GDT) ... 79, 84
base address, 64-bit mode ... 86
first entry ... 84
limit check, long mode ... 86
global descriptor-table register (GDTR) ... 85
base address ... 86
limit ... 86
loading ... 181
storing ... 182
global interrupt flag (GIF) ... 553
global page (G), page-translation tables ... 158
global pages ... 50, 160
granularity (G), descriptor ... 92, 123

[AMD Public Use]

GS register 82
GS.Base register 760, 767
guest mode 521
Guest page tables (gPT) 571

H

halt 183
Hardware errata 810
HLT 183, 537
host 520
HostMultiKeyMemEncrModeEn bit 63
HWCR 72
HWCR register
  CpbDis bit 72
  CpuidUserDis 72
  IRPerfEn bit 72
  SmmLock bit 72
hypervisor 520

I

I/O interrupts 660
I/O Permissions Map 538
I/O privilege level field (rFLAGS.IOPL) 54
I/O space accesses 565
I/O, memory-mapped 234
I/O-permission bitmap
  in 32-bit TSS 376
  in 64-bit TSS 379
I/O-permission bitmap (IOPB) 377
IBS filtering 444
ICEBP 538
ID bit 56
IDT 89
IDTR 90, 524, 536
IE exception 259, 262
IF bit 54, 294
IGN lv
illegal state 524
immediate operand 31
imprecise exceptions and interrupts 245
IN/OUT 539
INC instruction 34
indirect lv
inexact-result exception 259, 262
INIT 558
initialization 502
initialization (INIT)
  processor state 503
In-Service Register 680
instruction-based sampling virtualization 654
instructions (system-management) 172
INT3 instruction 251, 410
integer bit 366
intercept 522
  Ferr_Freeze 546, 547
  shutdown 547
  task switch 546
Intercept Exit Codes 801
Interprocessor interrupt (IPI) 586, 673
  INIT 586
  Startup 586
Interrupt Control 660
interrupt descriptor table (IDT) 89
  limit check, long mode 90
interrupt descriptor-table register (IDTR) 90
  loading 181
  storing 182
interrupt flag (rFLAGS.IF) 54, 180
interrupt gate 97, 283
  IST field 104
interrupt handler, definition 244
interrupt intercept 544
  INIT 546
  INTR 544
  NMI 544
  SMI 545
  virtual 546
interrupt redirection 281, 292
Interrupt Request Register 679
interrupt shadow stack table 289
interrupt shadows 557
INTERRUPT_SHADOW 524
interrupt-descriptor table (IDT)
  index 244, 273, 283
  protected mode 272
  real-address mode 270
interrupt-redirection bitmap 377
interrupts
  definition of external 244
  definition of software 244
  definition of vector 247
  differences in long mode 37
  external 263
  external maskable 246
  external nonmaskable 246
  external-interrupt priorities 269
  imprecise 245
  long mode summary 283
  precise 244
  priorities 266
  returning from 64-bit mode 289
  returns 279
  software 263
  stack alignment, long mode 286
  stack pointer push, long mode 284
  stack switch, long mode 37, 286

[AMD Public Use]

to higher privilege 275
to same privilege 274
while in SMM 337
interrupt-stack table (IST) 37, 104, 287
in 64-bit TSS 379
interrupt-vector table 270
INTn 537
INTn instruction 263, 410
INTO instruction 252
INTWB bit 59
invalid arithmetic-operand exception 259
invalidate page 552
invalid-opcode exception (#UD) 35, 252
invalid-operation exception (IE) 259, 262
invalid-TSS exception (#TS) 255
INVD 184, 185, 213, 537
INVLPG 537
INVLPG instruction 160, 184, 185
INVLPGA 537, 552
IOPB 376, 377
IOPL 539
IOPL field 54, 280
IOPL-sensitive instruction 290
IOPM 538
IOPM_BASE_PA 538
IORRBasen registers 236, 763, 769
IORRMaskn registers 237, 769
IORRs, variable-range 236
IOSPE 565
IRET 537
less privilege 279, 280
long mode 38, 289
same privilege 279
IST field 104

J

J bit 367
jump
See call gate and control transfer.

K

KernelGSbase register 179, 760, 767

L

L bit 100
L1 data cache 188
L1 instruction cache 188
L2 cache 188
L2I_PerfEvtSel registers 422
L3 cache range reservation 215
registers 215
L3 range reservation
operation 218
programming considerations 218
register programming 217
used as range lock 217
L3 range reserve
maximum address 216
way mask 217
LAR instruction 182
last branch record virtualization 562
last branch stack registers 402
LastBranchFromIP 562
LastBranchFromIP register 758, 772
LastBranchStackFromIp register 402
LastBranchStackSelect register 404
LastBranchToIP 562
LastBranchToIP register 758, 772
LASTEXCPFROM 563
LASTEXCPTO 563
LastIntFromIP 562
LastIntFromIP register 758, 772
LastIntToIP 562
LastIntToIP register 758, 772
LastStackToIp register 402
LDT 86
selector field 376
LDTR 87, 536
Legacy Interrupts 660
legacy mode ..... lv, 14, 23
legacy PAE mode 579
legacy x86 lv
LFENCE 191
LFENCE instruction 212
LGDT 181, 536
LIDT 181, 536
limit 86, 89, 91, 371
linear address 3
Link field 376
LINT0 670
LINT1 670
LLDT 182, 536
LMA bit 58
LME bit 57
LMSLE 58
LMSW 179, 536
load ordering 212
Local APIC 661
ID 664
interrupt masking 558, 684
local descriptor table (LDT) 79, 86
base address, 64-bit mode 89
limit check, long mode 89
local descriptor-table register (LDTR) 87

[AMD Public Use]

attributes 89
base address 88
hidden portion 87
LDT selector 88
limit 89
loading 182
storing 182
Local Interrupts 666
locality 159
logging
unauthorized access 570
logical address 2
long attribute (L)
code segment 100
effect on D bit 100
long mode lvi, 12, 23
2-Mbyte physical-page translation 147
activating 512
consistency checks 513
CR3 register 142
differences from legacy mode 39
enabling 511
enabling versus activating 511
GDT requirements 510
IDT requirements 510
leaving 514
page translation-table requirements 511
relocating descriptor tables 514
relocating page tables 514
TSS requirements 511
use of CS.L and CS.D 512
virtual to physical address translation 130
long mode active (EFER.LMA) 58, 511
long mode enable (EFER.LME) 57, 511
long-mode
4-Kbyte translation-table formats 145
LSB lvi
lsb lvi
LSTAR register 177, 760, 767
LTR 182, 536

M

M bit 367
machine check
error codes 313
error-reporting address register (MCi_ADDR) 313
error-reporting control register (MCi_CTL) 310
error-reporting miscellaneous register (MCi_MISC) 313
error-reporting register banks 308, 316
error-reporting status register (MCi_STATUS) 310
global-capabilities register (MCG_CAP) 303
global-control register (MCG_CTL) 305
global-status register (MCG_STATUS) 304
initialization 321
machine check registers 303
machine-check enable (CR4.MCE) 50, 261
machine-check exception (#MC) 261
mask lvi
masking
definition of interrupt 244
MBZ lvi
MCA error code field 311
MCE bit 50
MCG_CAP register 303, 758, 770
MCG_CTL register 305, 758, 770
MCG_CTL Register Present bit 304
MCG_STATUS register 304, 758, 770
MCi Bank Count field 304
MCi_ADDR registers 313, 759, 770
MCi_CTL registers 310, 759
MCi_MISC registers 760, 770
MCi_STATUS registers 310, 759, 770
MCIP bit 305
MCOMMIT bit 59
Media Extension Control and Status Register (MXCSR) 347
memory 186
memory addressing
canonical address form 4
effective address 2
linear address 3
logical address 2
near pointers 2
physical address 3
real address 10
RIP-relative address 31
segment offset 2
virtual address 3
memory consistency 577
memory management 5
memory serialization 212
memory system 186
memory type 198
determining effective 227
uncacheable (UC) 198
write-combining (WC) 199
write-combining plus (WC+) 199
write-protect (WP) 199
memory-access ordering
description 189
read ordering 189
write ordering 190
memory-mapped I/O
directing reads and writes to 234, 238
memory-type range register (MTRR) 29
combined with PAT 232

[AMD Public Use]

effect of paging cache controls 227
effects with large page sizes 228
fixed range 221
identifying features 226
initial value 506
IORRBase 236
IORRMask 237
MTRRcap 226
MTRRdefType 226
MTRRfix16K 222
MTRRfix4K 222
MTRRfix64K 222
MTRRphysBase 223
MTRRphysMask 224
overlapping ranges 228
type field, default 220
type field, extended 234
variable range 223
variable range size and alignment 225
MFENCE instruction 212
MISCV bit 312
MMX registers 348
model field 506
model-specific error code field 312
model-specific registers (MSRs) 29, 60, 180
control-transfer recording 401
debug extensions 64
debug-control MSR (DebugCtl) 399
initializing 508
machine check 65, 66, 67, 303
memory typing 64, 220
PAT 229
performance monitoring 65, 414
SYSCFG 61
system linkage 63, 177
time-stamp counter 65, 425
TOP_MEM 238
TOP_MEM2 238
modes 11
64-bit 13
compatibility ..... liii, 13
legacy ..... lv, 14
long ..... lvi, 12
protected ..... lvii, 14
real ..... lvii, 4, 14
virtual-8086 ..... lix, 14
moffset lvi
monitor coprocessor (CR0.MP) 43
MOV CRn instruction 179
MOV DRn instruction 180
MOV TO CR0 536
MOV TO/FROM CR0 536
MOV TO/FROM CRn 536, 712
MOV TO/FROM DRn 536, 712
MOVSXD instruction 34
MP bit 43, 507
MSB lvi
msb lvi
MSR lxi
MSR permissions map (MSRPM) 540
MSR_PROT 541
MSRs 60
MTRRcap register 226, 757, 768
MTRRdefType register 226, 759, 769
MTRRfix16K_n registers 222
MTRRfix4K_n registers 222
MTRRfix64K_n registers 222, 759, 769
MtrrFixDramEn bit 62, 235
MtrrFixDramModEn bit 62, 235
MTRRphysBasen registers 223, 758, 768
MTRRphysMaskn registers 224, 769
MTRRs 220, 577
MtrrTom2En bit 62, 240
MtrrVarDramEn bit 62, 240
multiprocessor issues 565
MXCSR register 347
field 365
MXCSR_MASK field 365

N

NB_PerfEvtSel registers 421
NE bit 44, 507
near branch
operand size, 64-bit mode 32
near control transfer 111
near pointers 2
near return 122
Nested page tables (hPT) 571
nested paging 570
nested task (rFLAGS.NT) 54, 389
nestedtable walk 573
NEXT_RIP 524
NMI 251
NMI support 559
NMI virtualization 559
no-execute (NX)
page-translation tables, bit in 158
nonmaskable interrupt exception (NMI) 251
while in SMM 337
non-PAE paging
CR3 format 133
NOP instruction 34
not write-through (CR0.NW) 45, 210
NP_ENABLE 573
NT bit 54
null selector 80

[AMD Public Use]

64-bit mode far return 123
interrupt return from 64-bit mode..... 289
long mode interrupts 286, 288
long mode stack switch 121
numeric error (CR0.NE)..... 44, 260
NW bit..... 45
NX bit..... 158
NXE bit 58

O

octword..... lvi
OE exception..... 259, 262
offset ..... lvi, 98
operand-size prefix 30
operating modes 11
OS FXSAVE/FXRSTOR support (CR4.OSFXSR)... 171, 346
OS unmasked exception support (CR4.OSXMMEXCPT) . 261, 346, 347
OSFXSR bit 50
OS-visible workarounds (OSVW)..... 810
OSVW ID..... 810
OSVW status 812
OSVW_ID_Length..... 812
OSXMMEXCPT bit..... 50, 51
OVER bit..... 313
overflow lvii
overflow exception (#OF)..... 252
overflow exception (OE) 259, 262

P

P bit..... 92, 156, 381
packed lvii
PAE..... 525
PAE bit 49, 52, 131
PAE paging 25, 133
CR3 format..... 46, 133
CR3 format, long mode..... 142
legacy mode..... 137
long mode..... 143
page directory..... 133
page size (PS) 132, 136, 138
page directory pointer 133, 138
page faults
guest level..... 575
page size (PS), page-translation tables 157
page splintering 579
page table..... 133
page translation 128
page-attribute table (PAT)..... 229
combined with MTRR 232
effect on memory access 231
identifying support 231
indexing..... 230
page-translation tables, bit in 158
Paged Real Mode..... 554
page-fault exception (#PF)..... 156, 258
page-fault virtual address..... 258
page-global enable (CR4.PGE) 50, 160
page-level cache disable (PCD)..... 210
CR3, bit in 134
page-translation tables, bit in 157
page-level write-through (PWT)..... 211
CR3, bit in 134
page-translation tables, bit in 157
page-map level-4 141
page-map level-5 141
page-size extensions (CR4.PSE)..... 25, 26, 49, 132, 136
40-bit physical address support 132, 137
unsupported in long mode 132
page-translation cache 159
page-translation options..... 131
page-translation tables..... 25
accessed (A)..... 157
available to software (AVL)..... 158
dirty (D)..... 157
global page (G)..... 158
hierarchy..... 130
no-execute..... 158
page directory entry (PDE) 133
page size (PS)..... 157
page table entry (PTE) 133
page-attribute table (PAT)..... 158
page-directory pointer entry (PDPE) 25, 133, 138
page-level cache disable (PCD)..... 157
page-level write-through (PWT) 157
page-map level-4 entry (PML4E)..... 25, 141
page-map level-4 entry (PML5E)..... 141
physical-page base address 156
present (P)..... 156
read/write (R/W)..... 156
translation-table base address..... 155
user/supervisor (U/S) 157
paging..... 7, 25, 128
See also PAE paging and non-PAE paging.
effect of segment protection..... 170
protection across translation hierarchy..... 169
protection checks..... 163
supported translations 130
paging enable (CR0.PG)..... 45, 131
activating long mode..... 131, 513
parameter count field..... 99
PAT..... 578
See page-attribute table (PAT).
PAT bit..... 158
PAT register..... 229, 759, 769
PAUSE..... 537

[AMD Public Use]

PCC bit 312
PCD bit 134, 143, 157
PCE bit 50
PDE 133
PDPE 133, 525, 579
PE bit 43
PE exception 259, 262
PerfCntrGlobalCtl 419
PerfCtr registers 413, 419, 420, 421, 762, 773, 774
PerfEvtSel registers 413, 762, 773
performance counter 180
performance counter enable (CR4.PCE) 50, 180, 415
performance counter hardware support-detecting 423
performance monitor counter interrupts 670
performance optimization 21, 413
performance-monitoring counters
  L2I_PerfEvtSeln 422
  NB_PerfEvtSeln 421
  overflow 424
  PerfCntrGlobalStatus 419
  PerfCtrn 414
  PerfEvtSeln 415
  starting and stopping 424
PG bit 45, 131
PGE bit 50
physical address 3, 24
  as index into cache 208
physical memory 4
physical-address extensions (CR4.PAE) ... 25, 26, 49, 52, 131, 141
  activating long mode 131, 513
  See also PAE paging.
POP instruction 180
POPF 536
precise exceptions and interrupts 244
precision exception (PE) 259, 262
PREFETCH instruction 212
present (P)
  descriptor 92, 381
  page-translation tables 156
principle of locality 159
priorities, interrupt 266
privilege level 107
probe, cache 188
  during cache disable 210
processor feature identification (rFLAGS.ID) 56
processor halt 183
processor modes
  16-bit liii
  32-bit liii
  64-bit liii
processor state 503
processor states 357
protected mode ..... lvii, 14, 523
  initial operating environment 509
protected-mode virtual interrupts (CR4.PVI) 49
protection checks
  adjusting RPL 183
  call gate 116
  checking access rights 182
  data segment 108
  direct call, conforming 113
  direct call, nonconforming 111
  enabling 76
  far return 122
  interrupt return 279, 280
  interrupt to higher privilege 276
  limit check, 64-bit mode 123
  long mode changes 27
  long mode interrupt 286
  long mode interrupt return 289
  stack segment 109
  type check 125
  verifying read/write access 182
protection domains 564
protection enable (CR0.PE) 43, 76, 84
PS bit 132, 157
PSE bit 49
PSE paging 25
P-State 696, 710
  control 696, 710
  current limit register 697
  status register 698
PTE 133
PUSH instruction 180
PUSHF 536
PVI bit 49
PWT bit 134, 143, 157

Q

quadword lvii

R

R bit 93
R/W bit 156
r8–r15 lxi
RA1 lvii
RAX 524
rAX–rSP lxi
RAZ lvii
RdMem, MTRR type field 62, 234
RDMSR 60, 180, 540
RDP field 365
RDPMC 50, 536
RDPMC instruction 180
RDTSC 49, 65, 180, 425, 536

[AMD Public Use]

RDTSCP 49, 65, 181, 425, 538
read hit 188
read miss 188
read ordering 212
read/write (R/W)
  page-translation tables, bit in 156
readable (R), code segment 93
real address 10
real address mode. See real mode
real mode ..... lvii, 4, 14
  initial operating environment 509
registers
  See also entries for individual registers.
  address-breakpoint registers (DR0-DR3) 394
  control registers 29, 41
  control-transfer recording MSRs 401
  CR0 42
  CR2 258
  CR3 ..... 25, 46, 133, 142
  CR4 47
  CSTAR 177
  debug registers 29, 393
  debug-control MSR (DebugCtl) 399
  debug-control register (DR7) 396
  debug-extension MSRs 64
  descriptor-table registers 26, 78
  eAX–eSP lx
  EFER 29, 56, 68, 70
  eFLAGS lx
  eIP lx
  FPR 350, 352
  FS and GS 82
  GDTR 85
  GPRs 28
  HWCR 72
  IBS fetch control 428
  IbsBrTarget 443
  IbsDcLinAd 442
  IbsDcPhysAd 443
  IbsFetchCtl 428
  IbsFetchLinAd 430
  IbsFetchPhysAd 431, 433
  IbsOpCtl 435
  IbsOpData1 438
  IbsOpData2 439
  IbsOpData3 439
  IbsOpRip 437
  IDTR 90
  IORRBase 236
  IORRMask 237
  L2I_PerfEvtSeln 422
  last x87 data pointer 350, 352, 365
  last x87 instruction pointer 350, 352, 364
  LDTR 87
  LSTAR 177
  machine-check MSRs 65, 66, 67
  MCG_CAP 303
  MCG_CTL 305
  MCG_STATUS 304
  MCi_ADDR 313
  MCi_CTL 310
  MCi_MISC 313
  MCi_STATUS 310
  memory-type range register (MTRR) 29, 64, 220
  MMX 348
  model-specific registers (MSRs) 29
  MTRR, fixed range 221
  MTRR, variable range 223
  MTRRcap 226
  MTRRdefType 226
  MTRRfix16K 222
  MTRRfix4K 222
  MTRRfix64K 222
  MTRRphysBase 223
  MTRRphysMask 224
  MXCSR 347
  NB_PerfEvtSeln 421
  PAT 229
  PerfCntrGlobalCtl 419
  PerfCntrGlobalStatus 419
  PerfCntrGlobalStatusClr 421
  PerfCntrGlobalStatusSet 420
  PerfCtrn 414
  PerfEvtSeln 415
  performance-monitoring MSRs 65
  r8–r15 lxi
  rAX–rSP lxi
  rFLAGS ..... lxi, 28, 52
  rIP lxii
  rSP 29
  segment registers 81
  SSE registers 28
  STAR 177
  SYSCFG 61
  SYSENTER_CS 178
  SYSENTER_EIP 178
  SYSENTER_ESP 178
  system-linkage MSRs 63
  task-priority register (CR8) 38, 52, 269
  time-stamp counter 65, 425
  TOP_MEM 62, 238
  TOP_MEM2 ..... 62, 238
  x87 FCW 350, 352, 364
  x87 floating-point processor state 349
  x87 FSW 349, 350, 352, 364
  x87 FTW 349, 350, 352, 364
  x87 opcode 350, 352, 364
relative lvii
replacement, cache-line 188
replicated state 572
reserved lviii

[AMD Public Use]

reset ... 502
  processor state ... 503
RESET# signal ... 502
resume flag (rFLAGS.RF) ... 55, 251, 410
RET instruction ... 122
  from 64-bit mode ... 123
  long mode ... 33, 122
  popping null selector, 64-bit mode ... 123
  stack switch ... 122
retire, instruction ... 189
revision history ... xxxvi
REX prefix ... 30
RF bit ... 55
RFLAGS ... 524
rFLAGS ... 28, 52
  alignment check (AC) ... 55, 260
  I/O privilege level field (IOPL) ... 54
  interrupt flag (IF) ... 54, 180
  nested task (NT) ... 54, 389
  processor feature identification (ID) ... 56
  resume flag (RF) ... 55, 251, 410
  trap flag (TF) ... 54
  virtual interrupt (VIF) ... 55, 291
  virtual interrupt pending (VIP) ... 56, 291
  virtual-8086 mode (VM) ... 55
rFLAGS register ... lxi
RIP ... 524
rIP ... 28
rIP register ... lxii
RIP-relative address ... 31
RIP-relative addressing ... lviii
RIPV bit ... 305
RPL ... 80, 108, 371
  adjusting ... 183
  definition ... 108
  in call gate protection ... 117
  in data segment protection ... 108, 322
  in far return ... 122
  in IRET instruction ... 279
  in protecting conforming CS ... 114
  in protecting nonconforming CS ... 112
  in stack segment protection ... 110
RSM ... 326, 342, 537
RSP ... 524
rSP ... 29
  call gate stack switch ... 119
  implicit reference ... 31

S

S bit ... 92, 381
SBZ ... lviii
SCE bit ... 57
secure initialization ... 570
secure loader (SL) ... 582
secure loader (SL) image ... 582
secure loader block ... 583
secure MP initialization ... 586
secure multi-key memory encryption ... 243
  EncrKeyID ... 243
SecureNestedPagingEn bit ... 63
security exception (#SX) ... 582, 588
segment base ... 91
segment limit ... 91
segment offset ... 2
segment registers ... 78, 81
  64-bit mode ... 82
  accessing ... 181
  hidden portion ... 81
  initializing unused registers ... 80
segmentation ... 5, 26
  64-bit mode ... 77
  combining with paging ... 8
  flat segmentation ... 6, 9, 77
  multi-segmented model ... 76
segment-not-present exception (#NP) ... 256
segment-override prefix ... 30
selector ... 78, 79, 80, 88, 98, 371
selector index ... 80
self-modifying code ... 208
SEOI Register ... 665, 685
serializing instructions ... 213
set ... lviii
SEV-SNP instruction virtualization ... 645
SF exception ... 259
SFENCE ... 191
SFENCE instruction ... 212
SGDT ... 182, 536
shadow page tables (SPTs) ... 551
shadow stack registers ... 66
shut down ... 254
SIDT ... 182, 536
SIMD floating-point exception (#XF) ... 50, 261, 346, 347
single-step
  all instructions ... 393, 410
  control-transfers ... 393, 412
SKINIT ... 537, 570, 582
SL abort ... 586
SLDT ... 182, 536
SMBASE register ... 328
SMI ... 326
  external, synchronous ... 560
  internal, synchronous ... 560
  xternal, asynchronous ... 561
SMM ... 326
SMM interrupts ... 337

[AMD Public Use]

SMM revision identifier 333
SMM state-save area 329
SMM_CTL MSR 609
SMRAM 327
SMRAM state-save area 329
SMSW 536
SMSW instruction 179
SMT protection 642
specific EOI (SEOI) 684, 686, 688, 689, 691
speculative execution 189
Spurious Interrupts 672
SS register 82, 524
SS.SEL 524
SSE Instructions
  subset support 344
SSE instructions
  enabling 171, 346
  saving state 351
  YMM/XMM registers 28, 347
SSM Containerization 804
stack exception (#SS) 256
stack pointers
  in 32-bit TSS 376
  in 64-bit TSS 378
stack segment 82, 94
  64-bit mode 82
  default operand size (D) 96
  expand down (E) 95
  privilege checks 109
stack switch
  call gate 119
  call gate, long mode 33, 120
  far return 122
  interrupt 275
  interrupt return 279, 280
  interrupt, long mode 37
stack-fault exception (SF) 259
STAR register 177, 760, 767
state switch 522
status word 179
stepping ID field 506
STGI 537, 553
STI instruction 180
sticky bits lviii
store ordering 212
STR 536
STR instruction 182
SVM 560
SVM support 560
SVME bit 58
SWAPGS instruction 178
SYSCALL Flag Mask register 177
SYSCALL, SYSRET instructions 57, 176
SYSCFG register 61, 763, 769
  HostMultiKeyMemEncrModeEn 63
  HostMultiKeyMemEncrModeEn bit 243
  MtrrFixDramEn 62, 235
  MtrrFixDramModEn 62, 235
  MtrrTom2En 240
  MtrrVarDramEn 62, 240
  SecureNestedPagingEn 63
  Tom2ForceMemTypeWB 63
SYSENTER_CS register 178, 757, 768
SYSENTER_EIP register 178, 757, 768
SYSENTER_ESP register 178, 757, 768
SYSENTER, SYSEXIT instructions 178
  illegal in long mode 178
system call and return 175
system data structures 17
system management interrupt (SMI) 326, 336, 560
  while in SMM 337
system management mode (SMM) 15, 24, 560
  leaving 342
  long mode differences 326
  operating environment 336
  revision identifier 333
  saving processor state 339
  SMBASE register 328
  SMRAM 327
  state-save area, AMD64 architecture 329
  state-save area, legacy 332
system registers 15
system segment 27, 91, 96
  ignored fields in 64-bit mode 102
  illegal types in long mode 101
  long mode 101
  type field 96
system-call extension (EFER.SCE) 57
system-linkage MSRs 63, 177

T

T bit 376
table indicator, selector 80
tagged TLB 521
task gate 97
  in task switching 387
  long mode 105
Task Register (TR) 78
task register (TR) 372
  loading 182
  selector 371
  storing 182
task switch 368, 382
  disabled in long mode 38
  lazy context switch 44, 367
  nesting tasks 389

[AMD Public Use]

preventing recursion 390
task switched (CR0.TS) 44, 180
task, execution space 368
task-priority register (CR8) 38, 52, 269
task-state segment (TSS)
descriptor 371
dynamic fields 376
I/O-permission bitmap 376, 379
interrupt-redirection bitmap 377
interrupt-stack table 379
legacy 32-bit 374
link field 389
software-defined fields 376
stack pointers 376, 378
static fields 376
TCE bit 59
TF bit 54
Thermal Sensor 660
Thermal Sensor Interrupts 671
TI bit 80, 371
time-stamp counter 180, 425
time-stamp disable (CR4.TSD) 49, 180, 425
TLB 158, 159, 523, 524
explicit invalidation 161, 184, 185
implicit invalidation 161
TLB Control 551
TLB entry upgrades 162
TLB flush 551
TLB_CONTROL 552
Tom2ForceMemTypeWB bit 63
top of memory 238
top of memory, virtual 633
TOP_MEM register 62, 238, 763, 769
TOP_MEM2 register 62, 238, 763, 769
TPM 584
TPR register 38, 52, 269
TR register 369, 372, 536
translation lookaside buffer (TLB) 159
trap 245
trap flag (rFLAGS.TF) 54
trap gate 97, 283
Trigger Mode Register 680
Trusted Platform Module (TPM) 582
trusted software 582
TS bit 44
TSC register 425, 757, 773
TSD bit 49
TSS ..... lix, 369, 374
TSS descriptor 369
TSS selector 98, 369
type check 125
Type field 92, 372, 381

U

U/S bit 157
UAIE bit 59
UC bit 313
UC memory type 198
UD2 instruction 252
UE exception 259, 262
uncacheable (UC-), memory type 230
underflow lix
underflow exception (UE) 259, 262
user segment 90
user/supervisor (U/S)
page-translation tables, bit in 157

V

V_IGN_TPR 557
V_INTR_MASKING 555
V_INTR_PRIO 557
V_INTR_VECTOR 557
V_IRQ 525, 557
V_TPR 525, 556, 557
VAL bit 313
Variable-range IORRs 236
vector lix
vector, interrupt 247
VERR instruction 182
VERW instruction 182
VIF bit 55
VIP bit 56
virtual #INTR 556
virtual address 3, 24
virtual APIC register access behavior 591
virtual APIC register accesses 591
virtual interrupt (rFLAGS.VIF) 55, 291
virtual interrupt pending (rFLAGS.VIP) 56, 291
virtual interrupts 54, 55, 56, 290, 291, 521, 559
virtual interrupts, protected mode 293
virtual machine control block (VMCB) 523
virtual machine monitor 520
virtual memory 3
virtual-8086 mode ..... lix, 14
interrupt to protected mode 281
interrupts 280
virtual-8086 mode (rFLAGS.VM) 55
virtual-8086 mode extensions (CR4.VME) 48, 290
VM bit 55
VM_HSAVE_AREA 524
VM_SAVE_PA MSR 610
VMCB 523
AVIC fields 595
NMI virtualization 559

[AMD Public Use]

virtual interrupt control word 595
VME 290
VME bit 48, 280
VMLOAD 537
VMM 520
VMMCALL 537, 553
VMPL
permission mask 632
VMPL supervisor shadow stack 633
VMPLEn bit 63
VMRUN 521, 523, 537
VMSAVE 537

W

W bit 95
WAIT/FWAIT instruction 43
WB memory type 200
WBINVD 183, 185, 212, 538
WC memory type 199
WC+ 578
world switch 520
WP bit 44
WP memory type 199
writable (W), data segment 95
write buffer 188, 204
emptying 204
write hit 188
write miss 188
write ordering 190, 212
write protect (CR0.WP) 44
write-back (WB), memory type 200
writeback, cache line 188
write-combining buffer 188, 205
emptying 205
write-combining plus memory type 199
write-though (WT)
memory type 200
WrMem, MTRR type field 62, 234
WRMSR 60, 180, 540
WT memory type 200

X

x2AVIC 607
x87 control word 350, 352, 364
x87 data pointer register 350, 352, 365
x87 environment 352
x87 floating-point instructions
initializing 506
processor state 349
saving state 351
x87 instruction pointer register 350, 352, 364
x87 opcode register 350, 352, 364
x87 status word 349, 350, 352, 364
x87 tag word 349, 350, 352, 364
FXSAVE format 365
XSS Register 67

Y

YMM states 359

Z

ZE exception 259, 262
zero extension 30, 31
zero-divide exception (ZE) 259, 262
ZMM states 359