Preliminary Information

AMD 64-Bit Technology

The AMD x86-64™ Architecture Programmers Overview

Publication # **24108** Rev: **A**
Issue Date: **August 2000**

Contents

Introduction

AMD 64-bit technology includes the x86-64™ architecture, which is a 64-bit extension of the x86 architecture. The x86-64 architecture supports legacy 16-bit and 32-bit applications and operating systems without modification. It provides recompiled 64-bit applications and operating systems with these new features:

- 64-bit flat virtual addressing.
- 8 new general-purpose registers (GPRs).
- 8 new registers for streaming SIMD extensions (SSE).
- 64-bit-wide GPRs and instruction pointer.

The x86-64 architecture has a legacy mode in which it supports binary compatibility with existing operating systems and applications, and a new mode in which it supports both the new features for recompiled code as well as binary compatibility with existing applications. The architecture also adds a new instruction-pointer relative-addressing mode, uniform byte-register addressing, and a fast interrupt-prioritizing mechanism.

This document describes the new features of AMD's x86-64 architecture and their differences from legacy x86 architecture.

Motivation for a 64-Bit Architecture

The need for a 64-bit x86 architecture is driven by applications that address large amounts of virtual and physical memory, such as high-performance servers, database management systems, and CAD tools. These applications benefit from both 64-bit addresses and an increased number of registers.

The small number of registers available in the legacy x86 architecture also limits performance in computation-intensive applications such as graphics transform and lighting, circuit simulation, and scientific algorithms. Increasing the number of registers provides a performance boost to many applications.

Features of the x86-64™ Architecture

The x86-64 architecture extends the legacy x86 architecture by introducing two major features: a 64-bit extension called *long mode* and register extensions.

Long Mode

Long mode consists of two sub-modes: *64-bit mode* and *compatibility mode*. Compatibility mode supports binary compatibility with existing 16-bit and 32-bit applications. In addition to long mode, the architecture also 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 systems.

Table 1 shows the supported operating modes.

Throughout this document, references to *long mode* refer 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*.

Table 1. Operating Modes

Mode		Operating System Required	Application Recompile Required	Defaults[1]			
				Address Size (bits)	Operand Size (bits)	Register Extensions[2]	GPR Width (bits)
Long Mode[3]	64-Bit Mode	New 64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16			
Legacy Mode[4]		Legacy 32-bit or 16-bit OS	no	32	32	no	32
				16	16		

1. Defaults can be overridden in most modes using an instruction prefix or system control bit.
2. Register extensions includes eight new GPRs and eight new XMM registers (also called SSE registers).
3. Long mode supports only x86 protected mode. It does not support x86 real mode or virtual-8086 mode. Also, it does not support task switching.
4. Legacy mode supports x86 real mode, virtual-8086 mode, and protected mode.

64-Bit Mode

64-bit mode supports the following new features:

- 64-bit virtual addresses (implementations can have less).
- Register extensions through a new prefix (REX):
  - Adds eight GPRs (R8–R15).
  - Widens GPRs to 64 bits.
  - Adds eight 128-bit streaming SIMD extension (SSE) registers (XMM8–XMM15).
- 64-bit instruction pointer (RIP).
- New RIP-relative data addressing mode.
- Flat address space with single code, data, and stack space.

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 prefixes. A new REX prefix is introduced for specifying 64-bit operand size and the new registers.

The mode is enabled by the operating system on an individual code-segment basis. Because 64-bit mode supports a 64-bit virtual-address space, it requires a 64-bit operating system and tool chain. A few instruction opcodes and prefix bytes are redefined to allow the register extensions and 64-bit addressing. These differences are described in Appendix A, "Integer Instructions in 64-Bit Mode", on page 73, and in Appendix B, "Long Mode Differences", on page 98.

Register Extensions. 64-bit mode supports register extensions, shown in Figure 1, through a new REX prefix. These extensions add eight 64-bit GPRs (R8–R15), eight 128-bit streaming SIMD extensions (SSE) registers (XMM8–XMM15), and widens all GPRs to 64 bits.

The REX prefix also provides a new byte-register capability that makes the least-significant byte of any GPR available for byte operations. This results in a uniform set of byte, word, dword, and qword registers better suited for a compiler's register allocation.

The instruction pointer is also widened to 64 bits.

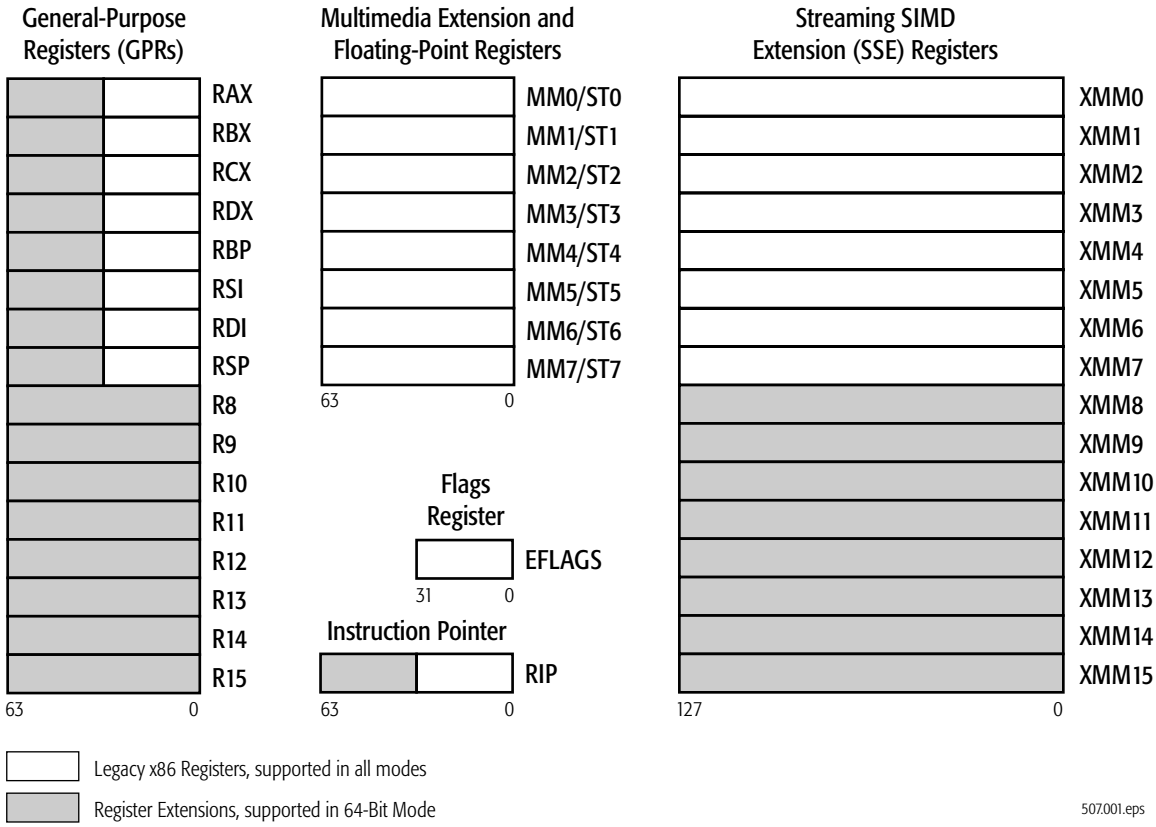


Figure 1. Register Extensions

RIP-Relative Data Addressing. In 64-bit mode, the architecture also supports data addressing relative to the 64-bit instruction pointer (RIP). The legacy x86 architecture supports IP-relative addressing only in control-transfer instructions. The 64-bit mode's RIP-relative addressing improves the efficiency of position-independent code and code that addresses global data.

Compatibility Mode

Compatibility mode allows operating systems to implement binary compatibility with existing 16-bit and 32-bit x86 applications. It allows these applications to run, without recompilation, under a 64-bit operating system in long mode, as shown in Table 1 on page 2.

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

As with 64-bit mode, compatibility mode is enabled by the operating system on an individual code-segment basis. Unlike 64-bit mode, however, x86 segmentation functions normally, using 16-bit or 32-bit protected-mode semantics. From the application's viewpoint, compatibility mode looks like a legacy x86 protected-mode environment. From the operating system's viewpoint, address translation, interrupt and exception handling, and system data structures use the 64-bit long mode mechanisms.

Legacy Mode

Legacy mode is completely compatible with existing 32-bit implementations of the x86 architecture.

Definitions

Many of the following definitions assume an in-depth knowledge of the legacy x86 architecture.

Notation

16-bit mode — Legacy mode or compatibility mode in which a 16-bit address size is active.

32-bit mode — Legacy mode or compatibility mode in which a 32-bit address size is active.

64-bit mode — 64-bit mode, which has a 64-bit address size.

#GP(0) — General-protection exception (#GP) with error code of 0.

CPL — Current privilege level.

CR0.PE=1 — The PE bit of the CR0 register has a value of 1.

dword — 32 bits, four bytes.

EFER.LME=0 — The LME bit of the EFER register has a value of 0.

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

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

FF/0 — FF is first byte of an opcode, and /0 means that a sub-field in the second byte is zero.

IGN — Ignore. Field is ignored.

IMP — Implementation dependent.

Legacy — The legacy x86 architecture.

ModRM — A byte following an instruction opcode that specifies address calculation based on mode (mod), register (r), and memory (m).

MBZ — Must be zero. If not zero, a general-protection exception (#GP) occurs.

	PAE	Physical-address extensions.
	PCR	Processor control region.
	qword	64 bits, eight bytes.
	RAZ	Read as zero (0).
	REX	A new instruction prefix that specifies a 64-bit operand size and provides access to additional registers.
	SBZ	Should be zero (0). Non-zero values produce unpredictable results.
	SIB	A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).
	SIMD	Single instruction, multiple data.
	SSE	Streaming SIMD extensions.
	TEB	Thread environment block.
	word	16 bits, two bytes.
Registers	eAX–eSP	The AX, BX, CX, DX, DI, SI, BP, SP registers or the EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP registers.
	EBP	Extended base pointer.
	BP	Base pointer.
	CR*n*	Control register number *n*.
	CS	Code segment.
	EFER	Extended features enable register.
	EFLAGS	Extended flags register.
	EIP	Extended instruction pointer.
	GDTR	Global descriptor table register.
	GPRs	General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP. For the 64-bit

	data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.
IDTR	Interrupt descriptor table register.
IP	Instruction pointer.
LDTR	Local descriptor table register.
MSR	Model-specific register.
RAX–RSP	The AX, BX, CX, DX, DI, SI, BP, SP registers, or the EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP registers, or the RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP registers.
RAX	64-bit version of EAX register.
RBP	64-bit version of EBP register.
RBX	64-bit version of EBX register.
RCX	64-bit version of ECX register.
RDI	64-bit version of EDI register.
RDX	64-bit version of EDX register.
RIP	64-bit instruction pointer.
RSI	64-bit version of ESI register.
RSP	64-bit version of ESP register.
rSP	Stack pointer. The “r” variable should be replaced by nothing for 16-bit stack size, “E” for 32-bit stack size, or “R” for 64-bit stack size.
SP	Stack pointer.
SS	Stack segment.
TPR	Task priority register, a new register introduced in the x86-64 architecture to speed interrupt management.
TR	Task register.
TSS	Task state segment.

Application Programming

Overview

This section describes the application register set, instruction-set conventions, memory addressing, and application-relevant aspects of procedure-call conventions. It is intended primarily for experienced x86 programmers writing applications, compilers, or assemblers. System programmers writing privileged code should also read this section.

Additional application-programming information can be found in the following appendixes:

- Appendix A, "Integer Instructions in 64-Bit Mode", on page 73.
- Appendix B, "Long Mode Differences", on page 98.
- Appendix D, "Implementation Considerations", on page 105.

Application Software Registers and Data Structures

Table 2 compares the registers and data structures visible to application software in 64-bit mode with those visible in the legacy x86 architecture. The legacy x86 values also apply to the x86-64 architecture compatibility and legacy modes. Gray shading indicates differences between the architectures. The register differences (not including stack-width difference) represent what is called the *register extensions* of the x86-64 architecture.

Table 2. Application-Software Differences in Registers and Data Structures

Software-Visible Register or Data Structure	Legacy and Compatibility Modes			64-Bit Mode		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General-Purpose Registers	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8-15	16	64
Floating-Point Registers[1]	ST0-7	8	80	ST0-7	8	80
Multimedia Extension Registers[1]	MM0-7	8	64	MM0-7	8	64
Streaming SIMD Extension Registers	XMM0-7	8	128	XMM0-15	16	128
Instruction Pointer	EIP	1	32	RIP	1	64
Flags	EFLAGS	1	32	EFLAGS	1	32
Stack Width	—		16 or 32	—		64

1. Legacy ST and MMX™ technology registers share the same register space.

CPUID

Application software can use the CPUID instruction to determine processor features, such as whether the x86-64 architecture's long mode is supported by the processor. To do this, software typically loads a function code into the EAX register and executes the CPUID instruction. Processor feature information is returned in the EAX, EBX, ECX, and EDX registers.

Loading EAX with the function code 8000_0001h causes the CPUID instruction to return the extended feature flags shown in Table 3 to EAX. If bit 29 is set to 1, long mode is available.

Table 3. Long-Mode Feature Flag Returned to EAX for CPUID 8000_0001h

Bit	Feature	
29	LM	Long Mode

Application Registers

Figure 1 on page 4 shows the application registers, including both the legacy x86 register set and the register extensions available in 64-bit mode. Only the GPR and XMM registers are extended in 64-bit mode. The floating-point and MMX™ technology registers are not extended.

General-Purpose Registers (GPRs)

Legacy-Mode and Compatibility-Mode GPRs. In compatibility and legacy modes, the GPRs consist only of the eight legacy registers. When the operand size is 32-bit, the registers are EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP. When the operand size is 16-bit, the registers are AX, BX, CX, DX, DI, SI, BP, SP. All legacy rules apply for determining operand size.

64-Bit Mode GPRs. In 64-bit mode, the GPRs are 64 bits and eight additional GPRs are available. Their names are RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and the new R8–R15 registers.

Figure 2 shows the GPRs in 64-bit mode.

The default operand size in 64-bit mode is 32 bits, which gives access to the EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. To access the full 64-bit operand size, an instruction must contain a new REX prefix byte. To access 16-bit operand sizes, an instruction must contain an operand-size prefix (66h). These prefix bytes are described in "Address-Size and Operand-Size Prefixes" on page 17.

64-bit mode provides a uniform set of byte, word, dword, and qword registers better suited for register allocation by compilers. This view of the byte registers is enabled by REX prefixes, as described in "REX Prefixes" on page 18.

The *black shading* in Figure 2 indicates legacy high-byte registers, with the register-number encoding shown in parentheses. The *dark gray shading* in Figure 2 indicates new byte registers within legacy registers. Access to these new byte registers requires a REX instruction prefix. An instruction cannot reference both a legacy high-byte register and a new byte register at the same time. However, an instruction can reference a legacy low-byte register and a new byte register at the same time.

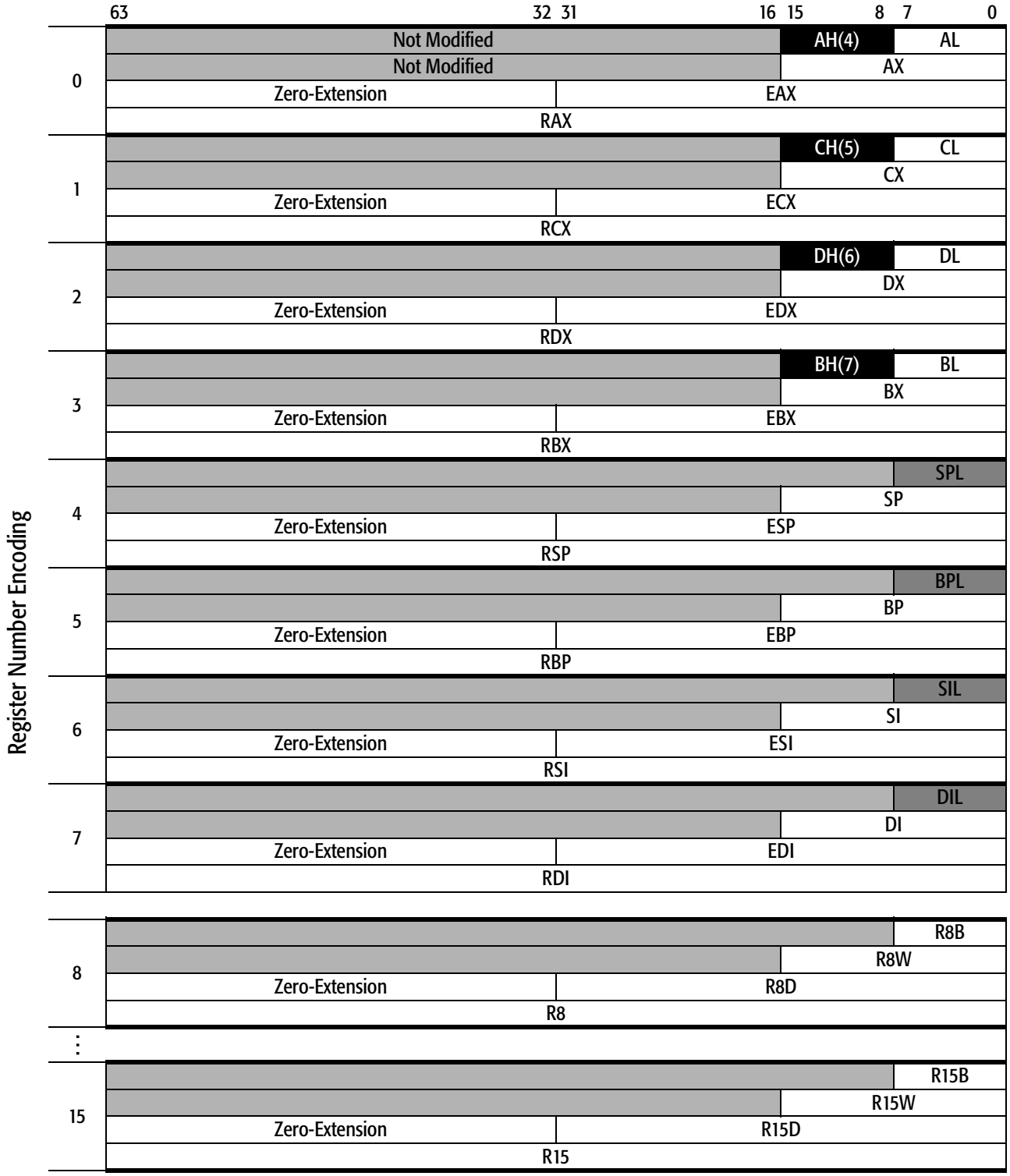


Figure 2. GPRs in 64-Bit Mode

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. 8-bit and 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.

The results of 8-bit, 16-bit, and 32-bit operations should be explicitly sign-extended to the full width before use in 64-bit address calculations.

The following four code examples show how 64-bit, 32-bit, 16-bit, and 8-bit adds work. In these examples, "48" is a REX prefix specifying 64-bit operand size, and "01C3" and "00C3" are the opcode and ModRM bytes of each instruction. See "REX Prefixes" on page 18 for a description of the REX prefix.

Example 1: 64-bit add

```
Before:RAX =0002_0001_8000_2201
       RBX =0002_0002_0123_3301

       48 01C3 ADD RBX,RAX ;48 is a REX prefix for size.

Result:RBX = 0004_0003_8123_5502
```

Example 2: 32-bit add

```
Before:RAX =0002_0001_8000_2201
       RBX =0002_0002_0123_3301

       01C3 ADD EBX,EAX ;32-bit add

       Result:RBX = 0000_0000_8123_5502
       (32-bit result is zero extended)
```

Example 3: 16-bit add

```
Before:RAX =0002_0001_8000_2201
       RBX =0002_0002_0123_3301

       66 01C3 ADD BX,AX ;66 is 16-bit size override

       Result:RBX = 0002_0002_0123_5502
       (bits 63:16 are preserved)
```

Example 4: 8-bit add

```
Before:RAX =0002_0001_8000_2201
       RBX =0002_0002_0123_3301

       00C3 ADD BL,AL ;8-bit add

       Result:RBX = 0002_0002_0123_3302
       (bits 63:08 are preserved)
```

Preservation of GPR High 32 Bits Across Mode Switches. The processor does not preserve the upper 32 bits of the 64-bit GPRs across switches from 64-bit mode to compatibility or legacy modes. When using 32-bit operands in compatibility or legacy mode, the high 32 bits of GPRs are undefined. Software must not rely on these undefined bits, because they can change on a cycle-to-cycle basis within a given implementation.

Streaming SIMD Extension (SSE) Registers

In compatibility and legacy modes, the SSE registers consist of the eight 128-bit legacy registers, XMM0–XMM7. In 64-bit mode, eight additional 128-bit SSE registers are available, XMM8–XMM15. These are part of the register extensions illustrated in Figure 1 on page 4. Access to these registers is controlled on an instruction-by-instruction basis using a REX instruction prefix, as described in "REX Prefixes" on page 18.

Memory Organization

Address Calculations in 64-Bit Mode

Effective Addresses. In 64-bit mode if there is no address-size override, the size of effective address calculations is 64 bits. An effective-address calculation uses 64-bit base and index registers and sign-extends displacements to 64 bits. Due to the flat address space in 64-bit mode, virtual addresses are equal to effective addresses. (For an exception to this general rule, see "Special Treatment of FS and GS Segments" on page 42.)

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*. Figure 3 shows the relationship between RIP, EIP, and IP.

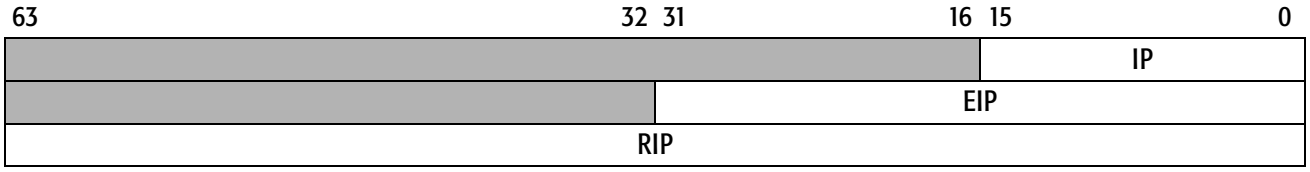


Figure 3. Instruction Pointer

Displacement and Immediates. Generally, displacements and immediates in 64-bit mode are not extended to 64 bits. They are still limited to 32 bits and 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. See "Displacement" on page 24 and "Immediates" on page 24 for more information on this.

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.

Because of this, 16-bit and 32-bit applications running in compatibility mode can access only the low 4GBytes of the long-mode virtual-address space. Likewise, a 32-bit address generated in 64-bit mode can access only the low 4GBytes of the long-mode virtual-address space.

FS and GS As Base of Address Calculation

The FS and GS segment-base registers can be used as base registers for address calculation, as described in "Special Treatment of FS and GS Segments" on page 42.

Instruction-Set Conventions

Address-Size and Operand-Size Prefixes

Address-Size Overrides. In 64-bit mode, the default address size is 64 bits and the default operand size is 32 bits. As in legacy mode, these defaults can be overridden using instruction prefixes. The address-size and operand-size prefixes allow mixing of 32-bit and 64-bit data and addresses on an instruction-by-instruction basis.

Table 4 shows the instruction prefixes for address-size overrides in all operating modes. 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. 16-bit addresses are not supported in 64-bit mode. In compatibility and legacy mode, address sizes function the same as in x86 legacy architecture.

Table 4. Address-Size Overrides

Mode		Default Address Size (Bits)	Effective Address Size (Bits)	Address-Size Prefix (67h)[1] Required?
Long Mode	64-Bit Mode	64	64	no
			32	yes
	Compatibility Mode	32	32	no
			16	yes
		16	32	yes
			16	no
Legacy Mode		32	32	no
			16	yes
		16	32	yes
			16	no

1. “no’ indicates the default address size.

Operand-Size Overrides. Table 5 shows the instruction prefixes for operand-size overrides in all operating modes. In 64-bit mode, the default operand size is 32 bits. A REX prefix (see "REX Prefixes" on page 18) can specify a 64-bit operand size. Software can use an operand-size (66h) prefix to toggle to 16-bit operand size. The REX prefix takes precedence over the operand-size (66h) prefix.

Table 5. Operand-Size Overrides

Mode		Default Operand Size (Bits)	Effective Operand Size (Bits)	Instruction Prefix	
				66h[1]	REX
Long Mode	64-Bit Mode	64	64	x	yes
			32	no	no
			16	yes	no
	Compatibility Mode	32	32	no	Not Applicable
			16	yes	
		16	32	yes	
			16	no	
Legacy Mode		32	32	no	
			16	yes	
		16	32	yes	
			16	no	

1. "no' indicates the default operand size. "x" means don't care.

REX Prefixes

REX prefixes are a new family of instruction-prefix bytes used in 64-bit mode to:

- Specify the new GPRs and SSE registers shown in Figure 1 on page 4.
- Specify a 64-bit operand size.
- Specify extended control registers (used by system software).

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.

Number and Position. An instruction can have only one REX prefix. This prefix, if used, must immediately precede the opcode byte or the two-byte opcode escape prefix (if used) of an instruction. Any other placement of a REX prefix is ignored. The legacy instruction-size limit of 15 bytes still applies to instructions that contain a REX prefix.

Figure 4 shows how a REX prefix fits within the byte-order of instructions.

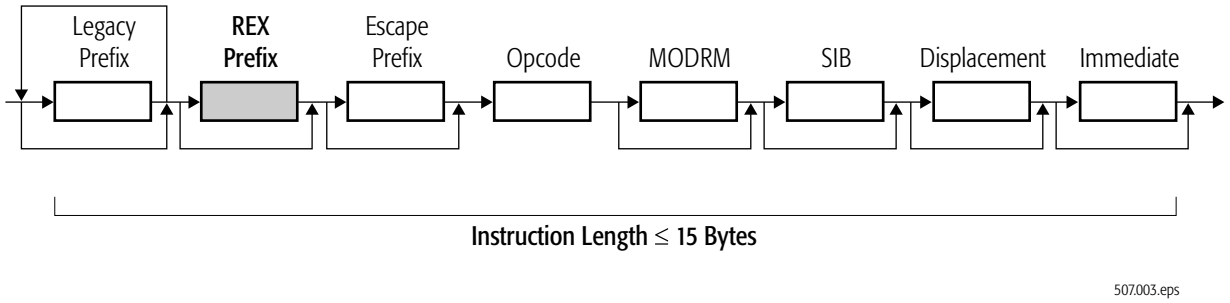


Figure 4. Instruction Byte Order

Encoding. x86 instruction formats specify up to three registers by using 3-bit fields in the instruction encoding, depending on the format:

- ModRM: the *reg* and *r/m* fields of the ModRM byte.
- ModRM with SIB: the *reg* field of the ModRM byte and the *base* and *index* fields of the SIB (scale, index, base) byte.
- Instructions without the ModRM: the *reg* field of the opcode.

In 64-bit mode, these fields and formats are not altered. Instead, the bits needed to extend each field for accessing the additional registers are provided by the new REX prefixes.

REX Prefix Fields

REX prefixes are a set of sixteen values that span one row of the main opcode map and occupy entries 40h to 4Fh. Table 6 and Figure 5 show the prefix fields and their uses.

Table 6. REX Prefix Fields

Field Name	Bit Position	Definition
—	7:4	0100
W	3	0 = Default operand size 1 = 64-bit operand size
R	2	High extension of the ModRM *reg* field.
X	1	High extension of the SIB *index* field.
B	0	High extension of the ModRM *r/m* field, SIB *base* field, or opcode *reg* field.

REX.W: Operand Width. Setting the REX.W bit specifies a 64-bit operand size. Like the existing 66h operand size prefix, the REX 64-bit operand size override has no effect on byte operations. For non-byte operations, the REX operand-size override takes precedence over the 66h prefix. If a 66h prefix is used together with a REX prefix with bit 3 set to 1, the 66h prefix is ignored. However, if a 66h prefix is used together with a REX prefix with REX.W cleared to 0, the 66h prefix is not ignored and the operand size becomes 16 bits.

REX.R: Register. The REX.R bit modifies the ModRM *reg* field when that field encodes a GPR or SSE register. REX.R does not modify ModRM *reg* when that field specifies other registers or opcodes. REX.R is ignored in such cases.

REX.X: Index. The REX.X bit modifies the SIB *index* field.

REX.B: Base. The REX.B bit either modifies the base in the ModRM *r/m* field or SIB *base* field, or modifies the opcode *reg* field used for accessing GPRs, control registers, or debug registers.

Encoding Examples. Figure 5 shows four examples of how the R, X, and B bits of REX prefixes are concatenated with fields from the ModRM byte, SIB byte, and opcode to specify register and memory addressing. The R, X, and B bits are described in Table 6.

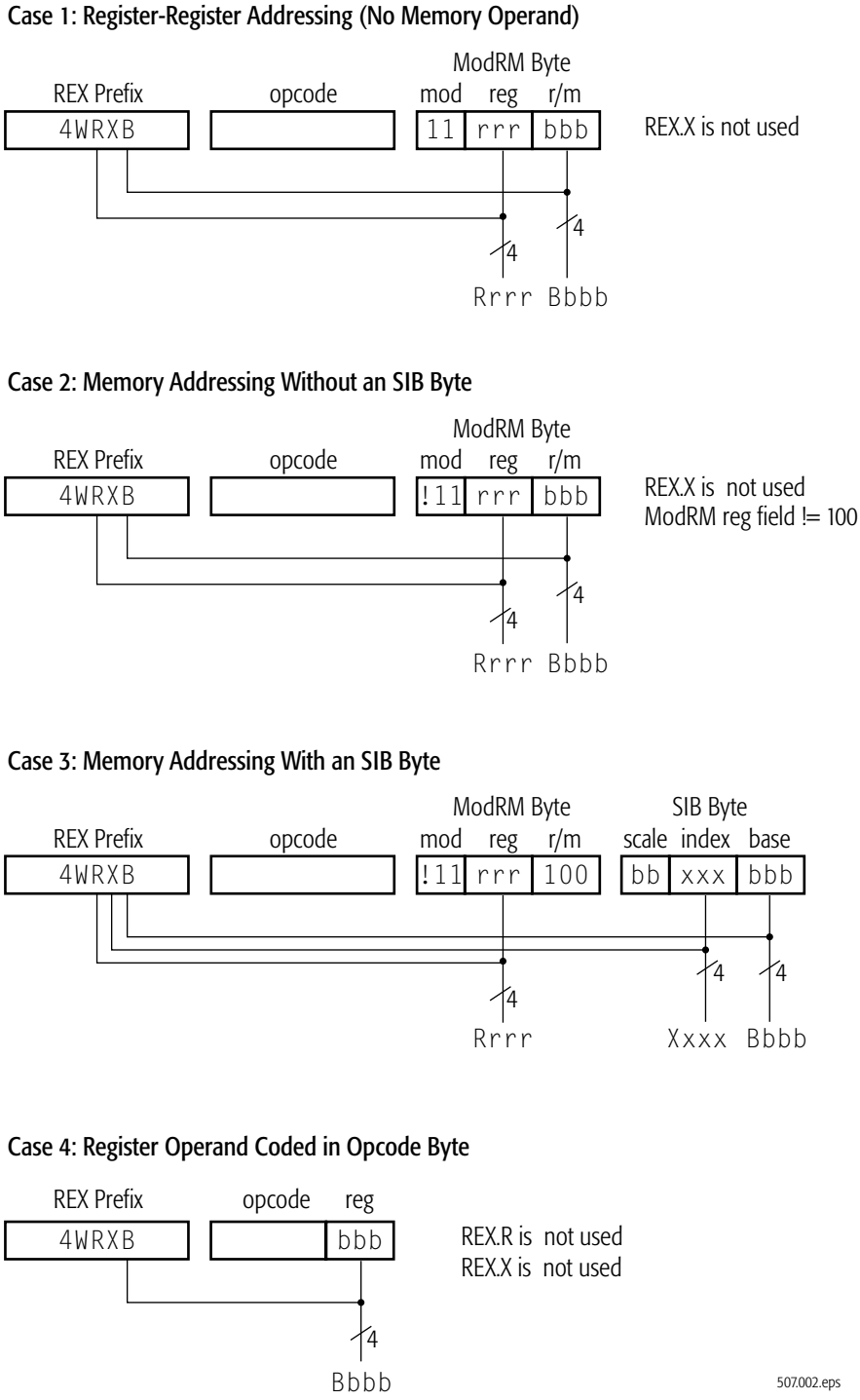


Figure 5. Encoding Examples of REX-Prefix R, X, and B Bits

Byte-Register Addressing. In the legacy architecture, the byte registers (AH, AL, BH, BL, CH, CL, DH, and DL, shown in Figure 2 on page 13) are encoded in the ModRM *reg* or *r/m* field or opcode *reg* field as registers 0 through 7. The REX prefix provides an additional byte-register addressing capability that makes the least-significant byte of any GPR available for byte operations. This provides a uniform set of byte, word, dword, and qword registers better suited for register allocation by compilers.

Special Encodings for Registers. Readers who need to know the details of instruction encodings should be aware that certain combinations of the ModRM and SIB fields have special meaning for register encodings. For some of these combinations, the instruction fields expanded by the REX prefix are not decoded (treated as don't cares), thereby creating aliases of these encodings in the extended registers. Table 7 describes how each of these cases behaves.

Implications for INC and DEC Instructions. The REX prefix values are taken from the 16 single-byte INC register and DEC register instructions, one for each of the eight GPRs. 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).

Table 7. Special REX Encodings for Registers

ModRM and SIB Encodings	Meaning in Legacy and Compatibility Modes	Implications in Legacy and Compatibility Modes	Additional REX Implications
ModRM Byte: • mod ! = 11 • r/m[1] == b100 (ESP)	SIB byte is present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (B), which is not decoded (don't care). Therefore, SIB byte also required for R12-based addressing.
ModRM Byte: • mod == 00 • r/m[1] == b101 (EBP)	Base register is not used.	Using EBP without displacement must be done via mod = 01 with a displacement of 0.	REX prefix adds a fourth bit (B), which is not decoded (don't care). Therefore, using RBP or R13 without displacement must be done via mod = 01 with a displacement of 0.
SIB Byte: • index[1] == 0100 (ESP)	Index register is not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (X), which is decoded. Therefore, there are no additional implications. The expanded index field is used to distinguish RSP from R12, allowing R12 to be used as an index.
SIB Byte: • base[1] == 0101 (EBP)	Base register is unused if mod = 00.	Base register depends on mod encoding.	REX prefix adds a fourth bit (X), which is decoded. Therefore, there are no additional implications. The expanded base field is used to distinguish RBP from R13, allowing R13 to be used as a SIB base regardless of mod.

1. The REX-prefix bit is shown in the fourth (most-significant) bit position of the encodings for the ModRM *r/m*, SIB *index*, and SIB *base* fields. The lower-case "b" for ModRM *r/m* indicates that the bit is not decoded (don't care).

Displacement

Addressing in 64-bit mode uses the existing 32-bit ModRM and SIB encodings. In particular, the ModRM and SIB displacement sizes do not change. They remain either 8 or 32 bits and are sign-extended to 64 bits.

Direct Memory-Offset MOVs

In 64-bit mode, the direct memory-offset forms of the MOV instruction, shown in Table 8, are extended to specify a full 64-bit immediate absolute address, called a *moffset*.

Table 8. Direct Memory-Offset (moffset) Forms of MOV

Opcode	Instruction
A0	MOV AL,moffset
A1	MOV EAX,moffset
A2	MOV moffset,AL
A3	MOV moffset,EAX

No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default, which is 64 bits in 64-bit mode.

Immediates

In 64-bit mode, the maximum size of immediate operands remains 32 bits, except that 64-bit immediates can be moved into a 64-bit GPR.

When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to using them. Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions—opcodes B8h–BFh—move a 16-bit or 32-bit immediate (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits (i.e., in 64-bit mode) these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default to a 64-bit operand size. For example:

```
48 B8 8877665544332211  MOV RAX, 1122334455667788h
```

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.

In legacy x86 architecture, addressing relative to the instruction pointer is available only in control-transfer instructions. In the 64-bit mode, an instruction that uses ModRM addressing can use RIP-relative addressing. The feature is particularly useful for addressing data in position-independent code and for code that addresses global data.

Without RIP-relative addressing, ModRM instructions address memory relative to zero. With RIP-relative addressing, ModRM instructions can address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP.

Programs usually have many references to data, especially global data, that are not register-based. To load such a program, the loader typically selects a location for the program in memory and then adjusts the program's references to global data based on the load location. RIP-relative addressing of data makes this adjustment unnecessary.

Encoding. Table 9 shows the ModRM and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-only addressing exist in the current ModRM and SIB encodings. There is one ModRM encoding and several SIB encodings. RIP-relative addressing is encoded using one of the redundant forms. In 64-bit mode, the ModRM *Disp32* (32-bit displacement) encoding is re-defined to be *RIP+Disp32* rather than displacement-only.

Table 9. RIP-Relative Addressing Encoding

ModRM and SIB Encodings	Meaning in Legacy and Compatibility Modes	Meaning in 64-bit Mode	Additional 64-bit Implications
ModRM Byte: • mod = 00 • r/m = 101 (none)	Disp32	RIP + Disp32	Zero-based (normal) displacement addressing must use SIB form (see next row).
SIB Byte: • base = 101 (none) • index =100 (none) • scale =0, 1, 2, 4	If mod = 00, Disp32	Same as Legacy	None

Effect of REX Prefix on RIP-Relative Addressing. ModRM encoding for RIP-relative addressing does not depend on a REX prefix. In particular, the *r/m* encoding of 101, used to select RIP-relative addressing, is not affected by the REX prefix. For example, selecting R13 (REX.B=1, r/m=101) with mod=00 still results in RIP-relative addressing.

The 4-bit *r/m* field of ModRM is not fully decoded. Therefore, in order to address R13 with no displacement, software must encode it as R13+0 using a 1-byte displacement of zero.

Effect of Address-Size Prefix on RIP-relative addressing. RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. Conversely, use of the address-size prefix ("Address-Size and Operand-Size Prefixes" on page 17) does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits, like any other addressing mode.

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. See "Near Branches" on page 27 for details.
- All instructions, except far branches, that implicitly reference the RSP. See "Stack Pointer" on page 26 for details.

See Appendix A, "Integer Instructions in 64-Bit Mode", on page 73 for a complete list of the instructions affected and their opcodes.

Stack Pointer

In 64-bit mode, the stack pointer size 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.

Except far branches, all instructions, that implicitly reference the RSP default to 64-bit operand size in 64-bit mode. The instructions affected include PUSH, POP, PUSHF, POPF, ENTER, and LEAVE. Pushes and pops of 32-bit stack values are not possible in 64-bit mode with these instructions.

The 64-bit default operation-size eliminates the need for a REX prefix to precede these instructions when registers RAX–RSP are used as operands. A REX prefix is still required if the R8–R15 registers are used as operands, because the prefix is required to address the new extended registers.

Branches

The long-mode 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.

Near Branches. In 64-bit mode, the operand size for all near branches (CALL, RET, JCC, JCXZ, JMP, and LOOP) is forced to 64 bits. Therefore, these instructions update the full 64-bit RIP without the need for a REX operand-size prefix.

The following aspects of near branches are controlled by the effective operand size:

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

In 64-bit mode, all of the above actions are forced to 64 bits. However, the size of the displacement field for relative branches is still limited to 32 bits.

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.

Far Branches Through Long-Mode Call Gates. Software typically uses far branches to change privilege levels. The legacy x86 architecture provides the call-gate mechanism to allow software to branch from one privilege level to another, although call gates can also be used for branches that do not change privilege levels. When call gates are used, the selector portion of the direct or indirect pointer references the gate descriptor, and the offset portion is ignored. The offset into the destination's code segment is taken from the call-gate descriptor.

Long mode redefines the 32-bit call-gate descriptor type as a 64-bit call-gate descriptor and expands the descriptor's size to hold a 64-bit offset (see "Call Gates" on page 58). The long-mode call-gate descriptor allows far branches to reference any location in the supported virtual-address space. Long-mode call gates also hold the target code selector (CS), allowing changes to privilege level and default size as a result of the gate transition.

Stack Switches. For details on stack switches, see "Stack Switching" on page 68.

Branches to 64-Bit Offsets. Because immediates 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. Long mode expands the semantics of the SYSCALL and SYSRET instructions so that they specify a 64-bit code offset. Two such 64-bit offsets are defined—one for compatibility-mode callers and another for 64-bit-mode callers. See "System Target-Address Registers" on page 62 for system-level details.

The SYSCALL and SYSRET instructions were designed for operating systems that use a flat memory model, in which segmentation is disregarded. This makes them ideally suited for long mode. The memory model is flat in long mode, so these instructions simplify calls and returns by eliminating unneeded checks.

Software should not alter the CS or SS descriptors in a manner that violates the following assumptions made by the SYSCALL and SYSRET instructions:

- The CS and SS base and limit remain the same for all processes, including the operating system.
- The CS of the SYSCALL target has a privilege level of 0.
- The CS of the SYSRET target has a privilege level of 3.

SYSCALL and SYSRET do not check for violations of these assumptions. For details on operating system support of SYSCALL and SYSRET, see "SYSCALL and SYSRET" on page 61.

System Programming

Overview

This section is intended for programmers writing operating systems, loaders, linkers, device drivers, or utilities that require privileged instructions. It assumes an understanding of x86-64 architecture application-level programming that is described in "Application Programming" on page 9.

Additional system-programming information can be found in the following appendixes:

- Appendix A, "Integer Instructions in 64-Bit Mode", on page 73.
- Appendix B, "Long Mode Differences", on page 98.
- Appendix C, "Initialization Example", on page 100.
- Appendix D, "Implementation Considerations", on page 105.

Canonical Address Form

Long mode defines 64 bits of virtual address, but implementations can support less. The first implementation of AMD's family of processors that implement the x86-64 architecture (code named the "Hammer family) will support 48 bits of virtual address. Although implementations might not use all 64 bits of the virtual address, they will 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 said to be in *canonical address form*. If a virtual-memory reference is not in canonical form, the implementation generates a general-protection exception (#GP).

CPUID

The CPUID instruction reports the presence of processor features and capabilities. Typically, software loads a function code into the EAX register and executes the CPUID instruction. As a result of executing the CPUID instruction, processor feature information is returned in the EAX, EBX, ECX, and EDX registers.

8000_0000h. Executing CPUID with a function code of 8000_0000h returns a value in EAX that indicates the largest extended-function code recognized by the implementation. The largest extended-function code recognized in the first implementation of the Hammer family of processors is 8000_0008h.

8000_0001h. When the function code is 8000_0001h, the CPUID instruction returns the extended-feature flags in EAX. Table 10 shows the long-mode extended-feature flag.

Table 10. Extended-Feature Flags for First Implementation

Bit	Feature	
29	LM	Long Mode

8000_0008h. When the function code is 8000_0008h, the CPUID instruction returns the address-size information for long mode in the EAX register. Figure 6 shows the format. Registers EBX, ECX and EDX are reserved.

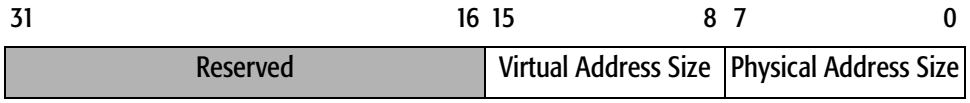


Figure 6. EAX Format Returned by Function 8000_0008h

The returned virtual-address and physical-address size indicate the address widths, in bits, supported by the implementation. In the first implementation of the Hammer family of processors, the supported virtual-address size is 48 bits (30h) and the physical-address size is 40 bits (28h). These address sizes apply to long mode only. The values returned by this extended-function code are not influenced by enabling or disabling either long mode or physical address extensions (PAE).

System Registers

The x86-64 architecture introduces changes to several system registers:

- **Model-Specific Registers (MSR)**. New control bits are added to the extended feature enable register (EFER). It contains control bits for enabling and disabling features of the x86-64 architecture.
- **Control Registers**. All control registers are expanded to 64 bits, and a new control register, the task priority register (CR8 or TPR) is added.
- **Descriptor Table Registers**. The global descriptor table register (GDTR) and interrupt descriptor table register (IDTR) are expanded to 10 bytes, to hold the full 64-bit base address.
- **Debug Registers.** All debug registers are expanded to 64 bits.

Extended Feature Enable Register (EFER)

The extended feature enable register (EFER) contains control bits that enable extended features of the processor. The EFER is an model-specific register (MSR) with an address of C000_0080h. It can be read and written only by privileged software.

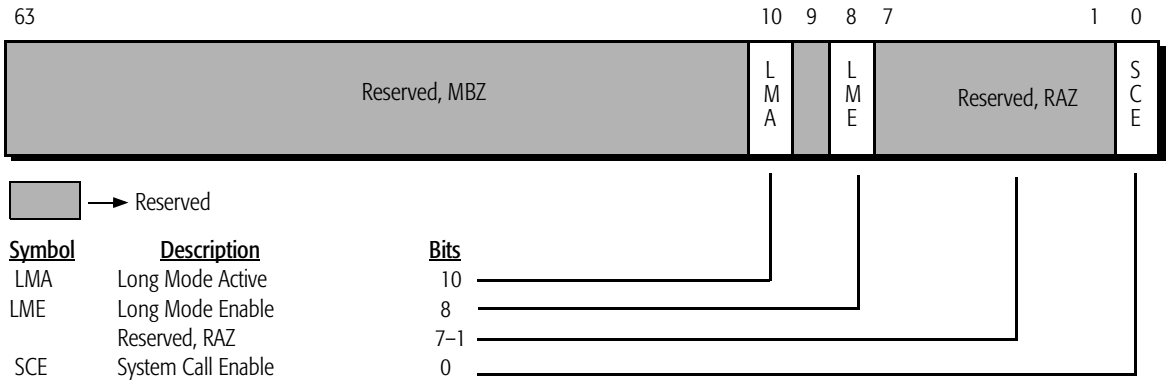


Figure 7. Extended Feature Enable Register (EFER)

Figure 7 shows the EFER register. The EFER bits are as follows:

SCE **System call extension (bit 0).** Setting this bit to 1 enables the SYSCALL and SYSRET instructions. Software can use these instructions for low-latency system calls and returns in non-segmented (flat address space) operating systems.

LMA **Long mode active (bit 10).** This bit is a read-only status bit indicating that long mode is active. The processor sets LMA to 1 when both long mode and paging have been enabled. See "Activating Long Mode" on page 36 for details.

When LMA=1, the processor is running either in compatibility mode or 64-bit mode, depending on the values of the code segment descriptor's L and D bits as shown in Table 11 on page 35.

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.

LME **Long mode enable (bit 8).** Setting this bit to 1 enables the processor to switch to long mode. Long mode is not activated until software enables paging some time later. When paging is enabled while 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.

All other EFER bits are reserved and must be written with zeros (MBZ).

Control Registers

Control registers CR0-CR4 are expanded to 64 bits in the x86-64 architecture hardware. 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 upper 32 bits with zeros and control register reads return only the lower 32 bits.

In 64-bit mode, the upper 32 bits of CR0 and CR4 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). All 64 bits of CR2 and CR3 are writable by software. However, the MOV CR*n* instructions do not check that addresses written to CR2 or CR3 are within the virtual-address or physical-address limitations of the implementation.

Task Priority Register (TPR). The x86-64 architecture introduces a new control register, CR8, defined as the task priority register (TPR). Operating systems can use the TPR to control whether or not external interrupts are allowed to interrupt the processor, based on the interrupt's priority level.

Figure 8 shows the TPR. Only the low four bits are used. The remaining 60 bits are reserved and must be written with zeros.

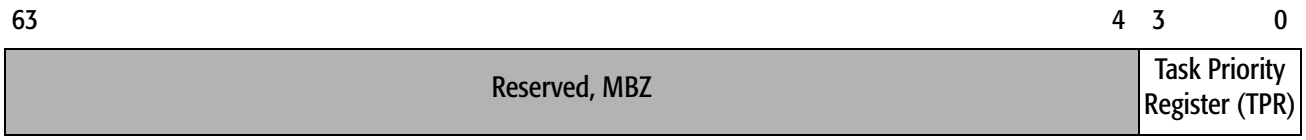


Figure 8. Task Priority Register (CR8)

For details on how the operating system can use the TPR, see "Task Priority" on page 70.

Descriptor Table Registers

The four system-descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded in hardware to hold 64-bit base addresses. This allows operating systems running in long mode to locate system-descriptor tables anywhere in the virtual-address space supported by the implementation.

Figure 9 shows the GDTR and IDTR. Figure 10 shows the LDTR and TR. In all cases, only 48 bits of base address are supported in the first implementation of the Hammer family of processors.

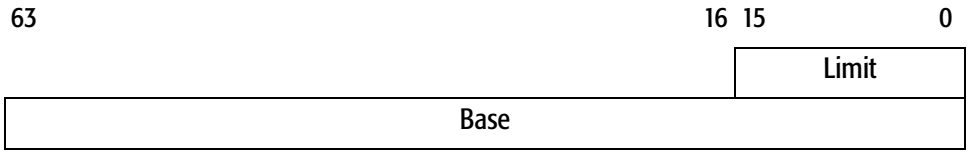


Figure 9. GDTR and IDTR

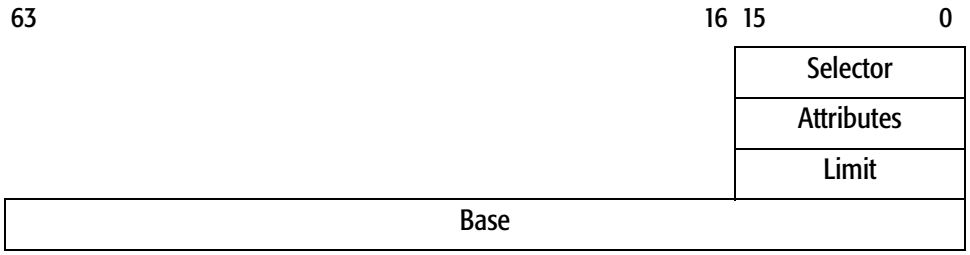


Figure 10. LDTR and TR

For details on how the operating system uses system-descriptor-tables, see "System Descriptors" on page 44.

Debug Registers

Like the control registers, debug registers DR0–DR7 are expanded to 64 bits. In 64-bit mode, the MOV DR*n* instructions read or write all 64 register bits. Operand-size prefixes are ignored.

In all 16-bit or 32-bit modes (legacy or compatibility modes), writes to a debug register fill the upper 32 bits with zeros and reads from a debug register return only the lower 32 bits.

In 64-bit mode, the upper 32 bits of DR6 and DR7 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). All 64 bits of DR0–DR3 are writable by software. However, the MOV DR*n* instructions do not check that addresses written to DR0–DR3 are within the virtual-address limits of the implementation.

Enabling and Activating Long Mode

Table 11 shows the control-bit settings for enabling and activating the various operating modes of the x86-64 architecture. The default address and data sizes are shown for each mode. For the methods of overriding these default address and data sizes, see "Address-Size and Operand-Size Prefixes" on page 17.

Table 11. Processor Modes

Mode		Encoding			Default Address Size (bits)[2]	Default Data Size (bits)[3]
		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

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 36.
2. See Table 4 on page 17 for overrides to default address sizes.
3. See Table 5 on page 18 for overrides to default operand sizes.

Processor Modes

Long mode uses two code-segment-descriptor bits, CS.L and CS.D, to control the operating submodes. See "Code Segments" on page 40 for details of code-segment attributes and overrides in long mode.

If long mode is active, CS.L = 1, and CS.D = 0, the processor is running in 64-bit mode, as shown in Table 11. 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 second encoding of CS.L and CS.D in 64-bit-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 11. In this 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.

Activating Long Mode

Long mode is *enabled* by setting the EFER.LME bit to 1 (see Figure 7 on page 32 for this bit). However, long mode is not *activated* until software also enables paging. When software enables paging, the processor activates long mode and signals this by setting the EFER.LMA status bit (long mode active) to 1.

Long mode requires the use of physical-address extensions (PAE) in order to support physical-address sizes greater than 32 bits. Software enables physical-address extensions by setting the CR4.PAE bit to 1. Physical-address extensions must be enabled prior to enabling paging (CR0.PG=1).

Switching the processor to long mode requires several steps. The process must start the switch in real mode or non-paged (CR0.PG=0) protected mode. Software must follow this general 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).
2. In any order:
   - Enable physical-address extensions by setting CR4.PAE to 1.
   - Load CR3 with the physical base address of the level-4 page map table (PML4). See "PML4" on page 48 for details.
   - 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 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.

To return from long mode to legacy paged-protected mode, software must deactivate and disable long mode using the following general sequence:

1. 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.
2. Load CR3 with the physical base address of the legacy page-table-directory base address.
3. Disable long mode by clearing EFER.LME to 0.
4. Enable legacy paged-protected mode 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.

Throughout this document, the phrase *in long mode* means that long mode is both enabled and active.

System Descriptor Table Considerations. Immediately after activating long mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy descriptor tables. The tables referenced by the descriptors all reside in the lower 4GBytes 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 "System Descriptors" on page 44 for details on descriptor tables in long mode.

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 interrupt-descriptor table (IDT). This is because the IDT remains in its legacy form immediately after long mode is activated. Long mode requires that 64-bit interrupt gate descriptors be stored in the IDT. If an interrupt or exception occurred prior to 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) must be disabled using external hardware. See "Interrupts" on page 65 for details on interrupts in long mode.

Long-Mode Page Table Considerations. The long-mode paging tables must be located in the first 4GBytes of physical-address space prior to activating long mode. This is necessary because the MOV CR3 instruction used to initialize the page-directory base must be executed in legacy mode prior to activating long mode (setting CR0.PG to 1 to enable paging). Because the MOV CR3 is executed in legacy mode, only the low 32 bits of the register are written, limiting the table location to the low 4GBytes of memory. Software can relocate the page tables anywhere in physical memory after long mode is activated.

Consistency Checks. The processor performs long-mode consistency checks whenever software attempts to modify any of the enable bits directly involved in activating long mode (EFER.LME, CR0.PG, and CR4.PAE). The processor generates a general protection fault (#GP) when a consistency check fails. Long-mode consistency checks ensure that the processor does not enter an undefined mode or state with unpredictable behavior.

Long-mode consistency checks fail in the following circumstances:

- 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 prior to enabling physical-address extensions (PAE).
- Long mode is active and an attempt is made to disable physical-address extensions (PAE).

Table 12 summarizes the long-mode consistency checks.

Table 12. Long Mode Consistency Checks

Register	Bit	Check
EFER	LME 0->1	if (CR0.PG=1) then #GP(0)
	LME 1->0	if (CR0.PG=1) then #GP(0)
CR0	PG 0->1	if ((EFER.LME=1) & (CR4.PAE=0)) then #GP(0)
CR4	PAE 1->0	if (EFER.LMA=1) then #GP(0)

Virtual-8086 Mode

The legacy virtual-8086 mode allows an operating system to run 16-bit real-mode software on a virtualized 8086 processor. Virtual-8086 mode is not supported when the processor is operating in long mode. When long mode is enabled, any attempt to set the EFLAGS.VM bit is silently ignored.

Compatibility Mode: Support for Legacy Applications

Compatibility mode, within long mode, maintains binary compatibility with legacy x86 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 operating system along with 64-bit applications running in 64-bit mode. An operating system running in long mode can execute existing 16-bit and 32-bit applications by clearing their code-segment descriptor's CS.L bit to 0.

When the CS.L bit is cleared to 0, the legacy x86 meanings of the CS.D bit and the address-size and operand-size prefixes are observed, and segmentation is enabled. From the application's viewpoint, the processor is in a legacy 16-bit or 32-bit (depending on CS.D) operating environment, even though long mode is activated.

Long-Mode Semantics

In compatibility mode, the following two system-level mechanisms continue to operate using the long-mode architectural semantics:

- Virtual-to-physical address translation uses the long-mode extended page-translation mechanism.
- Interrupts and exceptions are handled using the long-mode mechanisms.
- System calls (calls through call gates and SYSCALL/SYSRET) are handled using the long mode mechanisms.

Switching Between 64-Bit Mode and Compatibility Mode

The processor does not preserve the upper 32 bits of the 64-bit GPRs across switches from 64-bit mode to compatibility or legacy modes. In compatibility or legacy mode, the upper 32 bits of the GPRs are undefined and not accessible to software.

See "Registers" on page 7 for details on the extended registers introduced by the long-mode architecture.

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.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit virtual-address space. 64-bit mode treats the segment base as a zero, creating a logical address that is equivalent to the virtual (or linear) address. The exceptions are the FS and GS segments, whose segment registers (which hold the segment base) can be used as an additional base register in address calculations. This facilitates addressing thread-local data and certain operating-system data structures. See "Special Treatment of FS and GS Segments" on page 42 for details about the FS and GS segments in 64-bit mode.

Code Segments

Code segments continue to exist in 64-bit mode. Code segments and their associated descriptors and selectors are needed to establish the processor's operating mode as well as execution privilege-level.

Most of the code-segment (CS) descriptor content is ignored in 64-bit mode. Only the L (long), D (default operation size), and DPL (descriptor privilege level) attributes are used by long mode. All remaining CS attributes are ignored, as are the CS base and limit fields. For address calculations in 64-bit mode, the segment base is treated as if it is zero.

Impacts on Segment Attributes. Long mode uses a previously unused bit in the CS descriptor. Bit 53 is defined as the long (L) bit and is used to select between 64-bit and compatibility modes when long mode is activated (EFER.LMA=1). Figure 11 shows a legacy CS descriptor with the L bit added.

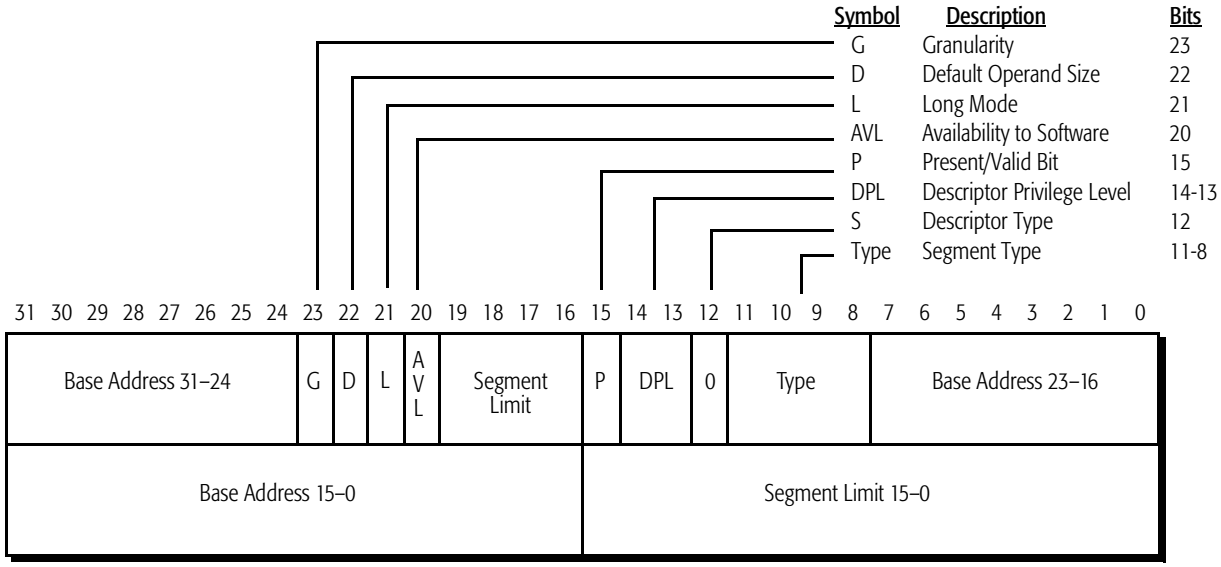


Figure 11. Code Segment Descriptor

The CS descriptor's D bit selects the default operand and address sizes. When the CS.L bit is set to 1, the only valid setting of CS.D is cleared to 0. This setting produces a default operand size of 32 bits and a default address size of 64 bits. The combination CS.L=1 and CS.D=1 is reserved for future use.

If CS.L is cleared to 0 while long mode is activated, the processor is running in compatibility mode. In this case, CS.D selects the default size for both data and addresses as it does in legacy mode. If CS.D is cleared to 0, the default data and address sizes are 16 bits, whereas setting CS.D to 1 selects a default data and address size of 32 bits.

Table 11 on page 35 shows the effect of CS.L and CS.D 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, as described in "Address-Size and Operand-Size Prefixes" on page 17 and "REX Prefixes" on page 18.

In long mode, the CS descriptor's DPL is used for execution privilege checks just as in legacy mode.

Data and Stack Segments

In 64-bit mode, the contents of the ES, DS, and SS segment registers are ignored. All fields (base, limit, and attribute) in the corresponding segment descriptor registers (hidden part) are also ignored.

Address calculations in 64-bit mode that reference the ES, DS, or SS segments, are treated as if the segment base is zero. Rather than perform limit checks, the processor instead checks that all virtual-address references are in canonical form.

Neither enabling and activating long mode or switching between 64-bit and compatibility modes changes the contents of the segment registers or the associated descriptor registers. These registers are also not changed during 64-bit mode execution, unless explicit segment loads are performed.

Segment Loads. Segment-load instructions (MOV to Sreg, POP Sreg) 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. The descriptor-register base, limit, and attribute fields are all loaded. However, the contents of data and stack segment selector and descriptor registers are ignored.

The ability to use segment-load instructions allows a 64-bit operating system to set up a compatibility-mode application's segment registers prior to switching to compatibility mode.

Special Treatment of FS and GS Segments. The FS and GS segment registers are used by the Windows NT™ operating system to locate the thread-environment-block (TEB) and processor-control-region (PCR) data structures. The FS and GS segment-override prefixes provide quick access to these data structures in an otherwise unsegmented (flat address space) operating system. To facilitate compatible access to these structures, 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 + index + displacement.

In 64-bit mode, FS.base and GS.base are expanded to the full virtual-address size supported by the implementation. The resultant EA calculation is allowed to wrap across positive and negative addresses.

In 64-bit mode, FS-segment and GS-segment overrides are not checked for limit or attributes.

Normal segment loads (MOV to Sreg and POP Sreg) into FS and GS only load a standard 32-bit base value into the hidden portion of the segment descriptor register. The base address bits above the standard 32 bits are cleared to 0. Because the first implementation of the Hammer family of processors supports 48 virtual-address bits, a segment-load instruction loads the base value into the lower 32 address bits and clears the high 16 bits to 0.

To load all address bits supported by a 64-bit implementation, the FS.base and GS.base hidden descriptor register fields are physically mapped to MSRs. Privileged software (CPL=0) can load all supported virtual-address bits into FS.base or GS.base using a single WRMSR instruction. The FS.base MSR index is C000_0100h while the GS.base index is C000_0101h.

The addresses written into the expanded FS.base and GS.base registers must be in canonical form. A WRMSR instruction that attempts to write a non-canonical address to those registers generates a general-protection exception, #GP.

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

System Descriptors

In certain modes, system descriptors are expanded by 64 bits to handle 64-bit base addresses (with additional room to spare). The mode in which this size-expansion occurs depends on the purpose served by the descriptor, as follows:

- Expansion Only In 64-Bit Mode: Descriptors and pseudo-descriptors that are loaded into the GDTR, IDTR, LDTR, and TR registers are used to define system tables. These descriptors are expanded only in 64-bit mode but not in compatibility mode. For example, see "LDT and TSS Descriptors" on page 45.
- Expansion In Long Mode: Descriptors that populate system tables and are actually referenced by application programs are expanded in long mode (both 64-bit mode and compatibility mode). These descriptors include call gates, interrupt gates, and trap gates. (Task gates are not used in long mode.) For example, see "Call Gates" on page 58.

Descriptor-Table Base Registers. The GDTR, LDTR, IDTR, and TR system descriptor registers are used by the processor to locate the GDT, LDT, and IDT system-descriptor tables and to locate the task state segment (TSS) of the current process. These system-descriptor registers are changed by long mode to support the expanded memory addressing. See "Descriptor Table Registers" on page 34 for details of the long mode changes.

Descriptor Tables. The base address of the LDT and TSS are specified by their associated descriptors. In order to hold 64-bit bases, the LDT and TSS descriptors are expanded, as described in "LDT and TSS Descriptors" on page 45. The GDT and IDT, on the other hand, do not have descriptors. Instead, their bases are loaded from the operands of the LGDT and LIDT instructions, as described "LGDT and LIDT Instructions" on page 45. For 64-bit mode (but not compatibility or legacy modes), the size of the operands for these instructions is increased to specify a 64-bit base.

The processor checks descriptor-table limits in long mode. The limit-field sizes in all four descriptor-table registers are unchanged from their legacy x86 sizes, because the offsets into the descriptor tables are not extended in long mode. Thus, the GDTR and IDTR limits remain 16 bits, and the LDTR and TR limits remain 32 bits.

The size of the segment-attribute fields in the LDTR and TR registers are also unchanged in long mode. The existing LDT type field (02h) and the existing 32-bit TSS type field (09h) are redefined in 64-bit mode for use as the 64-bit LDT and TSS types.

LDT and TSS Descriptors. In 64-bit mode, the LDT and TSS system descriptors are expanded by 64 bits, as shown in Figure 12. This allows them to hold 64-bit base addresses.

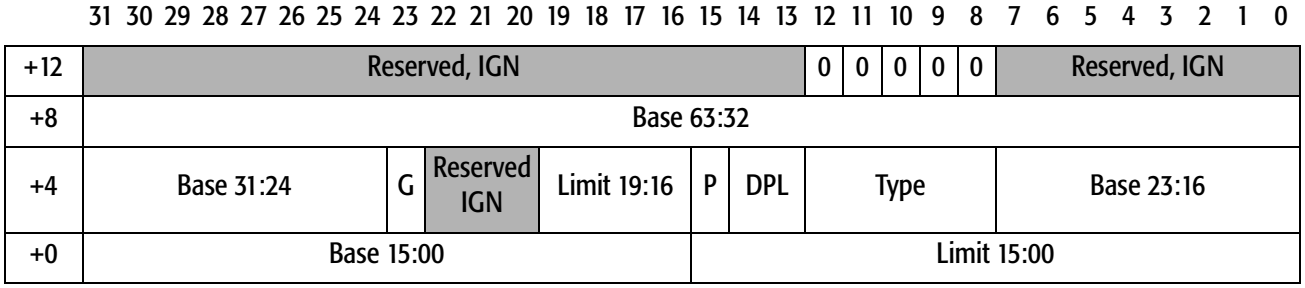


Figure 12. LDT and TSS Descriptors in 64-Bit Mode

Bytes 11:8 hold the upper 32 bits of the base address in canonical form. A second type field, used for consistency checking, is defined in bits 12:8 of the highest dword. This entire field must be cleared to 0, indicating an illegal type. This illegal type (00h) serves to generate a general-protection exception (#GP) if an attempt is made to access the upper half of a 64-bit-mode descriptor as a legacy x86 descriptor.

The existing LDT type field (02h) and 32-bit TSS type field (09h) are redefined in 64-bit mode for use as the 64-bit LDT and 64-bit TSS types. In compatibility mode, a 02h type continues to refer to a 32-bit LDT, and a 09h type continues to refer to a 32-bit TSS. No other type-field codes are defined or redefined.

The 64-bit base address specified in the descriptor must be in canonical form. If it is not, a general-protection exception, #GP(selector), is generated.

LGDT and LIDT Instructions. These instructions load a pseudo-descriptor into the GDTR or the IDTR register. The first two bytes loaded in all modes (legacy, compatibility, and 64-bit) are a 16-bit limit. The next bytes loaded depend on the mode, as follows:

- In any 16-bit or 32-bit mode (legacy or compatibility mode), the next 4 bytes loaded are the base, for a total of 6 bytes.
- In 64-bit mode, the next 8 bytes loaded are the base, for a total of 10 bytes.

Operand-size prefixes are ignored by the LGDT and LIDT instructions. In 64-bit mode, the 64-bit base address loaded into the GDTR and IDTR registers must be in canonical form, otherwise a general-protection exception, #GP(selector), is generated.

LLDT and LTR Instructions. These instructions load a system descriptor into the processor's internal LDTR and TR segment descriptor registers (hidden portion). In 64-bit mode, the expanded descriptor format and redefined descriptor types give rise to the following restrictions on the descriptors that these instructions can load:

- The 64-bit base address loaded by an LLDT or LTR must be in canonical form, otherwise a general-protection exception, #GP(selector), is generated.
- A general-protection exception, #GP(selector), is generated if an attempt is made to load the second type field (bits 12:8 of the highest dword) with a value other than 00h.
- A general-protection exception, #GP(selector), is generated if an LTR instruction references either a busy or 16-bit TSS.

In 64-bit mode, the LTR instruction still changes a task's state to busy (descriptor type set to 0Bh). Because long mode does not support task switches, a task descriptor's busy bit is never automatically cleared. If the operating system has previously loaded the task descriptor using the LTR instruction, the operating system is responsible for clearing the task's busy bit (setting descriptor type to 09h).

Virtual Addressing and Paging

In long mode, a virtual address is a uniform 64-bit offset into the virtual address space. The mechanisms that translate virtual addresses into physical addresses are changed to support the larger virtual-address size.

Virtual-Address and Physical-Address Size

The long-mode architecture provides for 64 bits of virtual-address space and 52 bits of physical-address space. The maximum supported virtual-address space is 2^{64} bytes (16 exabytes) while the maximum supported physical-address space is 2^{52} bytes (4 petabytes).

Implementation Specifics. Implementations can support smaller virtual-address and physical-address spaces than the maximums defined by the long-mode architecture.

The first implementation of the Hammer family of processors supports 48 bits of virtual address and 40 bits of physical address. The CPUID instruction returns the number of virtual-address and physical-address bits supported by the implementation. See "CPUID" on page 30 for details.

Canonical Address Form. The long-mode 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).

Checking canonical address form prevents software from exploiting upper unused bits of pointers for other purposes. Software complying with canonical-address form on a specific implementation can run unchanged on long-mode implementations supporting larger virtual-address spaces.

In the first implementation of the Hammer family of processors, canonical address form checking is performed on virtual address bits [63:47].

Paging Data Structures

The long-mode architecture expands the physical address extension (PAE) paging structures to support mapping a 64-bit virtual address into a 52-bit physical address. In the first implementation of the Hammer family of processors, the PAE

paging structures are extended to support translation of a 48-bit virtual address into a 40-bit physical address.

Physical-Address Extensions. Prior to activating long mode, PAE must be enabled by setting CR4.PAE to 1. PAE expands the size of an individual page-directory entry (PDE) and page-table entry (PTE) from 32 bits to 64 bits, allowing physical-address sizes of greater than 32 bits. Activating long mode prior to enabling PAE results in a general-protection exception (#GP).

PML4. The long-mode architecture adds a new table, called the page map level 4 (PML4) table, to the virtual-address translation hierarchy. The PML4 table sits above the page directory pointer (PDP) table in the page-translation hierarchy. The PML4 contains 512 eight-byte entries, with each entry pointing to a PDP table. Nine virtual-address bits are used to index into the PML4.

PML4 tables are used in page translation only when long mode is activated. They are not used when long mode is disabled, regardless of whether or not PAE is enabled.

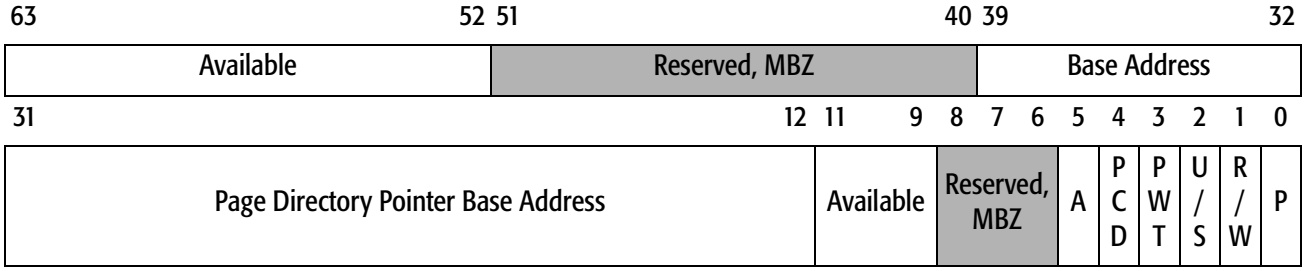
PDP. The existing page-directory pointer table is expanded by the long-mode architecture to 512 eight-byte entries from four entries. As a result, nine bits of the virtual address are used to index into a PDP table rather than two bits.

PDE, PTE, and Page Offsets. The size of both page-directory entry (PDE) tables and page-table entry (PTE) tables remains 512 eight-byte entries, each indexed by nine virtual-address bits.

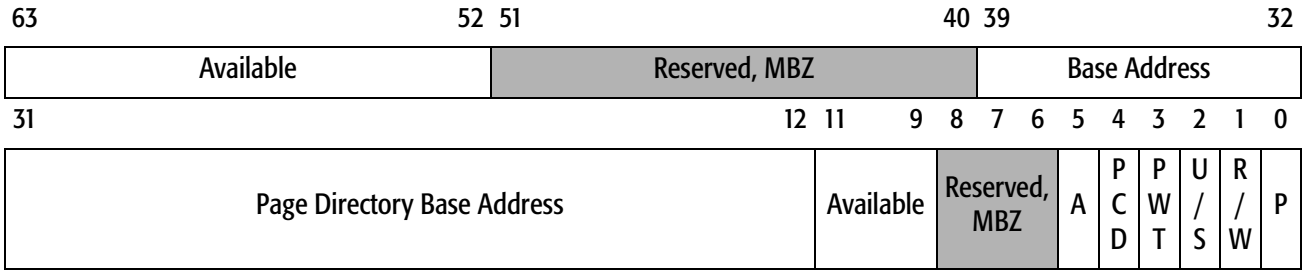
The total of virtual-address index bits into the collection of paging data structures (PML4 + PDP + PDE + PTE + page offset) defined above is 48 (9+9+9+9+12). The method for translating the high-order 16 virtual-address bits into a physical address is currently reserved.

Large Page Sizes. The PS flag in the page directory entry (PDE.PS) selects between 4KByte and 2MByte page sizes. Because PDE.PS is used to control large page selection, the CR4.PSE bit is ignored.

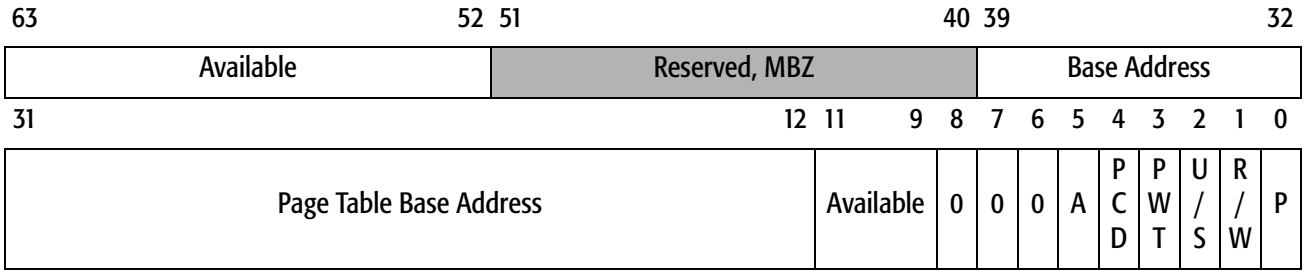
Page Table Formats for 4K Page Size. Figure 13 shows the long-mode PML4, PDP, PDE, and PTE formats when 4KByte pages are enabled.



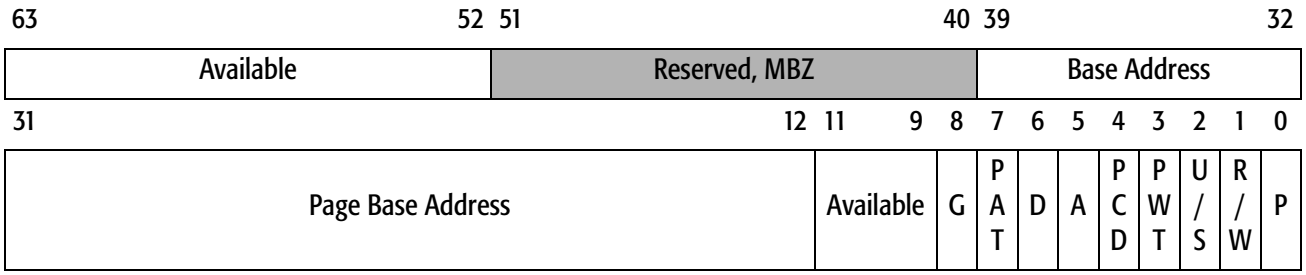
Page Map Level 4 Entry (PML4)



Page Directory Pointer Table Entry



Page Directory Entry (PDE)



Page Table Entry (PTE)

Figure 13. Long-Mode Page Table Formats (4KByte Pages)

The physical base-address field in all four table entry formats is extended by the long-mode architecture to bits 51:12. This allows paging tables to be located anywhere in the physical memory supported by a long-mode implementation. Implementations that do not support the maximum physical-address size reserve the unsupported high-order bits and require that they be cleared to zeros. The physical base-address field in the first implementation of the Hammer family of processors is specified by bits 39:12.

Bits 63:52 in all page-table entry formats are available for use by system software. In the long-mode architecture, future implementations leave bits 63:52 available for software use.

Other than the extensions made to the base-address field and the addition of the software-available field at bits 63:52, all other PDE and PTE fields are the same as in legacy mode.

PDP Table-Entry Exceptions for 4K Page Size. Fields within the PDP table entry are similar to legacy mode PDP table entries, with the following exceptions. The exceptions reflect changes necessary to indicate that a higher-level paging structure (PML4) now references the PDP tables:

- Bit 0 is no longer reserved. Long mode defines this bit as the present (P) flag to indicate whether or not the PDE table referenced by the PDP entry is currently stored in physical memory. A page-fault exception (#PF) is generated when the processor accesses a PDP entry with the P flag cleared to 0.
- Bit 1 is no longer reserved. Long mode defines this bit as the read/write (R/W) flag.
- Bit 2 is no longer reserved. Long mode defines this bit as the user/supervisor (U/S) flag.
- Bit 5 is no longer reserved. Long mode defines this bit as the accessed (A) flag.
- The base-address field extensions, as specified above.
- Bits 63:52 available to software, as specified above.

The format of a PML4 table entry is identical to the long-mode PDP table-entry format.

Page Table Formats for 2M Page Size. Figure 14 shows the long-mode PML4, PDP, and PDE formats when 2MByte pages are enabled. As with legacy mode, 2MByte pages are enabled by setting the PDE page-size bit (PDE.PS) to 1. Control of 2M page sizes is not dependent on CR4.PSE.

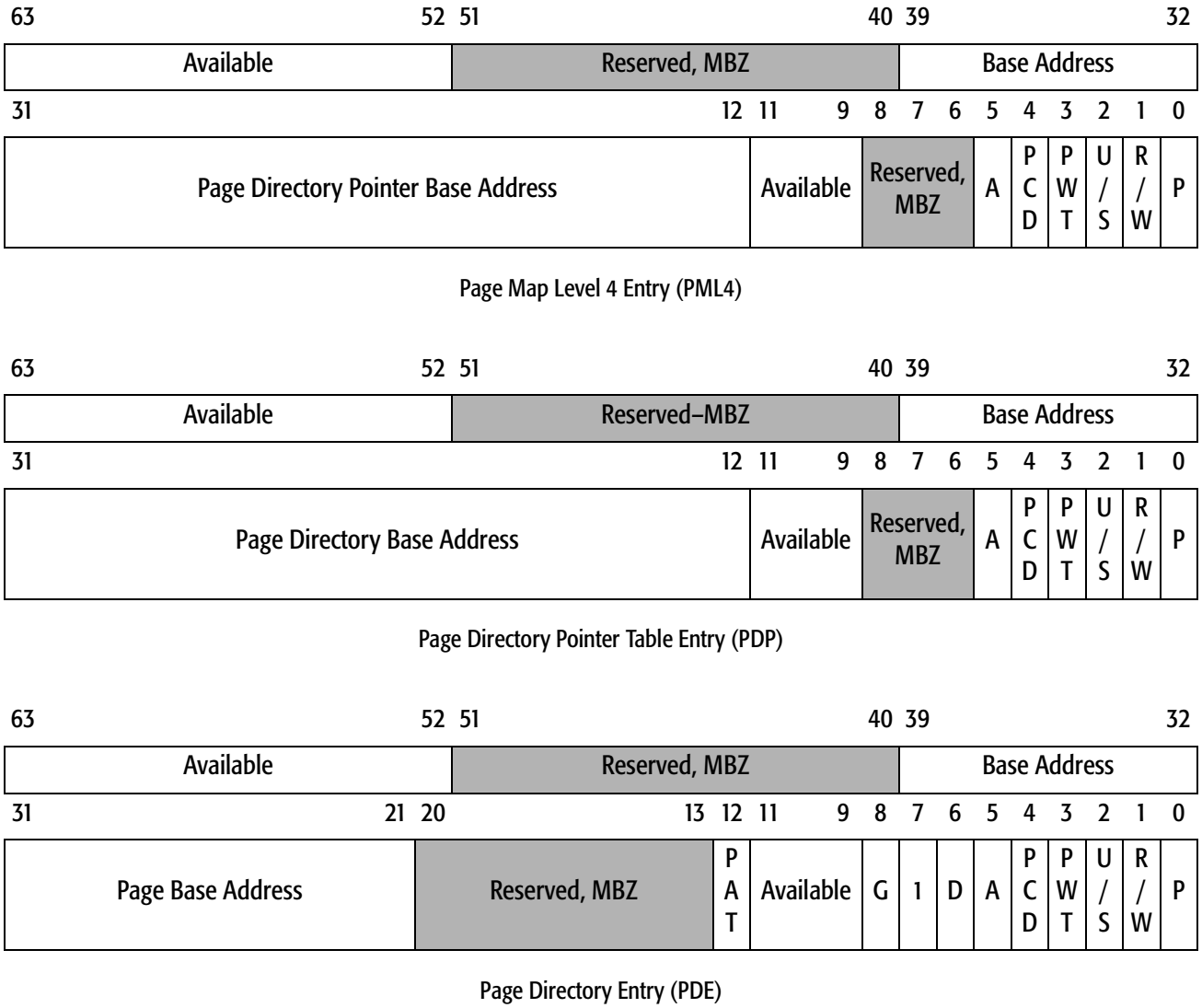


Figure 14. Long-Mode Page Table Formats (2MByte Pages)

The physical base-address field in all three table-entry formats is extended by the long-mode architecture to bits 51:12. This allows paging tables to be located anywhere in the physical memory supported by a long-mode implementation. Implementations that do not support the maximum physical-address size reserve the unsupported high-order bits and require that they be cleared to zeros. The physical base-address field in the first implementation of the Hammer family of processors is specified by bits 39:12.

Bits 63:52 in all page-table entry formats are available for use by system software. In the long-mode architecture, future implementations will leave bits 63:52 available for software use.

When 2MByte pages are selected, the PDE points directly to the physical page, and not to a PTE. Other than the extensions made to the base-address field and the addition of the software-available field at bits 63:52, all other PDE fields are the same as in legacy mode.

PDP Table-Entry Exceptions for 2M Page Size. Fields within the PDP table entry are similar to legacy-mode PDP table entries, with the following exceptions. The exceptions reflect changes necessary to indicate that a higher-level paging structure (PML4) now references the PDP tables:

- Bit 0 is no longer reserved. Long mode defines this bit as the present (P) flag to indicate whether or not the PDE table referenced by the PDP entry is currently stored in physical memory. A page-fault exception (#PF) is generated when the processor accesses a PDP entry with the P flag cleared to 0.
- Bit 1 is no longer reserved. Long mode defines this bit as the read/write (R/W) flag.
- Bit 2 is no longer reserved. Long mode defines this bit as the user/supervisor (U/S) flag.
- Bit 5 is no longer reserved. Long mode defines this bit as the accessed (A) flag.
- The base-address field extensions, as specified above.
- Bits 63:52 available to software, as specified above.

The format of a PML4 table entry is identical to the long-mode PDP table-entry format.

Enhanced Legacy-Mode Paging

Some changes made to legacy x86 paging data structures to support the larger physical address sizes used in long mode are also available in legacy mode. Specifically, 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.

PAE in Legacy Mode. As described in "Physical-Address Extensions" on page 48, setting CR4.PAE to 1 expands the size of an individual PDE and PTE from 32 bits to 64 bits, allowing physical-address sizes of greater than 32 bits. Previous legacy x86 implementations limit the larger physical-address size to 36-bits.

The x86-64 architecture allows legacy-mode software to load up to 52-bit physical addresses into the PDE and PTE, as limited by the maximum physical-address size supported by a specific implementation. Unsupported physical-address bits are reserved and must be cleared to zero. In the first implementation of the Hammer family of processors legacy-mode software can use up to 40 bits of physical address in PDE and PTE entries. Software must clear bits 51:40 to 0.

The long mode architecture defines PDE and PTE bits 63:52 as available to operating system software. Implementations of the x86-64 architecture make those same PDE and PTE bits available to legacy-mode software.

PSE in Legacy Mode. Legacy-mode page-size extensions (PSE) are enabled by setting the page-size enable bit in CR4 (CR4.PSE) to 1. PSE modifies the original 4-byte PDE format to support 4MByte pages in addition to legacy 4KByte pages. 4MByte pages are selected by setting the PDE page size bit (PDE.PS) to 1 while clearing PDE.PS to 0 selects 4KByte pages.

When PDE.PS=1, the processor combines PDE bits 31:22 with virtual address bits 21:0 to form a 32-bit physical address into a 4MByte page. Legacy PTEs are not used in a 4MByte page translation. Because the PTEs are not used, PDE bits 21:12 are reserved in the original PSE mode definition.

Updates to PSE mode change the 4-byte PDE format to also support 36-bit physical addresses without requiring the 8-byte format used by PAE. This is accomplished by using previously

reserved PDE bits 16:13 to hold four additional high-order physical address bits. Bits 21:17 are reserved.

The x86-64 architecture further modifies the 4-byte PDE format in PSE mode to increase physical address size support to 40 bits. This is accomplished by defining previously reserved PDE bits 20:17 to hold four additional high-order physical address bits. Bit 21 is reserved and must be cleared to 0.

Figure 15 shows the format of the PDE when PSE mode is enabled. The high-order physical address bits 39:32 are located in PDE[20:13] while physical address bits 31:22 are located in PDE[31:22].

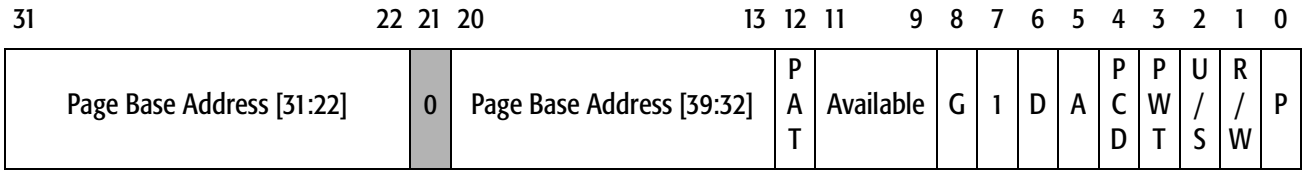


Figure 15. Legacy-Mode Page-Directory Entry for 4MByte Pages

CR2 and CR3

The size of CR2 (page-fault address register) is increased to 64 bits by the long-mode architecture to hold 64-bit virtual addresses.

The long-mode architecture also increases the size of CR3 (page-directory base register) to 64 bits. This allows the first level of the paging structures to be located anywhere in physical memory, subject to the implementation-dependent physical-address size limits.

Figure 16 shows the long-mode format of CR3. The Base Address field specifies the most-significant bits of the page-directory base address above bit 11. The Page-Directory Base field holds the most-significant physical-address bits of the top-level paging structure. Bits 51:12 of CR3 define the maximum base address allowed by the long-mode architecture, but specific implementations can support smaller physical-address spaces. The lower 12 bits (11:0) of the base address are always assumed to be zero, forcing the top-level paging structure to be aligned on a 4KByte boundary. CR3[39:12] specify the top-level paging-structure (PML4) base address in the first implementation of the Hammer family of processors, indicating that 40 bits of physical-address space are supported. CR3[63:40] are reserved and must be cleared to zero.

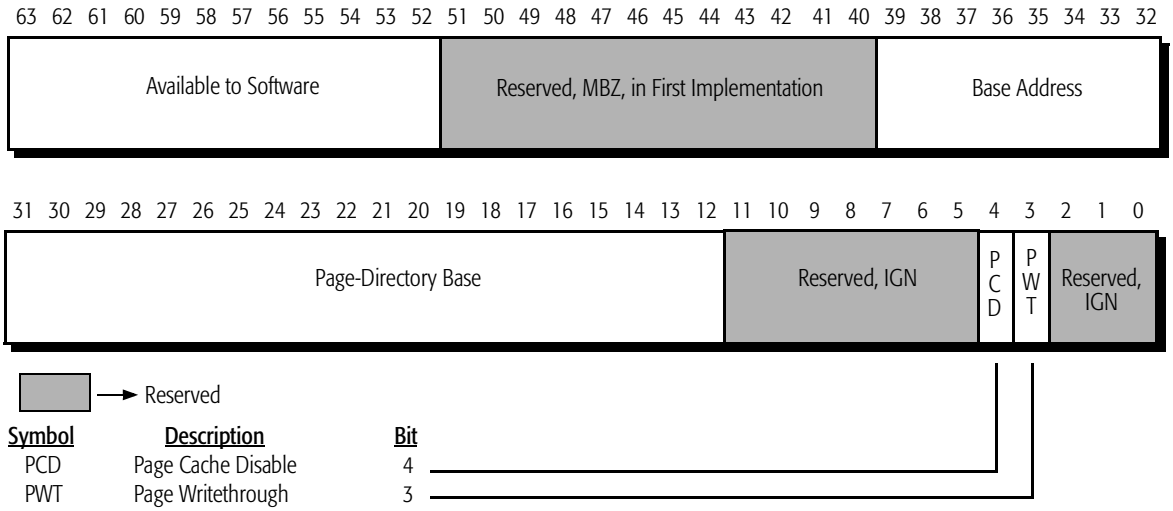


Figure 16. Long-Mode Control Register 3 (CR3)

CR3 bits [63:52] are available to software. They are read/write are not used by the processor. CR3[51:40] are reserved for future expansion of the page-directory base address. In the first implementation of the Hammer family of processors, the processor checks that these bits are written as zeros and generates a general-protection exception, #GP(0), if they are not.

The MOV to CR3 instruction is not affected by operand size in long mode. In 64-bit mode, all 64 bits of CR3 are loaded from the source register. In compatibility mode, only the lower 32 bits of CR3 are loaded from the source register and the upper 32 bits are cleared to 0.

Address Translation

When paging is used in long mode, the processor divides the virtual address into a collection of table and physical-page offsets, much like in legacy mode. However, the long-mode architecture extends how the processor divides the virtual address to support the 64-bit virtual-address size and deeper paging data-structure hierarchy.

4KB Pages. 4KByte pages are enabled by clearing the PDE page-size flag (PDE.PS) to 0. Because the first implementation of the Hammer family of processors supports a maximum 48 bits of virtual address, this paging option supports 2^{36} 4KByte pages spanning a virtual-address space of 2^{48} bytes (256 terabytes).

The 48-bit virtual address is broken into five fields to index into the 4-level paging structure, as follows and in Figure 17:

- 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.

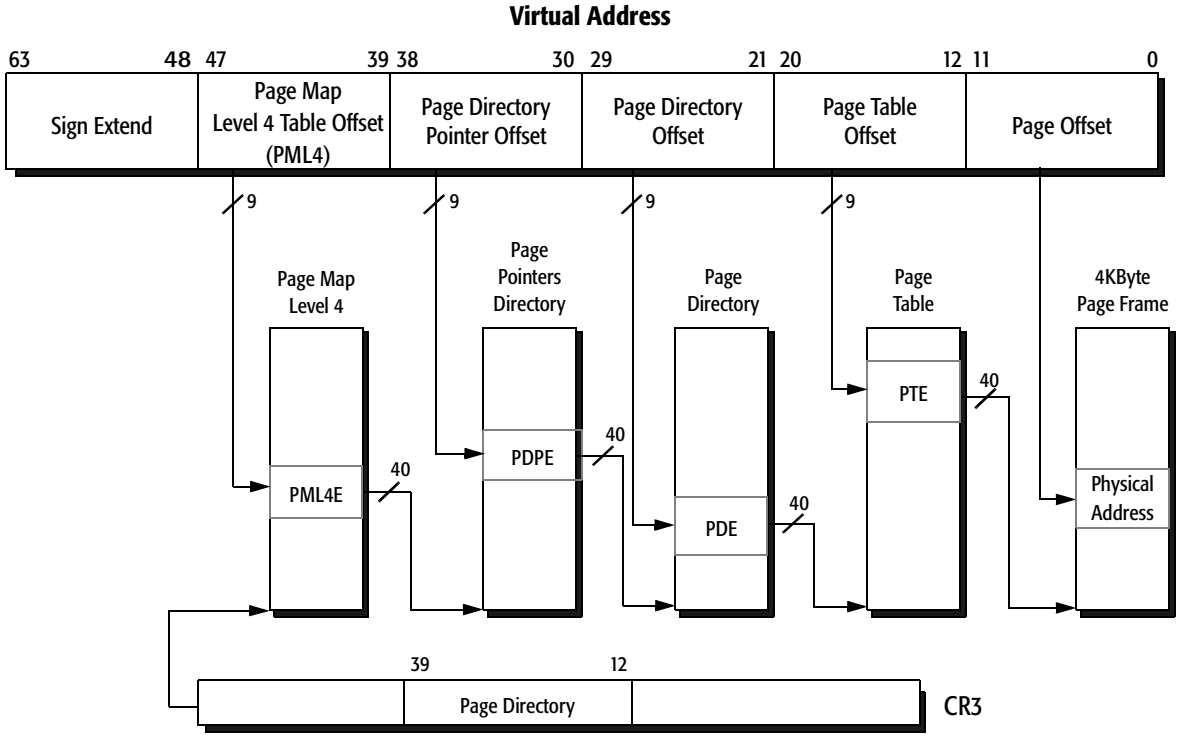


Figure 17. 4KB-Page Translation

2MB Pages. 2MByte pages are enabled by setting the PDE page size flag (PDE.PS) to 1. Because the first implementation of the Hammer family of processors supports a maximum 48 bits of virtual address, this paging option supports 2^{27} 2MByte pages spanning a virtual-address space of 2^{48} bytes (256 terabytes).

The 48-bit virtual address is broken up into four fields to index into the 3-level paging structure, as follows and in Figure 18:

- 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.

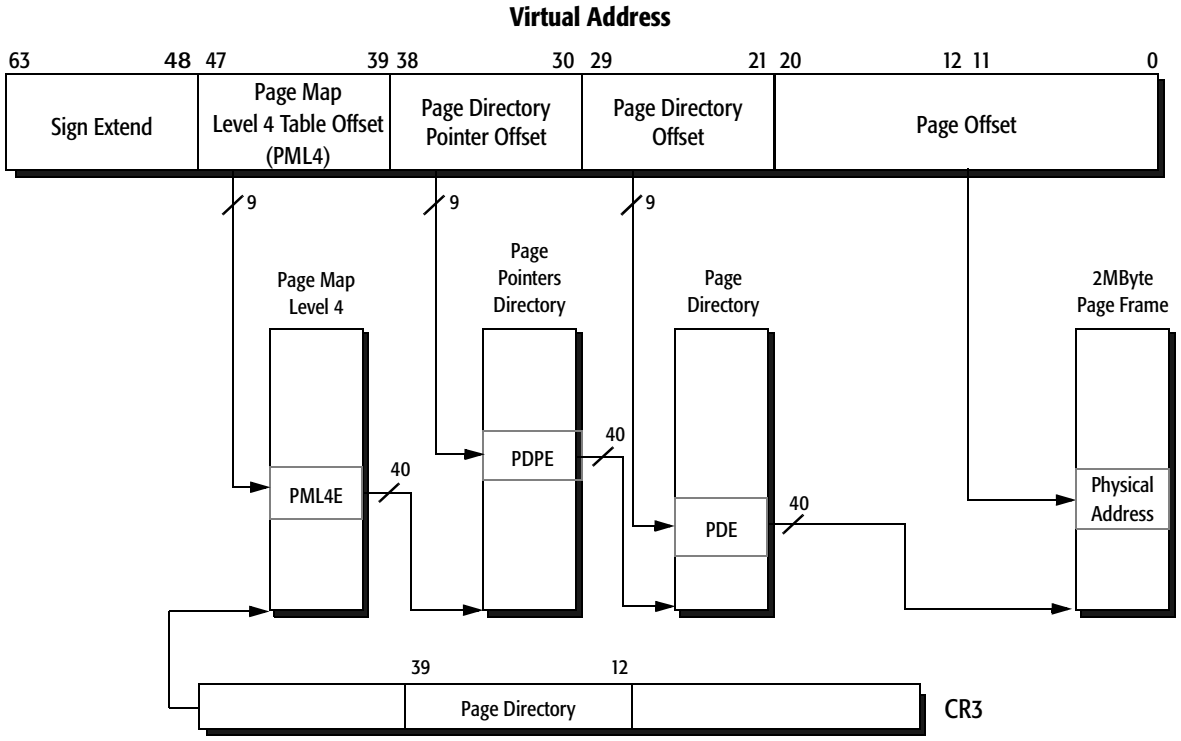


Figure 18. 2MB-Page Translation

Privilege-Level Transitions and Far Transfers

The long-mode architecture provides two mechanisms for changing privilege levels:

- Call gates and interrupt gates. See "Interrupts" on page 65 for details on interrupts.
- SYSCALL and SYSRET instructions.

Call Gates

The call-gate mechanism provides a public entry point into the operating system. It also provides a means for changing privilege levels and stacks when calling the operating system.

Gate Descriptor Format. Legacy x86 call-gate descriptors provide a 32-bit offset for the instruction pointer (EIP). The long-mode architecture doubles the size of legacy call gates to provide a 64-bit offset for the instruction pointer (RIP).

Figure 19 shows the layout of a call-gate descriptor in long mode. Table 13 describes the fields in a long-mode call gate.

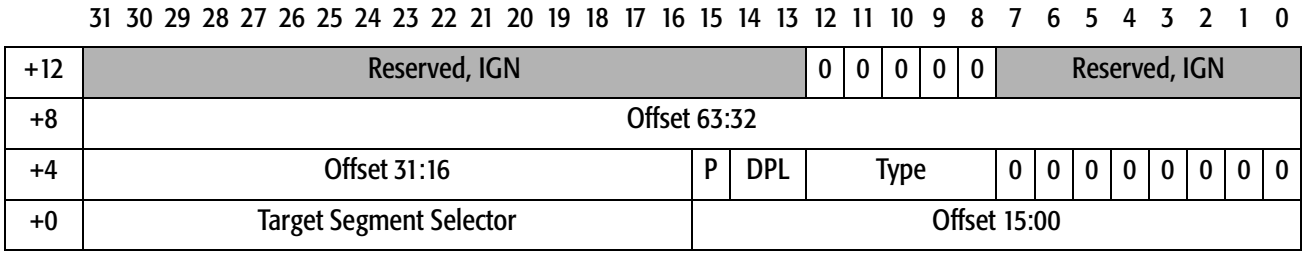


Figure 19. Call Gates in Long Mode

Table 13. Long-Mode Call-Gate Fields

Gate Field	Function
+12[31:13]	Unused
+12[12:8]	Must be 0
+12[7:0]	Unused
+8[31:0]	Offset 63:32 in canonical form
+4[31:16]	Offset 31:16
+4[15:13]	Present and Descriptor Privilege Level

Table 13. Long-Mode Call-Gate Fields (continued)

Gate Field	Function
+4[12:8]	Long-mode Call Gate Type (0Ch)
+4[7:0]	Unused
+0[31:16]	Target Segment Selector
+0[15:0]	Offset 15:0

The first eight bytes (bytes 7:0) of a long-mode call gate are identical to legacy 32-bit call gates. Bytes 11:8 hold the upper 32 bits of the target-segment offset in canonical form. A general-protection exception (#GP) is generated if software attempts to use a call gate with a target offset that is not in canonical form.

The target code segment referenced by the call gate must be a 64-bit code segment (CS.L=1, CS.D=0). If it is not, a general-protection exception, #GP(selector), is generated with the target CS selector reported as the error code.

The double-sized descriptors can reside in the same descriptor table as 16-bit and 32-bit legacy descriptors. A second type field, used for consistency checking, is defined in bits 12:8 of the highest dword and must be cleared to zero. This illegal type (00h) results in a general-protection exception (#GP) if an attempt is made to access the upper half of the 64-bit mode descriptor as a legacy descriptor.

Call Gates in Long Mode. Only long-mode call gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy 32-bit call gate type (0Ch) is redefined in long mode as the 64-bit call-gate type. No 32-bit call-gate type exists in long mode. If a far call references a 16-bit call gate type (04h), a general-protection exception (#GP) is generated.

Far Call Operand Size. The operand size of a far call determines the size of operand-stack pushes and the size of the instruction-pointer update. Far-call instructions that reference a call gate use the gate size inferred by the type field to set the operand size.

When a CALL references a long-mode call gate, the actions taken are identical to those taken in legacy calls through a 32-bit gate, with the following exceptions:

- Stack pushes are performed in eight-byte increments.
- A 64-bit RIP is pushed onto the stack.

The matching far-return instruction must be performed with a 64-bit operand-size override in order to process the stack correctly.

Privilege-Level Changes and Stack Switching. A call gate can be used to change to a more-privileged code segment. Although the protection-check rules for call gates are unchanged in long mode from legacy mode, the associated stack-switch changes slightly in long mode.

Legacy-mode stack pointers consist of an SS:eSP pair (16-bit selector and 16-bit or 32-bit offset). Stack pointers for privilege levels 0, 1 and 2 are created by the operating system and stored in the current TSS. In legacy mode, call-gate transfers that change privilege levels cause the processor to automatically perform a stack switch from the current stack to the inner-level stack defined for the new privilege level. A new SS:eSP pair is loaded from the TSS and the stack switch is initiated. After completing the stack switch, the processor pushes the old SS:eSP pair onto the new stack so that the subsequent far returns will restore the old stack.

In long mode, the target of any call gate must be a 64-bit code segment. 64-bit mode does not use segmentation, and stack pointers consist solely of the 64-bit stack pointer (RSP). The SS segment register is ignored in 64-bit mode.

When stacks are switched as part of a long-mode privilege-level change through a call gate, a new SS descriptor is not loaded. Long mode only loads an inner-level RSP from the TSS. The old SS and RSP are saved on the new stack, as shown in Figure 20.

Figure 20. Long-Mode Stack Layout After CALLF with CPL Change

Although SS is not changed, the old SS value is popped on the subsequent RETF. This is done to allow an operating system to set up SS for a compatibility-mode process, by doing a RETF (or IRET) that changes privilege level.

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. SS.RPL is restored from the return-target CS.RPL on the subsequent privilege-level-changing far return.

All long-mode stack pushes resulting from a privilege-level-changing far call or far return are eight-bytes wide and increment the RSP by eight.

Automatic Parameter Copy. Long mode does not support the automatic parameter-copy feature found in legacy mode. The call-gate count field is ignored by long mode. Software can access the old stack, if necessary, by referencing the old stack-segment selector and stack pointer saved on the new process stack.

SYSCALL and SYSRET

The SYSCALL and SYSRET instructions were designed for operating systems that use a flat memory model where segmentation is not used. This makes the instructions ideally suited for long mode. The semantics of SYSCALL and SYSRET are expanded by the long-mode architecture to specify a 64-bit code offset. In addition, two such offsets are defined, one for 32-bit compatibility-mode callers and another for 64-bit-mode callers.

System Target-Address Registers. The legacy system target-address register (STAR) cannot be expanded to provide a 64-bit target RIP address, because the upper 32 bits of that MSR already contain the target CS and SS selectors. Long mode provides two new STAR registers—long STAR (LSTAR) and compatibility STAR (CSTAR)—that hold a 64-bit target RIP. LSTAR holds the target RIP used by a SYSCALL when long mode is activated and the calling program is in 64-bit mode. CSTAR holds the target RIP used by a SYSCALL when long mode is activated and the calling program is in compatibility mode.

The SYSCALL and SYSRET CS and SS selectors used in long mode and legacy mode are stored in the STAR.

Figure 21 shows the layout and MSR numbers for the STAR, LSTAR, and CSTAR registers.

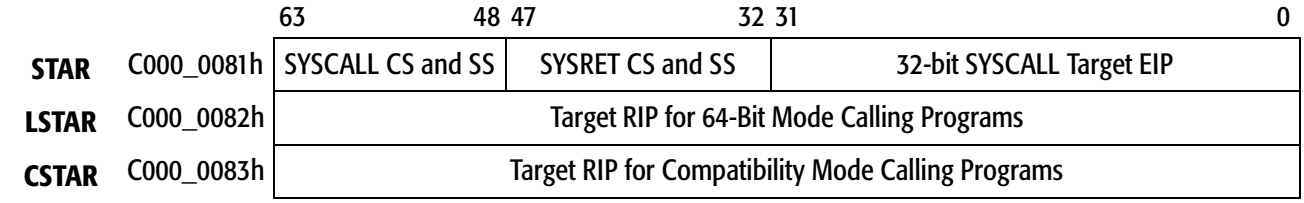


Figure 21. STAR, LSTAR, and CSTAR Model-Specific Registers (MSRs)

Operation. SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from the LSTAR (64-bit mode) or CSTAR (compatibility mode). Upon return, SYSRET copies the value saved in RCX into the RIP.

When SYSCALL is executed in long mode, the processor assumes a 64-bit operating system is being called. SYSCALL sets the target CS.L to 1, regardless of the current operating mode (64-bit or compatibility), forcing the processor into 64-bit mode.

Because a SYSCALLed operating system can be entered from either 64-bit mode or compatibility mode, the corresponding SYSRET must know the mode to which it must return. The called operating-system service routine must have different entry points for 32-bit compatibility-mode callers and 64-bit-mode callers in order to translate arguments. In the service-routine entry point code, a flag can be set indicating which type of SYSRET is needed upon exiting the called routine. A REX-

prefix operand-size override can be used with SYSRET to put the processor into 64-bit mode on return. Executing SYSRET without a REX-prefix operand-size override puts the processor in compatibility mode on return.

Task State Segments

The legacy x86 task-switching architecture is not supported in long mode. Long mode requires that task management and switching be performed by software. The processor issues a general-protection exception (#GP) if any of the following is attempted in long mode:

- A control transfer to a TSS or a task gate via a JMP, CALL, INTn, or interrupt.
- An IRET with EFLAGS.NT (nested task) set to 1.

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. Figure 22 shows the format of a 64-bit TSS. This 64-bit TSS holds several pieces of information important to long mode and not directly related to the task-switch mechanism. These are:

- **RSP*n*.** The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0–2 are stored in these fields.
- **IST*n*.** The full 64-bit canonical forms of the interrupt stack table (IST) pointers. See "Interrupt Stack Table" on page 69 for a description of the IST mechanism.
- **I/O Map Base Address.** The 16-bit offset to the I/O permission bit map from the 64-bit TSS base.

The operating system must create at least one 64-bit TSS 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.

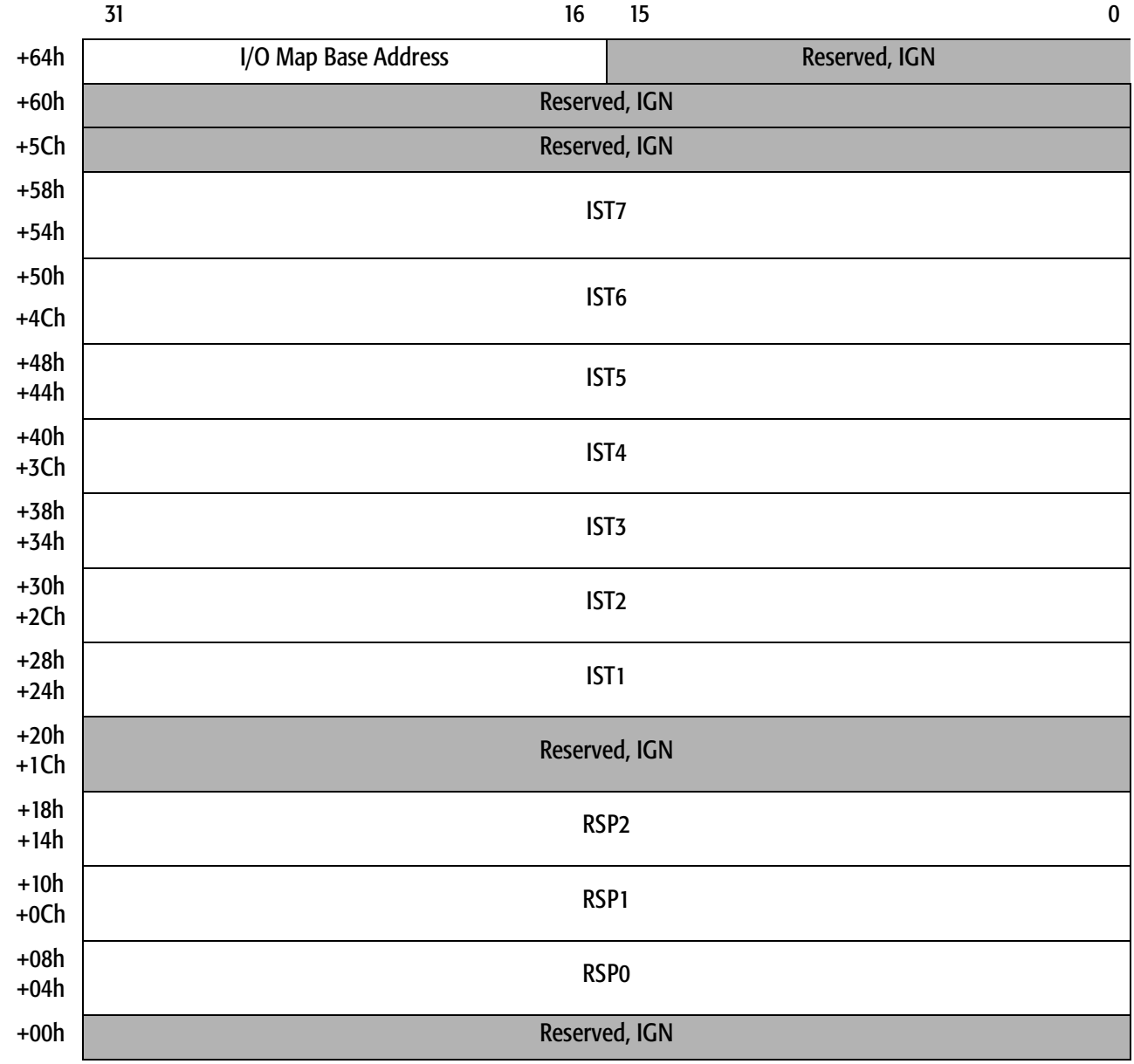


Figure 22. TSS Format in Long Mode

Interrupts

Interrupts and exceptions force control transfers to occur from the currently executing program to an interrupt service routine that handles the particular interrupt. The interrupt-handling and exception-handling mechanism saves the interrupted program's execution state, transfers control to the interrupt service routine, and ultimately returns to the interrupted program.

Throughout this section, the term "interrupt" covers both asynchronous events generated external to the processor (interrupts) and synchronous events related to instruction execution (exceptions, faults and traps).

The long-mode architecture expands the legacy x86 interrupt-processing and exception-processing mechanism to support 64-bit operating systems and applications. These changes include:

- All interrupt handlers are 64-bit code.
- The size of interrupt-stack pushes is fixed at 64 bits.
- The stack pointer, SS:RSP, is pushed unconditionally on interrupts, rather than conditionally based on a change in current privilege level (CPL).
- No SS switch on interrupts.
- IRET behavior changes.
- New interrupt stack-switch mechanism.

Gate Descriptor Format

The interrupt descriptor table (IDT) contains gate descriptors that are used to locate the service routine for each interrupt vector. Legacy interrupt-gate descriptors provide a 32-bit offset for the instruction pointer (EIP). The long-mode architecture doubles the size of legacy interrupt gates from eight bytes to 16 bytes in order to provide a 64-bit offset for the instruction pointer (RIP). The 64-bit RIP referenced by an interrupt-gate descriptor allows an interrupt service routine to be located anywhere in the virtual-address space.

Figure 23 shows the layout of long-mode interrupt-gate and trap-gate descriptors. Table 14 describes the fields in a 64-bit interrupt and trap gate.

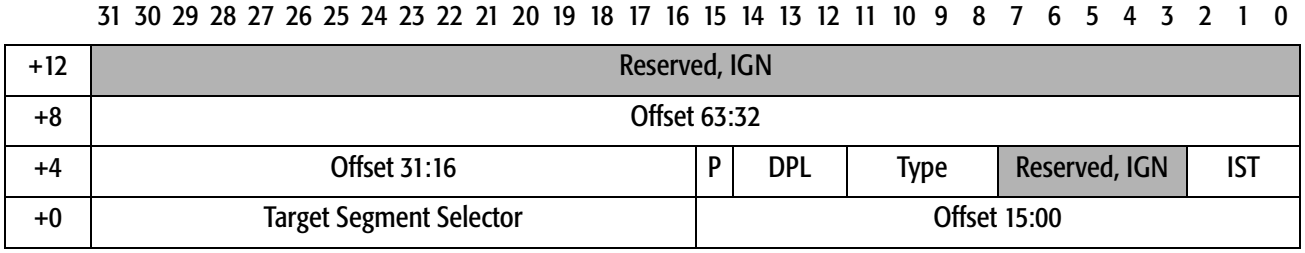


Figure 23. Interrupt and Trap Gate in Long Mode

Table 14. Long-Mode Interrupt- and Trap-Gate Fields

Gate Field	Function
+12[31:0]	Unused
+8[31:0]	Offset bits 63:32
+4[31:16]	Offset bits 31:16
+4[15:13]	Present and Descriptor Privilege Level
+4[12:8]	64-bit Interrupt or Trap Gate Type (0Eh or 0Fh)
+4[7:3]	Unused
+4[2:0]	IST Index
+0[31:16]	Target Segment Selector
+0[15:0]	Offset bits 15:0

In legacy mode, the IDT index is formed by scaling the interrupt vector by eight. In long mode, the IDT index is formed by scaling the interrupt vector by 16.

The first eight bytes (bytes 7:0) of a long-mode interrupt gate are identical to legacy 32-bit interrupt gates. Bytes 11:8 hold the upper 32 bits of the target RIP (interrupt segment offset) in canonical form. A general-protection exception, #GP(0) is generated if software attempts to reference an interrupt gate with a target RIP that is not in canonical form.

The target code segment referenced by the interrupt gate 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), is generated with the IDT vector number reported as the error code.

Only 64-Bit Interrupt Gates in Long Mode. Only 64-bit interrupt gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy 32-bit interrupt or trap gate types (0Eh or 0Fh) are redefined in long mode as the 64-bit interrupt- and trap-gate types. No 32-bit interrupt or trap gate types exists in long mode. If a reference is made to a 16-bit interrupt or trap gate (06h or 07h), a general-protection exception, #GP(0), is generated.

Stack Frame

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 only long-mode gates can be referenced. Long mode also pushes SS:RSP unconditionally, rather than pushing only on a CPL change.

Aside from error codes, pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stack-frame size across all interrupts. Interrupt service-routine entry points that handle interrupts generated by the INTn instruction or external INTR# signal can push an error code for consistency.

IRET

IRET semantics change in long mode. IRET must be executed with an 8-byte operand size. In 64-bit mode, SS:RSP is popped unconditionally. In compatibility and legacy modes, SS:RSP is popped only if the CPL changes.

Because interrupt stack-frame pushes are always eight bytes in long mode, an IRET must pop eight byte items off the stack. This is accomplished by preceding the IRET with a 64-bit operand-size prefix.

IRET pops SS:RSP unconditionally off the interrupt stack frame only when executed in 64-bit mode. In compatibility mode, IRET pops SS:RSP 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.

64-bit interrupt service routines that exit with an IRET unconditionally pop SS:RSP off of the interrupt stack frame, even if the target code segment is running in 64-bit mode or at CPL=0. This is done because the original interrupt always pushes SS:RSP.

Stack Switching

The legacy x86 architecture provides a mechanism to automatically switch stack frames in response to an interrupt. The long-mode architecture implements a slightly modified version of the legacy stack-switching mechanism and an alternative stack-switching mechanism called the interrupt stack table (IST).

Stack Switches Legacy Mode and Long Mode. In legacy mode, the legacy x86 stack-switch mechanism is unchanged. Legacy-mode stack pointers consist of an SS:eSP pair (16-bit selector and a 16-bit or 32-bit offset). The operating system must create stack pointers for privilege levels 0, 1 and 2 and store them in the current TSS. In legacy mode, switching to a new privilege level as the result of an interrupt causes the processor to automatically perform a stack switch from the current stack to the inner-level stack defined for the new privilege level. A new SS:eSP pair is loaded from the TSS and the stack switch is initiated. After completing the stack switch, the processor pushes the old SS:eSP pair onto the new stack so that the subsequent IRETs restore the old stack.

In long mode, the legacy stack-switch mechanism is modified. When stacks are switched as part of a long-mode privilege-level change resulting from an interrupt, a new SS descriptor is not loaded. Long mode only loads an inner-level RSP from the TSS. The old SS and RSP are saved on the new stack, as shown in Figure 24.

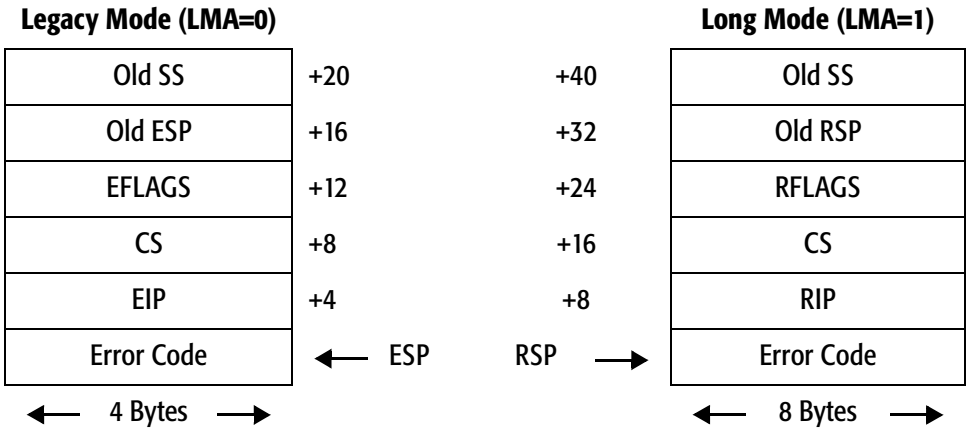


Figure 24. Long-Mode Stack Layout After Interrupt with CPL Change

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 described above. This IST mechanism unconditionally switches stacks when it is enabled. It can be enabled on an individual interrupt-vector basis via a field in the IDT entry. Thus, some interrupt vectors can use the modified legacy mechanism and others can use the IST mechanism. The IST mechanism is only available in long mode. It is part of the long-mode TSS shown in Figure 22 on page 64.

The primary motivation for the IST mechanism is to provide 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 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 legacy task-switch mechanism is not supported in long mode.

The IST mechanism is part of the long-mode task state segment (TSS) shown in Figure 22 on page 64. It provides up to seven IST pointers located in the TSS. The pointers are referenced by an interrupt-gate descriptor in the interrupt-descriptor table (IDT), as shown in Figure 23 on page 66. The gate descriptor contains a 3-bit IST index field that provides an offset into the IST section of the TSS.

If the IST index for an interrupt gate is not zero, the IST pointer corresponding to the index is loaded into the RSP when an

interrupt occurs. The value of SS is not changed. The old SS, RSP, RFLAGS, CS, and RIP are pushed onto the new stack. Interrupt processing then proceeds as normal.

If the IST index is zero, the modified legacy stack-switching mechanism described above is used.

Task Priority

The x86-64 architecture defines 15 external interrupt-priority classes. Priority class 1 is the lowest and 15 is the highest. How external interrupts are organized into these priority classes is implementation-dependent.

Operating systems can use the TPR to temporarily block specific (generally 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 of 9 or less, while allowing all interrupts with a priority 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.

Software can read and write the TPR using a MOV CR8 instruction. The new priority level is established when the MOV instruction completes execution. Software does not need to force serialization after loading TPR. Consider, for example, two sequential TPR loads, in which a low value is first loaded into TPR and immediately followed by a load of a higher value. Any pending, lower-priority interrupt enabled by the first MOV to TPR is recognized between the two MOVs.

Use of the MOV CR*n* instruction requires a privilege level of 0. Programs running at any other privilege level cannot read or write the TPR. An attempt to do so results in a general-protection exception, #GP(0).

The TPR is abstracted from the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external device, such as an 8259, or it can be integrated on-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 considered implementation-dependent, with the underlying priority mechanisms subject to change.

The TPR, by contrast, is part of the x86-64 architecture. Software can depend on this definition remaining unchanged.

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.

See Appendix D, "Implementation Considerations", on page 105 for information on the implementation-specific IC in the first implementation of the Hammer family of processors.

Appendix A Integer Instructions in 64-Bit Mode

This appendix provides details of the integer instruction set, in 64-bit mode, and its differences from legacy and compatibility modes. No changes have been made to x87 floating-point instructions, so those instructions are not covered here.

General Rules

In 64-bit mode, the following general rules apply to changes in instructions and their operands:

- **Effective Operand Size**: If an instruction's operand size in the legacy architecture depends on the effective operand size (thus, dependent on CS.D and prefix overrides), then the operand-size choices are extended in 64-bit mode to include 64 bits. Such instructions are said to be *promoted to 64 bits* in Table 15.
- **Fixed-Size Operand**: If an instruction's operand size is fixed in the legacy architecture (thus, independent of CS.D and prefix overrides), that operand size is usually fixed at the same size in 64-bit mode. For example, CPUID and SMSW operate on the same-sized operand in legacy and 64-bit mode. (There are some exceptions, however. For example, BSWAP.)
- **Zero-Fills in High 32 Bits**: Operations on 32-bit operands usually fill the upper 32 bits of 64-bit destination registers with zeros in 64-bit mode.
- **Shifts and Rotates**. Shifts and rotates on 64-bit operands use one additional bit to specify shift-count. The shift count is masked to six bits.

Operand Size in 64-Bit Mode

Table 15 lists the integer instructions, showing operand size in 64-bit mode and the state of the high 32 bits of destination registers when 32-bit operands are used.

Table 15. Operand Size in 64-Bit Mode

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
ADC—ADD with Carry	Promoted to 64 bits.	Zero-extend 32-bit register results.
0001 000w : 11 reg1 reg2		
0001 001w : 11 reg1 reg2		
0001 000w : mod reg r/m		
0001 001w : mod reg r/m		
0001 010w : immediate data		
1000 00sw : 11 010 reg : immediate data		
1000 00sw : mod 010 r/m : immediate data		
ADD—Add	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 000w : 11 reg1 reg2		
0000 001w : 11 reg1 reg2		
0000 000w : mod reg r/m		
0000 001w : mod reg r/m		
1000 00sw : 11 000 reg : immediate data		
1000 00sw : mod 000 r/m : immediate data		
0000 010w : immediate data		
AND—And	Promoted to 64 bits.	Zero-extend 32-bit register results.
0010 000w : 11 reg1 reg2		
0010 001w : 11 reg1 reg2		
0010 000w : mod reg r/m		
0010 001w : mod reg r/m		
1000 00sw : 11 100 reg : immediate data		
1000 00sw : mod 100 r/m : immediate data		
0010 010w : immediate data		
BSF—Bit Scan Forward	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1011 1100		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
BSR—Bit Scan Reverse	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1011 1101		
BSWAP—Byte Swap	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1100 1 reg		
BT—Bit Test	Promoted to 64 bits.	No register result.
0000 1111 : 1011 1010 : 11 100 reg: imm8 data		
0000 1111 : 1011 1010 : mod 100 r/m : imm8 data		
0000 1111 : 1010 0011 : 11 reg2 reg1		
0000 1111 : 1010 0011 : mod reg r/m		
BTC—Bit Test and Complement	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1011 1010 : 11 111 reg: imm8 data		
0000 1111 : 1011 1010 : mod 111 r/m : imm8 data		
0000 1111 : 1011 1011 : 11 reg2 reg1		
0000 1111 : 1011 1011 : mod reg r/m		
BTR—Bit Test and Reset	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1011 1010 : 11 110 reg: imm8 data		
0000 1111 : 1011 1010 : mod 110 r/m : imm8 data		
0000 1111 : 1011 0011 : 11 reg2 reg1		
0000 1111 : 1011 0011 : mod reg r/m		
BTS—Bit Test and Set	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1011 1010 : 11 101 reg: imm8 data		
0000 1111 : 1011 1010 : mod 101 r/m : imm8 data		
0000 1111 : 1010 1011 : 11 reg2 reg1		
0000 1111 : 1010 1011 : mod reg r/m		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
CALL—Call Procedure (in same segment)		Operand size is fixed at 64 bits.
1110 1000 : full displacement, direct	RIP=RIP+32-bit displacement.	
1111 1111 : 11 010 reg, indirect	RIP=64-bit absolute offset.	
1111 1111 : mod 010 r/m, indirect	RIP=64-bit absolute offset.	
CALL—Call Procedure (in other segment)		
1001 1010 : unsigned full offset, selector	If selector points to a gate then RIP=RIP+64-bit offset from gate, else RIP=RIP+32-bit offset.	
1111 1111 : mod 011 r/m	If selector points to a gate then RIP=RIP+64-bit offset from gate, else RIP=RIP+32-bit offset.	
CBW—Convert Byte to Word	RAX = sign extend of EAX. (name for this instruction: "Convert DoubleWord to QuadWord" "CDQE").	Zero-extend 32-bit register results.
1001 1000		
CDQ—Convert Doubleword to Qword	RDX:RAX = sign extend of RAX. (name for this instruction: "Convert QuadWord to OctWord" -- "CQO").	Zero-extend 32-bit EDX register results. RAX is unchanged.
1001 1001		
CLC—Clear Carry Flag	No register result.	No register result.
1111 1000		
CLD—Clear Direction Flag	No register result.	No register result.
1111 1100		
CLI—Clear Interrupt Flag	No register result.	No register result.
1111 1010		
CLTS—Clear Task-Switched Flag in CR0	No register result.	No register result.
0000 1111 : 0000 0110		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
CMC—Complement Carry Flag	No register result.	No register result.
1111 0101		
CMOVcc—Conditional Move	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111: 0100 tttn : 11 reg1 reg2		
0000 1111: 0100 tttn : mod mem r/m		
CMP—Compare Two Operands	Promoted to 64 bits.	No register result.
0011 100w : 11 reg1 reg2		
0011 101w : 11 reg1 reg2		
0011 100w : mod reg r/m		
0011 101w : mod reg r/m		
1000 00sw : 11 111 reg : immediate data		
0011 110w : immediate data		
1000 00sw : mod 111 r/m		
CMPS/**CMPSB**/**CMPSW**/**CMPSD**—Compare String Operands	Promoted to 64 bits.	Zero-extend 32-bit register results.
1010 011w		
CMPXCHG—Compare and Exchange	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1011 000w : 11 reg2 reg1		
0000 1111 : 1011 000w : mod reg r/m		
CMPXCHG8B—Compare and Exchange 8 Bytes	Mode-independent. Operand size is fixed at 64 bits.	Zero-extend EDX and EAX.
0000 1111 : 1100 0111 : mod reg r/m		
CPUID—CPU Identification	Mode-independent. Operand is fixed at 32 bits	Zero-extend 32-bit register results.
0000 1111 : 1010 0010		
CWD—Convert Word to Doubleword	See CDQ.	
1001 1001		
CWDE —Convert Word to Doubleword	See CBW.	
1001 1000		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
DEC—Decrement by 1		
0100 1 reg	Used as REX prefix.	
1111 111w : 11 001 reg	Promoted to 64 bits.	Zero-extend 32-bit register results.
1111 111w : mod 001 r/m		
DIV—Unsigned Divide	RDX:RAX/64-bit extended to yield 64-bit quotient and remainder.	Zero-extend 32-bit register results.
1111 011w : 11 110 reg		
1111 011w : mod 110 r/m		
ENTER—Make Stack Frame for High Level Procedure	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1100 1000 : 16-bit displacement : 8-bit level (L)		
HLT—Halt	No register result.	No register result.
1111 0100		
IDIV—Signed Divide	RDX:RAX/64-bit extended to yield 64-bit quotient and remainder .	Zero-extend 32-bit register results.
1111 011w : 11 111 reg		
1111 011w : mod 111 r/m		
IMUL - Signed Multiply	Natural 128-bit or 64-bit result (sign-extend any immediates as needed).	Zero-extend 32-bit register results.
1111 011w : 11 101 reg	RDX:RAX = RAX * r64	
1111 011w : mod 101 reg	RDX:RAX = RAX * m64	
0000 1111 : 1010 1111 : 11 : reg1 reg2	r64 = r64 * r64	
0000 1111 : 1010 1111 : mod reg r/m	R64 = r64 * m64	
0110 10s1 : 11 reg1 reg2 : immediate data	6B : r64 = r64 *imm8 69: r64 = r64 * imm32	
0110 10s1 : mod reg r/m : immediate data	6B : r64 = m64 *imm8 69: r64 = m64 * imm32	

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
IN—Input From Port	Not promoted (stays 1, 2, or 4 bytes).	Zero-extend 32-bit register results.
fixed port 1110 010w : port number		
variable port 1110 110w		
INC—Increment by 1		
0100 0 reg	Used as REX prefix.	
1111 111w : 11 000 reg	Promoted to 64 bits.	Zero-extend 32-bit register results.
1111 111w : mod 000 r/m		
INS—Input from DX Port	Not promoted (stays 1, 2, or 4 bytes).	Zero-extend 32-bit register results.
0110 110w		
INT n—Interrupt Type n	See Long Mode Arch Specification.	
1100 1101 : type		
INT—Single-Step Interrupt 3	See Long Mode Arch Specification.	
1100 1100		
INVD—Invalidate Cache	Mode-independent.	No register result.
0000 1111 : 0000 1000		
INVLPG—Invalidate TLB Entry	In 64-bit mode, the source operand is a 64-bit linear address.	No register result.
0000 1111 : 0000 0001 : mod 111 r/m		
IRET/IRETD—Interrupt Return	See Long Mode Arch Specification.	
1100 1111		
Jcc—Jump if Condition is Met	No register result.	No register result. Operand size is fixed at 64 bits.
0111 tttn : 8-bit displacement		
0000 1111 : 1000 tttn : full displacement		
JCXZ/JECXZ—Jump on CX/ECX Zero	No register result.	No register result. Operand size is fixed at 64 bits.
1110 0011 : 8-bit displacement		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
JMP—Unconditional Jump (to same segment)	No register result.	No register result. Operand size is fixed at 64 bits.
1110 1011 : 8-bit displacement		
1110 1001 : full displacement		
1111 1111 : 11 100 reg		
1111 1111 : mod 100 r/m		
JMP—Unconditional Jump (to other segment)	No register result.	No register result.
1110 1010 : unsigned full offset, selector		
1111 1111 : mod 101 r/m		
LAR—Load Access Rights Byte	Same as 32-bit mode, except writes zero-extended 32-bit result into 64-bit destination register.	Zero-extend 32-bit register results.
0000 1111 : 0000 0010 : 11 reg1 reg2		
0000 1111 : 0000 0010 : mod reg r/m		
LEAVE—High Level Procedure Exit	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1100 1001		
LES—Load Pointer to ES	Segment Instruction, no changes.	Zero-extend 32-bit register results.
1100 0100 : mod reg r/m		
LFS—Load Pointer to FS	Segment Instruction, no changes.	Zero-extend 32-bit register results.
0000 1111 : 1011 0100 : mod reg r/m		
LGDT—Load Global Descriptor Table Register	Operand size increased for 64-bit base.	No register result.
0000 1111 : 0000 0001 : mod 010 r/m		
LGS—Load Pointer to GS	Segment Instruction, no changes.	Zero-extend 32-bit register results.
0000 1111 : 1011 0101 : mod reg r/m		
LIDT—Load Interrupt Descriptor Table Register	Operand size increased for 64-bit base.	No register result.
0000 1111 : 0000 0001 : mod 011 r/m		
LLDT—Load Local Descriptor Table Register	References expanded descriptor to load 64-bit base.	No register result.
0000 1111 : 0000 0000 : 11 010 reg		
0000 1111 : 0000 0000 : mod 010 r/m		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
LMSW—Load Machine Status Word	Mode-independent.	No register result.
0000 1111 : 0000 0001 : 11 110 reg		
0000 1111 : 0000 0001 : mod 110 r/m		
LODS/LODSB/LODSW/LODSD—Load String Operand	Promoted to 64 bits.	Zero-extend 32-bit register results.
1010 110w		
LOOP—Loop Count	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1110 0010 : 8-bit displacement		
LOOPZ/LOOPE—Loop Count while Zero/Equal	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1110 0001 : 8-bit displacement		
LOOPNZ/LOOPNE—Loop Count while not Zero/ Equal	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1110 0000 : 8-bit displacement		
LSL—Load Segment Limit	Same as 32-bit mode, zero-extends 32-bit result into 64-bit destination register.	Zero-extend 32-bit register results.
0000 1111 : 0000 0011 : 11 reg1 reg2		
0000 1111 : 0000 0011 : mod reg r/m		
LSS —Load Pointer to SS	Segment Instruction, no changes.	Zero-extend 32-bit register results.
0000 1111 : 1011 0010 : mod reg r/m		
LTR—Load Task Register	References expanded descriptor to load 64-bit base.	No register result.
from register 0000 1111 : 0000 0000 : 11 011 reg		
from memory 0000 1111 : 0000 0000 : mod 011 r/m		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
MOV—Move Data	Promoted to 64 bits.	Zero-extend 32-bit register results.
1000 100w : 11 reg1 reg2		
1000 101w : 11 reg1 reg2		
1000 101w : mod reg r/m		
1000 100w : mod reg r/m		
1100 011w : 11 000 reg : immediate data		
1011 w reg : immediate data	64-bit immediate.	
1100 011w : mod 000 r/m : immediate data	Promoted to 64 bits.	
1010 000w : full displacement	Displacement is 64-bit literal.	
1010 001w : full displacement	Displacement is 64-bit literal.	
MOV—Move to/from Control Registers	Mode-independent - operand size is fixed at 64 bits.	Operand size is fixed at 64 bits.
0000 1111 : 0010 0010 : 11 000 reg		
0000 1111 : 0010 0010 : 11 010reg		
0000 1111 : 0010 0010 : 11 011 reg		
0000 1111 : 0010 0010 : 11 100 reg		
0000 1111 : 0010 0000 : 11 eee reg		
MOV—Move to/from Debug Registers	Mode-independent - operand size is fixed at 64 bits.	Operand size is fixed at 64 bits.
0000 1111 : 0010 0011 : 11 eee reg		
0000 1111 : 0010 0011 : 11 eee reg		
0000 1111 : 0010 0011 : 11 eee reg		
0000 1111 : 0010 0001 : 11 eee reg		
0000 1111 : 0010 0001 : 11 eee reg		
0000 1111 : 0010 0001 : 11 eee reg		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
MOV—Move to/from Segment Registers	Segment Instruction, no changes.	Zero-extend 32-bit register results.
1000 1110 : 11 sreg3 reg		
1000 1110 : 11 sreg3 reg		
1000 1110 : mod sreg3 r/m		
1000 1110 : mod sreg3 r/m		
1000 1100 : 11 sreg3 reg		
1000 1100 : mod sreg3 r/m		
MOVD—Move to/from MMX™ technology registers	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111: 0110 1110: 11 mmm reg		
0000 1111: 0111 1110: 11 reg mmmm		
0000 1111: 0110 1110: mod mmm r/m		
0000 1111: 0111 1110: mod mmm r/m		
MOVS/**MOVSB**/**MOVSW**/**MOVSD**—Move Data from String to String	Promoted to 64 bits.	Zero-extend 32-bit register results.
1010 010w		
MOVSX—Move with Sign-Extend	Promoted to 64 bits. "0F BE" is "Move Byte to Quadword". "0F BF' is "Move Word to Quadword".	Zero-extend 32-bit register results.
0000 1111 : 1011 111w : 11 reg1 reg2		
0000 1111 : 1011 111w : mod reg r/m		
MOVSXD—Move with Sign Extend Double	New instruction.	Zero-extend 32 bit register results.
0110 0011 :11 reg1 reg2		
0110 0011 :mod reg r/m		
MOVZX—Move with Zero-Extend	Promoted to 64 bits. "0F B6" is "Move Byte to Quadword". "0F B7' is "Move Word to Quadword".	Zero-extend 32-bit register results.
0000 1111 : 1011 011w : 11 reg1 reg2		
0000 1111 : 1011 011w : mod reg r/m		
MUL—Unsigned Multiply	Promoted to 64 bits: RDX:RAX=RAX * r/m qword.	Zero extend 32-bit results. This is an unsigned operand.
1111 011w : 11 100 reg		
1111 011w : mod 100 reg		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
NEG—Twos Complement Negation	Promoted to 64 bits.	Zero-extend 32-bit register results.
1111 011w : 11 011 reg		
1111 011w : mod 011 r/m		
NOP—No Operation	No register result.	No register result.
1001 0000		
NOT—Ones Complement Negation	Promoted to 64 bits.	Zero-extend 32-bit register results.
1111 011w : 11 010 reg		
1111 011w : mod 010 r/m		
OR—Logical Inclusive OR	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 100w : 11 reg1 reg2		
0000 101w : 11 reg1 reg2		
0000 101w : mod reg r/m		
0000 100w : mod reg r/m		
1000 00sw : 11 001 reg : immediate data		
0000 110w : immediate data		
1000 00sw : mod 001 r/m : immediate data		
OUT—Output to Port	Not promoted (stays 1, 2, or 4 bytes).	Zero-extend 32-bit register results.
1110 011w : port number		
1110 111w		
OUTS—Output to DX Port	Not promoted (stays 1, 2, or 4 bytes).	Zero-extend 32-bit register results.
0110 111w		
POP—Pop a Value from the Stack	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1000 1111 : 11 000 reg		
0101 1 reg		
1000 1111 : mod 000 r/m		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
POP—Pop a Segment Register from the Stack	Not promoted.	Zero-extend 32-bit register results.
0000 1111: 10 sreg3 001		
POPF/POPFD—Pop Stack into FLAGS or EFLAGS Register	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1001 1101		
PUSH—Push Operand onto the Stack	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1111 1111 : 11 110 reg		
0101 0 reg		
1111 1111 : mod 110 r/m		
0110 10s0 : immediate data		
PUSH—Push Segment Register onto the Stack	Promoted to 64 bits.	Operand size is fixed at 64 bits.
segment register FS,GS 0000 1111: 10 sreg3 000		
PUSHF/PUSHFD—Push Flags Register onto the Stack	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1001 1100		
RCL—Rotate through Carry Left	64-bit extension (Uses 6-bit count).	Zero-extend 32-bit register results (Uses 5-bit count).
1101 000w : 11 010 reg		
1101 000w : mod 010 r/m		
1101 001w : 11 010 reg		
1101 001w : mod 010 r/m		
1100 000w : 11 010 reg : imm8 data		
1100 000w : mod 010 r/m : imm8 data		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
RCR—Rotate through Carry Right	64-bit extension (Uses 6-bit count).	Zero-extend 32-bit register results (Uses 5-bit count).
1101 000w : 11 011 reg		
1101 000w : mod 011 r/m		
1101 001w : 11 011 reg		
1101 001w : mod 011 r/m		
1100 000w : 11 011 reg : imm8 data		
1100 000w : mod 011 r/m : imm8 data		
RDMSR—Read from Model-Specific Register	Returns upper 32bits in RDX, lower 32 bits in RAX.	Returns data in EDX:EAX.
0000 1111 : 0011 0010		
RDPMC—Read Performance Monitoring Counters	Returns upper 32bits in RDX, lower 32 bits in RAX.	Returns data in EDX:EAX.
0000 1111 : 0011 0011		
RDTSC—Read Time-Stamp Counter	Returns upper 32bits in RDX, lower 32 bits in RAX.	Returns data in EDX:EAX.
0000 1111 : 0011 0001		
REP INS—Input String	Uses RSI. IO size remains 1, 2, 4 bytes.	Zero extends ESI to 64 bits.
1111 0011 : 0110 110w		
REP LODS—Load String	Uses RCX, RAX, RSI.	Zero-extend ECX, EAX, ESI.
1111 0011 : 1010 110w		
REP MOVS—Move String	Uses RCX, RDI, RSI.	Zero-extend ECX, EDI, ESI.
1111 0011 : 1010 010w		
REP OUT—Output String	Uses RDI, RCX. IO size remains 1,2 or 4 bytes.	Zero-extends ECX, EDI.
1111 0011 : 0110 111w		
R**EP STOS—**Store String	Uses RCX, RAX, RDI.	Zero-extend ECX, EAX, EDI.
1111 0011 : 1010 101w		
REPx CMPS —Compare String	Uses RCX, RDI, RSI.	Zero-extend ECX, EDI, ESI.
1111 0011 : 1010 011w		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
REPx SCAS —Scan String	Uses RCX, RAX, RSI.	Zero-extend ECX, EAX, ESI.
1111 0011 : 1010 111w		
RET—Return from Procedure (to same segment)	Promoted to 64 bits.	Operand size is fixed at 64 bits.
1100 0011		
1100 0010 : 16-bit displacement		
RET—Return from Procedure (to other segment)	See Long Mode Arch Specification.	
1100 1011		
1100 1010 : 16-bit displacement		
ROL—Rotate Left	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).
1101 000w : 11 000 reg		
1101 000w : mod 000 r/m		
1101 001w : 11 000 reg		
1101 001w : mod 000 r/m		
1100 000w : 11 000 reg : imm8 data		
1100 000w : mod 000 r/m : imm8 data		
ROR—Rotate Right	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).
1101 000w : 11 001 reg		
1101 000w : mod 001 r/m		
1101 001w : 11 001 reg		
1101 001w : mod 001 r/m		
1100 000w : 11 001 reg : imm8 data		
1100 000w : mod 001 r/m : imm8 data		
RSM—Resume from System Management Mode	(New SMM state save area : TBD.)	
0000 1111 : 1010 1010		
SAL—Shift Arithmetic Left	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
SAR—Shift Arithmetic Right	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).
1101 000w : 11 111 reg		
1101 000w : mod 111 r/m		
1101 001w : 11 111 reg		
1101 001w : mod 111 r/m		
1100 000w : 11 111 reg : imm8 data		
1100 000w : mod 111 r/m : imm8 data		
SBB—Integer Subtraction with Borrow	Promoted to 64 bits.	Zero-extend 32-bit register results.
0001 100w : 11 reg1 reg2		
0001 101w : 11 reg1 reg2		
0001 101w : mod reg r/m		
0001 100w : mod reg r/m		
1000 00sw : 11 011 reg : immediate data		
0001 110w : immediate data		
1000 00sw : mod 011 r/m : immediate data		
SCAS/**SCASB**/**SCASW**/**SCASD**—Scan String	Promoted to 64 bits.	Zero-extend 32-bit register results.
1101 111w		
SETcc—Byte Set on Condition	Same as 32-bit.	Zero-extend 32-bit register results.
0000 1111 : 1001 tttn : 11 000 reg		
0000 1111 : 1001 tttn : mod 000 r/m		
SGDT—Store Global Descriptor Table Register	Stores 8-byte base and 2-byte limit.	Stores 8-byte base and 2-byte limit.
0000 1111 : 0000 0001 : mod 000 r/m		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
SHL—Shift Left	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).
1101 000w : 11 1x0 reg		
1101 000w : mod 1x0 r/m		
1101 001w : 11 1x0 reg		
1101 001w : mod 1x0 r/m		
1100 000w : 11 1x0 reg : imm8 data		
1100 000w : mod 1x0 r/m : imm8 data		
SHLD—Double Precision Shift Left	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).
0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8		
0000 1111 : 1010 0100 : mod reg r/m : imm8		
0000 1111 : 1010 0101 : 11 reg2 reg1		
0000 1111 : 1010 0101 : mod reg r/m		
SHR—Shift Right	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).
1101 000w : 11 101 reg		
1101 000w : mod 101 r/m		
1101 001w : 11 101 reg		
1101 001w : mod 101 r/m		
1100 000w : 11 101 reg : imm8 data		
1100 000w : mod 101 r/m : imm8 data		
SHRD—Double Precision Shift Right	Promoted to 64 bits. (Uses 6-bit count).	Zero-extend 32-bit register results. (Uses 5-bit count).
0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8		
0000 1111 : 1010 1100 : mod reg r/m : imm8		
0000 1111 : 1010 1101 : 11 reg2 reg1		
0000 1111 : 1010 1101 : mod reg r/m		
SIDT—Store Interrupt Descriptor Table Register	Stores 8-byte base and 2-byte limit.	Stores 8-byte base and 2-byte limit.
0000 1111 : 0000 0001 : mod 001 r/m		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
SLDT—Store Local Descriptor Table Register		
0000 1111 : 0000 0000 : 11 000 reg	Zero extends 2 byte selector.	Zero-extend 32-bit register results.
0000 1111 : 0000 0000 : mod 000 r/m	Mode-independent.	
SMSW—Store Machine Status Word		
0000 1111 : 0000 0001 : 11 100 reg	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 0000 0001 : mod 100 r/m	Mode-independent - fixed at 2 bytes.	
STC—Set Carry Flag 1111 1001	No register result.	No register result.
STD—Set Direction Flag 1111 1101	No register result.	No register result.
STI - Set Interrupt Flag 1111 1011	No register result.	No register result.
STOS/**STOSB**/**STOSW**/**STOSD** - Store String Data 1010 101w	Promoted to 64 bits.	Zero-extend 32-bit register results.
STR—Store Task Register		
0000 1111 : 0000 0000 : 11 001 reg	Zero extends 2 byte selector.	Zero-extend 32-bit register results.
0000 1111 : 0000 0000 : mod 001 r/m	Mode-independent.	

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
SUB—Integer Subtraction	Promoted to 64 bits.	Zero-extend 32-bit register results.
0010 100w : 11 reg1 reg2		
0010 101w : 11 reg1 reg2		
0010 101w : mod reg r/m		
0010 100w : mod reg r/m		
1000 00sw : 11 101 reg : immediate data		
0010 110w : immediate data		
1000 00sw : mod 101 r/m : immediate data		
SYSCALL—Call Operating System	Promoted to 64 bits.	If EFER.LMA = 0, then zero-extend 32-bit ECX results.
0000 1111 : 0000 0101		
SYSRET—Return from Operating System	Promoted to 64 bits.	No register result.
0000 1111 : 0000 0111		
TEST—Logical Compare	No register result.	No register result.
1000 010w : 11 reg1 reg2		
1000 010w : mod reg r/m		
1111 011w : 11 000 reg : immediate data		
1010 100w : immediate data		
1111 011w : mod 000 r/m : immediate data		
UD2—Undefined instruction	No register result.	No register result.
0000 FFFF : 0000 1011		
VERR—Verify a Segment for Reading	Segment Instruction, no changes.	No register result.
0000 1111 : 0000 0000 : 11 100 reg		
0000 1111 : 0000 0000 : mod 100 r/m		
VERW—Verify a Segment for Writing	Segment Instruction, no changes.	No register result.
0000 1111 : 0000 0000 : 11 101 reg		
0000 1111 : 0000 0000 : mod 101 r/m		

Table 15. Operand Size in 64-Bit Mode (continued)

Instruction	64-Bit Operand Size	32-Bit Operand Size: Upper 32 Bits?
WAIT—Wait	No register result.	No register result.
1001 1011		
WBINVD—Writeback and Invalidate Data Cache	No register result.	No register result.
0000 1111 : 0000 1001		
WRMSR—Write to Model-Specific Register	RDX[31:00] contain MSR[63:32]. RAX[31:00] contains MSR[31:00].	No register result.
0000 1111 : 0011 0000		
XADD—Exchange and Add	Promoted to 64 bits.	Zero-extend 32-bit register results.
0000 1111 : 1100 000w : 11 reg2 reg1		
0000 1111 : 1100 000w : mod reg r/m		
XCHG—Exchange Register/Memory with Register	Promoted to 64 bits.	Zero-extend 32-bit register results.
1000 011w : 11 reg1 reg2		
1000 011w : mod reg r/m		
1001 0 reg		
XLAT/**XLATB** - Table Look-up Translation	Writes AL preserves 63:08.	Writes AL preserves 63:08.
1101 0111		
XOR—Logical Exclusive OR	Promoted to 64 bits.	Zero-extend 32-bit register results.
0011 000w : 11 reg1 reg2		
0011 001w : 11 reg1 reg2		
0011 001w : mod reg r/m		
0011 000w : mod reg r/m		
1000 00sw : 11 110 reg : immediate data		
0011 010w : immediate data		
1000 00sw : mod 110 r/m : immediate data		

Instruction Differences

The following differences exist between instructions running in 64-bit mode vs. legacy and compatibility modes. All differences apply to integer instructions. There are no differences for the x87 floating-point instructions.

Invalid Instructions

Table 16 lists instructions that are illegal in 64-bit mode. Attempted use of these instructions generates an invalid-opcode exception (#UD).

Table 16. Invalid Instructions in 64-Bit Mode

Mnemonic	Opcode (hex)	Description
AAA	37	ASCII adjust after add
AAD	D5	ASCII adjust after divide
AAM	D4	ASCII adjust after multiply
AAS	3F	ASCII adjust after subtract
ARPL	63	Adjust requested privilege level of selector
BOUND	62	Check register against bounds
CALLF Ap	9A	Call far to sel:offset in immediate operand
DAA	27	Decimal adjust after add
DAS	2F	Decimal adjust after subtract
INC and DEC	40-4F	Increment and Decrement by 1 (see "Single-Byte INC and DEC Instructions" on page 94)
INTO	CE	Interrupt 4 if overflow = 1
JMPF Ap	EA	Jump far to sel:offset in immediate operand
LAHF	9F	Load status flags into AH register
LDS	C5	Load far pointer into DS:reg pair
LES	C4	Load far pointer into ES:reg pair
POP DS	1F	Pop top of stack into DS segment
POP ES	07	Pop top of stack into ES segment
POP SS	17	Pop top of stack into SS segment

Table 16. Invalid Instructions in 64-Bit Mode (continued)

Mnemonic	Opcode (hex)	Description
POPA	61	Pop legacy GPRs (EAX–ESP)
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	60	Push legacy GPRs (EAX–ESP)
Redundant Grp1	82	Redundant encoding of group1 Eb,Ib opcodes
SAHF	9E	Store status flags from AH register
SALC	D6	Set AL according to CF (undocumented)

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, as described in "REX Prefixes" on page 18. 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).

NOP

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 XCHG EAX, EAX as a true NOP, regardless of operand size or the presence of a REX 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 will not result in a no-operation, because the default operation size is 32 bits in 64-bit mode.

MOVSXD

MOVSXD is a new instruction in 64-bit mode. It reads a fixed-size 32-bit source operand (register or memory) and sign-extends the value to 64 bits. MOVSXD is analogous to the existing MOVSXB and MOVSXW instructions.

Table 17 shows the encoding of MOVSXD. The actual size of the result written to the destination depends on the effective operand size. A REX prefix selects a 64-bit operand size, as described in "REX Prefixes" on page 18.

Table 17. MOVSXD Instruction

Mnemonic	Opcode (hex)	Description
MOVSXD r64, r/m32	63h	Move Dword to Qword, sign extension

Instructions that Reference RSP

Table 18 lists the instructions that implicitly reference RSP and default to a 64-bit operand size.

Table 18. Instructions Defaulting to 64-Bits in 64-Bit Mode

Mnemonic	Opcode (hex)	Description
ENTER	C8	Make stack frame
LEAVE	C9	Delete stack frame
POP r/m	8F /0	Pop memory (or register)
POP reg	58-5F	Pop register
POPF	9D	POP EFLAGS
PUSH imm32	68	Push sign-extended dword
PUSH imm8	6A	Push sign-extended byte
PUSH r/m	FF /6	Push memory (or register)
PUSH reg	50-57	Push register
PUSHF	9C	Push EFLAGS

Operand-size overrides are ignored for these instructions. Pushing and popping 16-bit or 32-bit stack values is not possible in 64-bit mode with the instructions listed in Table 18.

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.

The FS and GS segment-override prefixes are treated as true 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 “"Special Treatment of FS and GS Segments" on page 42” for details.

FXSAVE and FXRSTOR

The FXSAVE and FXRSTOR instructions are used to save and restore the entire FPU and MMX environment during a context switch. The legacy x86 FPU and MMX technology-based environment contains fields for storing the 16-bit code and data segments as well as the 32-bit instruction and data pointers. 64-bit FPU and MMX technology-based software, however, must be able to save and restore the full 64-bit instruction and data pointers when the FXSAVE and FXRSTOR instructions are executed.

Figure 25 shows the first 32-bytes of the legacy 32-bit FPU and MMX environment. Figure 26 shows the same 32-bytes as redefined for 64-bit software to hold the full 64-bit instruction and data pointers. All other FPU and MMX technology-based environment fields not shown are identical for legacy 32-bit and 64-bit software.

Byte	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+16	Reserved				MXCSR				Reserved		DS		Data Pointer			
+0	Reserved		CS		EIP				FOP		FTW		FSW		FCW	

Figure 25. First 32-Bytes of FXSAVE and FXRSTOR Data (32-Bit Format)

Byte	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+16	Reserved				MXCSR				Data Pointer							
+0	RIP								FOP		FTW		FSW		FCW	

Figure 26. First 32-Bytes of FXSAVE and FXRSTOR Data (64-Bit Format)

Because all long-mode interrupt handlers are executed in 64-bit mode (see "Interrupts" on page 65) a 64-bit interrupt handler must be able to save and restore the FPU and MMX technology-based environment on behalf of both compatibility-mode and 64-bit-mode software. Selection of the 32-bit or 64-bit format is accomplished by using the corresponding operand size in 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 shown in Figure 25 is used. When 64-bit software executes an FXSAVE and FXRSTOR with a 64-bit operand size, the 64-bit format shown in Figure 26 is used.

New Encodings for Control and Debug Registers

In 64-bit mode, additional encodings for control and debug registers are available. The REX.B bit is used to modify the ModRM *reg* field when that field encodes a control or debug register, as shown in Table 6 on page 20. These additional encodings enable the processor to address CR8–CR15 and DR8–DR15. One additional control register, CR8, is defined in 64-bit mode.

In the first implementation of the Hammer family of processors, CR9–CR15 and DR8–DR15 are not implemented, and CR8 becomes the TPR, described in "Task Priority" on page 70. Any attempt to access the unimplemented registers results in an invalid-opcode exception (#UD).

Appendix B Long Mode Differences

Table 19 summarizes the major differences between 64-bit mode and legacy x86 protected mode. The third column indicates whether the difference also applies to compatibility mode.

Table 19. Differences Between Long Mode and Legacy Mode

Type	Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
Application Programming	Addressing	RIP-relative addressing available	no
	Data and Address Sizes	Default data size is 32 bits	
		REX Prefix toggles data size to 64 bits	
		Default address size is 64 bits	
		Address size prefix toggles address size to 32 bits	
	Instruction Differences	Various opcodes are invalid or changed (see Table 16 on page 93)	
		MOV reg,imm32 becomes MOV reg,imm64 (with REX operand size prefix)	
		REX is always enabled	
		Direct-offset forms of MOV to or from accumulator become 64-bit offsets	
		MOVD extended to MOV 64 bits between MMX™ registers and long GPRs (with REX operand-size prefix)	

Table 19. Differences Between Long Mode and Legacy Mode (continued)

Type	Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
System Programming	x86 Modes	Real and virtual-8086 modes not supported	yes
	Task Switching	Task switching not supported	yes
	Addressing	64-bit virtual addresses	yes
		4-level paging structures	
		PAE must always be enabled	
	Segmentation	CS, DS, ES, SS segment bases are ignored	no
		CS, DS, ES, FS, GS, SS segment limits 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	
	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, as shown in Figure 9 on page 34	no
		LLDT and LTR use expanded 16-byte table entries, as shown in Figure 10 on page 34	

Appendix C Initialization Example

This section describes the steps, and provides sample code, for placing the processor into long mode.

```
mydata  segment para

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; generic data segment, holds pseudo-descriptors used
; by lgdt and lidt instructions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;
; 32-bit temporary GDT and IDT
;

pGDT32  label   fword                   ; used by lgdt
                dw      gdt32_limit     ; limit ...
                dd      gdt32_base      ; 32-bit base

pIDT32  label   fword                   ; used by lidt
                dw      idt32_limit     ; limit ...
                dd      idt32_base      ; 32-bit base


;
; 64-bit  GDT and IDT (64-bit linear base address)
;


pGDT64  label   tbyte                   ; used by lgdt
                dw      gdt64_limit     ; limit ...
                dq      gdt64_base      ; and 64-bit base

pIDT64  label   tbyte                   ; used by lidt
                dw      idt64_limit     ; limit ...
                dq      idt64_base      ; and 64-bit 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 real-mode ss:sp, in case of
;interrupts and exceptions
;
   cli
   mov   ax, seg mydata
   mov   ds, ax
   mov   ax, seg mystack
   mov   ss, ax
   mov   sp, esp0
;
; Use CPUID to determine if Long Mode feature is available
;

   mov   eax, 80000000h ; extended function 8000000h
   cpuid                ; largest extended function
   cmp   eax, 80000000h ; any function > 80000000h?
   jbe   no_long_mode   ; no extended features, no LM
   mov   eax, 80000001h ; extended features function
   cpuid                ; edx = extended features flag
   bt    edx, 29        ; test if Long Mode feature present
   jnc   no_long_mode   ; exit if no LM

;
; load GDT before entering protected mode.
;  this gdt contains at minimum:
;  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 for ss
;
; load 32-bit IDT (in case any interrupts and exceptions occur
; after entering protected mode but before enabling long mode)
;

   lgdt  ds:[pGDT32]
   lidt  ds:[pIDT32]


; enable protected mode (PE=1)

   mov   eax, 000000011h
   mov   cr0, eax


; far jump to turn protected mode on
; code16_sel points to the gdt descriptor for the code
;  currently being executed
```

```
   db    0eah  ;far jump
   dw    offset now_in_prot;
   dw    code16_sel;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; at this point we are in 16-bit protected mode
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
now_in_prot:


; set protected mode ss:esp
; stack_sel points to a gdt descriptor for a read/write data
;  segment
; 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 64-bit paging entries (PAE=1)
; (this is _required_ before activating long mode)
; notice that we don't enable paging until after long mode is
; activated

   mov   eax, cr4
   bts   eax, 5
   mov   cr4, eax

; establish Long Mode page tables by
; pointing the 64-bit cr3 to the base of the pml4 page table
; (which must be located <4GB because only 32 bits of CR3 are
; loaded when not in 64-bit mode

   mov   eax, pml4_base ; pointer to 4-level page table
   mov   cr3, eax       ;  establish PDBR ( <4GB )

;  set Long Mode enable (EFER.LME=1)

   mov   ecx, 0c0000080h   ; EFER MSR number
   rdmsr
   bts   eax, 8            ; set LME
   wrmsr                      ; write EFER
;
; enable paging and activate Long Mode (CR0.PG=1)
;
   mov   eax, cr0
   bts   eax, 31      ; set Paging Enable
   mov   cr0, eax     ; enable paging and activate Long Mode
```

```
; at this point we are in 16-bit compatibility mode
;   ( LMA=1, CS.L=0, CS.D=0 )
; Now -
; jump to 64-bit code segment
;  - the offset must be _linear_ address of the 64-bit entry point
;    because no segmentation in long mode
;    the selector points 32/64-bit code selector in the current gdt

   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 in true 64-bit code

; point the 64-bit rsp register to the linear address of
;  the stack (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 actual long mode gdt should be
; located at a linear address that's >4GB .  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

   lgdt  [pGDT64_linear]


; load 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 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 current LDT. ldt_sel should point to a 64-bit ldt
; descriptor in the current GDT

   mov   ax, ldt_sel
   lldt  ax

; using fs: and gs: prefixes on memory accesses still use the
; 32-bit fs.base and gs.base.  Reload these 2 registers before using
; the fs, 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, or use WRMSR to assign 64-bit
; base values into 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 4GB
; Because the original CR3 load was done in 32 bit mode, it could only load
; 32 bits into CR3. Thus current page tables are located in the lower 4GB of
; physical memory
; This MOV to CR3 is only needed if the actual long mode page tables
; are located above 4GB physical should be
; located at a linear address that's >4GB .
;

   mov   rax, final_pml4_ptr   ; point to PML4
   mov   cr3, rax              ; load 64-bit CR3

;  enable interrupts
         sti                              ;enabled INTR


      <insert 64-bit code here>
```

Appendix D Implementation Considerations

This section describes software considerations specific to the first implementation of the Hammer family of processors. It is possible, but not guaranteed, that these same issues will also be applicable to future implementations of the x86-86 architecture.

Address Size

In the first implementation of the Hammer family of processors, the supported virtual-address size in long mode is 48 bits (30h) and the physical-address size is 40 bits (28h). See "CPUID" on page 30 for details.

Operand Alignment

The first implementation of the Hammer family of processors has a penalty for loading data that crosses a cache-line (64-byte) boundary. The minimum penalty is one cycle. If the load hits a previous store that has not yet written the data cache, the penalty can be greater.

This alignment penalty becomes an issue in long mode, because of the more-frequent occurrence of 8-byte data than in legacy mode. For optimal performance, the compiler should ensure that 8-byte data does not cross a cache-block boundary.

The compiler should also be careful not to let items on the stack cross a cache-line boundary. Stack-alignment issues exist in all operand sizes (16-, 32-, and 64-bit) and modes. They are more acute in 64-bit mode, because a mixture of 4-byte data items and 8-byte stack pointers might be pushed onto the stack. For best performance, the compiler should keep procedure locals and function parameters aligned on the stack with respect to cache-line (64-byte) boundaries.

CR8 Interactions with APIC

The first implementation of the Hammer family of processors includes an external interrupt controller (EIC) based on an x86 local advanced programmable interrupt controller (APIC). Some aspects of this local APIC affect the operation of the architecturally defined task priority register (CR8.TPR), described in "Task Priority Register (TPR)" on page 33.

The notable CR8 and APIC interactions are:

- The processor powers up with the local APIC disabled.
- The APIC must be enabled for CR8 to function as the TPR for the following:
  - Writes to CR8 are reflected into the APIC's TPR register.
  - Writes to APIC's TPR are reflected into CR8.

The interrupt priority, to which CR8.TPR is compared, is determined by the following equation:

```
interrupt priority = (interrupt vector)/16
```

In other words, the interrupt priority is determined by the high-order interrupt vector bits 7:4.

See the *AMD Athlon™ Processors BIOS, Software, and Debug Developer's Guide* for complete information regarding the local APIC.

Physical Address Fields in MSRs

Memory Type Range Registers. The memory type range registers (MTRRs) are legacy MSRs that apply memory-type classifications to ranges of physical memory. Eight pairs of variable range MTRRs are defined in the first implementation of the Hammer family of processors, each pair consisting of a physical base address and type register (MTRRphysBase) and a physical address range mask register (MTRRphysMask).

The legacy MTRRs are architecturally defined as 64 bits and can accommodate the maximum 52-bit physical address allowed by the long mode architecture. The MTRRs in the first implementation of the Hammer family of processors are 40 bits wide and can hold the 40-bit physical address supported by the implementation (see Figure 27). Bits 63:40 in the MTRRphysBase and MTRRphysMask registers are reserved. The processor will generate a #GP fault if software attempts to set any of the reserved MTTR bits to 1.

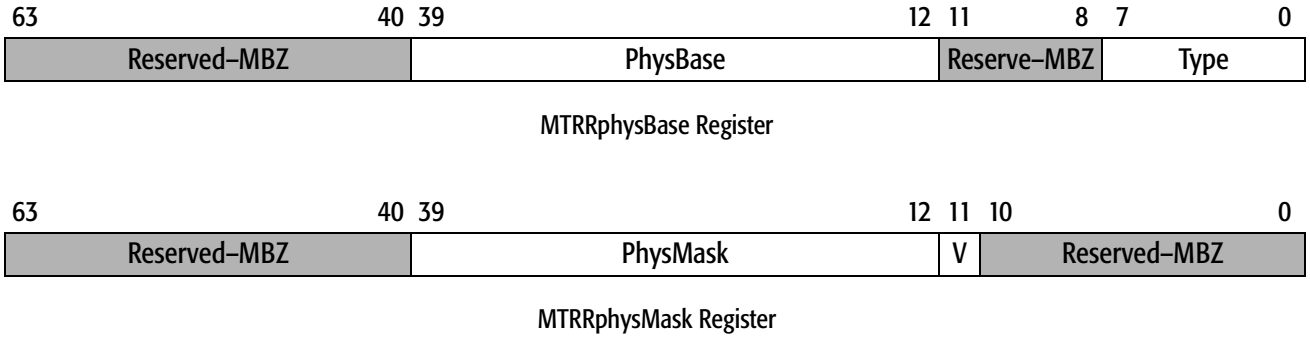


Figure 27. MTRRphysBase and MTRRphysMask Register Pair Formats

When the MTTRs are in use, the processor references the entire 40-bit value in both registers regardless of mode. Legacy mode software is responsible for writing MTTR bits 39:36 with 0's to ensure the registers operate properly.

Other MSRs. A number of other model-specific registers (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 in legacy mode. Previous implementations, however, support a maximum address size of 36 bits.

MSRs that hold physical addresses are increased in size to 40 bits in the first implementation of the Hammer family of processors. This means that the MSRs can hold the 40-bit physical address supported by the implementation. Bits 63:40 in those registers are reserved, and the processor will generate a #GP fault if software attempts to set any of the reserved MSR bits to 1.

When physical addresses are read from MSRs by the processor, the entire 40-bit value is read regardless of the operating mode. Legacy mode software is responsible for writing physical address values into the MSRs such that the implemented bits above bit 36 are cleared to zero. This ensures the features using the MSR contents operate properly. In the first implementation of the Hammer family of processors, legacy software is responsible for clearing physical address bits 39:36 to zero when writing those physical addresses to an MSR.