Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 3B: System Programming Guide, Part 2

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference, A-L*, Order Number 253666; *Instruction Set Reference, M-U*, Order Number 253667; *Instruction Set Reference, V*, Order Number 326018; *Instruction Set Reference*, W-Z, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers,* Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Order Number: 253669-083US
March 2024

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

All product plans and roadmaps are subject to change without notice.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.

CHAPTER 15
POWER AND THERMAL MANAGEMENT

This chapter describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

15.1 ENHANCED INTEL SPEEDSTEP® TECHNOLOGY

Enhanced Intel SpeedStep® Technology was introduced in the Pentium M processor. The technology enables the management of processor power consumption via performance state transitions. These states are defined as discrete operating points associated with different voltages and frequencies.

Enhanced Intel SpeedStep Technology differs from previous generations of Intel SpeedStep® Technology in two ways:

- Centralization of the control mechanism and software interface in the processor by using model-specific registers.
- Reduced hardware overhead; this permits more frequent performance state transitions.

Previous generations of the Intel SpeedStep Technology require processors to be a deep sleep state, holding off bus master transfers for the duration of a performance state transition. Performance state transitions under the Enhanced Intel SpeedStep Technology are discrete transitions to a new target frequency.

Support is indicated by CPUID, using ECX feature bit 07. Enhanced Intel SpeedStep Technology is enabled by setting IA32_MISC_ENABLE MSR, bit 16. On reset, bit 16 of IA32_MISC_ENABLE MSR is cleared.

15.1.1 Software Interface For Initiating Performance State Transitions

State transitions are initiated by writing a 16-bit value to the IA32_PERF_CTL register, see Figure 15-2. If a transition is already in progress, transition to a new value will subsequently take effect.

Reads of IA32_PERF_CTL determine the last targeted operating point. The current operating point can be read from IA32_PERF_STATUS. IA32_PERF_STATUS is updated dynamically.

The 16-bit encoding that defines valid operating points is model-specific. Applications and performance tools are not expected to use either IA32_PERF_CTL or IA32_PERF_STATUS and should treat both as reserved. Performance monitoring tools can access model-specific events and report the occurrences of state transitions.

15.2 P-STATE HARDWARE COORDINATION

The Advanced Configuration and Power Interface (ACPI) defines performance states (P-states) that are used to facilitate system software's ability to manage processor power consumption. Different P-states correspond to different performance levels that are applied while the processor is actively executing instructions. Enhanced Intel SpeedStep Technology supports P-states by providing software interfaces that control the operating frequency and voltage of a processor.

With multiple processor cores residing in the same physical package, hardware dependencies may exist for a subset of logical processors on a platform. These dependencies may impose requirements that impact the coordination of P-state transitions. As a result, multi-core processors may require an OS to provide additional software support for coordinating P-state transitions for those subsets of logical processors.

ACPI firmware can choose to expose P-states as dependent and hardware-coordinated to OS power management (OSPM) policy. To support OSPMs, multi-core processors must have additional built-in support for P-state hardware coordination and feedback.

Intel 64 and IA-32 processors with dependent P-states amongst a subset of logical processors permit hardware coordination of P-states and provide a hardware-coordination feedback mechanism using IA32_MPERF MSR and

IA32_APERF MSR. See Figure 15-1 for an overview of the two 64-bit MSRs and the bullets below for a detailed description.

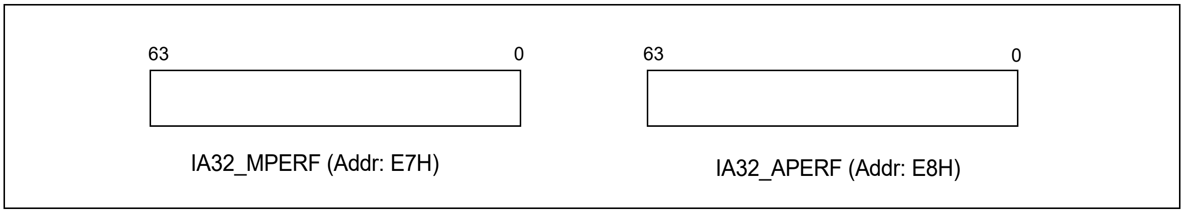


Figure 15-1. IA32_MPERF MSR and IA32_APERF MSR for P-state Coordination

- Use CPUID to check the P-State hardware coordination feedback capability bit. CPUID.06H.ECX[Bit 0] = 1 indicates IA32_MPERF MSR and IA32_APERF MSR are present.
- IA32_MPERF MSR (E7H) increments in proportion to a fixed frequency, which is configured when the processor is booted.
- IA32_APERF MSR (E8H) increments in proportion to actual performance, while accounting for hardware coordination of P-state and TM1/TM2; or software initiated throttling.
- The MSRs are per logical processor; they measure performance only when the targeted processor is in the C0 state.
- Only the IA32_APERF/IA32_MPERF ratio is architecturally defined; software should not attach meaning to the content of the individual of IA32_APERF or IA32_MPERF MSRs.
- When either MSR overflows, both MSRs are reset to zero and continue to increment.
- Both MSRs are full 64-bits counters. Each MSR can be written to independently. However, software should follow the guidelines illustrated in Example 15-1.

If P-states are exposed by the BIOS as hardware coordinated, software is expected to confirm processor support for P-state hardware coordination feedback and use the feedback mechanism to make P-state decisions. The OSPM is expected to either save away the current MSR values (for determination of the delta of the counter ratio at a later time) or reset both MSRs (execute WRMSR with 0 to these MSRs individually) at the start of the time window used for making the P-state decision. When not resetting the values, overflow of the MSRs can be detected by checking whether the new values read are less than the previously saved values.

Example 15-1 demonstrates steps for using the hardware feedback mechanism provided by IA32_APERF MSR and IA32_MPERF MSR to determine a target P-state.

Example 15-1. Determine Target P-state From Hardware Coordinated Feedback

```
DWORD PercentBusy; // Percentage of processor time not idle.
    // Measure “PercentBusy“ during previous sampling window.
    // Typically, “PercentBusy“ is measure over a time scale suitable for
    // power management decisions
    //
    // RDMSR of MCNT and ACNT should be performed without delay.
    // Software needs to exercise care to avoid delays between
    // the two RDMSRs (for example, interrupts).
    MCNT = RDMSR(IA32_MPERF);
    ACNT = RDMSR(IA32_APERF);

    // PercentPerformance indicates the percentage of the processor
    // that is in use. The calculation is based on the PercentBusy,
    // that is the percentage of processor time not idle and the P-state
    // hardware coordinated feedback using the ACNT/MCNT ratio.
    // Note that both values need to be calculated over the same
    // time window.
        PercentPerformance = PercentBusy * (ACNT/MCNT);


    // This example does not cover the additional logic or algorithms
    //  necessary to coordinate multiple logical processors to a target P-state.

    TargetPstate = FindPstate(PercentPerformance);

    if (TargetPstate ≠ currentPstate) {
        SetPState(TargetPstate);
    }
    // WRMSR of MCNT and ACNT should be performed without delay.
    // Software needs to exercise care to avoid delays between
    // the two WRMSRs (for example, interrupts).
     WRMSR(IA32_MPERF, 0);
     WRMSR(IA32_APERF, 0);
```

15.3 SYSTEM SOFTWARE CONSIDERATIONS AND OPPORTUNISTIC PROCESSOR PERFORMANCE OPERATION

An Intel 64 processor may support a form of processor operation that takes advantage of design headroom to opportunistically increase performance. The Intel® Turbo Boost Technology can convert thermal headroom into higher performance across multi-threaded and single-threaded workloads. The Intel® Dynamic Acceleration Technology feature can convert thermal headroom into higher performance if only one thread is active.

15.3.1 Intel® Dynamic Acceleration Technology

The Intel Core 2 Duo processor T 7700 introduces Intel Dynamic Acceleration Technology. Intel Dynamic Acceleration Technology takes advantage of thermal design headroom and opportunistically allows a single core to operate at a higher performance level when the operating system requests increased performance.

15.3.2 System Software Interfaces for Opportunistic Processor Performance Operation

Opportunistic processor performance operation, applicable to Intel Dynamic Acceleration Technology and Intel® Turbo Boost Technology, has the following characteristics:

- A transition from a normal state of operation (e.g., Intel Dynamic Acceleration Technology/Turbo mode disengaged) to a target state is not guaranteed, but may occur opportunistically after the corresponding enable mechanism is activated, the headroom is available and certain criteria are met.
- The opportunistic processor performance operation is generally transparent to most application software.
- System software (BIOS and Operating system) must be aware of hardware support for opportunistic processor performance operation and may need to temporarily disengage opportunistic processor performance operation when it requires more predictable processor operation.
- When opportunistic processor performance operation is engaged, the OS should use hardware coordination feedback mechanisms to prevent un-intended policy effects if it is activated during inappropriate situations.

15.3.2.1 Discover Hardware Support and Enabling of Opportunistic Processor Performance Operation

If an Intel 64 processor has hardware support for opportunistic processor performance operation, the power-on default state of IA32_MISC_ENABLE[38] indicates the presence of such hardware support. For Intel 64 processors that support opportunistic processor performance operation, the default value is 1, indicating its presence. For processors that do not support opportunistic processor performance operation, the default value is 0. The power-on default value of IA32_MISC_ENABLE[38] allows BIOS to detect the presence of hardware support of opportunistic processor performance operation.

IA32_MISC_ENABLE[38] is shared across all logical processors in a physical package. It is written by BIOS during platform initiation to enable/disable opportunistic processor performance operation in conjunction of OS power management capabilities, see Section 15.3.2.2. BIOS can set IA32_MISC_ENABLE[38] with 1 to disable opportunistic processor performance operation; it must clear the default value of IA32_MISC_ENABLE[38] to 0 to enable opportunistic processor performance operation. OS and applications must use CPUID leaf 06H if it needs to detect processors that have opportunistic processor performance operation enabled.

When CPUID is executed with EAX = 06H on input, Bit 1 of EAX in Leaf 06H (i.e., CPUID.06H:EAX[1]) indicates opportunistic processor performance operation, such as Intel Dynamic Acceleration Technology, has been enabled by BIOS.

Opportunistic processor performance operation can be disabled by setting bit 38 of IA32_MISC_ENABLE. This mechanism is intended for BIOS only. If IA32_MISC_ENABLE[38] is set, CPUID.06H:EAX[1] will return 0.

15.3.2.2 OS Control of Opportunistic Processor Performance Operation

There may be phases of software execution in which system software cannot tolerate the non-deterministic aspects of opportunistic processor performance operation. For example, when calibrating a real-time workload to make a CPU reservation request to the OS, it may be undesirable to allow the possibility of the processor delivering increased performance that cannot be sustained after the calibration phase.

System software can temporarily disengage opportunistic processor performance operation by setting bit 32 of the IA32_PERF_CTL MSR (0199H), using a read-modify-write sequence on the MSR. The opportunistic processor performance operation can be re-engaged by clearing bit 32 in IA32_PERF_CTL MSR, using a read-modify-write sequence. The DISENGAGE bit in IA32_PERF_CTL is not reflected in bit 32 of the IA32_PERF_STATUS MSR (0198H), and it is not shared between logical processors in a physical package. In order for OS to engage Intel Dynamic Acceleration Technology/Turbo mode, the BIOS must:

- Enable opportunistic processor performance operation, as described in Section 15.3.2.1.
- Expose the operating points associated with Intel Dynamic Acceleration Technology/Turbo mode to the OS.

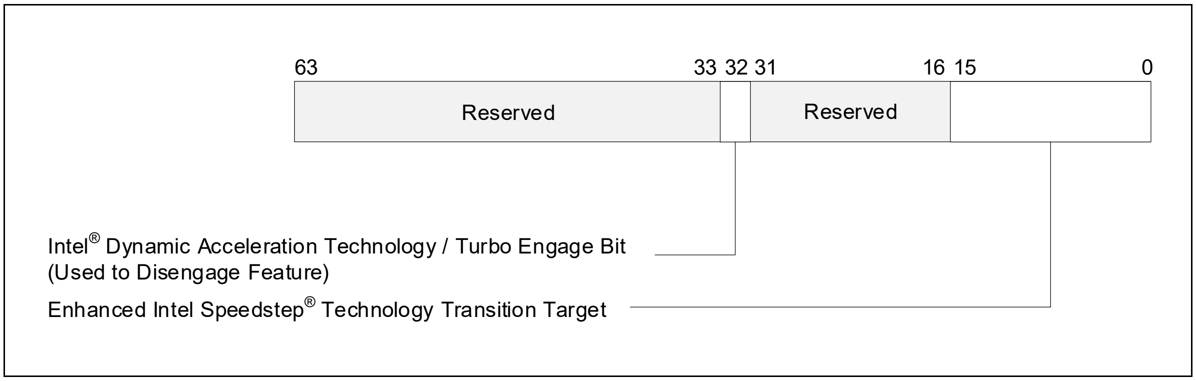

63
33 32 31
16 15
0
Reserved
Reserved
Intel® Dynamic Acceleration Technology / Turbo Engage Bit
(Used to Disengage Feature)
Enhanced Intel Speedstep® Technology Transition Target

Figure 15-2. IA32_PERF_CTL Register

15.3.2.3 Required Changes to OS Power Management P-State Policy

Intel Dynamic Acceleration Technology and Intel Turbo Boost Technology can provide opportunistic performance greater than the performance level corresponding to the Processor Base frequency of the processor (see CPUID's processor frequency information). System software can use a pair of MSRs to observe performance feedback. Software must query for the presence of IA32_APERF and IA32_MPERF (see Section 15.2). The ratio between IA32_APERF and IA32_MPERF is architecturally defined and a value greater than unity indicates performance increase occurred during the observation period due to Intel Dynamic Acceleration Technology. Without incorporating such performance feedback, the target P-state evaluation algorithm can result in a non-optimal P-state target.

There are other scenarios under which OS power management may want to disable Intel Dynamic Acceleration Technology, some of these are listed below:

- When engaging ACPI defined passive thermal management, it may be more effective to disable Intel Dynamic Acceleration Technology for the duration of passive thermal management.
- When the user has indicated a policy preference of power savings over performance, OS power management may want to disable Intel Dynamic Acceleration Technology while that policy is in effect.

15.3.3 Intel® Turbo Boost Technology

Intel Turbo Boost Technology is supported in Intel Core i7 processors and Intel Xeon processors based on Nehalem microarchitecture. It uses the same principle of leveraging thermal headroom to dynamically increase processor performance for single-threaded and multi-threaded/multi-tasking environment. The programming interface described in Section 15.3.2 also applies to Intel Turbo Boost Technology.

15.3.4 Performance and Energy Bias Hint Support

Intel 64 processors may support additional software hint to guide the hardware heuristic of power management features to favor increasing dynamic performance or conserve energy consumption.

Software can detect the processor's capability to support the performance-energy bias preference hint by examining bit 3 of ECX in CPUID leaf 6. The processor supports this capability if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H).

Software can program the lowest four bits of IA32_ENERGY_PERF_BIAS MSR with a value from 0 - 15. The values represent a sliding scale, where a value of 0 (the default reset value) corresponds to a hint preference for highest performance and a value of 15 corresponds to the maximum energy savings. A value of 7 roughly translates into a hint to balance performance with energy consumption.

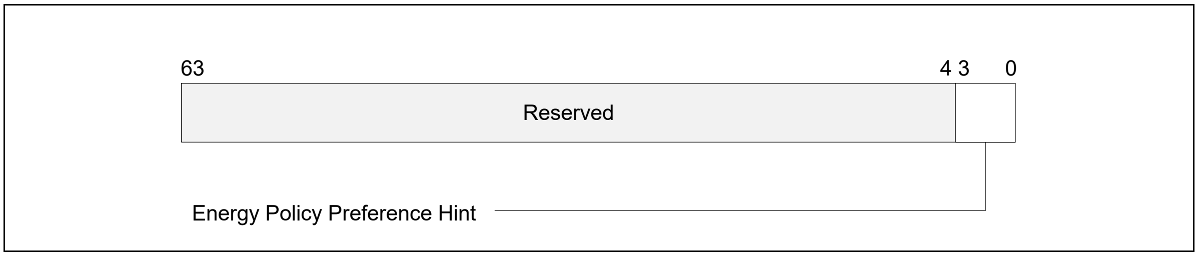


Figure 15-3. IA32_ENERGY_PERF_BIAS Register

The layout of IA32_ENERGY_PERF_BIAS is shown in Figure 15-3. The scope of IA32_ENERGY_PERF_BIAS is per logical processor, which means that each of the logical processors in the package can be programmed with a different value. This may be especially important in virtualization scenarios, where the performance / energy requirements of one logical processor may differ from the other. Conflicting "hints" from various logical processors at higher hierarchy level will be resolved in favor of performance over energy savings.

Software can use whatever criteria it sees fit to program the MSR with an appropriate value. However, the value only serves as a hint to the hardware and the actual impact on performance and energy savings is model specific.

15.4 HARDWARE-CONTROLLED PERFORMANCE STATES (HWP)

Intel processors may contain support for Hardware-Controlled Performance States (HWP), which autonomously selects performance states while utilizing OS supplied performance guidance hints. The Enhanced Intel Speed-Step® Technology provides a means for the OS to control and monitor discrete frequency-based operating points via the IA32_PERF_CTL and IA32_PERF_STATUS MSRs.

In contrast, HWP is an implementation of the ACPI-defined Collaborative Processor Performance Control (CPPC), which specifies that the platform enumerates a continuous, abstract unit-less, performance value scale that is not tied to a specific performance state / frequency by definition. While the enumerated scale is roughly linear in terms of a delivered integer workload performance result, the OS is required to characterize the performance value range to comprehend the delivered performance for an applied workload.

When HWP is enabled, the processor autonomously selects performance states as deemed appropriate for the applied workload and with consideration of constraining hints that are programmed by the OS. These OS-provided hints include minimum and maximum performance limits, preference towards energy efficiency or performance, and the specification of a relevant workload history observation time window. The means for the OS to override HWP's autonomous selection of performance state with a specific desired performance target is also provided, however, the effective frequency delivered is subject to the result of energy efficiency and performance optimizations.

15.4.1 HWP Programming Interfaces

The programming interfaces provided by HWP include the following:

- The CPUID instruction allows software to discover the presence of HWP support in an Intel processor. Specifically, execute CPUID instruction with EAX=06H as input will return 5 bit flags covering the following aspects in bits 7 through 11 of CPUID.06H:EAX:
  - Availability of HWP baseline resource and capability, CPUID.06H:EAX[bit 7]: If this bit is set, HWP provides several new architectural MSRs: IA32_PM_ENABLE, IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS.
  - Availability of HWP Notification upon dynamic Guaranteed Performance change, CPUID.06H:EAX[bit 8]: If this bit is set, HWP provides IA32_HWP_INTERRUPT MSR to enable interrupt generation due to dynamic Performance changes and excursions.
  - Availability of HWP Activity window control, CPUID.06H:EAX[bit 9]: If this bit is set, HWP allows software to program activity window in the IA32_HWP_REQUEST MSR.
  - Availability of HWP energy/performance preference control, CPUID.06H:EAX[bit 10]: If this bit is set, HWP allows software to set an energy/performance preference hint in the IA32_HWP_REQUEST MSR.
  - Availability of HWP package level control, CPUID.06H:EAX[bit 11]:If this bit is set, HWP provides the IA32_HWP_REQUEST_PKG MSR to convey OS Power Management’s control hints for all logical processors in the physical package.

Table 15-1. Architectural and Non-Architectural MSRs Related to HWP

Address	Architectural	Register Name	Description
770H	Y	IA32_PM_ENABLE	Enable/Disable HWP.
771H	Y	IA32_HWP_CAPABILITIES	Enumerates the HWP performance range (static and dynamic).
772H	Y	IA32_HWP_REQUEST_PKG	Conveys OSPM’s control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for all logical processor in the physical package.
773H	Y	IA32_HWP_INTERRUPT	Controls HWP native interrupt generation (Guaranteed Performance changes, excursions).
774H	Y	IA32_HWP_REQUEST	Conveys OSPM’s control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for a single logical processor.
775H	Y	IA32_HWP_PECI_REQUEST_INFO	Conveys embedded system controller requests to override some of the OS HWP Request settings via the PECI mechanism.
777H	Y	IA32_HWP_STATUS	Status bits indicating changes to Guaranteed Performance and excursions to Minimum Performance.
19CH	Y	IA32_THERM_STATUS[bits 15:12]	Conveys reasons for performance excursions.
64EH	N	MSR_PPERF	Productive Performance Count.

- Additionally, HWP may provide a non-architectural MSR, MSR_PPERF, which provides a quantitative metric to software of hardware's view of workload scalability. This hardware's view of workload scalability is implementation specific.

15.4.2 Enabling HWP

The layout of the IA32_PM_ENABLE MSR is shown in Figure 15-4. The bit fields are described below:

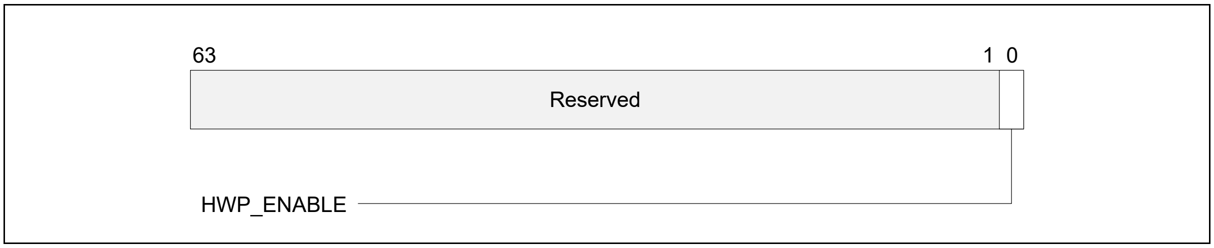


Figure 15-4. IA32_PM_ENABLE MSR

- **HWP_ENABLE (bit 0, R/W1Once)** — Software sets this bit to enable HWP with autonomous selection of processor P-States. When set, the processor will disregard input from the legacy performance control interface (IA32_PERF_CTL). Note this bit can only be enabled once from the default value. Once set, writes to the HWP_ENABLE bit are ignored. Only RESET will clear this bit. Default = zero (0).
- Bits 63:1 are reserved and must be zero.

After software queries CPUID and verifies the processor's support of HWP, system software can write 1 to IA32_PM_ENABLE.HWP_ENABLE (bit 0) to enable hardware controlled performance states. The default value of IA32_PM_ENABLE MSR at power-on is 0, i.e., HWP is disabled.

Additional MSRs associated with HWP may only be accessed after HWP is enabled, with the exception of IA32_HWP_INTERRUPT and MSR_PPERF. Accessing the IA32_HWP_INTERRUPT MSR requires only HWP is present as enumerated by CPUID but does not require enabling HWP.

IA32_PM_ENABLE is a package level MSR, i.e., writing to it from any logical processor within a package affects all logical processors within that package.

15.4.3 HWP Performance Range and Dynamic Capabilities

The OS reads the IA32_HWP_CAPABILITIES MSR to comprehend the limits of the HWP-managed performance range as well as the dynamic capability, which may change during processor operation. The enumerated performance range values reported by IA32_HWP_CAPABILITIES directly map to initial frequency targets (prior to workload-specific frequency optimizations of HWP). However the mapping is processor family specific.

The layout of the IA32_HWP_CAPABILITIES MSR is shown in Figure 15-5. The bit fields are described below:

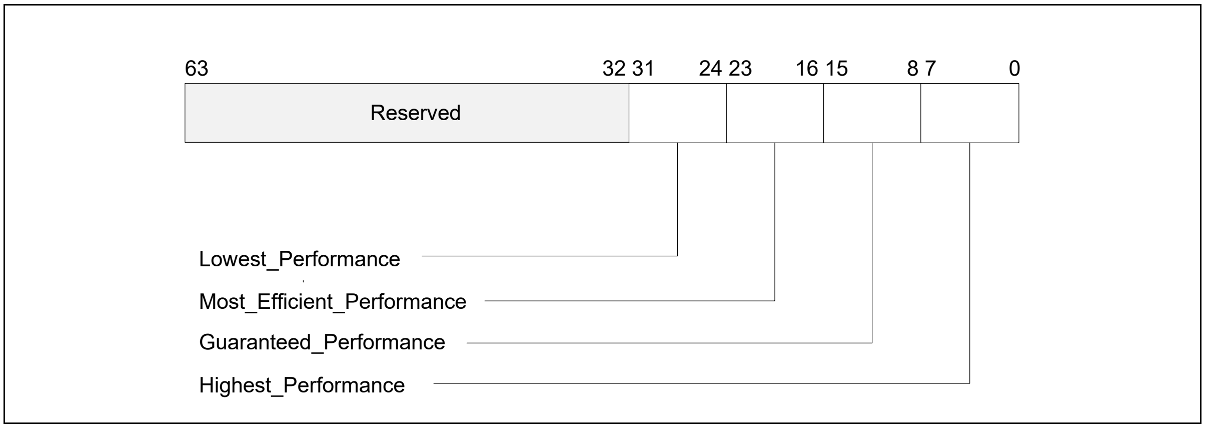


Figure 15-5. IA32_HWP_CAPABILITIES Register

- **Highest_Performance (bits 7:0, RO)** — Value for the maximum non-guaranteed performance level.
- **Guaranteed_Performance (bits 15:8, RO)** — Current value for the guaranteed performance level. This value can change dynamically as a result of internal or external constraints, e.g., thermal or power limits.
- **Most_Efficient_Performance (bits 23:16, RO)** — Current value of the most efficient performance level. This value can change dynamically as a result of workload characteristics.
- **Lowest_Performance (bits 31:24, RO)** — Value for the lowest performance level that software can program to IA32_HWP_REQUEST.
- Bits 63:32 are reserved and must be zero.

The value returned in the **Guaranteed_Performance** field is hardware's best-effort approximation of the available performance given current operating constraints. Changes to the Guaranteed_Performance value will primarily occur due to a shift in operational mode. This includes a power or other limit applied by an external agent, e.g., RAPL (see Figure 15.10.1), or the setting of a Configurable TDP level (see model-specific controls related to Programmable TDP Limit in Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.). Notification of a change to the Guaranteed_Performance occurs via interrupt (if configured) and the IA32_HWP_Status MSR. Changes to Guaranteed_Performance are indicated when a macroscopically meaningful change in performance occurs i.e., sustained for greater than one second. Consequently, notification of a change in Guaranteed Performance will typically occur no more frequently than once per second. Rapid changes in platform configuration, e.g., docking/undocking, with corresponding changes to a Configurable TDP level could potentially cause more frequent notifications.

The value returned by the **Most_Efficient_Performance** field provides the OS with an indication of the practical lower limit for the IA32_HWP_REQUEST. The processor may not honor IA32_HWP_REQUEST.Maximum Performance settings below this value.

15.4.4 Managing HWP

15.4.4.1 IA32_HWP_REQUEST MSR (Address: 774H Logical Processor Scope)

Typically, the operating system controls HWP operation for each logical processor via the writing of control hints / constraints to the IA32_HWP_REQUEST MSR. The layout of the IA32_HWP_REQUEST MSR is shown in Figure 15-6. The bit fields are described below Figure 15-6.

Operating systems can control HWP by writing both IA32_HWP_REQUEST and IA32_HWP_REQUEST_PKG MSRs (see Section 15.4.4.2). Five valid bits within the IA32_HWP_REQUEST MSR let the operating system flexibly select which of its five hint / constraint fields should be derived by the processor from the IA32_HWP_REQUEST MSR and which should be derived from the IA32_HWP_REQUEST_PKG MSR. These five valid bits are supported if CPUID[6].EAX[17] is set.

When the IA32_HWP_REQUEST MSR Package Control bit is set, any valid bit that is NOT set indicates to the processor to use the respective field value from the IA32_HWP_REQUEST_PKG MSR. Otherwise, the values are derived from the IA32_HWP_REQUEST MSR. The valid bits are ignored when the IA32_HWP_REQUEST MSR Package Control bit is zero.

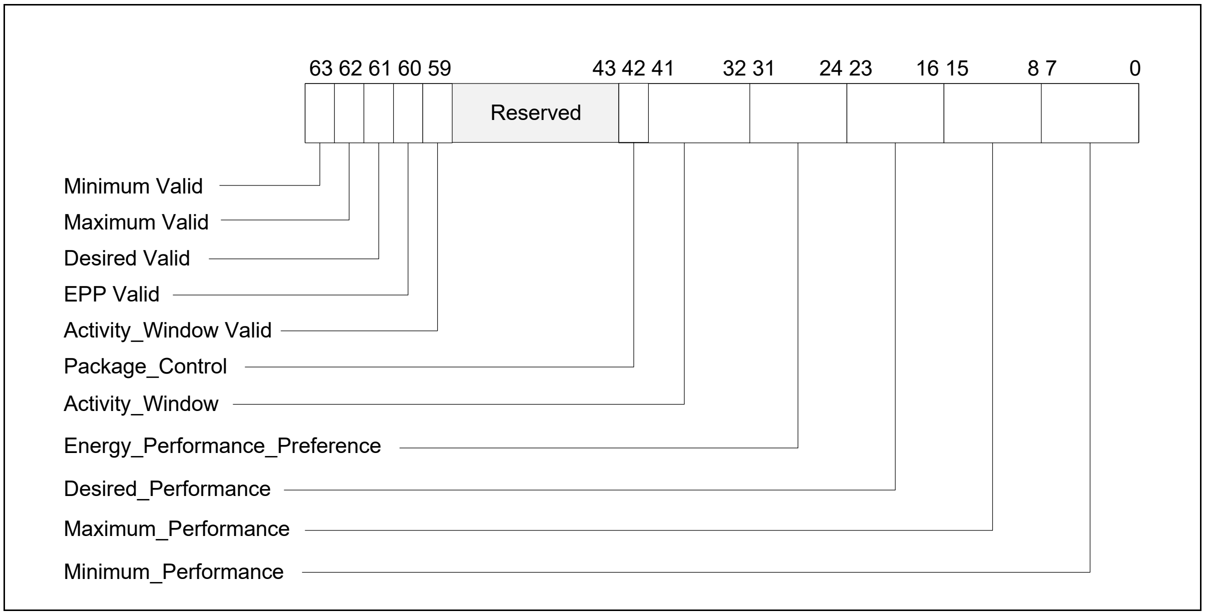


Figure 15-6. IA32_HWP_REQUEST Register

- **Minimum_Performance (bits 7:0, RW)** — Conveys a hint to the HWP hardware. The OS programs the minimum performance hint to achieve the required quality of service (QOS) or to meet a service level agreement (SLA) as needed. Note that an excursion below the level specified is possible due to hardware constraints. The default value of this field is IA32_HWP_CAPABILITIES.Lowest_Performance.
- **Maximum_Performance (bits 15:8, RW)** — Conveys a hint to the HWP hardware. The OS programs this field to limit the maximum performance that is expected to be supplied by the HWP hardware. Excursions above the limit requested by OS are possible due to hardware coordination between the processor cores and other components in the package. The default value of this field is IA32_HWP_CAPABILITIES.Highest_Performance.
- **Desired_Performance (bits 23:16, RW)** — Conveys a hint to the HWP hardware. When set to zero, hardware autonomous selection determines the performance target. When set to a non-zero value (between the range of Lowest_Performance and Highest_Performance of IA32_HWP_CAPABILITIES) conveys an explicit performance request hint to the hardware; effectively disabling HW Autonomous selection. The Desired_Performance input is non-constraining in terms of Performance and Energy Efficiency optimizations, which are independently controlled. The default value of this field is 0.
- **Energy_Performance_Preference (bits 31:24, RW)** — Conveys a hint to the HWP hardware. The OS may write a range of values from 0 (performance preference) to 0FFH (energy efficiency preference) to influence the rate of performance increase /decrease and the result of the hardware's energy efficiency and performance optimizations. The default value of this field is 80H. Note: If CPUID.06H:EAX[bit 10] indicates that this field is not supported, HWP uses the value of the IA32_ENERGY_PERF_BIAS MSR to determine the energy efficiency / performance preference.
- **Activity_Window (bits 41:32, RW)** — Conveys a hint to the HWP hardware specifying a moving workload history observation window for performance/frequency optimizations. If 0, the hardware will determine the appropriate window size. When writing a non-zero value to this field, this field is encoded in the format of bits 38:32 as a 7-bit mantissa and bits 41:39 as a 3-bit exponent value in powers of 10. The resultant value is in microseconds. Thus, the minimal/maximum activity window size is 1 microsecond/1270 seconds. Combined with the Energy_Performance_Preference input, Activity_Window influences the rate of performance increase

/ decrease. This non-zero hint only has meaning when Desired_Performance = 0. The default value of this field is 0.

- **Package_Control (bit 42, RW)** — When set, causes this logical processor's IA32_HWP_REQUEST control inputs to be derived from the IA32_HWP_REQUEST_PKG MSR.
- Bits 58:43 are reserved and must be zero.
- **Activity_Window Valid (bit 59, RW)** — When set, indicates to the processor to derive the Activity Window field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **EPP Valid (bit 60, RW)** — When set, indicates to the processor to derive the EPP field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **Desired Valid (bit 61, RW)** — When set, indicates to the processor to derive the Desired Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **Maximum Valid (bit 62, RW)** — When set, indicates to the processor to derive the Maximum Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.
- **Minimum Valid (bit 63, RW)** — When set, indicates to the processor to derive the Minimum Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.

The HWP hardware clips and resolves the field values as necessary to the valid range. Reads return the last value written not the clipped values.

Processors may support a subset of IA32_HWP_REQUEST fields as indicated by CPUID. Reads of non-supported fields will return 0. Writes to non-supported fields are ignored.

The OS may override HWP's autonomous selection of performance state with a specific performance target by setting the Desired_Performance field to a non-zero value, however, the effective frequency delivered is subject to the result of energy efficiency and performance optimizations, which are influenced by the Energy Performance Preference field.

Software may disable all hardware optimizations by setting Minimum_Performance = Maximum_Performance (subject to package coordination).

Note: The processor may run below the Minimum_Performance level due to hardware constraints including: power, thermal, and package coordination constraints. The processor may also run below the Minimum_Performance level for short durations (few milliseconds) following C-state exit, and when Hardware Duty Cycling (see Section 15.5) is enabled.

When the IA32_HWP_REQUEST MSR is set to fast access mode, writes of this MSR are posted, i.e., the WRMSR instruction retires before the data reaches its destination within the processor. It may retire even before all preceding IA stores are globally visible, i.e., it is not an architecturally serializing instruction anymore (no store fence). A new CPUID bit indicates this new characteristic of the IA32_HWP_REQUEST MSR (see Section 15.4.8 for additional details).

15.4.4.2 IA32_HWP_REQUEST_PKG MSR (Address: 772H Package Scope)

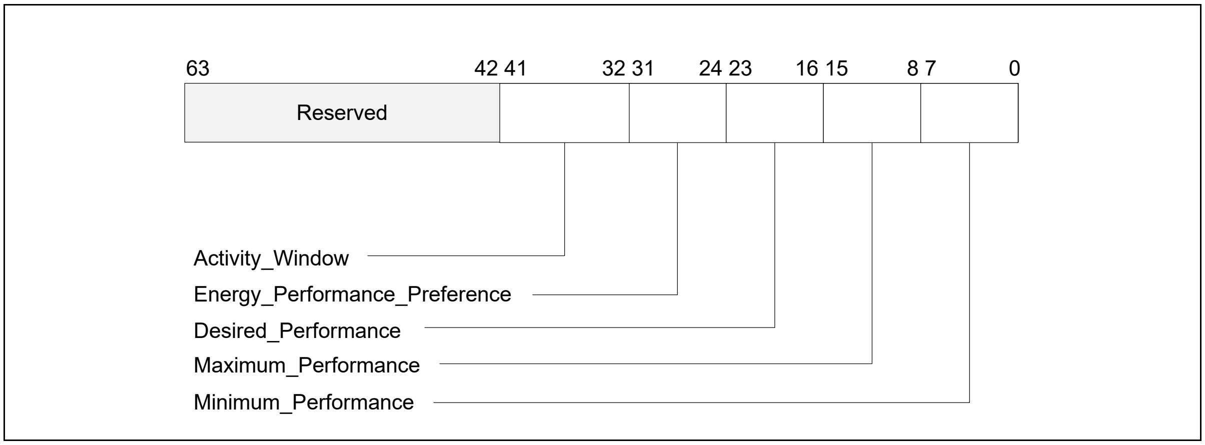


Figure 15-7. IA32_HWP_REQUEST_PKG Register

The structure of the IA32_HWP_REQUEST_PKG MSR (package-level) is identical to the IA32_HWP_REQUEST MSR with the exception of the the Package Control bit field and the five valid bit fields, which do not exist in the IA32_HWP_REQUEST_PKG MSR. Field values written to this MSR apply to all logical processors within the physical package with the exception of logical processors whose IA32_HWP_REQUEST.Package Control field is clear (zero). Single P-state Control mode is only supported when IA32_HWP_REQUEST_PKG is not supported.

15.4.4.3 IA32_HWP_PECI_REQUEST_INFO MSR (Address 775H Package Scope)

When an embedded system controller is integrated in the platform, it can override some of the OS HWP Request settings via the PECI mechanism. PECI initiated settings take precedence over the relevant fields in the IA32_HWP_REQUEST MSR and in the IA32_HWP_REQUEST_PKG MSR, irrespective of the Package Control bit or the Valid Bit values described above. PECI can independently control each of: Minimum Performance, Maximum Performance and EPP fields. This MSR contains both the PECI induced values and the control bits that indicate whether the embedded controller actually set the processor to use the respective value.

PECI override is supported if CPUID[6].EAX[16] is set.

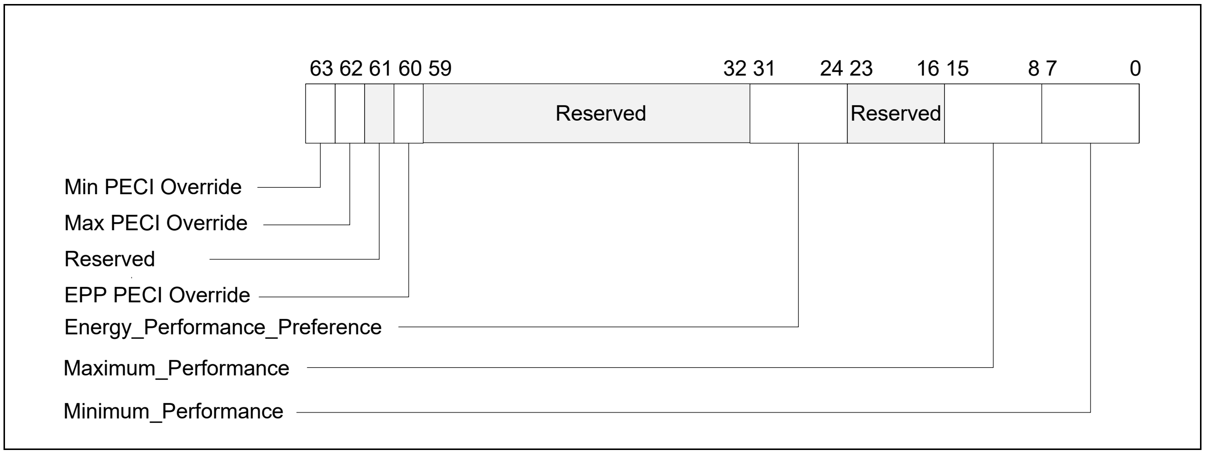


Figure 15-8. IA32_HWP_PECI_REQUEST_INFO MSR

The layout of the IA32_HWP_PECI_REQUEST_INFO MSR is shown in Figure 15-8. This MSR is writable by the embedded controller but is read-only by software executing on the CPU. This MSR has Package scope. The bit fields are described below:

- **Minimum_Performance (bits 7:0, RO)** — Used by the OS to read the latest value of PECI minimum performance input.
- **Maximum_Performance (bits 15:8, RO)** — Used by the OS to read the latest value of PECI maximum performance input.
- Bits 23:16 are reserved and must be zero.
- **Energy_Performance_Preference (bits 31:24, RO)** — Used by the OS to read the latest value of PECI energy performance preference input.
- Bits 59:32 are reserved and must be zero.
- **EPP_PECI_Override (bit 60, RO)** — Indicates whether PECI if currently overriding the Energy Performance Preference input. If set(1), PECI is overriding the Energy Performance Preference input. If clear(0), OS has control over Energy Performance Preference input.
- Bit 61 is reserved and must be zero.
- **Max_PECI_Override (bit 62, RO)** — Indicates whether PECI if currently overriding the Maximum Performance input. If set(1), PECI is overriding the Maximum Performance input. If clear(0), OS has control over Maximum Performance input.
- **Min_PECI_Override (bit 63, RO)** — Indicates whether PECI if currently overriding the Minimum Performance input. If set(1), PECI is overriding the Minimum Performance input. If clear(0), OS has control over Minimum Performance input.

HWP Request Field Hierarchical Resolution

HWP Request field resolution is fed by three MSRs: IA32_HWP_REQUEST, IA32_HWP_REQUEST_PKG, and IA32_HWP_PECI_REQUEST_INFO. The flow that the processor goes through to resolve which field value is chosen is shown below.

```
For each of the two HWP Request fields; Desired and Activity Window:
    If IA32_HWP_REQUEST.PACKAGE_CONTROL = 1 and IA32_HWP_REQUEST.<field> valid bit = 0
        Resolved Field Value = IA32_HWP_REQUEST_PKG.<field>
    Else
        Resolved Field Value = IA32_HWP_REQUEST.<field>
For each of the three HWP Request fields; Min, Max, and EPP:
    If IA32_HWP_PECI_REQUEST_INFO.<field> PECI Override bit = 1
        Resolved Field Value = IA32_HWP_PECI_REQUEST_INFO.<field>
    Else if IA32_HWP_REQUEST.PACKAGE_CONTROL = 1 and IA32_HWP_REQUEST.<field> valid bit = 0
        Resolved Field Value = IA32_HWP_REQUEST_PKG.<field>
    Else
        Resolved Field Value = IA32_HWP_REQUEST.<field>
```

15.4.4.4 IA32_HWP_CTL MSR (Address: 776H Logical Processor Scope)

IA32_HWP_CTL[0] controls the behavior of IA32_HWP_REQUEST Package Control [bit 42]. This control bit allows the IA32_HWP_REQUEST MSR to stay in INIT mode most of the time (Control Bit is equal to its RESET value of 0) thus avoiding actual saving/restoring of the MSR contents when the OS adds it to the register set saved and restored by XSAVES/XRSTORS.

- When IA32_HWP_CTL[0] = 0:
  - If IA32_HWP_REQUEST[42] = 0, the processor ignores all fields of the IA32_HWP_REQUEST_PKG MSR and selects all HWP values (Min, Max, EPP, Desired, Activity Window) from the IA32_HWP_REQUEST MSR.
  - If IA32_HWP_REQUEST[42] = 1, the processor selects the HWP values (Min, Max, EPP, Desired, Activity Window) either from the IA32_HWP_REQUEST MSR or from the IA32_HWP_REQUEST_PKG MSR according

to the values contained in the IA32_HWP_REQUEST MSR bit fields [bits 63:59]. See Section 15.4.4.1 for additional details.

- When IA32_HWP_CTL[0] = 1, the behavior is reversed:
  - If IA32_HWP_REQUEST[42] = 1, the processor ignores all fields of the IA32_HWP_REQUEST_PKG MSR and selects all HWP values (Min, Max, EPP, Desired, Activity Window) from the IA32_HWP_REQUEST MSR.
  - If IA32_HWP_REQUEST[42] = 0, the processor selects the HWP values (Min, Max, EPP, Desired, Activity Window) either from the IA32_HWP_REQUEST MSR or from the IA32_HWP_REQUEST_PKG MSR according to the values contained in the IA32_HWP_REQUEST MSR bit fields [bits 63:59]. See Section 15.4.4.1 for additional details.

Section 15-2 summarizes the IA32_HWP_CTL MSR bit 0 control behavior.

Table 15-2. IA32_HWP_CTL MSR Bit 0 Behavior

Field	Description		
Thread request PKG CTL meaning	Defines which HWP Request MSR is used, whether thread level or package level. When the package MSR is used, the thread MSR valid bits define which thread MSR fields override the package (default 0).		
	IA32_HWP_CTL[PKG_CTL_PLR]	**IA32_HWP_REQUEST[PKG_CTL]**	**HWP Request MSR Used**
	0	0	IA32_HWP_REQUEST MSR
	0	1	IA32_HWP_REQUEST_PKG MSR
	1	0	IA32_HWP_REQUEST_PKG MSR
	1	1	IA32_HWP_REQUEST MSR

This MSR is supported if CPUID[6].EAX[22] is set.

If the IA32_PM_ENABLE[HWP_ENABLE] (bit 0) is not set, access to this MSR will generate a #GP fault.

15.4.5 HWP Feedback

The processor provides several types of feedback to the OS during HWP operation.

The IA32_MPERF MSR and IA32_APERF MSR mechanism (see Section 15.2) allows the OS to calculate the resultant effective frequency delivered over a time period. Energy efficiency and performance optimizations directly impact the resultant effective frequency delivered.

The layout of the IA32_HWP_STATUS MSR is shown in Figure 15-9. It provides feedback regarding changes to IA32_HWP_CAPABILITIES.Guaranteed_Performance, IA32_HWP_CAPABILITIES.Highest_Performance, excursions to IA32_HWP_CAPABILITIES.Minimum_Performance, and PECI_Override entry/exit events. The bit fields are described below:

- **Guaranteed_Performance_Change (bit 0, RWC0)** — If set (1), a change to Guaranteed_Performance has occurred. Software should query IA32_HWP_CAPABILITIES.Guaranteed_Performance value to ascertain the new Guaranteed Performance value and to assess whether to re-adjust HWP hints via IA32_HWP_REQUEST. Software must clear this bit by writing a zero (0).
- Bit 1 is reserved and must be zero.
- **Excursion_To_Minimum (bit 2, RWC0)** — If set (1), an excursion to Minimum_Performance of IA32_HWP_REQUEST has occurred. Software must clear this bit by writing a zero (0).
- **Highest_Change (bit 3, RWC0)** — If set (1), a change to Highest Performance has occurred. Software should query IA32_HWP_CAPABILITIES to ascertain the new Highest Performance value. Software must clear this bit by writing a zero (0). Interrupts upon Highest Performance change are supported if CPUID[6].EAX[15] is set.
- **PECI_Override_Entry (bit 4, RWC0)** — If set (1), an embedded/management controller has started a PECI override of one or more OS control hints (Min, Max, EPP) specified in IA32_HWP_REQUEST or IA32_HWP_REQUEST_PKG. Software may query IA32_HWP_PECI_REQUEST_INFO MSR to ascertain which fields are now overridden via the PECI mechanism and what their values are (see Section 15.4.4.3 for additional details).

Software must clear this bit by writing a zero (0). Interrupts upon PECI override entry are supported if CPUID[6].EAX[16] is set.

- **PECI_Override_Exit (bit 5, RWC0)** — If set (1), an embedded/management controller has stopped overriding one or more OS control hints (Min, Max, EPP) specified in IA32_HWP_REQUEST or IA32_HWP_REQUEST_PKG. Software may query IA32_HWP_PECI_REQUEST_INFO MSR to ascertain which fields are still overridden via the PECI mechanism and which fields are now back under software control (see Section 15.4.4.3 for additional details). Software must clear this bit by writing a zero (0). Interrupts upon PECI override exit are supported if CPUID[6].EAX[16] is set.
- Bits 63:6 are reserved and must be zero.

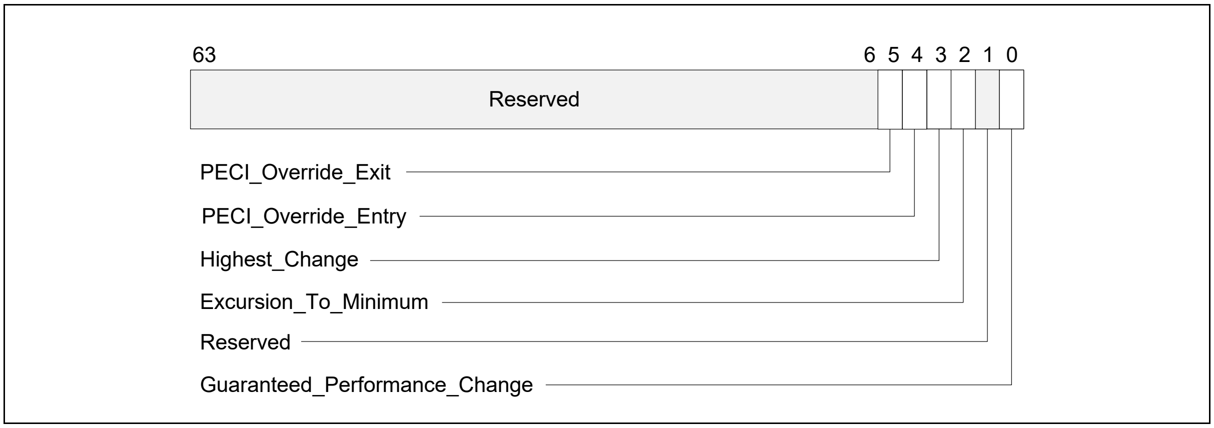


Figure 15-9. IA32_HWP_STATUS MSR

The status bits of IA32_HWP_STATUS must be cleared (0) by software so that a new status condition change will cause the hardware to set the bit again and issue the notification. Status bits are not set for “normal” excursions, e.g., running below Minimum Performance for short durations during C-state exit. Changes to Guaranteed_Performance, Highest_Performance, excursions to Minimum_Performance, or PECI_Override entry/exit will occur no more than once per second.

The OS can determine the specific reasons for a Guaranteed_Performance change or an excursion to Minimum_Performance in IA32_HWP_REQUEST by examining the associated status and log bits reported in the IA32_THERM_STATUS MSR. The layout of the IA32_HWP_STATUS MSR that HWP uses to support software query of HWP feedback is shown in Figure 15-10. The bit fields of IA32_THERM_STATUS associated with HWP feedback are described below (Bit fields of IA32_THERM_STATUS unrelated to HWP can be found in Section 15.8.5.2).

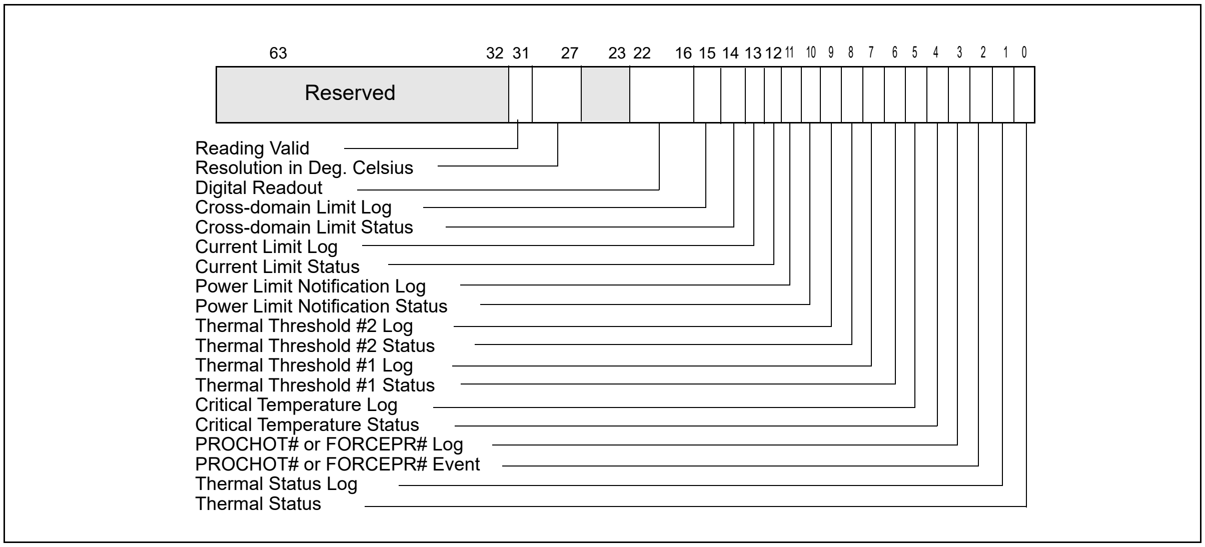


Figure 15-10. IA32_THERM_STATUS Register With HWP Feedback

- Bits 11:0, See Section 15.8.5.2.
- **Current Limit Status (bit 12, RO)** — If set (1), indicates an electrical current limit (e.g., Electrical Design Point/IccMax) is being exceeded and is adversely impacting energy efficiency optimizations.
- **Current Limit Log (bit 13, RWC0)** — If set (1), an electrical current limit has been exceeded that has adversely impacted energy efficiency optimizations since the last clearing of this bit or a reset. This bit is sticky, software may clear this bit by writing a zero (0).
- **Cross-domain Limit Status (bit 14, RO)** — If set (1), indicates another hardware domain (e.g., processor graphics) is currently limiting energy efficiency optimizations in the processor core domain.
- **Cross-domain Limit Log (bit 15, RWC0)** — If set (1), indicates another hardware domain (e.g., processor graphics) has limited energy efficiency optimizations in the processor core domain since the last clearing of this bit or a reset. This bit is sticky, software may clear this bit by writing a zero (0).
- Bits 63:16, See Section 15.8.5.2.

15.4.5.1 Non-Architectural HWP Feedback

The Productive Performance (MSR_PPERF) MSR (non-architectural) provides hardware's view of workload scalability, which is a rough assessment of the relationship between frequency and workload performance, to software. The layout of the MSR_PPERF is shown in Figure 15-11.

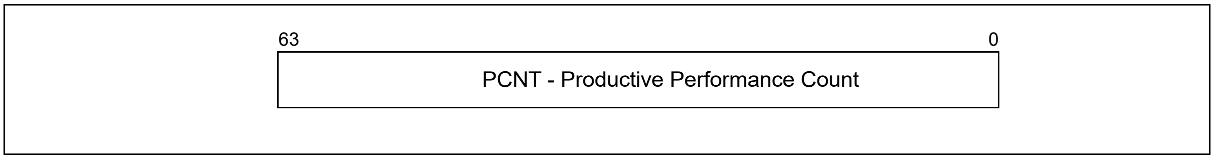


Figure 15-11. MSR_PPERF MSR

- **PCNT (bits 63:0, RO)** — Similar to IA32_APERF but only counts cycles perceived by hardware as contributing to instruction execution (e.g., unhalted and unstalled cycles). This counter increments at the same rate as IA32_APERF, where the ratio of (ΔPCNT/ΔACNT) is an indicator of workload scalability (0% to 100%). Note that values in this register are valid even when HWP is not enabled.

15.4.6 HWP Notifications

Processors may support interrupt-based notification of changes to HWP status as indicated by CPUID. If supported, the IA32_HWP_INTERRUPT MSR is used to enable interrupt-based notifications. Notification events, when enabled, are delivered using the existing thermal LVT entry. The layout of the IA32_HWP_INTERRUPT is shown in Figure 15-12. The bit fields are described below:

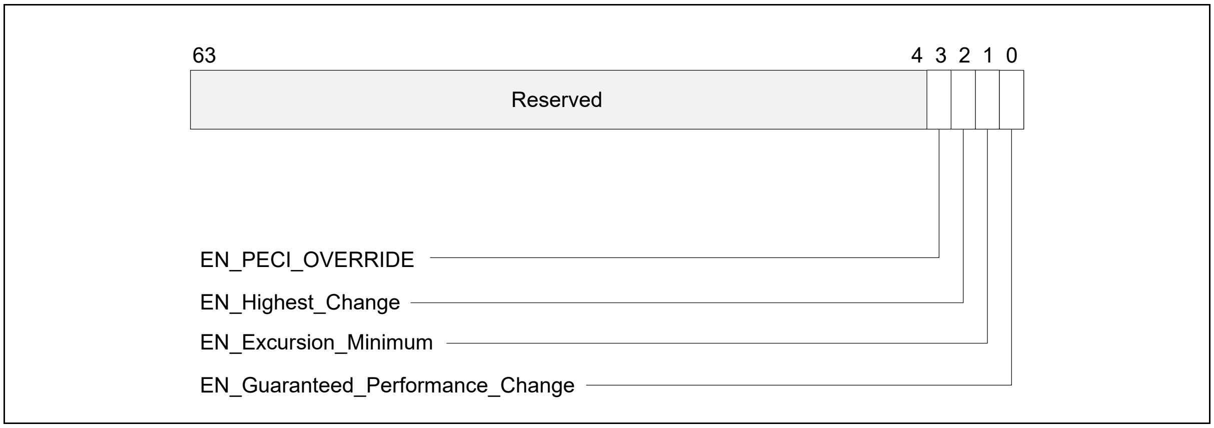


Figure 15-12. IA32_HWP_INTERRUPT MSR

- **EN_Guaranteed_Performance_Change (bit 0, RW)** — When set (1), an HWP Interrupt will be generated whenever a change to the IA32_HWP_CAPABILITIES.Guaranteed_Performance occurs. The default value is 0 (Interrupt generation is disabled).
- **EN_Excursion_Minimum (bit 1, RW)** — When set (1), an HWP Interrupt will be generated whenever the HWP hardware is unable to meet the IA32_HWP_REQUEST.Minimum_Performance setting. The default value is 0 (Interrupt generation is disabled).
- **EN_Highest_Change (bit 2, RW)** — When set (1), an HWP Interrupt will be generated whenever a change to the IA32_HWP_CAPABILITIES.Highest_Performance occurs. The default value is 0 (interrupt generation is disabled). Interrupts upon Highest Performance change are supported if CPUID[6].EAX[15] is set.
- **EN_PECI_OVERRIDE (bit 3, RW)** — When set (1), an HWP Interrupt will be generated whenever PECI starts or stops overriding any of the three HWP fields described in Section 15.4.4.3. The default value is 0 (interrupt generation is disabled). See Section 15.4.5 and Section 15.4.4.3 for details on how the OS learns what is the current set of HWP fields that are overridden by PECI. Interrupts upon PECI override change are supported if CPUID[6].EAX[16] is set.
- Bits 63:4 are reserved and must be zero.

15.4.7 Idle Logical Processor Impact on Core Frequency

Intel processors use one of two schemes for setting core frequency:

1. All cores share same frequency.
2. Each physical core is set to a frequency of its own.

In both cases the two logical processors that share a single physical core are set to the same frequency, so the processor accounts for the IA32_HWP_REQUEST MSR fields of both logical processors when defining the core frequency or the whole package frequency.

When **CPUID[6].EAX[20]** is set and only one logical processor of the two is active, while the other is idle (in any **C1 sub-state** or in a deeper sleep state), only the **active logical processor's** IA32_HWP_REQUEST MSR fields are considered, i.e., the HWP Request fields of a logical processor in the C1E sub-state or in a deeper sleep state are ignored.

Note: when a logical processor is in **C1 state** its HWP Request fields are accounted for.

15.4.8 Fast Write of Uncore MSR (Model Specific Feature)

There are a few logical processor scope MSRs whose values need to be observed outside the logical processor. The WRMSR instruction takes over 1000 cycles to complete (retire) for those MSRs. This overhead forces operating systems to avoid writing them too often whereas in many cases it is preferable that the OS writes them quite frequently for optimal power/performance operation of the processor.

The model specific "Fast Write MSR" feature reduces this overhead by an order of magnitude to a level of 100 cycles for a selected subset of MSRs.

Note: Writes to Fast Write MSRs are posted, i.e., when the WRMSR instruction completes, the data may still be "in transit" within the processor. Software can check the status by querying the processor to ensure data is already visible outside the logical processor (see Section 15.4.8.3 for additional details). Once the data is visible outside the logical processor, software is ensured that later writes by the same logical processor to the same MSR will be visible later (will not bypass the earlier writes).

MSRs that are selected for Fast Write are specified in a special capability MSR (see Section 15.4.8.1). Architectural MSRs that existed prior to the introduction of this feature and are selected for Fast Write, thus turning from slow to fast write MSRs, will be noted as such via a new CPUID bit. New MSRs that are fast upon introduction will be documented as such without an additional CPUID bit.

Three model specific MSRs are associated with the feature itself. They enable enumerating, controlling, and monitoring it. All three are logical processor scope.

15.4.8.1 FAST_UNCORE_MSRS_CAPABILITY (Address: 0x65F, Logical Processor Scope)

Operating systems or BIOS can read the FAST_UNCORE_MSRS_CAPABILITY MSR to enumerate those MSRs that are Fast Write MSRs.

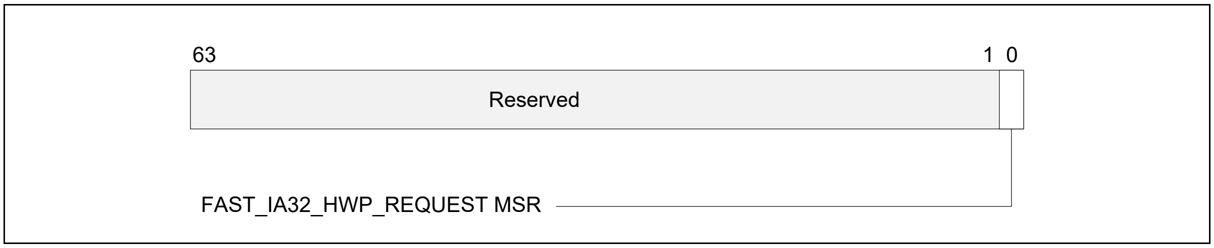


Figure 15-13. FAST_UNCORE_MSRS_CAPABILITY MSR

- **FAST_IA32_HWP_REQUEST MSR (bit 0, RO)** — When set (1), indicates that the IA32_HWP_REQUEST MSR is supported as a Fast Write MSR. A value of 0 indicates the IA32_HWP_REQUEST MSR is not supported as a Fast Write MSR.
- Bits 63:1 are reserved and must be zero.

15.4.8.2 FAST_UNCORE_MSRS_CTL (Address: 0x657, Logical Processor Scope)

Operating Systems or BIOS can use the FAST_UNCORE_MSRS_CTL MSR to opt-in or opt-out for fast write of specific MSRs that are enabled for Fast Write by the processor.

Note: Not all MSRs that are selected for this feature will necessarily have this opt-in/opt-out option. They may be supported in fast write mode only.

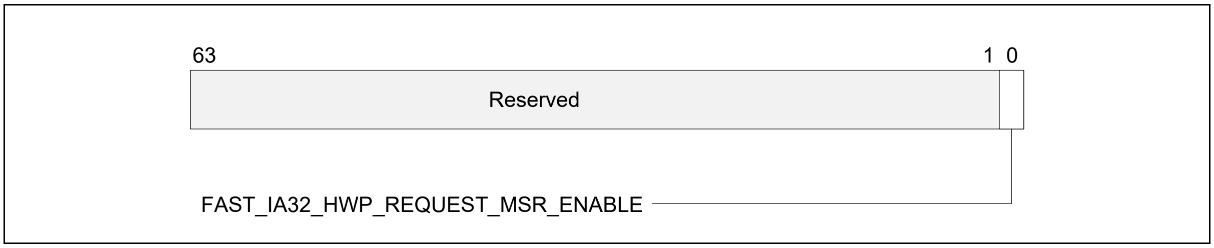


Figure 15-14. FAST_UNCORE_MSRS_CTL MSR

- **FAST_IA32_HWP_REQUEST_MSR_ENABLE (bit 0, RW)** — When set (1), enables fast access mode for the IA32_HWP_REQUEST MSR and sets the low latency, posted IA32_HWP_REQUESRT MSR' CPUID[6].EAX[18]. The default value is 0. Note that this bit can only be enabled once from the default value. Once set, writes to this bit are ignored. Only RESET will clear this bit.
- Bits 63:1 are reserved and must be zero.

15.4.8.3 FAST_UNCORE_MSRS_STATUS (Address: 0x65E, Logical Processor Scope)

Software that executes the WRMSR instruction of a Fast Write MSR can check whether the data is already visible outside the logical processor by reading the FAST_UNCORE_MSRS_STATUS MSR. For each Fast Write MSR there is a status bit that indicates whether the data is already visible outside the logical processor or is still in "transit".

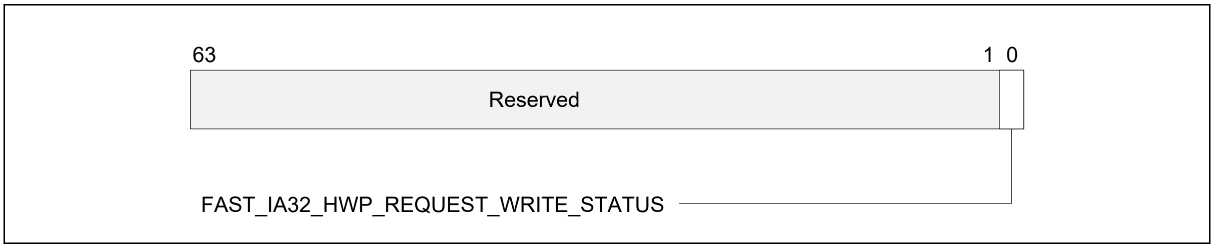


Figure 15-15. FAST_UNCORE_MSRS_STATUS MSR

- **FAST_IA32_HWP_REQUEST_WRITE_STATUS (bit 0, RO)** — Indicates whether the CPU is still in the middle of writing IA32_HWP_REQUEST MSR, even after the WRMSR instruction has retired. A value of 1 indicates the last write of IA32_HWP_REQUEST is still ongoing. A value of 0 indicates the last write of IA32_HWP_REQUEST is visible outside the logical processor.
- Bits 63:1 are reserved and must be zero.

15.4.9 Fast_IA32_HWP_REQUEST CPUID

IA32_HWP_REQUEST is an architectural MSR that exists in processors whose CPUID[6].EAX[7] is set (HWP BASE is enabled). This MSR has logical processor scope, but after its contents are written the contents become visible outside the logical processor. When the FAST_IA32_HWP_REQUEST CPUID[6].EAX[18] bit is set, writes to the IA32_HWP_REQUEST MSR are visible outside the logical processor via the "Fast Write" feature described in Section 15.4.8.

15.4.10 Recommendations for OS use of HWP Controls

Common Cases of Using HWP

The default HWP control field values are expected to be suitable for many applications. The OS can enable autonomous HWP for these common cases by

- Setting IA32_HWP_REQUEST.Desired Performance = 0 (hardware autonomous selection determines the performance target). Set IA32_HWP_REQUEST.Activity Window = 0 (enable HW dynamic selection of window size).

To maximize HWP benefit for the common cases, the OS should set

- IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_CAPABILITIES.Lowest_Performance and
- IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Highest_Performance.

Setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance is functionally equivalent to using of the IA32_PERF_CTL interface and is therefore not recommended (bypassing HWP).

Calibrating HWP for Application-Specific HWP Optimization

In some applications, the OS may have Quality of Service requirements that may not be met by the default values. The OS can characterize HWP by:

- keeping IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance to prevent non-linearity in the characterization process,
- utilizing the range values enumerated from the IA32_HWP_CAPABILITIES MSR to program IA32_HWP_REQUEST while executing workloads of interest and observing the power and performance result.

The power and performance result of characterization is also influenced by the IA32_HWP_REQUEST.Energy Performance Preference field, which must also be characterized.

Characterization can be used to set IA32_HWP_REQUEST.Minimum_Performance to achieve the required QOS in terms of performance. If IA32_HWP_REQUEST.Minimum_Performance is set higher than IA32_HWP_CAPABILITIES.Guaranteed Performance then notification of excursions to Minimum Performance may be continuous.

If autonomous selection does not deliver the required workload performance, the OS should assess the current delivered effective frequency and for the duration of the specific performance requirement set IA32_HWP_REQUEST.Desired_Performance ≠ 0 and adjust IA32_HWP_REQUEST.Energy_Performance_Preference as necessary to achieve the required workload performance. The MSR_PPERF.PCNT value can be used to better comprehend the potential performance result from adjustments to IA32_HWP_REQUEST.Desired_Performance. The OS should set IA32_HWP_REQUEST.Desired_Performance = 0 to re-enable autonomous selection.

Tuning for Maximum Performance or Lowest Power Consumption

Maximum performance will be delivered by setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Highest_Performance and setting IA32_HWP_REQUEST.Energy_Performance_Preference = 0 (performance preference).

Lowest power will be achieved by setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Lowest_Performance and setting IA32_HWP_REQUEST.Energy_Performance_Preference = 0FFH (energy efficiency preference).

Mixing Logical Processor and Package Level HWP Field Settings

Using the IA32_HWP_REQUEST Package_Control bit and the five valid bits in that MSR, the OS can mix and match between selecting the Logical Processor scope fields and the Package level fields. For example, the OS can set all logical cores' IA32_HWP_REQUEST.Package_Control bit to ‘1’, and for those logical processors if it prefers a different EPP value than the one set in the IA32_HWP_REQUEST_PKG MSR, the OS can set the desired EPP value and the EPP valid bit. This overrides the package EPP value for only a subset of the logical processors in the package.

Additional Guidelines

Set IA32_HWP_REQUEST.Energy_Performance_Preference as appropriate for the platform's current mode of operation. For example, a mobile platforms' setting may be towards performance preference when on AC power and more towards energy efficiency when on DC power.

The use of the Running Average Power Limit (RAPL) processor capability (see section 14.7.1) is highly recommended when HWP is enabled. Use of IA32_HWP_Request.Maximum_Performance for thermal control is subject to limitations and can adversely impact the performance of other processor components, e.g., graphics

If default values deliver undesirable performance latency in response to events, the OS should set IA32_HWP_REQUEST. Activity_Window to a low (non-zero) value and IA32_HWP_REQUEST.Energy_Performance_Preference towards performance (0) for the event duration.

Similarly, for "real-time" threads, set IA32_HWP_REQUEST.Energy_Performance_Preference towards performance (0) and IA32_HWP_REQUEST. Activity_Window to a low value, e.g., 01H, for the duration of their execution.

When executing low priority work that may otherwise cause the hardware to deliver high performance, set IA32_HWP_REQUEST. Activity_Window to a longer value and reduce the IA32_HWP_Request.Maximum_Performance value as appropriate to control energy efficiency. Adjustments to IA32_HWP_REQUEST.Energy_Performance_Preference may also be necessary.

15.5 HARDWARE DUTY CYCLING (HDC)

Intel processors may contain support for Hardware Duty Cycling (HDC), which enables the processor to autonomously force its components inside the physical package into idle state. For example, the processor may selectively force only the processor cores into an idle state.

HDC is disabled by default on processors that support it. System software can dynamically enable or disable HDC to force one or more components into an idle state or wake up those components previously forced into an idle state. Forced Idling (and waking up) of multiple components in a physical package can be done with one WRMSR to a packaged-scope MSR from any logical processor within the same package.

HDC does not delay events such as timer expiration, but it may affect the latency of short (less than 1 msec) software threads, e.g., if a thread is forced to idle state just before completion and entering a "natural idle".

HDC forced idle operation can be thought of as operating at a lower effective frequency. The effective average frequency computed by software will include the impact of HDC forced idle.

The primary use of HDC is enable system software to manage low active workloads to increase the package level C6 residency. Additionally, HDC can lower the effective average frequency in case or power or thermal limitation.

When HDC forces a logical processor, a processor core or a physical package to enter an idle state, its C-State is set to C3 or deeper. The deep "C-states" referred to in this section are processor-specific C-states.

15.5.1 Hardware Duty Cycling Programming Interfaces

The programming interfaces provided by HDC include the following:

- The CPUID instruction allows software to discover the presence of HDC support in an Intel processor. Specifically, execute CPUID instruction with EAX=06H as input, bit 13 of EAX indicates the processor's support of the following aspects of HDC.
  - Availability of HDC baseline resource, CPUID.06H:EAX[bit 13]: If this bit is set, HDC provides the following architectural MSRs: IA32_PKG_HDC_CTL, IA32_PM_CTL1, and the IA32_THREAD_STALL MSRs.
- Additionally, HDC may provide several non-architectural MSR.

Table 15-3. Architectural and non-Architecture MSRs Related to HDC

Address	Architectural	Register Name	Description
DB0H	Y	IA32_PKG_HDC_CTL	Package Enable/Disable HDC.
DB1H	Y	IA32_PM_CTL1	Per-logical-processor select control to allow/block HDC forced idling.
DB2H	Y	IA32_THREAD_STALL	Accumulate stalled cycles on this logical processor due to HDC forced idling.
653H	N	MSR_CORE_HDC_RESIDENCY	Core level stalled cycle counter due to HDC forced idling on one or more logical processor.
655H	N	MSR_PKG_HDC_SHALLOW_RESIDENCY	Accumulate the cycles the package was in C2[1] state and at least one logical processor was in forced idle
656H	N	MSR_PKG_HDC_DEEP_RESIDENCY	Accumulate the cycles the package was in the software specified Cx[1] state and at least one logical processor was in forced idle. Cx is specified in MSR_PKG_HDC_CONFIG_CTL.
652H	N	MSR_PKG_HDC_CONFIG_CTL	HDC configuration controls

NOTES:

[1] The package "C-states" referred to in this section are processor-specific C-states.

15.5.2 Package level Enabling HDC

The layout of the IA32_PKG_HDC_CTL MSR is shown in Figure 15-16. IA32_PKG_HDC_CTL is a writable MSR from any logical processor in a package. The bit fields are described below:

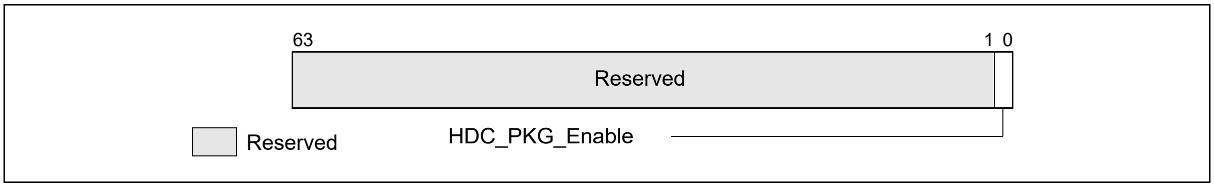


Figure 15-16. IA32_PKG_HDC_CTL MSR

- **HDC_PKG_Enable (bit 0, R/W)** — Software sets this bit to enable HDC operation by allowing the processor to force to idle all "HDC-allowed" (see Figure 15.5.3) logical processors in the package. Clearing this bit disables HDC operation in the package by waking up all the processor cores that were forced into idle by a previous '0'-to-'1' transition in IA32_PKG_HDC_CTL.HDC_PKG_Enable. This bit is writable only if CPUID.06H:EAX[bit 13] = 1. Default = zero (0).
- Bits 63:1 are reserved and must be zero.

After processor support is determined via CPUID, system software can enable HDC operation by setting IA32_PKG_HDC_CTL.HDC_PKG_Enable to 1. At reset, IA32_PKG_HDC_CTL.HDC_PKG_Enable is cleared to 0. A '0'-to-'1' transition in HDC_PKG_Enable allows the processor to force to idle all HDC-allowed (indicated by the non-zero state of IA32_PM_CTL1[bit 0]) logical processors in the package. A '1'-to-'0' transition wakes up those HDC force-idled logical processors.

Software can enable or disable HDC using this package level control multiple times from any logical processor in the package. Note the latency of writing a value to the package-visible IA32_PKG_HDC_CTL.HDC_PKG_Enable is longer than the latency of a WRMSR operation to a Logical Processor MSR (as opposed to package level MSR) such as: IA32_PM_CTL1 (described in Section 15.5.3). Propagation of the change in IA32_PKG_HDC_CTL.HDC_PKG_Enable and reaching all HDC idled logical processor to be woken up may take on the order of core C6 exit latency.

15.5.3 Logical-Processor Level HDC Control

The layout of the IA32_PM_CTL1 MSR is shown in Figure 15-17. Each logical processor in a package has its own IA32_PM_CTL1 MSR. The bit fields are described below:

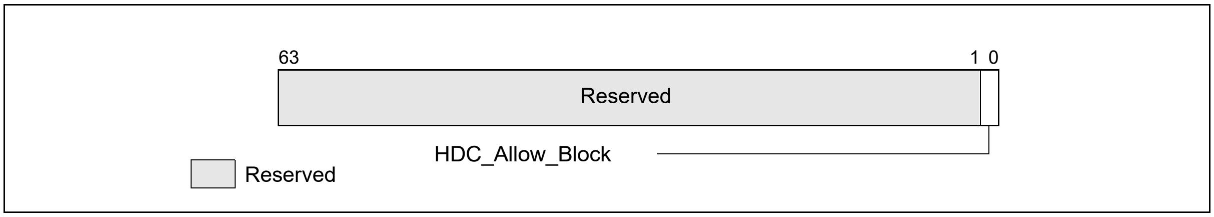


Figure 15-17. IA32_PM_CTL1 MSR

- **HDC_Allow_Block (bit 0, R/W)** — Software sets this bit to allow this logical processors to honor the package-level IA32_PKG_HDC_CTL.HDC_PKG_Enable control. Clearing this bit prevents this logical processor from using the HDC. This bit is writable only if CPUID.06H:EAX[bit 13] = 1. Default = one (1).
- Bits 63:1 are reserved and must be zero.

Fine-grain OS control of HDC operation at the granularity of per-logical-processor is provided by IA32_PM_CTL1. At RESET, all logical processors are allowed to participate in HDC operation such that OS can manage HDC using the package-level IA32_PKG_HDC_CTL.

Writes to IA32_PM_CTL1 complete with the latency that is typical to WRMSR to a Logical Processor level MSR. When the OS chooses to manage HDC operation at per-logical-processor granularity, it can write to IA32_PM_CTL1 on one or more logical processors as desired. Each write to IA32_PM_CTL1 must be done by code that executes on the logical processor targeted to be allowed into or blocked from HDC operation.

Blocking one logical processor for HDC operation may have package level impact. For example, the processor may decide to stop duty cycling of all other Logical Processors as well.

The propagation of IA32_PKG_HDC_CTL.HDC_PKG_Enable in a package takes longer than a WRMSR to IA32_PM_CTL1. The last completed write to IA32_PM_CTL1 on a logical processor will be honored when a ‘0’-to-’1’ transition of IA32_PKG_HDC_CTL.HDC_PKG_Enable arrives to a logical processor.

15.5.4 HDC Residency Counters

There is a collection of counters available for software to track various residency metrics related to HDC operation. In general, HDC residency time is defined as the time in HDC forced idle state at the granularity of per-logical-processor, per-core, or package. At the granularity of per-core/package-level HDC residency, at least one of the logical processor in a core/package must be in the HDC forced idle state.

15.5.4.1 IA32_THREAD_STALL

Software can track per-logical-processor HDC residency using the architectural MSR IA32_THREAD_STALL.The layout of the IA32_THREAD_STALL MSR is shown in Figure 15-18. Each logical processor in a package has its own IA32_THREAD_STALL MSR. The bit fields are described below:

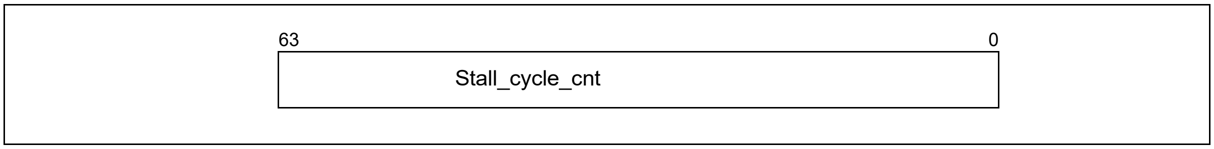


Figure 15-18. IA32_THREAD_STALL MSR

- **Stall_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after the logical processor exits from the forced idled C-state. At each update, the number of cycles that the logical processor was stalled due to forced-idle will be added to the counter. This counter is available only if CPUID.06H:EAX[bit 13] = 1. Default = zero (0).

A value of zero in IA32_THREAD_STALL indicates either HDC is not supported or the logical processor never serviced any forced HDC idle. A non-zero value in IA32_THREAD_STALL indicates the HDC forced-idle residency times of the logical processor. It also indicates the forced-idle cycles due to HDC that could appear as C0 time to traditional OS accounting mechanisms (e.g., time-stamping OS idle/exit events).

Software can read IA32_THREAD_STALL irrespective of the state of IA32_PKG_HDC_CTL and IA32_PM_CTL1, as long as CPUID.06H:EAX[bit 13] = 1.

15.5.4.2 Non-Architectural HDC Residency Counters

Processors that support HDC operation may provide the following model-specific HDC residency counters.

MSR_CORE_HDC_RESIDENCY

Software can track per-core HDC residency using the counter MSR_CORE_HDC_RESIDENCY. This counter increments when the core is in C3 state or deeper (all logical processors in this core are idle due to either HDC or other mechanisms) and at least one of the logical processors is in HDC forced idle state. The layout of the MSR_CORE_HDC_RESIDENCY is shown in Figure 15-19. Each processor core in a package has its own MSR_CORE_HDC_RESIDENCY MSR. The bit fields are described below:

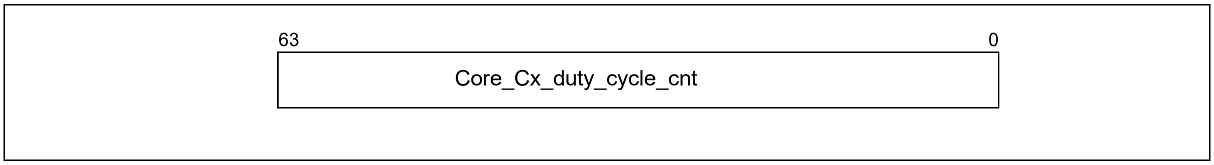


Figure 15-19. MSR_CORE_HDC_RESIDENCY MSR

- **Core_Cx_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after core C-state exit from a forced idled C-state. At each update, the increment counts cycles when the core is in a Cx state (all its logical processor are idle) and at least one logical processor in this core was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR will cause a #GP fault. Default = zero (0).

A value of zero in MSR_CORE_HDC_RESIDENCY indicates either HDC is not supported or this processor core never serviced any forced HDC idle.

MSR_PKG_HDC_SHALLOW_RESIDENCY

The counter MSR_PKG_HDC_SHALLOW_RESIDENCY allows software to track HDC residency time when the package is in C2 state, all processor cores in the package are not active and at least one logical processor was forced into idle state due to HDC. The layout of the MSR_PKG_HDC_SHALLOW_RESIDENCY is shown in Figure 15-20. There is one MSR_PKG_HDC_SHALLOW_RESIDENCY per package. The bit fields are described below:

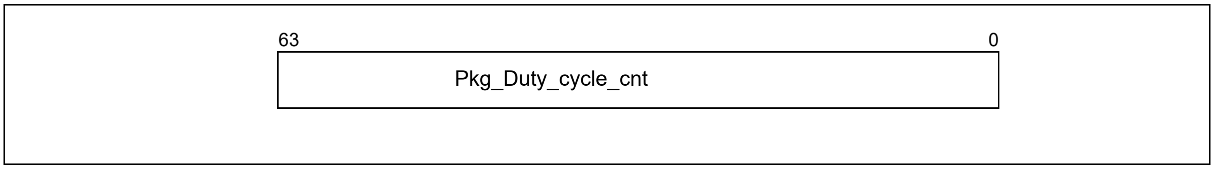


Figure 15-20. MSR_PKG_HDC_SHALLOW_RESIDENCY MSR

- **Pkg_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. Package shallow residency may be implementation specific. In the initial implementation, the threshold is package C2-state. The count is updated only after package C2-state exit from a forced idled C-state. At each update, the increment counts cycles when the package is in C2 state and at least one processor core in this package was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).

A value of zero in MSR_PKG_HDC_SHALLOW_RESIDENCY indicates either HDC is not supported or this processor package never serviced any forced HDC idle.

MSR_PKG_HDC_DEEP_RESIDENCY

The counter MSR_PKG_HDC_DEEP_RESIDENCY allows software to track HDC residency time when the package is in a software-specified package Cx state, all processor cores in the package are not active and at least one logical processor was forced into idle state due to HDC. Selection of a specific package Cx state can be configured using MSR_PKG_HDC_CONFIG. The layout of the MSR_PKG_HDC_DEEP_RESIDENCY is shown in Figure 15-21. There is one MSR_PKG_HDC_DEEP_RESIDENCY per package. The bit fields are described below:

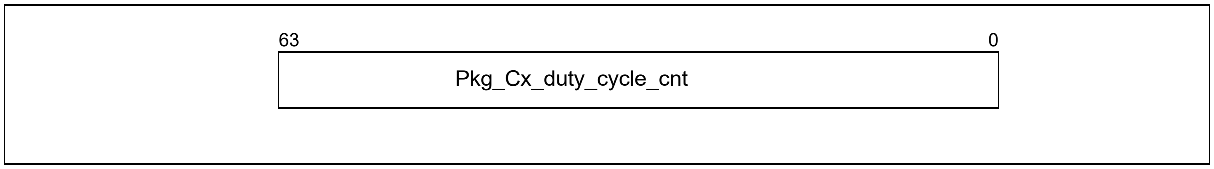


Figure 15-21. MSR_PKG_HDC_DEEP_RESIDENCY MSR

- **Pkg_Cx_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after package C-state exit from a forced idle state. At each update, the increment counts cycles when the package is in the software-configured Cx state and at least one processor core in this package was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).

A value of zero in MSR_PKG_HDC_SHALLOW_RESIDENCY indicates either HDC is not supported or this processor package never serviced any forced HDC idle.

MSR_PKG_HDC_CONFIG

MSR_PKG_HDC_CONFIG allows software to configure the package Cx state that the counter MSR_PKG_HDC_DEEP_RESIDENCY monitors. The layout of the MSR_PKG_HDC_CONFIG is shown in Figure 15-22. There is one MSR_PKG_HDC_CONFIG per package. The bit fields are described below:

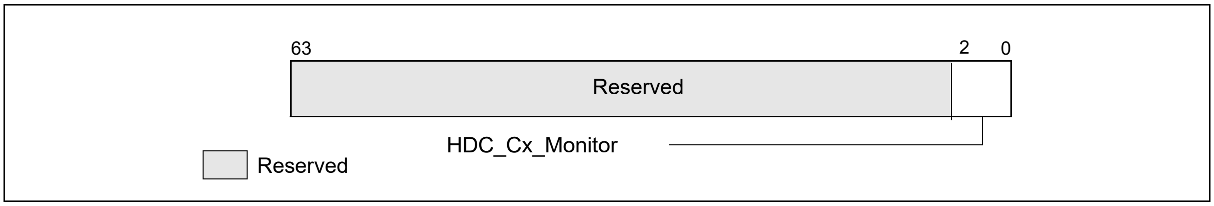


Figure 15-22. MSR_PKG_HDC_CONFIG MSR

- **Pkg_Cx_Monitor (bits 2:0, R/W)** — Selects which package C-state the MSR_HDC_DEEP_RESIDENCY counter will monitor. The encoding of the HDC_Cx_Monitor field are: **0**: no-counting; **1**: count package C2 only, **2**: count package C3 and deeper; **3**: count package C6 and deeper; **4**: count package C7 and deeper; other encodings are reserved. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).
- Bits 63:3 are reserved and must be zero.

15.5.5 MPERF and APERF Counters Under HDC

HDC operation can be thought of as an average effective frequency drop due to all or some of the Logical Processors enter an idle state period.

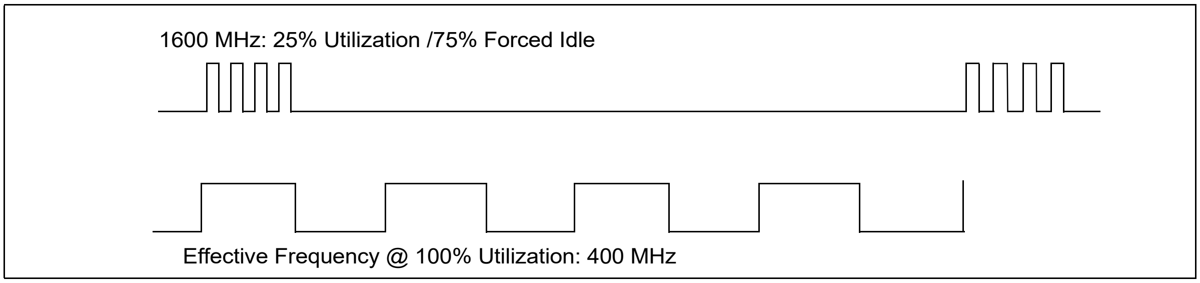

1600 MHz: 25% Utilization /75% Forced Idle
Effective Frequency @ 100% Utilization: 400 MHz

Figure 15-23. Example of Effective Frequency Reduction and Forced Idle Period of HDC

By default, the IA32_MPERF counter counts during forced idle periods as if the logical processor was active. The IA32_APERF counter does not count during forced idle state. This counting convention allows the OS to compute the average effective frequency of the Logical Processor between the last MWAIT exit and the next MWAIT entry (OS visible C0) by ∆ACNT/∆MCNT * TSC Frequency.

15.6 HARDWARE FEEDBACK INTERFACE AND INTEL® THREAD DIRECTOR

Intel processors that enumerate CPUID.06H.0H:EAX.HW_FEEDBACK[bit 19] as 1 support Hardware Feedback Interface (HFI). Hardware provides guidance to the Operating System (OS) scheduler to perform optimal workload scheduling through a hardware feedback interface structure in memory. Details on this table structure are described in Section 15.6.1.

Intel processors that enumerate CPUID.06H.0H:EAX[bit 23] as 1 support Intel® Thread Director. Hardware provides guidance to the Operating System (OS) scheduler to perform optimal workload scheduling through a memory resident table and software thread specific index (Class ID) that points into that table and selects which data to use for that software thread. Details on this table structure are described in Section 15.6.2.

15.6.1 Hardware Feedback Interface Table Structure

This structure has a global header that is 16 bytes in size. Following this global header, there is one 8 byte entry per logical processor in the socket. The structure is designed as follows.

Table 15-4. Hardware Feedback Interface Structure

Byte Offset	Size (Bytes)	Description
0	16	Global Header
16	8	Per Logical Processor Entry
24	8	Per Logical Processor Entry
...	...	...
16 + n*8	8	Per Logical Processor Entry

The global header is structured as shown in Table 15-5.

Table 15-5. Hardware Feedback Interface Global Header Structure

Byte Offset	Size (Bytes)	Field Name	Description
0	8	Timestamp	Timestamp of when the table was last updated by hardware. This is a timestamp in crystal clock units. Initialized by the OS to 0.
8	1	Performance Capability Flags	If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.
9	1	Energy Efficiency Capability Changed	If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.
10	6	Reserved	Initialized by the OS to 0.

The per logical processor scheduler feedback entry is structured as follows. The operating system can determine the index of the logical processor feedback entry for a logical processor using CPUID.06H.0H:EDX[31:16] by executing CPUID on that logical processor.

Table 15-6. Hardware Feedback Interface Logical Processor Entry Structure

Byte Offset	Size (Bytes)	Field Name	Description
0	1	Performance Capability	Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. The OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback. CPUID.06H.0H:EDX[0] enumerates support for Performance capability reporting.
1	1	Energy Efficiency Capability	Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. An Energy Efficiency capability of 255 indicates which logical processors have the highest relative energy efficiency capability. In addition, the value 255 is an explicit recommendation for the OS to consolidate work on those logical processors for energy efficiency reasons. The OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback. CPUID.06H.0H:EDX[1] enumerates support for Energy Efficiency capability reporting.
2	6	Reserved	The OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback.

15.6.2 Intel® Thread Director Table Structure

This structure has a global header that is at least 16 bytes in size. Its size depends on the number of classes and capabilities enumerated by the CPUID instruction (see notes below Table 15-7). Following this global header there are multiple Logical Processor related entries. The structure is designed as follows.

Table 15-7. Intel® Thread Director Table Structure

Byte Offset[1,2,3]	Size (Bytes)	Description
0	8 + CP[4]*CL[4] + R8[5]	Global Header
8 + CP*CL + R8	CL*CP + R8	Per Logical Processor Entry$_0$[6]
8 + 2*(CP*CL + R8)	CL*CP + R8	Per Logical Processor Entry$_1$
...	...	...
8 + (N[7] -1)*(CP*CL + R8)	CL*CP +R8	Logical Processor Entry$_{N-1}$

NOTES:

1. Byte offset of Capability$_{cp}$ of Class$_{cl}$ change indication: 8 + CP * cl + cp.
2. Byte offset of LP Entry$_i$: 8 + (i+1) * (CP * CL + R8).
3. Byte offset of capability$_{cp}$ of class$_{cl}$ of LP Entry$_i$: 8 + (i+1) * (CP * CL + R8) + CP * cl + cp.
4. Both upper case CL and CP denote total number of classes and capabilities defined for the processor. Lower case cl and cp denote one instance of a class or capability. cl and cp are counted starting at zero. See "CPUID—CPU Identification" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A for the number of classes (CL) and the number of supported capabilities (CP). CP (# of capabilities): number of enumerated bits in CPUID.06H.0H.EDX[7:0] and CL (# of classes): CPUID.06H.0H.ECX[15:8].
5. R8 is the number of bytes necessary to round up the Capability Change Indication array and the Logical Processor Entry to whole multiple of 8 bytes.
6. Table size: 8 + (N+1)* (CP * CL + R8).
7. N is the number of Logical Processor Entries in the table. It is not greater than the number of Logical Processors on the socket, but may be lower.
8. The Operating System can determine the index for the Logical Processor Entry within the Intel Thread Director table using CPUID.06H.0H:EDX[31:16] by executing the CPUID instruction on that Logical Processor.
9. The Operating System should allocate space to accommodate for one such structure per socket in the system.
10. The Intel Thread Director table structure extends the Hardware Feedback Interface table structure without breaking backward compatibility. The Hardware Feedback Interface can be viewed as having two capabilities and a single class.

The global header is structured as shown in Table 15-8.

Table 15-8. Intel® Thread Director Global Header Structure

Byte Offset	Size (Bytes)	Description	
0	8	Time-stamp of when the table was last updated by hardware. This is a time-stamp in crystal clock units. Initialized by the OS to 0.	
8	1	Class 0 Performance Capability Flags	If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.
8 + 1	1	Class 0 Energy Efficiency Capability Flags	If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.
...			
8 + CP - 1	1	Class 0 change indication for Capability #(CP-1) if exists	Unavailable for capabilities that are not enumerated.
8 + CP	1	Class 1 Performance Capability Flags	If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.
8 + CP + 1	1	Class 1 Energy Efficiency Capability Flags	If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.
...			
8 + 2*CP - 1	1	Class 1 change indication for Capability #(CP-1) if exists	Unavailable for capabilities that are not enumerated.
...			Change indication for Capabilities of additional Classes if exist.
8 + (CL-1)*CP	1	Class #(CL-1) Performance Capability Flags	If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.
8 + (CL-1)*CP + 1	1	Class #(CL-1) Energy Efficiency Capability Flags	If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.

Table 15-8. Intel® Thread Director Global Header Structure (Contd.)

Byte Offset	Size (Bytes)	Description	
...			
8 + CL*CP - 1	1	Class #(CL-1) change indication for Capability #(CP-1) if exists	Unavailable for capabilities that are not enumerated.
8 + CL*CP	R8	Padding	Padding to nearest multiple of 8 bytes. Initialized by the OS to 0 prior to enabling Intel Thread Director.

The logical processor capability entry in the Intel Thread Director table is structured as follows.

Table 15-9. Intel® Thread Director Logical Processor Entry Structure

Byte Offset	Size (Bytes)	Field Name	Description
0	1	Performance Capability	Class 0 Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a single logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. Initialized by the OS to 0.
1	1	Energy Efficiency Capability	Class 0 Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. An Energy Efficiency capability of 255 indicates which logical processors have the highest relative energy efficiency capability. In addition, the value 255 is an explicit recommendation for the OS to consolidate work on those logical processors for energy efficiency reasons. Initialized by the OS to 0.
...			
CP - 1	1	Capability #(CP-1)	Class 0 Capability #(CP-1) if exists. If the capability does not exist (is not enumerated in the CPUID), the entry is unavailable (no space is reserved for future use here).
CP	R8	Padding	Padding to nearest multiple of 8 bytes. Initialized by the OS to 0 prior to enabling Intel Thread Director.
CP + R8	1	Performance Capability	Class 1 Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a single logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. Initialized by the OS to 0.
CP + 1	1	Energy Efficiency Capability	Class 1 Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. An Energy Efficiency capability of 255 indicates which logical processors have the highest relative energy efficiency capability. In addition, the value 255 is an explicit recommendation for the OS to consolidate work on those logical processors for energy efficiency reasons. Initialized by the OS to 0.
...			
2*CP - 1	1	Capability #(CP-1)	Class 1 Capability #(CP-1) if exists. If the capability does not exist (is not enumerated in the CPUID), the entry is unavailable (no space is reserved for future use here).

Table 15-9. Intel® Thread Director Logical Processor Entry Structure (Contd.)

Byte Offset	Size (Bytes)	Field Name	Description
2*CP	R8	Padding	Padding to nearest multiple of 8 bytes. Initialized by the OS to 0 prior to enabling Intel Thread Director.
...			
(CL-1)*CP	1	Performance Capability	Class #(CL-1) Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a single logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. Initialized by the OS to 0.
(CL-1)*CP + 1	1	Energy Efficiency Capability	Class #(CL-1) Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. An Energy Efficiency capability of 255 indicates which logical processors have the highest relative energy efficiency capability. In addition, the value 255 is an explicit recommendation for the OS to consolidate work on those logical processors for energy efficiency reasons. Initialized by the OS to 0.
...			
CL*CP - 1	1	Capability #(CP-1)	Class #(CL-1) Capability #(CP-1) if exists. If the capability does not exist (is not enumerated in the CPUID), the entry is unavailable (no space is reserved for future use here).
CL*CP	R8	Padding	Padding to nearest multiple of 8 bytes. Initialized by the OS to 0 prior to enabling Intel Thread Director.

15.6.3 Intel® Thread Director Usage Model

When the OS Scheduler needs to decide which one of multiple free logical processors to assign to a software thread that is ready to execute, it can choose one of the following options:

1. The free logical processor with the highest performance value of that software thread class, if the system is scheduling for performance.
2. The free logical processor with the highest energy efficiency value of that software thread class, if the system is scheduling for energy efficiency.

When the OS Scheduler needs to decide which of two logical processors (i,j) to assign to which of two software threads whose Class IDs are k1 and k2, it can compute the two performance ratios: Perf Ratio_1 = Perf_{ik1} / Perf_{jk1} and Perf Ratio_2 = Perf_{ik2} / Perf_{jk2}, or two energy efficiency ratios: Energy Eff. Ratio_1 = Energy Eff_{ik1} / Energy Eff_{jk1} and Energy Eff. Ratio_2 = Energy Eff_{ik2} / Energy Eff_{jk2} between the two logical processors for each of the two classes, depending on whether the OS is scheduling for performance or for energy efficiency.

For example, assume that the system is scheduling for performance and that Perf Ratio_1 > Perf Ratio_2. The OS Scheduler will assign the software thread whose Class ID is k1 to logical processor i, and the one whose Class ID is k2 to logical processor j.

When the two software threads in question belong to the same Class ID, the OS Scheduler can schedule to higher performance logical processors within that class when scheduling for performance and to higher energy efficiency logical processors within that class when scheduling for energy efficiency.

The highest to lowest ordering may be different between classes across cores and between the performance column and the energy efficiency column of the same class across cores.

15.6.4 Hardware Feedback Interface Pointer

The physical address of the HFI/Intel Thread Director structure is programmed by the OS into a package scoped MSR named IA32_HW_FEEDBACK_PTR. The MSR is structured as follows:

- Bits 63:MAXPHYADDR[1] – Reserved.
- Bits MAXPHYADDR-1:12 – ADDR. This is the physical address of the page frame of the first page of this structure.
- Bits 11:1 – Reserved.
- Bit 0 – Valid. When set to 1, indicates a valid pointer is programmed into the ADDR field of the MSR.

The address of this MSR is 17D0H. This MSR is cleared on processor reset to its default value of 0. It retains its value upon INIT.

CPUID.06H.0H:EDX[11:8] enumerates the size of memory that must be allocated by the OS for this structure.

15.6.5 Hardware Feedback Interface Configuration

The operating system enables HFI/Intel Thread Director using a package scoped MSR named IA32_HW_FEEDBACK_CONFIG (address 17D1H). This MSR is cleared on processor reset to its default value of 0. It retains its value upon INIT.

The MSR is structured as follows:

- Bits 63:2 – Reserved.
- Bit 1 – Enable Intel Thread Director (or multi-class support). Both bits 0 and 1 must be set for Intel Thread Director to be enabled. The extra class columns in the Intel Thread Director table are updated by hardware immediately following setting those two bits, as well as during run time as necessary.
- Bit 0 – Enable. When set to 1, enables HFI.

Before enabling HFI, the OS must set a valid hardware feedback interface structure using IA32_HW_FEEDBACK_PTR.

When the OS sets bit 0 only, the hardware populates class 0 capabilities only in the HFI structure. When bit 1 is set after or together with bit 0, the Intel Thread Director multi-class structure is populated.

When either the HFI structure or the Intel Thread Director structure are ready to use by the OS, the hardware sets IA32_PACKAGE_THERM_STATUS[bit 26]. An interrupt is generated by the hardware if IA32_PACKAGE_THERM_INTERRUPT[bit 25] is set.

When the OS clears bit 1 but leaves bit 0 set, Intel Thread Director is disabled, but HFI is kept operational. IA32_PACKAGE_THERM_STATUS[bit 26] is NOT set in this case.

Clearing bit 0 disables both HFI and Intel Thread Director, independent of the bit 1 state. Setting bit 1 to '1' while keeping bit 0 at '0' is an invalid combination which is quietly ignored.

When the OS clears bit 0, hardware sets the IA32_PACKAGE_THERM_STATUS[bit 26] to 1 to acknowledge disabling of the interface. The OS should wait for this bit to be set to 1 to reclaim the memory of the Intel Thread Director structure, as by setting IA32_PACKAGE_THERM_STATUS[bit 26] hardware guarantees not to write into the Intel Thread Director structure anymore.

The OS may clear bit 0 only after receiving an indication from the hardware that the structure initialization is complete via the same IA32_PACKAGE_THERM_STATUS[bit 26], following enabling of HFI/Intel Thread Director, thus avoiding a race condition between OS and hardware.

Bit 1 is valid only if CPUID[6].EAX[bit 23] is set. When setting this bit while support is not enumerated, the hardware generates #GP.

Table 15-10 summarizes the control options described above.

See Section 15.6.9 for details on scenarios where IA32_HW_FEEDBACK_CONFIG bits are implicitly reset by the hardware.

[1] MAXPHYADDR is reported in CPUID.80000008H:EAX[7:0].

Table 15-10. IA32_HW_FEEDBACK_CONFIG Control Options

Pre-Bit 1	Pre-Bit 0	Post-Bit 1	Post-Bit 0	Action	IA32_PACKAGE_THERM_STATUS [bit 26] and Interrupt
0	0	0	0	Reset value.	Both Hardware Feedback Interface and Intel Thread Director are disabled, no status bit set, no interrupt is generated.
0	0	0	1	Enable HFI structure.	Set the status bit and generate interrupt if enabled.
0	0	1	0	Invalid option; quietly ignored by the hardware.	No action (no update in the table).
0	0	1	1	Enable HFI and Intel Thread Director.	Set the status bit and generate interrupt if enabled.
0	1	0	0	Disable HFI support.	Set the status bit and generate interrupt if enabled.
0	1	1	0	Disable HFI and Intel Thread Director.	Set the status bit and generate interrupt if enabled.
0	1	1	1	Enable Intel Thread Director.	Set the status bit and generate interrupt if enabled.
1	0	0	0	No action; keeps HFI and Intel Thread Director disabled.	No action (no update in the table).
1	0	0	1	Enable HFI.	Set the status bit and generate interrupt if enabled.
1	0	1	1	Enable HFI and Intel Thread Director.	Set the status bit and generate interrupt if enabled.
1	1	0	0	Disable HFI and Intel Thread Director.	Set the status bit and generate interrupt if enabled.
1	1	0	1	Disable Intel Thread Director; keep HFI enabled.	No action (no update in the table).
1	1	1	0	Disable HFI and Intel Thread Director.	Set the status bit and generate interrupt if enabled.

15.6.6 Hardware Feedback Interface Notifications

The IA32_PACKAGE_THERM_STATUS MSR is extended with a new bit, hardware feedback interface structure change status (bit 26, R/WC0), to indicate that the hardware has updated the HFI/Intel Thread Director structure. This is a sticky bit and once set, indicates that the OS should read the structure to determine the change and adjust its scheduling decisions. Once set, the hardware will not generate any further updates to this structure until the OS clears this bit by writing 0.

The OS can enable interrupt-based notifications when the structure is updated by hardware through a new enable bit, hardware feedback interrupt enable (bit 25, R/W), in the IA32_PACKAGE_THERM_INTERRUPT MSR. When this bit is set to 1, it enables the generation of an interrupt when the HFI/Intel Thread Director structure is updated by hardware. When the enable bit transitions from 0 to 1, hardware will generate an initial notify, with the IA32_PACKAGE_THERM_STATUS bit 26 set to 1, to indicate that the OS should read the current HFI/Intel Thread Director structure.

15.6.7 Hardware Feedback Interface and Intel® Thread Director Structure Dynamic Update

The HFI/Intel Thread Director structure can be updated dynamically during run time. Changes to the structure may occur to one or more of its cells. Such changes may occur for one or more logical processors. The hardware sets a non-zero value in the "capability change" field of the HFI/Intel Thread Director structure as an indication for the OS to read that capability for all logical processors. A thermal interrupt is delivered to indicate to the OS that the structure has just changed. Section 15.6.6 contains more details on this notification mechanism. The hardware clears all "capability change" fields after the OS resets IA32_PACKAGE_THERM_STATUS[bit 26].

Zeroing a performance or energy efficiency cell hints to the OS that it is beneficial not to schedule software threads of that class on the associated logical processor for performance or energy efficiency reasons, respectively. If SMT is supported, it may be the case that the hardware zeroes one of the core's logical processors only. Zeroing the performance and energy efficiency cells of all classes for a logical processor implies that the hardware provides a hint to the OS to completely avoid scheduling work on that logical processor.

Zeroing a performance and energy efficiency cell hint of a logical processor across all classes along with Capability Flag bit 1 set to 1 across all capabilities and classes, indicates to the OS to force idle logical processor(s), and if affinitized activity occurs on those logical processor(s), the OS should inject idle periods such that overall utilization of those idled cores has a minimal-to-no impact to power. Capability Flag bit 1 will be set to 1 while this hint persists.

When EE=255 is set on one or more logical processors, it represents a request that the OS attempt to consolidate work to those logical processors with EE=255. These requests are made when the SOC has knowledge that consolidating the work to a subset of cores will result in significantly better platform energy efficiency. Examples of consolidating work would include, but not limited to, delaying less important work as needed to provide compute bandwidth for more important work, and routing interrupts to the logical processors with EE=255. When the cumulative workload requires performance greater than that which is available on the subset of cores with EE=255, it is expected that the OS will scale the work out to additional logical processors.

A few example reasons for runtime changes in the HGS/Intel Thread Director Table:

- Over clocking run time update that changes the capability values.
- Change in run time physical constraints.
- Run time performance or energy efficiency optimization.
- Change in core frequency, voltage, or power budget.

15.6.8 Logical Processor Scope Intel® Thread Director Configuration

The operating system enables Intel Thread Director at the logical processor scope using a logical processor scope MSR named IA32_HW_FEEDBACK_THREAD_CONFIG (address 17D4H).

The MSR is read/write and is structured as follows:

- Bits 63:1 – Reserved.
- Bit 0 – Enables Intel Thread Director. When set to 1, logical processor scope Intel Thread Director is enabled. Default is 0 (disabled).

Bit 0 of the logical processor scope configuration MSR can be cleared or set regardless of the state of the HFI/Intel Thread Director package configuration MSR state. Even when bit 0 of all logical processor configuration MSRs is clear, the processor can still update the Intel Thread Director structure if it is still enabled in the IA32_HW_FEEDBACK_CONFIG package scope MSR. When the operating system clears IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0], hardware clears the history accumulated on that logical processor which otherwise drives assigning the Class ID to the software thread that executes on that logical processor. As long as IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0] is set, the Class ID is available for the operating system to read, independent of the state of the package scope IA32_HW_FEEDBACK_CONFIG[1:0] bits.

See Section 15.6.9 for details on scenarios where IA32_HW_FEEDBACK_CONFIG bits are implicitly reset by the hardware.

15.6.9 Implicit Reset of Package and Logical Processor Scope Configuration MSRs

HFI/Intel Thread Director enable bits are reset by hardware in the following scenarios:

1. When GETSEC[SENTER] is executed:
   a. The processor implicitly resets the HFI/Intel Thread Director enable bits in the IA32_HW_FEEDBACK_CONFIG MSR on all sockets (packages) in the system.
   b. The processor implicitly resets the Intel Thread Director enable bit in the IA32_HW_FEEDBACK_THREAD_CONFIG MSR on all logical processors in the system across all sockets.
   c. The processor implicitly clears the HFI/Intel Thread Director table structure pointer in the IA32_HW_FEEDBACK_PTR package MSR across all sockets.
2. When GETSEC[ENTERACCS] is executed:
   a. The processor implicitly resets the HFI/Intel Thread Director enable bits in the IA32_HW_FEEDBACK_CONFIG MSR on the socket where the GETSEC[ENTERACCS] instruction was executed.
   b. The processor implicitly resets the Intel Thread Director enable bit in the IA32_HW_FEEDBACK_THREAD_CONFIG MSR on all logical processors on the socket where the GETSEC[ENTERACCS] instruction was executed.
   c. The processor implicitly clears the HFI/Intel Thread Director table structure pointer in the IA32_HW_FEEDBACK_PTR package MSR on the socket where the GETSEC[ENTERACCS] instruction was executed.
3. When an INIT or a wait-for-SIPI state are processed by a logical processor:
   a. The processor implicitly resets the Intel Thread Director enable bit in the IA32_HW_FEEDBACK_THREAD_CONFIG MSR on that logical processor, whether the signal was in the context of GETSEC[ENTERACCS] or not.

If the OS requires HFI/Intel Thread Director to be active after exiting the measured environment or when processing a SIPI event, it should re-enable HFI/Intel Thread Director.

15.6.10 Logical Processor Scope Intel® Thread Director Run Time Characteristics

The processor provides the operating system with run time feedback about the execution characteristics of the software thread executing on logical processors whose IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0] is set.

The run time feedback is communicated via a read-only MSR named IA32_THREAD_FEEDBACK_CHAR. This is a logical processor scope MSR whose address is 17D2H. This MSR is structured as follows:

- Bit 63 – Valid bit. When set to 1 the OS Scheduler can use the Class ID (in bits 7:0) for its scheduling decisions. If this bit is 0, the Class ID field should be ignored. It is recommended that the OS uses the last known Class ID of the software thread for its scheduling decisions.
- Bits 62:8 – Reserved.
- Bits 7:0 – Application Class ID, pointing into the Intel Thread Director structure described in Table 15-8.

This MSR is valid only if CPUID.06H:EAX[bit 23] is set.

The valid bit is cleared by the hardware in the following cases:

- The hardware does not have enough information to provide the operating system with a reliable Class ID.
- The operating system cleared the logical processor's IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0] bit.

The HRESET instruction is executed while configured to reset the Intel Thread Director history.

15.6.11 Logical Processor Scope History

The operating system can reset the Intel Thread Director related history accumulated on the current logical processor it is executing on by issuing the HRESET instruction. See "CPUID—CPU Identification" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for enumeration of the HRESET instruction. See also the “HRESET—History Reset” instruction description in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

15.6.11.1 Enabling Intel® Thread Director History Reset

The IA32_HRESET_ENABLE MSR is a read/write MSR and is structured as follows:

- Bits 63:32 – Reserved.
- Bits 31:1 – Reserved for other capabilities that can be reset by the HRESET instruction.
- Bit 0 – Enable reset of the Intel Thread Director history.

The operating system should set IA32_HRESET_ENABLE[bit 0] to enable Intel Thread Director history reset via the HRESET instruction.

15.6.11.2 Implicit Intel® Thread Director History Reset

The Intel Thread Director history is implicitly reset in the following scenarios:

1. When the processor enters or exits SMM mode and IA32_DEBUGCTL MSR.FREEZE_WHILE_SMM (bit 14) is set, the Intel Thread Director history is implicitly reset by the processor.
2. When GETSEC[SENTER] is executed, the processor resets the Intel Thread Director history on all logical processors in the system, including logical processors on other sockets (other than the one GETSEC[SENTER] is executed).
3. When GETSEC[ENTERACCS] is executed, the processor resets the Intel Thread Director history on the logical processor it is executed on.
4. When an INIT or a wait-for-SIPI state are processed by a logical processor, the Intel Thread Director history is reset whether the signal was a result of GETSEC[ENTERACCS] or not.

If the operating system requires HFI/Intel Thread Director to be active after exiting the measured environment or when processing a SIPI event, it should re-enable HFI/Intel Thread Director.

15.7 MWAIT EXTENSIONS FOR ADVANCED POWER MANAGEMENT

IA-32 processors may support a number of C-states[1] that reduce power consumption for inactive states. Intel Core Solo and Intel Core Duo processors support both deeper C-state and MWAIT extensions that can be used by OS to implement power management policy.

Software should use CPUID to discover if a target processor supports the enumeration of MWAIT extensions. If CPUID.05H.ECX[Bit 0] = 1, the target processor supports MWAIT extensions and their enumeration (see Chapter 4, “Instruction Set Reference, M-U,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B).

If CPUID.05H.ECX[Bit 1] = 1, the target processor supports using interrupts as break-events for MWAIT, even when interrupts are disabled. Use this feature to measure C-state residency as follows:

- Software can write to bit 0 in the MWAIT Extensions register (ECX) when issuing an MWAIT to enter into a processor-specific C-state or sub C-state.
- When a processor comes out of an inactive C-state or sub C-state, software can read a timestamp before an interrupt service routine (ISR) is potentially executed.

CPUID.05H.EDX allows software to enumerate processor-specific C-states and sub C-states available for use with MWAIT extensions. IA-32 processors may support more than one C-state of a given C-state type. These are called sub C-states. Numerically higher C-state have higher power savings and latency (upon entering and exiting) than lower-numbered C-state.

[1] The processor-specific C-states defined in MWAIT extensions can map to ACPI defined C-state types (C0, C1, C2, C3). The mapping relationship depends on the definition of a C-state by processor implementation and is exposed to OSPM by the BIOS using the ACPI defined _CST table.

At CPL = 0, system software can specify desired C-state and sub C-state by using the MWAIT hints register (EAX). Processors will not go to C-state and sub C-state deeper than what is specified by the hint register. If CPL > 0 and if MONITOR/MWAIT is supported at CPL > 0, the processor will only enter C1-state (regardless of the C-state request in the hints register).

Executing MWAIT generates an exception on processors operating at a privilege level where MONITOR/MWAIT are not supported.

NOTE

If MWAIT is used to enter a C-state (including sub C-state) that is numerically higher than C1, a store to the address range armed by MONITOR instruction will cause the processor to exit MWAIT if the store was originated by other processor agents. A store from non-processor agent may not cause the processor to exit MWAIT.

15.8 THERMAL MONITORING AND PROTECTION

The IA-32 architecture provides the following mechanisms for monitoring temperature and controlling thermal power:

1. The **catastrophic shutdown detector** forces processor execution to stop if the processor's core temperature rises above a preset limit.
2. **Automatic and adaptive thermal monitoring mechanism**s force the processor to reduce it's power consumption in order to operate within predetermined temperature limits.
3. The **software controlled clock modulation mechanism** permits operating systems to implement power management policies that reduce power consumption; this is in addition to the reduction offered by automatic thermal monitoring mechanisms.
4. **On-die digital thermal sensor and interrupt mechanisms** permit the OS to manage thermal conditions natively without relying on BIOS or other system board components.

The first mechanism is not visible to software. The other three mechanisms are visible to software using processor feature information returned by executing CPUID with EAX = 1.

The second mechanism includes:

- **Automatic thermal monitoring** provides two modes of operation. One mode modulates the clock duty cycle; the second mode changes the processor's frequency. Both modes are used to control the core temperature of the processor.
- **Adaptive thermal monitoring** can provide flexible thermal management on processors made of multiple cores.

The third mechanism modulates the clock duty cycle of the processor. As shown in Figure 15-24, the phrase 'duty cycle' does not refer to the actual duty cycle of the clock signal. Instead it refers to the time period during which the clock signal is allowed to drive the processor chip. By using the stop clock mechanism to control how often the processor is clocked, processor power consumption can be modulated.

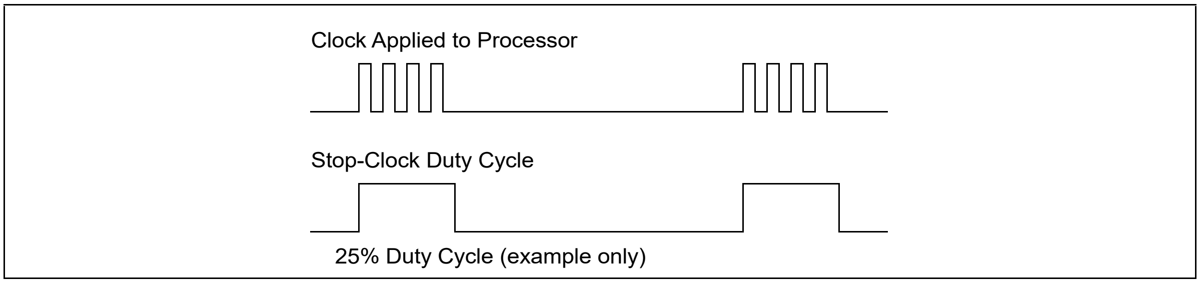


Figure 15-24. Processor Modulation Through Stop-Clock Mechanism

For previous automatic thermal monitoring mechanisms, software controlled mechanisms that changed processor operating parameters to impact changes in thermal conditions. Software did not have native access to the native thermal condition of the processor; nor could software alter the trigger condition that initiated software program control.

The fourth mechanism (listed above) provides access to an on-die digital thermal sensor using a model-specific register and uses an interrupt mechanism to alert software to initiate digital thermal monitoring.

15.8.1 Catastrophic Shutdown Detector

P6 family processors introduced a thermal sensor that acts as a catastrophic shutdown detector. This catastrophic shutdown detector was also implemented in Pentium 4, Intel Xeon and Pentium M processors. It is always enabled. When processor core temperature reaches a factory preset level, the sensor trips and processor execution is halted until after the next reset cycle.

15.8.2 Thermal Monitor

Pentium 4, Intel Xeon and Pentium M processors introduced a second temperature sensor that is factory-calibrated to trip when the processor’s core temperature crosses a level corresponding to the recommended thermal design envelop. The trip-temperature of the second sensor is calibrated below the temperature assigned to the catastrophic shutdown detector.

15.8.2.1 Thermal Monitor 1

The Pentium 4 processor uses the second temperature sensor in conjunction with a mechanism called Thermal Monitor 1 (TM1) to control the core temperature of the processor. TM1 controls the processor’s temperature by modulating the duty cycle of the processor clock. Modulation of duty cycles is processor model specific. Note that the processors STPCLK# pin is not used here; the stop-clock circuitry is controlled internally.

Support for TM1 is indicated by CPUID.1:EDX.TM[bit 29] = 1.

TM1 is enabled by setting the thermal-monitor enable flag (bit 3) in IA32_MISC_ENABLE; see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. Following a power-up or reset, the flag is cleared, disabling TM1. BIOS is required to enable only one automatic thermal monitoring modes. Operating systems and applications must not disable the operation of these mechanisms.

15.8.2.2 Thermal Monitor 2

An additional automatic thermal protection mechanism, called Thermal Monitor 2 (TM2), was introduced in the Intel Pentium M processor and also incorporated in newer models of the Pentium 4 processor family. Intel Core Duo and Solo processors, and Intel Core 2 Duo processor family all support TM1 and TM2. TM2 controls the core temperature of the processor by reducing the operating frequency and voltage of the processor and offers a higher performance level for a given level of power reduction than TM1.

TM2 is triggered by the same temperature sensor as TM1. The mechanism to enable TM2 may be implemented differently across various IA-32 processor families with different CPUID signatures in the family encoding value, but will be uniform within an IA-32 processor family.

Support for TM2 is indicated by CPUID.1:ECX.TM2[bit 8] = 1.

15.8.2.3 Two Methods for Enabling TM2

On processors with CPUID family/model/stepping signature encoded as 0x69n or 0x6Dn (early Pentium M processors), TM2 is enabled if the TM_SELECT flag (bit 16) of the MSR_THERM2_CTL register is set to 1 (Figure 15-25) and bit 3 of the IA32_MISC_ENABLE register is set to 1.

Following a power-up or reset, the TM_SELECT flag may be cleared. BIOS is required to enable either TM1 or TM2. Operating systems and applications must not disable mechanisms that enable TM1 or TM2. If bit 3 of the IA32_-MISC_ENABLE register is set and TM_SELECT flag of the MSR_THERM2_CTL register is cleared, TM1 is enabled.

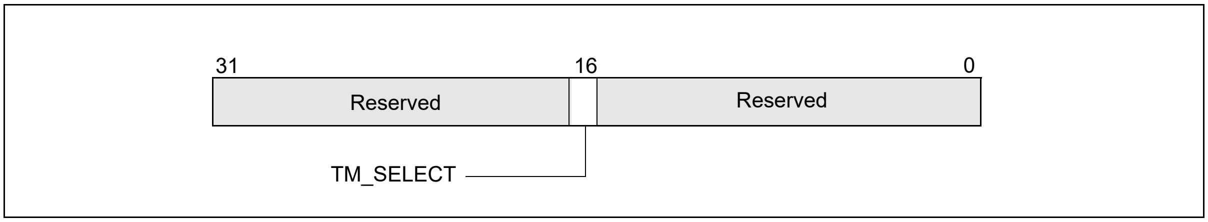


Figure 15-25. MSR_THERM2_CTL Register On Processors with CPUID Family/Model/Stepping Signature Encoded as 0x69n or 0x6Dn

On processors introduced after the Pentium 4 processor (this includes most Pentium M processors), the method used to enable TM2 is different. TM2 is enable by setting bit 13 of IA32_MISC_ENABLE register to 1. This applies to Intel Core Duo, Core Solo, and Intel Core 2 processor family.

The target operating frequency and voltage for the TM2 transition after TM2 is triggered is specified by the value written to MSR_THERM2_CTL, bits 15:0 (Figure 15-26). Following a power-up or reset, BIOS is required to enable at least one of these two thermal monitoring mechanisms. If both TM1 and TM2 are supported, BIOS may choose to enable TM2 instead of TM1. Operating systems and applications must not disable the mechanisms that enable TM1or TM2; and they must not alter the value in bits 15:0 of the MSR_THERM2_CTL register.

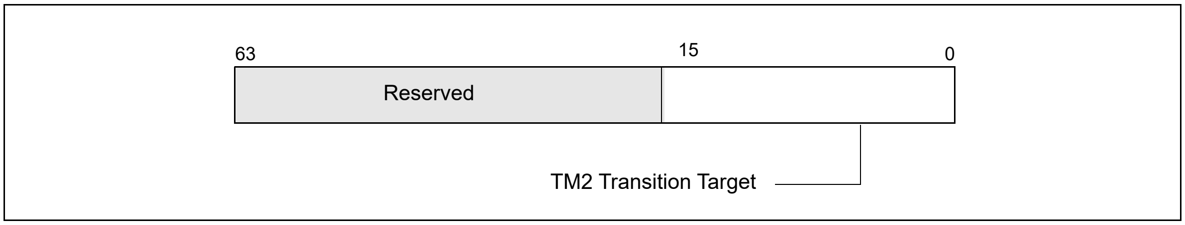


Figure 15-26. MSR_THERM2_CTL Register for Supporting TM2

15.8.2.4 Performance State Transitions and Thermal Monitoring

If the thermal control circuitry (TCC) for thermal monitor (TM1/TM2) is active, writes to the IA32_PERF_CTL will effect a new target operating point as follows:

- If TM1 is enabled and the TCC is engaged, the performance state transition can commence before the TCC is disengaged.
- If TM2 is enabled and the TCC is engaged, the performance state transition specified by a write to the IA32_PERF_CTL will commence after the TCC has disengaged.

15.8.2.5 Thermal Status Information

The status of the temperature sensor that triggers the thermal monitor (TM1/TM2) is indicated through the thermal status flag and thermal status log flag in the IA32_THERM_STATUS MSR (see Figure 15-27).

The functions of these flags are:

- **Thermal Status flag, bit 0** — When set, indicates that the processor core temperature is currently at the trip temperature of the thermal monitor and that the processor power consumption is being reduced via either TM1 or TM2, depending on which is enabled. When clear, the flag indicates that the core temperature is below the thermal monitor trip temperature. This flag is read only.

- **Thermal Status Log flag, bit 1** — When set, indicates that the thermal sensor has tripped since the last power-up or reset or since the last time that software cleared this flag. This flag is a sticky bit; once set it remains set until cleared by software or until a power-up or reset of the processor. The default state is clear.

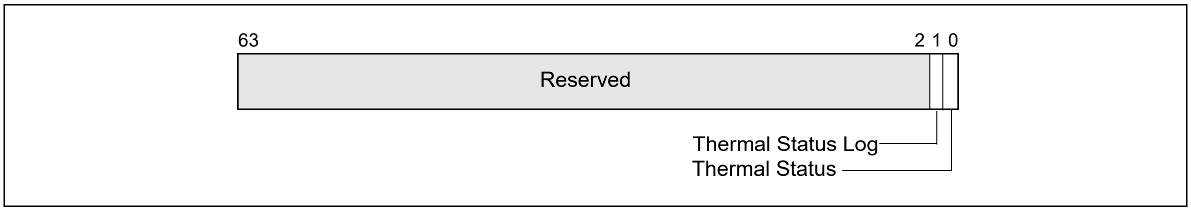


Figure 15-27. IA32_THERM_STATUS MSR

After the second temperature sensor has been tripped, the thermal monitor (TM1/TM2) will remain engaged for a minimum time period (on the order of 1 ms). The thermal monitor will remain engaged until the processor core temperature drops below the preset trip temperature of the temperature sensor, taking hysteresis into account.

While the processor is in a stop-clock state, interrupts will be blocked from interrupting the processor. This holding off of interrupts increases the interrupt latency, but does not cause interrupts to be lost. Outstanding interrupts remain pending until clock modulation is complete.

The thermal monitor can be programmed to generate an interrupt to the processor when the thermal sensor is tripped; this is called a thermal interrupt. The delivery mode, mask, and vector for this interrupt can be programmed through the thermal entry in the local APIC's LVT (see Section 11.5.1, "Local Vector Table"). The low-temperature interrupt enable and high-temperature interrupt enable flags in the IA32_THERM_INTERRUPT MSR (see Figure 15-28) control when the interrupt is generated; that is, on a transition from a temperature below the trip point to above and/or vice-versa.

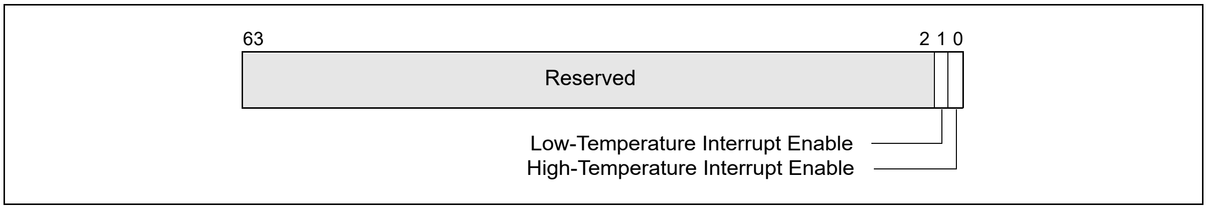

63
2 1 0
Reserved
Low-Temperature Interrupt Enable
High-Temperature Interrupt Enable

Figure 15-28. IA32_THERM_INTERRUPT MSR

- **High-Temperature Interrupt Enable flag, bit 0** — Enables an interrupt to be generated on the transition from a low-temperature to a high-temperature when set; disables the interrupt when clear.(R/W).
- **Low-Temperature Interrupt Enable flag, bit 1** — Enables an interrupt to be generated on the transition from a high-temperature to a low-temperature when set; disables the interrupt when clear.

The thermal interrupt can be masked by the thermal LVT entry. After a power-up or reset, the low-temperature interrupt enable and high-temperature interrupt enable flags in the IA32_THERM_INTERRUPT MSR are cleared (interrupts are disabled) and the thermal LVT entry is set to mask interrupts. This interrupt should be handled either by the operating system or system management mode (SMM) code.

Note that the operation of the thermal monitoring mechanism has no effect upon the clock rate of the processor's internal high-resolution timer (time stamp counter).

15.8.2.6 Adaptive Thermal Monitor

The Intel Core 2 Duo processor family supports enhanced thermal management mechanism, referred to as Adaptive Thermal Monitor (Adaptive TM).

Unlike TM2, Adaptive TM is not limited to one TM2 transition target. During a thermal trip event, Adaptive TM (if enabled) selects an optimal target operating point based on whether or not the current operating point has effectively cooled the processor.

Similar to TM2, Adaptive TM is enable by BIOS. The BIOS is required to test the TM1 and TM2 feature flags and enable all available thermal control mechanisms (including Adaptive TM) at platform initiation.

Adaptive TM is available only to a subset of processors that support TM2.

In each chip-multiprocessing (CMP) silicon die, each core has a unique thermal sensor that triggers independently. These thermal sensor can trigger TM1 or TM2 transitions in the same manner as described in Section 15.8.2.1 and Section 15.8.2.2. The trip point of the thermal sensor is not programmable by software since it is set during the fabrication of the processor.

Each thermal sensor in a processor core may be triggered independently to engage thermal management features. In Adaptive TM, both cores will transition to a lower frequency and/or lower voltage level if one sensor is triggered.

Triggering of this sensor is visible to software via the thermal interrupt LVT entry in the local APIC of a given core.

15.8.3 Software Controlled Clock Modulation

Pentium 4, Intel Xeon and Pentium M processors also support software-controlled clock modulation. This provides a means for operating systems to implement a power management policy to reduce the power consumption of the processor. Here, the stop-clock duty cycle is controlled by software through the IA32_CLOCK_MODULATION MSR (see Figure 15-29).

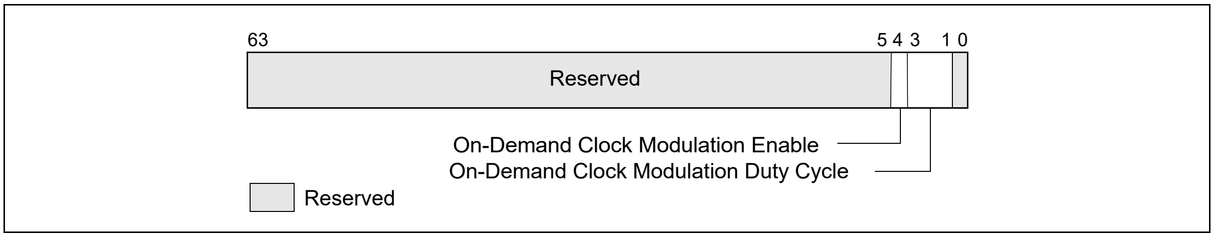

63
5 4 3
1 0
Reserved
On-Demand Clock Modulation Enable
On-Demand Clock Modulation Duty Cycle
Reserved

Figure 15-29. IA32_CLOCK_MODULATION MSR

The IA32_CLOCK_MODULATION MSR contains the following flag and field used to enable software-controlled clock modulation and to select the clock modulation duty cycle:

- **On-Demand Clock Modulation Enable, bit 4** — Enables on-demand software controlled clock modulation when set; disables software-controlled clock modulation when clear.
- **On-Demand Clock Modulation Duty Cycle, bits 1 through 3** — Selects the on-demand clock modulation duty cycle (see Table 15-11). This field is only active when the on-demand clock modulation enable flag is set.

Note that the on-demand clock modulation mechanism (like the thermal monitor) controls the processor's stop-clock circuitry internally to modulate the clock signal. The STPCLK# pin is not used in this mechanism.

Table 15-11. On-Demand Clock Modulation Duty Cycle Field Encoding

Duty Cycle Field Encoding	Duty Cycle
000B	Reserved
001B	12.5% (Default)
010B	25.0%
011B	37.5%
100B	50.0%
101B	63.5%
110B	75%
111B	87.5%

The on-demand clock modulation mechanism can be used to control processor power consumption. Power management software can write to the IA32_CLOCK_MODULATION MSR to enable clock modulation and to select a modulation duty cycle. If on-demand clock modulation and TM1 are both enabled and the thermal status of the processor is hot (bit 0 of the IA32_THERM_STATUS MSR is set), clock modulation at the duty cycle specified by TM1 takes precedence, regardless of the setting of the on-demand clock modulation duty cycle.

For Hyper-Threading Technology enabled processors, the IA32_CLOCK_MODULATION register is duplicated for each logical processor. In order for the On-demand clock modulation feature to work properly, the feature must be enabled on all the logical processors within a physical processor. If the programmed duty cycle is not identical for all the logical processors, the processor core clock will modulate to the highest duty cycle programmed for processors with any of the following CPUID DisplayFamily_DisplayModel signatures (see CPUID instruction in Chapter3, “Instruction Set Reference, A-L” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A): 06_1A, 06_1C, 06_1E, 06_1F, 06_25, 06_26, 06_27, 06_2C, 06_2E, 06_2F, 06_35, 06_36, and 0F_xx. For all other processors, if the programmed duty cycle is not identical for all logical processors in the same core, the processor core will modulate at the lowest programmed duty cycle.

For multiple processor cores in a physical package, each processor core can modulate to a programmed duty cycle independently.

For the P6 family processors, on-demand clock modulation was implemented through the chipset, which controlled clock modulation through the processor’s STPCLK# pin.

15.8.3.1 Extension of Software Controlled Clock Modulation

Extension of the software controlled clock modulation facility supports on-demand clock modulation duty cycle with 4-bit dynamic range (increased from 3-bit range). Granularity of clock modulation duty cycle is increased to 6.25% (compared to 12.5%).

Four bit dynamic range control is provided by using bit 0 in conjunction with bits 3:1 of the IA32_CLOCK_MODULATION MSR (see Figure 15-30).

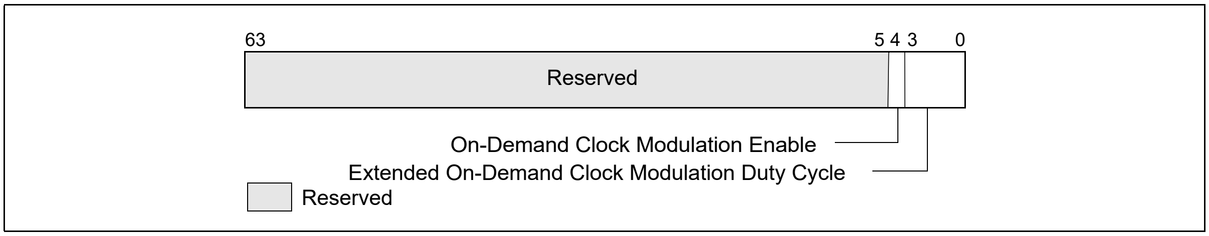

63
5
4
3
0
Reserved
On-Demand Clock Modulation Enable
Extended On-Demand Clock Modulation Duty Cycle
Reserved

Figure 15-30. IA32_CLOCK_MODULATION MSR with Clock Modulation Extension

Extension to software controlled clock modulation is supported only if CPUID.06H:EAX[Bit 5] = 1. If CPUID.06H:EAX[Bit 5] = 0, then bit 0 of IA32_CLOCK_MODULATION is reserved.

15.8.4 Detection of Thermal Monitor and Software Controlled Clock Modulation Facilities

The ACPI flag (bit 22) of the CPUID feature flags indicates the presence of the IA32_THERM_STATUS, IA32_THERM_INTERRUPT, IA32_CLOCK_MODULATION MSRs, and the xAPIC thermal LVT entry.

The TM1 flag (bit 29) of the CPUID feature flags indicates the presence of the automatic thermal monitoring facilities that modulate clock duty cycles.

15.8.4.1 Detection of Software Controlled Clock Modulation Extension

Processor’s support of software controlled clock modulation extension is indicated by CPUID.06H:EAX[Bit 5] = 1.

15.8.5 On Die Digital Thermal Sensors

On die digital thermal sensor can be read using an MSR (no I/O interface). In Intel Core Duo processors, each core has a unique digital sensor whose temperature is accessible using an MSR. The digital thermal sensor is the preferred method for reading the die temperature because (a) it is located closer to the hottest portions of the die, (b) it enables software to accurately track the die temperature and the potential activation of thermal throttling.

15.8.5.1 Digital Thermal Sensor Enumeration

The processor supports a digital thermal sensor if CPUID.06H.EAX[0] = 1. If the processor supports digital thermal sensor, EBX[bits 3:0] determine the number of thermal thresholds that are available for use.

Software sets thermal thresholds by using the IA32_THERM_INTERRUPT MSR. Software reads output of the digital thermal sensor using the IA32_THERM_STATUS MSR.

15.8.5.2 Reading the Digital Sensor

Unlike traditional analog thermal devices, the output of the digital thermal sensor is a temperature relative to the maximum supported operating temperature of the processor.

Temperature measurements returned by digital thermal sensors are always at or below TCC activation temperature. Critical temperature conditions are detected using the “Critical Temperature Status” bit. When this bit is set, the processor is operating at a critical temperature and immediate shutdown of the system should occur. Once the “Critical Temperature Status” bit is set, reliable operation is not guaranteed.

See Figure 15-31 for the layout of IA32_THERM_STATUS MSR. Bit fields include:

- **Thermal Status (bit 0, RO)** — This bit indicates whether the digital thermal sensor high-temperature output signal (PROCHOT#) is currently active. Bit 0 = 1 indicates the feature is active. This bit may not be written by software; it reflects the state of the digital thermal sensor.
- **Thermal Status Log (bit 1, R/WC0)** — This is a sticky bit that indicates the history of the thermal sensor high temperature output signal (PROCHOT#). Bit 1 = 1 if PROCHOT# has been asserted since a previous RESET or the last time software cleared the bit. Software may clear this bit by writing a zero.
- **PROCHOT# or FORCEPR# Event (bit 2, RO)** — Indicates whether PROCHOT# or FORCEPR# is being asserted by another agent on the platform.

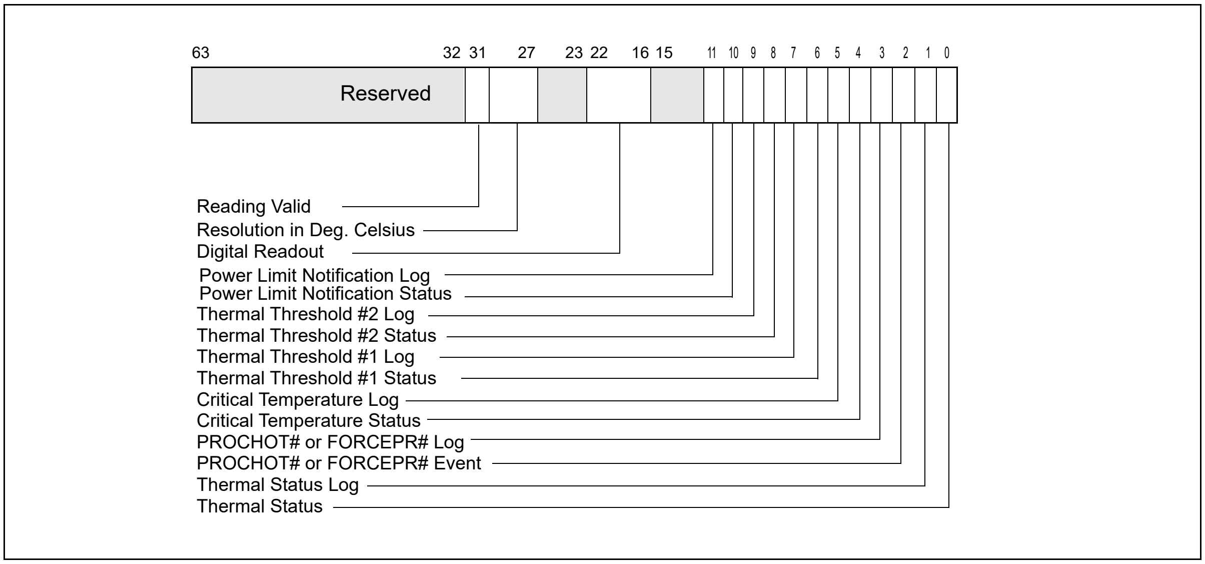


Figure 15-31. IA32_THERM_STATUS Register

- **PROCHOT# or FORCEPR# Log (bit 3, R/WC0)** — Sticky bit that indicates whether PROCHOT# or FORCEPR# has been asserted by another agent on the platform since the last clearing of this bit or a reset. If bit 3 = 1, PROCHOT# or FORCEPR# has been externally asserted. Software may clear this bit by writing a zero. External PROCHOT# assertions are only acknowledged if the Bidirectional Prochot feature is enabled.
- **Critical Temperature Status (bit 4, RO)** — Indicates whether the critical temperature detector output signal is currently active. If bit 4 = 1, the critical temperature detector output signal is currently active.
- **Critical Temperature Log (bit 5, R/WC0)** — Sticky bit that indicates whether the critical temperature detector output signal has been asserted since the last clearing of this bit or reset. If bit 5 = 1, the output signal has been asserted. Software may clear this bit by writing a zero.
- **Thermal Threshold #1 Status (bit 6, RO)** — Indicates whether the actual temperature is currently higher than or equal to the value set in Thermal Threshold #1. If bit 6 = 0, the actual temperature is lower. If bit 6 = 1, the actual temperature is greater than or equal to TT#1. Quantitative information of actual temperature can be inferred from Digital Readout, bits 22:16.
- **Thermal Threshold #1 Log (bit 7, R/WC0)** — Sticky bit that indicates whether the Thermal Threshold #1 has been reached since the last clearing of this bit or a reset. If bit 7 = 1, the Threshold #1 has been reached. Software may clear this bit by writing a zero.
- **Thermal Threshold #2 Status (bit 8, RO)** — Indicates whether actual temperature is currently higher than or equal to the value set in Thermal Threshold #2. If bit 8 = 0, the actual temperature is lower. If bit 8 = 1, the actual temperature is greater than or equal to TT#2. Quantitative information of actual temperature can be inferred from Digital Readout, bits 22:16.
- **Thermal Threshold #2 Log (bit 9, R/WC0)** — Sticky bit that indicates whether the Thermal Threshold #2 has been reached since the last clearing of this bit or a reset. If bit 9 = 1, the Thermal Threshold #2 has been reached. Software may clear this bit by writing a zero.
- **Power Limitation Status (bit 10, RO)** — Indicates whether the processor is currently operating below OS-requested P-state (specified in IA32_PERF_CTL) or OS-requested clock modulation duty cycle (specified in IA32_CLOCK_MODULATION). This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be delivered independently to IA32_PACKAGE_THERM_STATUS MSR.
- **Power Notification Log (bit 11, R/WCO)** — Sticky bit that indicates the processor went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification is indicated independently in IA32_PACKAGE_THERM_STATUS MSR.
- **Digital Readout (bits 22:16, RO)** — Digital temperature reading in 1 degree Celsius relative to the TCC activation temperature.

  0: TCC Activation temperature,

  1: (TCC Activation - 1) , etc. See the processor’s data sheet for details regarding TCC activation.

  A lower reading in the Digital Readout field (bits 22:16) indicates a higher actual temperature.
- **Resolution in Degrees Celsius (bits 30:27, RO)** — Specifies the resolution (or tolerance) of the digital thermal sensor. The value is in degrees Celsius. It is recommended that new threshold values be offset from the current temperature by at least the resolution + 1 in order to avoid hysteresis of interrupt generation.
- **Reading Valid (bit 31, RO)** — Indicates if the digital readout in bits 22:16 is valid. The readout is valid if bit 31 = 1.

Changes to temperature can be detected using two thresholds (see Figure 15-32); one is set above and the other below the current temperature. These thresholds have the capability of generating interrupts using the core's local APIC which software must then service. Note that the local APIC entries used by these thresholds are also used by the Intel® Thermal Monitor; it is up to software to determine the source of a specific interrupt.

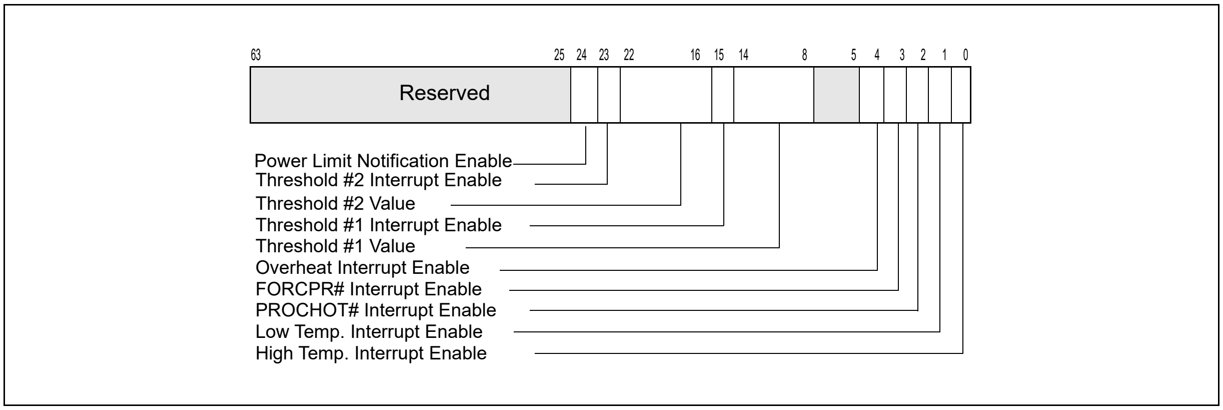

63
25
24
23
22
16
15
14
8
5
4
3
2
1
0
Reserved
Power Limit Notification Enable
Threshold #2 Interrupt Enable
Threshold #2 Value
Threshold #1 Interrupt Enable
Threshold #1 Value
Overheat Interrupt Enable
FORCPR# Interrupt Enable
PROCHOT# Interrupt Enable
Low Temp. Interrupt Enable
High Temp. Interrupt Enable

Figure 15-32. IA32_THERM_INTERRUPT Register

See Figure 15-32 for the layout of IA32_THERM_INTERRUPT MSR. Bit fields include:

- **High-Temperature Interrupt Enable (bit 0, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from low-temperature to a high-temperature threshold. Bit 0 = 0 (default) disables interrupts; bit 0 = 1 enables interrupts.
- **Low-Temperature Interrupt Enable (bit 1, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from high-temperature to a low-temperature (TCC de-activation). Bit 1 = 0 (default) disables interrupts; bit 1 = 1 enables interrupts.
- **PROCHOT# Interrupt Enable (bit 2, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when PROCHOT# has been asserted by another agent on the platform and the Bidirectional Prochot feature is enabled. Bit 2 = 0 disables the interrupt; bit 2 = 1 enables the interrupt.
- **FORCEPR# Interrupt Enable (bit 3, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when FORCEPR# has been asserted by another agent on the platform. Bit 3 = 0 disables the interrupt; bit 3 = 1 enables the interrupt.
- **Critical Temperature Interrupt Enable (bit 4, R/W)** — Enables the generation of an interrupt when the Critical Temperature Detector has detected a critical thermal condition. The recommended response to this condition is a system shutdown. Bit 4 = 0 disables the interrupt; bit 4 = 1 enables the interrupt.
- **Threshold #1 Value (bits 14:8, R/W)** — A temperature threshold, encoded relative to the TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Digital Readout and is used to generate the Thermal Threshold #1 Status and Log bits as well as the Threshold #1 thermal interrupt delivery.
- **Threshold #1 Interrupt Enable (bit 15, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Threshold #1 setting in any direction. Bit 15 = 1 enables the interrupt; bit 15 = 0 disables the interrupt.
- **Threshold #2 Value (bits 22:16, R/W)** —A temperature threshold, encoded relative to the TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Digital Readout and is used to generate the Thermal Threshold #2 Status and Log bits as well as the Threshold #2 thermal interrupt delivery.
- **Threshold #2 Interrupt Enable (bit 23, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Threshold #2 setting in any direction. Bit 23 = 1enables the interrupt; bit 23 = 0 disables the interrupt.
- **Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of power notification events when the processor went below OS-requested P-state or OS-requested clock modulation duty cycle. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be enabled independently by IA32_PACKAGE_THERM_INTERRUPT MSR.

15.8.6 Power Limit Notification

Platform firmware may be capable of specifying a power limit to restrict power delivered to a platform component, such as a physical processor package. This constraint imposed by platform firmware may occasionally cause the processor to operate below OS-requested P or T-state. A power limit notification event can be delivered using the existing thermal LVT entry in the local APIC.

Software can enumerate the presence of the processor's support for power limit notification by verifying CPUID.06H:EAX[bit 4] = 1.

If CPUID.06H:EAX[bit 4] = 1, then IA32_THERM_INTERRUPT and IA32_THERM_STATUS provides the following facility to manage power limit notification:

- Bits 10 and 11 in IA32_THERM_STATUS informs software of the occurrence of processor operating below OS-requested P-state or clock modulation duty cycle setting (see Figure 15-31).
- Bit 24 in IA32_THERM_INTERRUPT enables the local APIC to deliver a thermal event when the processor went below OS-requested P-state or clock modulation duty cycle setting (see Figure 15-32).

15.9 PACKAGE LEVEL THERMAL MANAGEMENT

The thermal management facilities like IA32_THERM_INTERRUPT and IA32_THERM_STATUS are often implemented with a processor core granularity. To facilitate software manage thermal events from a package level granularity, two architectural MSR is provided for package level thermal management. The IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT MSRs use similar interfaces as IA32_THERM_STATUS and IA32_THERM_INTERRUPT, but are shared in each physical processor package.

Software can enumerate the presence of the processor's support for package level thermal management facility (IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT) by verifying CPUID.06H:EAX[bit 6] = 1.

The layout of IA32_PACKAGE_THERM_STATUS MSR is shown in Figure 15-33.

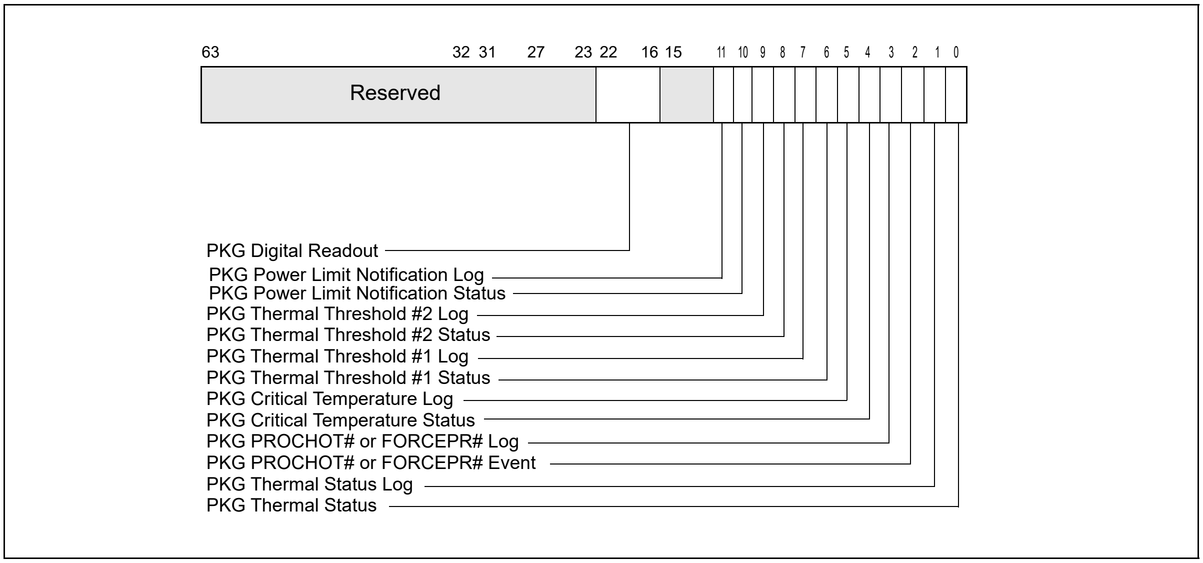

63
32 31
27
23 22
16 15
11 10 9 8 7 6 5 4 3 2 1 0
Reserved
PKG Digital Readout
PKG Power Limit Notification Log
PKG Power Limit Notification Status
PKG Thermal Threshold #2 Log
PKG Thermal Threshold #2 Status
PKG Thermal Threshold #1 Log
PKG Thermal Threshold #1 Status
PKG Critical Temperature Log
PKG Critical Temperature Status
PKG PROCHOT# or FORCEPR# Log
PKG PROCHOT# or FORCEPR# Event
PKG Thermal Status Log
PKG Thermal Status

Figure 15-33. IA32_PACKAGE_THERM_STATUS Register

- **Package Thermal Status (bit 0, RO)** — This bit indicates whether the digital thermal sensor high-temperature output signal (PROCHOT#) for the package is currently active. Bit 0 = 1 indicates the feature is active. This bit may not be written by software; it reflects the state of the digital thermal sensor.

- **Package Thermal Status Log (bit 1, R/WC0)** — This is a sticky bit that indicates the history of the thermal sensor high temperature output signal (PROCHOT#) of the package. Bit 1 = 1 if package PROCHOT# has been asserted since a previous RESET or the last time software cleared the bit. Software may clear this bit by writing a zero.
- **Package PROCHOT# Event (bit 2, RO)** — Indicates whether package PROCHOT# is being asserted by another agent on the platform.
- **Package PROCHOT# Log (bit 3, R/WC0)** — Sticky bit that indicates whether package PROCHOT# has been asserted by another agent on the platform since the last clearing of this bit or a reset. If bit 3 = 1, package PROCHOT# has been externally asserted. Software may clear this bit by writing a zero.
- **Package Critical Temperature Status (bit 4, RO)** — Indicates whether the package critical temperature detector output signal is currently active. If bit 4 = 1, the package critical temperature detector output signal is currently active.
- **Package Critical Temperature Log (bit 5, R/WC0)** — Sticky bit that indicates whether the package critical temperature detector output signal has been asserted since the last clearing of this bit or reset. If bit 5 = 1, the output signal has been asserted. Software may clear this bit by writing a zero.
- **Package Thermal Threshold #1 Status (bit 6, RO)** — Indicates whether the actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #1. If bit 6 = 0, the actual temperature is lower. If bit 6 = 1, the actual temperature is greater than or equal to PTT#1. Quantitative information of actual package temperature can be inferred from Package Digital Readout, bits 22:16.
- **Package Thermal Threshold #1 Log (bit 7, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #1 has been reached since the last clearing of this bit or a reset. If bit 7 = 1, the Package Threshold #1 has been reached. Software may clear this bit by writing a zero.
- **Package Thermal Threshold #2 Status (bit 8, RO)** — Indicates whether actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #2. If bit 8 = 0, the actual temperature is lower. If bit 8 = 1, the actual temperature is greater than or equal to PTT#2. Quantitative information of actual temperature can be inferred from Package Digital Readout, bits 22:16.
- **Package Thermal Threshold #2 Log (bit 9, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #2 has been reached since the last clearing of this bit or a reset. If bit 9 = 1, the Package Thermal Threshold #2 has been reached. Software may clear this bit by writing a zero.
- **Package Power Limitation Status (bit 10, RO)** — Indicates package power limit is forcing one ore more processors to operate below OS-requested P-state. Note that package power limit violation may be caused by processor cores or by devices residing in the uncore. Software can examine IA32_THERM_STATUS to determine if the cause originates from a processor core (see Figure 15-31).
- **Package Power Notification Log (bit 11, R/WCO)** — Sticky bit that indicates any processor in the package went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET.
- **Package Digital Readout (bits 22:16, RO)** — Package digital temperature reading in 1 degree Celsius relative to the package TCC activation temperature.

  0: Package TCC Activation temperature,

  1: (PTCC Activation - 1) , etc. See the processor’s data sheet for details regarding PTCC activation.

  A lower reading in the Package Digital Readout field (bits 22:16) indicates a higher actual temperature.

The layout of IA32_PACKAGE_THERM_INTERRUPT MSR is shown in Figure 15-34.

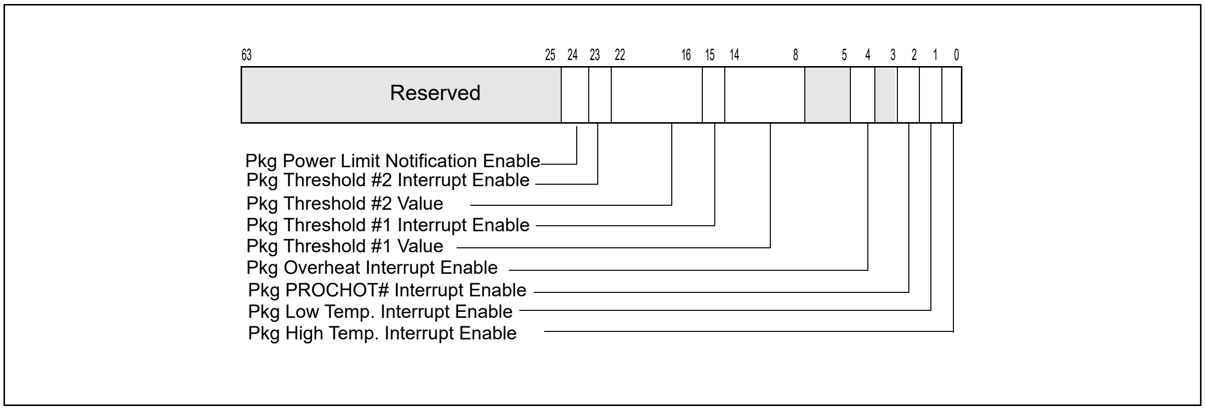

63
25
24
23
22
16
15
14
8
5
4
3
2
1
0
Reserved
Pkg Power Limit Notification Enable
Pkg Threshold #2 Interrupt Enable
Pkg Threshold #2 Value
Pkg Threshold #1 Interrupt Enable
Pkg Threshold #1 Value
Pkg Overheat Interrupt Enable
Pkg PROCHOT# Interrupt Enable
Pkg Low Temp. Interrupt Enable
Pkg High Temp. Interrupt Enable

Figure 15-34. IA32_PACKAGE_THERM_INTERRUPT Register

- **Package High-Temperature Interrupt Enable (bit 0, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from low-temperature to a package high-temperature threshold. Bit 0 = 0 (default) disables interrupts; bit 0 = 1 enables interrupts.
- **Package Low-Temperature Interrupt Enable (bit 1, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from high-temperature to a low-temperature (TCC de-activation). Bit 1 = 0 (default) disables interrupts; bit 1 = 1 enables interrupts.
- **Package PROCHOT# Interrupt Enable (bit 2, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when Package PROCHOT# has been asserted by another agent on the platform and the Bidirectional Prochot feature is enabled. Bit 2 = 0 disables the interrupt; bit 2 = 1 enables the interrupt.
- **Package Critical Temperature Interrupt Enable (bit 4, R/W)** — Enables the generation of an interrupt when the Package Critical Temperature Detector has detected a critical thermal condition. The recommended response to this condition is a system shutdown. Bit 4 = 0 disables the interrupt; bit 4 = 1 enables the interrupt.
- **Package Threshold #1 Value (bits 14:8, R/W)** — A temperature threshold, encoded relative to the Package TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Package Digital Readout and is used to generate the Package Thermal Threshold #1 Status and Log bits as well as the Package Threshold #1 thermal interrupt delivery.
- **Package Threshold #1 Interrupt Enable (bit 15, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #1 setting in any direction. Bit 15 = 1 enables the interrupt; bit 15 = 0 disables the interrupt.
- **Package Threshold #2 Value (bits 22:16, R/W)** —A temperature threshold, encoded relative to the PTCC Activation temperature (using the same format as the Package Digital Readout). This threshold is compared against the Package Digital Readout and is used to generate the Package Thermal Threshold #2 Status and Log bits as well as the Package Threshold #2 thermal interrupt delivery.
- **Package Threshold #2 Interrupt Enable (bit 23, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #2 setting in any direction. Bit 23 = 1 enables the interrupt; bit 23 = 0 disables the interrupt.
- **Package Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of package power notification events.

15.9.1 Support for Passive and Active cooling

Passive and active cooling may be controlled by the OS power management agent through ACPI control methods. On platforms providing package level thermal management facility described in the previous section, it is recommended that active cooling (FAN control) should be driven by measuring the package temperature using the IA32_PACKAGE_THERM_INTERRUPT MSR.

Passive cooling (frequency throttling) should be driven by measuring (a) the core and package temperatures, or (b) only the package temperature. If measured package temperature led the power management agent to choose which core to execute passive cooling, then all cores need to execute passive cooling. Core temperature is measured using the IA32_THERMAL_STATUS and IA32_THERMAL_INTERRUPT MSRs. The exact implementation details depend on the platform firmware and possible solutions include defining two different thermal zones (one for core temperature and passive cooling and the other for package temperature and active cooling).

15.10 PLATFORM SPECIFIC POWER MANAGEMENT SUPPORT

This section covers power management interfaces that are not architectural but addresses the power management needs of several platform specific components. Specifically, RAPL (Running Average Power Limit) interfaces provide mechanisms to enforce power consumption limit. Power limiting usages have specific usages in client and server platforms.

For client platform power limit control and for server platforms used in a data center, the following power and thermal related usages are desirable:

- Platform Thermal Management: Robust mechanisms to manage component, platform, and group-level thermals, either proactively or reactively (e.g., in response to a platform-level thermal trip point).
- Platform Power Limiting: More deterministic control over the system's power consumption, for example to meet battery life targets on rack-level or container-level power consumption goals within a datacenter.
- Power/Performance Budgeting: Efficient means to control the power consumed (and therefore the sustained performance delivered) within and across platforms.

The server and client usage models are addressed by RAPL interfaces, which expose multiple domains of power rationing within each processor socket. Generally, these RAPL domains may be viewed to include hierarchically:

- Package domain is the processor die.
- Memory domain includes the directly-attached DRAM; an additional power plane may constitute a separate domain.

In order to manage the power consumed across multiple sockets via RAPL, individual limits must be programmed for each processor complex. Programming specific RAPL domain across multiple sockets is not supported.

15.10.1 RAPL Interfaces

RAPL interfaces consist of non-architectural MSRs. Each RAPL domain supports the following set of capabilities, some of which are optional as stated below.

- Power limit - MSR interfaces to specify power limit, time window; lock bit, clamp bit etc.
- Energy Status - Power metering interface providing energy consumption information.
- Perf Status (Optional) - Interface providing information on the performance effects (regression) due to power limits. It is defined as a duration metric that measures the power limit effect in the respective domain. The meaning of duration is domain specific.
- Power Info (Optional) - Interface providing information on the range of parameters for a given domain, minimum power, maximum power etc.
- Policy (Optional) - 4-bit priority information that is a hint to hardware for dividing budget between sub-domains in a parent domain.

Each of the above capabilities requires specific units in order to describe them. Power is expressed in Watts, Time is expressed in Seconds, and Energy is expressed in Joules. Scaling factors are supplied to each unit to make the information presented meaningful in a finite number of bits. Units for power, energy, and time are exposed in the read-only MSR_RAPL_POWER_UNIT MSR.

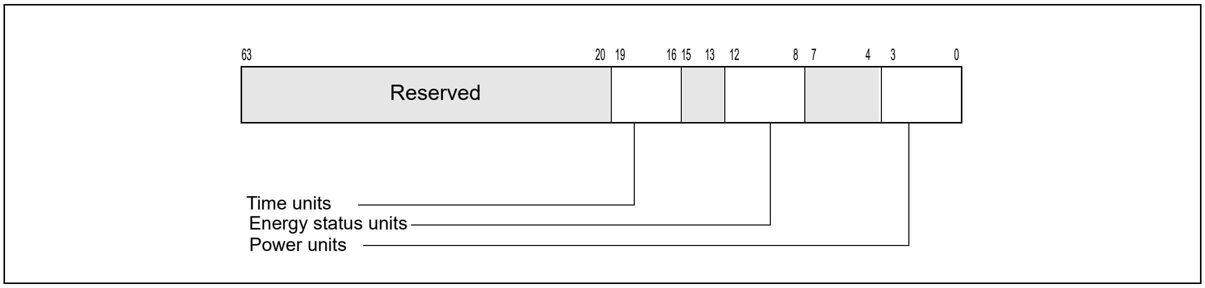


Figure 15-35. MSR_RAPL_POWER_UNIT Register

MSR_RAPL_POWER_UNIT (Figure 15-35) provides the following information across all RAPL domains:

- **Power Units** (bits 3:0): Power related information (in Watts) is based on the multiplier, 1/ 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0011b, indicating power unit is in 1/8 Watts increment.
- **Energy Status Units** (bits 12:8): Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 10000b, indicating energy status unit is in 15.3 micro-Joules increment.
- **Time Units** (bits 19:16): Time related information (in Seconds) is based on the multiplier, 1/ 2^TU; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating time unit is in 976 micro-seconds increment.

15.10.2 RAPL Domains and Platform Specificity

The specific RAPL domains available in a platform vary across product segments. Platforms targeting the client segment support the following RAPL domain hierarchy:

- Package
- Two power planes: PP0 and PP1 (PP1 may reflect to uncore devices)

Platforms targeting the server segment support the following RAPL domain hierarchy:

- Package
- Power plane: PP0
- DRAM

Each level of the RAPL hierarchy provides a respective set of RAPL interface MSRs. Table 15-12 lists the RAPL MSR interfaces available for each RAPL domain. The power limit MSR of each RAPL domain is located at offset 0 relative to an MSR base address which is non-architectural; see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. The energy status MSR of each domain is located at offset 1 relative to the MSR base address of respective domain.

Table 15-12. RAPL MSR Interfaces and RAPL Domains

Domain	Power Limit (Offset 0)	Energy Status (Offset 1)	Policy (Offset 2)	Perf Status (Offset 3)	Power Info (Offset 4)
PKG	MSR_PKG_POWER_LIMIT	MSR_PKG_ENERGY_STATUS	RESERVED	MSR_PKG_PERF_STATUS	MSR_PKG_POWER_INFO
DRAM	MSR_DRAM_POWER_LIMIT	MSR_DRAM_ENERGY_STATUS	RESERVED	MSR_DRAM_PERF_STATUS	MSR_DRAM_POWER_INFO
PP0	MSR_PP0_POWER_LIMIT	MSR_PP0_ENERGY_STATUS	MSR_PP0_POLICY	MSR_PP0_PERF_STATUS	RESERVED
PP1	MSR_PP1_POWER_LIMIT	MSR_PP1_ENERGY_STATUS	MSR_PP1_POLICY	RESERVED	RESERVED

The presence of the optional MSR interfaces (the three right-most columns of Table 15-12) may be model-specific. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for details.

15.10.3 Package RAPL Domain

The MSR interfaces defined for the package RAPL domain are:

- MSR_PKG_POWER_LIMIT allows software to set power limits for the package and measurement attributes associated with each limit,
- MSR_PKG_ENERGY_STATUS reports measured actual energy usage,
- MSR_PKG_POWER_INFO reports the package power range information for RAPL usage.

MSR_PKG_PERF_STATUS can report the performance impact of power limiting, but its availability may be model-specific.

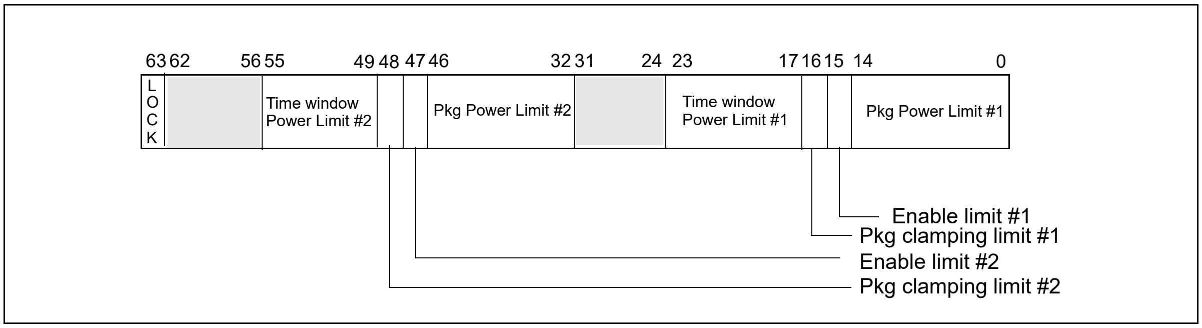


Figure 15-36. MSR_PKG_POWER_LIMIT Register

MSR_PKG_POWER_LIMIT allows a software agent to define power limitation for the package domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_PKG_POWER_LIMIT. Two power limits can be specified, corresponding to time windows of different sizes. Each power limit provides independent clamping control that would permit the processor cores to go below OS-requested state to meet the power limits. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and un-modifiable until next RESET.

The bit fields of MSR_PKG_POWER_LIMIT (Figure 15-36) are:

- **Package Power Limit #1**(bits 14:0): Sets the average power usage limit of the package domain corresponding to time window # 1. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.

- **Enable Power Limit #1**(bit 15): 0 = disabled; 1 = enabled.
- **Package Clamping Limitation #1** (bit 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit #1** (bits 23:17): Indicates the time window for power limit #1

  Time limit = 2^Y * (1.0 + Z/4.0) * Time_Unit

  Here "Y" is the unsigned integer value represented. by bits 21:17, "Z" is an unsigned integer represented by bits 23:22. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Package Power Limit #2**(bits 46:32): Sets the average power usage limit of the package domain corresponding to time window # 2. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #2**(bit 47): 0 = disabled; 1 = enabled.
- **Package Clamping Limitation #2** (bit 48): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit #2** (bits 55:49): Indicates the time window for power limit #2

  Time limit = 2^Y * (1.0 + Z/4.0) * Time_Unit

  Here "Y" is the unsigned integer value represented. by bits 53:49, "Z" is an unsigned integer represented by bits 55:54. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT. This field may have a hard-coded value in hardware and ignores values written by software.
- **Lock** (bit 63): If set, all write attempts to this MSR are ignored until next RESET.

MSR_PKG_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the package domain. This MSR is updated every ~1msec. It has a wraparound time of around 60 secs when power consumption is high, and may be longer otherwise.

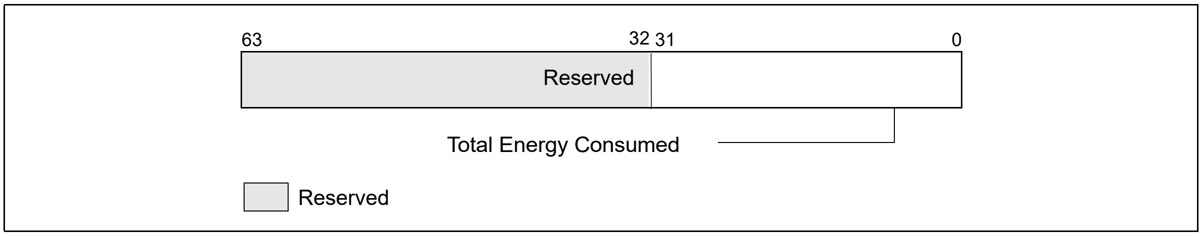


Figure 15-37. MSR_PKG_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PKG_POWER_INFO is a read-only MSR. It reports the package power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the package domain. It also provides the largest possible time window for software to program the RAPL interface.

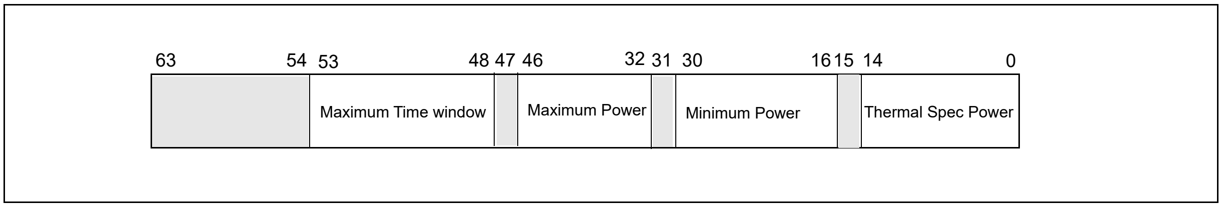


Figure 15-38. MSR_PKG_POWER_INFO Register

- **Thermal Spec Power** (bits 14:0): The unsigned integer value is the equivalent of thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.

- **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Time Window** (bits 53:48): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_PKG_POWER_LIMIT. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

MSR_PKG_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific; see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

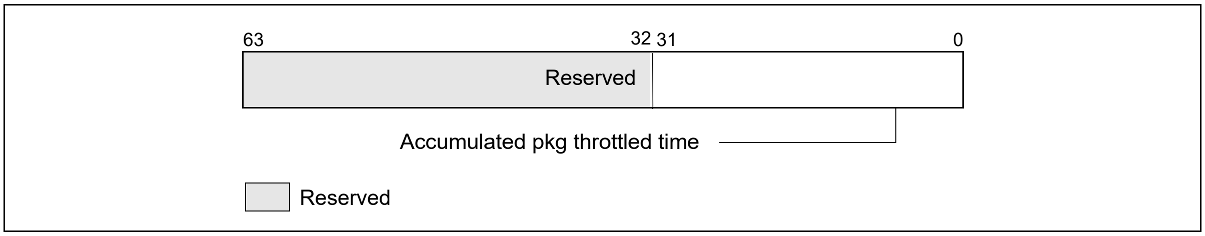


Figure 15-39. MSR_PKG_PERF_STATUS MSR

- **Accumulated Package Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the package has throttled. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

15.10.4 PP0/PP1 RAPL Domains

The MSR interfaces defined for the PP0 and PP1 domains are identical in layout. Generally, PP0 refers to the processor cores. The availability of PP1 RAPL domain interface is platform-specific. For a client platform, the PP1 domain refers to the power plane of a specific device in the uncore. For server platforms, the PP1 domain is not supported, but its PP0 domain supports the MSR_PP0_PERF_STATUS interface.

- MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allow software to set power limits for the respective power plane domain.
- MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS report actual energy usage on a power plane.
- MSR_PP0_POLICY/MSR_PP1_POLICY allow software to adjust balance for respective power plane.

MSR_PP0_PERF_STATUS can report the performance impact of power limiting, but it is not available in client platforms.

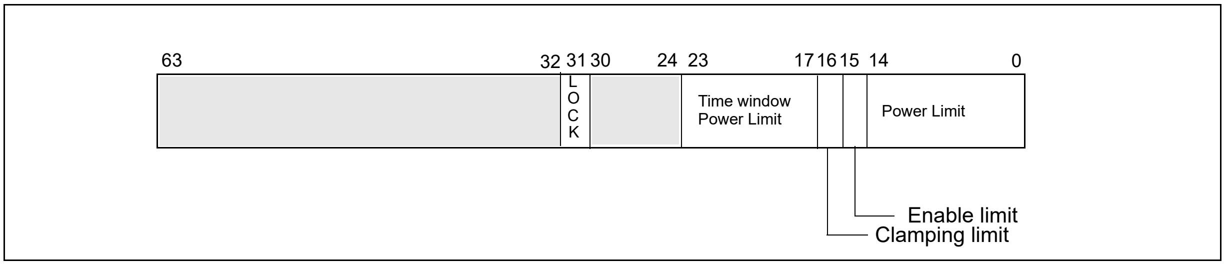


Figure 15-40. MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT Register

MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allow a software agent to define power limitation for the respective power plane domain. A lock mechanism in each power plane domain allows the software agent to enforce power limit settings independently. Once a lock bit is set, the power limit settings in that power plane are static and un-modifiable until next RESET.

The bit fields of MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT (Figure 15-40) are:

- **Power Limit** (bits 14:0): Sets the average power usage limit of the respective power plane domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit** (bit 15): 0 = disabled; 1 = enabled.
- **Clamping Limitation** (bit 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit #1 will be used by the processor. The numeric value encoded by bits 23:17 is represented by the product of 2^Y *F; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Lock** (bit 31): If set, all write attempts to the MSR and corresponding policy MSR_PP0_POLICY/MSR_PP1_POLICY are ignored until next RESET.

MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS are read-only MSRs. They report the actual energy use for the respective power plane domains. These MSRs are updated every ~1msec.

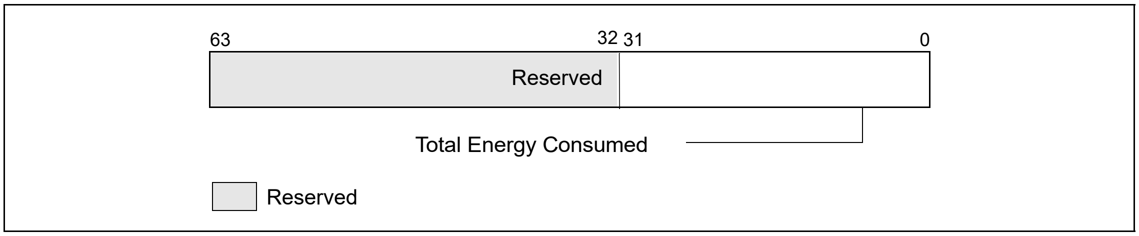


Figure 15-41. MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since the last time this register was cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PP0_POLICY/MSR_PP1_POLICY provide balance power policy control for each power plane by providing inputs to the power budgeting management algorithm. On platforms that support PP0 (IA cores) and PP1 (uncore graphic device), the default values give priority to the non-IA power plane. These MSRs enable the PCU to balance power consumption between the IA cores and uncore graphic device.

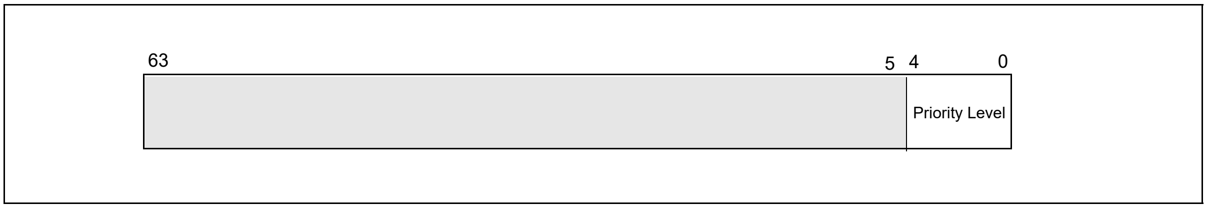


Figure 15-42. MSR_PP0_POLICY/MSR_PP1_POLICY Register

- **Priority Level** (bits 4:0): Priority level input to the PCU for respective power plane. PP0 covers the IA processor cores, PP1 covers the uncore graphic device. The value 31 is considered highest priority.

MSR_PP0_PERF_STATUS is a read-only MSR. It reports the total time for which the PP0 domain was throttled due to the power limits. This MSR is supported only in server platform. Throttling in this context is defined as going below the OS-requested P-state or T-state.

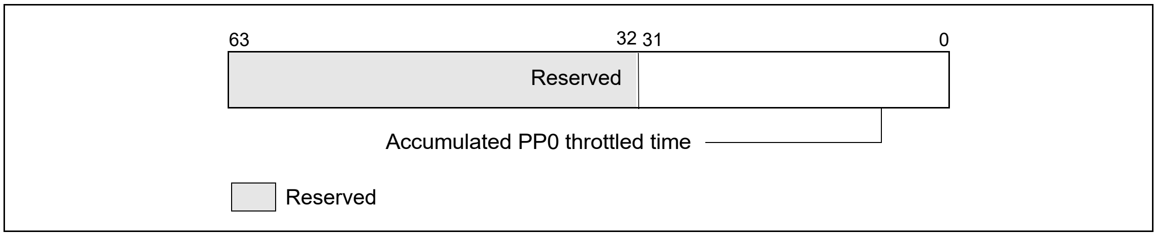


Figure 15-43. MSR_PP0_PERF_STATUS MSR

- **Accumulated PP0 Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the PP0 domain has throttled. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

15.10.5 DRAM RAPL Domain

The MSR interfaces defined for the DRAM domains are supported only in the server platform. The MSR interfaces are:

- MSR_DRAM_POWER_LIMIT allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.
- MSR_DRAM_ENERGY_STATUS reports measured actual energy usage.
- MSR_DRAM_POWER_INFO reports the DRAM domain power range information for RAPL usage.
- MSR_DRAM_PERF_STATUS can report the performance impact of power limiting.

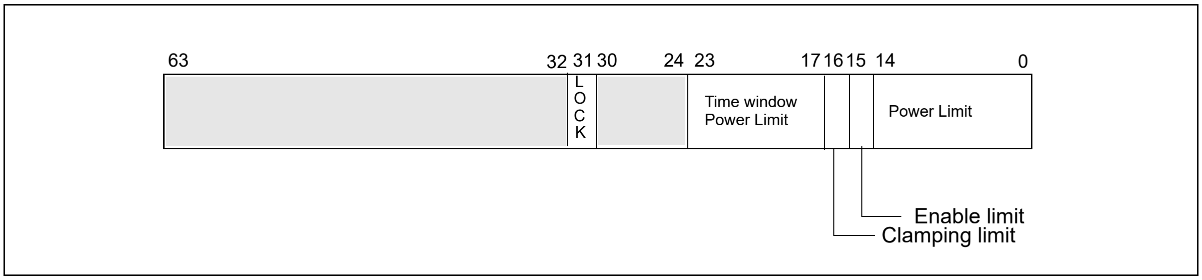


Figure 15-44. MSR_DRAM_POWER_LIMIT Register

MSR_DRAM_POWER_LIMIT allows a software agent to define power limitation for the DRAM domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_DRAM_POWER_LIMIT. A power limit can be specified along with a time window. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and un-modifiable until next RESET.

The bit fields of MSR_DRAM_POWER_LIMIT (Figure 15-44) are:

- **DRAM Power Limit #1**(bits 14:0): Sets the average power usage limit of the DRAM domain corresponding to time window # 1. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #1**(bit 15): 0 = disabled; 1 = enabled.
- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit will be used by the processor. The numeric value encoded by bits 23:17 is represented by the product of 2^Y *F; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Lock** (bit 31): If set, all write attempts to this MSR are ignored until next RESET.

MSR_DRAM_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the DRAM domain. This MSR is updated every ~1msec.

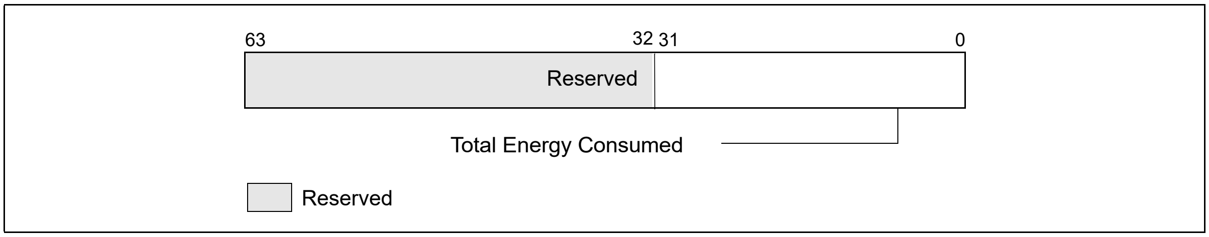


Figure 15-45. MSR_DRAM_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_POWER_INFO is a read-only MSR. It reports the DRAM power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the DRAM domain. It also provides the largest possible time window for software to program the RAPL interface.

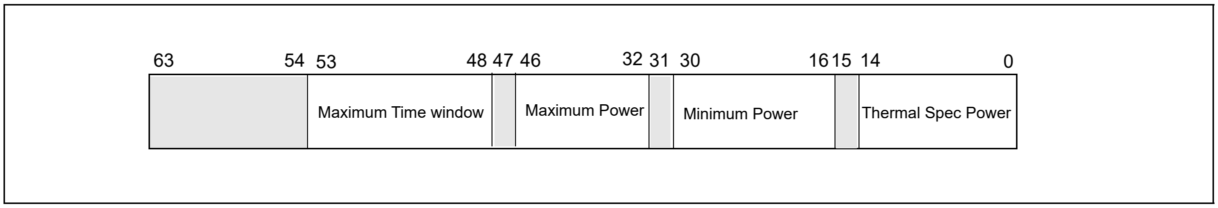


Figure 15-46. MSR_DRAM_POWER_INFO Register

- **Thermal Spec Power** (bits 14:0): The unsigned integer value is the equivalent of thermal specification power of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Time Window** (bits 53:48): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_DRAM_POWER_LIMIT. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific; see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

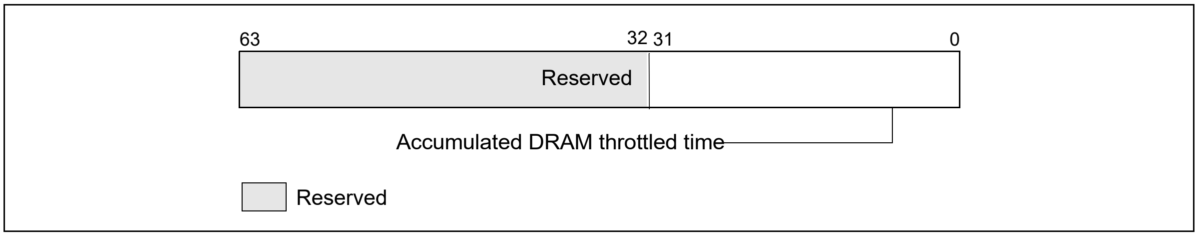


Figure 15-47. MSR_DRAM_PERF_STATUS MSR

- **Accumulated Package Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the DRAM domain has throttled. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

CHAPTER 16
MACHINE-CHECK ARCHITECTURE

This chapter describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. See Chapter 6, "Interrupt 18—Machine-Check Exception (#MC)," for more information on machine-check exceptions. A brief description of the Pentium processor's machine check capability is also given.

Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

16.1 MACHINE-CHECK ARCHITECTURE

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors implement a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as: system bus errors, ECC errors, parity errors, cache errors, and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs used for recording errors that are detected.

The processor signals the detection of an uncorrected machine-check error by generating a machine-check exception (#MC), which is an abort class exception. The implementation of the machine-check architecture does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check-exception handler can collect information about the machine-check error from the machine-check MSRs.

Starting with 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH; see the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. The processor can report information on corrected machine-check errors and deliver a programmable interrupt for software to respond to MC errors, referred to as corrected machine-check error interrupt (CMCI). See Section 16.5 for details.

Intel 64 processors supporting machine-check architecture and CMCI may also support an additional enhancement, namely, support for software recovery from certain uncorrected recoverable machine check errors. See Section 16.6 for details.

16.2 COMPATIBILITY WITH PENTIUM PROCESSOR

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support and extend the machine-check exception mechanism introduced in the Pentium processor. The Pentium processor reports the following machine-check errors:

- Data parity errors during read cycles.
- Unsuccessful completion of a bus cycle.

The above errors are reported using the P5_MC_TYPE and P5_MC_ADDR MSRs (implementation specific for the Pentium processor). Use the RDMSR instruction to read these MSRs. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for the addresses.

The machine-check error reporting mechanism that Pentium processors use is similar to that used in Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. When an error is detected, it is recorded in P5_MC_TYPE and P5_MC_ADDR; the processor then generates a machine-check exception (#MC).

See Section 16.3.3, "Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture," and Section 16.10.2, "Pentium Processor Machine-Check Exception Handling," for information on compatibility between machine-check code written to run on the Pentium processors and code written to run on P6 family processors.

16.3 MACHINE-CHECK MSRS

Machine check MSRs in the Pentium 4, Intel Atom, Intel Xeon, and P6 family processors consist of a set of global control and status registers and several error-reporting register banks. See Figure 16-1.

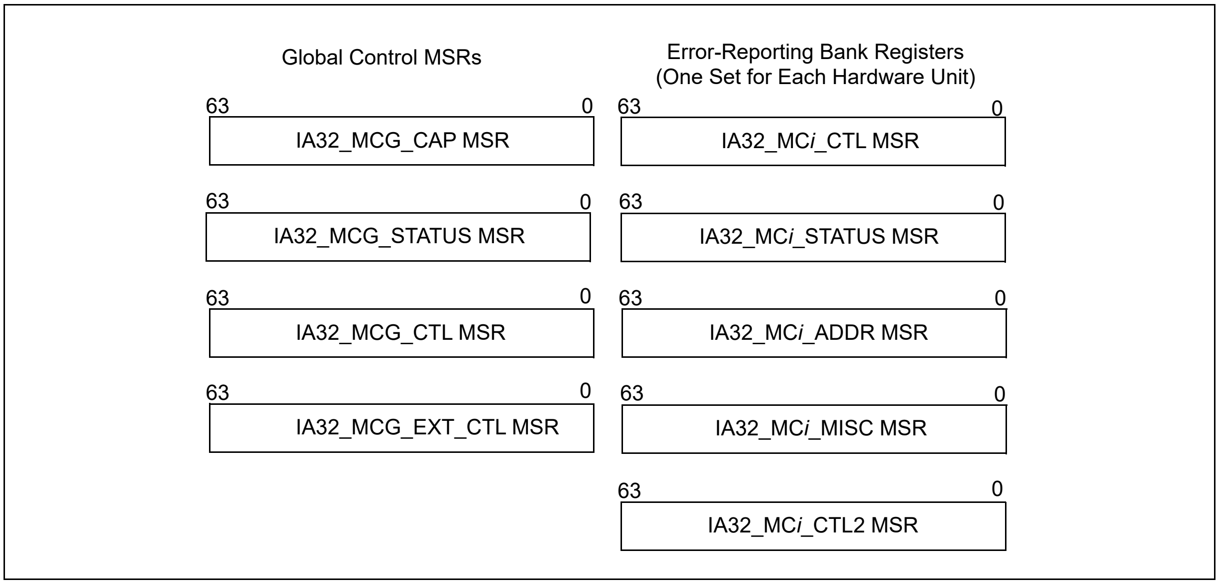


Figure 16-1. Machine-Check MSRs

Each error-reporting bank is associated with a specific hardware unit (or group of hardware units) in the processor. Use RDMSR and WRMSR to read and to write these registers.

16.3.1 Machine-Check Global Control MSRs

The machine-check global control MSRs include the IA32_MCG_CAP, IA32_MCG_STATUS, and optionally IA32_MCG_CTL and IA32_MCG_EXT_CTL. See Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for the addresses of these registers.

16.3.1.1 IA32_MCG_CAP MSR

The IA32_MCG_CAP MSR is a read-only register that provides information about the machine-check architecture of the processor. Figure 16-2 shows the layout of the register.

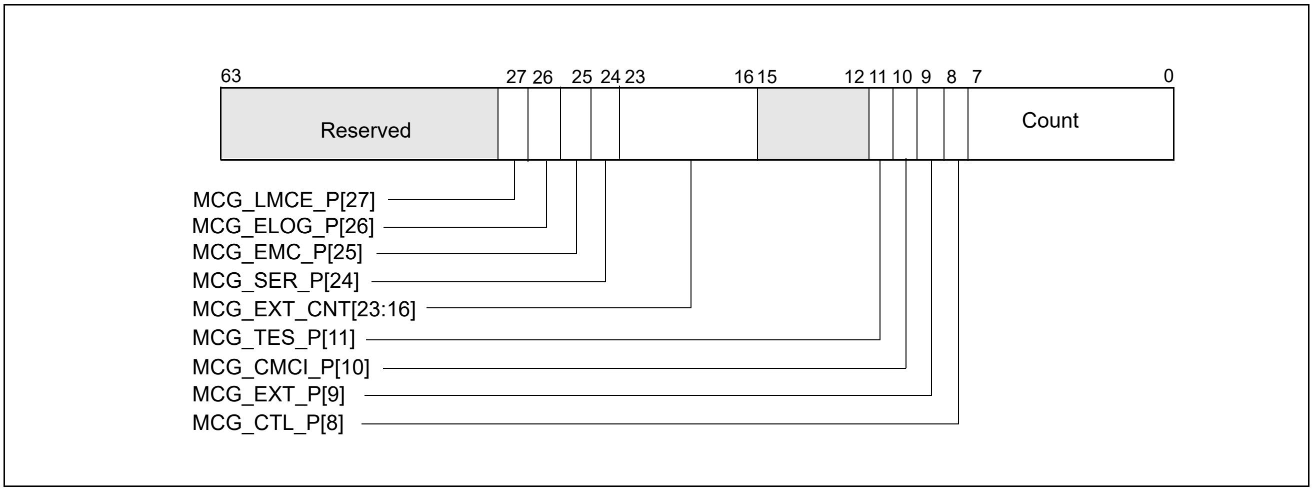


Figure 16-2. IA32_MCG_CAP Register

Where:

- **Count field, bits 7:0** — Indicates the number of hardware unit error-reporting banks available in a particular processor implementation.
- **MCG_CTL_P (control MSR present) flag, bit 8** — Indicates that the processor implements the IA32_MCG_CTL MSR when set; this register is absent when clear.
- **MCG_EXT_P (extended MSRs present) flag, bit 9** — Indicates that the processor implements the extended machine-check state registers found starting at MSR address 180H; these registers are absent when clear.
- **MCG_CMCI_P (Corrected MC error counting/signaling extension present) flag, bit 10** — Indicates (when set) that extended state and associated MSRs necessary to support the reporting of an interrupt on a corrected MC error event and/or count threshold of corrected MC errors, is present. When this bit is set, it does not imply this feature is supported across all banks. Software should check the availability of the necessary logic on a bank by bank basis when using this signaling capability (i.e., bit 30 settable in individual IA32_MCi_CTL2 register).
- **MCG_TES_P (threshold-based error status present) flag, bit 11** — Indicates (when set) that bits 56:53 of the IA32_MCi_STATUS MSR are part of the architectural space. Bits 56:55 are reserved, and bits 54:53 are used to report threshold-based error status. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific.
- **MCG_EXT_CNT, bits 23:16** — Indicates the number of extended machine-check state registers present. This field is meaningful only when the MCG_EXT_P flag is set.
- **MCG_SER_P (software error recovery support present) flag, bit 24** — Indicates (when set) that the processor supports software error recovery (see Section 16.6), and IA32_MCi_STATUS MSR bits 56:55 are used to report the signaling of uncorrected recoverable errors and whether software must take recovery actions for uncorrected errors. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific. If MCG_TES_P is set but MCG_SER_P is not set, bits 56:55 are reserved.
- **MCG_EMC_P (Enhanced Machine Check Capability) flag, bit 25** — Indicates (when set) that the processor supports enhanced machine check capabilities for firmware first signaling.
- **MCG_ELOG_P (extended error logging) flag, bit 26** — Indicates (when set) that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format “Generic Error Data Entry” that augments the data included in machine check bank registers.

  For additional information about extended error logging interface, see https://cdrdv2.intel.com/v1/dl/getContent/671064.
- **MCG_LMCE_P (local machine check exception) flag, bit 27** — Indicates (when set) that the following interfaces are present:

- — an extended state LMCE_S (located in bit 3 of IA32_MCG_STATUS), and
- — the IA32_MCG_EXT_CTL MSR, necessary to support Local Machine Check Exception (LMCE).

A non-zero MCG_LMCE_P indicates that, when LMCE is enabled as described in Section 16.3.1.5, some machine check errors may be delivered to only a single logical processor.

The effect of writing to the IA32_MCG_CAP MSR is undefined.

16.3.1.2 IA32_MCG_STATUS MSR

The IA32_MCG_STATUS MSR describes the current state of the processor after a machine-check exception has occurred (see Figure 16-3).

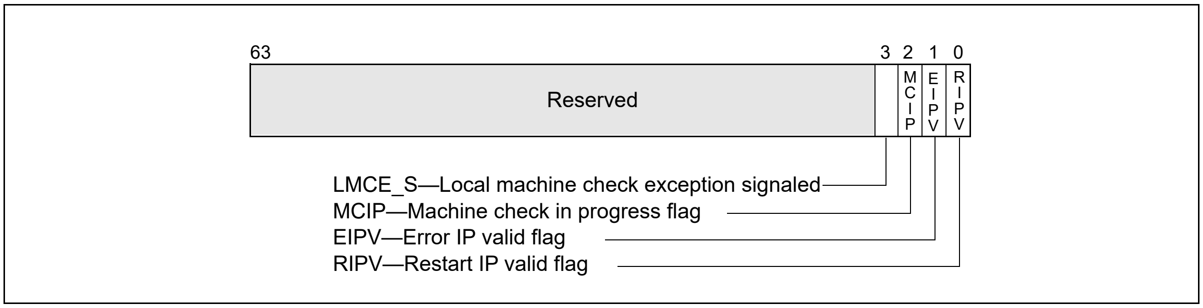

63
3
2
1
0
Reserved
MCIP
EIPV
RIPV
LMCE_S—Local machine check exception signaled
MCIP—Machine check in progress flag
EIPV—Error IP valid flag
RIPV—Restart IP valid flag

Figure 16-3. IA32_MCG_STATUS Register

Where:

- **RIPV (restart IP valid) flag, bit 0** — Indicates (when set) that program execution can be restarted reliably at the instruction pointed to by the instruction pointer pushed on the stack when the machine-check exception is generated. When clear, the program cannot be reliably restarted at the pushed instruction pointer.
- **EIPV (error IP valid) flag, bit 1** — Indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.
- **MCIP (machine check in progress) flag, bit 2** — Indicates (when set) that a machine-check exception was generated. Software can set or clear this flag. The occurrence of a second Machine-Check Event while MCIP is set will cause the processor to enter a shutdown state. For information on processor behavior in the shutdown state, please refer to the description in Chapter 6, “Interrupt and Exception Handling”: “Interrupt 8—Double Fault Exception (#DF)”.
- **LMCE_S (local machine check exception signaled), bit 3** — Indicates (when set) that a local machine-check exception was generated. This indicates that the current machine-check event was delivered to only this logical processor.

Bits 63:04 in the IA32_MCG_STATUS MSR are reserved. An attempt to write to the IA32_MCG_STATUS MSR’s reserved bits with any value other than 0 results in #GP.

16.3.1.3 IA32_MCG_CTL MSR

The IA32_MCG_CTL MSR is present if the capability flag MCG_CTL_P is set in the IA32_MCG_CAP MSR.

IA32_MCG_CTL controls the reporting of machine-check exceptions. If present, writing 1s to this register enables machine-check features and writing all 0s disables machine-check features. All other values are undefined and/or implementation specific.

16.3.1.4 IA32_MCG_EXT_CTL MSR

The IA32_MCG_EXT_CTL MSR is present if the capability flag MCG_LMCE_P is set in the IA32_MCG_CAP MSR.

IA32_MCG_EXT_CTL.LMCE_EN (bit 0) allows the processor to signal some MCEs to only a single logical processor in the system.

If MCG_LMCE_P is not set in IA32_MCG_CAP, or platform software has not enabled LMCE by setting IA32_FEATURE_CONTROL.LMCE_ENABLED (bit 20), any attempt to write or read IA32_MCG_EXT_CTL will result in #GP.

The IA32_MCG_EXT_CTL MSR is cleared on RESET.

Figure 16-4 shows the layout of the IA32_MCG_EXT_CTL register

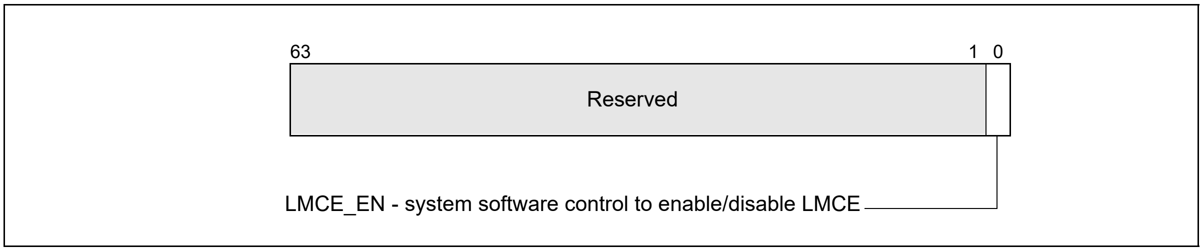


Figure 16-4. IA32_MCG_EXT_CTL Register

where

- **LMCE_EN (local machine check exception enable) flag, bit 0** - System software sets this to allow hardware to signal some MCEs to only a single logical processor. System software can set LMCE_EN only if the platform software has configured IA32_FEATURE_CONTROL as described in Section 16.3.1.5.

16.3.1.5 Enabling Local Machine Check

The intended usage of LMCE requires proper configuration by both platform software and system software. Platform software can turn LMCE on by setting bit 20 (LMCE_ENABLED) in IA32_FEATURE_CONTROL MSR (MSR address 3AH).

System software must ensure that both IA32_FEATURE_CONTROL.Lock (bit 0)and IA32_FEATURE_CONTROL.LMCE_ENABLED (bit 20) are set before attempting to set IA32_MCG_EXT_CTL.LMCE_EN (bit 0). When system software has enabled LMCE, then hardware will determine if a particular error can be delivered only to a single logical processor. Software should make no assumptions about the type of error that hardware can choose to deliver as LMCE. The severity and override rules stay the same as described in Table 16-8 to determine the recovery actions.

16.3.2 Error-Reporting Register Banks

Each error-reporting register bank can contain the IA32_MCi_CTL, IA32_MCi_STATUS, IA32_MCi_ADDR, and IA32_MCi_MISC MSRs. The number of reporting banks is indicated by bits [7:0] of IA32_MCG_CAP MSR (address 0179H). The first error-reporting register (IA32_MC0_CTL) always starts at address 400H.

See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for addresses of the error-reporting registers in the Pentium 4, Intel Atom, and Intel Xeon processors; and for addresses of the error-reporting registers P6 family processors.

16.3.2.1 IA32_MCi_CTL MSRs

The IA32_MC*i*_CTL MSR controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units). Each of the 64 flags (EE*j*) represents a potential error. Setting an EE*j* flag enables signaling #MC of the associated error and clearing it disables signaling of the error. Error logging happens regardless of the setting of these bits. The processor drops writes to bits that are not implemented. Figure 16-5 shows the bit fields of IA32_MC*i*_CTL.

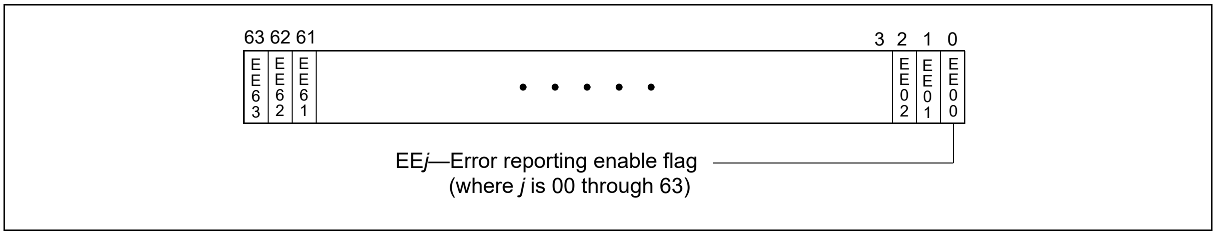

63 62 61
3 2 1 0
EE63
EE62
EE61
EE02
EE01
EE00
. . . . .
EE*j*—Error reporting enable flag
(where *j* is 00 through 63)

Figure 16-5. IA32_MC*i*_CTL Register

NOTE

For P6 family processors, processors based on Intel Core microarchitecture (excluding those on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH and onward): the operating system or executive software must not modify the contents of the IA32_MC0_CTL MSR. This MSR is internally aliased to the EBL_CR_POWERON MSR and controls platform-specific error handling features. System specific firmware (the BIOS) is responsible for the appropriate initialization of the IA32_MC0_CTL MSR. P6 family processors only allow the writing of all 1s or all 0s to the IA32_MC*i*_CTL MSR.

16.3.2.2 IA32_MCi_STATUS MSRS

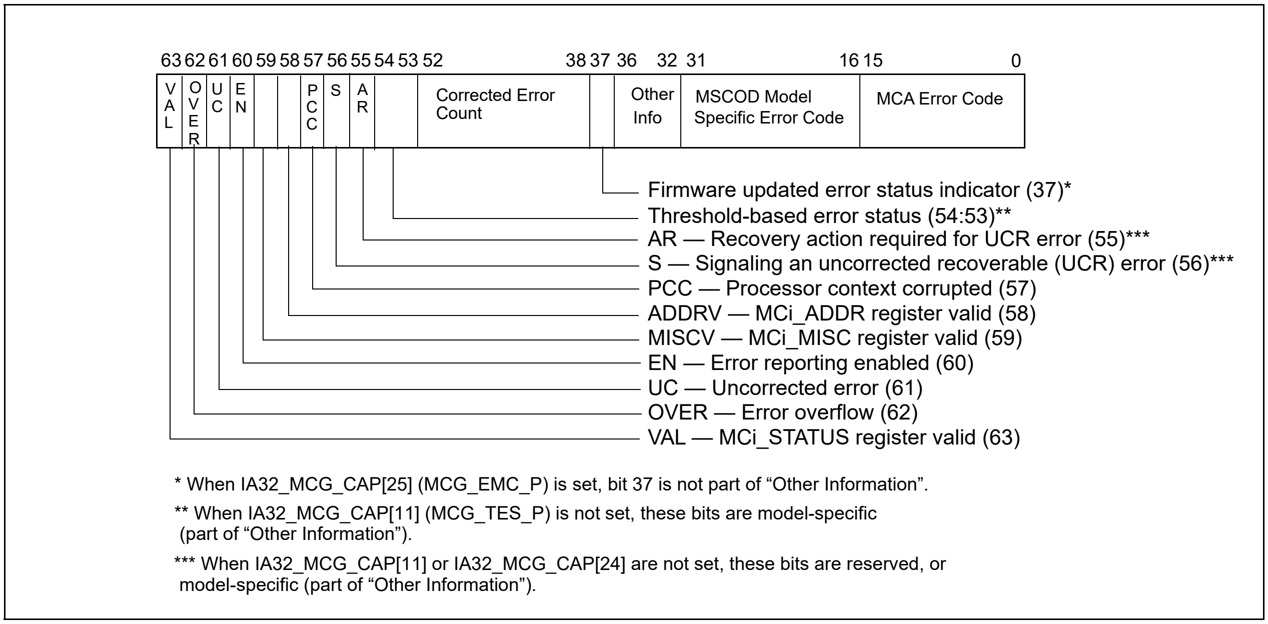
Each IA32_MCi_STATUS MSR contains information related to a machine-check error if its VAL (valid) flag is set (see Figure 16-6). Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.

NOTE

Figure 16-6 depicts the IA32_MCi_STATUS MSR when IA32_MCG_CAP[24] = 1, IA32_MCG_CAP[11] = 1 and IA32_MCG_CAP[10] = 1. When IA32_MCG_CAP[24] = 0 and IA32_MCG_CAP[11] = 1, bits 56:55 is reserved and bits 54:53 for threshold-based error reporting. When IA32_MCG_CAP[11] = 0, bits 56:53 are part of the "Other Information" field. The use of bits 54:53 for threshold-based error reporting began with Intel Core Duo processors, and is currently used for cache memory. See Section 16.4, "Enhanced Cache Error reporting," for more information. When IA32_MCG_CAP[10] = 0, bits 52:38 are part of the "Other Information" field. The use of bits 52:38 for corrected MC error count is introduced with Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH.

Where:

- **MCA (machine-check architecture) error code field, bits 15:0** — Specifies the machine-check architecture-defined error code for the machine-check error condition detected. The machine-check architecture-defined error codes are guaranteed to be the same for all IA-32 processors that implement the machine-check architecture. See Section 16.9, "Interpreting the MCA Error Codes," and Chapter 17, "Interpreting Machine Check Error Codes," for information on machine-check error codes.
- **Model-specific error code field, bits 31:16** — Specifies the model-specific error code that uniquely identifies the machine-check error condition detected. The model-specific error codes may differ among IA-32 processors for the same machine-check error condition. See Chapter 17, "Interpreting Machine Check Error Codes," for information on model-specific error codes.
- **Reserved, Error Status, and Other Information fields, bits 56:32** —
    - If IA32_MCG_CAP.MCG_EMC_P[bit 25] is 0, bits 37:32 contain "Other Information" that is implementation-specific and is not part of the machine-check architecture.
    - If IA32_MCG_CAP.MCG_EMC_P is 1, "Other Information" is in bits 36:32. If bit 37 is 0, system firmware has not changed the contents of IA32_MCi_STATUS. If bit 37 is 1, system firmware may have edited the contents of IA32_MCi_STATUS.
    - If IA32_MCG_CAP.MCG_CMCI_P[bit 10] is 0, bits 52:38 also contain "Other Information" (in the same sense as bits 37:32).


63 62 61 60 59 58 57 56 55 54 53 52
38 37 36
32 31
16 15
0
VAL
OVER
UC
EN
PCC
S
AR
Corrected Error Count
Other Info
MSCOD Model Specific Error Code
MCA Error Code
Firmware updated error status indicator (37)*
Threshold-based error status (54:53)**
AR — Recovery action required for UCR error (55)***
S — Signaling an uncorrected recoverable (UCR) error (56)***
PCC — Processor context corrupted (57)
ADDRV — MCi_ADDR register valid (58)
MISCV — MCi_MISC register valid (59)
EN — Error reporting enabled (60)
UC — Uncorrected error (61)
OVER — Error overflow (62)
VAL — MCi_STATUS register valid (63)

* When IA32_MCG_CAP[25] (MCG_EMC_P) is set, bit 37 is not part of “Other Information”.

** When IA32_MCG_CAP[11] (MCG_TES_P) is not set, these bits are model-specific (part of “Other Information”).

*** When IA32_MCG_CAP[11] or IA32_MCG_CAP[24] are not set, these bits are reserved, or model-specific (part of “Other Information”).

Figure 16-6. IA32_MC*i*_STATUS Register

- If IA32_MCG_CAP[10] is 1, bits 52:38 are architectural (not model-specific). In this case, bits 52:38 reports the value of a 15 bit counter that increments each time a corrected error is observed by the MCA recording bank. This count value will continue to increment until cleared by software. The most significant bit, 52, is a sticky count overflow bit.
- If IA32_MCG_CAP[11] is 0, bits 56:53 also contain “Other Information” (in the same sense).
- If IA32_MCG_CAP[11] is 1, bits 56:53 are architectural (not model-specific). In this case, bits 56:53 have the following functionality:
  - If IA32_MCG_CAP[24] is 0, bits 56:55 are reserved.
  - If IA32_MCG_CAP[24] is 1, bits 56:55 are defined as follows:
  - S (Signaling) flag, bit 56 - Signals the reporting of UCR errors in this MC bank. See Section 16.6.2 for additional details.
  - AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. See Section 16.6.2 for additional details.
  - If the UC bit (Figure 16-6) is 1, bits 54:53 are undefined.
  - If the UC bit (Figure 16-6) is 0, bits 54:53 indicate the status of the hardware structure that reported the threshold-based error. See Table 16-1.

Table 16-1. Bits 54:53 in IA32_MCi_STATUS MSRs when IA32_MCG_CAP[11] = 1 and UC = 0

Bits 54:53	Meaning
00	**No tracking** - No hardware status tracking is provided for the structure reporting this event.
01	**Green** - Status tracking is provided for the structure posting the event; the current status is green (below threshold). For more information, see Section 16.4, “Enhanced Cache Error reporting.”
10	**Yellow** - Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). For more information, see Section 16.4, “Enhanced Cache Error reporting.”
11	Reserved

- **PCC (processor context corrupt) flag, bit 57** — Indicates (when set) that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor's state, and software may be able to restart. When system software supports recovery, consult Section 16.10.4, "Machine-Check Software Handler Guidelines for Error Recovery," for additional rules that apply.
- **ADDRV (IA32_MC*i*_ADDR register valid) flag, bit 58** — Indicates (when set) that the IA32_MCi_ADDR register contains the address where the error occurred (see Section 16.3.2.3, "IA32_MCi_ADDR MSRs"). When clear, this flag indicates that the IA32_MCi_ADDR register is either not implemented or does not contain the address where the error occurred. Do not read these registers if they are not implemented in the processor.
- **MISCV (IA32_MC*i*_MISC register valid) flag, bit 59** — Indicates (when set) that the IA32_MC*i*_MISC register contains additional information regarding the error. When clear, this flag indicates that the IA32_MC*i*_MISC register is either not implemented or does not contain additional information regarding the error. Do not read these registers if they are not implemented in the processor.
- **EN (error enabled) flag, bit 60** — Indicates (when set) that the error was enabled by the associated EEj bit of the IA32_MC*i*_CTL register.
- **UC (error uncorrected) flag, bit 61** — Indicates (when set) that the processor did not or was not able to correct the error condition. When clear, this flag indicates that the processor was able to correct the error condition.
- **OVER (machine check overflow) flag, bit 62** — Indicates (when set) that a machine-check error occurred while the results of a previous error were still in the error-reporting register bank (that is, the VAL bit was already set in the IA32_MC*i*_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. In general, enabled errors are written over disabled errors, and uncorrected errors are written over corrected errors. Uncorrected errors are not written over previous valid uncorrected errors. When MCG_CMCI_P is set, corrected errors may not set the OVER flag. Software can rely on corrected error count in IA32_MCi_Status[52:38] to determine if any additional corrected errors may have occurred. For more information, see Section 16.3.2.2.1, "Overwrite Rules for Machine Check Overflow."
- **VAL (IA32_MC*i*_STATUS register valid) flag, bit 63** — Indicates (when set) that the information within the IA32_MCi_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the IA32_MCi_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

16.3.2.2.1 Overwrite Rules for Machine Check Overflow

Table 16-2 shows the overwrite rules for how to treat a second event if the MC bank already contains a valid log from an earlier event – that is, what to do if the valid bit for an MC bank already is set to 1. When more than one structure posts events in a given bank, these rules specify whether a new event will overwrite a previous posting or not. These rules define a priority for uncorrected (highest priority), yellow, and green/unmonitored (lowest priority) status.

In Table 16-2, the values in the two left-most columns are IA32_MCi_STATUS[54:53].

Table 16-2. Overwrite Rules for Enabled Errors

First Event	Second Event	UC bit	Color	MCA Info
00/green	00/green	0	00/green	either
00/green	yellow	0	yellow	second error
yellow	00/green	0	yellow	first error
yellow	yellow	0	yellow	either
00/green/yellow	UC	1	undefined	second
UC	00/green/yellow	1	undefined	first

If a second event overwrites a previously posted event, the information (as guarded by individual valid bits) in the MCi bank is entirely from the second event. Similarly, if a first event is retained, all of the information previously posted for that event is retained. In general, when the logged error or the recent error is a corrected error, the OVER bit (MCi_Status[62]) may be set to indicate an overflow. When MCG_CMCI_P is set in IA32_MCG_CAP, system software should consult IA32_MCi_STATUS[52:38] to determine if additional corrected errors may have occurred. Software may re-read IA32_MCi_STATUS, IA32_MCi_ADDR, and IA32_MCi_MISC appropriately to ensure data collected represent the last error logged.

After software polls a posting and clears the register, the valid bit is no longer set and therefore the meaning of the rest of the bits, including the yellow/green/00 status field in bits 54:53, is undefined. The yellow/green indication will only be posted for events associated with monitored structures – otherwise the unmonitored (00) code will be posted in IA32_MC*i*_STATUS[54:53].

16.3.2.3 IA32_MCi_ADDR MSRs

The IA32_MC*i*_ADDR MSR contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MC*i*_STATUS register is set (see Section 16-7, "IA32_MCi_ADDR MSR"). The IA32_MCi_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_M-Ci_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception.

The address returned is an offset into a segment, linear address, or physical address. This depends on the error encountered. When these registers are implemented, these registers can be cleared by explicitly writing 0s to these registers. Writing 1s to these registers will cause a general-protection exception. See Figure 16-7.

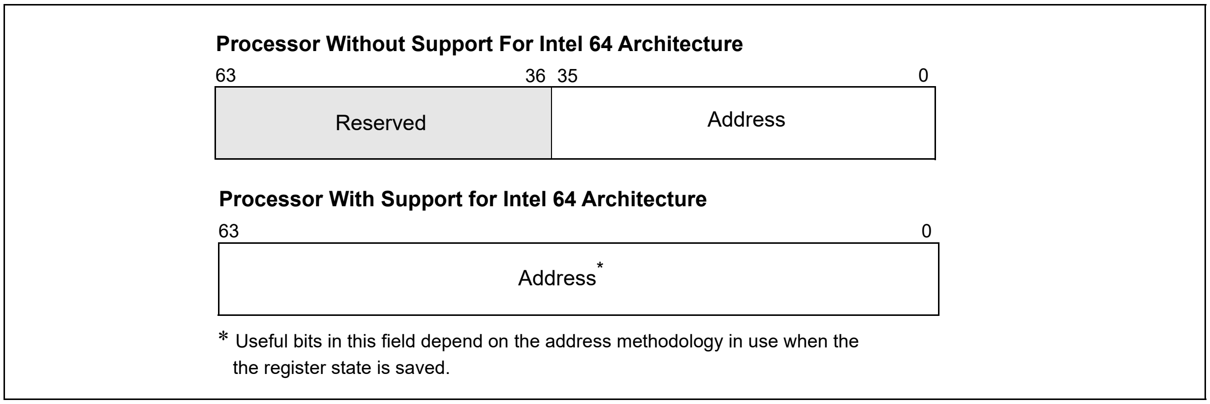

Processor Without Support For Intel 64 Architecture
63
36
35
0
Reserved
Address
Processor With Support for Intel 64 Architecture
63
0
Address*
* Useful bits in this field depend on the address methodology in use when the
the register state is saved.

Figure 16-7. IA32_MC*i*_ADDR MSR

16.3.2.4 IA32_MCi_MISC MSRs

The IA32_MC*i*_MISC MSR contains additional information describing the machine-check error if the MISCV flag in the IA32_MC*i*_STATUS register is set. The IA32_MCi_MISC_MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MCi_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception. When implemented in a processor, these registers can be cleared by explicitly writing all 0s to them; writing 1s to them causes a general-protection exception to be generated. This register is not implemented in any of the error-reporting register banks for the P6 or Intel Atom family processors.

If both MISCV and IA32_MCG_CAP[24] are set, the IA32_MCi_MISC_MSR is defined according to Figure 16-8 to support software recovery of uncorrected errors (see Section 16.6).

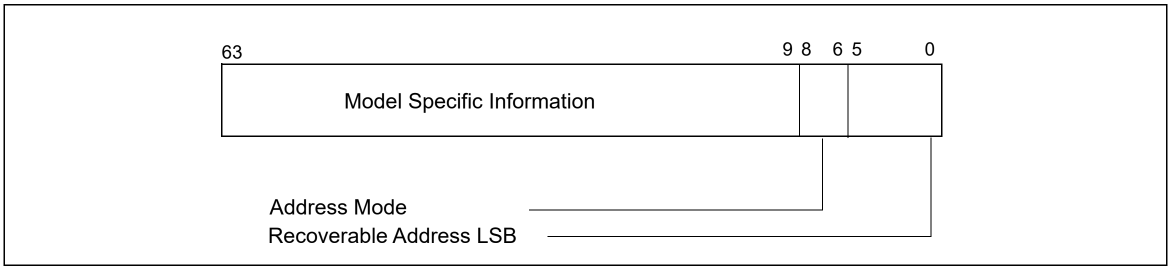


Figure 16-8. UCR Support in IA32_MCi_MISC Register

- Recoverable Address LSB (bits 5:0): The lowest valid recoverable address bit. Indicates the position of the least significant bit (LSB) of the recoverable error address. For example, if the processor logs bits [43:9] of the address, the LSB sub-field in IA32_MCi_MISC is 01001b (9 decimal). For this example, bits [8:0] of the recoverable error address in IA32_MCi_ADDR should be ignored.
- Address Mode (bits 8:6): Address mode for the address logged in IA32_MCi_ADDR. The supported address modes are given in Table 16-3.

Table 16-3. Address Mode in IA32_MCi_MISC[8:6]

IA32_MCi_MISC[8:6] Encoding	Definition
000	Segment Offset
001	Linear Address
010	Physical Address
011	Memory Address
100 to 110	Reserved
111	Generic

- Model Specific Information (bits 63:9): Not architecturally defined.

16.3.2.4.2 IOMCA

Logging and Signaling of errors from PCI Express domain is governed by PCI Express Advanced Error Reporting (AER) architecture. PCI Express architecture divides errors in two categories: Uncorrectable errors and Correctable errors. Uncorrectable errors can further be classified as Fatal or Non-Fatal. Uncorrected IO errors are signaled to the system software either as AER Message Signaled Interrupt (MSI) or via platform specific mechanisms such as NMI. Generally, the signaling mechanism is controlled by BIOS and/or platform firmware. Certain processors support an error handling mode, called IOMCA mode, where Uncorrected PCI Express errors are signaled in the form of machine check exception and logged in machine check banks.

When a processor is in this mode, Uncorrected PCI Express errors are logged in the MCACOD field of the IA32_MCi_STATUS register as Generic I/O error. The corresponding MCA error code is defined in Table 15-8. IA32_MCi_Status [15:0] Simple Error Code Encoding. Machine check logging complements and does not replace AER logging that occurs inside the PCI Express hierarchy. The PCI Express Root Complex and Endpoints continue to log the error in accordance with PCI Express AER mechanism. In IOMCA mode, MCi_MISC register in the bank that logged IOMCA can optionally contain information that link the Machine Check logs with the AER logs or proprietary logs. In such a scenario, the machine check handler can utilize the contents of MCi_MISC to locate the next level of error logs corresponding to the same error. Specifically, if MCi_Status.MISCV is 1 and MCACOD is 0x0E0B, MCi_MISC contains the PCI Express address of the Root Complex device containing the AER Logs. Software can consult the header type and class code registers in the Root Complex device's PCIe Configuration space to determine what type of device it is. This Root Complex device can either be a PCI Express Root Port, PCI Express Root Complex Event Collector or a proprietary device.

Errors that originate from PCI Express or Legacy Endpoints are logged in the corresponding Root Port in addition to the generating device. If MISCV=1 and MCi_MISC contains the address of the Root Port or a Root Complex Event collector, software can parse the AER logs to learn more about the error.

If MISCV=1 and MCi_MISC points to a device that is neither a Root Complex Event Collector not a Root Port, software must consult the Vendor ID/Device ID and use device specific knowledge to locate and interpret the error log registers. In some cases, the Root Complex device configuration space may not be accessible to the software and both the Vendor and Device ID read as 0xFFFF.

- The format of MCi_MISC for IOMCA errors is shown in Table 16-4.

Table 16-4. Address Mode in IA32_MCi_MISC[8:6]

63:40	39:32	31:16	15:9	8:6	5:0
RSVD	PCI Express Segment number	PCI Express Requestor ID	RSVD	ADDR MODE[1]	RECOV ADDR LSB[1]

NOTES:

1. Not Applicable if ADDRV=0.

Refer to PCI Express Specification 3.0 for definition of PCI Express Requestor ID and AER architecture. Refer to PCI Firmware Specification 3.0 for an explanation of PCI Ex-press Segment number and how software can access configuration space of a PCI Ex-press device given the segment number and Requestor ID.

16.3.2.5 IA32_MCi_CTL2 MSRs

The IA32_MC*i*_CTL2 MSR provides the programming interface to use corrected MC error signaling capability that is indicated by IA32_MCG_CAP[10] = 1. Software must check for the presence of IA32_MC*i*_CTL2 on a per-bank basis.

When IA32_MCG_CAP[10] = 1, the IA32_MCi_CTL2 MSR for each bank exists, i.e., reads and writes to these MSR are supported. However, signaling interface for corrected MC errors may not be supported in all banks.

The layout of IA32_MC*i*_CTL2 is shown in Figure 16-9.

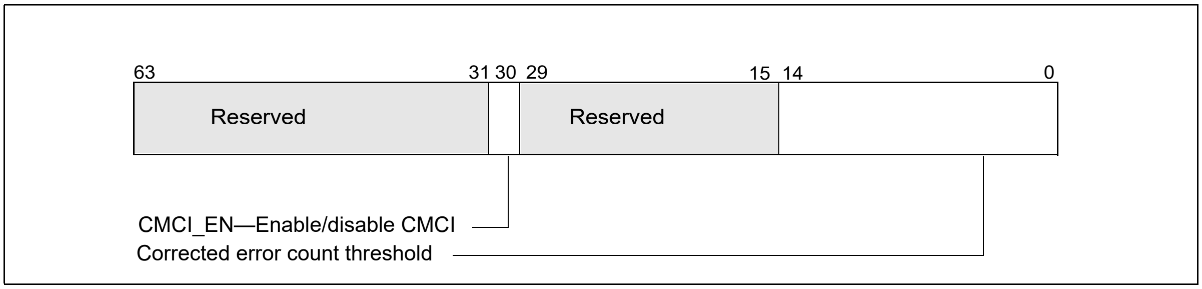


Figure 16-9. IA32_MCi_CTL2 Register

- **Corrected error count threshold, bits 14:0** — Software must initialize this field. The value is compared with the corrected error count field in IA32_MCi_STATUS, bits 38 through 52. An overflow event is signaled to the CMCI LVT entry (see Table 11-1) in the APIC when the count value equals the threshold value. The new LVT entry in the APIC is at 02F0H offset from the APIC_BASE. If CMCI interface is not supported for a particular bank (but IA32_MCG_CAP[10] = 1), this field will always read 0.
- **CMCI_EN (Corrected error interrupt enable/disable/indicator), bits 30** — Software sets this bit to enable the generation of corrected machine-check error interrupt (CMCI). If CMCI interface is not supported for a particular bank (but IA32_MCG_CAP[10] = 1), this bit is writeable but will always return 0 for that bank. This bit also indicates CMCI is supported or not supported in the corresponding bank. See Section 16.5 for details of software detection of CMCI facility.

Some microarchitectural sub-systems that are the source of corrected MC errors may be shared by more than one logical processors. Consequently, the facilities for reporting MC errors and controlling mechanisms may be shared by more than one logical processors. For example, the IA32_MC*i*_CTL2 MSR is shared between logical processors sharing a processor core. Software is responsible to program IA32_MC*i*_CTL2 MSR in a consistent manner with CMCI delivery and usage.

After processor reset, IA32_MC*i*_CTL2 MSRs are zeroed.

16.3.2.6 IA32_MCG Extended Machine Check State MSRs

The Pentium 4 and Intel Xeon processors implement a variable number of extended machine-check state MSRs. The MCG_EXT_P flag in the IA32_MCG_CAP MSR indicates the presence of these extended registers, and the MCG_EXT_CNT field indicates the number of these registers actually implemented. See Section 16.3.1.1, “IA32_MCG_CAP MSR.” Also see Table 16-5.

Table 16-5. Extended Machine Check State MSRs in Processors Without Support for Intel® 64 Architecture

MSR	Address	Description
IA32_MCG_EAX	180H	Contains state of the EAX register at the time of the machine-check error.
IA32_MCG_EBX	181H	Contains state of the EBX register at the time of the machine-check error.
IA32_MCG_ECX	182H	Contains state of the ECX register at the time of the machine-check error.
IA32_MCG_EDX	183H	Contains state of the EDX register at the time of the machine-check error.
IA32_MCG_ESI	184H	Contains state of the ESI register at the time of the machine-check error.
IA32_MCG_EDI	185H	Contains state of the EDI register at the time of the machine-check error.
IA32_MCG_EBP	186H	Contains state of the EBP register at the time of the machine-check error.
IA32_MCG_ESP	187H	Contains state of the ESP register at the time of the machine-check error.
IA32_MCG_EFLAGS	188H	Contains state of the EFLAGS register at the time of the machine-check error.
IA32_MCG_EIP	189H	Contains state of the EIP register at the time of the machine-check error.
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.

In processors with support for Intel 64 architecture, 64-bit machine check state MSRs are aliased to the legacy MSRs. In addition, there may be registers beyond IA32_MCG_MISC. These may include up to five reserved MSRs (IA32_MCG_RESERVED[1:5]) and save-state MSRs for registers introduced in 64-bit mode. See Table 16-6.

Table 16-6. Extended Machine Check State MSRs In Processors With Support for Intel® 64 Architecture

MSR	Address	Description
IA32_MCG_RAX	180H	Contains state of the RAX register at the time of the machine-check error.
IA32_MCG_RBX	181H	Contains state of the RBX register at the time of the machine-check error.
IA32_MCG_RCX	182H	Contains state of the RCX register at the time of the machine-check error.
IA32_MCG_RDX	183H	Contains state of the RDX register at the time of the machine-check error.
IA32_MCG_RSI	184H	Contains state of the RSI register at the time of the machine-check error.
IA32_MCG_RDI	185H	Contains state of the RDI register at the time of the machine-check error.
IA32_MCG_RBP	186H	Contains state of the RBP register at the time of the machine-check error.
IA32_MCG_RSP	187H	Contains state of the RSP register at the time of the machine-check error.
IA32_MCG_RFLAGS	188H	Contains state of the RFLAGS register at the time of the machine-check error.
IA32_MCG_RIP	189H	Contains state of the RIP register at the time of the machine-check error.
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.

Table 16-6. Extended Machine Check State MSRs In Processors With Support for Intel® 64 Architecture (Contd.)

MSR	Address	Description
IA32_MCG_ RSERVED[1:5]	18BH- 18FH	These registers, if present, are reserved.
IA32_MCG_R8	190H	Contains state of the R8 register at the time of the machine-check error.
IA32_MCG_R9	191H	Contains state of the R9 register at the time of the machine-check error.
IA32_MCG_R10	192H	Contains state of the R10 register at the time of the machine-check error.
IA32_MCG_R11	193H	Contains state of the R11 register at the time of the machine-check error.
IA32_MCG_R12	194H	Contains state of the R12 register at the time of the machine-check error.
IA32_MCG_R13	195H	Contains state of the R13 register at the time of the machine-check error.
IA32_MCG_R14	196H	Contains state of the R14 register at the time of the machine-check error.
IA32_MCG_R15	197H	Contains state of the R15 register at the time of the machine-check error.

When a machine-check error is detected on a Pentium 4 or Intel Xeon processor, the processor saves the state of the general-purpose registers, the R/EFLAGS register, and the R/EIP in these extended machine-check state MSRs. This information can be used by a debugger to analyze the error.

These registers are read/write to zero registers. This means software can read them; but if software writes to them, only all zeros is allowed. If software attempts to write a non-zero value into one of these registers, a general-protection (#GP) exception is generated. These registers are cleared on a hardware reset (power-up or RESET), but maintain their contents following a soft reset (INIT reset).

16.3.3 Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture

The Pentium processor reports machine-check errors using two registers: P5_MC_TYPE and P5_MC_ADDR. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors map these registers to the IA32_MC*i*_STATUS and IA32_MC*i*_ADDR in the error-reporting register bank. This bank reports on the same type of external bus errors reported in P5_MC_TYPE and P5_MC_ADDR.

The information in these registers can then be accessed in two ways:

- By reading the IA32_MC*i*_STATUS and IA32_MC*i*_ADDR registers as part of a general machine-check exception handler written for Pentium 4, Intel Atom and P6 family processors.
- By reading the P5_MC_TYPE and P5_MC_ADDR registers using the RDMSR instruction.

The second capability permits a machine-check exception handler written to run on a Pentium processor to be run on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor. There is a limitation in that information returned by the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors is encoded differently than information returned by the Pentium processor. To run a Pentium processor machine-check exception handler on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor; the handler must be written to interpret P5_MC_TYPE encodings correctly.

16.4 ENHANCED CACHE ERROR REPORTING

Starting with Intel Core Duo processors, cache error reporting was enhanced. In earlier Intel processors, cache status was based on the number of correction events that occurred in a cache. In the new paradigm, called “threshold-based error status”, cache status is based on the number of lines (ECC blocks) in a cache that incur repeated corrections. The threshold is chosen by Intel, based on various factors. If a processor supports threshold-based error status, it sets IA32_MCG_CAP[11] (MCG_TES_P) to 1; if not, to 0.

A processor that supports enhanced cache error reporting contains hardware that tracks the operating status of certain caches and provides an indicator of their “health”. The hardware reports a “green” status when the number of lines that incur repeated corrections is at or below a pre-defined threshold, and a “yellow” status when the number of affected lines exceeds the threshold. Yellow status means that the cache reporting the event is operating correctly, but you should schedule the system for servicing within a few weeks.

Intel recommends that you rely on this mechanism for structures supported by threshold-base error reporting.

The CPU/system/platform response to a yellow event should be less severe than its response to an uncorrected error. An uncorrected error means that a serious error has actually occurred, whereas the yellow condition is a warning that the number of affected lines has exceeded the threshold but is not, in itself, a serious event: the error was corrected and system state was not compromised.

The green/yellow status indicator is not a foolproof early warning for an uncorrected error resulting from the failure of two bits in the same ECC block. Such a failure can occur and cause an uncorrected error before the yellow threshold is reached. However, the chance of an uncorrected error increases as the number of affected lines increases.

16.5 CORRECTED MACHINE CHECK ERROR INTERRUPT

Corrected machine-check error interrupt (CMCI) is an architectural enhancement to the machine-check architecture. It provides capabilities beyond those of threshold-based error reporting (Section 16.4). With threshold-based error reporting, software is limited to use periodic polling to query the status of hardware corrected MC errors. CMCI provides a signaling mechanism to deliver a local interrupt based on threshold values that software can program using the IA32_MCi_CTL2 MSRs.

CMCI is disabled by default. System software is required to enable CMCI for each IA32_MCi bank that support the reporting of hardware corrected errors if IA32_MCG_CAP[10] = 1.

System software use IA32_MCi_CTL2 MSR to enable/disable the CMCI capability for each bank and program threshold values into IA32_MCi_CTL2 MSR. CMCI is not affected by the CR4.MCE bit, and it is not affected by the IA32_MCi_CTL MSRs.

To detect the existence of thresholding for a given bank, software writes only bits 14:0 with the threshold value. If the bits persist, then thresholding is available (and CMCI is available). If the bits are all 0's, then no thresholding exists. To detect that CMCI signaling exists, software writes a 1 to bit 30 of the MCi_CTL2 register. Upon subsequent read, if bit 30 = 0, no CMCI is available for this bank and no corrected or UCNA errors will be reported on this bank. If bit 30 = 1, then CMCI is available and enabled.

16.5.1 CMCI Local APIC Interface

The operation of CMCI is depicted in Figure 16-10.

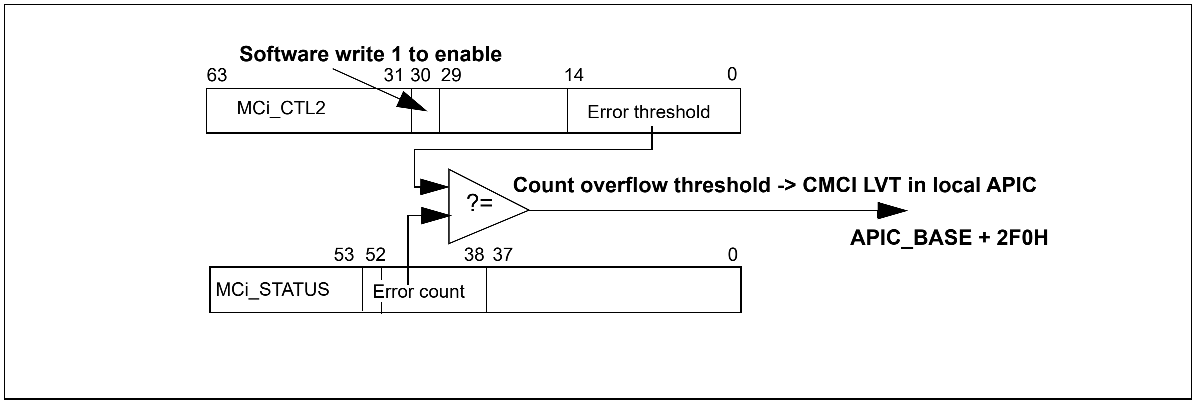


Figure 16-10. CMCI Behavior

CMCI interrupt delivery is configured by writing to the LVT CMCI register entry in the local APIC register space at default address of APIC_BASE + 2F0H. A CMCI interrupt can be delivered to more than one logical processors if multiple logical processors are affected by the associated MC errors. For example, if a corrected bit error in a cache shared by two logical processors caused a CMCI, the interrupt will be delivered to both logical processors sharing that microarchitectural sub-system. Similarly, package level errors may cause CMCI to be delivered to all logical processors within the package. However, system level errors will not be handled by CMCI.

See Section 11.5.1, “Local Vector Table,” for details regarding the LVT CMCI register.

16.5.2 System Software Recommendation for Managing CMCI and Machine Check Resources

System software must enable and manage CMCI, set up interrupt handlers to service CMCI interrupts delivered to affected logical processors, program CMCI LVT entry, and query machine check banks that are shared by more than one logical processors.

This section describes techniques system software can implement to manage CMCI initialization, service CMCI interrupts in a efficient manner to minimize contentions to access shared MSR resources.

16.5.2.1 CMCI Initialization

Although a CMCI interrupt may be delivered to more than one logical processors depending on the nature of the corrected MC error, only one instance of the interrupt service routine needs to perform the necessary service and make queries to the machine-check banks. The following steps describes a technique that limits the amount of work the system has to do in response to a CMCI.

- To provide maximum flexibility, system software should define per-thread data structure for each logical processor to allow equal-opportunity and efficient response to interrupt delivery. Specifically, the per-thread data structure should include a set of per-bank fields to track which machine check bank it needs to access in response to a delivered CMCI interrupt. The number of banks that needs to be tracked is determined by IA32_MCG_CAP[7:0].
- Initialization of per-thread data structure. The initialization of per-thread data structure must be done serially on each logical processor in the system. The sequencing order to start the per-thread initialization between different logical processor is arbitrary. But it must observe the following specific detail to satisfy the shared nature of specific MSR resources:
  a. Each thread initializes its data structure to indicate that it does not own any MC bank registers.
  b. Each thread examines IA32_MCi_CTL2[30] indicator for each bank to determine if another thread has already claimed ownership of that bank.
     - If IA32_MCi_CTL2[30] had been set by another thread. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
     - If IA32_MCi_CTL2[30] = 0, proceed to step c.
  c. Check whether writing a 1 into IA32_MCi_CTL2[30] can return with 1 on a subsequent read to determine this bank can support CMCI.
     - If IA32_MCi_CTL2[30] = 0, this bank does not support CMCI. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
     - If IA32_MCi_CTL2[30] = 1, modify the per-thread data structure to indicate this thread claims ownership to the MC bank; proceed to initialize the error threshold count (bits 15:0) of that bank as described in Chapter 16, “CMCI Threshold Management”. Then proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
- After the thread has examined all of the machine check banks, it sees if it owns any MC banks to service CMCI. If any bank has been claimed by this thread:
  — Ensure that the CMCI interrupt handler has been set up as described in Chapter 16, “CMCI Interrupt Handler”.
  — Initialize the CMCI LVT entry, as described in Section 16.5.1, “CMCI Local APIC Interface.”
  — Log and clear all of IA32_MCi_Status registers for the banks that this thread owns. This will allow new errors to be logged.

16.5.2.2 CMCI Threshold Management

The Corrected MC error threshold field, IA32_MCi_CTL2[14:0], is architecturally defined. Specifically, all these bits are writable by software, but different processor implementations may choose to implement less than 15 bits as threshold for the overflow comparison with IA32_MCi_STATUS[52:38]. The following describes techniques that software can manage CMCI threshold to be compatible with changes in implementation characteristics:

- Software can set the initial threshold value to 1 by writing 1 to IA32_MCi_CTL2[14:0]. This will cause overflow condition on every corrected MC error and generates a CMCI interrupt.
- To increase the threshold and reduce the frequency of CMCI servicing:
  a. Find the maximum threshold value a given processor implementation supports. The steps are:
     - Write 7FFFH to IA32_MCi_CTL2[14:0],
     - Read back IA32_MCi_CTL2[14:0]; these 15 bits (14:0) contain the maximum threshold supported by the processor.

  b. Increase the threshold to a value below the maximum value discovered using step a.

16.5.2.3 CMCI Interrupt Handler

The following describes techniques system software may consider to implement a CMCI service routine:

- The service routine examines its private per-thread data structure to check which set of MC banks it has ownership. If the thread does not have ownership of a given MC bank, proceed to the next MC bank. Ownership is determined at initialization time which is described in Section 16.5.2.1.

If the thread had claimed ownership to an MC bank, this technique will allow each logical processors to handle corrected MC errors independently and requires no synchronization to access shared MSR resources. Consult Example 16-5 for guidelines on logging when processing CMCI.

16.6 RECOVERY OF UNCORRECTED RECOVERABLE (UCR) ERRORS

Recovery of uncorrected recoverable machine check errors is an enhancement in machine-check architecture. The first processor that supports this feature is 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_2EH; see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. This allows system software to perform recovery action on a certain class of uncorrected errors and continue execution.

16.6.1 Detection of Software Error Recovery Support

Software must use bit 24 of IA32_MCG_CAP (MCG_SER_P) to detect the presence of software error recovery support (see Figure 16-2). When IA32_MCG_CAP[24] is set, this indicates that the processor supports software error recovery. When this bit is clear, this indicates that there is no support for error recovery from the processor and the primary responsibility of the machine check handler is logging the machine check error information and shutting down the system.

The new class of architectural MCA errors from which system software can attempt recovery is called Uncorrected Recoverable (UCR) Errors. UCR errors are uncorrected errors that have been detected and signaled but have not corrupted the processor context. For certain UCR errors, this means that once system software has performed a certain recovery action, it is possible to continue execution on this processor. UCR error reporting provides an error containment mechanism for data poisoning. The machine check handler will use the error log information from the error reporting registers to analyze and implement specific error recovery actions for UCR errors.

16.6.2 UCR Error Reporting and Logging

IA32_MCi_STATUS MSR is used for reporting UCR errors and existing corrected or uncorrected errors. The definitions of IA32_MCi_STATUS, including bit fields to identify UCR errors, is shown in Figure 16-6. UCR errors can be signaled through either the corrected machine check interrupt (CMCI) or machine check exception (MCE) path depending on the type of the UCR error.

When IA32_MCG_CAP[24] is set, a UCR error is indicated by the following bit settings in the IA32_MCi_STATUS register:

- Valid (bit 63) = 1
- UC (bit 61) = 1
- PCC (bit 57) = 0

Additional information from the IA32_MCi_MISC and the IA32_MCi_ADDR registers for the UCR error are available when the ADDRV and the MISCV flags in the IA32_MCi_STATUS register are set (see Section 16.3.2.4). The MCA error code field of the IA32_MCi_STATUS register indicates the type of UCR error. System software can interpret the MCA error code field to analyze and identify the necessary recovery action for the given UCR error.

In addition, the IA32_MCi_STATUS register bit fields, bits 56:55, are defined (see Figure 16-6) to provide additional information to help system software to properly identify the necessary recovery action for the UCR error:

- S (Signaling) flag, bit 56 - Indicates (when set) that a machine check exception was generated for the UCR error reported in this MC bank and system software needs to check the AR flag and the MCA error code fields in the IA32_MCi_STATUS register to identify the necessary recovery action for this error. When the S flag in the IA32_MCi_STATUS register is clear, this UCR error was not signaled via a machine check exception and instead was reported as a corrected machine check (CMC). System software is not required to take any recovery action when the S flag in the IA32_MCi_STATUS register is clear.
- AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. This recovery action must be completed successfully before any additional work is scheduled for this processor. When the RIPV flag in the IA32_MCG_STATUS is clear, an alternative execution stream needs to be provided; when the MCA error code specific recovery specific recovery action cannot be successfully completed, system software must shut down the system. When the AR flag in the IA32_MCi_STATUS register is clear, system software may still take MCA error code specific recovery action but this is optional; system software can safely resume program execution at the instruction pointer saved on the stack from the machine check exception when the RIPV flag in the IA32_MCG_STATUS register is set.

Both the S and the AR flags in the IA32_MCi_STATUS register are defined to be sticky bits, which mean that once set, the processor does not clear them. Only software and good power-on reset can clear the S and the AR-flags. Both the S and the AR flags are only set when the processor reports the UCR errors (MCG_CAP[24] is set).

16.6.3 UCR Error Classification

With the S and AR flag encoding in the IA32_MCi_STATUS register, UCR errors can be classified as:

- Uncorrected no action required (UCNA) - is a UCR error that is not signaled via a machine check exception and, instead, is reported to system software as a corrected machine check error. UCNA errors indicate that some data in the system is corrupted, but the data has not been consumed and the processor state is valid and you may continue execution on this processor. UCNA errors require no action from system software to continue execution. A UCNA error is indicated with UC=1, PCC=0, S=0 and AR=0 in the IA32_MCi_STATUS register.
- Software recoverable action optional (SRAO) - a UCR error is signaled either via a machine check exception or CMCI. System software recovery action is optional and not required to continue execution from this machine check exception. SRAO errors indicate that some data in the system is corrupt, but the data has not been consumed and the processor state is valid. SRAO errors provide the additional error information for system software to perform a recovery action. An SRAO error when signaled as a machine check is indicated with UC=1, PCC=0, S=1, EN=1 and AR=0 in the IA32_MC*i*_STATUS register. In cases when SRAO is signaled via CMCI the error signature is indicated via UC=1, PCC=0, S=0. Recovery actions for SRAO errors are MCA error code specific. The MISCV and the ADDRV flags in the IA32_MC*i*_STATUS register are set when the additional error information is available from the IA32_MC*i*_MISC and the IA32_MC*i*_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MC*i*_STATUS register to identify the specific recovery action for a given SRAO error. If MISCV and ADDRV are not set, it is recommended that no system software error recovery be performed however, system software can resume execution.
- Software recoverable action required (SRAR) - a UCR error that requires system software to take a recovery action on this processor before scheduling another stream of execution on this processor. SRAR errors indicate

that the error was detected and raised at the point of the consumption in the execution flow. An SRAR error is indicated with UC=1, PCC=0, S=1, EN=1 and AR=1 in the IA32_MCi_STATUS register. Recovery actions are MCA error code specific. The MISCV and the ADDRV flags in the IA32_MCi_STATUS register are set when the additional error information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MCi_STATUS register to identify the specific recovery action for a given SRAR error. If MISCV and ADDRV are not set, it is recommended that system software shutdown the system.

Table 16-7 summarizes UCR, corrected, and uncorrected errors.

Table 16-7. MC Error Classifications

Type of Error[1]	UC	EN	PCC	S	AR	Signaling	Software Action	Example
Uncorrected Error (UC)	1	1	1	x	x	MCE	If EN=1, reset the system, else log and OK to keep the system running.	
SRAR	1	1	0	1	1	MCE	For known MCACOD, take specific recovery action; For unknown MCACOD, must bugcheck. If OVER=1, reset system, else take specific recovery action.	Cache to processor load error.
SRAO	1	x[2]	0	x[2]	0	MCE/CMC	For known MCACOD, take specific recovery action; For unknown MCACOD, OK to keep the system running.	Patrol scrub and explicit writeback poison errors.
UCNA	1	x	0	0	0	CMC	Log the error and Ok to keep the system running.	Poison detection error.
Corrected Error (CE)	0	x	x	x	x	CMC	Log the error and no corrective action required.	ECC in caches and memory.

NOTES:

1. SRAR, SRAO and UCNA errors are supported by the processor only when IA32_MCG_CAP[24] (MCG_SER_P) is set.
2. EN=1, S=1 when signaled via MCE. EN=x, S=0 when signaled via CMC.

16.6.4 UCR Error Overwrite Rules

In general, the overwrite rules are as follows:

- UCR errors will overwrite corrected errors.
- Uncorrected (PCC=1) errors overwrite UCR (PCC=0) errors.
- UCR errors are not written over previous UCR errors.
- Corrected errors do not write over previous UCR errors.

Regardless of whether the 1st error is retained or the 2nd error is overwritten over the 1st error, the OVER flag in the IA32_MCi_STATUS register will be set to indicate an overflow condition. As the S flag and AR flag in the IA32_MCi_STATUS register are defined to be sticky flags, a second event cannot clear these 2 flags once set, however the MC bank information may be filled in for the 2nd error. The table below shows the overwrite rules and how to treat a second error if the first event is already logged in a MC bank along with the resulting bit setting of the UC, PCC, and AR flags in the IA32_MCi_STATUS register. As UCNA and SRA0 errors do not require recovery action from system software to continue program execution, a system reset by system software is not required unless the AR flag or PCC flag is set for the UCR overflow case (OVER=1, VAL=1, UC=1, PCC=0).

Table 16-8 lists overwrite rules for uncorrected errors, corrected errors, and uncorrected recoverable errors.

Table 16-8. Overwrite Rules for UC, CE, and UCR Errors

First Event	Second Event	UC	PCC	S	AR	MCA Bank	Reset System
CE	UCR	1	0	0 if UCNA, else 1	1 if SRAR, else 0	second	yes, if AR=1
UCR	CE	1	0	0 if UCNA, else 1	1 if SRAR, else 0	first	yes, if AR=1

Table 16-8. Overwrite Rules for UC, CE, and UCR Errors

First Event	Second Event	UC	PCC	S	AR	MCA Bank	Reset System
UCNA	UCNA	1	0	0	0	first	no
UCNA	SRAO	1	0	1	0	first	no
UCNA	SRAR	1	0	1	1	first	yes
SRAO	UCNA	1	0	1	0	first	no
SRAO	SRAO	1	0	1	0	first	no
SRAO	SRAR	1	0	1	1	first	yes
SRAR	UCNA	1	0	1	1	first	yes
SRAR	SRAO	1	0	1	1	first	yes
SRAR	SRAR	1	0	1	1	first	yes
UCR	UC	1	1	undefined	undefined	second	yes
UC	UCR	1	1	undefined	undefined	first	yes

16.7 MACHINE-CHECK AVAILABILITY

The machine-check architecture and machine-check exception (#MC) are model-specific features. Software can execute the CPUID instruction to determine whether a processor implements these features. Following the execution of the CPUID instruction, the settings of the MCA flag (bit 14) and MCE flag (bit 7) in EDX indicate whether the processor implements the machine-check architecture and machine-check exception.

16.8 MACHINE-CHECK INITIALIZATION

To use the processors machine-check architecture, software must initialize the processor to activate the machine-check exception and the error-reporting mechanism.

Example 16-1 gives pseudocode for performing this initialization. This pseudocode checks for the existence of the machine-check architecture and exception; it then enables machine-check exception and the error-reporting register banks. The pseudocode shown is compatible with the Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors.

Following power up or power cycling, IA32_MC*i*_STATUS registers are not guaranteed to have valid data until after they are initially cleared to zero by software (as shown in the initialization pseudocode in Example 16-1).

Example 16-1. Machine-Check Initialization Pseudocode

```
Check CPUID Feature Flags for MCE and MCA support
IF CPU supports MCE
THEN
    IF CPU supports MCA
    THEN
        IF (IA32_MCG_CAP.MCG_CTL_P = 1)
        (* IA32_MCG_CTL register is present *)
        THEN
            IA32_MCG_CTL ← FFFFFFFFFFFFFFFFH;
            (* enables all MCA features *)
        FI

        IF (IA32_MCG_CAP.MCG_LMCE_P = 1 and IA32_FEATURE_CONTROL.LOCK = 1 and IA32_FEATURE_CONTROL.LMCE_ENABLED = 1)
        (* IA32_MCG_EXT_CTL register is present and platform has enabled LMCE to permit system software to use LMCE *)
        THEN
            IA32_MCG_EXT_CTL ← IA32_MCG_EXT_CTL | 01H;
            (* System software enables LMCE capability for hardware to signal MCE to a single logical processor*)
        FI
        (* Determine number of error-reporting banks supported *)
        COUNT← IA32_MCG_CAP.Count;
        MAX_BANK_NUMBER ← COUNT - 1;

        IF (Processor Family is 6H and Processor EXTMODEL:MODEL is less than 1AH)
        THEN
            (* Enable logging of all errors except for MC0_CTL register *)
            FOR error-reporting banks (1 through MAX_BANK_NUMBER)
            DO
                IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFFH;
            OD

        ELSE
            (* Enable logging of all errors including MC0_CTL register *)
            FOR error-reporting banks (0 through MAX_BANK_NUMBER)
            DO
                IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFFH;
            OD
        FI

        (* BIOS clears all errors only on power-on reset *)
        IF (BIOS detects Power-on reset)
        THEN
            FOR error-reporting banks (0 through MAX_BANK_NUMBER)
            DO
                IA32_MCi_STATUS ← 0;
            OD
        ELSE
            FOR error-reporting banks (0 through MAX_BANK_NUMBER)
            DO
                (Optional for BIOS and OS) Log valid errors
                (OS only) IA32_MCi_STATUS ← 0;
            OD

        FI
    FI

    Setup the Machine Check Exception (#MC) handler for vector 18 in IDT

    Set the MCE bit (bit 6) in CR4 register to enable Machine-Check Exceptions
FI
```

16.9 INTERPRETING THE MCA ERROR CODES

When the processor detects a machine-check error condition, it writes a 16-bit error code to the MCA error code field of one of the IA32_MC*i*_STATUS registers and sets the VAL (valid) flag in that register. The processor may also write a 16-bit model-specific error code in the IA32_MC*i*_STATUS register depending on the implementation of the machine-check architecture of the processor.

The MCA error codes are architecturally defined for Intel 64 and IA-32 processors. To determine the cause of a machine-check exception, the machine-check exception handler must read the VAL flag for each IA32_MC*i*_STATUS register. If the flag is set, the machine check-exception handler must then read the MCA error code field of the register. It is the encoding of the MCA error code field [15:0] that determines the type of error being reported and not the register bank reporting it.

There are two types of MCA error codes: simple error codes and compound error codes.

16.9.1 Simple Error Codes

Table 16-9 shows the simple error codes. These unique codes indicate global error information.

Table 16-9. IA32_MCi_Status [15:0] Simple Error Code Encoding

Error Code	Binary Encoding	Meaning
No Error	0000 0000 0000 0000	No error has been reported to this bank of error-reporting registers.
Unclassified	0000 0000 0000 0001	This error has not been classified into the MCA error classes.
Microcode ROM Parity Error	0000 0000 0000 0010	Parity error in internal microcode ROM
External Error	0000 0000 0000 0011	The BINIT# from another processor caused this processor to enter machine check.[1]
FRC Error	0000 0000 0000 0100	FRC (functional redundancy check) main/secondary error.
Internal Parity Error	0000 0000 0000 0101	Internal parity error.
SMM Handler Code Access Violation	0000 0000 0000 0110	An attempt was made by the SMM Handler to execute outside the ranges specified by SMRR.
Internal Timer Error	0000 0100 0000 0000	Internal timer error.
I/O Error	0000 1110 0000 1011	generic I/O error.
Internal Unclassified	0000 01xx xxxx xxxx	Internal unclassified errors. [2]

NOTES:

1. BINIT# assertion will cause a machine check exception if the processor (or any processor on the same external bus) has BINIT# observation enabled during power-on configuration (hardware strapping) and if machine check exceptions are enabled (by setting CR4.MCE = 1).
2. At least one X must equal one. Internal unclassified errors have not been classified.

16.9.2 Compound Error Codes

Compound error codes describe errors related to the TLBs, memory, caches, bus and interconnect logic, and internal timer. A set of sub-fields is common to all of compound errors. These sub-fields describe the type of access, level in the cache hierarchy, and type of request. Table 16-10 shows the general form of the compound error codes.

Table 16-10. IA32_MCi_Status [15:0] Compound Error Code Encoding

Type	Form	Interpretation
Generic Cache Hierarchy	000F 0000 0000 11LL	Generic cache hierarchy error
TLB Errors	000F 0000 0001 TTLL	{TT}TLB{LL}_ERR
Memory Controller Errors	000F 0000 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Cache Hierarchy Errors	000F 0001 RRRR TTLL	{TT}CACHE{LL}_{RRRR}_ERR
Extended Memory Errors	000F 0010 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Bus and Interconnect Errors	000F 1PPT RRRR IILL	BUS{LL}_{PP}_{RRRR}_{II}_{T}_ERR

The “Interpretation” column in the table indicates the name of a compound error. The name is constructed by substituting mnemonics for the sub-field names given within curly braces. For example, the error code ICACHEL1_RD_ERR is constructed from the form:

{TT}CACHE{LL}_{RRRR}_ERR,
where {TT} is replaced by I, {LL} is replaced by L1, and {RRRR} is replaced by RD.

For more information on the “Form” and “Interpretation” columns, see Section 16.9.2.1, “Correction Report Filtering (F) Bit,” through Section 16.9.2.5, “Bus and Interconnect Errors.”

16.9.2.1 Correction Report Filtering (F) Bit

Starting with Intel Core Duo processors, bit 12 in the “Form” column in Table 16-10 is used to indicate that a particular posting to a log may be the last posting for corrections in that line/entry, at least for some time:

- 0 in bit 12 indicates “normal” filtering (original P6/Pentium4/Atom/Xeon processor meaning).
- 1 in bit 12 indicates “corrected” filtering (filtering is activated for the line/entry in the posting). Filtering means that some or all of the subsequent corrections to this entry (in this structure) will not be posted. The enhanced error reporting introduced with the Intel Core Duo processors is based on tracking the lines affected by repeated corrections (see Section 16.4, “Enhanced Cache Error reporting”). This capability is indicated by IA32_MCG_CAP[11]. Only the first few correction events for a line are posted; subsequent redundant correction events to the same line are not posted. Uncorrected events are always posted.

The behavior of error filtering after crossing the yellow threshold is model-specific. Filtering has meaning only for corrected errors (UC=0 in IA32_MCi_STATUS MSR). System software must ignore filtering bit (12) for uncorrected errors.

16.9.2.2 Transaction Type (TT) Sub-Field

The 2-bit TT sub-field (Table 16-11) indicates the type of transaction (data, instruction, or generic). The sub-field applies to the TLB, cache, and interconnect error conditions. Note that interconnect error conditions are primarily associated with P6 family and Pentium processors, which utilize an external APIC bus separate from the system bus. The generic type is reported when the processor cannot determine the transaction type.

Table 16-11. Encoding for TT (Transaction Type) Sub-Field

Transaction Type	Mnemonic	Binary Encoding
Instruction	I	00
Data	D	01
Generic	G	10

16.9.2.3 Level (LL) Sub-Field

The 2-bit LL sub-field (see Table 16-12) indicates the level in the memory hierarchy where the error occurred (level 0, level 1, level 2, or generic). The LL sub-field also applies to the TLB, cache, and interconnect error conditions. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support two levels in the cache hierarchy and one level in the TLBs. Again, the generic type is reported when the processor cannot determine the hierarchy level.

Table 16-12. Level Encoding for LL (Memory Hierarchy Level) Sub-Field

Hierarchy Level	Mnemonic	Binary Encoding
Level 0	L0	00
Level 1	L1	01
Level 2	L2	10
Generic	LG	11

16.9.2.4 Request (RRRR) Sub-Field

The 4-bit RRRR sub-field (see Table 16-13) indicates the type of action associated with the error. Actions include read and write operations, prefetches, cache evictions, and snoops. Generic error is returned when the type of error cannot be determined. Generic read and generic write are returned when the processor cannot determine the type of instruction or data request that caused the error. Eviction and snoop requests apply only to the caches. All of the other requests apply to TLBs, caches, and interconnects.

Table 16-13. Encoding of Request (RRRR) Sub-Field

Request Type	Mnemonic	Binary Encoding
Generic Error	ERR	0000
Generic Read	RD	0001
Generic Write	WR	0010
Data Read	DRD	0011
Data Write	DWR	0100
Instruction Fetch	IRD	0101
Prefetch	PREFETCH	0110
Eviction	EVICT	0111
Snoop	SNOOP	1000
Page Walk	PW	1001
EPT Page Walk	EPW	1010

16.9.2.5 Bus and Interconnect Errors

The bus and interconnect errors are defined with the 2-bit PP (participation), 1-bit T (time-out), and 2-bit II (memory or I/O) sub-fields, in addition to the LL and RRRR sub-fields (see Table 16-14). The bus error conditions are implementation dependent and related to the type of bus implemented by the processor. Likewise, the interconnect error conditions are predicated on a specific implementation-dependent interconnect model that describes the connections between the different levels of the storage hierarchy. The type of bus is implementation dependent, and as such is not specified in this document. A bus or interconnect transaction consists of a request involving an address and a response.

Table 16-14. Encodings of PP, T, and II Sub-Fields

Sub-Field	Transaction	Mnemonic	Binary Encoding
PP (Participation)	Local processor* originated request	SRC	00
	Local processor* responded to request	RES	01
	Local processor* observed error as third party	OBS	10
	Generic		11
T (Time-out)	Request timed out	TIMEOUT	1
	Request did not time out	NOTIMEOUT	0
II (Memory or I/O)	Memory Access	M	00
	Reserved		01
	I/O	IO	10
	Other transaction		11

NOTE:

* Local processor differentiates the processor reporting the error from other system components (including the APIC, other processors, etc.).

16.9.2.6 Memory Controller and Extended Memory Errors

The memory controller errors are defined with the 3-bit MMM (memory transaction type), and 4-bit CCCC (channel) sub-fields. The encodings for MMM and CCCC are defined in Table 16-15. Extended Memory errors use the same encodings and are used to report errors in memory used as a cache.

Table 16-15. Encodings of MMM and CCCC Sub-Fields

Sub-Field	Transaction	Mnemonic	Binary Encoding
MMM	Generic undefined request	GEN	000
	Memory read error	RD	001
	Memory write error	WR	010
	Address/Command Error	AC	011
	Memory Scrubbing Error	MS	100
	Reserved		101-111
CCCC	Channel number	CHN	0000-1110
	Channel not specified		1111

Note that the CCCC channel number may be enumerated from zero separately by each memory controller on a system. On a multi-socket system, or a system with multiple memory controllers per socket, it is necessary to also consider which machine check bank logged the error. See Chapter 17 for details on specific implementations.

16.9.3 Architecturally Defined UCR Errors

Software recoverable compound error code are defined in this section.

16.9.3.1 Architecturally Defined SRAO Errors

The following two SRAO errors are architecturally defined.

- UCR Errors detected by memory controller scrubbing; and
- UCR Errors detected during L3 cache (L3) explicit writebacks.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 16-10). Their values and compound encoding format are given in Table 16-16.

Table 16-16. MCA Compound Error Code Encoding for SRAO Errors

Type	MCACOD Value	MCA Error Code Encoding[1]
Memory Scrubbing	C0H - CFH	0000_0000_1100_CCCC 000F 0000 1MMM CCCC (Memory Controller Error), where Memory subfield MMM = 100B (memory scrubbing) Channel subfield CCCC = channel # or generic
L3 Explicit Writeback	17AH	0000_0001_0111_1010 000F 0001 RRRR TTLL (Cache Hierarchy Error) where Request subfields RRRR = 0111B (Eviction) Transaction Type subfields TT = 10B (Generic) Level subfields LL = 10B

NOTES:

1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 16-17 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAO errors.

Table 16-17. IA32_MCi_STATUS Values for SRAO Errors

SRAO Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR	MCACOD
Memory Scrubbing	1	0	1	x[1]	1	1	0	x[1]	0	C0H-CFH
L3 Explicit Writeback	1	0	1	x[1]	1	1	0	x[1]	0	17AH

NOTES:

1. When signaled as MCE, EN=1 and S=1. If error was signaled via CMC, then EN=x, and S=0.

For both the memory scrubbing and L3 explicit writeback errors, the ADDRV and MISCV flags in the IA32_MCi_STATUS register are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the memory scrubbing and L3 explicit writeback errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors as outlined in Section 16.10.4.1. If LMCE is supported and enabled, some errors (not limited to UCR errors) may be delivered to only a single logical processor. System software should consult IA32_MCG_STATUS.LMCE_S to determine if the MCE signaled is only to this logical processor.

IA32_MCi_STATUS banks can be shared by logical processors within a core or within the same package. So several logical processors may find an SRAO error in the shared IA32_MCi_STATUS bank but other processors do not find it in any of the IA32_MCi_STATUS banks. Table 16-18 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the memory scrubbing and L3 explicit writeback errors on both the reporting and non-reporting logical processors.

Table 16-18. IA32_MCG_STATUS Flag Indication for SRAO Errors

SRAO Type	Reporting Logical Processors		Non-reporting Logical Processors	
	RIPV	EIPV	RIPV	EIPV
Memory Scrubbing	1	0	1	0
L3 Explicit Writeback	1	0	1	0

16.9.3.2 Architecturally Defined SRAR Errors

The following six SRAR errors are architecturally defined:

- UCR Errors detected on data load;
- UCR Errors detected on data page walk;
- UCR Errors detected on data page walk on EPT;
- UCR Errors detected on instruction fetch;
- UCR Errors detected on instruction fetch page walk; and
- UCR Errors detected on instruction fetch page walk on EPT.

The MCA error code encodings for these six architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 16-10). Their values and compound encoding format are given in Table 16-19.

Table 16-19. MCA Compound Error Code Encoding for SRAR Errors

Type	MCACOD Value	MCA Error Code Encoding[1]
Data Load	134H	0000_0001_0011_0100: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0011B (Data Load), Transaction Type subfield TT= 01B (Data), and Level subfield LL = 00B (Level 0).
Data Page Walk	194H	0000_0001_1001_0100: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1001B (Page Walk), Transaction Type subfield TT= 01B (Data), and Level subfield LL = 00B (Level 0).
Data Page Walk on EPT	1A4H	0000_0001_1010_0100: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1010B (EPT Page Walk), Transaction Type subfield TT= 01B (Data), and Level subfield LL = 00B (Level 0).
Instruction Fetch	150H	0000_0001_0101_0000: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0101B (Instruction Fetch), Transaction Type subfield TT= 00B (Instruction), and Level subfield LL = 00B (Level 0).
Instruction Fetch Page Walk	190H	0000_0001_1001_0000: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1001B (Page Walk), Transaction Type subfield TT= 00B (Instruction), and Level subfield LL = 00B (Level 0).
Instruction Fetch Page Walk on EPT	1A0H	0000_0001_1010_0000: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1010B (EPT Page Walk), Transaction Type subfield TT= 00B (Instruction), and Level subfield LL = 00B (Level 0).

NOTES:

[1] Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 16-20 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAR errors.

Table 16-20. IA32_MCi_STATUS Values for All Defined SRAR Errors

SRAR Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR
All defined SRAR errors defined in Table 16-19	1	0	1	1	1	1	0	1	1

For all defined SRAR errors, the ADDRV and MISCV flags in the IA32_MCi_STATUS register are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the data load and instruction fetch errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors on the system on which the UCR errors are supported, except when the processor supports LMCE and LMCE is enabled by system software (see Section 16.3.1.5). The IA32_MCG_STATUS MSR allows system software to distinguish the affected logical processor of an SRAR error amongst logical processors that observed SRAR via MCi_STATUS bank.

Table 16-21 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the data load and instruction fetch errors on both the reporting and non-reporting logical processors. The recoverable SRAR error reported by a processor may be continuable, where the system software can interpret the context of continuable as follows: the error was isolated, contained. If software can rectify the error condition in the current instruction stream, the execution context on that logical processor can be continued without loss of information.

Table 16-21. IA32_MCG_STATUS Flag Indication for SRAR Errors

SRAR Type	Affected Logical Processor			Non-Affected Logical Processors		
	RIPV	EIPV	Continuable	RIPV	EIPV	Continuable
Recoverable-continuable	1	1	Yes[1]	1	0	Yes
Recoverable-not-continuable	0	x	No			

NOTES:

[1] See the definition of the context of "continuable" above and additional details below.

SRAR Error And Affected Logical Processors

The affected logical processor is the one that has detected and raised an SRAR error at the point of the consumption in the execution flow. The affected logical processor should find the Data Load or the Instruction Fetch error information in the IA32_MCi_STATUS register that is reporting the SRAR error.

Table 16-21 list the actionable scenarios that system software can respond to an SRAR error on an affected logical processor according to RIPV and EIPV values:

- Recoverable-continuable SRAR Error (RIPV=1, EIPV=1):

  For recoverable-continuable SRAR errors, the affected logical processor should find that both the IA32_MCG_STATUS.RIPV and the IA32_MCG_STATUS.EIPV flags are set, indicating that system software may be able to restart execution from the interrupted context if it is able to rectify the error condition. If system software cannot rectify the error condition then it must treat the error as a recoverable error where restarting execution with the interrupted context is not possible. Restarting without rectifying the error condition will result in most cases with another SRAR error on the same instruction.

- Recoverable-not-continuable SRAR Error (RIPV=0, EIPV=x):

  For recoverable-not-continuable errors, the affected logical processor should find that either

  — IA32_MCG_STATUS.RIPV= 0, IA32_MCG_STATUS.EIPV=1, or

  — IA32_MCG_STATUS.RIPV= 0, IA32_MCG_STATUS.EIPV=0.

  In either case, this indicates that the error is detected at the instruction pointer saved on the stack for this machine check exception and restarting execution with the interrupted context is not possible. System software may take the following recovery actions for the affected logical processor:

  - The current executing thread cannot be continued. System software must terminate the interrupted stream of execution and provide a new stream of execution on return from the machine check handler for the affected logical processor.

SRAR Error And Non-Affected Logical Processors

The logical processors that observed but not affected by an SRAR error should find that the RIPV flag in the IA32_MCG_STATUS register is set and the EIPV flag in the IA32_MCG_STATUS register is cleared, indicating that it is safe to restart the execution at the instruction saved on the stack for the machine check exception on these processors after the recovery action is successfully taken by system software.

16.9.4 Multiple MCA Errors

When multiple MCA errors are detected within a certain detection window, the processor may aggregate the reporting of these errors together as a single event, i.e., a single machine exception condition. If this occurs, system software may find multiple MCA errors logged in different MC banks on one logical processor or find multiple MCA errors logged across different processors for a single machine check broadcast event. In order to handle multiple UCR errors reported from a single machine check event and possibly recover from multiple errors, system software may consider the following:

- Whether it can recover from multiple errors is determined by the most severe error reported on the system. If the most severe error is found to be an unrecoverable error (VAL=1, UC=1, PCC=1 and EN=1) after system software examines the MC banks of all processors to which the MCA signal is broadcast, recovery from the multiple errors is not possible and system software needs to reset the system.

- When multiple recoverable errors are reported and no other fatal condition (e.g., overflowed condition for SRAR error) is found for the reported recoverable errors, it is possible for system software to recover from the multiple recoverable errors by taking necessary recovery action for each individual recoverable error. However, system software can no longer expect one to one relationship with the error information recorded in the IA32_MCi_STATUS register and the states of the RIPV and EIPV flags in the IA32_MCG_STATUS register as the states of the RIPV and the EIPV flags in the IA32_MCG_STATUS register may indicate the information for the most severe error recorded on the processor. System software is required to use the RIPV flag indication in the IA32_MCG_STATUS register to make a final decision of recoverability of the errors and find the restart-ability requirement after examining each IA32_MCi_STATUS register error information in the MC banks.

  In certain cases where system software observes more than one SRAR error logged for a single logical processor, it can no longer rely on affected threads as specified in Table 15-20 above. System software is recommended to reset the system if this condition is observed.

16.9.5 Machine-Check Error Codes Interpretation

Chapter 17, "Interpreting Machine Check Error Codes," provides information on interpreting the MCA error code, model-specific error code, and other information error code fields. For P6 family processors, information has been included on decoding external bus errors. For Pentium 4 and Intel Xeon processors; information is included on external bus, internal timer and cache hierarchy errors.

16.10 GUIDELINES FOR WRITING MACHINE-CHECK SOFTWARE

The machine-check architecture and error logging can be used in three different ways:

- To detect machine errors during normal instruction execution, using the machine-check exception (#MC).
- To periodically check and log machine errors.
- To examine recoverable UCR errors, determine software recoverability and perform recovery actions via a machine-check exception handler or a corrected machine-check interrupt handler.

To use the machine-check exception, the operating system or executive software must provide a machine-check exception handler. This handler may need to be designed specifically for each family of processors.

A special program or utility is required to log machine errors.

Guidelines for writing a machine-check exception handler or a machine-error logging utility are given in the following sections.

16.10.1 Machine-Check Exception Handler

The machine-check exception (#MC) corresponds to vector 18. To service machine-check exceptions, a trap gate must be added to the IDT. The pointer in the trap gate must point to a machine-check exception handler. Two approaches can be taken to designing the exception handler:

1. The handler can merely log all the machine status and error information, then call a debugger or shut down the system.
2. The handler can analyze the reported error information and, in some cases, attempt to correct the error and restart the processor.

For Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors; virtually all machine-check conditions cannot be corrected (they result in abort-type exceptions). The logging of status and error information is therefore a baseline implementation requirement.

When IA32_MCG_CAP[24] is clear, consider the following when writing a machine-check exception handler:

- To determine the nature of the error, the handler must read each of the error-reporting register banks. The count field in the IA32_MCG_CAP register gives number of register banks. The first register of register bank 0 is at address 400H.

- The VAL (valid) flag in each IA32_MC*i*_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and do not need to be checked.
- To write a portable exception handler, only the MCA error code field in the IA32_MC*i*_STATUS register should be checked. See Section 16.9, "Interpreting the MCA Error Codes," for information that can be used to write an algorithm to interpret this field.
- Correctable errors are corrected automatically by the processor. The UC flag in each IA32_MCi_STATUS register indicates whether the processor automatically corrected an error.
- The RIPV, PCC, and OVER flags in each IA32_MCi_STATUS register indicate whether recovery from the error is possible. If PCC or OVER are set, recovery is not possible. If RIPV is not set, program execution can not be restarted reliably. When recovery is not possible, the handler typically records the error information and signals an abort to the operating system.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether the program can be restarted at the instruction indicated by the instruction pointer (the address of the instruction pushed on the stack when the exception was generated). If this flag is clear, the processor may still be able to be restarted (for debugging purposes) but not without loss of program continuity.
- For unrecoverable errors, the EIPV flag in the IA32_MCG_STATUS register indicates whether the instruction indicated by the instruction pointer pushed on the stack (when the exception was generated) is related to the error. If the flag is clear, the pushed instruction may not be related to the error.
- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. Before returning from the machine-check exception handler, software should clear this flag so that it can be used reliably by an error logging utility. The MCIP flag also detects recursion. The machine-check architecture does not support recursion. When the processor detects machine-check recursion, it enters the shutdown state.

Example 16-2 gives typical steps carried out by a machine-check exception handler.

Example 16-2. Machine-Check Exception Handler Pseudocode

```
IF CPU supports MCE
    THEN
        IF CPU supports MCA
            THEN
                call errorlogging routine; (* returns restartability *)
        FI;
    ELSE (* Pentium(R) processor compatible *)
        READ P5_MC_ADDR
        READ P5_MC_TYPE;
        report RESTARTABILITY to console;
FI;
IF error is not restartable
    THEN
        report RESTARTABILITY to console;
        abort system;
FI;
CLEAR MCIP flag in IA32_MCG_STATUS;
```

16.10.2 Pentium Processor Machine-Check Exception Handling

Machine-check exception handler on P6 family, Intel Atom and later processor families, should follow the guidelines described in Section 16.10.1 and Example 16-2 that check the processor's support of MCA.

NOTE

> On processors that support MCA (CPUID.1.EDX.MCA = 1) reading the P5_MC_TYPE and P5_MC_ADDR registers may produce invalid data.

When machine-check exceptions are enabled for the Pentium processor (MCE flag is set in control register CR4), the machine-check exception handler uses the RDMSR instruction to read the error type from the P5_MC_TYPE register and the machine check address from the P5_MC_ADDR register. The handler then normally reports these register values to the system console before aborting execution (see Example 16-2).

16.10.3 Logging Correctable Machine-Check Errors

The error handling routine for servicing the machine-check exceptions is responsible for logging uncorrected errors.

If a machine-check error is correctable, the processor does not generate a machine-check exception for it. To detect correctable machine-check errors, a utility program must be written that reads each of the machine-check error-reporting register banks and logs the results in an accounting file or data structure. This utility can be implemented in either of the following ways.

- A system daemon that polls the register banks on an infrequent basis, such as hourly or daily.
- A user-initiated application that polls the register banks and records the exceptions. Here, the actual polling service is provided by an operating-system driver or through the system call interface.
- An interrupt service routine servicing CMCI can read the MC banks and log the error. Please refer to Section 16.10.4.2 for guidelines on logging correctable machine checks.

Example 16-3 gives pseudocode for an error logging utility.

Example 16-3. Machine-Check Error Logging Pseudocode

```
Assume that execution is restartable;
IF the processor supports MCA
    THEN
    FOR each bank of machine-check registers
        DO
            READ IA32_MCi_STATUS;
            IF VAL flag in IA32_MCi_STATUS = 1
                THEN
                    IF ADDRV flag in IA32_MCi_STATUS = 1
                        THEN READ IA32_MCi_ADDR;
                    FI;
                    IF MISCV flag in IA32_MCi_STATUS = 1
                        THEN READ IA32_MCi_MISC;
                    FI;
                    IF MCIP flag in IA32_MCG_STATUS = 1
                        (* Machine-check exception is in progress *)
                        AND PCC flag in IA32_MCi_STATUS = 1
                        OR RIPV flag in IA32_MCG_STATUS = 0
                        (* execution is not restartable *)
                            THEN
                                RESTARTABILITY = FALSE;
                                return RESTARTABILITY to calling procedure;
                    FI;
                    Save time-stamp counter and processor ID;
                    Set IA32_MCi_STATUS to all 0s;
                    Execute serializing instruction (i.e., CPUID);
            FI;
        OD;
FI;
```

If the processor supports the machine-check architecture, the utility reads through the banks of error-reporting registers looking for valid register entries. It then saves the values of the IA32_MC*i*_STATUS, IA32_MC*i*_ADDR, IA32_MC*i*_MISC, and IA32_MCG_STATUS registers for each bank that is valid. The routine minimizes processing time by recording the raw data into a system data structure or file, reducing the overhead associated with polling. User utilities analyze the collected data in an off-line environment.

When the MCIP flag is set in the IA32_MCG_STATUS register, a machine-check exception is in progress and the machine-check exception handler has called the exception logging routine.

Once the logging process has been completed the exception-handling routine must determine whether execution can be restarted, which is usually possible when damage has not occurred (The PCC flag is clear, in the IA32_MC*i*_STATUS register) and when the processor can guarantee that execution is restartable (the RIPV flag is set in the IA32_MCG_STATUS register). If execution cannot be restarted, the system is not recoverable and the exception-handling routine should signal the console appropriately before returning the error status to the Operating System kernel for subsequent shutdown.

The machine-check architecture allows buffering of exceptions from a given error-reporting bank although the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors do not implement this feature. The error logging routine should provide compatibility with future processors by reading each hardware error-reporting bank's IA32_MC*i*_STATUS register and then writing 0s to clear the OVER and VAL flags in this register. The error logging utility should re-read the IA32_MC*i*_STATUS register for the bank ensuring that the valid bit is clear. The processor will write the next error into the register bank and set the VAL flags.

Additional information that should be stored by the exception-logging routine includes the processor's time-stamp counter value, which provides a mechanism to indicate the frequency of exceptions. A multiprocessing operating system stores the identity of the processor node incurring the exception using a unique identifier, such as the processor's APIC ID (see Section 11.8, "Handling Interrupts").

The basic algorithm given in Example 16-3 can be modified to provide more robust recovery techniques. For example, software has the flexibility to attempt recovery using information unavailable to the hardware. Specifically, the machine-check exception handler can, after logging carefully analyze the error-reporting registers when the error-logging routine reports an error that does not allow execution to be restarted. These recovery techniques can use external bus related model-specific information provided with the error report to localize the source of the error within the system and determine the appropriate recovery strategy.

16.10.4 Machine-Check Software Handler Guidelines for Error Recovery

16.10.4.1 Machine-Check Exception Handler for Error Recovery

When writing a machine-check exception (MCE) handler to support software recovery from Uncorrected Recoverable (UCR) errors, consider the following:

- When IA32_MCG_CAP [24] is zero, there are no recoverable errors supported and all machine-check are fatal exceptions. The logging of status and error information is therefore a baseline implementation requirement.
- When IA32_MCG_CAP [24] is 1, certain uncorrected errors called uncorrected recoverable (UCR) errors may be software recoverable. The handler can analyze the reported error information, and in some cases attempt to recover from the uncorrected error and continue execution.
- For processors on which CPUID reports DisplayFamily_DisplayModel as 06H_0EH and onward, an MCA signal is broadcast to all logical processors in the system; see the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. Due to the potentially shared machine check MSR resources among the logical processors on the same package/core, the MCE handler may be required to synchronize with the other processors that received a machine check error and serialize access to the machine check registers when analyzing, logging, and clearing the information in the machine check registers.
  - On processors that indicate ability for local machine-check exception (MCG_LMCE_P), hardware can choose to report the error to only a single logical processor if system software has enabled LMCE by setting IA32_MCG_EXT_CTL[LMCE_EN] = 1 as outlined in Section 16.3.1.5.
- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and should not be checked.
- The MCE handler is primarily responsible for processing uncorrected errors. The UC flag in each IA32_MCi_Status register indicates whether the reported error was corrected (UC=0) or uncorrected (UC=1). The MCE handler can optionally log and clear the corrected errors in the MC banks if it can implement software algorithm to avoid the undesired race conditions with the CMCI or CMC polling handler.
- For uncorrectable errors, the EIPV flag in the IA32_MCG_STATUS register indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is

generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.

- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. When a machine check exception is generated, it is expected that the MCIP flag in the IA32_MCG_STATUS register is set to 1. If it is not set, this machine check was generated by either an INT 18 instruction or some piece of hardware signaling an interrupt with vector 18.

When IA32_MCG_CAP [24] is 1, the following rules can apply when writing a machine check exception (MCE) handler to support software recovery:

- The PCC flag in each IA32_MCi_STATUS register indicates whether recovery from the error is possible for uncorrected errors (UC=1). If the PCC flag is set for enabled uncorrected errors (UC=1 and EN=1), recovery is not possible. When recovery is not possible, the MCE handler typically records the error information and signals the operating system to reset the system.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether restarting the program execution from the instruction pointer saved on the stack for the machine check exception is possible. When the RIPV is set, program execution can be restarted reliably when recovery is possible. If the RIPV flag is not set, program execution cannot be restarted reliably. In this case the recovery algorithm may involve terminating the current program execution and resuming an alternate thread of execution upon return from the machine check handler when recovery is possible. When recovery is not possible, the MCE handler signals the operating system to reset the system.
- When the EN flag is zero but the VAL and UC flags are one in the IA32_MCi_STATUS register, the reported uncorrected error in this bank is not enabled. As uncorrected errors with the EN flag = 0 are not the source of machine check exceptions, the MCE handler should log and clear non-enabled errors when the S bit is set and should continue searching for enabled errors from the other IA32_MCi_STATUS registers. Note that when IA32_MCG_CAP [24] is 0, any uncorrected error condition (VAL =1 and UC=1) including the one with the EN flag cleared are fatal and the handler must signal the operating system to reset the system. For the errors that do not generate machine check exceptions, the EN flag has no meaning.
- When the VAL flag is one, the UC flag is one, the EN flag is one and the PCC flag is zero in the IA32_MCi_STATUS register, the error in this bank is an uncorrected recoverable (UCR) error. The MCE handler needs to examine the S flag and the AR flag to find the type of the UCR error for software recovery and determine if software error recovery is possible.
- When both the S and the AR flags are clear in the IA32_MCi_STATUS register for the UCR error (VAL=1, UC=1, EN=x and PCC=0), the error in this bank is an uncorrected no-action required error (UCNA). UCNA errors are uncorrected but do not require any OS recovery action to continue execution. These errors indicate that some data in the system is corrupt, but that data has not been consumed and may not be consumed. If that data is consumed a non-UCNA machine check exception will be generated. UCNA errors are signaled in the same way as corrected machine check errors and the CMCI and CMC polling handler is primarily responsible for handling UCNA errors. Like corrected errors, the MCA handler can optionally log and clear UCNA errors as long as it can avoid the undesired race condition with the CMCI or CMC polling handler. As UCNA errors are not the source of machine check exceptions, the MCA handler should continue searching for uncorrected or software recoverable errors in all other MC banks.
- When the S flag in the IA32_MCi_STATUS register is set for the UCR error ((VAL=1, UC=1, EN=1 and PCC=0), the error in this bank is software recoverable and it was signaled through a machine-check exception. The AR flag in the IA32_MCi_STATUS register further clarifies the type of the software recoverable errors.
- When the AR flag in the IA32_MCi_STATUS register is clear for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action optional (SRAO) error. The MCE handler and the operating system can analyze the IA32_MCi_STATUS [15:0] to implement MCA error code specific optional recovery action, but this recovery action is optional. System software can resume the program execution from the instruction pointer saved on the stack for the machine check exception when the RIPV flag in the IA32_MCG_STATUS register is set.
- Even if the OVER flag in the IA32_MCi_STATUS register is set for the SRAO error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=0), the MCE handler can take recovery action for the SRAO error logged in the IA32_MCi_STATUS register. Since the recovery action for SRAO errors is optional, restarting the program execution from the instruction pointer saved on the stack for the machine check exception is still possible for the overflowed SRAO error if the RIPV flag in the IA32_MCG_STATUS is set.

- When the AR flag in the IA32_MCi_STATUS register is set for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action required (SRAR) error. The MCE handler and the operating system must take recovery action in order to continue execution after the machine-check exception. The MCA handler and the operating system need to analyze the IA32_MCi_STATUS [15:0] to determine the MCA error code specific recovery action. If no recovery action can be performed, the operating system must reset the system.
- When the OVER flag in the IA32_MCi_STATUS register is set for the SRAR error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=1), the MCE handler cannot take recovery action as the information of the SRAR error in the IA32_MCi_STATUS register was potentially lost due to the overflow condition. Since the recovery action for SRAR errors must be taken, the MCE handler must signal the operating system to reset the system.
- When the MCE handler cannot find any uncorrected (VAL=1, UC=1 and EN=1) or any software recoverable errors (VAL=1, UC=1, EN=1, PCC=0 and S=1) in any of the IA32_MCi banks of the processors, this is an unexpected condition for the MCE handler and the handler should signal the operating system to reset the system.
- Before returning from the machine-check exception handler, software must clear the MCIP flag in the IA32_MCG_STATUS register. The MCIP flag is used to detect recursion. The machine-check architecture does not support recursion. When the processor receives a machine check when MCIP is set, it automatically enters the shutdown state.

Example 16-4 gives pseudocode for an MC exception handler that supports recovery of UCR.

Example 16-4. Machine-Check Error Handler Pseudocode Supporting UCR

```
MACHINE CHECK HANDLER:  (* Called from INT 18 handler *)
NOERROR = TRUE;
ProcessorCount = 0;
IF CPU supports MCA
    THEN
        RESTARTABILITY = TRUE;
        IF (Processor Family = 6 AND DisplayModel ≥ 0EH) OR (Processor Family > 6)
            THEN
                IF ( MCG_LMCE = 1)
                    MCA_BROADCAST = FALSE;
                ELSE
                    MCA_BROADCAST = TRUE;
                FI;
                Acquire SpinLock;
                ProcessorCount++;  (* Allowing one logical processor at a time to examine machine check registers *)
                CALL MCA ERROR PROCESSING; (* returns RESTARTABILITY and NOERROR *)
            ELSE
                MCA_BROADCAST = FALSE;
                (* Implement a rendezvous mechanism with the other processors if necessary *)
                CALL MCA ERROR PROCESSING;
        FI;
    ELSE (* Pentium(R) processor compatible *)
        READ P5_MC_ADDR
        READ P5_MC_TYPE;
        RESTARTABILITY = FALSE;
FI;

IF NOERROR = TRUE
    THEN
        IF NOT (MCG_RIPV = 1 AND MCG_EIPV = 0)
            THEN
                RESTARTABILITY = FALSE;
        FI
FI;

IF RESTARTABILITY = FALSE
    THEN
        Report RESTARTABILITY to console;
        Reset system;
```

```
FI;

IF MCA_BROADCAST = TRUE
    THEN
        IF ProcessorCount = MAX_PROCESSORS
          AND NOERROR = TRUE
            THEN
                Report RESTARTABILITY to console;
                Reset system;
        FI;
        Release SpinLock;
        Wait till ProcessorCount = MAX_PROCESSRS on system;
        (* implement a timeout and abort function if necessary *)
FI;
CLEAR IA32_MCG_STATUS;
RESUME Execution;
(* End of MACHINE CHECK HANDLER*)

MCA ERROR PROCESSING:    (* MCA Error Processing Routine called from MCA Handler *)
IF MCIP flag in IA32_MCG_STATUS = 0
    THEN (* MCIP=0 upon MCA is unexpected *)
        RESTARTABILITY = FALSE;
FI;
FOR each bank of machine-check registers
    DO
        CLEAR_MC_BANK = FALSE;
        READ IA32_MCi_STATUS;
        IF VAL Flag in IA32_MCi_STATUS = 1
            THEN
                IF UC Flag in IA32_MCi_STATUS = 1
                    THEN
                        IF Bit 24 in IA32_MCG_CAP = 0
                            THEN (* the processor does not support software error recovery *)
                                RESTARTABILITY = FALSE;
                                NOERROR = FALSE;
                                GOTO LOG MCA REGISTER;
                        FI;
                        (* the processor supports software error recovery *)
                        IF EN Flag in IA32_MCi_STATUS = 0 AND OVER Flag in IA32_MCi_STATUS=0
                            THEN (* It is a spurious MCA Log. Log and clear the register *)
                                CLEAR_MC_BANK = TRUE;
                                GOTO LOG MCA REGISTER;
                        FI;
                        IF PCC = 1 and EN = 1 in IA32_MCi_STATUS
                            THEN (* processor context might have been corrupted *)
                                RESTARTABILITY = FALSE;
                            ELSE (* It is a uncorrected recoverable (UCR) error *)
                                IF S Flag in IA32_MCi_STATUS = 0
                                    THEN
                                        IF AR Flag in IA32_MCi_STATUS = 0
                                            THEN (* It is a uncorrected no action required (UCNA) error *)
                                                GOTO CONTINUE; (* let CMCI and CMC polling handler to process *)
                                            ELSE
                                                RESTARTABILITY = FALSE; (* S=0, AR=1 is illegal *)
                                        FI
                                FI;
                                IF RESTARTABILITY = FALSE
                                    THEN (* no need to take recovery action if RESTARTABILITY is already false *)
                                        NOERROR = FALSE;
                                        GOTO LOG MCA REGISTER;
                                FI;
                                (* S in IA32_MCi_STATUS = 1 *)
                                IF AR Flag in IA32_MCi_STATUS = 1
                                    THEN (* It is a software recoverable and action required (SRAR) error *)
                                        IF OVER Flag in IA32_MCi_STATUS = 1
```

```
                                 THEN
                                    RESTARTABILITY = FALSE;
                                    NOERROR = FALSE;
                                    GOTO LOG MCA REGISTER;
                              FI
                              IF MCACOD Value in IA32_MCi_STATUS is recognized
                                AND Current Processor is an Affected Processor
                                 THEN
                                    Implement MCACOD specific recovery action;
                                    CLEAR_MC_BANK = TRUE;
                                 ELSE
                                    RESTARTABILITY = FALSE;
                              FI;
                           ELSE (* It is a software recoverable and action optional (SRAO) error *)
                              IF OVER Flag in IA32_MCi_STATUS = 0 AND
                               MCACOD in IA32_MCi_STATUS is recognized
                                 THEN
                                    Implement MCACOD specific recovery action;
                              FI;
                              CLEAR_MC_BANK = TRUE;
                        FI; AR
                  FI; PCC
                  NOERROR = FALSE;
                  GOTO LOG MCA REGISTER;
               ELSE  (* It is a corrected error; continue to the next IA32_MCi_STATUS *)
                  GOTO CONTINUE;
            FI; UC
      FI; VAL
LOG MCA REGISTER:
      SAVE IA32_MCi_STATUS;
      If MISCV in IA32_MCi_STATUS
         THEN
            SAVE IA32_MCi_MISC;
      FI;
      IF ADDRV in IA32_MCi_STATUS
         THEN
            SAVE IA32_MCi_ADDR;
      FI;
      IF CLEAR_MC_BANK = TRUE
         THEN
            SET all 0 to IA32_MCi_STATUS;
            If MISCV in IA32_MCi_STATUS
               THEN
                  SET all 0 to IA32_MCi_MISC;
            FI;
            IF ADDRV in IA32_MCi_STATUS
               THEN
                  SET all 0 to IA32_MCi_ADDR;
            FI;
      FI;
      CONTINUE:
   OD;
( *END FOR *)
RETURN;
(* End of MCA ERROR PROCESSING*)
```

16.10.4.2 Corrected Machine-Check Handler for Error Recovery

When writing a corrected machine check handler, which is invoked as a result of CMCI or called from an OS CMC Polling dispatcher, consider the following:

- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank does not contain valid error information and does not need to be checked.

- The CMCI or CMC polling handler is responsible for logging and clearing corrected errors. The UC flag in each IA32_MCi_Status register indicates whether the reported error was corrected (UC=0) or not (UC=1).
- When IA32_MCG_CAP [24] is one, the CMC handler is also responsible for logging and clearing uncorrected no-action required (UCNA) errors. When the UC flag is one but the PCC, S, and AR flags are zero in the IA32_MCi_STATUS register, the reported error in this bank is an uncorrected no-action required (UCNA) error. In cases when SRAO error are signaled as UCNA error via CMCI, software can perform recovery for those errors identified in Table 16-16.
- In addition to corrected errors and UCNA errors, the CMC handler optionally logs uncorrected (UC=1 and PCC=1), software recoverable machine check errors (UC=1, PCC=0 and S=1), but should avoid clearing those errors from the MC banks. Clearing these errors may result in accidentally removing these errors before these errors are actually handled and processed by the MCE handler for attempted software error recovery.

Example 16-5 gives pseudocode for a CMCI handler with UCR support.

Example 16-5. Corrected Error Handler Pseudocode with UCR Support

```
Corrected Error HANDLER: (* Called from CMCI handler or OS CMC Polling Dispatcher*)
IF CPU supports MCA
    THEN
        FOR each bank of machine-check registers
            DO
                READ IA32_MCi_STATUS;
                IF VAL flag in IA32_MCi_STATUS = 1
                    THEN
                        IF UC Flag in IA32_MCi_STATUS = 0 (* It is a corrected error *)
                            THEN
                                GOTO LOG CMC ERROR;
                            ELSE
                                IF Bit 24 in IA32_MCG_CAP = 0
                                    THEN
                                        GOTO CONTINUE;
                                FI;
                                IF S Flag in IA32_MCi_STATUS = 0 AND AR Flag in IA32_MCi_STATUS = 0
                                    THEN (* It is a uncorrected no action required error *)
                                        GOTO LOG CMC ERROR
                                FI
                                IF EN Flag in IA32_MCi_STATUS = 0
                                    THEN (* It is a spurious MCA error *)
                                        GOTO LOG CMC ERROR
                                FI;
                        FI;
                FI;
                GOTO CONTINUE;
            LOG CMC ERROR:
                SAVE IA32_MCi_STATUS;
                If MISCV Flag in IA32_MCi_STATUS
                    THEN
                        SAVE IA32_MCi_MISC;
                        SET all 0 to IA32_MCi_MISC;
                FI;
                IF ADDRV Flag in IA32_MCi_STATUS
                    THEN
                        SAVE IA32_MCi_ADDR;
                        SET all 0 to IA32_MCi_ADDR
                FI;
                SET all 0 to IA32_MCi_STATUS;
                CONTINUE:
            OD;
    ( *END FOR *)
FI;
```

CHAPTER 17
INTERPRETING MACHINE CHECK ERROR CODES

Encoding of the model-specific and other information fields is different across processor families. The differences are documented in the following sections.

17.1 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 06H, MACHINE ERROR CODES FOR MACHINE CHECK

This section provides information for interpreting additional model-specific fields for external bus errors relating to processor family 06H. The references to processor family 06H refers to only IA-32 processors with CPUID signatures listed in Table 17-1.

Table 17-1. CPUID DisplayFamily_DisplayModel Signatures for Processor Family 06H

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_0EH	Intel® Core™ Duo processor, Intel® Core™ Solo processor
06_0DH	Intel Pentium M processor
06_09H	Intel Pentium M processor
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon Processor, Intel Pentium III Processor
06_03H, 06_05H	Intel Pentium II Xeon Processor, Intel Pentium II Processor
06_01H	Intel Pentium Pro Processor

These errors are reported in the IA32_MCi_STATUS MSRs. They are reported architecturally as compound errors with a general form of **0000 1PPT RRRR IILL** in the MCA error code field. See Chapter 16 for information on the interpretation of compound error codes. Incremental decoding information is listed in Table 17-2.

Table 17-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes for Machine Check

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0		
Model Specific Errors	18:16	Reserved	Reserved
	24:19	Bus Queue Request Type	000000: BQ_DCU_READ_TYPE error. 000010: BQ_IFU_DEMAND_TYPE error. 000011: BQ_IFU_DEMAND_NC_TYPE error. 000100: BQ_DCU_RFO_TYPE error. 000101: BQ_DCU_RFO_LOCK_TYPE error. 000110: BQ_DCU_ITOM_TYPE error. 001000: BQ_DCU_WB_TYPE error. 001010: BQ_DCU_WCEVICT_TYPE error. 001011: BQ_DCU_WCLINE_TYPE error. 001100: BQ_DCU_BTM_TYPE error.

Table 17-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes for Machine Check

Type	Bit No.	Bit Function	Bit Description
			001101: BQ_DCU_INTACK_TYPE error. 001110: BQ_DCU_INVALL2_TYPE error. 001111: BQ_DCU_FLUSHL2_TYPE error. 010000: BQ_DCU_PART_RD_TYPE error. 010010: BQ_DCU_PART_WR_TYPE error. 010100: BQ_DCU_SPEC_CYC_TYPE error. 011000: BQ_DCU_IO_RD_TYPE error. 011001: BQ_DCU_IO_WR_TYPE error. 011100: BQ_DCU_LOCK_RD_TYPE error. 011110: BQ_DCU_SPLOCK_RD_TYPE error. 011101: BQ_DCU_LOCK_WR_TYPE error.
	27:25	Bus Queue Error Type	000: BQ_ERR_HARD_TYPE error. 001: BQ_ERR_DOUBLE_TYPE error. 010: BQ_ERR_AERR2_TYPE error. 100: BQ_ERR_SINGLE_TYPE error. 101: BQ_ERR_AERR1_TYPE error.
	28	FRC Error	1 if FRC error active.
	29	BERR	1 if BERR is driven.
	30	Internal BINIT	1 if BINIT driven for this processor.
	31	Reserved	Reserved
Other Information	34:32	Reserved	Reserved
	35	External BINIT	1 if BINIT is received from external bus.
	36	Response Parity Error	This bit is asserted in IA32_MC*i*_STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.
	37	Bus BINIT	This bit is asserted in IA32_MC*i*_STATUS if this component has received a hard error response on a split transaction one access that has needed to be split across the 64-bit external bus interface into two accesses).
	38	Timeout BINIT	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 15-bit ROB time-out counter carries a 1 out of its high order bit. [2] The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock (the bus clock is 1:2, 1:3, 1:4 of the core clock[3]). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	41:39	Reserved	Reserved
	42	Hard Error	This bit is asserted in IA32_MC*i*_STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.
	43	IERR	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.

Table 17-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes for Machine Check

Type	Bit No.	Bit Function	Bit Description
	44	AERR	This bit is asserted in IA32_MC*i*_STATUS if this component has initiated 2 failing bus transactions which have failed due to Address Parity Errors AERR asserted). While this bit is asserted, it cannot be overwritten.
	45	UECC	The Uncorrectable ECC error bit is asserted in IA32_MC*i*_STATUS for uncorrected ECC errors. While this bit is asserted, the ECC syndrome field will not be overwritten.
	46	CECC	The correctable ECC error bit is asserted in IA32_MC*i*_STATUS for corrected ECC errors.
	54:47	ECC Syndrome	The ECC syndrome field in IA32_MCi_STATUS contains the 8-bit ECC syndrome only if the error was a correctable/uncorrectable ECC error and there wasn't a previous valid ECC error syndrome logged in IA32_MCi_STATUS. A previous valid ECC error in IA32_MCi_STATUS is indicated by IA32_MCi_STATUS.bit45 uncorrectable error occurred) being asserted. After processing an ECC error, machine check handling software should clear IA32_MCi_STATUS.bit45 so that future ECC error syndromes can be logged.
	56:55	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.
2. For processors with a CPUID signature of 06_0EH, a ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit.
3. For processors with a CPUID signature of 6_06_60H and later, the PIC timer will count crystal clock cycles.

17.2 INCREMENTAL DECODING INFORMATION: INTEL® CORE™ 2 PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-4 provides information for interpreting additional model-specific fields for external bus errors relating to processors based on Intel® Core™ microarchitecture, which implements the P4 bus specification. Table 17-3 lists the CPUID signatures for Intel 64 processors that are covered by Table 17-4. These errors are reported in the IA32_MCi_STATUS MSRs. They are reported architecturally as compound errors with a general form of **0000 1PPT RRRR IILL** in the MCA error code field. See Chapter 16 for information on the interpretation of compound error codes.

Table 17-3. CPUID DisplayFamily_DisplayModel Signatures for Processors Based on Intel® Core™ Microarchitecture

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_1DH	Intel® Xeon® Processor 7400 series
06_17H	Intel® Xeon® Processor 5200, 5400 series, Intel® Core™ 2 Quad processor Q9650
06_0FH	Intel® Xeon® Processor 3000, 3200, 5100, 5300, 7300 series, Intel® Core™ 2 Quad, Intel® Core™ 2 Extreme, Intel® Core™ 2 Duo processors, Intel Pentium dual-core processors

Table 17-4. Incremental Bus Error Codes of Machine Check for Processors Based on Intel® Core™ Microarchitecture

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0		
Model Specific Errors	18:16	Reserved	Reserved
	24:19	Bus Queue Request Type	'000001: BQ_PREF_READ_TYPE error. 000000: BQ_DCU_READ_TYPE error. 000010: BQ_IFU_DEMAND_TYPE error 000011: BQ_IFU_DEMAND_NC_TYPE error. 000100: BQ_DCU_RFO_TYPE error. 000101: BQ_DCU_RFO_LOCK_TYPE error. 000110: BQ_DCU_ITOM_TYPE error. 001000: BQ_DCU_WB_TYPE error. 001010: BQ_DCU_WCEVICT_TYPE error. 001011: BQ_DCU_WCLINE_TYPE error. 001100: BQ_DCU_BTM_TYPE error. 001101: BQ_DCU_INTACK_TYPE error. 001110: BQ_DCU_INVALL2_TYPE error. 001111: BQ_DCU_FLUSHL2_TYPE error. 010000: BQ_DCU_PART_RD_TYPE error. 010010: BQ_DCU_PART_WR_TYPE error. 010100: BQ_DCU_SPEC_CYC_TYPE error. 011000: BQ_DCU_IO_RD_TYPE error. 011001: BQ_DCU_IO_WR_TYPE error. 011100: BQ_DCU_LOCK_RD_TYPE error. 011110: BQ_DCU_SPLOCK_RD_TYPE error. 011101: BQ_DCU_LOCK_WR_TYPE error. 100100: BQ_L2_WI_RFO_TYPE error. 100110: BQ_L2_WI_ITOM_TYPE error.
	27:25	Bus Queue Error Type	'001: Address Parity Error. '010: Response Hard Error. '011: Response Parity Error.
	28	MCE Driven	1 if MCE is driven.
	29	MCE Observed	1 if MCE is observed.
	30	Internal BINIT	1 if BINIT driven for this processor.
	31	BINIT Observed	1 if BINIT is observed for this processor.
Other Information	33:32	Reserved	Reserved
	34	PIC and FSB Data Parity	Data Parity detected on either PIC or FSB access.
	35	Reserved	Reserved
	36	Response Parity Error	This bit is asserted in IA32_MC*i*_STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.

Table 17-4. Incremental Bus Error Codes of Machine Check for Processors Based on Intel® Core™ Microarchitecture (Contd.)

Type	Bit No.	Bit Function	Bit Description
	37	FSB Address Parity	Address parity error detected: 1: Address parity error detected. 0: No address parity error.
	38	Timeout BINIT	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit. The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	41:39	Reserved	Reserved
	42	Hard Error	This bit is asserted in IA32_MC*i*_STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.
	43	IERR	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.
	44	Reserved	Reserved
	45	Reserved	Reserved
	46	Reserved	Reserved
	54:47	Reserved	Reserved
	56:55	Reserved	Reserved.
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.2.1 Model-Specific Machine Check Error Codes for Intel® Xeon® Processor 7400 Series

The Intel® Xeon® processor 7400 series has machine check register banks that generally follow the description of Chapter 16 and Section 17.2. Additional error codes specific to the Intel Xeon processor 7400 series are described in this section.

MC4_STATUS[63:0] is the main error logging for the processor’s L3 and front side bus errors for the Intel Xeon processor 7400 series. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.

17.2.1.1 Processor Machine Check Status Register, Incremental MCA Error Code Definition

The Intel Xeon processor 7400 series uses compound MCA Error Codes for logging its Bus internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines incremental Machine Check error types (IA32_MC6_STATUS[15:0]) beyond those defined in Chapter 16. Table 17-5 lists these incremental MCA error code types that apply to IA32_MC6_STATUS. Error code details are specified in MC6_STATUS [31:16] (see Section 17.2.2), the "Model Specific Error Code" field. The information in the "Other_Info" field (MC4_STATUS[56:32]) is common to the three processor error types. It contains a correctable event count and specifies the MC6_MISC register format.

Table 17-5. Incremental MCA Error Code Types for Intel® Xeon® Processor 7400

Processor MCA_Error_Code (MC6_STATUS[15:0])			
Type	**Error Code**	**Binary Encoding**	**Meaning**
C	Internal Error	0000 0100 0000 0000	Internal Error Type Code.
B	Bus and Interconnect Error	0000 100x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations.
		0000 101x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations.
		0000 110x 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations.
		0000 1110 0000 1111	Bus and Interconnection Error Type Code.
		0000 1111 0000 1111	Not used but this encoding is reserved for compatibility with other MCA implementations.

The **Bold faced** binary encodings are the only encodings used by the processor for MC4_STATUS[15:0].

17.2.2 Intel® Xeon® Processor 7400 Model Specific Error Code Field

17.2.2.1 Processor Model Specific Error Code Field, Type B: Bus and Interconnect Error Codes

The Model Specific Error Code field in MC6_STATUS (bits 31:16) is defined in Table 17-6.

Table 17-6. Type B: Bus and Interconnect Error Codes

Bit Number	Sub-Field Name	Description
16	FSB Request Parity	Parity error detected during FSB request phase.
19:17	Reserved	Reserved
20	FSB Hard Fail Response	"Hard Failure" response received for a local transaction.
21	FSB Response Parity	Parity error on FSB response field detected.
22	FSB Data Parity	FSB data parity error on inbound data detected.
31:23	Reserved	Reserved

17.2.2.2 Processor Model Specific Error Code Field, Type C: Cache Bus Controller Error Codes

Table 17-7. Type C: Cache Bus Controller Error Codes

MC4_STATUS[31:16] (MSCE) Value	Error Description
0000_0000_0000_0001 0001H	Inclusion Error from Core 0.
0000_0000_0000_0010 0002H	Inclusion Error from Core 1.
0000_0000_0000_0011 0003H	Write Exclusive Error from Core 0.
0000_0000_0000_0100 0004H	Write Exclusive Error from Core 1.
0000_0000_0000_0101 0005H	Inclusion Error from FSB.
0000_0000_0000_0110 0006H	SNP Stall Error from FSB.
0000_0000_0000_0111 0007H	Write Stall Error from FSB.
0000_0000_0000_1000 0008H	FSB Arb Timeout Error.
0000_0000_0000_1010 000AH	Inclusion Error from Core 2.
0000_0000_0000_1011 000BH	Write Exclusive Error from Core 2.
0000_0010_0000_0000 0200H	Internal Timeout Error.
0000_0011_0000_0000 0300H	Internal Timeout Error.
0000_0100_0000_0000 0400H	Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow.
0000_0101_0000_0000 0500H	Quiet cycle Timeout Error (correctable).
1100_0000_0000_0010 C002H	Correctable ECC event on outgoing Core 0 data.
1100_0000_0000_0100 C004H	Correctable ECC event on outgoing Core 1 data.
1100_0000_0000_1000 C008H	Correctable ECC event on outgoing Core 2 data.
1110_0000_0000_0010 E002H	Uncorrectable ECC error on outgoing Core 0 data.
1110_0000_0000_0100 E004H	Uncorrectable ECC error on outgoing Core 1 data.
1110_0000_0000_1000 E008H	Uncorrectable ECC error on outgoing Core 2 data.
— All other encodings —	Reserved

17.3 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR 3400, 3500, 5500 SERIES, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-8 through Table 17-12 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor 3400, 3500, 5500 series with CPUID DisplayFamily_DisplaySignature 06_1AH, which supports Intel® QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC0 and IA32_MC1, incremental error codes for internal machine check are reported in the register bank IA32_MC7, and incremental error codes for the memory controller unit are reported in the register bank IA32_MC8.

17.3.1 Intel® QPI Machine Check Errors

Table 17-8. Intel® QPI Machine Check Error Codes for IA32_MC0_STATUS and IA32_MC1_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model Specific Errors	16	Header Parity	If 1, QPI Header had bad parity.
	17	Data Parity	If 1, QPI Data packet had bad parity.
	18	Retries Exceeded	If 1, the number of QPI retries was exceeded.
	19	Received Poison	if 1, received a data packet that was marked as poisoned by the sender.
	21:20	Reserved	Reserved
	22	Unsupported Message	If 1, QPI received a message encoding it does not support.
	23	Unsupported Credit	If 1, QPI credit type is not supported.
	24	Receive Flit Overrun	If 1, sender sent too many QPI flits to the receiver.
	25	Received Failed Response	If 1, indicates that sender sent a failed response to receiver.
	26	Receiver Clock Jitter	If 1, clock jitter detected in the internal QPI clocking.
	56:27	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

Table 17-9. Intel® QPI Machine Check Error Codes for IA32_MC0_MISC and IA32_MC1_MISC

Type	Bit No.	Bit Function	Bit Description
Model Specific Errors[1]	7:0	QPI Opcode	Message class and opcode from the packet with the error.
	13:8	RTID	QPI Request Transaction ID.
	15:14	Reserved	Reserved
	18:16	RHNID	QPI Requestor/Home Node ID.
	23:19	Reserved	Reserved
	24	IIB	QPI Interleave/Head Indication Bit.

NOTES:

1. Which of these fields are valid depends on the error type.

17.3.2 Internal Machine Check Errors

Table 17-10. Machine Check Error Codes for IA32_MC7_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	
Model Specific Errors	23:16	Reserved	Reserved
	31:24	Reserved, except for the following	00H: No error. 03H: Reset firmware did not complete. 08H: Received an invalid CMPD. 0AH: Invalid Power Management Request. 0DH: Invalid S-state transition. 11H: VID controller does not match POC controller selected. 1AH: MSID from POC does not match CPU MSID.
	56:32	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.3.3 Memory Controller Errors

Table 17-11. Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory error format: 1MMMCCCC
Model Specific Errors	16	Read ECC Error	If 1, ECC occurred on a read.
	17	RAS ECC Error	If 1, ECC occurred on a scrub.
	18	Write Parity Error	If 1, bad parity on a write.
	19	Redundancy Loss	if 1, error in half of redundant memory.
	20	Reserved	Reserved
	21	Memory Range Error	If 1, memory access out of range.
	22	RTID Out of Range	If 1, Internal ID invalid.
	23	Address Parity Error	If 1, bad address parity.
	24	Byte Enable Parity Error	If 1, bad enable parity.
Other Information	37:25	Reserved	Reserved
	52:38	CORE_ERR_CNT	Corrected error count.
	56:53	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

Table 17-12. Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_MISC

Type	Bit No.	Bit Function	Bit Description
Model Specific Errors[1]	7:0	RTID	Transaction Tracker ID.
	15:8	Reserved	Reserved
	17:16	DIMM	DIMM ID which received the error.
	19:18	Channel	Channel ID which received the error.
	31:20	Reserved	Reserved
	63:32	Syndrome	ECC Syndrome.

NOTES:

1. Which of these fields are valid depends on the error type.

17.4 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-13 through Table 17-15 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor E5 Family with CPUID DisplayFamily_DisplaySignature 06_2DH, which supports Intel QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC6 and IA32_MC7, incremental error codes for internal machine check error from PCU controller are reported in the register bank IA32_MC4, and incremental error codes for the memory controller unit are reported in the register banks IA32_MC8—IA32_MC11.

17.4.1 Internal Machine Check Errors

Table 17-13. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	
Model Specific Errors	19:16	Reserved, except for the following	0000b: No Error 0001b: Non_IMem_Sel 0010b: I_Parity_Error 0011b: Bad_OpCode 0100b: I_Stack_Underflow 0101b: I_Stack_Overflow 0110b: D_Stack_Underflow 0111b: D_Stack_Overflow 1000b: Non-DMem_Sel 1001b: D_Parity_Error

Table 17-13. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

Type	Bit No.	Bit Function	Bit Description
	23:20	Reserved	Reserved
	31:24	Reserved, except for the following	00H: No Error 0DH: MC_IMC_FORCE_SR_S3_TIMEOUT 0EH: MC_CPD_UNCPD_ST_TIMEOUT 0FH: MC_PKGS_SAFE_WP_TIMEOUT 43H: MC_PECI_MAILBOX_QUIESCE_TIMEOUT 5CH: MC_MORE_THAN_ONE_LT_AGENT 60H: MC_INVALID_PKGS_REQ_PCH 61H: MC_INVALID_PKGS_REQ_QPI 62H: MC_INVALID_PKGS_RES_QPI 63H: MC_INVALID_PKGC_RES_PCH 64H: MC_INVALID_PKG_STATE_CONFIG 70H: MC_WATCHDG_TIMEOUT_PKGC_SECONDARY 71H: MC_WATCHDG_TIMEOUT_PKGC_MAIN 72H: MC_WATCHDG_TIMEOUT_PKGS_MAIN 7AH: MC_HA_FAILSTS_CHANGE_DETECTED 81H: MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.4.2 Intel® QPI Machine Check Errors

Table 17-14. Intel® QPI MC Error Codes for IA32_MC6_STATUS and IA32_MC7_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model Specific Errors	56:16	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.4.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC8_STATUS–IA32_MC11_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture." MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in the IA32_MCi_STATUS and IA32_MCi_MISC, where i = 8, 11.

Table 17-15. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 8, 11)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model Specific Errors	31:16	Reserved, except for the following	001H: Address parity error. 002H: HA Wrt buffer Data parity error. 004H: HA Wrt byte enable parity error. 008H: Corrected patrol scrub error. 010H: Uncorrected patrol scrub error. 020H: Corrected spare error. 040H: Uncorrected spare error.
	36:32	Other Info	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first device error when corrected error is detected during normal read.
	37	Reserved	Reserved
	56:38		See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Table 17-16. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 8, 11)

Type	Bit No.	Bit Function	Bit Description
MCA Addr Info[1]	8:0		See Chapter 16, “Machine-Check Architecture.”
Model Specific Errors	13:9		▪ When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second device error when corrected error is detected during normal read. ▪ Otherwise, contains parity error if MCi_Status indicates HA_WB_Data or HA_W_BE parity error.
	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error bit mask.
	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error failing rank.
	58:56	Reserved	Reserved
	61:59	Reserved	Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data from the first correctable error in a memory device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error information indicated by bit 62.

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.5 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V2 AND INTEL® XEON® PROCESSOR E7 V2 FAMILIES, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v2 family and the Intel® Xeon® processor E7 v2 family are based on the Ivy Bridge-EP microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_3EH. Incremental error codes for internal machine check error from the PCU controller is reported in the register bank IA32_MC4; Table 17-17 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register bank IA32_MC5. Information listed in Table 17-14 for QPI MC error codes apply to IA32_MC5_STATUS. Incremental error codes for the memory controller unit are reported in the register banks IA32_MC9—IA32_MC16. Table 17-18 lists model-specific error codes that apply to IA32_MCi_STATUS, where i = 9-16.

17.5.1 Internal Machine Check Errors

Table 17-17. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	
Model Specific Errors	19:16	Reserved, except for the following	0000b: No Error 0001b: Non_IMem_Sel 0010b: I_Parity_Error 0011b: Bad_OpCode 0100b: I_Stack_Underflow 0101b: I_Stack_Overflow 0110b: D_Stack_Underflow 0111b: D_Stack_Overflow 1000b: Non-DMem_Sel 1001b: D_Parity_Error
	23:20	Reserved	Reserved
	31:24	Reserved, except for the following	00H: No Error 0DH: MC_IMC_FORCE_SR_S3_TIMEOUT 0EH: MC_CPD_UNCPD_ST_TIMEOUT 0FH: MC_PKGS_SAFE_WP_TIMEOUT 43H: MC_PECI_MAILBOX_QUIESCE_TIMEOUT 44H: MC_CRITICAL_VR_FAILED 45H: MC_ICC_MAX-NOTSUPPORTED 5CH: MC_MORE_THAN_ONE_LT_AGENT 60H: MC_INVALID_PKGS_REQ_PCH 61H: MC_INVALID_PKGS_REQ_QPI 62H: MC_INVALID_PKGS_RES_QPI 63H: MC_INVALID_PKGC_RES_PCH 64H: MC_INVALID_PKG_STATE_CONFIG 70H: MC_WATCHDG_TIMEOUT_PKGC_SECONDARY 71H: MC_WATCHDG_TIMEOUT_PKGC_MAIN 72H: MC_WATCHDG_TIMEOUT_PKGS_MAIN

Table 17-17. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

Type	Bit No.	Bit Function	Bit Description
			7AH: MC_HA_FAILSTS_CHANGE_DETECTED 7BH: MC_PCIE_R2PCIE-RW_BLOCK_ACK_TIMEOUT 81H: MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.5.2 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS—IA32_MC16_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC, where i = 9—16.

IA32_MCi_STATUS (i=9—12) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCi_STATUS (i=13—16).

Table 17-18. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—16)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory Controller error format: 000F 0000 1MMM CCCC
Model Specific Errors	31:16	Reserved, except for the following	001H: Address parity error. 002H: HA Wrt buffer data parity error. 004H: HA Wrt byte enable parity error. 008H: Corrected patrol scrub error.
			010H: Uncorrected patrol scrub error. 020H: Corrected spare error. 040H: Uncorrected spare error. 080H: Corrected memory read error. (Only applicable with iMC's "Additional Error logging" Mode-1 enabled.) 100H - iMC, WDB, parity errors
	36:32	Other Info	When MSR_ERROR_CONTROL.[1] is set, logs an encoded value from the first error device.
	37	Reserved	Reserved
	56:38		See Chapter 16, "Machine-Check Architecture."
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

Table 17-19. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 9—16)

Type	Bit No.	Bit Function	Bit Description
MCA Addr Info[1]	8:0		See Chapter 16, "Machine-Check Architecture."
Model Specific Errors	13:9		If the error logged is a MCWrDataPar error or a MCWrBEPar error, this field is the WDB ID that has the parity error; OR if the second error logged is a correctable read error, MC logs the second error device in this field.
	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error bit mask.
	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error failing rank.
	61:56		Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data from a correctable error from memory read associated with first error device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error info indicated by bit 62.

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.5.3 Home Agent Machine Check Errors

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors to the IA32_MC8_{STATUS,ADDR,MISC} registers.

17.6 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V3 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v3 family is based on the Haswell-E microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_3FH. Incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-20 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC5, IA32_MC20, and IA32_MC21. Table 17-21 contains information for QPI MC error codes. Incremental error codes for the memory controller unit are reported in the register banks IA32_MC9—IA32_MC16. Table 17-22 lists model-specific error codes that apply to IA32_MCi_STATUS, where i = 9—16.

17.6.1 Internal Machine Check Errors

Table 17-20. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	
MCACOD[2]	15:0	Internal Errors	0402H: PCU internal errors. 0403H: PCU internal errors. 0406H: Intel TXT errors 0407H: Other UBOX internal errors. An IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS. After a 3-strike, the core MCA banks will be unavailable.
Model Specific Errors	19:16	Reserved, except for the following	0000b: No error. 00xxb: PCU internal error.
	23:20	Reserved	Reserved
	31:24	Reserved, except for the following	00H: No Error 09H: MC_MESSAGE_CHANNEL_TIMEOUT 13H: MC_DMI_TRAINING_TIMEOUT 15H: MC_DMI_CPU_RESET_ACK_TIMEOUT 1EH: MC_VR_ICC_MAX_LT_FUSED_ICC_MAX 25H: MC_SVID_COMMAND_TIMEOUT 29H: MC_VR_VOUT_MAC_LT_FUSED_SVID 2BH: MC_PKGC_WATCHDOG_HANG_CBZ_DOWN 2CH: MC_PKGC_WATCHDOG_HANG_CBZ_UP 44H: MC_CRITICAL_VR_FAILED 46H: MC_VID_RAMP_DOWN_FAILED 49H: MC_SVID_WRITE_REG_VOUT_MAX_FAILED 4BH: MC_BOOT_VID_TIMEOUT; timeout setting boot VID for DRAM 0. 4FH: MC_SVID_COMMAND_ERROR 52H: MC_FIVR_CATAS_OVERVOL_FAULT 53H: MC_FIVR_CATAS_OVERCUR_FAULT 57H: MC_SVID_PKGC_REQUEST_FAILED 58H: MC_SVID_IMON_REQUEST_FAILED 59H: MC_SVID_ALERT_REQUEST_FAILED 62H: MC_INVALID_PKGS_RSP_QPI 64H: MC_INVALID_PKG_STATE_CONFIG 67H: MC_HA_IMC_RW_BLOCK_ACK_TIMEOUT 6AH: MC_MSGCH_PMREQ_CMP_TIMEOUT 72H: MC_WATCHDG_TIMEOUT_PKGS_MASTER 81H: MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

[2] The internal error codes may be model-specific.

17.6.2 Intel® QPI Machine Check Errors

MC error codes associated with the Intel QPI agents are reported in the IA32_MC5_STATUS, IA32_MC20_STATUS, and IA32_MC21_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1PPTRRRRIILL**; see Chapter 16, "Machine-Check Architecture."

Table 17-21 lists model-specific fields to interpret error codes applicable to IA32_MC5_STATUS, IA32_MC20_STATUS, and IA32_MC21_STATUS.

Table 17-21. Intel® QPI MC Error Codes for IA32_MCi_STATUS (i = 5, 20, 21)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Bus error format: 1PPTRRRRIILL
Model Specific Errors	31:16	MSCOD	02H: Intel QPI physical layer detected drift buffer alarm. 03H: Intel QPI physical layer detected latency buffer rollover. 10H: Intel QPI link layer detected control error from R3QPI. 11H: Rx entered LLR abort state on CRC error. 12H: Unsupported or undefined packet. 13H: Intel QPI link layer control error. 15H: RBT used un-initialized value. 20H: Intel QPI physical layer detected a QPI in-band reset but aborted initialization. 21H: Link failover data self-healing. 22H: Phy detected in-band reset (no width change). 23H: Link failover clock failover. 30H: Rx detected CRC error; successful LLR after Phy re-init. 31H: Rx detected CRC error; successful LLR without Phy re-init. All other values are reserved.
	37:32	Reserved	Reserved
	52:38	Corrected Error Cnt	
	56:53	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.6.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS—IA32_MC16_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC, where i = 9—16.

IA32_MCi_STATUS (i=9—12) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCi_STATUS (i=13—16).

Table 17-22. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—16)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model Specific Errors	31:16	Reserved, except for the following	0001H: DDR3 address parity error. 0002H: Uncorrected HA write data error.
			0004H: Uncorrected HA data byte enable error. 0008H: Corrected patrol scrub error. 0010H: Uncorrected patrol scrub error. 0020H: Corrected spare error. 0040H: Uncorrected spare error. 0080H: Corrected memory read error. (Only applicable with iMC's "Additional Error logging" Mode-1 enabled.) 0100H: iMC, write data buffer parity errors. 0200H: DDR4 command address parity error.
	36:32	Other Info	When MSR_ERROR_CONTROL.[1] is set, logs an encoded value from the first error device.
	37	Reserved	Reserved
	56:38		See Chapter 16, "Machine-Check Architecture."
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

Table 17-23. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 9—16)

Type	Bit No.	Bit Function	Bit Description
MCA Addr Info[1]	8:0		See Chapter 16, "Machine-Check Architecture."
Model Specific Errors	13:9		If the error logged is an MCWrDataPar error or an MCWrBEPar error, this field is the WDB ID that has the parity error; OR if the second error logged is a correctable read error, MC logs the second error device in this field.
	29:14	ErrMask_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error bit mask.
	45:30	ErrMask_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error bit mask.
	50:46	FailRank_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log first-device error failing rank.
	55:51	FailRank_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, allows the iMC to log second-device error failing rank.
	61:56	Reserved	Reserved
	62	Valid_1stErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data from a correctable error from a memory read associated with first error device.
	63	Valid_2ndErrDev	When MSR_ERROR_CONTROL.[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error information indicated by bit 62.

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.6.4 Home Agent Machine Check Errors

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors in the IA32_MC8_{STATUS,ADDR,MISC} registers.

17.7 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR D FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor D family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_56H. Incremental error codes for internal machine check error from the PCU controller are reported in the register bank IA32_MC4. Table 17-24 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental error codes for the memory controller unit are reported in the register banks IA32_MC9—IA32_MC10. Table 17-18 lists model-specific error codes that apply to IA32_M-Ci_STATUS, where i = 9—10.

17.7.1 Internal Machine Check Errors

Table 17-24. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	
MCACOD[2]	15:0	Internal Errors	0402H: PCU internal errors. 0403H: Internal errors. 0406H: Intel TXT errors. 0407H: Other UBOX internal errors. On an IERR caused by a core 3-strike, the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS. After a 3-strike, the core MCA banks will be unavailable.
Model Specific Errors	19:16	Reserved, except for the following	0000b: No error. 00x1b: PCU internal error. 001xb: PCU internal error.
	23:20	Reserved, except for the following	x1xxb: UBOX error.
	31:24	Reserved, except for the following	00H: No Error 09H: MC_MESSAGE_CHANNEL_TIMEOUT 13H: MC_DMI_TRAINING_TIMEOUT 15H: MC_DMI_CPU_RESET_ACK_TIMEOUT 1EH: MC_VR_ICC_MAX_LT_FUSED_ICC_MAX 25H: MC_SVID_COMMAND_TIMEOUT 26H: MCA_PKGC_DIRECT_WAKE_RING_TIMEOUT 29H: MC_VR_VOUT_MAC_LT_FUSED_SVID 2BH: MC_PKGC_WATCHDOG_HANG_CBZ_DOWN 2CH: MC_PKGC_WATCHDOG_HANG_CBZ_UP 44H: MC_CRITICAL_VR_FAILED 46H: MC_VID_RAMP_DOWN_FAILED 49H: MC_SVID_WRITE_REG_VOUT_MAX_FAILED

Table 17-24. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

Type	Bit No.	Bit Function	Bit Description
			4BH: MC_PP1_BOOT_VID_TIMEOUT. Timeout setting boot VID for DRAM 0. 4FH: MC_SVID_COMMAND_ERROR. 52H: MC_FIVR_CATAS_OVERVOL_FAULT. 53H: MC_FIVR_CATAS_OVERCUR_FAULT. 57H: MC_SVID_PKGC_REQUEST_FAILED 58H: MC_SVID_IMON_REQUEST_FAILED 59H: MC_SVID_ALERT_REQUEST_FAILED 62H: MC_INVALID_PKGS_RSP_QPI 64H: MC_INVALID_PKG_STATE_CONFIG 67H: MC_HA_IMC_RW_BLOCK_ACK_TIMEOUT 6AH: MC_MSGCH_PMREQ_CMP_TIMEOUT 72H: MC_WATCHDG_TIMEOUT_PKGS_MASTER 81H: MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56:32	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.
2. The internal error codes may be model-specific.

17.7.2 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS—IA32_MC10_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC, where i = 9—10.

Table 17-25. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—10)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model Specific Errors	31:16	Reserved, except for the following	0001H: DDR3 address parity error. 0002H: Uncorrected HA write data error. 0004H: Uncorrected HA data byte enable error. 0008H: Corrected patrol scrub error. 0010H: Uncorrected patrol scrub error. 0100H: iMC, write data buffer parity errors. 0200H: DDR4 command address parity error.
	36:32	Other Info	Reserved
	37	Reserved	Reserved
	56:38		See Chapter 16, "Machine-Check Architecture."
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.8 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V4 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v4 family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_4FH. Incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-20 in Section 17.6.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC5, IA32_MC20, and IA32_MC21. Information listed in Table 17-21 of Section 17.6.1 covers QPI MC error codes.

17.8.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS—IA32_MC16_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

Table 17-26 lists model-specific error codes that apply to IA32_MCi_STATUS, where i = 9—16.

IA32_MCi_STATUS (i=9—12) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCi_STATUS (i=13—16).

Table 17-26. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—16)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model Specific Errors	31:16	Reserved, except for the following	0001H: DDR3 address parity error. 0002H: Uncorrected HA write data error. 0004H: Uncorrected HA data byte enable error. 0008H: Corrected patrol scrub error. 0010H: Uncorrected patrol scrub error. 0020H: Corrected spare error. 0040H: Uncorrected spare error. 0100H: iMC, write data buffer parity errors. 0200H: DDR4 command address parity error.
	36:32	Other Info	Reserved
	37	Reserved	Reserved
	56:38		See Chapter 16, "Machine-Check Architecture."
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.8.2 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the IA32_MC7_MISC and IA32_MC8_MISC MSRs. Table 17-27 lists model-specific error codes that apply to IA32_MCi_MISC, where i = 7, 8.

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors in the IA32_MC8_{STATUS,ADDR,MISC} registers.

Table 17-27. Intel HA MC Error Codes for IA32_MCi_MISC (i= 7, 8)

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 16-8.
8:6	Address Mode	See Table 16-3.
40:9	Reserved	Reserved
41	Failover	Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.
42	Mirrorcorr	Error was corrected by mirroring and primary channel scrubbed successfully.
63:43	Reserved	Reserved

17.9 INCREMENTAL DECODING INFORMATION: INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the Intel® Xeon® Scalable Processor Family with CPUID DisplayFamily_DisplaySignature 06_55H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-28 in Section 17.9.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

17.9.1 Internal Machine Check Errors

Table 17-28. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	
MCACOD[2]	15:0	Internal Errors	0402H: PCU internal errors. 0403H: PCU internal errors. 0406H: Intel TXT errors. 0407H: Other UBOX internal errors. On an IERR caused by a core 3-strike, the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS. After a 3-strike, the core MCA banks will be unavailable.
Model Specific Errors	19:16	Reserved, except for the following	0000b: No error. 00xxb: PCU internal error.

Table 17-28. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
	23:20	Reserved	Reserved
	31:24	Reserved, except for the following	00H: No Error 0DH: MCA_DMI_TRAINING_TIMEOUT 0FH: MCA_DMI_CPU_RESET_ACK_TIMEOUT 10H: MCA_MORE_THAN_ONE_LT_AGENT 1EH: MCA_BIOS_RST_CPL_INVALID_SEQ 1FH: MCA_BIOS_INVALID_PKG_STATE_CONFIG 25H: MCA_MESSAGE_CHANNEL_TIMEOUT 27H: MCA_MSGCH_PMREQ_CMP_TIMEOUT 30H: MCA_PKGC_DIRECT_WAKE_RING_TIMEOUT 31H: MCA_PKGC_INVALID_RSP_PCH 33H: MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN 34H: MCA_PKGC_WATCHDOG_HANG_CBZ_UP 38H: MCA_PKGC_WATCHDOG_HANG_C3_UP_SF 40H: MCA_SVID_VCCIN_VR_ICC_MAX_FAILURE 41H: MCA_SVID_COMMAND_TIMEOUT 42H: MCA_SVID_VCCIN_VR_VOUT_MAX_FAILURE 43H: MCA_SVID_CPU_VR_CAPABILITY_ERROR 44H: MCA_SVID_CRITICAL_VR_FAILED 45H: MCA_SVID_SA_ITD_ERROR 46H: MCA_SVID_READ_REG_FAILED 47H: MCA_SVID_WRITE_REG_FAILED 48H: MCA_SVID_PKGC_INIT_FAILED 49H: MCA_SVID_PKGC_CONFIG_FAILED 4AH: MCA_SVID_PKGC_REQUEST_FAILED 4BH: MCA_SVID_IMON_REQUEST_FAILED 4CH: MCA_SVID_ALERT_REQUEST_FAILED 4DH: MCA_SVID_MCP_VP_ABSENT_OR_RAMP_ERROR 4EH: MCA_SVID_UNEXPECTED_MCP_VP_DETECTED 51H: MCA_FIVR_CATAS_OVERVOL_FAULT 52H: MCA_FIVR_CATAS_OVERCUR_FAULT 58H: MCA_WATCHDG_TIMEOUT_PKGC_SECONDARY 59H: MCA_WATCHDG_TIMEOUT_PKGC_MAIN 5AH: MCA_WATCHDG_TIMEOUT_PKGS_MAIN 61H: MCA_PKGS_CPD_UNPCD_TIMEOUT 63H: MCA_PKGS_INVALID_REQ_PCH 64H: MCA_PKGS_INVALID_REQ_INTERNAL 65H: MCA_PKGS_INVALID_RSP_INTERNAL 6BH: MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT 81H: MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	52:32	Reserved	Reserved
	54:53	CORR_ERR_STATUS	Reserved

Table 17-28. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
	56:55	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

2. The internal error codes may be model-specific.

17.9.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the IA32_MC5_STATUS, IA32_MC12_STATUS, and IA32_MC19_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1PPTRRRRIILL**; see Chapter 16, "Machine-Check Architecture."

Table 17-29 lists model-specific fields to interpret error codes applicable to IA32_MCi_STATUS, i= 5, 12, 19.

Table 17-29. Interconnect MC Error Codes for IA32_MCi_STATUS (i = 5, 12, 19)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Bus error format: 1PPTRRRRIILL The two supported compound error codes: ▪ 0x0C0F: Unsupported/Undefined Packet. ▪ 0x0E0F: For all other corrected and uncorrected errors.
Model Specific Errors	21:16	MSCOD	The encoding of Uncorrectable (UC) errors are: 00H: UC Phy Initialization Failure. 01H: UC Phy detected drift buffer alarm. 02H: UC Phy detected latency buffer rollover. 10H: UC link layer Rx detected CRC error: unsuccessful LLR entered abort state. 11H: UC LL Rx unsupported or undefined packet. 12H: UC LL or Phy control error. 13H: UC LL Rx parameter exchange exception. 1FH: UC LL detected control error from the link-mesh interface. The encoding of correctable (COR) errors are: 20H: COR Phy initialization abort. 21H: COR Phy reset. 22H: COR Phy lane failure, recovery in x8 width. 23H: COR Phy L0c error corrected without Phy reset. 24H: COR Phy L0c error triggering Phy reset. 25H: COR Phy L0p exit error corrected with Phy reset. 30H: COR LL Rx detected CRC error; successful LLR without Phy re-init. 31H: COR LL Rx detected CRC error; successful LLR with Phy re-init. All other values are reserved.

Table 17-29. Interconnect MC Error Codes for IA32_MCi_STATUS (i = 5, 12, 19)

Type	Bit No.	Bit Function	Bit Description
	31:22	MSCOD_SPARE	The definition below applies to MSCOD 12h (UC LL or Phy Control Errors) [Bit 22] : Phy Control Error. [Bit 23] : Unexpected Retry.Ack flit. [Bit 24] : Unexpected Retry.Req flit. [Bit 25] : RF parity error. [Bit 26] : Routeback Table error. [Bit 27] : Unexpected Tx Protocol flit (EOP, Header or Data). [Bit 28] : Rx Header-or-Credit BGF credit overflow/underflow. [Bit 29] : Link Layer Reset still in progress when Phy enters L0 (Phy training should not be enabled until after LL reset is complete as indicated by KTILCL.LinkLayerReset going back to 0). [Bit 30] : Link Layer reset initiated while protocol traffic not idle. [Bit 31] : Link Layer Tx Parity Error.
	37:32	Reserved	Reserved
	52:38	Corrected Error Cnt	
	56:53	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.9.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC13_STATUS—IA32_MC18_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

IA32_MCi_STATUS (i=13,14,17) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCi_STATUS (i=15,16,18).

Table 17-30. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—18)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model Specific Errors	31:16	Reserved, except for the following	0001H: Address parity error. 0002H: HA write data parity error. 0004H: HA write byte enable parity error. 0008H: Corrected patrol scrub error. 0010H: Uncorrected patrol scrub error. 0020H: Corrected spare error. 0040H: Uncorrected spare error. 0080H: Any HA read error. 0100H: WDB read parity error. 0200H: DDR4 command address parity error. 0400H: Uncorrected address parity error. 0800H: Unrecognized request type. 0801H: Read response to an invalid scoreboard entry. 0802H: Unexpected read response. 0803H: DDR4 completion to an invalid scoreboard entry. 0804H: Completion to an invalid scoreboard entry. 0805H: Completion FIFO overflow. 0806H: Correctable parity error. 0807H: Uncorrectable error. 0808H: Interrupt received while outstanding interrupt was not ACKed. 0809H: ERID FIFO overflow. 080AH: Error on Write credits. 080BH: Error on Read credits. 080CH: Scheduler error. 080DH: Error event.
	36:32	Other Info	MC logs the first error device. This is an encoded 5-bit value of the device.
	37	Reserved	Reserved
	56:38		See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.9.4 M2M Machine Check Errors

MC error codes associated with M2M are reported in the IA32_MC7_STATUS and IA32_MC8_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, “Machine-Check Architecture.”

Table 17-31. M2M MC Error Codes for IA32_MCi_STATUS (i= 7, 8)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Compound error format: 0000 0000 1MMM CCCC
Model Specific Errors	16	MscodDataRdErr	Logged an MC read data error.
	17	Reserved	Reserved
	18	MscodPtlWrErr	Logged an MC partial write data error.
	19	MscodFullWrErr	Logged a full write data error.
	20	MscodBgfErr	Logged an M2M clock-domain-crossing buffer (BGF) error.
	21	MscodTimeOut	Logged an M2M time out.
	22	MscodParErr	Logged an M2M tracker parity error.
	23	MscodBucket1Err	Logged a fatal Bucket1 error.
	31:24	Reserved	Reserved
	36:32	Other Info	MC logs the first error device. This is an encoded 5-bit value of the device.
	37	Reserved	Reserved
	56:38		See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.9.5 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the IA32_MC7_MISC and IA32_MC8_MISC MSRs. Table 17-32 lists model-specific error codes that apply to IA32_MCi_MISC, where i = 7, 8.

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors in the IA32_MC8_{STATUS,ADDR,MISC} registers.

Table 17-32. Intel HA MC Error Codes for IA32_MCi_MISC (i= 7, 8)

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 16-8.
8:6	Address Mode	See Table 16-3.
40:9	Reserved	Reserved
61:41	Reserved	Reserved
62	Mirrorcorr	Error was corrected by mirroring and primary channel scrubbed successfully.
63	Failover	Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.

17.10 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_5FH, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel Atom® processors based on Goldmont Microarchitecture with CPUID DisplayFamily_DisplaySignature 06_5FH (Denverton), incremental error codes for the memory controller unit are reported in the register banks IA32_MC6 and IA32_MC7. Table 17-33 in Section 17.10.1 lists model-specific fields to interpret error codes applicable to IA32_MCi_STATUS, where i = 6, 7.

17.10.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC6_STATUS and IA32_MC7_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

Table 17-33. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 6, 7)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	
Model Specific Errors	31:16	Reserved, except for the following	01H: Cmd/Addr parity. 02H: Corrected Demand/Patrol Scrub error. 04H: Uncorrected patrol scrub error. 08H: Uncorrected demand read error. 10H: WDB read ECC.
	36:32	Other Info	
	37	Reserved	Reserved
	56:38		See Chapter 16, "Machine-Check Architecture."
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.11 INCREMENTAL DECODING INFORMATION: 3RD GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the 3rd generation Intel® Xeon® Scalable Processor Family with CPUID DisplayFamily_DisplaySignatures of 06_6AH and 06_6CH, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-34 in Section 17.11.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

17.11.1 Internal Machine Check Errors

Table 17-34. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
Machine Check Error Codes[1]	15:0	MCCOD	
MCCOD	15:0	Internal Errors	The value of this field will be 0402H for the PCU and 0406H for internal firmware errors. This applies for any logged error.
Model Specific Errors	19:16	Reserved, except for the following	Model specific error code bits 19:16. This logs the type of HW UC (PCU/VCU) error that has occurred. There are 7 errors defined. 01H: Instruction address out of valid space. 02H: Double bit RAM error on Instruction Fetch. 03H: Invalid OpCode seen. 04H: Stack Underflow. 05H: Stack Overflow. 06H: Data address out of valid space. 07H: Double bit RAM error on Data Fetch.
	23:20	Reserved, except for the following	Model specific error code bits 23:20. This logs the type of HW FSM error that has occurred. There are 3 errors defined. 04H: Clock/power IP response timeout. 05H: SMBus controller raised SMI. 09H: PM controller received invalid transaction.
	31:24	Reserved, except for the following	0DH: MCA_LLC_BIST_ACTIVE_TIMEOUT 0EH: MCA_DMI_TRAINING_TIMEOUT 0FH: MCA_DMI_STRAP_SET_ARRIVAL_TIMEOUT 10H: MCA_DMI_CPU_RESET_ACK_TIMEOUT 11H: MCA_MORE_THAN_ONE_LT_AGENT 14H: MCA_INCOMPATIBLE_PCH_TYPE 1EH: MCA_BIOS_RST_CPL_INVALID_SEQ 1FH: MCA_BIOS_INVALID_PKG_STATE_CONFIG 2DH: MCA_PCU_PMAX_CALIB_ERROR 2EH: MCA_TSC100_SYNC_TIMEOUT 3AH: MCA_GPSB_TIMEOUT 3BH: MCA_PMSB_TIMEOUT 3EH: MCA_IOSFSB_PMREQ_CMP_TIMEOUT 40H: MCA_SVID_VCCIN_VR_ICC_MAX_FAILURE 42H: MCA_SVID_VCCIN_VR_VOUT_FAILURE 43H: MCA_SVID_CPU_VR_CAPABILITY_ERROR 44H: MCA_SVID_CRITICAL_VR_FAILED 45H: MCA_SVID_SA_ITD_ERROR 46H: MCA_SVID_READ_REG_FAILED 47H: MCA_SVID_WRITE_REG_FAILED 4AH: MCA_SVID_PKGC_REQUEST_FAILED

Table 17-34. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

Type	Bit No.	Bit Function	Bit Description
			4BH: MCA_SVID_IMON_REQUEST_FAILED 4CH: MCA_SVID_ALERT_REQUEST_FAILED 4DH: MCA_SVID_MCP_VR_RAMP_ERROR 56H: MCA_FIVR_PD_HARDERR 58H: MCA_WATCHDOG_TIMEOUT_PKGC_SECONDARY 59H: MCA_WATCHDOG_TIMEOUT_PKGC_MAIN 5AH: MCA_WATCHDOG_TIMEOUT_PKGS_MAIN 5BH: MCA_WATCHDOG_TIMEOUT_MSG_CH_FSM 5CH: MCA_WATCHDOG_TIMEOUT_BULK_CR_FSM 5DH: MCA_WATCHDOG_TIMEOUT_IOSFSB_FSM 60H: MCA_PKGS_SAFE_WP_TIMEOUT 61H: MCA_PKGS_CPD_UNCPD_TIMEOUT 62H: MCA_PKGS_INVALID_REQ_PCH 63H: MCA_PKGS_INVALID_REQ_INTERNAL 64H: MCA_PKGS_INVALID_RSP_INTERNAL 65H-7AH: MCA_PKGS_RESET_PREP_TIMEOUT 7BH: MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT 7CH: MCA_PKGS_SMBUS_MCP_PAUSE_TIMEOUT 7DH: MCA_PKGS_SMBUS_SPD_PAUSE_TIMEOUT 80H: MCA_PKGC_DISP_BUSY_TIMEOUT 81H: MCA_PKGC_INVALID_RSP_PCH 83H: MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN 84H: MCA_PKGC_WATCHDOG_HANG_CBZ_UP 87H: MCA_PKGC_WATCHDOG_HANG_C2_BLKMASTER 88H: MCA_PKGC_WATCHDOG_HANG_C2_PSLIMIT 89H: MCA_PKGC_WATCHDOG_HANG_SETDISP 8BH: MCA_PKGC_ALLOW_L1_ERROR 90H: MCA_RECOVERABLE_DIE_THERMAL_TOO_HOT A0H: MCA_ADR_SIGNAL_TIMEOUT A1H: MCA_BCLK_FREQ_OC_ABOVE_THRESHOLD B0H: MCA_DISPATCHER_RUN_BUSY_TIMEOUT
	37:32	ENH_MCA_AVAIL0	Available when Enhanced MCA is in use.
	52:38	CORR_ERR_COUNT	Correctable error count.
	54:53	CORRERRORSTATUSIND	These bits are used to indicate when the number of corrected errors has exceeded the safe threshold to the point where an uncorrected error has become more likely to happen. Table 3 shows the encoding of these bits.
	56:55	ENH_MCA_AVAIL1	Available when Enhanced MCA is in use.
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.11.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the IA32_MC5_STATUS, IA32_MC7_STATUS, and IA32_MC8_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type **1PPTRRRRIILL**; see Chapter 16, "Machine-Check Architecture."

NOTE

> The interconnect machine check errors in this section apply only to the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6AH. These do not apply to the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6CH.

Table 17-35 lists model-specific fields to interpret error codes applicable to IA32_MCi_STATUS, where i= 5, 7, 8.

Table 17-35. Interconnect MC Error Codes for IA32_MCi_STATUS (i = 5, 7, 8)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Bus error format: 1PPTRRRRIILL The two supported compound error codes: ▪ 0x0C0F: Unsupported/Undefined Packet. ▪ 0x0E0F: For all other corrected and uncorrected errors.
Model Specific Errors	21:16	MSCOD	The encoding of Uncorrectable (UC) errors are: 00H: Phy Initialization Failure (NumInit). 01H: Phy Detected Drift Buffer Alarm. 02H: Phy Detected Latency Buffer Rollover. 10H: LL Rx detected CRC error: unsuccessful LLR (entered Abort state). 11H: LL Rx Unsupported/Undefined packet. 12H: LL or Phy Control Error. 13H: LL Rx Parameter Exception. 1FH: LL Detected Control Error. The encoding of correctable (COR) errors are: 20H: Phy Initialization Abort. 21H: Phy Inband Reset. 22H: Phy Lane failure, recovery in x8 width. 23H: Phy L0c error corrected without Phy reset. 24H: Phy L0c error triggering Phy reset. 25H: Phy L0p exit error corrected with reset. 30H: LL Rx detected CRC error: successful LLR without Phy Re-init. 31H: LL Rx detected CRC error: successful LLR with Phy Re-init. 32H: Tx received LLR. All other values are reserved.

Table 17-35. Interconnect MC Error Codes for IA32_MCi_STATUS (i = 5, 7, 8) (Contd.)

Type	Bit No.	Bit Function	Bit Description
	31:22	MSCOD_SPARE	The definition below applies to MSCOD 12h (UC LL or Phy Control Errors). [Bit 22] : Phy Control Error. [Bit 23] : Unexpected Retry.Ack flit. [Bit 24] : Unexpected Retry.Req flit. [Bit 25] : RF parity error. [Bit 26] : Routeback Table error. [Bit 27] : Unexpected Tx Protocol flit (EOP, Header or Data). [Bit 28] : Rx Header-or-Credit BGF credit overflow/underflow. [Bit 29] : Link Layer Reset still in progress when Phy enters L0 (Phy training should not be enabled until after LL reset is complete as indicated by KTILCL.LinkLayerReset going back to 0). [Bit 30] : Link Layer reset initiated while protocol traffic not idle. [Bit 31] : Link Layer Tx Parity Error.
	37:32	OTHER_INFO	Other Info.
	56:38	Corrected Error Cnt	See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.11.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers for the 3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture are defined in Table 17-37.

The MSRs reporting MC error codes differ depending on the CPUID DisplayFamily_DisplaySignature of the processor. See Table 17-36 for details.

Table 17-36. MSRs Reporting MC Error Codes by CPUID DisplayFamily_DisplaySignature

Processor	CPUID DisplayFamily_DisplaySignature	MSRs Reporting MC Error Codes
3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture	06_6AH	IA32_MC13_STATUS—IA32_MC14_STATUS IA32_MC17_STATUS—IA32_MC18_STATUS IA32_MC21_STATUS—IA32_MC22_STATUS IA32_MC25_STATUS—IA32_MC26_STATUS
3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture	06_6CH	IA32_MC13_STATUS—IA32_MC14_STATUS IA32_MC17_STATUS—IA32_MC18_STATUS

The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

Table 17-37. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—14, 17—18, 21—22, 25—26)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model Specific Errors	27:16	Error Codes	0000H: Uncorrectable spare error. 0001H: End to End address parity error. 0002H: Write data parity error. 0003H: End to End uncorrectable/correctable write data ECC error. 0004H: Write byte enable parity error. 0007H: Transaction ID parity error. 0008H: Correctable patrol scrub error. 0010H: Uncorrectable patrol scrub error. 0020H: Correctable spare error. 0080H: Transient or correctable error for demand or underfill reads or read 2LM metadata error. 00A0H: Uncorrectable error for demand or underfill reads. 0100H: WDB read parity error. 0108H: DDR/DDRT link failure. 0111H: PCLS address CSR parity error. 0112H: PCLS illegal ADDDC configuration error. 0200H: DDR4 command / address parity error. 0400H: RPQ scheduler address parity error. 0800H: 2LM unrecognized request type. 0801H: 2LM read response to an invalid scoreboard entry. 0802H: 2LM unexpected read response. 0803H: 2LM DDR4 completion to an invalid scoreboard entry. 0804H: 2LM DDRT completion to an invalid scoreboard entry. 0805H: 2LM completion FIFO overflow. 0806H: DDRT link parity error. 0807H: DDRT RID uncorrectable error. 0809H: DDRT RID FIFO overflow. 080AH: DDRT error on FNV write credits. 080BH: DDRT error on FNV read credits. 080CH: DDRT scheduler error. 080DH: DDRT FNV error. 080EH: DDRT FNV thermal error. 080FH: DDRT unexpected data packet during CMI idle. 0810H: DDRT RPQ request parity error. 0811H: DDRT WPQ request parity error. 0812H: 2LM NmFillWr CAM multiple hit error. 0813H: CMI credit oversubscription error. 0814H: CMI total credit count error. 0815H: CMI reserved credit pool error. 0816H: DDRT link ECC error.

Table 17-37. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—14, 17—18, 21—22, 25—26) (Contd.)

Type	Bit No.	Bit Function	Bit Description
			0817H: WDB FIFO overflow or underflow errors. 0818H: CMI request FIFO overflow error. 0819H: CMI request FIFO underflow error. 081AH: CMI response FIFO overflow error. 081BH: CMI response FIFO underflow error. 081CH: CMI miscellaneous credit errors. 081DH: CMI MC arbiter errors. 081EH: DDRT write completion FIFO overflow error. 081FH: DDRT write completion FIFO underflow error. 0820H: CMI read completion FIFO overflow error. 0821H: CMI read completion FIFO underflow error. 0822H: TME key RF parity error. 0823H: TME miscellaneous CMI errors. 0824H: TME CMI overflow error. 0825H: TME CMI underflow error. 0826H: Intel® SGX TEM secure bit mismatch detected on demand read. 0827H: TME detected underfill read completion data parity error. 0828H: 2LM Scoreboard Overflow Error. 1008H: Correctable patrol scrub error (mirror secondary example).
	28	Mirror secondary error.	Mirror secondary error.
	31:29	Reserved	Reserved
	37:32	Other Info	Other Info.
	56:38		See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Additional information is reported in the IA32_MC13_MISC—IA32_MC14_MISC, IA32_MC17_MISC—IA32_MC18_MISC, IA32_MC21_MISC—IA32_MC22_MISC, and IA32_MC25_MISC—IA32_MC26_MISC MSRs. Table 17-38 lists the information reported in IA32_MCi_MISC, where i = 13—14, 17—18, 21—22, and 25—26.

Table 17-38. Additional Information Reported in IA32_MCi_MISC (i= 13—14, 17—18, 21—22, 25—26)

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 16-8.
8:6	Address Mode	See Table 16-3.
18:9	Column	Component of sub-DIMM address. Bits 18-17: Reserved. Bit 16: Column 9. Bit 15: Column 8. Bit 14: Column 7. Bit 13: Column 6. Bit 12: Column 5. Bit 11: Column 4. Bit 10: Column 3. Bit 9: Reserved.
39:19	Row	Component of sub-DIMM address.
45:40	Bank	Component of sub-DIMM address. Bit 45: Reserved. Bit 44: Bank group 2. Bit 43: Bank address 1. Bit 42: Bank address 0. Bit 41: Bank group 1. Bit 40: Bank group 0.
51:46	Failed Device	Failing device for correctable error (not valid for uncorrectable or transient errors).
55:52	CBit	CBit
58:56	Chip Select	Chip Select
62:59	ECC Mode	0000b: SDDC 2LM. 0001b: SDDC 1LM. 0010b: SDDC + 1 2LM. 0011b: SDDC + 1 1LM. 0100b: ADDDC 2LM. 0101b: ADDDC 1LM. 0110b: ADDDC + 1 2LM. 0111b: ADDDC + 1 1LM. 1000b: Read from DDRT. 1001b: x8 SDDC. 1010b: x8 SDDC + 1. 1011b: Not a valid ECC mode. Other values: Reserved.
63	Transient	0b: 1b: Error was transient.

17.11.4 M2M Machine Check Errors

MC error codes associated with M2M for the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6AH are reported in the IA32_MC12_STATUS, IA32_MC16_STATUS, IA32_MC20_STATUS, and IA32_MC24_STATUS MSRs.

MC error codes associated with M2M for the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6CH are reported in the IA32_MC12_STATUS and IA32_MC16_STATUS MSRs.

The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

Table 17-39. M2M MC Error Codes for IA32_MCi_STATUS (i= 12, 16, 20, 24)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Compound error format: 0000 0000 1MMM CCCC
Model Specific Errors	23:16	MSCOD	Logged an MC error.
	25:24	MscodDDRType	Logged a DDR/DDRT specific error.
	26	MscodFailoverWhileResetPrep	Logged a failover specific error while preparing to reset.
	31:27	Reserved	Reserved
	37:32	Other Info	Other information.
	56:38		See Chapter 16, "Machine-Check Architecture."
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

MC error codes associated with mirrored memory corrections are reported in the IA32_MC12_MISC, IA32_MC16_MISC, IA32_MC20_MISC, and IA32_MC24_MISC MSRs. The model-specific error codes listed in Table 17-32 also apply to IA32_MCi_MISC, where i = 12, 16, 20, 24.

17.12 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_86H, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel Atom® processors based on Tremont microarchitecture with CPUID DisplayFamily_DisplaySignature 06_86H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-34 in Section 17.11.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

17.12.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC13_STATUS—IA32_MC15_STATUS. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

The IA32_MCi_STATUS MSR (where i = 13, 14, 15) contains information related to a machine check error if its VAL(valid) flag is set. Bit definitions are the same as those found in Table 17-37 "Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—14, 17—18, 21—22, 25—26)."

The IA32_MCi_MISC MSR (where i = 13, 14, 15) contains information related memory corrections. Bit definitions are the same as those found in Table 17-38 "Additional Information Reported in IA32_MCi_MISC (i= 13—14, 17—18, 21—22, 25—26)."

17.12.2 M2M Machine Check Errors

MC error codes associated with M2M are reported in the IA32_MC12_STATUS MSR. The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

Bit definitions are the same as those found in Table 17-39 "M2M MC Error Codes for IA32_MCi_STATUS (i= 12, 16, 20, 24)."

17.13 INCREMENTAL DECODING INFORMATION: 4TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the 4th generation Intel® Xeon® Scalable Processor Family with CPUID DisplayFamily_DisplaySignature of 06_8FH, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-40 in Section 17.13.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

17.13.1 Internal Machine Check Errors

Table 17-40. Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCACOD[1]	15:0	Internal Errors	The value of this field will be 0402H for the PCU and 0406H for internal firmware errors. This applies for any logged error.
Model Specific Errors	19:16	Reserved, except for the following	Model specific error code bits 19:16. If MACOD = 40CH, MSCOD encoding should be interpreted as: 01H: MCE when CR4.MCE is clear. 02H: MCE when MCIP bit is set. 03H: MCE under WPS. 04H: Unrecoverable error during security flow execution. 05H: Software triple fault shutdown. 06H: VMX-exit-consistency-check failures. 07H: RSM-consistency-check failures. 08H: Invalid conditions on protected mode SMM entry. 09H: Unrecoverable error during security flow execution. For all other MACOD values, MSCOD logs the type of hardware UC (PCU/VCU) error that has occurred. There are seven errors defined: 01H: Instruction address out of valid space. 02H: Double bit RAM error on Instruction Fetch. 03H: Invalid OpCode seen. 04H: Stack Underflow. 05H: Stack Overflow. 06H: Data address out of valid space. 07H: Double bit RAM error on Data Fetch.

Table 17-40. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

Type	Bit No.	Bit Function	Bit Description
	23:20	Reserved, except for the following	Model specific error code bits 23:20. This logs the type of HW FSM error that has occurred. There are 3 errors defined: 04H: Clock/power IP response timeout. 05H: SMBus controller raised SMI. 09H: PM controller received invalid transaction.
	31:24	Reserved, except for the following	0DH: MCA_LLC_BIST_ACTIVE_TIMEOUT 0EH: MCA_DMI_TRAINING_TIMEOUT 0FH: MCA_DMI_STRAP_SET_ARRIVAL_TIMEOUT 10H: MCA_DMI_CPU_RESET_ACK_TIMEOUT 11H: MCA_MORE_THAN_ONE_LT_AGENT 14H: MCA_INCOMPATIBLE_PCH_TYPE 1EH: MCA_BIOS_RST_CPL_INVALID_SEQ 1FH: MCA_BIOS_INVALID_PKG_STATE_CONFIG 2DH: MCA_PCU_PMAX_CALIB_ERROR 2EH: MCA_TSC100_SYNC_TIMEOUT 3AH: MCA_GPSB_TIMEOUT 3BH: MCA_PMSB_TIMEOUT 3EH: MCA_IOSFSB_PMREQ_CMP_TIMEOUT 40H: MCA_SVID_VCCIN_VR_ICC_MAX_FAILURE 42H: MCA_SVID_VCCIN_VR_VOUT_FAILURE 43H: MCA_SVID_CPU_VR_CAPABILITY_ERROR 44H: MCA_SVID_CRITICAL_VR_FAILED 45H: MCA_SVID_SA_ITD_ERROR 46H: MCA_SVID_READ_REG_FAILED 47H: MCA_SVID_WRITE_REG_FAILED 4AH: MCA_SVID_PKGC_REQUEST_FAILED 4BH: MCA_SVID_IMON_REQUEST_FAILED 4CH: MCA_SVID_ALERT_REQUEST_FAILED 4DH: MCA_SVID_MCP_VR_RAMP_ERROR 56H: MCA_FIVR_PD_HARDERR 58H: MCA_WATCHDOG_TIMEOUT_PKGC_SECONDARY 59H: MCA_WATCHDOG_TIMEOUT_PKGC_MAIN 5AH: MCA_WATCHDOG_TIMEOUT_PKGS_MAIN 5BH: MCA_WATCHDOG_TIMEOUT_MSG_CH_FSM 5CH: MCA_WATCHDOG_TIMEOUT_BULK_CR_FSM 5DH: MCA_WATCHDOG_TIMEOUT_IOSFSB_FSM 60H: MCA_PKGS_SAFE_WP_TIMEOUT 61H: MCA_PKGS_CPD_UNCPD_TIMEOUT 62H: MCA_PKGS_INVALID_REQ_PCH 63H: MCA_PKGS_INVALID_REQ_INTERNAL 64H: MCA_PKGS_INVALID_RSP_INTERNAL 65H-7AH: MCA_PKGS_RESET_PREP_TIMEOUT 7BH: MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT

Table 17-40. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

Type	Bit No.	Bit Function	Bit Description
			7CH: MCA_PKGS_SMBUS_MCP_PAUSE_TIMEOUT 7DH: MCA_PKGS_SMBUS_SPD_PAUSE_TIMEOUT 80H: MCA_PKGC_DISP_BUSY_TIMEOUT 81H: MCA_PKGC_INVALID_RSP_PCH 83H: MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN 84H: MCA_PKGC_WATCHDOG_HANG_CBZ_UP 87H: MCA_PKGC_WATCHDOG_HANG_C2_BLKMASTER 88H: MCA_PKGC_WATCHDOG_HANG_C2_PSLIMIT 89H: MCA_PKGC_WATCHDOG_HANG_SETDISP 8BH: MCA_PKGC_ALLOW_L1_ERROR 90H: MCA_RECOVERABLE_DIE_THERMAL_TOO_HOT A0H: MCA_ADR_SIGNAL_TIMEOUT A1H: MCA_BCLK_FREQ_OC_ABOVE_THRESHOLD B0H: MCA_DISPATCHER_RUN_BUSY_TIMEOUT C0H: MCA_DISPATCHER_RUN_BUSY_TIMEOUT
	37:32	ENH_MCA_AVAIL0	Available when Enhanced MCA is in use.
	52:38	CORR_ERR_COUNT	Correctable error count.
	54:53	CORRERRORSTATUSIND	These bits are used to indicate when the number of corrected errors has exceeded the safe threshold to the point where an uncorrected error has become more likely to happen. Table 3 shows the encoding of these bits.
	56:55	ENH_MCA_AVAIL1	Available when Enhanced MCA is in use.
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.13.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the IA32_MC5_STATUS MSR. The supported error codes follow the architectural MCACOD definition type **1PPTRRRRIILL**; see Chapter 16, “Machine-Check Architecture.”

Table 17-41 lists model-specific fields to interpret error codes applicable to IA32_MC5_STATUS.

Table 17-41. Interconnect MC Error Codes for IA32_MC5_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Bus error format: 1PPTRRRRIILL The two supported compound error codes: ▪ 0x0C0F: Unsupported/Undefined Packet. ▪ 0x0E0F: For all other corrected and uncorrected errors.
Model Specific Errors	21:16	MSCOD	The encoding of Uncorrectable (UC) errors are: 00H: UC Phy Initialization Failure. 01H: UC Phy Detected Drift Buffer Alarm. 02H: UC Phy Detected Latency Buffer Rollover. 10H: UC LL Rx detected CRC error: unsuccessful LLR (entered Abort state). 11H: UC LL Rx Unsupported/Undefined packet. 12H: UC LL or Phy Control Error. 13H: UC LL Rx Parameter Exception. 15H: UC LL Rx SGX MAC Error. 1FH: UC LL Detected Control Error. The encoding of correctable (COR) errors are: 20H: COR Phy Initialization Abort. 21H: COR Phy Inband Reset. 22H: COR Phy Lane failure, recovery in x8 width. 23H: COR Phy L0c error corrected without Phy reset. 24H: COR Phy L0c error triggering Phy reset. 25H: COR Phy L0p exit error corrected with reset. 30H: COR LL Rx detected CRC error: successful LLR without Phy Re-init. 31H: COR LL Rx detected CRC error: successful LLR with Phy Re-init. All other values are reserved.
	31:22	MSCOD_SPARE	The definition below applies to MSCOD 12H (UC LL or Phy Control Errors). [Bit 22]: Phy Control Error. [Bit 23]: Unexpected Retry.Ack flit. [Bit 24]: Unexpected Retry.Req flit. [Bit 25]: RF parity error. [Bit 26]: Routeback Table error. [Bit 27]: Unexpected Tx Protocol flit (EOP, Header, or Data). [Bit 28]: Rx Header-or-Credit BGF credit overflow/underflow. [Bit 29]: Link Layer Reset still in progress when Phy enters L0 (Phy training should not be enabled until after LL reset is complete as indicated by KTILCL.LinkLayerReset going back to 0). [Bit 30]: Link Layer reset initiated while protocol traffic not idle. [Bit 31]: Link Layer Tx Parity Error.
	37:32	OTHER_INFO	Other Info.
	56:38	Corrected Error Cnt	See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.13.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers for the 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture are reported in the IA32_MC13_STATUS—IA32_MC20_STATUS MSRs.

The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, "Machine-Check Architecture."

Table 17-42. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—20)

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Memory Controller error format: 0000 0000 1MMM CCCC
Model Specific Errors	31:16	Reserved, except for the following	0001H: Address parity error. 0002H: Data parity error. 0003H: Data ECC error. 0004H: Data byte enable parity error. 0007H: Transaction ID parity error. 0008H: Corrected patrol scrub error. 0010H: Uncorrected patrol scrub error. 0020H: Corrected spare error. 0040H: Uncorrected spare error. 0080H: Corrected read error. 00A0H: Uncorrected read error. 00C0H: Uncorrected metadata. 0100H: WDB read parity error. 0108H: DDR link failure. 0200H: DDR5 command / address parity error. 0400H: RPQ0 parity (primary) error. 0800H: DDR-T bad request. 0801H: DDR Data response to an invalid entry. 0802H: DDR data response to an entry not expecting data. 0803H: DDR5 completion to an invalid entry. 0804H: DDR-T completion to an invalid entry. 0805H: DDR data/completion FIFO overflow. 0806H: DDR-T ERID correctable parity error. 0807H: DDR-T ERID uncorrectable error. 0808H: DDR-T interrupt received while outstanding interrupt was not ACKed. 0809H: ERID FIFO overflow. 080AH: DDR-T error on FNV write credits. 080BH: DDR-T error on FNV read credits. 080CH: DDR-T scheduler error. 080DH: DDR-T FNV error event. 080EH: DDR-T FNV thermal event. 080FH: CMI packet while idle. 0810H: DDR_T_RPQ_REQ_PARITY_ERR. 0811H: DDR_T_WPQ_REQ_PARITY_ERR. 0812H: 2LM_NMFILLWR_CAM_ERR.

Table 17-42. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—20)

Type	Bit No.	Bit Function	Bit Description
			0813H: CMI_CREDIT_OVERSUB_ERR. 0814H: CMI_CREDIT_TOTAL_ERR. 0815H: CMI_CREDIT_RSVD_POOL_ERR. 0816H: DDR_T_RD_ERROR. 0817H: WDB_FIFO_ERR. 0818H: CMI_REQ_FIFO_OVERFLOW. 0819H: CMI_REQ_FIFO_UNDERFLOW. 081AH: CMI_RSP_FIFO_OVERFLOW. 081BH: CMI_RSP_FIFO_UNDERFLOW. 081CH: CMI_MISC_MC_CRDT_ERRORS. 081DH: CMI_MISC_MC_ARB_ERRORS. 081EH: DDR_T_WR_CMPL_FIFO_OVERFLOW. 081FH: DDR_T_WR_CMPL_FIFO_UNDERFLOW. 0820H: CMI_RD_CPL_FIFO_OVERFLOW. 0821H: CMI_RD_CPL_FIFO_UNDERFLOW. 0822H: TME_KEY_PAR_ERR. 0823H: TME_CMI_MISC_ERR. 0824H: TME_CMI_OVFL_ERR. 0825H: TME_CMI_UFL_ERR. 0826H: TME_TEM_SECURE_ERR. 0827H: TME_UFILL_PAR_ERR. 0829H: INTERNAL_ERR. 082AH: TME_INTEGRITY_ERR. 082BH: TME_TDX_ERR 082CH: TME_UFILL_TEM_SECURE_ERR. 082DH: TME_KEY_POISON_ERR. 082EH: TME_SECURITY_ENGINE_ERR. 1008H: CORR_PATSCRUB_MIRR2ND_ERR. 1010H: UC_PATSCRUB_MIRR2ND_ERR. 1020H: COR_SPARE_MIRR2ND_ERR. 1040H: UC_SPARE_MIRR2ND_ERR. 1080H: HA_RD_MIRR2ND_ERR. 10A0H: HA_UNCORR_RD_MIRR2ND_ERR.
	37:32	Other Info	Other Info.
	56:38		See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Additional information is reported in the IA32_MC13_MISC—IA32_MC20_MISC MSRs. Table 17-43 lists the information reported in IA32_MCi_MISC, where i = 13—20.

Table 17-43. Additional Information Reported in IA32_MCi_MISC (i= 13—20)

Bit No.	Bit Function	Bit Description
5:0	LSB	See Figure 16-8.
8:6	Address Mode	See Table 16-3.
18:9	Column	Column address for the last retry. To get the real column address from this field, shift the value left by 2.
36:19	Row	Component of sub-DIMM address.
42:37	Bank ID	Component of sub-DIMM address. Bit 42: Reserved. Bit 41: Bank group 2. Bit 40: Bank address 1. Bit 39: Bank address 0. Bit 38: Bank group 1. Bit 37: Bank group 0.
48:43	Failed Device	Failing device for correctable error (not valid for uncorrectable or transient errors).
50:49	Reserved	Reserved
55:51	Failed Device Number	In HBM mode, holds the failed device number for upper 32 bytes.
55:52	CBit	In DDR mode, bits 54-52: sub_rank[2:0]; bit 55: reserved.
58:56	Chip Select	Chip Select
62:59	ECC Mode	0000b: SDDC 2LM. 0001b: SDDC 1LM. 0010b: SDDC + 1 2LM. 0011b: SDDC + 1 1LM. 0100b: ADDDC 2LM. 0101b: ADDDC 1LM. 0110b: ADDDC + 1 2LM. 0111b: ADDDC + 1 1LM. 1000b: Read from DDRT. 1011b: Not a valid ECC mode. For HBM mode: 0001b: 64B read. 1001b: 32B read. Other values: Reserved.
63	Transient	Indicates if the error was a transient error. A transient error is only indicated for demand reads, underfill reads, and patrol. If there was a WDBParity Error, this field indicates the WDB ID bit 6.

17.13.4 M2M Machine Check Errors

MC error codes associated with M2M for the 4th generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_8FH are reported in the IA32_MC12_STATUS MSR.

The supported error codes follow the architectural MCACOD definition type **1MMMCCCC**; see Chapter 16, “Machine-Check Architecture.”

Table 17-44. M2M MC Error Codes for IA32_MC12_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0	MCACOD	Compound error format: 0000 0000 1MMM CCCC
Model Specific Errors	23:16	MscodDataRdErr	00H: No error (default). 01H: Read ECC error (MemSpecRd; MemRd; MemRdData; MemRdXto*; MemInv; MemInvXto*; MemInvItoX). 02H: Bucket1 error. 03H: RdTrkr Parity error. 05H: Prefetch channel mismatch. 07H: Read completion parity error. 08H: Response parity error. 09H: Timeout error. 0AH: CMI reserved credit pool error. 0BH: CMI total credit count error. 0CH: CMI credit oversubscription error.
	25:24	MscodDDRType	00: Not logged, whether error on DDR4 or DDRT. 01: HBM errors.
	31:26	Reserved	Reserved
	37:32	Other Info	Other Info.
	56:38		See Chapter 16, “Machine-Check Architecture.”
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.13.5 High Bandwidth Memory Machine Check Errors

MC error codes associated with high bandwidth memory for the 4th generation Intel Xeon Scalable Processor Family are reported in the IA32_MC29_STATUS—IA32_MC31_STATUS MSRs.

17.14 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 0FH, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-45 provides information for interpreting additional family 0FH model-specific fields for external bus errors. These errors are reported in the IA32_MCi_STATUS MSRs. They are reported architecturally as compound errors with a general form of **0000 1PPT RRRR IILL** in the MCA error code field. See Chapter 16 for information on the interpretation of compound error codes.

Table 17-45. Incremental Decoding Information: Processor Family 0FH, Machine Error Codes for Machine Check

Type	Bit No.	Bit Function	Bit Description
MCA Error Codes[1]	15:0		
Model-Specific Error Codes	16	FSB Address Parity	Address parity error detected: 1: Address parity error detected. 0: No address parity error.
	17	Response Hard Fail	Hardware failure detected on response.
	18	Response Parity	Parity error detected on response.
	19	PIC and FSB Data Parity	Data Parity detected on either PIC or FSB access.
	20	Processor Signature = 00000F04H: Invalid PIC Request	Processor Signature = 00000F04H: Indicates error due to an invalid PIC request access was made to PIC space with WB memory): 1: Invalid PIC request error. 0: No Invalid PIC request error.
		All other processors: Reserved	Reserved
	21	Pad State Machine	The state machine that tracks P and N data-strobe relative timing has become unsynchronized or a glitch has been detected.
	22	Pad Strobe Glitch	Data strobe glitch.
	23	Pad Address Glitch	Address strobe glitch.
Other Information	56:24	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Table 17-10 provides information on interpreting additional family 0FH model specific fields for cache hierarchy errors. These errors are reported in one of the IA32_MCi_STATUS MSRs. These errors are reported, architecturally, as compound errors with a general form of **0000 0001 RRRR TTLL** in the MCA error code field. See Chapter 16 for how to interpret the compound error code.

17.14.1 Model-Specific Machine Check Error Codes for the Intel® Xeon® Processor MP 7100 Series

The Intel Xeon processor MP 7100 series has five register banks which contain information related to Machine Check Errors. MCi_STATUS[63:0] refers to all five register banks. MC0_STATUS[63:0] through MC3_STATUS[63:0] is the same as previous generations of Intel Xeon processors within Family 0FH. MC4_STATUS[63:0] is the main error logging for the processor’s L3 and front side bus errors. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.

Table 17-46. MCi_STATUS Register Bit Definition

Bit Field Name	Bits	Description
MCA_Error_Code	15:0	This field specifies the machine check architecture defined error code for the machine check error condition detected. The machine check architecture defined error codes are guaranteed to be the same for all Intel Architecture processors that implement the machine check architecture. See tables below.
Model_Specific_Error_Code	31:16	This field specifies the model specific error code that uniquely identifies the machine check error condition detected. The model specific error codes may differ among Intel Architecture processors for the same Machine Check Error condition. See tables below.
Other_Info	56:32	The functions of the bits in this field are implementation specific and are not part of the machine check architecture. Software that is intended to be portable among Intel Architecture processors should not rely on the values in this field.
PCC	57	The Processor Context Corrupt flag indicates that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor's state. This bit will always be set for MC errors, which are not corrected.
ADDRV	58	The MC_ADDR register valid flag indicates that the MC_ADDR register contains the address where the error occurred. When clear, this flag indicates that the MC_ADDR register does not contain the address where the error occurred. The MC_ADDR register should not be read if the ADDRV bit is clear.
MISCV	59	The MC_MISC register valid flag indicates that the MC_MISC register contains additional information regarding the error. When clear, this flag indicates that the MC_MISC register does not contain additional information regarding the error. MC_MISC should not be read if the MISCV bit is not set.
EN	60	The error enabled flag indicates that reporting of the machine check exception for this error was enabled by the associated flag bit of the MC_CTL register. Note that correctable errors do not have associated enable bits in the MC_CTL register so the EN bit should be clear when a correctable error is logged.
UC	61	The error uncorrected flag indicates that the processor did not correct the error condition. When clear, this flag indicates that the processor was able to correct the event condition.
OVER	62	The machine check overflow flag indicates that a machine check error occurred while the results of a previous error were still in the register bank (i.e., the VAL bit was already set in the MC_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. Enabled errors are written over disabled errors, and uncorrected errors are written over corrected events. Uncorrected errors are not written over previous valid uncorrected errors.
VAL	63	The MC_STATUS register valid flag indicates that the information within the MC_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the MC_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

17.14.1.1 Processor Machine Check Status Register MCA Error Code Definition

The Intel Xeon processor MP 7100 series uses compound MCA Error Codes for logging its CBC internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines additional Machine Check error types (IA32_MC4_STATUS[15:0]) beyond those defined in Chapter 16. Table 17-47 lists these model-specific MCA error codes. Error code details are specified in MC4_STATUS [31:16]; see Section 17.14.3, the "Model Specific Error Code" field. The information in the "Other_Info" field (MC4_STATUS[56:32]) is common to the three processor error types and contains a correctable event count and specifies the MC4_MISC register format.

Table 17-47. Incremental MCA Error Code for Intel® Xeon® Processor MP 7100

Processor MCA_Error_Code (MC4_STATUS[15:0])			
Type	**Error Code**	**Binary Encoding**	**Meaning**
C	Internal Error	0000 0100 0000 0000	Internal Error Type Code.
A	L3 Tag Error	0000 0001 0000 1011	L3 Tag Error Type Code.
B	Bus and Interconnect Error	0000 100x 0000 1111	Not used, but this encoding is reserved for compatibility with other MCA implementations.
		0000 101x 0000 1111	Not used, but this encoding is reserved for compatibility with other MCA implementations.
		0000 110x 0000 1111	Not used, but this encoding is reserved for compatibility with other MCA implementations.
		0000 1110 0000 1111	Bus and Interconnection Error Type Code.
		0000 1111 0000 1111	Not used, but this encoding is reserved for compatibility with other MCA implementations.

The **bold faced** binary encodings are the only encodings used by the processor for MC4_STATUS[15:0].

17.14.2 Other_Info Field (All MCA Error Types)

The MC4_STATUS[56:32] field is common to the processor's three MCA error types (A, B, and C).

Table 17-48. Other Information Field Bit Definition

Bit Field Name	Bits	Description
39:32	8-bit Correctable Event Count	This field holds a count of the number of correctable events since cold reset. This is a saturating counter; the counter begins at 1 (with the first error) and saturates at a count of 255.
41:40	MC4_MISC Format Type	The value in this field specifies the format of information in the MC4_MISC register. Currently, only two values are defined. Valid only when MISCV is asserted.
43:42	Reserved	Reserved
51:44	ECC Syndrome	ECC syndrome value for a correctable ECC event when the “Valid ECC syndrome” bit is asserted.
52	Valid ECC Syndrome	Set when a correctable ECC event supplies the ECC syndrome.
54:53	Threshold-Based Error Status	00: No tracking. No hardware status tracking is provided for the structure reporting this event. 01: Green. Status tracking is provided for the structure posting the event; the current status is green (below threshold). 10: Yellow. Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). 11: Reserved for future use. Valid only if the Valid bit (bit 63) is set. Undefined if the UC bit (bit 61) is set.
56:55	Reserved	Reserved

17.14.3 Processor Model Specific Error Code Field

17.14.3.1 MCA Error Type A: L3 Error

Note: The Model Specific Error Code field in MC4_STATUS (bits 31:16).

Table 17-49. Type A: L3 Error Codes

Bit Num	Sub-Field Name	Description	Legal Value(s)
18:16	L3 Error Code	Describes the L3 error encountered	000: No error. 001: More than one way reporting a correctable event. 010: More than one way reporting an uncorrectable error. 011: More than one way reporting a tag hit. 100: No error. 101: One way reporting a correctable event. 110: One way reporting an uncorrectable error. 111: One or more ways reporting a correctable event while one or more ways are reporting an uncorrectable error.
20:19	---	Reserved	00
31:21	---	Fixed pattern	0010_0000_000

17.14.3.2 Processor Model Specific Error Code Field Type B: Bus and Interconnect Error

Note: The Model Specific Error Code field in MC4_STATUS (bits 31:16).

Table 17-50. Type B: Bus and Interconnect Error Codes

Bit Num	Sub-Field Name	Description
16	FSB Request Parity	Parity error detected during FSB request phase.
17	Core0 Addr Parity	Parity error detected on Core 0 request's address field.
18	Core1 Addr Parity	Parity error detected on Core 1 request's address field.
19	Reserved	Reserved
20	FSB Response Parity	Parity error on FSB response field detected.
21	FSB Data Parity	FSB data parity error on inbound data detected.
22	Core0 Data Parity	Data parity error on data received from Core 0 detected.
23	Core1 Data Parity	Data parity error on data received from Core 1 detected.
24	IDS Parity	Detected an Enhanced Defer parity error (phase A or phase B).
25	FSB Inbound Data ECC	Data ECC event to error on inbound data (correctable or uncorrectable).
26	FSB Data Glitch	Pad logic detected a data strobe 'glitch' (or sequencing error).
27	FSB Address Glitch	Pad logic detected a request strobe 'glitch' (or sequencing error).
31:28	Reserved	Reserved

Exactly one of the bits defined in the preceding table will be set for a Bus and Interconnect Error. The Data ECC can be correctable or uncorrectable; the MC4_STATUS.UC bit distinguishes between correctable and uncorrectable cases with the Other_Info field possibly providing the ECC Syndrome for correctable errors. All other errors for this processor MCA Error Type are uncorrectable.

17.14.3.3 Processor Model Specific Error Code Field Type C: Cache Bus Controller Error

Table 17-51. Type C: Cache Bus Controller Error Codes

MC4_STATUS[31:16] (MSCE) Value	Error Description
0000_0000_0000_0001 0001H	Inclusion Error from Core 0.
0000_0000_0000_0010 0002H	Inclusion Error from Core 1.
0000_0000_0000_0011 0003H	Write Exclusive Error from Core 0.
0000_0000_0000_0100 0004H	Write Exclusive Error from Core 1.
0000_0000_0000_0101 0005H	Inclusion Error from FSB.
0000_0000_0000_0110 0006H	SNP Stall Error from FSB.
0000_0000_0000_0111 0007H	Write Stall Error from FSB.
0000_0000_0000_1000 0008H	FSB Arb Timeout Error.
0000_0000_0000_1001 0009H	CBC OOD Queue Underflow/overflow.
0000_0001_0000_0000 0100H	Enhanced Intel SpeedStep Technology TM1-TM2 Error.
0000_0010_0000_0000 0200H	Internal Timeout Error.
0000_0011_0000_0000 0300H	Internal Timeout Error.
0000_0100_0000_0000 0400H	Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow.
1100_0000_0000_0001 C001H	Correctable ECC event on outgoing FSB data.
1100_0000_0000_0010 C002H	Correctable ECC event on outgoing Core 0 data.
1100_0000_0000_0100 C004H	Correctable ECC event on outgoing Core 1 data.
1110_0000_0000_0001 E001H	Uncorrectable ECC error on outgoing FSB data.
1110_0000_0000_0010 E002H	Uncorrectable ECC error on outgoing Core 0 data.
1110_0000_0000_0100 E004H	Uncorrectable ECC error on outgoing Core 1 data.
— All other encodings —	Reserved

All errors, except for the correctable ECC types, in this table are uncorrectable. The correctable ECC events may supply the ECC syndrome in the Other_Info field of the MC4_STATUS MSR.

Table 17-52. Decoding Family 0FH Machine Check Codes for Cache Hierarchy Errors

Type	Bit No.	Bit Function	Bit Description
MCA error codes[1]	15:0		
Model Specific Error Codes	17:16	Tag Error Code	Contains the tag error code for this machine check error: 00: No error detected. 01: Parity error on tag miss with a clean line. 10: Parity error/multiple tag match on tag hit. 11: Parity error/multiple tag match on tag miss.
	19:18	Data Error Code	Contains the data error code for this machine check error: 00: No error detected. 01: Single bit error. 10: Double bit error on a clean line. 11: Double bit error on a modified line.
	20	L3 Error	This bit is set if the machine check error originated in the L3 (it can be ignored for invalid PIC request errors): 1: L3 error. 0: L2 error.
	21	Invalid PIC Request	Indicates error due to invalid PIC request access was made to PIC space with WB memory: 1: Invalid PIC request error. 0: No invalid PIC request error.
	31:22	Reserved	Reserved
Other Information	39:32	8-bit Error Count	Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at a count of 255.
	56:40	Reserved	Reserved
Status Register Validity Indicators[1]	63:57		

NOTES:

[1] These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

CHAPTER 18
DEBUG, BRANCH PROFILE, TSC, AND INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FEATURES

NOTE

This chapter makes numerous references to last-branch recording (LBR) facilities. Unless noted otherwise, all such references in this chapter are to an earlier non-architectural form of the feature. Chapter 19 defines an architectural form of last-branch recording that is supported on newer processors.

Intel 64 and IA-32 architectures provide debug facilities for use in debugging code and monitoring performance. These facilities are valuable for debugging application software, system software, and multitasking operating systems. Debug support is accessed using debug registers (DR0 through DR7) and model-specific registers (MSRs):

- Debug registers hold the addresses of memory and I/O locations called breakpoints. Breakpoints are user-selected locations in a program, a data-storage area in memory, or specific I/O ports. They are set where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. A debug exception (#DB) is generated when a memory or I/O access is made to a breakpoint address.
- MSRs monitor branches, interrupts, and exceptions; they record addresses of the last branch, interrupt or exception taken and the last branch taken before an interrupt or exception.
- Time stamp counter is described in Section 18.17, "Time-Stamp Counter."
- Features that allow monitoring of shared platform resources such as the L3 cache are described in Section 18.18, "Intel® Resource Director Technology (Intel® RDT) Monitoring Features."
- Features that enable control over shared platform resources are described in Section 18.19, "Intel® Resource Director Technology (Intel® RDT) Allocation Features."
- Features that enable control over shared platform resources for non-CPU agents are described in Section 18.20, "Intel® Resource Director Technology (Intel® RDT) for Non-CPU Agents."[1]

18.1 OVERVIEW OF DEBUG SUPPORT FACILITIES

The following processor facilities support debugging and performance monitoring:

- **Debug exception (#DB) —** Transfers program control to a debug procedure or task when a debug event occurs.
- **Breakpoint exception (#BP) —** See breakpoint instruction (INT3) below.
- **Breakpoint-address registers (DR0 through DR3) —** Specifies the addresses of up to 4 breakpoints.
- **Debug status register (DR6) —** Reports the conditions that were in effect when a debug or breakpoint exception was generated.
- **Debug control register (DR7) —** Specifies the forms of memory or I/O access that cause breakpoints to be generated.
- **T (trap) flag, TSS —** Generates a debug exception (#DB) when an attempt is made to switch to a task with the T flag set in its TSS.
- **RF (resume) flag, EFLAGS register —** Suppresses multiple exceptions to the same instruction.
- **TF (trap) flag, EFLAGS register —** Generates a debug exception (#DB) after every execution of an instruction.

[1] Additional information about Intel® RDT can be found in the document titled "Intel® Resource Director Technology Architecture Specification," available here: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

- **Breakpoint instruction (INT3) —** Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set instruction breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.
- **Last branch recording facilities** — Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address. Send branch records out on the system bus as branch trace messages (BTMs).

These facilities allow a debugger to be called as a separate task or as a procedure in the context of the current program or task. The following conditions can be used to invoke the debugger:

- Task switch to a specific task.
- Execution of the breakpoint instruction.
- Execution of any instruction.
- Execution of an instruction at a specified address.
- Read or write to a specified memory address/range.
- Write to a specified memory address/range.
- Input from a specified I/O address/range.
- Output to a specified I/O address/range.
- Attempt to change the contents of a debug register.

18.2 DEBUG REGISTERS

Eight debug registers (see Figure 18-1 for 32-bit operation and Figure 18-2 for 64-bit operation) control the debug operation of the processor. These registers can be written to and read using the move to/from debug register form of the MOV instruction. A debug register may be the source or destination operand for one of these instructions.

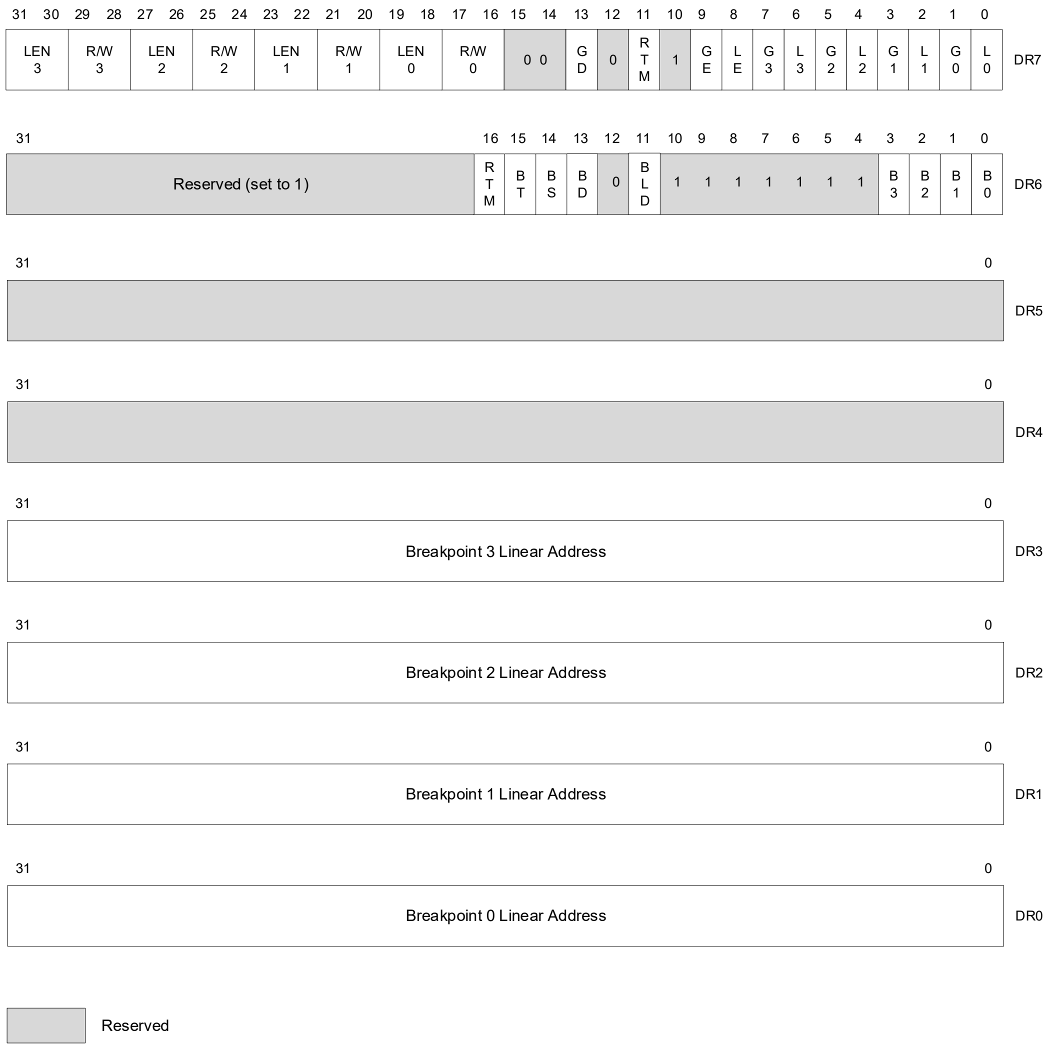


Figure 18-1. Debug Registers

Debug registers are privileged resources; a MOV instruction that accesses these registers can only be executed in real-address mode, in SMM or in protected mode at a CPL of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).

The primary function of the debug registers is to set up and monitor from 1 to 4 breakpoints, numbered 0 though 3. For each breakpoint, the following information can be specified:

- The linear address where the breakpoint is to occur.
- The length of the breakpoint location: 1, 2, 4, or 8 bytes (refer to the notes in Section 18.2.4).
- The operation that must be performed at the address for a debug exception to be generated.
- Whether the breakpoint is enabled.
- Whether the breakpoint condition was present when the debug exception was generated.

The following paragraphs describe the functions of flags and fields in the debug registers.

18.2.1 Debug Address Registers (DR0-DR3)

Each of the debug-address registers (DR0 through DR3) holds the 32-bit linear address of a breakpoint (see Figure 18-1). Breakpoint comparisons are made before physical address translation occurs. The contents of debug register DR7 further specifies breakpoint conditions.

18.2.2 Debug Registers DR4 and DR5

Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD). When debug extensions are not enabled (when the DE flag is clear), these registers are aliased to debug registers DR6 and DR7.

18.2.3 Debug Status Register (DR6)

The debug status register (DR6) reports debug conditions that were sampled at the time the last debug exception was generated (see Figure 18-1). Updates to this register only occur when an exception is generated. The flags in this register show the following information:

- **B0 through B3 (breakpoint condition detected) flags (bits 0 through 3) —** Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the LEN*n*, and R/W*n* flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the Ln or the Gn flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.
- **BLD (bus-lock detected) flag (bit 11) —** Indicates (when **clear**) that the debug exception was triggered by the assertion of a bus lock when CPL > 0 and OS bus-lock detection was enabled (see Section 18.3.1.6). Other debug exceptions do not modify this bit. To avoid confusion in identifying debug exceptions, software debug-exception handlers should set bit 11 to 1 before returning. (Software that never enables OS bus-lock detection need not do this as DR6[11] = 1 following reset.) This bit is always 1 if the processor does not support OS bus-lock detection.
- **BD (debug register access detected) flag (bit 13) —** Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set. See Section 18.2.4, "Debug Control Register (DR7)," for further explanation of the purpose of this flag.
- **BS (single step) flag (bit 14) —** Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.
- **BT (task switch) flag (bit 15) —** Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 8.2.1, "Task-State Segment (TSS)," for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.
- **RTM (restricted transactional memory) flag (bit 16)** — Indicates (when **clear**) that a debug exception (#DB) or breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 18.3.3). This bit is set for any other debug exception (including all those that occur when advanced debugging of RTM transactional regions is not enabled). This bit is always 1 if the processor does not support RTM.

Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register (except bit 16, which they should set) before returning to the interrupted task.

18.2.4 Debug Control Register (DR7)

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions (see Figure 18-1). The flags and fields in this register control the following things:

- **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6) —** Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated L*n* flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.
- **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7) —** Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated G*n* flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.
- **LE and GE (local and global exact breakpoint enable) flags (bits 8, 9) —** This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.
- **RTM (restricted transactional memory) flag (bit 11)** — Enables (when set) advanced debugging of RTM transactional regions (see Section 18.3.3). This advanced debugging is enabled only if IA32_DEBUGCTL.RTM is also set.
- **GD (general detect enable) flag (bit 13) —** Enables (when set) debug-register protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators.

  When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor.

  The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.
- **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29) —** Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the R/W*n* fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Break on I/O reads or writes.
  11 — Break on data reads or writes but not instruction fetches.

  When the DE flag is clear, the processor interprets the R/W*n* bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Undefined.
  11 — Break on data reads or writes but not instruction fetches.
- **LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31) —** Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:

  00 — 1-byte length.
  01 — 2-byte length.
  10 — Undefined (or 8 byte length, see note below).
  11 — 4-byte length.

If the corresponding RW*n* field in register DR7 is 00 (instruction execution), then the LEN*n* field should also be 00. The effect of using other lengths is undefined. See Section 18.2.5, “Breakpoint Field Recognition,” below.

NOTES

For Pentium® 4 and Intel® Xeon® processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), break point conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the LEN*n* field.

Encoding 10B is also supported in processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture, the respective CPUID signatures corresponding to family 6, model 15, and family 6, DisplayModel value 23 (see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). The Encoding 10B is supported in processors based on Intel Atom® microarchitecture, with CPUID signature of family 6, DisplayModel value 1CH. The encoding 10B is undefined for other processors.

18.2.5 Breakpoint Field Recognition

Breakpoint address registers (debug registers DR0 through DR3) and the LEN*n* fields for each breakpoint define a range of sequential byte addresses for a data or I/O breakpoint. The LEN*n* fields permit specification of a 1-, 2-, 4- or 8-byte range, beginning at the linear address specified in the corresponding debug register (DR*n*). Two-byte ranges must be aligned on word boundaries; 4-byte ranges must be aligned on doubleword boundaries, 8-byte ranges must be aligned on quadword boundaries. I/O addresses are zero-extended (from 16 to 32 bits, for comparison with the breakpoint address in the selected debug register). These requirements are enforced by the processor; it uses LEN*n* field bits to mask the lower address bits in the debug registers. Unaligned data or I/O breakpoint addresses do not yield valid results.

A data breakpoint for reading or writing data is triggered if any of the bytes participating in an access is within the range defined by a breakpoint address register and its LEN*n* field. Table 18-1 provides an example setup of debug registers and data accesses that would subsequently trap or not trap on the breakpoints.

A data breakpoint for an unaligned operand can be constructed using two breakpoints, where each breakpoint is byte-aligned and the two breakpoints together cover the operand. The breakpoints generate exceptions only for the operand, not for neighboring bytes.

Instruction breakpoint addresses must have a length specification of 1 byte (the LEN*n* field is set to 00). Instruction breakpoints for other operand sizes are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has prefixes, the breakpoint address must point to the first prefix.

Table 18-1. Breakpoint Examples

<table>
<tr><th colspan="4">Debug Register Setup</th></tr>
<tr><th>Debug Register</th><th>R/W<i>n</i></th><th>Breakpoint Address</th><th>LEN<i>n</i></th></tr>
<tr><td>DR0</td><td>R/W0 = 11 (Read/Write)</td><td>A0001H</td><td>LEN0 = 00 (1 byte)</td></tr>
<tr><td>DR1</td><td>R/W1 = 01 (Write)</td><td>A0002H</td><td>LEN1 = 00 (1 byte)</td></tr>
<tr><td>DR2</td><td>R/W2 = 11 (Read/Write)</td><td>B0002H</td><td>LEN2 = 01) (2 bytes)</td></tr>
<tr><td>DR3</td><td>R/W3 = 01 (Write)</td><td>C0000H</td><td>LEN3 = 11 (4 bytes)</td></tr>
<tr><th colspan="4">Data Accesses</th></tr>
<tr><th colspan="2">Operation</th><th>Address</th><th>Access Length (In Bytes)</th></tr>
<tr><td colspan="2">Data operations that trap</td><td></td><td></td></tr>
<tr><td colspan="2">- Read or write</td><td>A0001H</td><td>1</td></tr>
<tr><td colspan="2">- Read or write</td><td>A0001H</td><td>2</td></tr>
<tr><td colspan="2">- Write</td><td>A0002H</td><td>1</td></tr>
<tr><td colspan="2">- Write</td><td>A0002H</td><td>2</td></tr>
<tr><td colspan="2">- Read or write</td><td>B0001H</td><td>4</td></tr>
<tr><td colspan="2">- Read or write</td><td>B0002H</td><td>1</td></tr>
<tr><td colspan="2">- Read or write</td><td>B0002H</td><td>2</td></tr>
<tr><td colspan="2">- Write</td><td>C0000H</td><td>4</td></tr>
<tr><td colspan="2">- Write</td><td>C0001H</td><td>2</td></tr>
<tr><td colspan="2">- Write</td><td>C0003H</td><td>1</td></tr>
</table>

Table 18-1. Breakpoint Examples (Contd.)

Debug Register Setup			
Debug Register	**R/W*n***	**Breakpoint Address**	**LEN*n***
Data operations that do not trap			
- Read or write		A0000H	1
- Read		A0002H	1
- Read or write		A0003H	4
- Read or write		B0000H	2
- Read		C0000H	2
- Read or write		C0004H	4

18.2.6 Debug Registers and Intel® 64 Processors

For Intel 64 architecture processors, debug registers DR0–DR7 are 64 bits. In 16-bit or 32-bit modes (protected mode and compatibility mode), writes to a debug register fill the upper 32 bits with zeros. Reads from a debug register return the lower 32 bits. In 64-bit mode, MOV DRn instructions read or write all 64 bits. Operand-size prefixes are ignored.

In 64-bit mode, the upper 32 bits of DR6 and DR7 are reserved and must be written with zeros. Writing 1 to any of the upper 32 bits results in a #GP(0) exception (see Figure 18-2). All 64 bits of DR0–DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0–DR3 are in the linear-address limits of the processor implementation (address matching is supported only on valid addresses generated by the processor implementation). Breakpoint conditions for 8-byte memory read/writes are supported in all modes.

18.3 DEBUG EXCEPTIONS

The Intel 64 and IA-32 architectures dedicate two interrupt vectors to handling debug exceptions: vector 1 (debug exception, #DB) and vector 3 (breakpoint exception, #BP). The following sections describe how these exceptions are generated and typical exception handler operations.

18.3.1 Debug Exception (#DB)—Interrupt Vector 1

The debug-exception handler is usually a debugger program or part of a larger software system. The processor generates a debug exception for any of several conditions. The debugger checks flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions might apply. Table 18-2 shows the states of these flags following the generation of each kind of breakpoint condition.

Instruction-breakpoint and general-detect condition (see Section 18.3.1.3, “General-Detect Exception Condition”) result in faults; other debug-exception conditions result in traps. The debug exception may report one or both at one time. The following sections describe each class of debug exception.

The INT1 instruction generates a debug exception as a trap. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

See also: Chapter 6, “Interrupt 1—Debug Exception (#DB),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

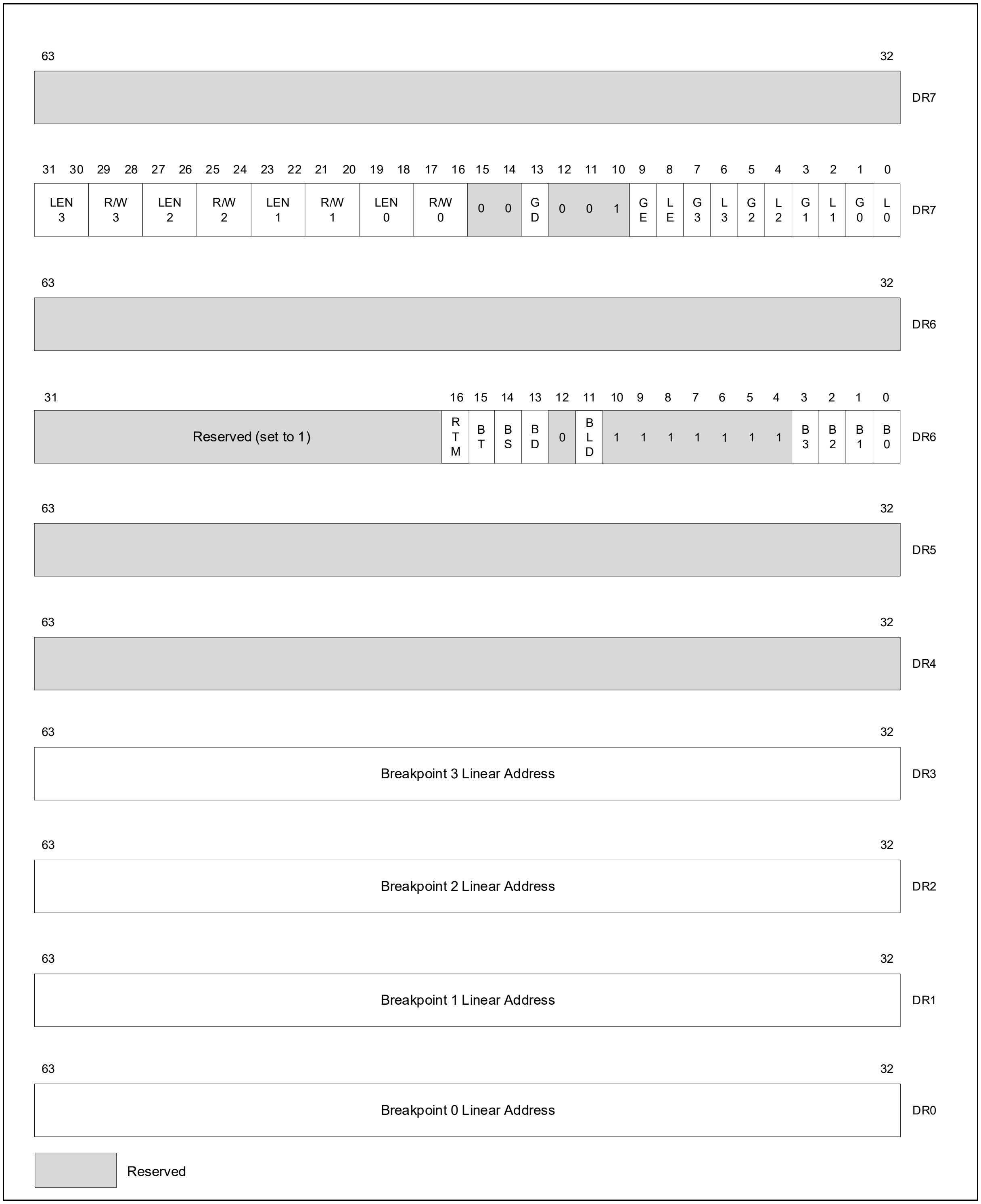


Figure 18-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture

Table 18-2. Debug Exception Conditions

Debug or Breakpoint Condition	DR6 Flags Tested	DR7 Flags Tested	Exception Class
Single-step trap	BS = 1		Trap
Instruction breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 0	Fault
Data write breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 1	Trap
I/O read or write breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 2	Trap
Data read or write (but not instruction fetches), at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 3	Trap
General detect fault, resulting from an attempt to modify debug registers (usually in conjunction with in-circuit emulation)	BD = 1	None	Fault
Task switch	BT = 1	None	Trap
INT1 instruction	None	None	Trap

18.3.1.1 Instruction-Breakpoint Exception Condition

The processor reports an instruction breakpoint when it attempts to execute an instruction at an address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect instruction execution (R/W flag is set to 0). Upon reporting the instruction breakpoint, the processor generates a fault-class, debug exception (#DB) before it executes the target instruction for the breakpoint.

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. However, if an instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint will be suppressed as if EFLAGS.RF were 1 (see the next paragraph and Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

Because the debug exception for an instruction breakpoint is generated before the instruction is executed, if the instruction breakpoint is not removed by the exception handler; the processor will detect the instruction breakpoint again when the instruction is restarted and generate another debug exception. To prevent looping on an instruction breakpoint, the Intel 64 and IA-32 architectures provide the RF flag (resume flag) in the EFLAGS register (see Section 2.3, “System Flags and Fields in the EFLAGS Register,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). When the RF flag is set, the processor ignores instruction breakpoints.

All Intel 64 and IA-32 processors manage the RF flag as follows. The RF Flag is cleared at the start of the instruction after the check for instruction breakpoints, CS limit violations, and FP exceptions. Task Switches and IRETD/IRETQ instructions transfer the RF image from the TSS/stack to the EFLAGS register.

When calling an event handler, Intel 64 and IA-32 processors establish the value of the RF flag in the EFLAGS image pushed on the stack:

- For any fault-class exception except a debug exception generated in response to an instruction breakpoint, the value pushed for RF is 1.
- For any interrupt arriving after any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For any trap-class exception generated by any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For other cases, the value pushed for RF is the value that was in EFLAG.RF at the time the event handler was called. This includes:
  - — Debug exceptions generated in response to instruction breakpoints
  - — Hardware-generated interrupts arriving between instructions (including those arriving after the last iteration of a repeated string instruction)

— Trap-class exceptions generated after an instruction completes (including those generated after the last iteration of a repeated string instruction)

— Software-generated interrupts (RF is pushed as 0, since it was cleared at the start of the software interrupt)

As noted above, the processor does not set the RF flag prior to calling the debug exception handler for debug exceptions resulting from instruction breakpoints. The debug exception handler can prevent recurrence of the instruction breakpoint by setting the RF flag in the EFLAGS image on the stack. If the RF flag in the EFLAGS image is set when the processor returns from the exception handler, it is copied into the RF flag in the EFLAGS register by IRETD/IRETQ or a task switch that causes the return. The processor then ignores instruction breakpoints for the duration of the next instruction. (Note that the POPF, POPFD, and IRET instructions do not transfer the RF image into the EFLAGS register.) Setting the RF flag does not prevent other types of debug-exception conditions (such as, I/O or data breakpoints) from being detected, nor does it prevent non-debug exceptions from being generated.

For the Pentium processor, when an instruction breakpoint coincides with another fault-type exception (such as a page fault), the processor may generate one spurious debug exception after the second exception has been handled, even though the debug exception handler set the RF flag in the EFLAGS image. To prevent a spurious exception with Pentium processors, all fault-class exception handlers should set the RF flag in the EFLAGS image.

18.3.1.2 Data Memory and I/O Breakpoint Exception Conditions

Data memory and I/O breakpoints are reported when the processor attempts to access a memory or I/O address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect data or I/O accesses (R/W flag is set to 1, 2, or 3). The processor generates the exception after it executes the instruction that made the access, so these breakpoint condition causes a trap-class exception to be generated.

Because data breakpoints are traps, an instruction that writes memory overwrites the original data before the debug exception generated by a data breakpoint is generated. If a debugger needs to save the contents of a write breakpoint location, it should save the original contents before setting the breakpoint. The handler can report the saved value after the breakpoint is triggered. The address in the debug registers can be used to locate the new value stored by the instruction that triggered the breakpoint.

If a data breakpoint is detected during an iteration of a string instruction executed with fast-string operation (see Section 7.3.9.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1), delivery of the resulting debug exception may be delayed until completion of the corresponding group of iterations.

Intel486 and later processors ignore the GE and LE flags in DR7. In Intel386 processors, exact data breakpoint matching does not occur unless it is enabled by setting the LE and/or the GE flags.

For repeated INS and OUTS instructions that generate an I/O-breakpoint debug exception, the processor generates the exception after the completion of the first iteration. Repeated INS and OUTS instructions generate a data-breakpoint debug exception after the iteration in which the memory address breakpoint location is accessed.

If an execution of the MOV or POP instruction loads the SS register and encounters a data breakpoint, the resulting debug exception is delivered after completion of the next instruction (the one after the MOV or POP).

Any pending data or I/O breakpoints are lost upon delivery of an exception. For example, if a machine-check exception (#MC) occurs following an instruction that encounters a data breakpoint (but before the resulting debug exception is delivered), the data breakpoint is lost. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, the data breakpoint is lost if the next instruction causes a fault.

Delivery of events due to INT *n*, INT3, or INTO does not cause a loss of data breakpoints. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, and the next instruction is software interrupt (INT *n*, INT3, or INTO), a debug exception (#DB) resulting from a data breakpoint will be delivered after the transition to the software-interrupt handler. The #DB handler should account for the fact that the #DB may have been delivered after a invocation of a software-interrupt handler, and in particular that the CPL may have changed between recognition of the data breakpoint and delivery of the #DB.

18.3.1.3 General-Detect Exception Condition

When the GD flag in DR7 is set, the general-detect debug exception occurs when a program attempts to access any of the debug registers (DR0 through DR7) at the same time they are being used by another application, such as an emulator or debugger. This protection feature guarantees full control over the debug registers when required. The debug exception handler can detect this condition by checking the state of the BD flag in the DR6 register. The processor generates the exception before it executes the MOV instruction that accesses a debug register, which causes a fault-class exception to be generated.

18.3.1.4 Single-Step Exception Condition

The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. The exception is a trap-class exception, because the exception is generated after the instruction is executed. The processor will not generate this exception after the instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction that follows the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The TF flag normally is not cleared by privilege changes inside a task. The INT *n*, INT3, and INTO instructions, however, do clear this flag. Therefore, software debuggers that single-step code must recognize and emulate INT *n* or INTO instructions rather than executing them directly. To maintain protection, the operating system should check the CPL after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that, if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This operation clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT *n* instruction that calls the interrupt handler.

If an occurrence of the MOV or POP instruction loads the SS register executes with EFLAGS.TF = 1, no single-step debug exception occurs following the MOV or POP instruction.

18.3.1.5 Task-Switch Exception Condition

The processor generates a debug exception after a task switch if the T flag of the new task's TSS is set. This exception is generated after program control has passed to the new task, and prior to the execution of the first instruction of that task. The exception handler can detect this condition by examining the BT flag of the DR6 register.

If entry 1 (#DB) in the IDT is a task gate, the T bit of the corresponding TSS should not be set. Failure to observe this rule will put the processor in a loop.

18.3.1.6 OS Bus-Lock Detection

OS bus-lock detection is a feature that causes the processor to generate a debug exception (called a **bus-lock detection debug exception**) if it detects that a bus lock has been asserted (see Section 9.1.2). Such an exception is a trap-class exception, because it is generated after execution of an instruction that asserts a bus lock. The exception thus does not prevent assertion of the bus lock. Delivery of a bus-lock detection debug exception clears DR6.BLD.

Software can enable OS bus-lock detection by setting IA32_DEBUGCTL.BLD[bit 2]. Bus-lock detection debug exceptions occur only if CPL > 0.

18.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3

The breakpoint exception (interrupt 3) is caused by execution of an INT3 instruction. See Chapter 6, “Interrupt 3—Breakpoint Exception (#BP).” Debuggers use breakpoint exceptions in the same way that they use the breakpoint registers; that is, as a mechanism for suspending program execution to examine registers and memory locations. With earlier IA-32 processors, breakpoint exceptions are used extensively for setting instruction breakpoints.

With the Intel386 and later IA-32 processors, it is more convenient to set breakpoints with the breakpoint-address registers (DR0 through DR3). However, the breakpoint exception still is useful for breakpointing debuggers, because a breakpoint exception can call a separate exception handler. The breakpoint exception is also useful when it is necessary to set more breakpoints than there are debug registers or when breakpoints are being placed in the source code of a program under development.

18.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)

Chapter 16, "Programming with Intel® Transactional Synchronization Extensions," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, describes Restricted Transactional Memory (RTM). This is an instruction-set interface that allows software to identify **transactional regions** (or critical sections) using the XBEGIN and XEND instructions.

Execution of an RTM transactional region begins with an XBEGIN instruction. If execution of the region successfully reaches an XEND instruction, the processor ensures that all memory operations performed within the region appear to have occurred instantaneously when viewed from other logical processors. Execution of an RTM transaction region does not succeed if the processor cannot commit the updates atomically. When this happens, the processor rolls back the execution, a process referred to as a **transactional abort**. In this case, the processor discards all updates performed in the region, restores architectural state to appear as if the execution had not occurred, and resumes execution at a fallback instruction address that was specified with the XBEGIN instruction.

If debug exception (#DB) or breakpoint exception (#BP) occurs within an RTM transaction region, a transactional abort occurs, the processor sets EAX[4], and no exception is delivered.

Software can enable **advanced debugging of RTM transactional regions** by setting DR7.RTM[bit 11] and IA32_DEBUGCTL.RTM[bit 15]. If these bits are both set, the transactional abort caused by a #DB or #BP within an RTM transaction region does **not** resume execution at the fallback instruction address specified with the XBEGIN instruction that begin the region. Instead, execution is resumed at that XBEGIN instruction, and a #DB is delivered. (A #DB is delivered even if the transactional abort was caused by a #BP.) Such a #DB will clear DR6.RTM[bit 16] (all other debug exceptions set DR6[16]).

18.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW

P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel Atom® processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory.

See the following sections for processor specific implementation of last branch, interrupt, and exception recording:

- Section 18.5, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel Atom® Processors)."
- Section 18.6, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture."
- Section 18.9, "Last Branch, Interrupt, and Exception Recording for Processors based on Nehalem Microarchitecture."
- Section 18.10, "Last Branch, Interrupt, and Exception Recording for Processors based on Sandy Bridge Microarchitecture."
- Section 18.11, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Haswell Microarchitecture."
- Section 18.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
- Section 18.14, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ Solo and Intel® Core™ Duo Processors)."
- Section 18.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."
- Section 18.16, "Last Branch, Interrupt, and Exception Recording (P6 Family Processors)."

The following subsections of Section 18.4 describe common features of profiling branches. These features are generally enabled using the IA32_DEBUGCTL MSR (older processor may have implemented a subset or model-specific features, see definitions of MSR_DEBUGCTLA, MSR_DEBUGCTLB, MSR_DEBUGCTL).

18.4.1 IA32_DEBUGCTL MSR

The **IA32_DEBUGCTL** MSR provides bit field controls to enable debug trace interrupts, debug trace stores, trace messages enable, single stepping on branches, last branch record recording, and to control freezing of LBR stack or performance counters on a PMI request. IA32_DEBUGCTL MSR is located at register address 01D9H.

See Figure 18-3 for the MSR layout and the bullets below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the Section 18.5.1, "LBR Stack" (Intel® Core™2 Duo and Intel Atom® processor family) and Section 18.9.1, "LBR Stack" (processors based on Nehalem microarchitecture).
- **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, "Single-Stepping on Branches," for more information about the BTF flag.
- **BLD (bus-lock detection) flag (bit 2) —** If this bit is set, OS bus-lock detection is enabled when CPL > 0. See Section 18.3.1.6.
- **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, "Branch Trace Messages," for more information about the TR flag.
- **BTS (branch trace store) flag (bit 7) —** When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.9, "BTS and DS Save Area."
- **BTINT (branch trace interrupt) flag (bit 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

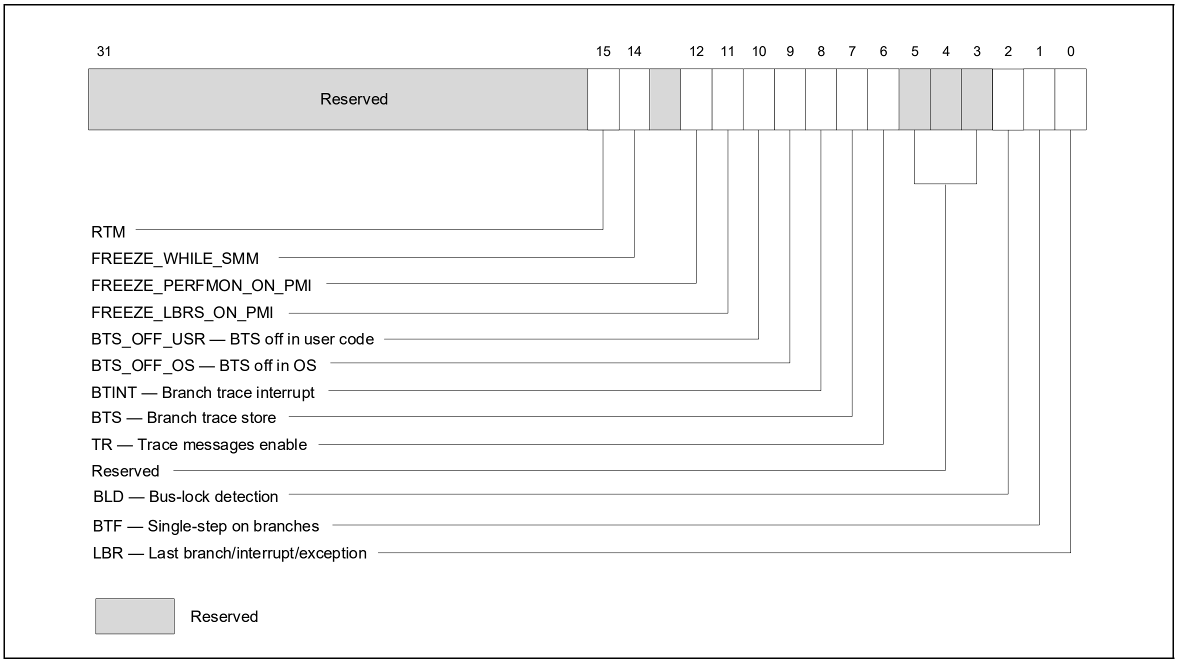

31
15
14
12
11
10
9
8
7
6
5
4
3
2
1
0
Reserved
RTM
FREEZE_WHILE_SMM
FREEZE_PERFMON_ON_PMI
FREEZE_LBRS_ON_PMI
BTS_OFF_USR — BTS off in user code
BTS_OFF_OS — BTS off in OS
BTINT — Branch trace interrupt
BTS — Branch trace store
TR — Trace messages enable
Reserved
BLD — Bus-lock detection
BTF — Single-step on branches
LBR — Last branch/interrupt/exception
Reserved

Figure 18-3. IA32_DEBUGCTL MSR for Processors Based on Intel® Core™ Microarchitecture

- **BTS_OFF_OS (branch trace off in privileged code) flag (bit 9) —** When set, BTS or BTM is skipped if CPL is 0. See Section 18.13.2.
- **BTS_OFF_USR (branch trace off in user code) flag (bit 10) —** When set, BTS or BTM is skipped if CPL is greater than 0. See Section 18.13.2.
- **FREEZE_LBRS_ON_PMI flag (bit 11) —** When set, the LBR stack is frozen on a hardware PMI request (e.g., when a counter overflows and is configured to trigger PMI). See Section 18.4.7 for details.
- **FREEZE_PERFMON_ON_PMI flag (bit 12) —** When set, the performance counters (IA32_PMCx and IA32_FIXED_CTRx) are frozen on a PMI request. See Section 18.4.7 for details.
- **FREEZE_WHILE_SMM (bit 14) —** If this bit is set, upon the delivery of an SMI, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler. If Intel Thread Director support was enabled before transferring control to the SMI handler, then the processor will also reset the Intel Thread Director history (see Section 15.6.11 for more details about Intel Thread Director enable, reset, and history reset operations).

  Subsequently, the enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its service. If Intel Thread Director support is enabled when RSM is executed, then the processor resets the Intel Thread Director history.

  Note that system software must check if the processor supports the IA32_DEBUGCTL.FREEZE_WHILE_SMM control bit. IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 20.8 for details of detecting the presence of IA32_PERF_CAPABILITIES MSR.
- **RTM (bit 15)** — If this bit is set, advanced debugging of RTM transactional regions is enabled if DR7.RTM is also set. See Section 18.3.3.

18.4.2 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag (bit 0) in the IA32_DEBUGCTL MSR is set, the processor automatically begins recording branch records for taken branches, interrupts, and exceptions (except for debug exceptions) in the LBR stack MSRs.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler. This action does not clear previously stored LBR stack MSRs.

A debugger can use the linear addresses in the LBR stack to re-set breakpoints in the breakpoint address registers (DR0 through DR3). This allows a backward trace from the manifestation of a particular bug toward its source.

On some processors, if the LBR flag is cleared and TR flag in the IA32_DEBUGCTL MSR remains set, the processor will continue to update LBR stack MSRs. This is because those processors use the entries in the LBR stack in the process of generating BTM/BTS records. A #DB does not automatically clear the TR flag.

18.4.3 Single-Stepping on Branches

When software sets both the BTF flag (bit 1) in the IA32_DEBUGCTL MSR and the TF flag in the EFLAGS register, the processor generates a single-step debug exception only after instructions that cause a branch.[1] This mechanism allows a debugger to single-step on control transfers caused by branches. This “branch single stepping” helps isolate a bug to a particular block of code before instruction single-stepping further narrows the search. The processor clears the BTF flag when it generates a debug exception. The debugger must set the BTF flag before resuming program execution to continue single-stepping on branches.

[1] Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 8.2.1, “Task-State Segment (TSS).”

18.4.4 Branch Trace Messages

Setting the TR flag (bit 6) in the IA32_DEBUGCTL MSR enables branch trace messages (BTMs). Thereafter, when the processor detects a branch, exception, or interrupt, it sends a branch record out on the system bus as a BTM. A debugging device that is monitoring the system bus can read these messages and synchronize operations with taken branch, interrupt, and exception events.

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 18.4.2, "Monitoring Branches, Exceptions, and Interrupts."

For the P6 processor family, Pentium M processor family, and processors based on Intel Core microarchitecture, TR and LBR bits can not be set at the same time due to hardware limitation. The content of LBR stack is undefined when TR is set.

For processors with Intel NetBurst microarchitecture, Intel Atom processors, and Intel Core and related Intel Xeon processors both starting with the Nehalem microarchitecture, the processor can collect branch records in the LBR stack and at the same time send/store BTMs when both the TR and LBR flags are set in the IA32_DEBUGCTL MSR (or the equivalent MSR_DEBUGCTLA, MSR_DEBUGCTLB).

The following exception applies:

- BTM may not be observable on Intel Atom processor families that do not provide an externally visible system bus (i.e., processors based on the Silvermont microarchitecture or later).

18.4.4.1 Branch Trace Message Visibility

Branch trace message (BTM) visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.

18.4.5 Branch Trace Store (BTS)

A trace of taken branches, interrupts, and exceptions is useful for debugging code by providing a method of determining the decision path taken to reach a particular code location. The LBR flag (bit 0) of IA32_DEBUGCTL provides a mechanism for capturing records of taken branches, interrupts, and exceptions and saving them in the last branch record (LBR) stack MSRs, setting the TR flag for sending them out onto the system bus as BTMs. The branch trace store (BTS) mechanism provides the additional capability of saving the branch records in a memory-resident BTS buffer, which is part of the DS save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved. The BTINT flag (bit 8) can be used to enable the generation of interrupt when the BTS buffer is full. See Section 18.4.9.2, "Setting Up the DS Save Area," for additional details.

Setting this flag (BTS) alone can greatly reduce the performance of the processor. CPL-qualified branch trace storing mechanism can help mitigate the performance impact of sending/logging branch trace messages.

18.4.6 CPL-Qualified Branch Trace Mechanism

CPL-qualified branch trace mechanism is available to a subset of Intel 64 and IA-32 processors that support the branch trace storing mechanism. The processor supports the CPL-qualified branch trace mechanism if CPUID.01H:ECX[bit 4] = 1.

The CPL-qualified branch trace mechanism is described in Section 18.4.9.4. System software can selectively specify CPL qualification to not send/store Branch Trace Messages associated with a specified privilege level. Two bit fields, BTS_OFF_USR (bit 10) and BTS_OFF_OS (bit 9), are provided in the debug control register to specify the CPL of BTMs that will not be logged in the BTS buffer or sent on the bus.

18.4.7 Freezing LBR and Performance Counters on PMI

Many issues may generate a performance monitoring interrupt (PMI); a PMI service handler will need to determine cause to handle the situation. Two capabilities that allow a PMI service routine to improve branch tracing and performance monitoring are available for processors supporting architectural performance monitoring version 2 or greater (i.e., CPUID.0AH:EAX[7:0] > 1). These capabilities provides the following interface in IA32_DEBUGCTL to reduce runtime overhead of PMI servicing, profiler-contributed skew effects on analysis or counter metrics:

- **Freezing LBRs on PMI (bit 11)**— Allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - Legacy Freeze_LBR_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the LBR is frozen on the overflowed condition of the buffer area, the processor clears the LBR bit (bit 0) in IA32_DEBUGCTL. Software must then re-enable IA32_DEBUGCTL.LBR to resume recording branches. When using this feature, software should be careful about writes to IA32_DEBUGCTL to avoid re-enabling LBRs by accident if they were just disabled.
  - Streamlined Freeze_LBR_on_PMI is supported for ArchPerfMonVerID >= 4. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the processor behaves as follows:
    - sets IA32_PERF_GLOBAL_STATUS.LBR_Frz =1 to disable recording, but does not change the LBR bit (bit 0) in IA32_DEBUGCTL. The LBRs are frozen on the overflowed condition of the buffer area.
- **Freezing PMCs on PMI** (bit 12) — Allows the PMI service routine to ensure the content in the performance counters are associated with the target workload and not polluted by the PMI and activities within the PMI service routine. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - Legacy Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_Perfmon_On_PMI = 1, the performance counters are frozen on the counter overflowed condition when the processor clears the IA32_PERF_GLOBAL_CTRL MSR (see Figure 20-3). The PMCs affected include both general-purpose counters and fixed-function counters (see Section 20.6.2.1, “Fixed-function Performance Counters”). Software must re-enable counts by writing 1s to the corresponding enable bits in IA32_PERF_GLOBAL_CTRL before leaving a PMI service routine to continue counter operation.
  - Streamlined Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID >= 4. The processor behaves as follows:
    - sets IA32_PERF_GLOBAL_STATUS.CTR_Frz =1 to disable counting on a counter overflow condition, but does not change the IA32_PERF_GLOBAL_CTRL MSR.

Freezing LBRs and PMCs on PMIs (both legacy and streamlined operation) occur when one of the following applies:

- A performance counter had an overflow and was programmed to signal a PMI in case of an overflow.
  - For the general-purpose counters; enabling PMI is done by setting bit 20 of the IA32_PERFEVTSELx register.
  - For the fixed-function counters; enabling PMI is done by setting the 3rd bit in the corresponding 4-bit control field of the MSR_PERF_FIXED_CTR_CTRL register (see Figure 20-1) or IA32_FIXED_CTR_CTRL MSR (see Figure 20-2).
- The PEBS buffer is almost full and reaches the interrupt threshold.
- The BTS buffer is almost full and reaches the interrupt threshold.

Table 18-3 compares the interaction of the processor with the PMI handler using the legacy versus streamlined Freeza_Perfmon_On_PMI interface.

Table 18-3. Legacy and Streamlined Operation with Freeze_Perfmon_On_PMI = 1, Counter Overflowed

Legacy Freeze_Perfmon_On_PMI	Streamlined Freeze_Perfmon_On_PMI	Comment
Processor freezes the counters on overflow	Processor freezes the counters on overflow	Unchanged
Processor clears IA32_PERF_GLOBAL_CTRL	Processor set IA32_PERF_GLOBAL_STATUS.CTR_FTZ	
Handler reads IA32_PERF_GLOBAL_STATUS(0x38E) to examine which counter(s) overflowed	**mask** = RDMSR(0x38E)	Similar
Handler services the PMI	Handler services the PMI	Unchanged
Handler writes 1s to IA32_PERF_GLOBAL_OVF_CTL (0x390)	Handler writes **mask** into IA32_PERF_GLOBAL_OVF_RESET (0x390)	
Processor clears IA32_PERF_GLOBAL_STATUS	Processor clears IA32_PERF_GLOBAL_STATUS	Unchanged
Handler re-enables IA32_PERF_GLOBAL_CTRL	None	Reduced software overhead

18.4.8 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel 64 and IA-32 processor families. However, the number of MSRs in the LBR stack and the valid range of TOS pointer value can vary between different processor families. Table 18-4 lists the LBR stack size and TOS pointer range for several processor families according to the CPUID signatures of DisplayFamily_DisplayModel encoding (see the CPUID instruction in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A).

Table 18-4. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Component of an LBR Entry	Range of TOS Pointer
06_5CH, 06_5FH	32	FROM_IP, TO_IP	0 to 31
06_4EH, 06_5EH, 06_8EH, 06_9EH, 06_55H, 06_66H, 06_7AH, 06_67H, 06_6AH, 06_6CH, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_A5H, 06_A6H, 06_A7H, 06_A8H, 06_86H, 06_8AH, 06_96H, 06_9CH	32	FROM_IP, TO_IP, LBR_INFO[1]	0 to 31
06_3DH, 06_47H, 06_4FH, 06_56H, 06_3CH, 06_45H, 06_46H, 06_3FH, 06_2AH, 06_2DH, 06_3AH, 06_3EH, 06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	16	FROM_IP, TO_IP	0 to 15
06_17H, 06_1DH, 06_0FH	4	FROM_IP, TO_IP	0 to 3
06_37H, 06_4AH, 06_4CH, 06_4DH, 06_5AH, 06_5DH, 06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	8	FROM_IP, TO_IP	0 to 7

NOTES:

[1] See Section 18.12.

The last branch recording mechanism tracks not only branch instructions (e.g., JMP, Jcc, LOOP, and CALL instructions), but also other operations that cause a change in the instruction pointer (e.g., external interrupts, traps, and faults). The branch recording mechanisms generally employs a set of MSRs, referred to as last branch record (LBR) stack. The size and exact locations of the LBR stack are generally model-specific (see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for model-specific MSR addresses).

- **Last Branch Record (LBR) Stack —** The LBR consists of N pairs of MSRs (N is listed in the LBR stack size column of Table 18-4) that store source and destination address of recent branches (see Figure 18-3):
  - MSR_LASTBRANCH_0_FROM_IP (address is model specific) through the next consecutive (N-1) MSR address store source addresses.
  - MSR_LASTBRANCH_0_TO_IP (address is model specific) through the next consecutive (N-1) MSR address store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant M bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address is model specific) contains an M-bit pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. The valid range of the M-bit POS pointer is given in Table 18-4.

18.4.8.1 LBR Stack and Intel® 64 Processors

LBR MSRs are 64-bits. In 64-bit mode, last branch records store the full address. Outside of 64-bit mode, the upper 32-bits of branch addresses will be stored as 0.

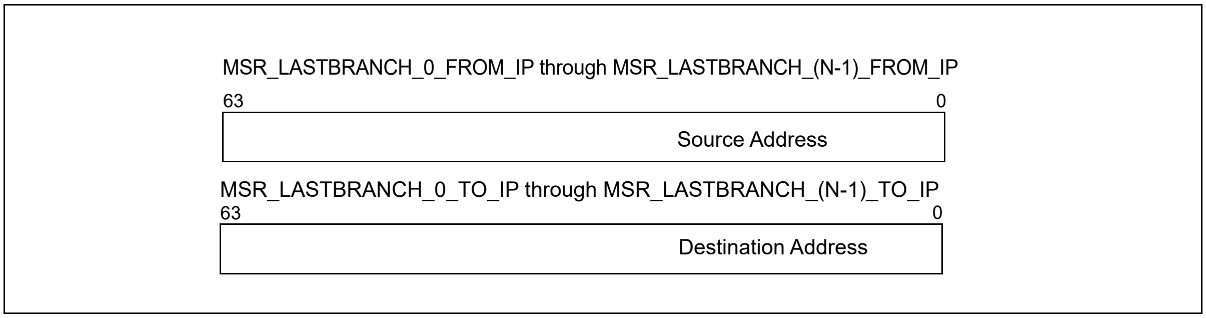

MSR_LASTBRANCH_0_FROM_IP through MSR_LASTBRANCH_(N-1)_FROM_IP
63
0
Source Address
MSR_LASTBRANCH_0_TO_IP through MSR_LASTBRANCH_(N-1)_TO_IP
63
0
Destination Address

Figure 18-4. 64-bit Address Layout of LBR MSR

Software should query an architectural MSR IA32_PERF_CAPABILITIES[5:0] about the format of the address that is stored in the LBR stack. Four formats are defined by the following encoding:

- **000000B (32-bit record format) —** Stores 32-bit offset in current CS of respective source/destination,
- **000001B (64-bit LIP record format) —** Stores 64-bit linear address of respective source/destination,
- **000010B (64-bit EIP record format) —** Stores 64-bit offset (effective address) of respective source/destination.
- **000011B (64-bit EIP record format) and Flags** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction info is reported in the upper bit of 'FROM' registers in the LBR stack. See LBR stack details below for flag support and definition.
- **000100B (64-bit EIP record format), Flags, and TSX** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction and TSX info are reported in the upper bits of 'FROM' registers in the LBR stack.
- **000101B (64-bit EIP record format), Flags, TSX, and LBR_INFO** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction, TSX, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.
- **000110B (64-bit LIP record format), Flags, and Cycles** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction info is reported in the upper bits of

'FROM' registers in the LBR stack. Elapsed cycles since the last LBR update are reported in the upper 16 bits of the 'TO' registers in the LBR stack (see Section 18.6).

- **000111B (64-bit LIP record format), Flags, and LBR_INFO** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.

Processor's support for the architectural MSR IA32_PERF_CAPABILITIES is provided by CPUID.01H:ECX[PERF_CAPAB_MSR] (bit 15).

18.4.8.2 LBR Stack and IA-32 Processors

The LBR MSRs in IA-32 processors introduced prior to Intel 64 architecture store the 32-bit "To Linear Address" and "From Linear Address" using the high and low half of each 64-bit MSR.

18.4.8.3 Last Exception Records and Intel 64 Architecture

Intel 64 and IA-32 processors also provide MSRs that store the branch record for the last branch taken prior to an exception or an interrupt. The location of the last exception record (LER) MSRs are model specific. The MSRs that store last exception records are 64-bits. If IA-32e mode is disabled, only the lower 32-bits of the address is recorded. If IA-32e mode is enabled, the processor writes 64-bit values into the MSR. In 64-bit mode, last exception records store 64-bit addresses; in compatibility mode, the upper 32-bits of last exception records are cleared.

18.4.9 BTS and DS Save Area

The **Debug store (DS)** feature flag (bit 21), returned by CPUID.1:EDX[21] indicates that the processor provides the debug store (DS) mechanism. The DS mechanism allows:

- BTMs to be stored in a memory-resident BTS buffer. See Section 18.4.5, "Branch Trace Store (BTS)."
- Processor event-based sampling (PEBS) also uses the DS save area provided by debug store mechanism. The capability of PEBS varies across different microarchitectures. See Section 20.6.2.4, "Processor Event Based Sampling (PEBS)," and the relevant PEBS sub-sections across the core PMU sections in Chapter 20, "Performance Monitoring."

When CPUID.1:EDX[21] is set:

- The BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags in the IA32_MISC_ENABLE MSR indicate (when clear) the availability of the BTS and PEBS facilities, including the ability to set the BTS and BTINT bits in the appropriate DEBUGCTL MSR.
- The IA32_DS_AREA MSR exists and points to the DS save area.

The debug store (DS) save area is a software-designated area of memory that is used to collect the following two types of information:

- **Branch records —** When the BTS flag in the IA32_DEBUGCTL MSR is set, a branch record is stored in the BTS buffer in the DS save area whenever a taken branch, interrupt, or exception is detected.
- **PEBS records —** When a performance counter is configured for PEBS, a PEBS record is stored in the PEBS buffer in the DS save area after the counter overflow occurs. This record contains the architectural state of the processor (state of the 8 general purpose registers, EIP register, and EFLAGS register) at the next occurrence of the PEBS event that caused the counter to overflow. When the state information has been logged, the counter is automatically reset to a specified value, and event counting begins again. The content layout of a PEBS record varies across different implementations that support PEBS. See Section 20.6.2.4.2 for details of enumerating PEBS record format.

NOTES

Prior to processors based on the Goldmont microarchitecture, PEBS facility only supports a subset of implementation-specific precise events. See Section 20.5.3.1 for a PEBS enhancement that can generate records for both precise and non-precise events.

The DS save area and recording mechanism are disabled on INIT, processor Reset or transition to system-management mode (SMM) or IA-32e mode. It is similarly disabled on the generation of a machine-check exception on 45nm and 32nm Intel Atom processors and on processors with Netburst or Intel Core microarchitecture.

The BTS and PEBS facilities may not be available on all processors. The availability of these facilities is indicated by the BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags, respectively, in the IA32_-MISC_ENABLE MSR (see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4).

The DS save area is divided into three parts: buffer management area, branch trace store (BTS) buffer, and PEBS buffer (see Figure 18-5). The buffer management area is used to define the location and size of the BTS and PEBS buffers. The processor then uses the buffer management area to keep track of the branch and/or PEBS records in their respective buffers and to record the performance counter reset value. The linear address of the first byte of the DS buffer management area is specified with the IA32_DS_AREA MSR.

The fields in the buffer management area are as follows:

- **BTS buffer base —** Linear address of the first byte of the BTS buffer. This address should point to a natural doubleword boundary.
- **BTS index —** Linear address of the first byte of the next BTS record to be written to. Initially, this address should be the same as the address in the BTS buffer base field.
- **BTS absolute maximum —** Linear address of the next byte past the end of the BTS buffer. This address should be a multiple of the BTS record size (12 bytes) plus 1.
- **BTS interrupt threshold —** Linear address of the BTS record on which an interrupt is to be generated. This address must point to an offset from the BTS buffer base that is a multiple of the BTS record size. Also, it must be several records short of the BTS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the BTS absolute maximum record.
- **PEBS buffer base —** Linear address of the first byte of the PEBS buffer. This address should point to a natural doubleword boundary.
- **PEBS index —** Linear address of the first byte of the next PEBS record to be written to. Initially, this address should be the same as the address in the PEBS buffer base field.
- **PEBS absolute maximum —** Linear address of the next byte past the end of the PEBS buffer. This address should be a multiple of the PEBS record size (40 bytes) plus 1.
- **PEBS interrupt threshold —** Linear address of the PEBS record on which an interrupt is to be generated. This address must point to an offset from the PEBS buffer base that is a multiple of the PEBS record size. Also, it must be several records short of the PEBS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the PEBS absolute maximum record.
- **PEBS counter reset value —** A 64-bit value that the counter is to be set to when a PEBS record is written. Bits beyond the size of the counter are ignored. This value allows state information to be collected regularly every time the specified number of events occur.

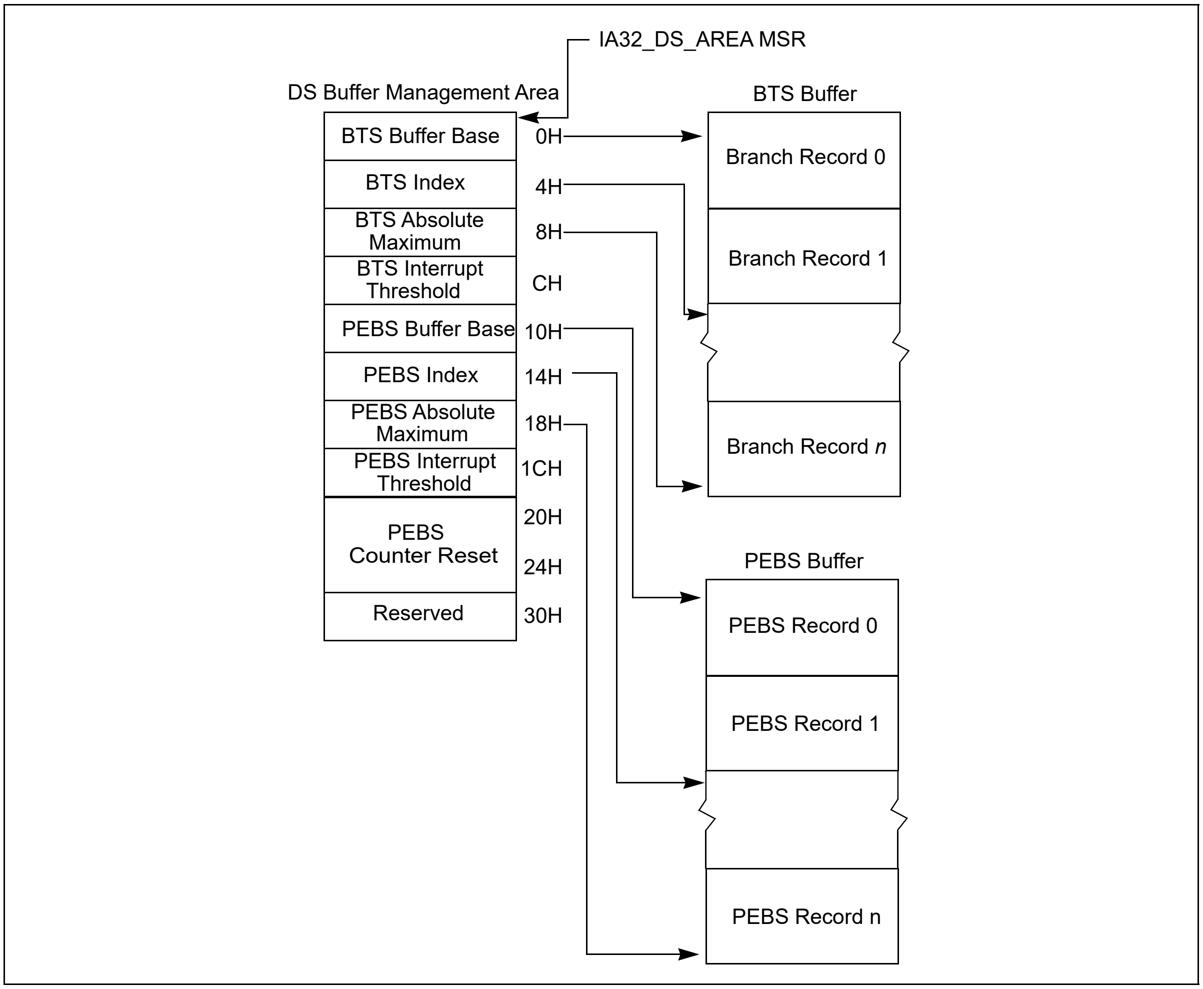


Figure 18-5. DS Save Area Example[1]

NOTES:

1. This example represents the format for a system that supports PEBS on only one counter.

Figure 18-6 shows the structure of a 12-byte branch record in the BTS buffer. The fields in each record are as follows:

- **Last branch from —** Linear address of the instruction from which the branch, interrupt, or exception was taken.
- **Last branch to —** Linear address of the branch target or the first instruction in the interrupt or exception service routine.
- **Branch predicted —** Bit 4 of field indicates whether the branch that was taken was predicted (set) or not predicted (clear).

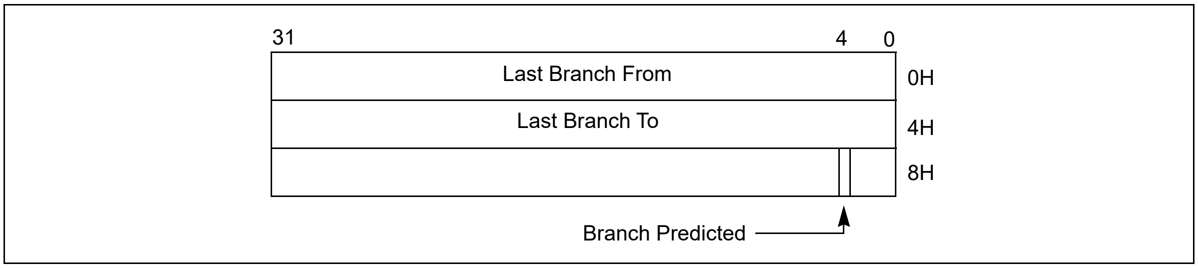


Figure 18-6. 32-bit Branch Trace Record Format

Figure 18-7 shows the structure of the 40-byte PEBS records. Nominally the register values are those at the beginning of the instruction that caused the event. However, there are cases where the registers may be logged in a partially modified state. The linear IP field shows the value in the EIP register translated from an offset into the current code segment to a linear address.

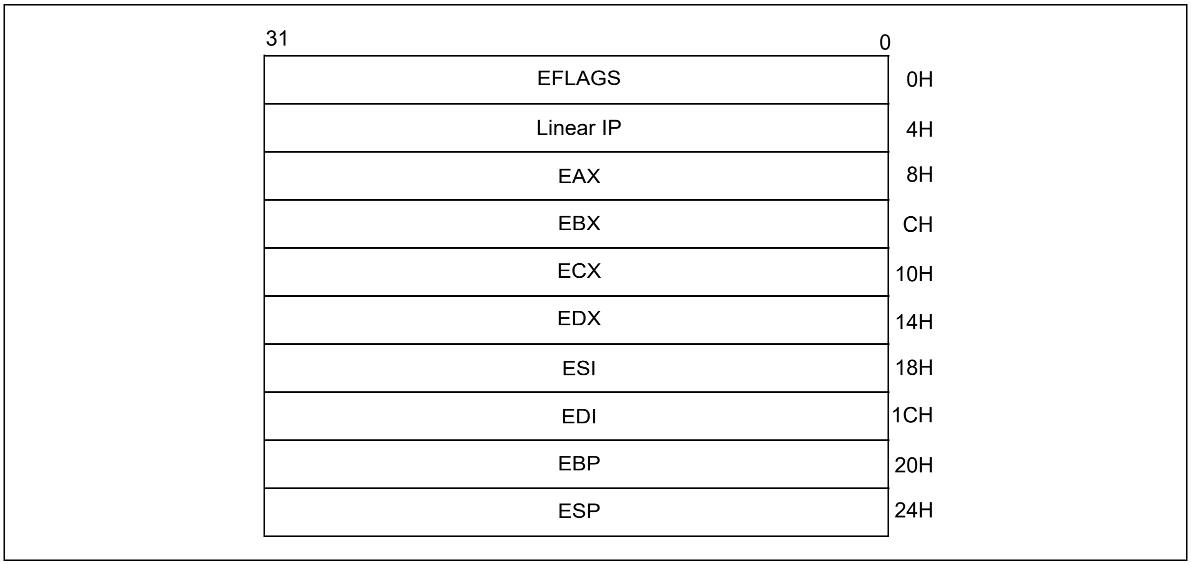


Figure 18-7. PEBS Record Format

18.4.9.1 64 Bit Format of the DS Save Area

When DTES64 = 1 (CPUID.1.ECX[2] = 1), the structure of the DS save area is shown in Figure 18-8.

When DTES64 = 0 (CPUID.1.ECX[2] = 0) and IA-32e mode is active, the structure of the DS save area is shown in Figure 18-8. If IA-32e mode is not active the structure of the DS save area is as shown in Figure 18-5.

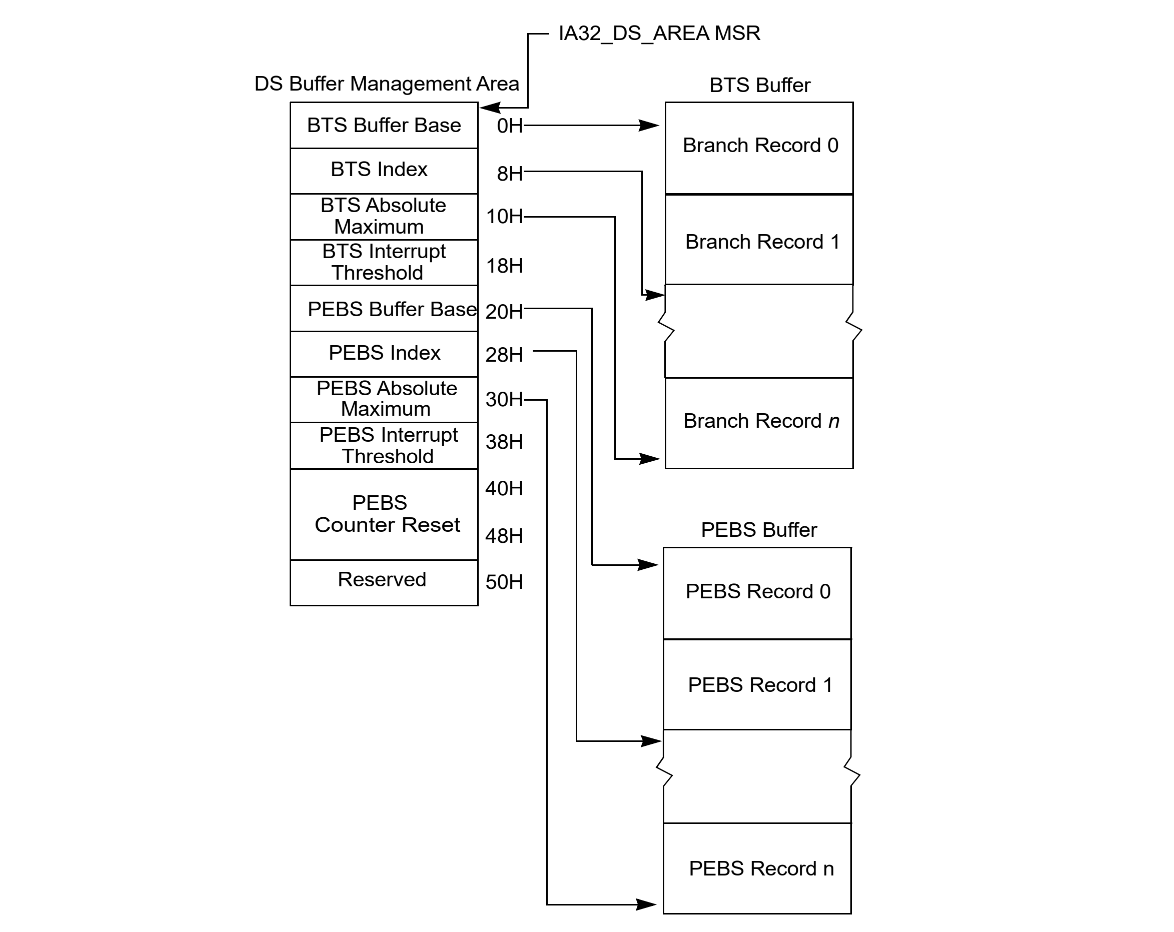


Figure 18-8. IA-32e Mode DS Save Area Example[1]

NOTES:

1. This example represents the format for a system that supports PEBS on only one counter.

The IA32_DS_AREA MSR holds the 64-bit linear address of the first byte of the DS buffer management area. The structure of a branch trace record is similar to that shown in Figure 18-6, but each field is 8 bytes in length. This makes each BTS record 24 bytes (see Figure 18-9). The structure of a PEBS record is similar to that shown in Figure 18-7, but each field is 8 bytes in length and architectural states include register R8 through R15. This makes the size of a PEBS record in 64-bit mode 144 bytes (see Figure 18-10).

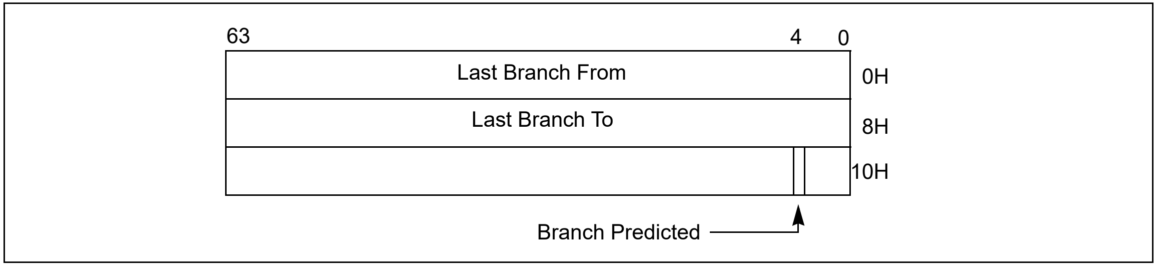


Figure 18-9. 64-bit Branch Trace Record Format

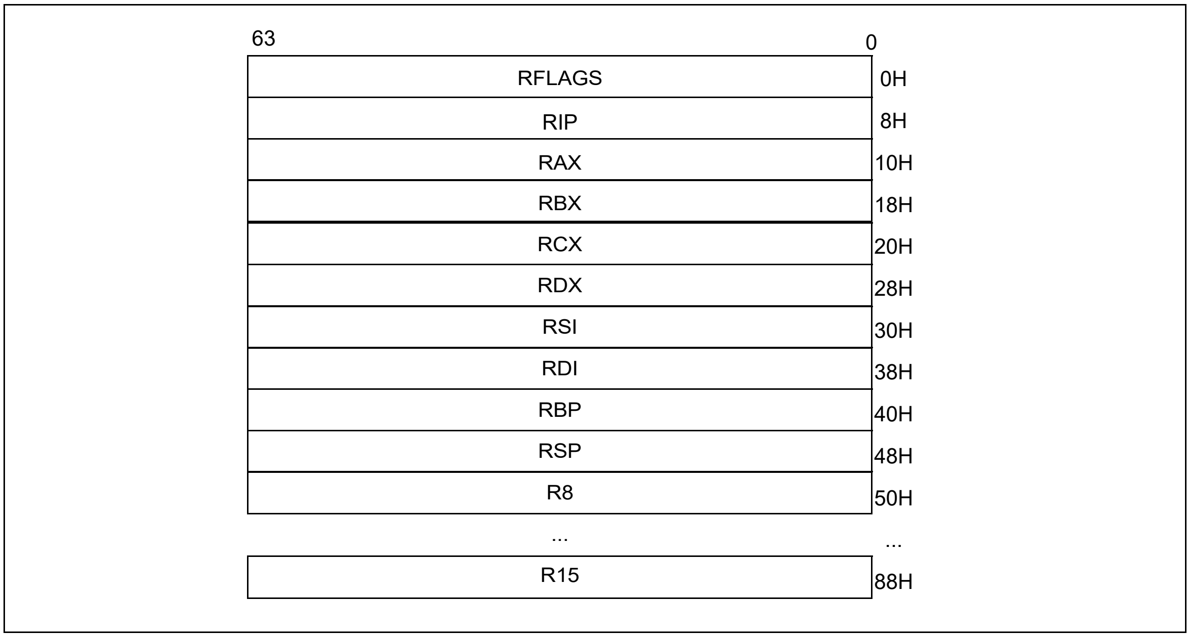


Figure 18-10. 64-bit PEBS Record Format

Fields in the buffer management area of a DS save area are described in Section 18.4.9.

The format of a branch trace record and a PEBS record are the same as the 64-bit record formats shown in Figures 18-9 and Figures 18-10, with the exception that the branch predicted bit is not supported by Intel Core microarchitecture or Intel Atom microarchitecture. The 64-bit record formats for BTS and PEBS apply to DS save area for all operating modes.

The procedures used to program IA32_DEBUGCTL MSR to set up a BTS buffer or a CPL-qualified BTS are described in Section 18.4.9.3 and Section 18.4.9.4.

Required elements for writing a DS interrupt service routine are largely the same on processors that support using DS Save area for BTS or PEBS records. However, on processors based on Intel NetBurst® microarchitecture, re-enabling counting requires writing to CCCRs. But a DS interrupt service routine on processors supporting architectural performance monitoring should:

- Re-enable the enable bits in IA32_PERF_GLOBAL_CTRL MSR if it is servicing an overflow PMI due to PEBS.
- Clear overflow indications by writing to IA32_PERF_GLOBAL_OVF_CTRL when a counting configuration is changed. This includes bit 62 (ClrOvfBuffer) and the overflow indication of counters used in either PEBS or general-purpose counting (specifically: bits 0 or 1; see Figures 20-3).

18.4.9.2 Setting Up the DS Save Area

To save branch records with the BTS buffer, the DS save area must first be set up in memory as described in the following procedure (See Section 20.6.2.4.1, “Setting up the PEBS Buffer,” for instructions for setting up a PEBS buffer, respectively, in the DS save area):

1. Create the DS buffer management information area in memory (see Section 18.4.9, “BTS and DS Save Area,” and Section 18.4.9.1, “64 Bit Format of the DS Save Area”). Also see the additional notes in this section.
2. Write the base linear address of the DS buffer management area into the IA32_DS_AREA MSR.
3. Set up the performance counter entry in the xAPIC LVT for fixed delivery and edge sensitive. See Section 11.5.1, “Local Vector Table.”
4. Establish an interrupt handler in the IDT for the vector associated with the performance counter entry in the xAPIC LVT.

5. Write an interrupt service routine to handle the interrupt. See Section 18.4.9.5, “Writing the DS Interrupt Service Routine.”

The following restrictions should be applied to the DS save area.

- The recording of branch records in the BTS buffer (or PEBS records in the PEBS buffer) may not operate properly if accesses to the linear addresses in any of the three DS save area sections cause page faults, VM exits, or the setting of accessed or dirty flags in the paging structures (ordinary or EPT). For that reason, system software should establish paging structures (both ordinary and EPT) to prevent such occurrences. Implications of this may be that an operating system should allocate this memory from a non-paged pool and that system software cannot do “lazy” page-table entry propagation for these pages. Some newer processor generations support “lazy” EPT page-table entry propagation for PEBS; see Section 20.3.9.1 and Section 20.9.5 for more information. A virtual-machine monitor may choose to allow use of PEBS by guest software only if EPT maps all guest-physical memory as present and read/write.
- The DS save area can be larger than a page, but the pages must be mapped to contiguous linear addresses. The buffer may share a page, so it need not be aligned on a 4-KByte boundary. For performance reasons, the base of the buffer must be aligned on a doubleword boundary and should be aligned on a cache line boundary.
- It is recommended that the buffer size for the BTS buffer and the PEBS buffer be an integer multiple of the corresponding record sizes.
- The precise event records buffer should be large enough to hold the number of precise event records that can occur while waiting for the interrupt to be serviced.
- The DS save area should be in kernel space. It must not be on the same page as code, to avoid triggering self-modifying code actions.
- There are no memory type restrictions on the buffers, although it is recommended that the buffers be designated as WB memory type for performance considerations.
- Either the system must be prevented from entering A20M mode while DS save area is active, or bit 20 of all addresses within buffer bounds must be 0.
- Pages that contain buffers must be mapped to the same physical addresses for all processes, such that any change to control register CR3 will not change the DS addresses.
- The DS save area is expected to used only on systems with an enabled APIC. The LVT Performance Counter entry in the APCI must be initialized to use an interrupt gate instead of the trap gate.

18.4.9.3 Setting Up the BTS Buffer

Three flags in the MSR_DEBUGCTLA MSR (see Table 18-5), IA32_DEBUGCTL (see Figure 18-3), or MSR_DEBUGCTLB (see Figure 18-16) control the generation of branch records and storing of them in the BTS buffer; these are TR, BTS, and BTINT. The TR flag enables the generation of BTMs. The BTS flag determines whether the BTMs are sent out on the system bus (clear) or stored in the BTS buffer (set). BTMs cannot be simultaneously sent to the system bus and logged in the BTS buffer. The BTINT flag enables the generation of an interrupt when the BTS buffer is full. When this flag is clear, the BTS buffer is a circular buffer.

Table 18-5. IA32_DEBUGCTL Flag Encodings

TR	BTS	BTINT	Description
0	X	X	Branch trace messages (BTMs) off
1	0	X	Generate BTMs
1	1	0	Store BTMs in the BTS buffer, used here as a circular buffer
1	1	1	Store BTMs in the BTS buffer, and generate an interrupt when the buffer is nearly full

The following procedure describes how to set up a DS Save area to collect branch records in the BTS buffer:

1. Place values in the BTS buffer base, BTS index, BTS absolute maximum, and BTS interrupt threshold fields of the DS buffer management area to set up the BTS buffer in memory.
2. Set the TR and BTS flags in the IA32_DEBUGCTL for Intel Core Solo and Intel Core Duo processors or later processors (or MSR_DEBUGCTLA MSR for processors based on Intel NetBurst Microarchitecture; or MSR_DEBUGCTLB for Pentium M processors).

3. Clear the BTINT flag in the corresponding IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR; or MSR_DEBUGCTLB) if a circular BTS buffer is desired.

NOTES

If the buffer size is set to less than the minimum allowable value (i.e., BTS absolute maximum < 1 + size of BTS record), the results of BTS is undefined.

In order to prevent generating an interrupt, when working with circular BTS buffer, SW need to set BTS interrupt threshold to a value greater than BTS absolute maximum (fields of the DS buffer management area). It's not enough to clear the BTINT flag itself only.

18.4.9.4 Setting Up CPL-Qualified BTS

If the processor supports CPL-qualified last branch recording mechanism, the generation of branch records and storing of them in the BTS buffer are determined by: TR, BTS, BTS_OFF_OS, BTS_OFF_USR, and BTINT. The encoding of these five bits are shown in Table 18-6.

Table 18-6. CPL-Qualified Branch Trace Store Encodings

TR	BTS	BTS_OFF_OS	BTS_OFF_USR	BTINT	Description
0	X	X	X	X	Branch trace messages (BTMs) off
1	0	X	X	X	Generates BTMs but do not store BTMs
1	1	0	0	0	Store all BTMs in the BTS buffer, used here as a circular buffer
1	1	1	0	0	Store BTMs with CPL > 0 in the BTS buffer
1	1	0	1	0	Store BTMs with CPL = 0 in the BTS buffer
1	1	1	1	X	Generate BTMs but do not store BTMs
1	1	0	0	1	Store all BTMs in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	1	0	1	Store BTMs with CPL > 0 in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	0	1	1	Store BTMs with CPL = 0 in the BTS buffer; generate an interrupt when the buffer is nearly full

18.4.9.5 Writing the DS Interrupt Service Routine

The BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector and interrupt service routine (called the debug store interrupt service routine or DS ISR). To handle BTS, non-precise event-based sampling, and PEBS interrupts: separate handler routines must be included in the DS ISR. Use the following guidelines when writing a DS ISR to handle BTS, non-precise event-based sampling, and/or PEBS interrupts.

- The DS interrupt service routine (ISR) must be part of a kernel driver and operate at a current privilege level of 0 to secure the buffer storage area.
- Because the BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector, the DS ISR must check for all the possible causes of interrupts from these facilities and pass control on to the appropriate handler.

  BTS and PEBS buffer overflow would be the sources of the interrupt if the buffer index matches/exceeds the interrupt threshold specified. Detection of non-precise event-based sampling as the source of the interrupt is accomplished by checking for counter overflow.
- There must be separate save areas, buffers, and state for each processor in an MP system.
- Upon entering the ISR, branch trace messages and PEBS should be disabled to prevent race conditions during access to the DS save area. This is done by clearing TR flag in the IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR) and by clearing the precise event enable flag in the MSR_PEBS_ENABLE MSR. These settings should be restored to their original values when exiting the ISR.

- The processor will not disable the DS save area when the buffer is full and the circular mode has not been selected. The current DS setting must be retained and restored by the ISR on exit.
- After reading the data in the appropriate buffer, up to but not including the current index into the buffer, the ISR must reset the buffer index to the beginning of the buffer. Otherwise, everything up to the index will look like new entries upon the next invocation of the ISR.
- The ISR must clear the mask bit in the performance counter LVT entry.
- The ISR must re-enable the counters to count via IA32_PERF_GLOBAL_CTRL/IA32_PERF_GLOBAL_OVF_CTRL if it is servicing an overflow PMI due to PEBS (or via CCCR's ENABLE bit on processor based on Intel NetBurst microarchitecture).
- The Pentium 4 Processor and Intel Xeon Processor mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

18.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL ATOM® PROCESSORS)

The Intel Core 2 Duo processor family and Intel Xeon processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture provide last branch interrupt and exception recording. The facilities described in this section also apply to 45 nm and 32 nm Intel Atom processors. These capabilities are similar to those found in Pentium 4 processors, including support for the following facilities:

- **Debug Trace and Branch Recording Control —** The IA32_DEBUGCTL MSR provide bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 18.4.1 for a description of the flags. See Figure 18-3 for the MSR layout.
- **Last branch record (LBR) stack —** There are a collection of MSR pairs that store the source and destination addresses related to recently executed branches. See Section 18.5.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts**
  - See Section 18.4.2 and Section 18.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
  - 45 nm and 32 nm Intel Atom processors clear the TR flag when the FREEZE_LBRS_ON_PMI flag is set.
- **Branch trace messages —** See Section 18.4.4.
- **Last exception records —** See Section 18.13.3.
- **Branch trace store and CPL-qualified BTS —** See Section 18.4.5.
- **FREEZE_LBRS_ON_PMI flag (bit 11) —** see Section 18.4.7 for legacy Freeze_LBRs_On_PMI operation.
- **FREEZE_PERFMON_ON_PMI flag (bit 12) —** see Section 18.4.7 for legacy Freeze_Perfmon_On_PMI operation.
- **FREEZE_WHILE_SMM (bit 14) —** FREEZE_WHILE_SMM is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 18.4.1.

18.5.1 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel Core 2, Intel Atom processor families, and Intel processors based on Intel NetBurst microarchitecture.

Four pairs of MSRs are supported in the LBR stack for Intel Core 2 processors families and Intel processors based on Intel NetBurst microarchitecture:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_3_FROM_IP (address 43H) store source addresses
  - MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_3_TO_IP (address 63H) store destination addresses

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 2 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

Eight pairs of MSRs are supported in the LBR stack for 45 nm and 32 nm Intel Atom processors:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_7_FROM_IP (address 47H) store source addresses
  - MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_7_TO_IP (address 67H) store destination addresses
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 18.4.8.1. The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

18.5.2 LBR Stack in Intel Atom® Processors based on the Silvermont Microarchitecture

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in Intel Atom processors based on the Silvermont and Airmont microarchitectures. Eight pairs of MSRs are supported in the LBR stack.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 18-11.

18.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Processors based on the Goldmont microarchitecture extend the capabilities described in Section 18.5.2 with the following enhancements:

- Supports new LBR format encoding 00110b in IA32_PERF_CAPABILITIES[5:0].
- Size of LBR stack increased to 32. Each entry includes MSR_LASTBRANCH_x_FROM_IP (address 0x680..0x69f) and MSR_LASTBRANCH_x_TO_IP (address 0x6c0..0x6df).
- LBR call stack filtering supported. The layout of MSR_LBR_SELECT is described in Table 18-13.
- Elapsed cycle information is added to MSR_LASTBRANCH_x_TO_IP. Format is shown in Table 18-7.
- Misprediction info is reported in the upper bits of MSR_LASTBRANCH_x_FROM_IP. MISPRED bit format is shown in Table 18-8.
- Streamlined Freeze_LBRs_On_PMI operation; see Section 18.12.2.
- LBR MSRs may be cleared when MWAIT is used to request a C-state that is numerically higher than C1; see Section 18.12.3.

Table 18-7. MSR_LASTBRANCH_x_TO_IP for the Goldmont Microarchitecture

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch to“ address. See Section 18.4.8.1 for address format.
Cycle Count (Saturating)	63:48	R/W	Elapsed core clocks since last update to the LBR stack.

18.7 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont Plus microarchitecture. Processors based on the Goldmont Plus microarchitecture extend the capabilities described in Section 18.6 with the following changes:

- Enumeration of new LBR format: encoding 00111b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 18.4.8.1.
- Each LBR stack entry consists of three MSRs:
  - MSR_LASTBRANCH_x_FROM_IP, the layout is simplified, see Table 18-9.
  - MSR_LASTBRANCH_x_TO_IP, the layout is the same as Table 18-9.
  - MSR_LBR_INFO_x, stores branch prediction flag, TSX info, and elapsed cycle data. Layout is the same as Table 18-16.

18.8 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR INTEL® XEON PHI™ PROCESSOR 7200/5200/3200

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in the Intel® Xeon Phi™ processor 7200/5200/3200 series based on the Knights Landing microarchitecture. Eight pairs of MSRs are supported in the LBR stack, per thread:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 680H) through MSR_LASTBRANCH_7_FROM_IP (address 687H) store source addresses.
  - MSR_LASTBRANCH_0_TO_IP (address 6C0H) through MSR_LASTBRANCH_7_TO_IP (address 6C7H) store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer —** The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 18-11.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 18.4.8.1.The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.

18.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON NEHALEM MICROARCHITECTURE

The processors based on Nehalem microarchitecture and Westmere microarchitecture support last branch interrupt and exception recording. These capabilities are similar to those found in Intel Core 2 processors and add additional capabilities:

- **Debug Trace and Branch Recording Control —** The IA32_DEBUGCTL MSR provides bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 18.4.1 for a description of the flags. See Figure 18-11 for the MSR layout.
- **Last branch record (LBR) stack —** There are 16 MSR pairs that store the source and destination addresses related to recently executed branches. See Section 18.9.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts —** See Section 18.4.2 and Section 18.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.

- **Branch trace messages —** The IA32_DEBUGCTL MSR provides bit fields for software to enable each logical processor to generate branch trace messages. See Section 18.4.4. However, not all BTM messages are observable using the Intel® QPI link.
- **Last exception records —** See Section 18.13.3.
- **Branch trace store and CPL-qualified BTS —** See Section 18.4.6 and Section 18.4.5.
- **FREEZE_LBRS_ON_PMI flag (bit 11) —** see Section 18.4.7 for legacy Freeze_LBRs_On_PMI operation.
- **FREEZE_PERFMON_ON_PMI flag (bit 12) —** see Section 18.4.7 for legacy Freeze_Perfmon_On_PMI operation.
- **UNCORE_PMI_EN (bit 13) —** When set. this logical processor is enabled to receive an counter overflow interrupt form the uncore.
- **FREEZE_WHILE_SMM (bit 14) —** FREEZE_WHILE_SMM is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 18.4.1.

Processors based on Nehalem microarchitecture provide additional capabilities:

- **Independent control of uncore PMI —** The IA32_DEBUGCTL MSR provides a bit field (see Figure 18-11) for software to enable each logical processor to receive an uncore counter overflow interrupt.
- **LBR filtering —** Processors based on Nehalem microarchitecture support filtering of LBR based on combination of CPL and branch type conditions. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT.

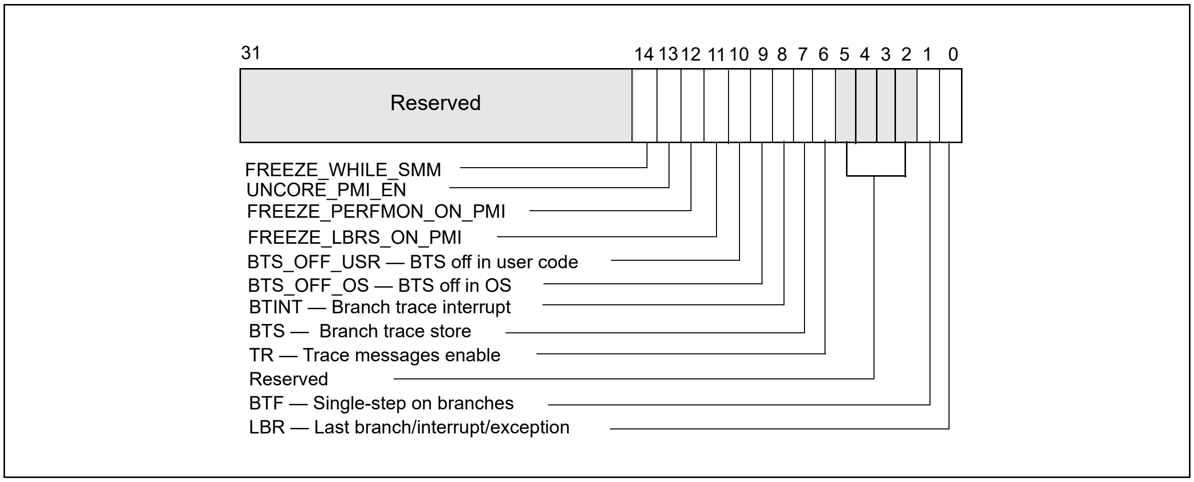

31
14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Reserved
FREEZE_WHILE_SMM
UNCORE_PMI_EN
FREEZE_PERFMON_ON_PMI
FREEZE_LBRS_ON_PMI
BTS_OFF_USR — BTS off in user code
BTS_OFF_OS — BTS off in OS
BTINT — Branch trace interrupt
BTS — Branch trace store
TR — Trace messages enable
Reserved
BTF — Single-step on branches
LBR — Last branch/interrupt/exception

Figure 18-11. IA32_DEBUGCTL MSR for Processors Based on Nehalem Microarchitecture

18.9.1 LBR Stack

Processors based on Nehalem microarchitecture provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is shown in Table 18-8 and Table 18-9.

Table 18-8. MSR_LASTBRANCH_x_FROM_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch from” address. See Section 18.4.8.1 for address format.
SIGN_EXt	62:48	R/W	Signed extension of bit 47 of this register.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

Table 18-9. MSR_LASTBRANCH_x_TO_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch to” address. See Section 18.4.8.1 for address format
SIGN_EXt	63:48	R/W	Signed extension of bit 47 of this register.

Processors based on Nehalem microarchitecture have an LBR MSR Stack as shown in Table 18-10.

Table 18-10. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_1AH	16	0 to 15

18.9.2 Filtering of Last Branch Records

MSR_LBR_SELECT is cleared to zero at RESET, and LBR filtering is disabled, i.e., all branches will be captured. MSR_LBR_SELECT provides bit fields to specify the conditions of subsets of branches that will not be captured in the LBR. The layout of MSR_LBR_SELECT is shown in Table 18-11.

Table 18-11. MSR_LBR_SELECT for Nehalem Microarchitecture

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

18.10 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SANDY BRIDGE MICROARCHITECTURE

Generally, all of the last branch record, interrupt, and exception recording facility described in Section 18.9, “Last Branch, Interrupt, and Exception Recording for Processors based on Nehalem Microarchitecture,” apply to processors based on Sandy Bridge microarchitecture. For processors based on Ivy Bridge microarchitecture, the same holds true.

One difference of note is that MSR_LBR_SELECT is shared between two logical processors in the same core. In Sandy Bridge microarchitecture, each logical processor has its own MSR_LBR_SELECT. The filtering semantics for “Near_ind_jmp” and “Near_rel_jmp” has been enhanced, see Table 18-12.

Table 18-12. MSR_LBR_SELECT for Sandy Bridge Microarchitecture

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

18.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON HASWELL MICROARCHITECTURE

Generally, all of the last branch record, interrupt, and exception recording facility described in Section 18.10, "Last Branch, Interrupt, and Exception Recording for Processors based on Sandy Bridge Microarchitecture," apply to next generation processors based on Haswell microarchitecture.

The LBR facility also supports an alternate capability to profile call stack profiles. Configuring the LBR facility to conduct call stack profiling is by writing 1 to the MSR_LBR_SELECT.EN_CALLSTACK[bit 9]; see Table 18-13. If MSR_LBR_SELECT.EN_CALLSTACK is clear, the LBR facility will capture branches normally as described in Section 18.10.

Table 18-13. MSR_LBR_SELECT for Haswell Microarchitecture

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches ending in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches ending in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
EN_CALLSTACK[1]	9		Enable LBR stack to use LIFO filtering to capture Call stack profile
Reserved	63:10		Must be zero

NOTES:

[1] Must set valid combination of bits 0-8 in conjunction with bit 9 (as described below), otherwise the contents of the LBR MSRs are undefined.

The call stack profiling capability is an enhancement of the LBR facility. The LBR stack is a ring buffer typically used to profile control flow transitions resulting from branches. However, the finite depth of the LBR stack often become less effective when profiling certain high-level languages (e.g., C++), where a transition of the execution flow is accompanied by a large number of leaf function calls, each of which returns an individual parameter to form the list of parameters for the main execution function call. A long list of such parameters returned by the leaf functions would serve to flush the data captured in the LBR stack, often losing the main execution context.

When the call stack feature is enabled, the LBR stack will capture unfiltered call data normally, but as return instructions are executed the last captured branch record is flushed from the on-chip registers in a last-in first-out (LIFO) manner. Thus, branch information relative to leaf functions will not be captured, while preserving the call stack information of the main line execution path.

The configuration of the call stack facility is summarized below:

- Set IA32_DEBUGCTL.LBR (bit 0) to enable the LBR stack to capture branch records. The source and target addresses of the call branches will be captured in the 16 pairs of From/To LBR MSRs that form the LBR stack.
- Program the Top of Stack (TOS) MSR that points to the last valid from/to pair. This register is incremented by 1, modulo 16, before recording the next pair of addresses.
- Program the branch filtering bits of MSR_LBR_SELECT (bits 0:8) as desired.
- Program the MSR_LBR_SELECT to enable LIFO filtering of return instructions with:
  - — The following bits in MSR_LBR_SELECT must be set to ‘1’: JCC, NEAR_IND_JMP, NEAR_REL_JMP, FAR_BRANCH, EN_CALLSTACK;
  - — The following bits in MSR_LBR_SELECT must be cleared: NEAR_REL_CALL, NEAR-IND_CALL, NEAR_RET;
  - — At most one of CPL_EQ_0, CPL_NEQ_0 is set.

Note that when call stack profiling is enabled, “zero length calls” are excluded from writing into the LBRs. (A “zero length call” uses the attribute of the call instruction to push the immediate instruction pointer on to the stack and then pops off that address into a register. This is accomplished without any matching return on the call.)

18.11.1 LBR Stack Enhancement

Processors based on Haswell microarchitecture provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is enumerated by IA32_PERF_CAPABILITIES[5:0] = 04H, and is shown in Table 18-14 and Table 18-9.

Table 18-14. MSR_LASTBRANCH_x_FROM_IP with TSX Information

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch from” address. See Section 18.4.8.1 for address format.
SIGN_EXT	60:48	R/W	Signed extension of bit 47 of this register.
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region, or EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

18.12 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SKYLAKE MICROARCHITECTURE

Processors based on the Skylake microarchitecture provide a number of enhancement with storing last branch records:

- enumeration of new LBR format: encoding 00101b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 18.4.8.1.
- Each LBR stack entry consists of a triplets of MSRs:

- MSR_LASTBRANCH_x_FROM_IP, the layout is simplified, see Table 18-9.
- MSR_LASTBRANCH_x_TO_IP, the layout is the same as Table 18-9.
- MSR_LBR_INFO_x, stores branch prediction flag, TSX info, and elapsed cycle data.

- Size of LBR stack increased to 32.

Processors based on the Skylake microarchitecture support the same LBR filtering capabilities as described in Table 18-13.

Table 18-15. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_4EH, 06_5EH	32	0 to 31

18.12.1 MSR_LBR_INFO_x MSR

The layout of each MSR_LBR_INFO_x MSR is shown in Table 18-16.

Table 18-16. MSR_LBR_INFO_x

Bit Field	Bit Offset	Access	Description
Cycle Count (saturating)	15:0	R/W	Elapsed core clocks since last update to the LBR stack.
Reserved	60:16	R/W	Reserved
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region OR EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

18.12.2 Streamlined Freeze_LBRs_On_PMI Operation

The FREEZE_LBRS_ON_PMI feature causes the LBRs to be frozen on a hardware request for a PMI. This prevents the LBRs from being overwritten by new branches, allowing the PMI handler to examine the control flow that preceded the PMI generation. Architectural performance monitoring version 4 and above supports a streamlined FREEZE_LBRs_ON_PMI operation for PMI service routine that replaces the legacy FREEZE_LBRs_ON_PMI operation (see Section 18.4.7).

While the legacy FREEZE_LBRS_ON_PMI clear the LBR bit in the IA32_DEBUGCTL MSR on a PMI request, the streamlined FREEZE_LBRS_ON_PMI will set the LBR_FRZ bit in IA32_PERF_GLOBAL_STATUS. Branches will not cause the LBRs to be updated when LBR_FRZ is set. Software can clear LBR_FRZ at the same time as it clears overflow bits by setting the LBR_FRZ bit as well as the needed overflow bit when writing to IA32_PERF_GLOBAL_STATUS_RESET MSR.

This streamlined behavior avoids race conditions between software and processor writes to IA32_DEBUGCTL that are possible with FREEZE_LBRS_ON_PMI clearing of the LBR enable.

18.12.3 LBR Behavior and Deep C-State

When MWAIT is used to request a C-state that is numerically higher than C1, then LBR state may be initialized to zero depending on optimized "waiting" state that is selected by the processor The affected LBR states include the FROM, TO, INFO, LAST_BRANCH, LER, and LBR_TOS registers. The LBR enable bit and LBR_FROZEN bit are not affected. The LBR-time of the first LBR record inserted after an exit from such a C-state request will be zero.

18.13 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE)

Pentium 4 and Intel Xeon processors based on Intel NetBurst microarchitecture provide the following methods for recording taken branches, interrupts, and exceptions:

- Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address.
- Send the branch records out on the system bus as branch trace messages (BTMs).
- Log BTMs in a memory-resident branch trace store (BTS) buffer.

To support these functions, the processor provides the following MSRs and related facilities:

- **MSR_DEBUGCTLA MSR** — Enables last branch, interrupt, and exception recording; single-stepping on taken branches; branch trace messages (BTMs); and branch trace store (BTS). This register is named DebugCtlMSR in the P6 family processors.
- **Debug store (DS) feature flag (CPUID.1:EDX.DS[bit 21])** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer.
- **CPL-qualified debug store (DS) feature flag (CPUID.1:ECX.DS-CPL[bit 4])** — Indicates that the processor provides a CPL-qualified debug store (DS) mechanism, which allows software to selectively skip sending and storing BTMs, according to specified current privilege level settings, into a memory-resident BTS buffer.
- **IA32_MISC_ENABLE MSR** — Indicates that the processor provides the BTS facilities.
- **Last branch record (LBR) stack** — The LBR stack is a circular stack that consists of four MSRs (MSR_LAST-BRANCH_0 through MSR_LASTBRANCH_3) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. The LBR stack consists of 16 MSR pairs (MSR_LASTBRANCH_0_FROM_IP through MSR_LASTBRANCH_15_FROM_IP and MSR_LASTBRANCH_0_TO_IP through MSR_LASTBRANCH_15_TO_IP) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].
- **Last branch record top-of-stack (TOS) pointer** — The TOS Pointer MSR contains a 2-bit pointer (0-3) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H]. See also: Table 18-17, Figure 18-12, and Section 18.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture."
- **Last exception record** — See Section 18.13.3, "Last Exception Records."

18.13.1 MSR_DEBUGCTLA MSR

The MSR_DEBUGCTLA MSR enables and disables the various last branch recording mechanisms described in the previous section. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 18-12 shows the flags in the MSR_DEBUGCTLA MSR. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. Each branch, interrupt, or exception is recorded as a 64-bit branch record. The processor clears this flag whenever a debug exception is generated (for example, when an instruction or data breakpoint or a single-step trap occurs). See Section 18.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

- **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, “Single-Stepping on Branches.”
- **TR (trace message enable) flag (bit 2) —** When set, branch trace messages are enabled. Thereafter, when the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, “Branch Trace Messages.”

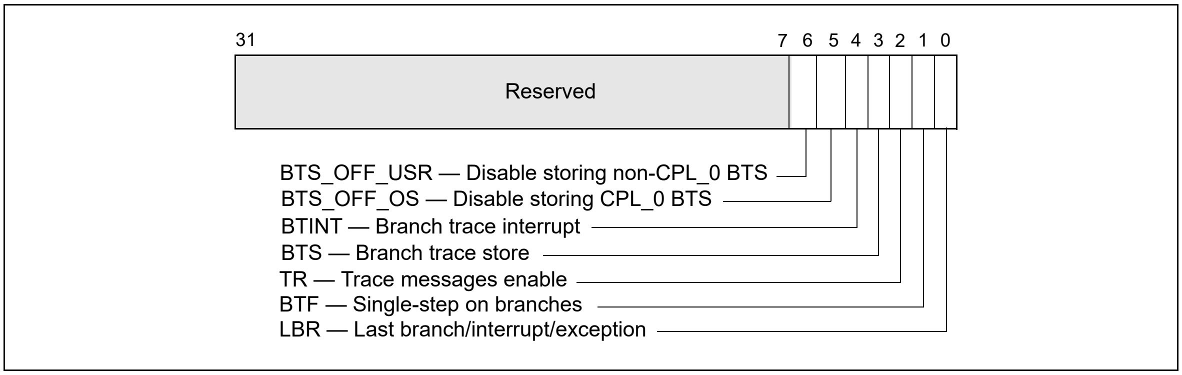

31
7 6 5 4 3 2 1 0
Reserved
BTS_OFF_USR — Disable storing non-CPL_0 BTS
BTS_OFF_OS — Disable storing CPL_0 BTS
BTINT — Branch trace interrupt
BTS — Branch trace store
TR — Trace messages enable
BTF — Single-step on branches
LBR — Last branch/interrupt/exception

Figure 18-12. MSR_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors

- **BTS (branch trace store) flag (bit 3) —** When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bits 4) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, “Branch Trace Store (BTS).”
- **BTS_OFF_OS (disable ring 0 branch trace store) flag (bit 5) —** When set, enables the BTS facilities to skip sending/logging CPL_0 BTMs to the memory-resident BTS buffer. See Section 18.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”
- **BTS_OFF_USR (disable ring 0 branch trace store) flag (bit 6) —** When set, enables the BTS facilities to skip sending/logging non-CPL_0 BTMs to the memory-resident BTS buffer. See Section 18.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

NOTE

The initial implementation of BTS_OFF_USR and BTS_OFF_OS in MSR_DEBUGCTLA is shown in Figure 18-12. The BTS_OFF_USR and BTS_OFF_OS fields may be implemented on other model-specific debug control register at different locations.

See Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for a detailed description of each of the last branch recording MSRs.

18.13.2 LBR Stack for Processors Based on Intel NetBurst® Microarchitecture

The LBR stack is made up of LBR MSRs that are treated by the processor as a circular stack. The TOS pointer (MSR_LASTBRANCH_TOS MSR) points to the LBR MSR (or LBR MSR pair) that contains the most recent (last) branch record placed on the stack. Prior to placing a new branch record on the stack, the TOS is incremented by 1. When the TOS pointer reaches it maximum value, it wraps around to 0. See Table 18-17 and Figure 18-12.

Table 18-17. LBR MSR Stack Size and TOS Pointer Range for the Pentium® 4 and the Intel® Xeon® Processor Family

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	4	0 to 3
Family 0FH, Models; MSRs at locations 680H-68FH.	16	0 to 15
Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	16	0 to 15

The registers in the LBR MSR stack and the MSR_LASTBRANCH_TOS MSR are read-only and can be read using the RDMSR instruction.

Figure 18-13 shows the layout of a branch record in an LBR MSR (or MSR pair). Each branch record consists of two linear addresses, which represent the "from" and "to" instruction pointers for a branch, interrupt, or exception. The contents of the from and to addresses differ, depending on the source of the branch:

- **Taken branch** — If the record is for a taken branch, the "from" address is the address of the branch instruction and the "to" address is the target instruction of the branch.
- **Interrupt** — If the record is for an interrupt, the "from" address the return instruction pointer (RIP) saved for the interrupt and the "to" address is the address of the first instruction in the interrupt handler routine. The RIP is the linear address of the next instruction to be executed upon returning from the interrupt handler.
- **Exception** — If the record is for an exception, the "from" address is the linear address of the instruction that caused the exception to be generated and the "to" address is the address of the first instruction in the exception handler routine.

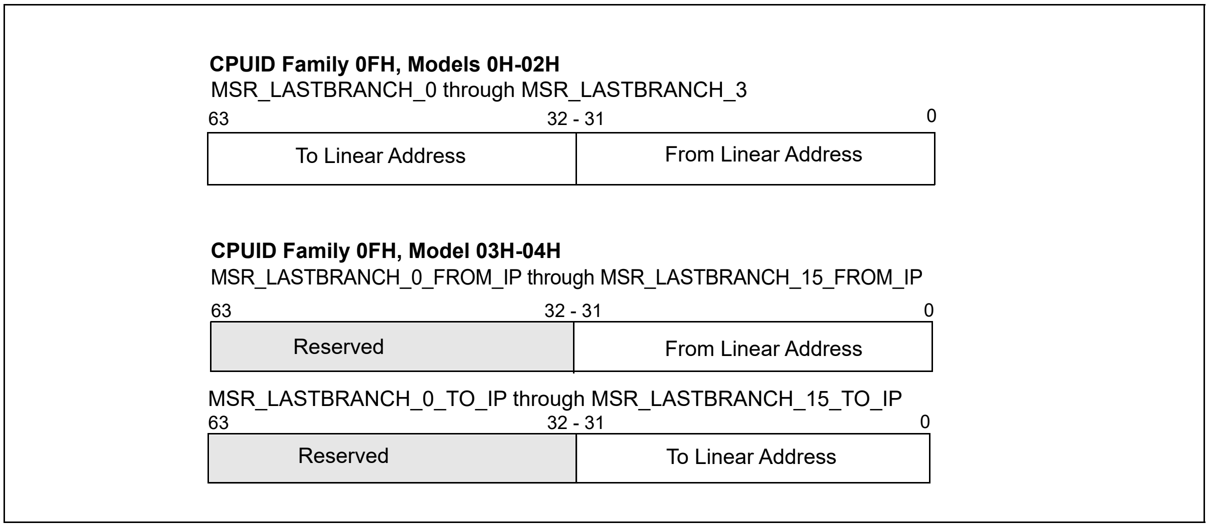

CPUID Family 0FH, Models 0H-02H
MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3
63
32 - 31
0
To Linear Address
From Linear Address
CPUID Family 0FH, Model 03H-04H
MSR_LASTBRANCH_0_FROM_IP through MSR_LASTBRANCH_15_FROM_IP
63
32 - 31
0
Reserved
From Linear Address
MSR_LASTBRANCH_0_TO_IP through MSR_LASTBRANCH_15_TO_IP
63
32 - 31
0
Reserved
To Linear Address

Figure 18-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel® Xeon® Processor Family

Additional information is saved if an exception or interrupt occurs in conjunction with a branch instruction. If a branch instruction generates a trap type exception, two branch records are stored in the LBR stack: a branch record for the branch instruction followed by a branch record for the exception.

If a branch instruction is immediately followed by an interrupt, a branch record is stored in the LBR stack for the branch instruction followed by a record for the interrupt.

18.13.3 Last Exception Records

The Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel Atom® processors provide two MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate the functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.

The MSR_LER_TO_LIP and MSR_LER_FROM_LIP MSRs contain a branch record for the last branch that the processor took prior to an exception or interrupt being generated.

18.14 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

Intel Core Solo and Intel Core Duo processors provide last branch interrupt and exception recording. This capability is almost identical to that found in Pentium 4 and Intel Xeon processors. There are differences in the stack and in some MSR names and locations.

Note the following:

- **IA32_DEBUGCTL MSR —** Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. IA32_DEBUGCTL MSR is located at register address 01D9H.

  See Figure 18-14 for the layout and the entries below for a description of the flags:

  — **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" below.

  — **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, "Single-Stepping on Branches," for more information about the BTF flag.

  — **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, "Branch Trace Messages," for more information about the TR flag.

  — **BTS (branch trace store) flag (bit 7) —** When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.9, "BTS and DS Save Area."

  — **BTINT (branch trace interrupt) flag (bits 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

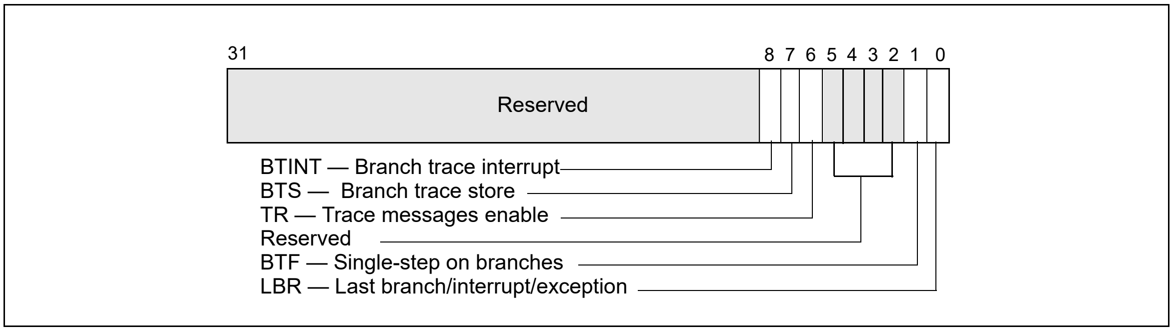

31
8 7 6 5 4 3 2 1 0
Reserved
BTINT — Branch trace interrupt
BTS — Branch trace store
TR — Trace messages enable
Reserved
BTF — Single-step on branches
LBR — Last branch/interrupt/exception

Figure 18-14. IA32_DEBUGCTL MSR for Intel® Core™ Solo and Intel® Core™ Duo Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 18.4.5, "Branch Trace Store (BTS)."
- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address (MSR addresses start at 40H). See Figure 18-15.

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Intel Core Solo and Intel Core Duo processors, this MSR is located at register address 01C9H.

For compatibility, the Intel Core Solo and Intel Core Duo processors provide two 32-bit MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

For details, see Section 18.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture," and Section 2.20, "MSRs In Intel® Core™ Solo and Intel® Core™ Duo Processors," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

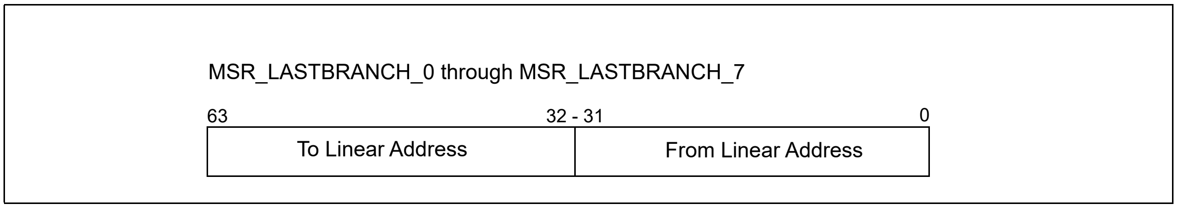


Figure 18-15. LBR Branch Record Layout for the Intel® Core™ Solo and Intel® Core™ Duo Processor

18.15 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PENTIUM M PROCESSORS)

Like the Pentium 4 and Intel Xeon processor family, Pentium M processors provide last branch interrupt and exception recording. The capability operates almost identically to that found in Pentium 4 and Intel Xeon processors. There are differences in the shape of the stack and in some MSR names and locations. Note the following:

- **MSR_DEBUGCTLB MSR —** Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. For Pentium M processors, this MSR is located at register address 01D9H. See Figure 18-16 and the entries below for a description of the flags.
  - **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" bullet below.
  - **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, "Single-Stepping on Branches," for more information about the BTF flag.
  - **PB*i* (performance monitoring/breakpoint pins) flags (bits 5-2) —** When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, "Branch Trace Messages," for more information about the TR flag.
  - **BTS (branch trace store) flag (bit 7) —** When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.9, "BTS and DS Save Area."
  - **BTINT (branch trace interrupt) flag (bits 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

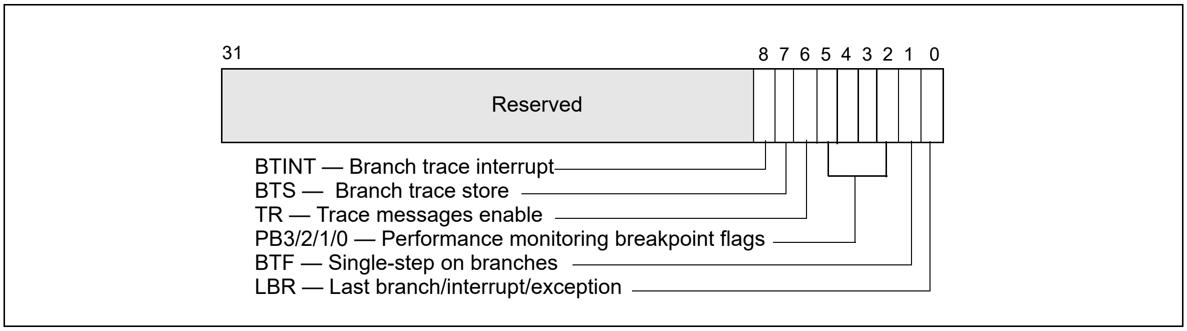

31
8 7 6 5 4 3 2 1 0
Reserved
BTINT — Branch trace interrupt
BTS — Branch trace store
TR — Trace messages enable
PB3/2/1/0 — Performance monitoring breakpoint flags
BTF — Single-step on branches
LBR — Last branch/interrupt/exception

Figure 18-16. MSR_DEBUGCTLB MSR for Pentium M Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 18.4.5, "Branch Trace Store (BTS)."
- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address. For Pentium M Processors, these pairs are located at register addresses 040H-047H. See Figure 18-17.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Pentium M Processors, this MSR is located at register address 01C9H.

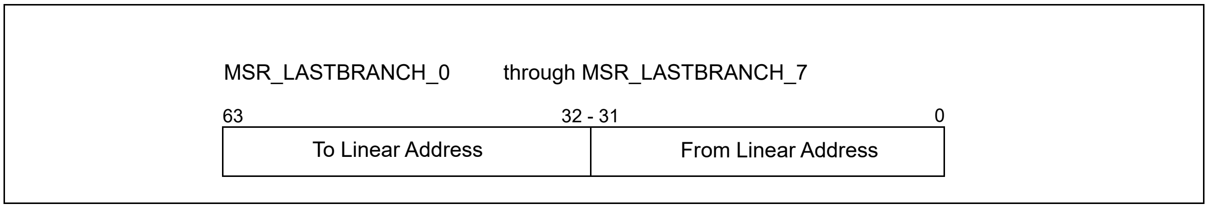


Figure 18-17. LBR Branch Record Layout for the Pentium M Processor

For more detail on these capabilities, see Section 18.13.3, "Last Exception Records," and Section 2.21, "MSRs In the Pentium M Processor," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

18.16 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (P6 FAMILY PROCESSORS)

The P6 family processors provide five MSRs for recording the last branch, interrupt, or exception taken by the processor: DEBUGCTLMSR, LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP. These registers can be used to collect last branch records, to set breakpoints on branches, interrupts, and exceptions, and to single-step from one branch to the next.

See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for a detailed description of each of the last branch recording MSRs.

18.16.1 DEBUGCTLMSR Register

The version of the DEBUGCTLMSR register found in the P6 family processors enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode.

A protected-mode operating system procedure is required to provide user access to this register. Figure 18-18 shows the flags in the DEBUGCTLMSR register for the P6 family processors. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records the source and target addresses (in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs) for the last branch and the last exception or interrupt taken by the processor prior to a debug exception being generated. The processor clears this flag whenever a debug exception, such as an instruction or data breakpoint or single-step trap occurs.

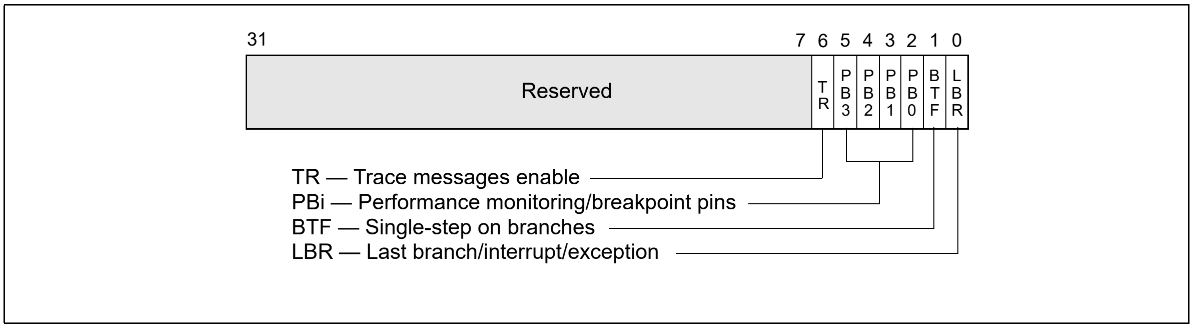

31
7
6
5
4
3
2
1
0
Reserved
TR
PB3
PB2
PB1
PB0
BTF
LBR
TR — Trace messages enable
PBi — Performance monitoring/breakpoint pins
BTF — Single-step on branches
LBR — Last branch/interrupt/exception

Figure 18-18. DEBUGCTLMSR Register (P6 Family Processors)

- **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag. See Section 18.4.3, “Single-Stepping on Branches.”
- **PB*i* (performance monitoring/breakpoint pins) flags (bits 2 through 5) —** When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
- **TR (trace message enable) flag (bit 6) —** When set, trace messages are enabled as described in Section 18.4.4, “Branch Trace Messages.” Setting this flag greatly reduces the performance of the processor. When trace messages are enabled, the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are undefined.

18.16.2 Last Branch and Last Exception MSRs

The LastBranchToIP and LastBranchFromIP MSRs are 32-bit registers for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated. When a branch occurs, the processor loads the address of the branch instruction into the LastBranchFromIP MSR and loads the target address for the branch into the LastBranchToIP MSR.

When an interrupt or exception occurs (other than a debug exception), the address of the instruction that was interrupted by the exception or interrupt is loaded into the LastBranchFromIP MSR and the address of the exception or interrupt handler that is called is loaded into the LastBranchToIP MSR.

The LastExceptionToIP and LastExceptionFromIP MSRs (also 32-bit registers) record the instruction pointers for the last branch that the processor took prior to an exception or interrupt being generated. When an exception or interrupt occurs, the contents of the LastBranchToIP and LastBranchFromIP MSRs are copied into these registers before the to and from addresses of the exception or interrupt are recorded in the LastBranchToIP and LastBranchFromIP MSRs.

These registers can be read using the RDMSR instruction.

Note that the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into the current code segment, as opposed to linear addresses, which are saved in last branch records for the Pentium 4 and Intel Xeon processors.

18.16.3 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag in the DEBUGCTLMSR register is set, the processor automatically begins recording branches that it takes, exceptions that are generated (except for debug exceptions), and interrupts that are serviced. Each time a branch, exception, or interrupt occurs, the processor records the to and from instruction pointers in the Last-BranchToIP and LastBranchFromIP MSRs. In addition, for interrupts and exceptions, the processor copies the contents of the LastBranchToIP and LastBranchFromIP MSRs into the LastExceptionToIP and LastExceptionFromIP MSRs prior to recording the to and from addresses of the interrupt or exception.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler, but does not touch the last branch and last exception MSRs. The addresses for the last branch, interrupt, or exception taken are thus retained in the LastBranchToIP and LastBranchFromIP MSRs and the addresses of the last branch prior to an interrupt or exception are retained in the LastExceptionToIP, and LastExceptionFromIP MSRs.

The debugger can use the last branch, interrupt, and/or exception addresses in combination with code-segment selectors retrieved from the stack to reset breakpoints in the breakpoint-address registers (DR0 through DR3), allowing a backward trace from the manifestation of a particular bug toward its source. Because the instruction pointers recorded in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into a code segment, software must determine the segment base address of the code segment associated with the control transfer to calculate the linear address to be placed in the breakpoint-address registers. The segment base address can be determined by reading the segment selector for the code segment from the stack and using it to locate the segment descriptor for the segment in the GDT or LDT. The segment base address can then be read from the segment descriptor.

Before resuming program execution from a debug-exception handler, the handler must set the LBR flag again to re-enable last branch and last exception/interrupt recording.

18.17 TIME-STAMP COUNTER

The Intel 64 and IA-32 architectures (beginning with the Pentium processor) define a time-stamp counter mechanism that can be used to monitor and identify the relative time occurrence of processor events. The counter's architecture includes the following components:

- **TSC flag —** A feature bit that indicates the availability of the time-stamp counter. The counter is available in an if the function CPUID.1:EDX.TSC[bit 4] = 1.
- **IA32_TIME_STAMP_COUNTER MSR** (called TSC MSR in P6 family and Pentium processors) **—** The MSR used as the counter.
- **RDTSC instruction —** An instruction used to read the time-stamp counter.
- **TSD flag —** A control register flag is used to enable or disable the time-stamp counter (enabled if CR4.TSD[bit 2] = 1).

The time-stamp counter (as implemented in the P6 family, Pentium, Pentium M, Pentium 4, Intel Xeon, Intel Core Solo and Intel Core Duo processors and later processors) is a 64-bit counter that is set to 0 following a RESET of the processor. Following a RESET, the counter increments even when the processor is halted by the HLT instruction or the external STPCLK# pin. Note that the assertion of the external DPSLP# pin may cause the time-stamp counter to stop.

Processor families increment the time-stamp counter differently:

- For Pentium M processors (family [06H], models [09H, 0DH]); for Pentium 4 processors, Intel Xeon processors (family [0FH], models [00H, 01H, or 02H]); and for P6 family processors: the time-stamp counter increments with every internal processor clock cycle.

  The internal processor clock cycle is determined by the current core-clock to bus-clock ratio. Intel® SpeedStep® technology transitions may also impact the processor clock.
- For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]); for Intel Core Solo and Intel Core Duo processors (family [06H], model [0EH]); for the Intel Xeon processor 5100 series and Intel Core 2 Duo processors (family [06H], model [0FH]); for Intel Core 2 and Intel Xeon processors (family [06H], DisplayModel [17H]); for Intel Atom processors (family [06H], DisplayModel [1CH]): the time-stamp counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the

processor or may be set by the maximum resolved frequency at which the processor is booted. The maximum resolved frequency may differ from the processor base frequency, see Section 20.7.2 for more detail. On certain processors, the TSC frequency may not be the same as the frequency in the brand string.

The specific processor configuration determines the behavior. Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency. This is the architectural behavior moving forward.

NOTE

To determine average processor clock frequency, Intel recommends the use of performance monitoring logic to count processor core clocks over the period of time for which the average is required. See Section 20.6.4.5, "Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture," and https://perfmon-events.intel.com/ for more information.

The RDTSC instruction reads the time-stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for a 64-bit counter wraparound. Intel guarantees that the time-stamp counter will not wraparound within 10 years after being reset. The period for counter wrap is longer for Pentium 4, Intel Xeon, P6 family, and Pentium processors.

Normally, the RDTSC instruction can be executed by programs and procedures running at any privilege level and in virtual-8086 mode. The TSD flag allows use of this instruction to be restricted to programs and procedures running at privilege level 0. A secure operating system would set the TSD flag during system initialization to disable user access to the time-stamp counter. An operating system that disables user access to the time-stamp counter should emulate the instruction through a user-accessible programming interface.

The RDTSC instruction is not serializing or ordered with other instructions. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed.

The RDMSR and WRMSR instructions read and write the time-stamp counter, treating the time-stamp counter as an ordinary MSR (address 10H). In the Pentium 4, Intel Xeon, and P6 family processors, all 64-bits of the time-stamp counter are read using RDMSR (just as with RDTSC). When WRMSR is used to write the time-stamp counter on processors before family [0FH], models [03H, 04H]: only the low-order 32-bits of the time-stamp counter can be written (the high-order 32 bits are cleared to 0). For family [0FH], models [03H, 04H, 06H]; for family [06H]], model [0EH, 0FH]; for family [06H]], DisplayModel [17H, 1AH, 1CH, 1DH]: all 64 bits are writable.

18.17.1 Invariant TSC

The time stamp counter in newer processors may support an enhancement, referred to as invariant TSC. Processor's support for invariant TSC is indicated by CPUID.80000007H:EDX[8].

The invariant TSC will run at a constant rate in all ACPI P-, C-. and T-states. This is the architectural behavior moving forward. On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers). TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource.

18.17.2 IA32_TSC_AUX Register and RDTSCP Support

Processors based on Nehalem microarchitecture provide an auxiliary TSC register, IA32_TSC_AUX that is designed to be used in conjunction with IA32_TSC. IA32_TSC_AUX provides a 32-bit field that is initialized by privileged software with a signature value (for example, a logical processor ID).

The primary usage of IA32_TSC_AUX in conjunction with IA32_TSC is to allow software to read the 64-bit time stamp in IA32_TSC and signature value in IA32_TSC_AUX with the instruction RDTSCP in an atomic operation. RDTSCP returns the 64-bit time stamp in EDX:EAX and the 32-bit TSC_AUX signature value in ECX. The atomicity of RDTSCP ensures that no context switch can occur between the reads of the TSC and TSC_AUX values.

Support for RDTSCP is indicated by CPUID.80000001H:EDX[27]. As with RDTSC instruction, non-ring 0 access is controlled by CR4.TSD (Time Stamp Disable flag).

User mode software can use RDTSCP to detect if CPU migration has occurred between successive reads of the TSC. It can also be used to adjust for per-CPU differences in TSC values in a NUMA system.

18.17.3 Time-Stamp Counter Adjustment

Software can modify the value of the time-stamp counter (TSC) of a logical processor by using the WRMSR instruction to write to the IA32_TIME_STAMP_COUNTER MSR (address 10H). Because such a write applies only to that logical processor, software seeking to synchronize the TSC values of multiple logical processors must perform these writes on each logical processor. It may be difficult for software to do this in a way that ensures that all logical processors will have the same value for the TSC at a given point in time.

The synchronization of TSC adjustment can be simplified by using the 64-bit IA32_TSC_ADJUST MSR (address 3BH). Like the IA32_TIME_STAMP_COUNTER MSR, the IA32_TSC_ADJUST MSR is maintained separately for each logical processor. A logical processor maintains and uses the IA32_TSC_ADJUST MSR as follows:

- On RESET, the value of the IA32_TSC_ADJUST MSR is 0.
- If an execution of WRMSR to the IA32_TIME_STAMP_COUNTER MSR adds (or subtracts) value X from the TSC, the logical processor also adds (or subtracts) value X from the IA32_TSC_ADJUST MSR.
- If an execution of WRMSR to the IA32_TSC_ADJUST MSR adds (or subtracts) value X from that MSR, the logical processor also adds (or subtracts) value X from the TSC.

Unlike the TSC, the value of the IA32_TSC_ADJUST MSR changes only in response to WRMSR (either to the MSR itself, or to the IA32_TIME_STAMP_COUNTER MSR). Its value does not otherwise change as time elapses. Software seeking to adjust the TSC can do so by using WRMSR to write the same value to the IA32_TSC_ADJUST MSR on each logical processor.

Processor support for the IA32_TSC_ADJUST MSR is indicated by CPUID.(EAX=07H, ECX=0H):EBX.TSC_ADJUST (bit 1).

18.17.4 Invariant Time-Keeping

The invariant TSC is based on the invariant timekeeping hardware (called Always Running Timer or ART), that runs at the core crystal clock frequency. The ratio defined by CPUID leaf 15H expresses the frequency relationship between the ART hardware and TSC.

If CPUID.15H:EBX[31:0] != 0 and CPUID.80000007H:EDX[InvariantTSC] = 1, the following linearity relationship holds between TSC and the ART hardware:

TSC_Value = (ART_Value * CPUID.15H:EBX[31:0])/ CPUID.15H:EAX[31:0] + K

Where 'K' is an offset that can be adjusted by a privileged agent[1].

When ART hardware is reset, both invariant TSC and K are also reset.

18.18 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) MONITORING FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of monitoring capabilities including Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM). The Intel® Xeon® processor E5 v3 family introduced resource monitoring capability in each logical processor to measure specific platform shared resource metrics, for example, L3 cache occupancy. The programming interface for these monitoring features is described in this section. Two features within the monitoring feature set provided are described - Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.

[1] IA32_TSC_ADJUST MSR and the TSC-offset field in the VM execution controls of VMCS are some of the common interfaces that privileged software can use to manage the time stamp counter for keeping time

Cache Monitoring Technology (CMT) allows an Operating System, Hypervisor or similar system management agent to determine the usage of cache by applications running on the platform. The initial implementation is directed at L3 cache monitoring (currently the last level cache in most server platforms).

Memory Bandwidth Monitoring (MBM), introduced in the Intel® Xeon® processor E5 v4 family, builds on the CMT infrastructure to allow monitoring of bandwidth from one level of the cache hierarchy to the next - in this case focusing on the L3 cache, which is typically backed directly by system memory. As a result of this implementation, memory bandwidth can be monitored.

The monitoring mechanisms described provide the following key shared infrastructure features:

- A mechanism to enumerate the presence of the monitoring capabilities within the platform (via a CPUID feature bit).
- A framework to enumerate the details of each sub-feature (including CMT and MBM, as discussed later, via CPUID leaves and sub-leaves).
- A mechanism for the OS or Hypervisor to indicate a software-defined ID for each of the software threads (applications, virtual machines, etc.) that are scheduled to run on a logical processor. These identifiers are known as Resource Monitoring IDs (RMIDs).
- Mechanisms in hardware to monitor cache occupancy and bandwidth statistics as applicable to a given product generation on a per software-id basis.
- Mechanisms for the OS or Hypervisor to read back the collected metrics such as L3 occupancy or Memory Bandwidth for a given software ID at any point during runtime.

18.18.1 Overview of Cache Monitoring Technology and Memory Bandwidth Monitoring

The shared resource monitoring features described in this chapter provide a layer of abstraction between applications and logical processors through the use of **Resource Monitoring ID**s (RMIDs). Each logical processor in the system can be assigned an RMID independently, or multiple logical processors can be assigned to the same RMID value (e.g., to track an application with multiple threads). For each logical processor, only one RMID value is active at a time. This is enforced by the IA32_PQR_ASSOC MSR, which specifies the active RMID of a logical processor. Writing to this MSR by software changes the active RMID of the logical processor from an old value to a new value.

The underlying platform shared resource monitoring hardware tracks cache metrics such as cache utilization and misses as a result of memory accesses according to the RMIDs and reports monitored data via a counter register (IA32_QM_CTR). The specific event types supported vary by generation and can be enumerated via CPUID. To read back monitored data, software configures an event selection MSR (IA32_QM_EVTSEL) to specify which metric is to be reported and the specific RMID for which the data should be returned.

Processor support of the monitoring framework and sub-features such as CMT is reported via the CPUID instruction. The resource type available to the monitoring framework is enumerated via a new leaf function in CPUID. Reading and writing to the monitoring MSRs requires the RDMSR and WRMSR instructions.

The Cache Monitoring Technology feature set provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the CMT feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- CMT-specific event codes to read occupancy for a given level of the cache hierarchy.

The Memory Bandwidth Monitoring feature provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the MBM feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- MBM-specific event codes to read bandwidth out to the next level of the hierarchy and various sub-event codes to read more specific metrics as discussed later (e.g., total bandwidth vs. bandwidth only from local memory controllers on the same package).

18.18.2 Enabling Monitoring: Usage Flow

Figure 18-19 illustrates the key steps for OS/VMM to detect support of shared resource monitoring features such as CMT and enable resource monitoring for available resource types and monitoring events.

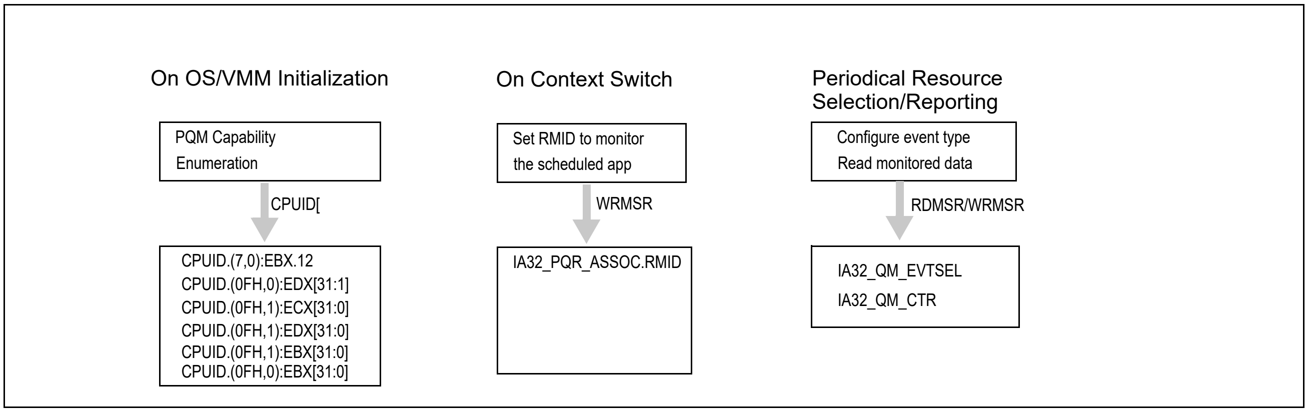


Figure 18-19. Platform Shared Resource Monitoring Usage Flow

18.18.3 Enumeration and Detecting Support of Cache Monitoring Technology and Memory Bandwidth Monitoring

Software can query processor support of shared resource monitoring features capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] reports 1, the processor provides the following programming interfaces for shared resource monitoring, including Cache Monitoring Technology:

- CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides information on available resource types (see Section 18.18.4), and monitoring capabilities for each resource type (see Section 18.18.5). Note CMT and MBM capabilities are enumerated as separate event vectors using shared enumeration infrastructure under a given resource type.
- IA32_PQR_ASSOC.RMID: The per-logical-processor MSR, IA32_PQR_ASSOC, that OS/VMM can use to assign an RMID to each logical processor, see Section 18.18.6.
- IA32_QM_EVTSEL: This MSR specifies an Event ID (EvtID) and an RMID which the platform uses to look up and provide monitoring data in the monitoring counter, IA32_QM_CTR, see Section 18.18.7.
- IA32_QM_CTR: This MSR reports monitored resource data when available along with bits to allow software to check for error conditions and verify data validity.

Software must follow the following sequence of enumeration to discover Cache Monitoring Technology capabilities:

1. Execute CPUID with EAX=0 to discover the “cpuid_maxLeaf” supported in the processor;
2. If cpuid_maxLeaf >= 7, then execute CPUID with EAX=7, ECX= 0 to verify CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] is set;
3. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] = 1, then execute CPUID with EAX=0FH, ECX= 0 to query available resource types that support monitoring;
4. If CPUID.(EAX=0FH, ECX=0):EDX.L3[bit 1] = 1, then execute CPUID with EAX=0FH, ECX= 1 to query the specific capabilities of L3 Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.
5. If CPUID.(EAX=0FH, ECX=0):EDX reports additional resource types supporting monitoring, then execute CPUID with EAX=0FH, ECX set to a corresponding resource type ID (ResID) as enumerated by the bit position of CPUID.(EAX=0FH, ECX=0):EDX.

18.18.4 Monitoring Resource Type and Capability Enumeration

CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides one sub-leaf (sub-function 0) that reports shared enumeration infrastructure, and one or more sub-functions that report feature-specific enumeration data:

- Monitoring leaf sub-function 0 enumerates available resources that support monitoring, i.e., executing CPUID with EAX=0FH and ECX=0H. In the initial implementation, L3 cache is the only resource type available. Each supported resource type is represented by a bit in CPUID.(EAX=0FH, ECX=0):EDX[31:1]. The bit position corresponds to the sub-leaf index (ResID) that software must use to query details of the monitoring capability of that resource type (see Figure 18-21 and Figure 18-22). Reserved bits of CPUID.(EAX=0FH, ECX=0):EDX[31:2] correspond to unsupported sub-leaves of the CPUID.0FH leaf. Additionally, CPUID.(EAX=0FH, ECX=0H):EBX reports the highest RMID value of any resource type that supports monitoring in the processor.

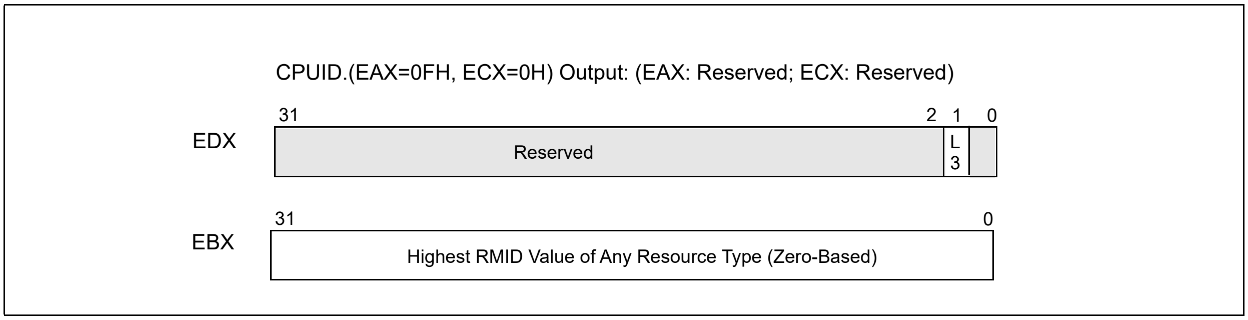

CPUID.(EAX=0FH, ECX=0H) Output: (EAX: Reserved; ECX: Reserved)
EDX
31
2
1
0
Reserved
L
3
EBX
31
0
Highest RMID Value of Any Resource Type (Zero-Based)

Figure 18-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration

18.18.5 Feature-Specific Enumeration

Each additional sub-leaf of CPUID.(EAX=0FH, ECX=ResID) enumerates the specific details for software to program monitoring MSRs using the resource type associated with the given ResID.

Note that in future Monitoring implementations the meanings of the returned registers may vary in other sub-leaves that are not yet defined. The registers will be specified and defined on a per-ResID basis.

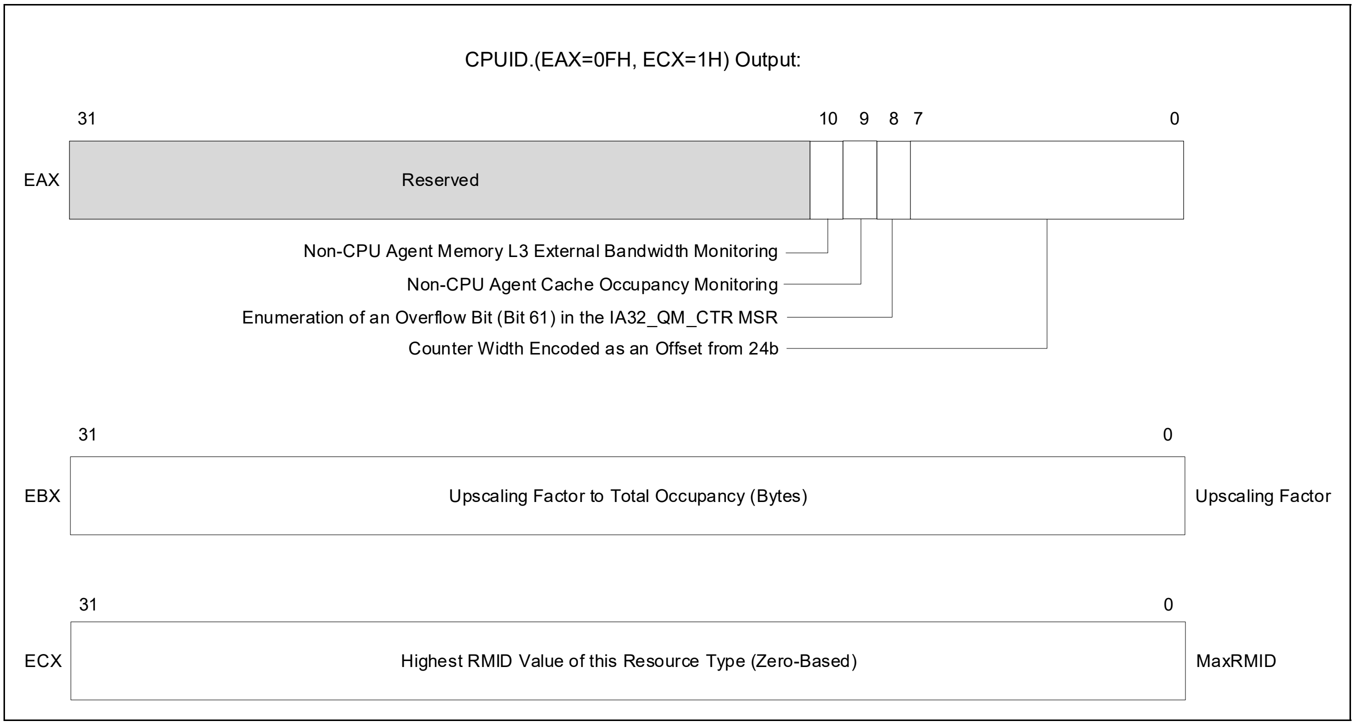

CPUID.(EAX=0FH, ECX=1H) Output:
31
10
9
8
7
0
EAX
Reserved
Non-CPU Agent Memory L3 External Bandwidth Monitoring
Non-CPU Agent Cache Occupancy Monitoring
Enumeration of an Overflow Bit (Bit 61) in the IA32_QM_CTR MSR
Counter Width Encoded as an Offset from 24b
31
0
EBX
Upscaling Factor to Total Occupancy (Bytes)
Upscaling Factor
31
0
ECX
Highest RMID Value of this Resource Type (Zero-Based)
MaxRMID

Figure 18-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H))

CPUID.(EAX=0FH, ECX=1H).EAX[7:0]: Encode counter width as offset from 24b. See Section 18.18.5.2 for details. Bits 31:11 of EAX are reserved.

CPUID.(EAX=0FH, ECX=1H).EAX[bit 8]: If 1, indicates the presence of an overflow bit in the IA32_QM_CTR MSR. See Section 18.18.5.2 for details. Bits 31:11 of EAX are reserved.

CPUID.(EAX=0FH, ECX=1H).EAX[bit 9]: If 1, indicates the presence of non-CPU agent Intel RDT CMT support. See Section 18.20 for details. Bits 31:11 of EAX are reserved.

CPUID.(EAX=0FH, ECX=1H).EAX[bit 10]:If 1, indicates the presence of non-CPU agent Intel RDT MBM support. See Section 18.20 for details. Bits 31:11 of EAX are reserved.

For each supported Cache Monitoring resource type, hardware supports only a finite number of RMIDs. CPUID.(EAX=0FH, ECX=1H).ECX enumerates the highest RMID value that can be monitored with this resource type, see Figure 18-21.

CPUID.(EAX=0FH, ECX=1H).EDX specifies a bit vector that is used to look up the EventID (See Figure 18-22 and Table 18-18) that software must program with IA32_QM_EVTSEL in order to retrieve event data. After software configures IA32_QMEVTSEL with the desired RMID and EventID, it can read the resulting data from IA32_QM_CTR. The raw numerical value reported from IA32_QM_CTR can be converted to the final value (occupancy in bytes or bandwidth in bytes per sampled time period) by multiplying the counter value by the value from CPUID.(EAX=0FH, ECX=1H).EBX, see Figure 18-21.

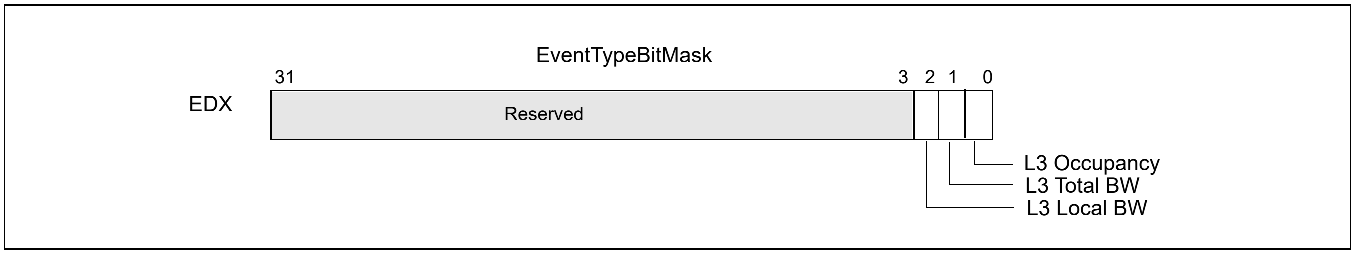


Figure 18-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H))

18.18.5.1 Cache Monitoring Technology

On processors for which Cache Monitoring Technology supports the L3 cache occupancy event, CPUID.(EAX=0FH, ECX=1H).EDX returns with bit 0 set. The corresponding event ID is shown in Table 18-18. The L3 occupancy data accumulated in the IA32_QM_CTR MSR can be converted to total occupancy (in bytes) by multiplying with CPUID.(EAX=0FH, ECX=1H).EBX.

Event codes for Cache Monitoring Technology are discussed in the next section.

18.18.5.2 Memory Bandwidth Monitoring

On processors that monitoring supports Memory Bandwidth Monitoring using ResID=1 (L3), two additional bits are defined in the vector at CPUID.(EAX=0FH, ECX=1H).EDX:

- CPUID.(EAX=0FH, ECX=1H).EDX[bit 1]: indicates the L3 total external bandwidth monitoring event is supported if set. This event monitors the L3 total external bandwidth to the next level of the cache hierarchy, including all demand and prefetch misses from the L3 to the next hierarchy of the memory system. In most platforms, this represents memory bandwidth.
- CPUID.(EAX=0FH, ECX=1H).EDX[bit 2]: indicates L3 local memory bandwidth monitoring event is supported if set. This event monitors the L3 external bandwidth satisfied by the local memory. In most platforms that support this event, L3 requests are likely serviced by a memory system with non-uniform memory architecture. This allows bandwidth to off-package memory resources to be tracked by subtracting local from total bandwidth (for instance, bandwidth over QPI to a memory controller on another physical processor could be tracked by subtraction). Note that it is not possible to read the local and total bandwidth atomically; multiple operations are needed. Because of this, it is possible for the counters to change in between the two reads.

The corresponding Event ID is shown in Table 18-18. The L3 bandwidth data accumulated in IA32_QM_CTR can be converted to total bandwidth (in bytes) using CPUID.(EAX=0FH, ECX=1H).EBX.

Table 18-18. Monitoring Supported Event IDs

Event Type	Event ID	Context
L3 Cache Occupancy	01H	Cache Monitoring Technology
L3 Total External Bandwidth	02H	MBM
L3 Local External Bandwidth	03H	MBM
Reserved	All other event codes	N/A

A field is added to CPUID to enumerate the MBM counter width in platforms that support the extensible MBM counter width feature.

- CPUID.(EAX=0FH, ECX=1H).EAX[7:0]: Encode counter width as offset from 24b in bits[7:0]. In EAX bits 7:0, the counter width is encoded as an offset from 24b. A value of zero in this field means 24-bit counters are supported. A value of 8 indicates that 32-bit counters are supported, as in the 3rd generation Intel Xeon Scalable Processor Family. With this enumerable counter width, the requirement that software polls at 1Hz is removed. Software may poll at a varying rate with a reduced risk of rollover. Under typical conditions, rollover will likely require hundreds of seconds (though this value is not explicitly specified and may vary and decrease in future processor generations as memory bandwidths increase). Suppose software seeks to ensure that rollover does not occur more than once between samples. In that case, sampling at 1Hz while consuming the enumerated counter widths' worth of data will provide this guarantee for a specific platform and counter width under all conditions.
- CPUID.(EAX=0FH, ECX=1H).EAX[8]: Enumeration of the presence of an overflow bit in the IA32_QM_CTR MSR via EAX bit[8]. Software that uses the MBM event retrieval MSR interface should be updated to comprehend this new format, which enables up to 62-bit MBM counters to be provided by future platforms. Higher-level software that consumes the resulting bandwidth values is not expected to be affected. An overflow bit is defined in the IA32_QM_CTR MSR, bit 61, if CPUID.(EAX=0FH, ECX=1H).EAX[bit 8] is set. This rollover bit will be set on overflow of the MBM counters and reset upon read. Current processors do not support this capability.

18.18.6 Monitoring Resource RMID Association

After Monitoring and sub-features have been enumerated, software can begin using the monitoring features. The first step is to associate a given software thread (or multiple threads as part of an application, VM, group of applications or other abstraction) with an RMID.

Note that the process of associating an RMID with a given software thread is the same for all shared resource monitoring features (CMT, MBM), and a given RMID number has the same meaning from the viewpoint of any logical processors in a package. Stated another way, a thread may be associated in a 1:1 mapping with an RMID, and that RMID may allow cache occupancy, memory bandwidth information or other monitoring data to be read back later with monitoring event codes (retrieving data is discussed in a previous section).

The association of an application thread with an RMID requires an OS to program the per-logical-processor MSR IA32_PQR_ASSOC at context swap time (updates may also be made at any other arbitrary points during program execution such as application phase changes). The IA32_PQR_ASSOC MSR specifies the active RMID that monitoring hardware will use to tag internal operations, such as L3 cache requests. The layout of the MSR is shown in Figure 18-23. Software specifies the active RMID to monitor in the IA32_PQR_ASSOC.RMID field. The width of the RMID field can vary from one implementation to another, and is derived from Ceil (LOG_2 (1 + CPUID.(EAX=0FH, ECX=0):EBX[31:0])). The value of IA32_PQR_ASSOC after power-on is 0.

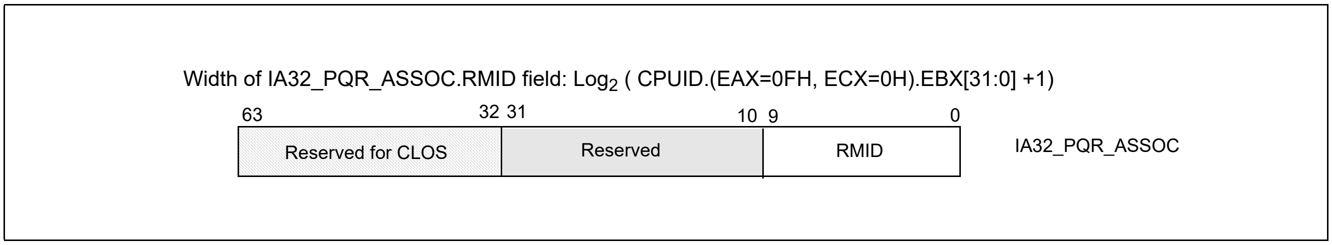

Width of IA32_PQR_ASSOC.RMID field: Log_2 (CPUID.(EAX=0FH, ECX=0H).EBX[31:0] +1)
63
32 31
10 9
0
Reserved for CLOS
Reserved
RMID
IA32_PQR_ASSOC

Figure 18-23. IA32_PQR_ASSOC MSR

In the initial implementation, the width of the RMID field is up to 10 bits wide, zero-referenced and fully encoded. However, software must use CPUID to query the maximum RMID supported by the processor. If a value larger than the maximum RMID is written to IA32_PQR_ASSOC.RMID, a #GP(0) fault will be generated.

RMIDs have a global scope within the physical package- if an RMID is assigned to one logical processor then the same RMID can be used to read multiple thread attributes later (for example, L3 cache occupancy or external bandwidth from the L3 to the next level of the cache hierarchy). In a multiple LLC platform the RMIDs are to be reassigned by the OS or VMM scheduler when an application is migrated across LLCs.

Note that in a situation where Monitoring supports multiple resource types, some upper range of RMIDs (e.g., RMID 31) may only be supported by one resource type but not by another resource type.

18.18.7 Monitoring Resource Selection and Reporting Infrastructure

The reporting mechanism for Cache Monitoring Technology and other related features is architecturally exposed as an MSR pair that can be programmed and read to measure various metrics such as the L3 cache occupancy (CMT) and bandwidths (MBM) depending on the level of Monitoring support provided by the platform. Data is reported back on a per-RMID basis. These events do not trigger based on event counts or trigger APIC interrupts (e.g., no Performance Monitoring Interrupt occurs based on counts). Rather, they are used to sample counts explicitly.

The MSR pair for the shared resource monitoring features (CMT, MBM) is separate from and not shared with architectural Perfmon counters, meaning software can use these monitoring features simultaneously with the Perfmon counters.

Access to the aggregated monitoring information is accomplished through the following programmable monitoring MSRs:

- IA32_QM_EVTSEL: This MSR provides a role similar to the event select MSRs for programmable performance monitoring described in Chapter 18. The simplified layout of the MSR is shown in Figure 18-24. IA32_QM_EVTSEL.EvtID (bits 7:0) specifies an event code of a supported resource type for hardware to report monitored data associated with IA32_QM_EVTSEL.RMID (bits 41:32). Software can configure IA32_QM_EVTSEL.RMID with any RMID that is active within the physical processor. The width of IA32_QM_EVTSEL.RMID matches that of IA32_PQR_ASSOC.RMID. Supported event codes for the IA32_QM_EVTSEL register are shown in Table 18-18. Note that valid event codes may not necessarily map directly to the bit position used to enumerate support for the resource via CPUID.

  Software can program an RMID / Event ID pair into the IA32_QM_EVTSEL MSR bit field to select an RMID to read a particular counter for a given resource. The currently supported list of Monitoring Event IDs is discussed in Section 18.18.5, which covers feature-specific details.

  Thread access to the IA32_QM_EVTSEL and IA32_QM_CTR MSR pair should be serialized (that is, treated as a critical section under lock) to avoid situations where one thread changes the RMID/EvtID just before another thread reads monitoring data from IA32_QM_CTR.
- IA32_QM_CTR: This MSR reports monitored data when available. It contains three bit fields. If software configures an unsupported RMID or event type in IA32_QM_EVTSEL, then IA32_QM_CTR.Error (bit 63) will be set, indicating there is no valid data to report. If IA32_QM_CTR.Unavailable (bit 62) is set, it indicates monitored data for the RMID is not available, and IA32_QM_CTR.data (bits 61:0) should be ignored. Therefore, IA32_QM_CTR.data (bits 61:0) is valid only if bits 63 and 62 are both clear. The IA32_QM_CTR.Overflow (bit 61) is present if CPUID.(EAX = 0FH, ECX=1):EAX[bit 8] is set. This bit is set on overflow of the MBM counters and will reset upon read. For Cache Monitoring Technology, software can convert IA32_QM_CTR.data into cache occupancy or bandwidth metrics expressed in bytes by multiplying with the conversion factor from CPUID.(EAX=0FH, ECX=1H).EBX.

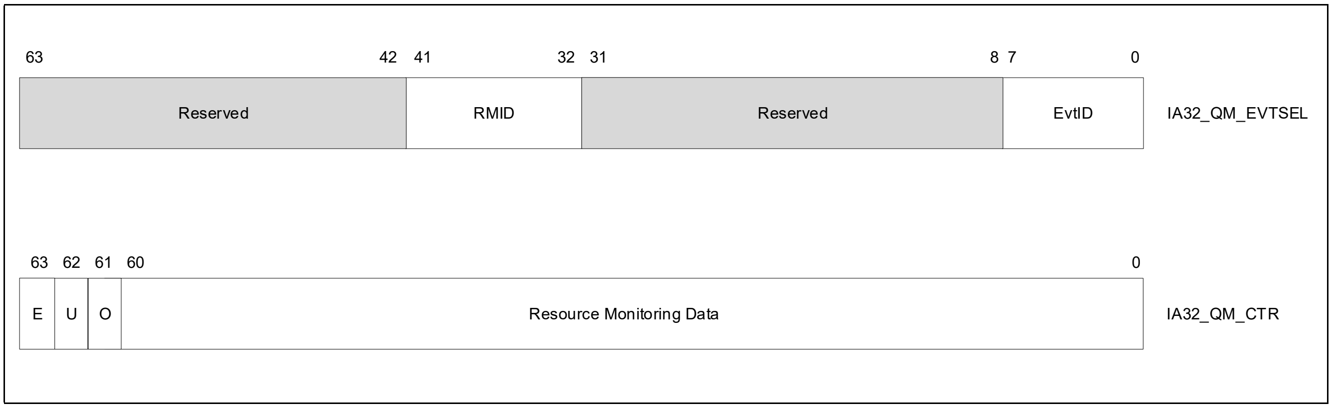


Figure 18-24. IA32_QM_EVTSEL and IA32_QM_CTR MSRs

18.18.8 Monitoring Programming Considerations

Figure 18-25 illustrates how system software can program IA32_QOSEVTSEL and IA32_QM_CTR to perform resource monitoring.

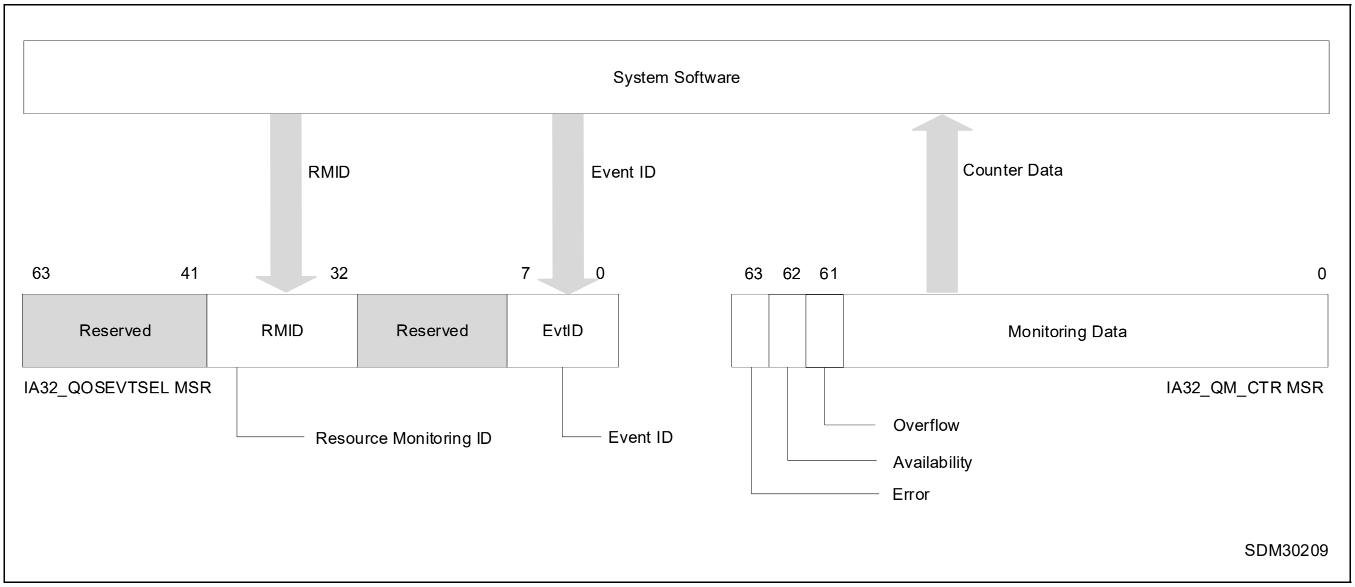


Figure 18-25. Software Usage of Cache Monitoring Resources

Though the field provided in IA32_QM_CTR allows for up to 62 bits of data to be returned, often a subset of bits are used. With Cache Monitoring Technology for instance, the number of bits used is the base-two logarithm of the total cache size divided by the Upscaling Factor from CPUID.

In Memory Bandwidth Monitoring, the initial counter size is 24 bits, and retrieving the value at 1Hz or faster is sufficient to ensure at most one rollover per sampling period. Any changes to counter width are enumerated to software; see Section 18.18.5.2 for details.

18.18.8.1 Monitoring Dynamic Configuration

Both the IA32_QM_EVTSEL and IA32_PQR_ASSOC registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,

- An RMID exceeding the maximum RMID is used.

18.18.8.2 Monitoring Operation With Power Saving Features

Some advanced power management features such as deep package C-states may shrink the L3 cache and cause CMT occupancy count to be reduced. MBM bandwidth counts may increase due to flushing cached data out of L3.

18.18.8.3 Monitoring Operation with Other Operating Modes

The states in IA32_PQR_ASSOC and monitoring counter are unmodified across an SMI delivery. Thus, the execution of SMM handler code and SMM handler's data can manifest as spurious contribution in the monitored data.

It is possible for an SMM handler to minimize the impact on of spurious contribution in the QOS monitoring counters by reserving a dedicated RMID for monitoring the SMM handler. Such an SMM handler can save the previously configured QOS Monitoring state immediately upon entering SMM, and restoring the QOS monitoring state back to the prev-SMM RMID upon exit.

18.18.8.4 Monitoring Operation with RAS Features

In general, the Reliability, Availability, and Serviceability (RAS) features present in Intel Platforms are not expected to significantly affect shared resource monitoring counts. In cases where software RAS features cause memory copies or cache accesses, these may be tracked and may influence the shared resource monitoring counter values.

18.19 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) ALLOCATION FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of allocation (resource control) capabilities including Cache Allocation Technology (CAT) and Code and Data Prioritization (CDP). The Intel Xeon processor E5 v4 family (and a subset of communication-focused processors in the Intel Xeon E5 v3 family) introduce capabilities to configure and make use of the Cache Allocation Technology (CAT) mechanisms on the L3 cache. Certain Intel Atom processors also provide support for control over the L2 cache, with capabilities as described below. The programming interface for Cache Allocation Technology and for the more general allocation capabilities are described in the rest of this chapter. The CAT and CDP capabilities, where architecturally supported, may be detected and enumerated in software using the CPUID instruction, as described in this chapter.

The Intel Xeon Scalable Processor Family introduces the Memory Bandwidth Allocation (MBA) feature which provides indirect control over the memory bandwidth available to CPU cores, and is discussed later in this chapter.

18.19.1 Introduction to Cache Allocation Technology (CAT)

Cache Allocation Technology enables an Operating System (OS), Hypervisor /Virtual Machine Manager (VMM) or similar system service management agent to specify the amount of cache space into which an application can fill (as a hint to hardware - certain features such as power management may override CAT settings). Specialized user-level implementations with minimal OS support are also possible, though not necessarily recommended (see notes below for OS/Hypervisor with respect to ring 3 software and virtual guests). Depending on the processor family, L2 or L3 cache allocation capability may be provided, and the technology is designed to scale across multiple cache levels and technology generations.

Software can determine which levels are supported in a given platform programmatically using CPUID as described in the following sections.

The CAT mechanisms defined in this document provide the following key features:

- A mechanism to enumerate platform Cache Allocation Technology capabilities and available resource types that provides CAT control capabilities. For implementations that support Cache Allocation Technology, CPUID provides enumeration support to query which levels of the cache hierarchy are supported and specific CAT capabilities, such as the max allocation bitmask size.

- A mechanism for the OS or Hypervisor to configure the amount of a resource available to a particular Class of Service via a list of allocation bitmasks.
- Mechanisms for the OS or Hypervisor to signal the Class of Service to which an application belongs.
- Hardware mechanisms to guide the LLC fill policy when an application has been designated to belong to a specific Class of Service.

Note that for many usages, an OS or Hypervisor may not want to expose Cache Allocation Technology mechanisms to Ring3 software or virtualized guests.

The Cache Allocation Technology feature enables more cache resources (i.e., cache space) to be made available for high priority applications based on guidance from the execution environment as shown in Figure 18-26. The architecture also allows dynamic resource reassignment during runtime to further optimize the performance of the high priority application with minimal degradation to the low priority app. Additionally, resources can be rebalanced for system throughput benefit across uses cases of OSes, VMMs, containers, and other scenarios by managing the CPUID and MSR interfaces. This section describes the hardware and software support required in the platform including what is required of the execution environment (i.e., OS/VMM) to support such resource control. Note that in Figure 18-26 the L3 Cache is shown as an example resource.

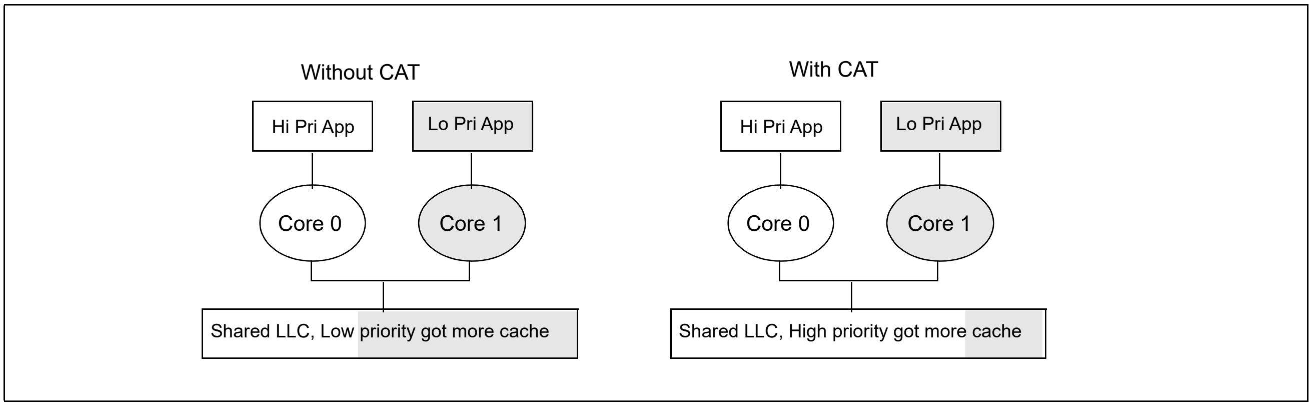


Figure 18-26. Cache Allocation Technology Enables Allocation of More Resources to High Priority Applications

18.19.2 Cache Allocation Technology Architecture

The fundamental goal of Cache Allocation Technology is to enable resource allocation based on application priority or Class of Service (CLOS). The processor exposes a set of Classes of Service into which applications (or individual threads) can be assigned. Cache allocation for the respective applications or threads is then restricted based on the class with which they are associated. Each Class of Service can be configured using capacity bitmasks (CBMs) which represent capacity and indicate the degree of overlap and isolation between classes. For each logical processor there is a register exposed (referred to here as the IA32_PQR_ASSOC MSR or PQR) to allow the OS/VMM to specify a CLOS when an application, thread or VM is scheduled.

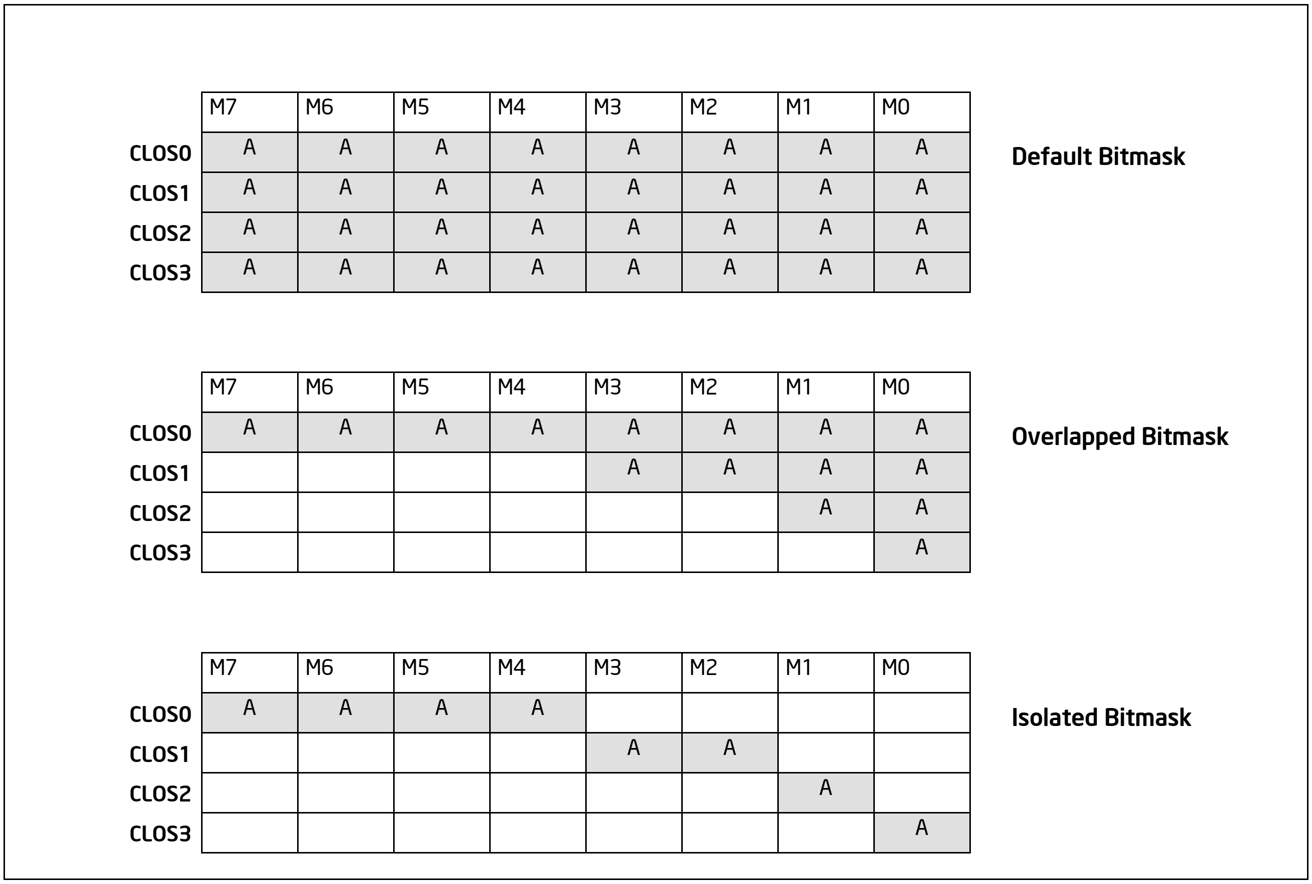
The usage of Classes of Service (CLOS) are consistent across resources and a CLOS may have multiple resource control attributes attached, which reduces software overhead at context swap time. Rather than adding new types of CLOS tags per resource for instance, the CLOS management overhead is constant. Cache allocation for the indicated application/thread/container/VM is then controlled automatically by the hardware based on the class and the bitmask associated with that class. Bitmasks are configured via the IA32_resourceType_MASK_n MSRs, where resourceType indicates a resource type (e.g., “L3” for the L3 cache) and “n” indicates a CLOS number.

The basic ingredients of Cache Allocation Technology are as follows:

- An architecturally exposed mechanism using CPUID to indicate whether CAT is supported, and what resource types are available which can be controlled.
- For each available resourceType, CPUID also enumerates the total number of Classes of Services and the length of the capacity bitmasks that can be used to enforce cache allocation to applications on the platform.
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to configure the behavior of different classes of service using the bitmasks available.

- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to assign a CLOS to an executing software thread (i.e., associating the active CR3 of a logical processor with the CLOS in IA32_PQR_ASSOC).
- Implementation-dependent mechanisms to indicate which CLOS is associated with a memory access and to enforce the cache allocation on a per CLOS basis.

A capacity bitmask (CBM) provides a hint to the hardware indicating the cache space an application should be limited to as well as providing an indication of overlap and isolation in the CAT-capable cache from other applications contending for the cache. The bit length of the capacity mask available generally depends on the configuration of the cache and is specified in the enumeration process for CAT in CPUID (this may vary between models in a processor family as well). Similarly, other parameters such as the number of supported CLOS may vary for each resource type, and these details can be enumerated via CPUID.



	M7	M6	M5	M4	M3	M2	M1	M0
CLOS0	A	A	A	A	A	A	A	A
CLOS1	A	A	A	A	A	A	A	A
CLOS2	A	A	A	A	A	A	A	A
CLOS3	A	A	A	A	A	A	A	A

Default Bitmask

	M7	M6	M5	M4	M3	M2	M1	M0
CLOS0	A	A	A	A	A	A	A	A
CLOS1					A	A	A	A
CLOS2							A	A
CLOS3								A

Overlapped Bitmask

	M7	M6	M5	M4	M3	M2	M1	M0
CLOS0	A	A	A	A				
CLOS1					A	A		
CLOS2							A	
CLOS3								A

Isolated Bitmask

Figure 18-27. Examples of Cache Capacity Bitmasks

Sample cache capacity bitmasks for a bit length of 8 are shown in Figure 18-27. Note that all (and only) contiguous '1' combinations are allowed (e.g., FFFFH, 0FF0H, 003CH, etc.), unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type. Attempts to program a value without contiguous '1's (including zero) will result in a general protection fault (#GP(0)). It is generally expected that in way-based implementations, one capacity mask bit corresponds to some number of ways in cache, but the specific mapping is implementation-dependent. In all cases, a mask bit set to '1' specifies that a particular Class of Service can allocate into the cache subset represented by that bit. A value of '0' in a mask bit specifies that a Class of Service cannot allocate into the given cache subset. In general, allocating more cache to a given application is usually beneficial to its performance.

Figure 18-27 also shows three examples of sets of Cache Capacity Bitmasks. For simplicity these are represented as 8-bit vectors, though this may vary depending on the implementation and how the mask is mapped to the available cache capacity. The first example shows the default case where all 4 Classes of Service (the total number of CLOS are implementation-dependent) have full access to the cache. The second case shows an overlapped case, which would allow some lower-priority threads to share cache space with the highest priority threads. The third case shows various non-overlapped partitioning schemes. As a matter of software policy for extensibility, CLOS0 should typically be considered and configured as the highest priority CLOS, followed by CLOS1, and so on, though there is no hardware restriction enforcing this mapping. When the system boots all threads are initialized to CLOS0, which has full access to the cache by default.

Though the representation of the CBMs looks similar to a way-based mapping they are independent of any specific enforcement implementation (e.g., way partitioning.) Rather, this is a convenient manner to represent capacity, overlap, and isolation of cache space. For example, executing a POPCNT instruction (population count of set bits) on the capacity bitmask can provide the fraction of cache space that a class of service can allocate into. In addition to the fraction, the exact location of the bits also shows whether the class of service overlaps with other classes of service or is entirely isolated in terms of cache space used.

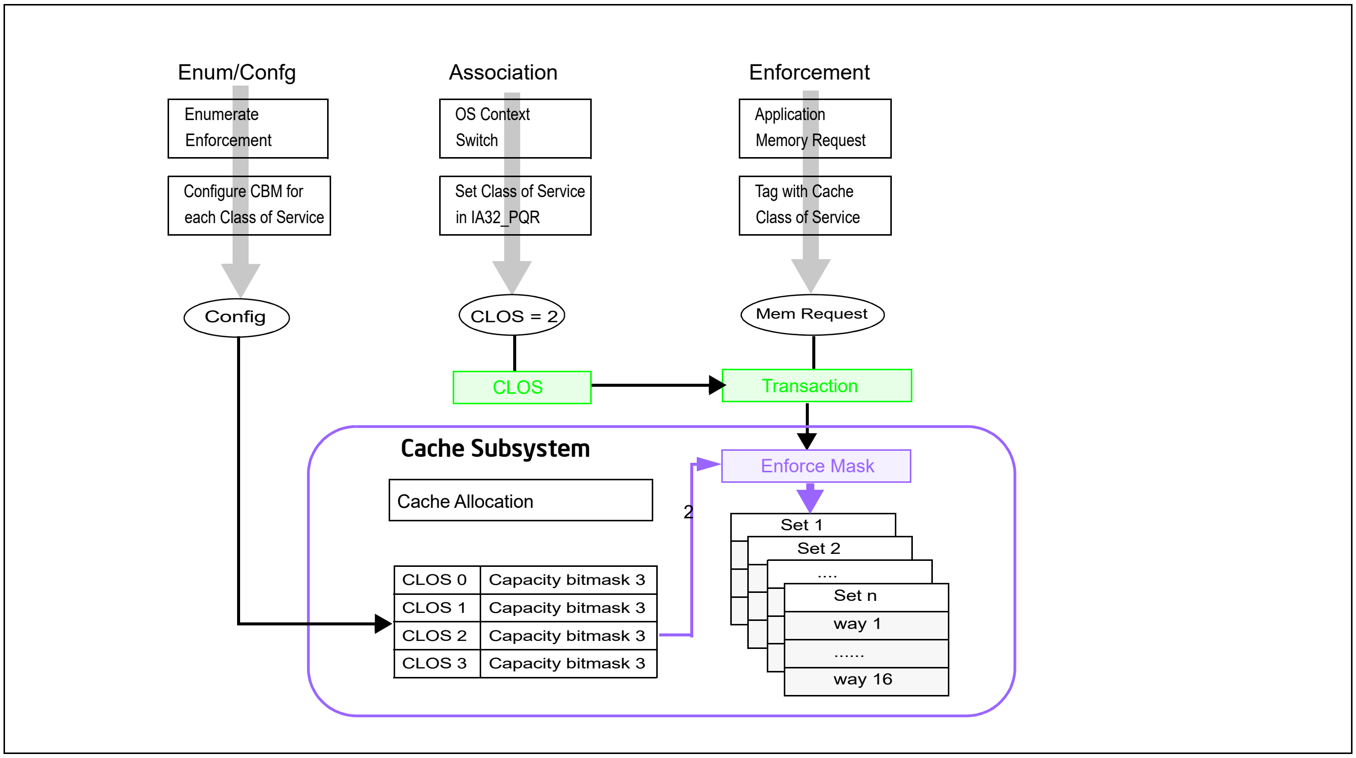


Figure 18-28. Class of Service and Cache Capacity Bitmasks

Figure 18-28 shows how the Cache Capacity Bitmasks and the per-logical-processor Class of Service are logically used to enable Cache Allocation Technology. All (and only) contiguous 1's in the CBM are permitted, unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type. The length of a CBM may vary from resource to resource or between processor generations and can be enumerated using CPUID. From the available mask set and based on the goals of the OS/VMM (shared or isolated cache, etc.) bitmasks are selected and associated with different classes of service. For the available Classes of Service the associated CBMs can be programmed via the global set of CAT configuration registers (in the case of L3 CAT, via the IA32_L3_MASK_n MSRs, where "n" is the Class of Service, starting from zero). In all architectural implementations supporting CPUID it is possible to change the CBMs dynamically, during program execution, unless stated otherwise by Intel.

The currently running application's Class of Service is communicated to the hardware through the per-logical-processor PQR MSR (IA32_PQR_ASSOC MSR). When the OS schedules an application thread on a logical processor, the application thread is associated with a specific CLOS (i.e., the corresponding CLOS in the PQR) and all requests to the CAT-capable resource from that logical processor are tagged with that CLOS (in other words, the application thread is configured to belong to a specific CLOS). The cache subsystem uses this tagged request information to enforce QoS. The capacity bitmask may be mapped into a way bitmask (or a similar enforcement entity based on the implementation) at the cache before it is applied to the allocation policy. For example, the capacity bitmask can be an 8-bit mask and the enforcement may be accomplished using a 16-way bitmask for a cache enforcement implementation based on way partitioning.

The following sections describe extensions of CAT such as Code and Data Prioritization (CDP), followed by details on specific features such as L3 CAT, L3 CDP, L2 CAT, and L2 CDP. Depending on the specific processor a mix of features may be supported, and CPUID provides enumeration capabilities to enable software to dynamically detect the set of supported features.

18.19.3 Code and Data Prioritization (CDP) Technology

Code and Data Prioritization Technology is an extension of CAT. CDP enables isolation and separate prioritization of code and data fetches to the L2 or L3 cache in a software configurable manner, depending on hardware support, which can enable workload prioritization and tuning of cache capacity to the characteristics of the workload. CDP extends Cache Allocation Technology (CAT) by providing separate code and data masks per Class of Service (CLOS). Support for the L2 CDP feature and the L3 CDP features are separately enumerated (via CPUID) and separately controlled (via remapping the L2 CAT MSRs or L3 CAT MSRs respectively). Section 18.19.6.3 and Section 18.19.7 provide details on enumerating, controlling, and enabling L3 and L2 CDP respectively, while this section provides a general overview.

The L3 CDP feature was first introduced on the Intel Xeon E5 v4 family of server processors, as an extension to L3 CAT. The L2 CDP feature is first introduced on future Intel Atom family processors, as an extension to L2 CAT.

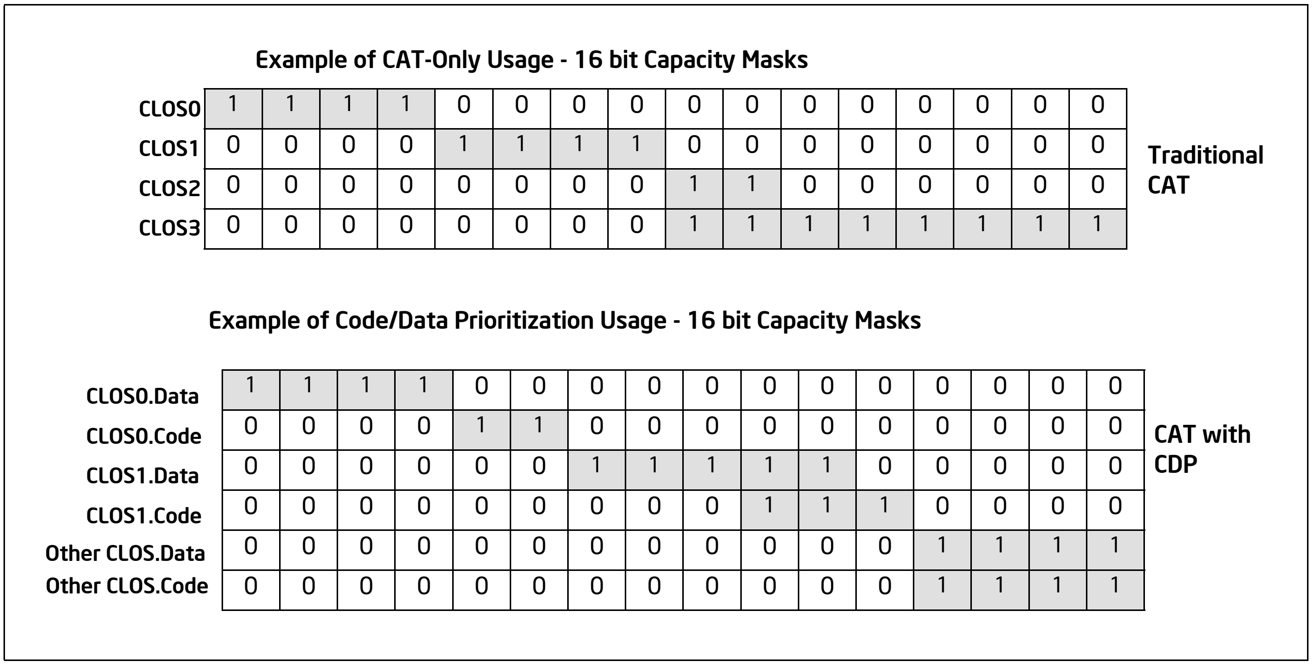
By default, CDP is disabled on the processor. If the CAT MSRs are used without enabling CDP, the processor operates in a traditional CAT-only mode. When CDP is enabled:

- The CAT mask MSRs are re-mapped into interleaved pairs of mask MSRs for data or code fetches (see Figure 18-29).
- The range of CLOS for CAT is re-indexed, with the lower-half of the CLOS range available for CDP.

Using the CDP feature, virtual isolation between code and data can be configured on the L2 or L3 cache if desired, similar to how some processor cache levels provide separate L1 data and L1 instruction caches.

Like the CAT feature, CDP may be dynamically configured by privileged software at any point during normal system operation, including dynamically enabling or disabling the feature provided that certain software configuration requirements are met (see Section 18.19.5).

An example of the operating mode of CDP is shown in Figure 18-29. Shown at the top are traditional CAT usage models where capacity masks map 1:1 with a CLOS number to enable control over the cache space which a given CLOS (and thus applications, threads or VMs) may occupy. Shown at the bottom are example mask configurations where CDP is enabled, and each CLOS number maps 1:2 to two masks, one for code and one for data. This enables code and data to be either overlapped or isolated to varying degrees either globally or on a per-CLOS basis, depending on application and system needs.



Example of CAT-Only Usage - 16 bit Capacity Masks

CLOS0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
CLOS1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
CLOS2	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
CLOS3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Traditional CAT

Example of Code/Data Prioritization Usage - 16 bit Capacity Masks

CLOS0.Data	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
CLOS0.Code	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
CLOS1.Data	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
CLOS1.Code	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
Other CLOS.Data	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
Other CLOS.Code	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

CAT with CDP

Figure 18-29. Code and Data Capacity Bitmasks of CDP

When CDP is enabled, the existing mask space for CAT-only operation is split. As an example if the system supports 16 CAT-only CLOS, when CDP is enabled the same MSR interfaces are used, however half of the masks correspond to code, half correspond to data, and the effective number of CLOS is reduced by half. Code/Data masks are defined per-CLOS and interleaved in the MSR space as described in subsequent sections.

In cases where CPUID exposes a non-even number of supported Classes of Service for the CAT or CDP features, software using CDP should use the lower matched pairs of code/data masks, and any upper unpaired masks should not be used. As an example, if CPUID exposes 5 CLOS, when CDP is enabled then two code/data pairs are available (masks 0/1 for CLOS[0] data/code and masks 2/3 for CLOS[1] data/code), however the upper un-paired mask should not be used (mask 4 in this case) or undefined behavior may result.

18.19.4 Enabling Cache Allocation Technology Usage Flow

Figure 18-30 illustrates the key steps for OS/VMM to detect support of Cache Allocation Technology and enable priority-based resource allocation for a CAT-capable resource.

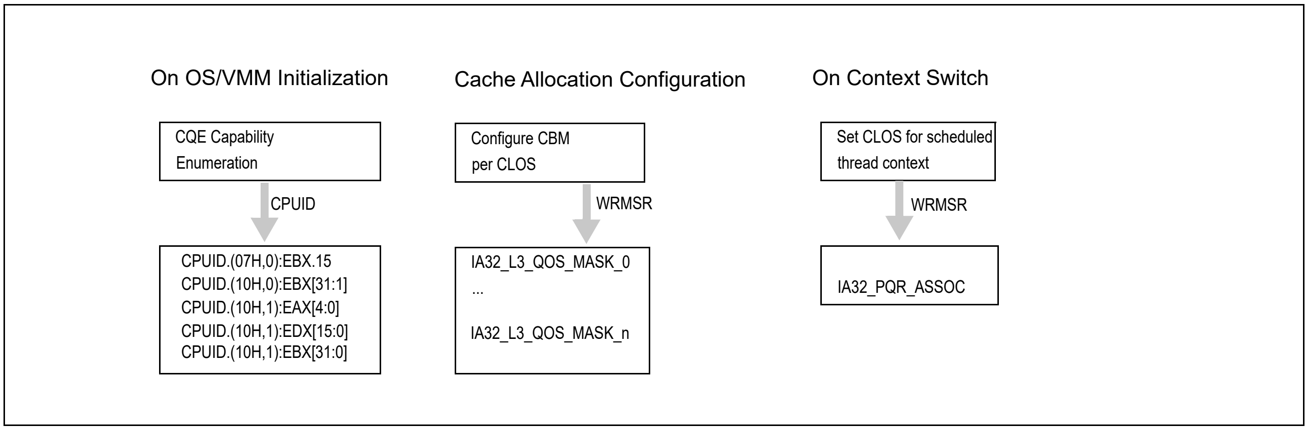


Figure 18-30. Cache Allocation Technology Usage Flow

Enumeration and configuration of L2 CAT is similar to L3 CAT, however CPUID details and MSR addresses differ. Common CLOS are used across the features.

18.19.4.1 Enumeration and Detection Support of Cache Allocation Technology

Software can query processor support of CAT capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software must use CPUID leaf 10H to enumerate additional details of available resource types, classes of services and capability bitmasks. The programming interfaces provided by Cache Allocation Technology include:

- CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) and its sub-functions provide information on available resource types, and CAT capability for each resource type (see Section 18.19.4.2).
- IA32_L3_MASK_n: A range of MSRs is provided for each resource type, each MSR within that range specifying a software-configured capacity bitmask for each class of service. For L3 with Cache Allocation support, the CBM is specified using one of the IA32_L3_QOS_MASK_n MSR, where 'n' corresponds to a number within the supported range of CLOS, i.e., the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive. See Section 18.19.4.3 for details.
- IA32_L2_MASK_n: A range of MSRs is provided for L2 Cache Allocation Technology, enabling software control over the amount of L2 cache available for each CLOS. Similar to L3 CAT, a CBM is specified for each CLOS using the set of registers, IA32_L2_QOS_MASK_n MSR, where 'n' ranges from zero to the maximum CLOS number reported for L2 CAT in CPUID. See Section 18.19.4.3 for details.

  The L2 mask MSRs are scoped at the same level as the L2 cache (similarly, the L3 mask MSRs are scoped at the same level as the L3 cache). Software may determine which logical processors share an MSR (for instance local to a core, or shared across multiple cores) by performing a write to one of these MSRs and noting which logical threads observe the change. Example flows for a similar method to determine register scope are described in Section 16.5.2, "System Software Recommendation for Managing CMCI and Machine Check Resources." Software may also use CPUID leaf 4 to determine the maximum number of logical processor IDs that may share a given level of the cache.
- IA32_PQR_ASSOC.CLOS: The IA32_PQR_ASSOC MSR provides a CLOS field that OS/VMM can use to assign a logical processor to an available CLOS. The set of CLOS are common across all allocation features, meaning that multiple features may be supported in the same processor without additional software CLOS management overhead at context swap time. See Section 18.19.4.4 for details.

18.19.4.2 Cache Allocation Technology: Resource Type and Capability Enumeration

CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) provides two or more sub-functions:

- CAT Enumeration leaf sub-function 0 enumerates available resource types that support allocation control, i.e., by executing CPUID with EAX=10H and ECX=0H. Each supported resource type is represented by a bit field in CPUID.(EAX=10H, ECX=0):EBX[31:1]. The bit position of each set bit corresponds to a Resource ID (ResID), for instance ResID=1 is used to indicate L3 CAT support, and ResID=2 indicates L2 CAT support. The ResID is also the sub-leaf index that software must use to query details of the CAT capability of that resource type (see Figure 18-31).

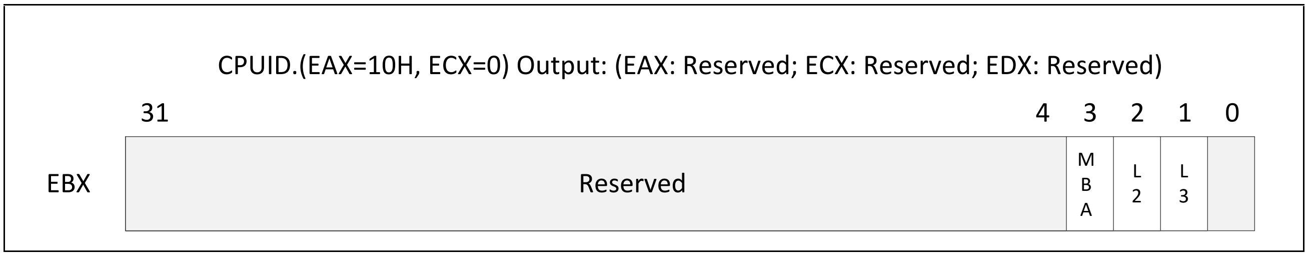

CPUID.(EAX=10H, ECX=0) Output: (EAX: Reserved; ECX: Reserved; EDX: Reserved)
31
4
3
2
1
0
EBX
Reserved
MBA
L2
L3

Figure 18-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification

- For ECX>0, EAX[4:0] reports the length of the capacity bitmask (ECX=1 or 2 for L3 CAT or L2 CAT respectively). Add one to the return value to get the result, e.g., a value of 15 corresponds to the capacity bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

- Sub-functions of CPUID.EAX=10H with a non-zero ECX input matching a supported ResID enumerate the specific enforcement details of the corresponding ResID. The capabilities enumerated include the length of the capacity bitmasks and the number of Classes of Service for a given ResID. Software should query the capability of each available ResID that supports CAT from a sub-leaf of leaf 10H using the sub-leaf index reported by the corresponding non-zero bit in CPUID.(EAX=10H, ECX=0):EBX[31:1] in order to obtain additional feature details.

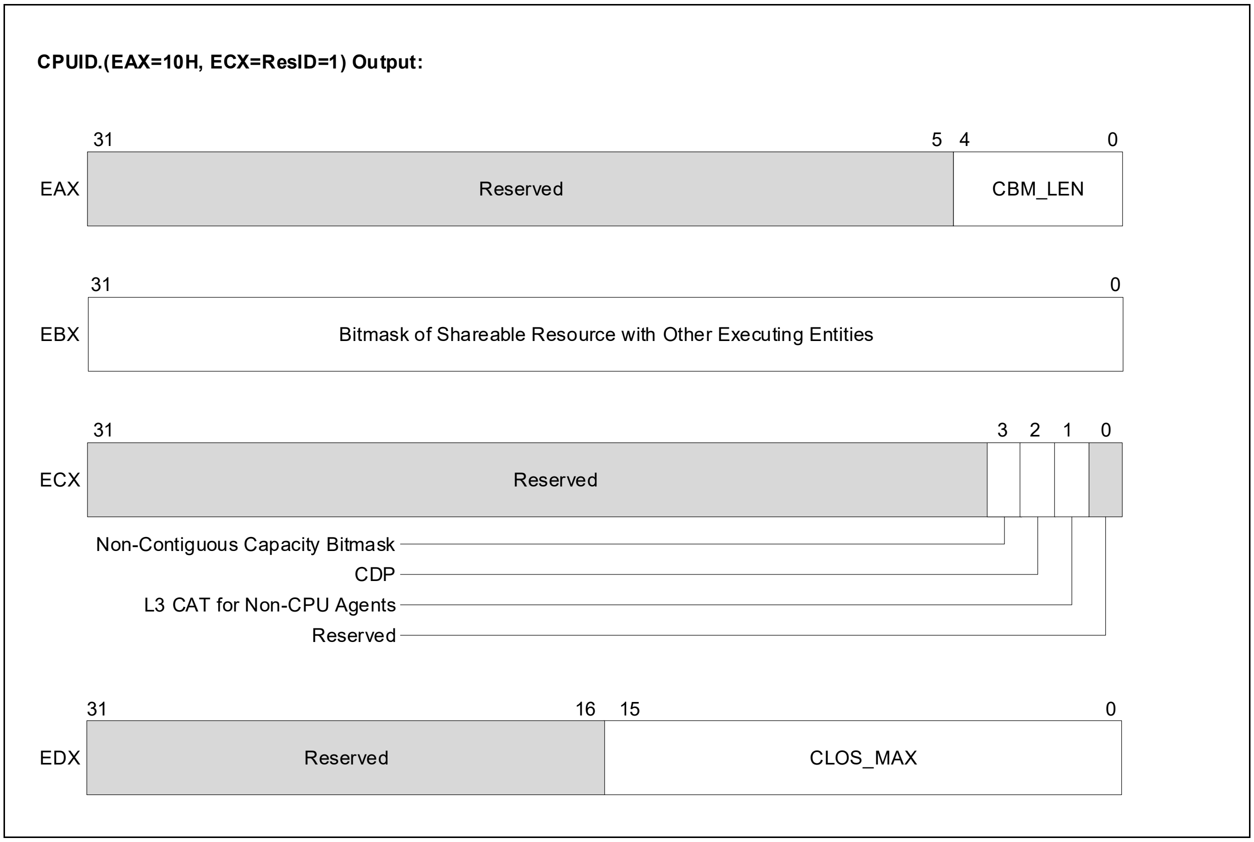

CPUID.(EAX=10H, ECX=ResID=1) Output:
EAX
31
5
4
0
Reserved
CBM_LEN
EBX
31
0
Bitmask of Shareable Resource with Other Executing Entities
ECX
31
3
2
1
0
Reserved
Non-Contiguous Capacity Bitmask
CDP
L3 CAT for Non-CPU Agents
Reserved
EDX
31
16
15
0
Reserved
CLOS_MAX

Figure 18-32. L3 Cache Allocation Technology and CDP Enumeration

- CAT capability for L3 is enumerated by CPUID.(EAX=10H, ECX=1H), see Figure 18-32. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=1) are:
  - CPUID.(EAX=10H, ECX=ResID=1):EAX[4:0] reports the length of the capacity bitmask. Add one to the return value to get the result, e.g., a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.
  - CPUID.(EAX=10H, ECX=1):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L3 allocation may be used by other entities in the platform (e.g., an integrated graphics engine or hardware units outside the processor core and have direct access to L3). Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.
  - CPUID.(EAX=10H, ECX=1):ECX[bit 1]: If 1, indicates L3 CAT for non-CPU agents is supported. Bits 0 and 31:4 of ECX are reserved. See section 18.20 for details.

- CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2]: If 1, indicates L3 Code and Data Prioritization Technology is supported (see Section 18.19.5). Bits 0 and 31:4 of ECX are reserved.
- CPUID.(EAX=10H, ECX=1):ECX[bit 3]: If 1, indicates non-contiguous capacity bitmask is supported. The bits that are set in the various IA32_L3_MASK_n registers do not have to be contiguous. Bits 0 and 31:4 of ECX are reserved.
- CPUID.(EAX=10H, ECX=1):EDX[15:0] reports the maximum CLOS supported for the resource (CLOS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported CLOS). Bits 31:16 are reserved.

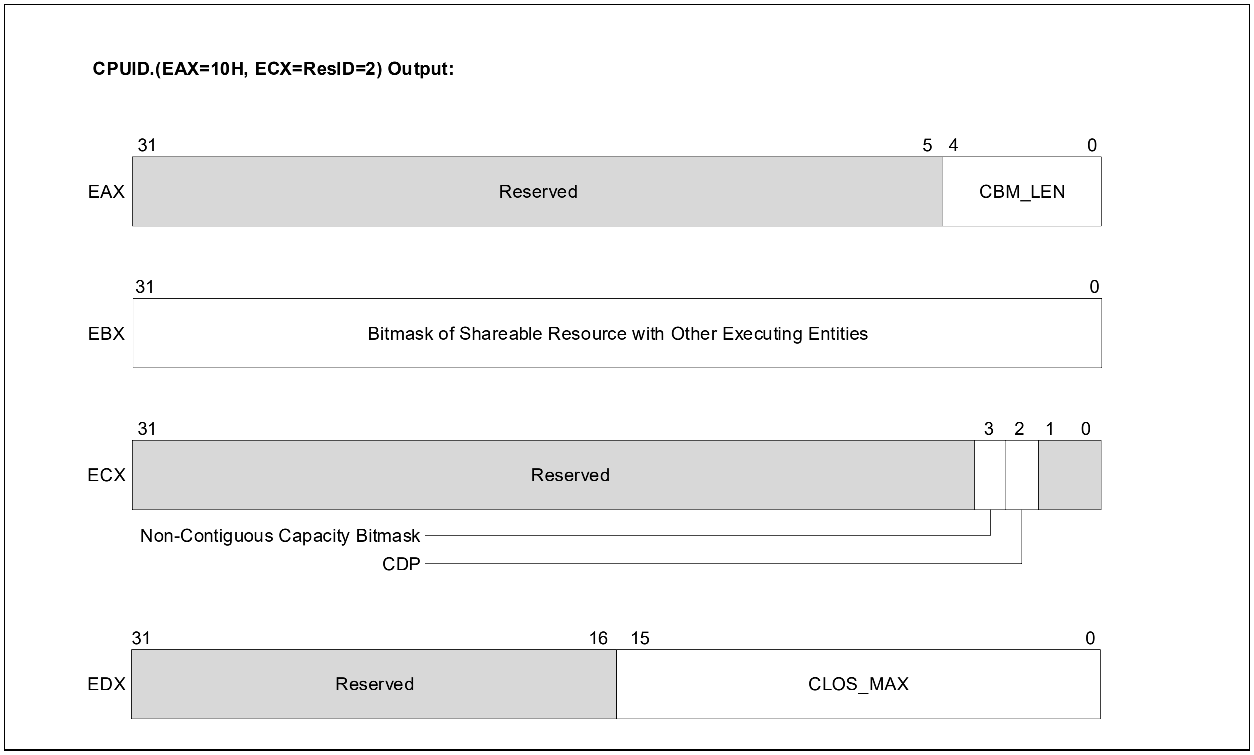

CPUID.(EAX=10H, ECX=ResID=2) Output:
31
5
4
0
EAX
Reserved
CBM_LEN
31
0
EBX
Bitmask of Shareable Resource with Other Executing Entities
31
3
2
1
0
ECX
Reserved
Non-Contiguous Capacity Bitmask
CDP
31
16
15
0
EDX
Reserved
CLOS_MAX

Figure 18-33. L2 Cache Allocation Technology

- CAT capability for L2 is enumerated by CPUID.(EAX=10H, ECX=2H), see Figure 18-33. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=2) are:
  - CPUID.(EAX=10H, ECX=ResID=2):EAX[4:0] reports the length of the capacity bitmask. Add one to the return value to get the result, e.g., a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.
  - CPUID.(EAX=10H, ECX=2):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L2 allocation may be used by other entities in the platform. Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.
  - CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2]: If 1, indicates L2 Code and Data Prioritization Technology is supported (see Section 17.19.6). Bits 1:0 and 31:4 of ECX are reserved.
  - CPUID.(EAX=10H, ECX=2):ECX[bit 3]: If 1, indicates non-contiguous capacity bitmask is supported. The bits which are set in the various IA32_L2_MASK_n registers do not have to be contiguous. Bits 1:0 and 31:4 of ECX are reserved.

— CPUID.(EAX=10H, ECX=2):EDX[15:0] reports the maximum CLOS supported for the resource (CLOS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported CLOS). Bits 31:16 are reserved.

A note on migration of Classes of Service (CLOS): Software should minimize migrations of CLOS across logical processors (across threads or cores), as a reduction in the performance of the Cache Allocation Technology feature may result if CLOS are migrated frequently. This is aligned with the industry-standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and CLOS migration across processor logical threads and processor cores.

18.19.4.3 Cache Allocation Technology: Cache Mask Configuration

After determining the length of the capacity bitmasks (CBM) and number of CLOS supported using CPUID (see Section 18.19.4.2), each CLOS needs to be programmed with a CBM to dictate its available cache via a write to the corresponding IA32_resourceType_MASK_n register, where 'n' corresponds to a number within the supported range of CLOS, i.e., the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive, and 'resourceType' corresponds to a specific resource as enumerated by the set bits of CPUID.(EAX=10H, ECX=0):EBX[31:1], for instance, ‘L2’ or ‘L3’ cache.

A hierarchy of MSRs is reserved for Cache Allocation Technology registers of the form IA32_resourceType_MASK_n:

- From 0C90H through 0D8FH (inclusive), providing support for multiple sub-ranges to support varying resource types. The first supported resource type is 'L3', corresponding to the L3 cache in a platform. The MSRs range from 0C90H through 0D0FH (inclusive), enables support for up to 128 L3 CAT Classes of Service.

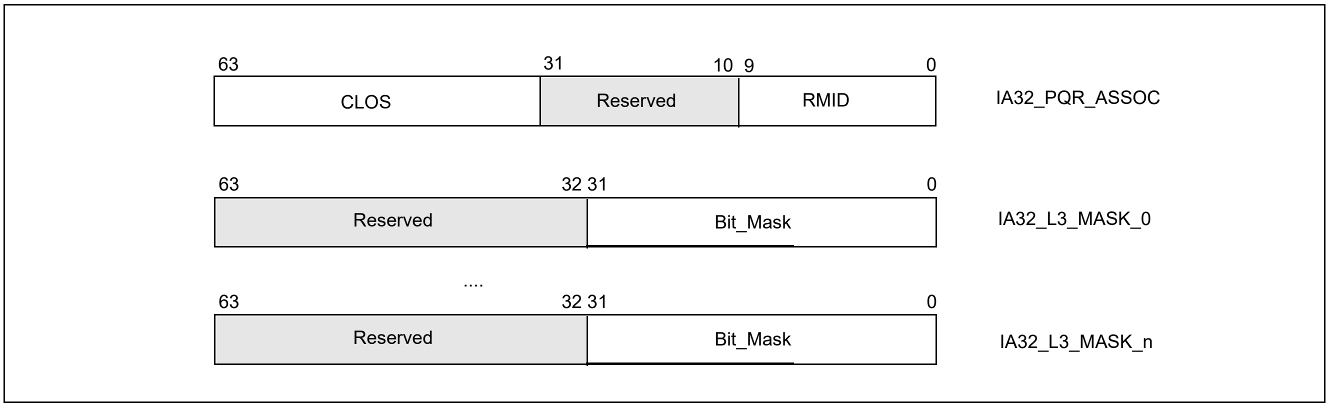


Figure 18-34. IA32_PQR_ASSOC, IA32_L3_MASK_n MSRs

- Within the same CAT range hierarchy, another set of registers is defined for resourceType 'L2', corresponding to the L2 cache in a platform, and MSRs IA32_L2_MASK_n are defined for n=[0,63] at addresses 0D10H through 0D4FH (inclusive).

Figure 18-34 and Figure 18-35 provide an overview of the relevant registers.

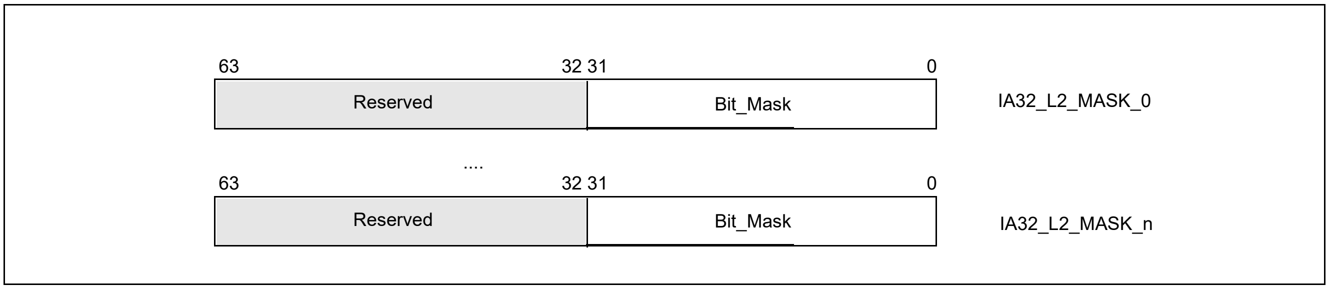


Figure 18-35. IA32_L2_MASK_n MSRs

All CAT configuration registers can be accessed using the standard RDMSR / WRMSR instructions.

Note that once L3 or L2 CAT masks are configured, threads can be grouped into Classes of Service (CLOS) using the IA32_PQR_ASSOC MSR as described in Section 18.19.4.4, "Class of Service to Cache Mask Association: Common Across Allocation Features."

18.19.4.4 Class of Service to Cache Mask Association: Common Across Allocation Features

After configuring the available classes of service with the preferred set of capacity bitmasks, the OS/VMM can set the IA32_PQR_ASSOC.CLOS of a logical processor to the class of service with the desired CBM when a thread context switch occurs. This allows the OS/VMM to indicate which class of service an executing thread/VM belongs within. Each logical processor contains an instance of the IA32_PQR_ASSOC register at MSR location 0C8FH, and Figure 18-34 shows the bit field layout for this register. Bits[63:32] contain the CLOS field for each logical processor.

Note that placing the RMID field within the same PQR register enables both RMID and CLOS to be swapped at context swap time for simultaneous use of monitoring and allocation features with a single register write for efficiency.

When CDP is enabled, Specifying a CLOS value in IA32_PQR_ASSOC.CLOS greater than MAX_CLOS_CDP = (CPUID.(EAX=10H, ECX=1):EDX[15:0] >> 1) will cause undefined performance impact to code and data fetches. In all cases, code and data masks for L2 and L3 CDP should be programmed with at least one bit set.

Note that if the IA32_PQR_ASSOC.CLOS is never written then the CAT capability defaults to using CLOS 0, which in turn is set to the default mask in IA32_L3_MASK_0 - which is all "1"s (on reset). This essentially disables the enforcement feature by default or for legacy operating systems and software.

See Section 18.19.7, "Introduction to Memory Bandwidth Allocation," for important CLOS programming considerations including maximum values when using CAT and CDP.

18.19.5 Code and Data Prioritization (CDP): Enumerating and Enabling L3 CDP Technology

L3 CDP is an extension of L3 CAT. The presence of the L3 CDP feature is enumerated via CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] (see Figure 18-32). Most of the CPUID.(EAX=10H, ECX=1) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=1):EDX.CLOS_MAX_CAT specifies the maximum CLOS applicable to CAT-only operation. For CDP operations, CLOS_MAX_CDP is equal to (CPUID.(EAX=10H, ECX=1):EDX.CLOS_MAX_CAT >>1).

If CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] =1, the processor supports CDP and provides a new MSR IA32_L3_QOS_CFG at address 0C81H. The layout of IA32_L3_QOS_CFG is shown in Figure 18-36. The bit field definition of IA32_L3_QOS_CFG are:

- Bit 0: L3 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into IA32_PQR_ASSOC.CLOS is CLOS_MAX_CDP.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a #GP(0).

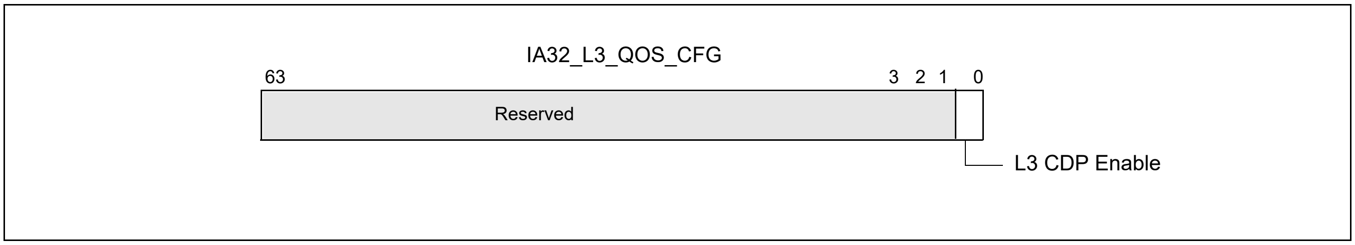


Figure 18-36. Layout of IA32_L3_QOS_CFG

IA32_L3_QOS_CFG default values are all 0s at RESET, the mask MSRs are all 1s. Hence, all logical processors are initialized in CLOS0 allocated with the entire L3 with CDP disabled, until software programs CAT and CDP. The scope of the IA32_L3_QOS_CFG MSR is defined to be the same scope as the L3 cache (e.g., typically per processor socket). Refer to Section 18.19.7 for software considerations while enabling or disabling L3 CDP.

18.19.5.1 Mapping Between L3 CDP Masks and CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per CLOS. The re-mapping is shown in Table 18-19.

Table 18-19. Re-indexing of CLOS Numbers and Mapping to CAT/CDP Mask MSRs

Mask MSR	CAT-only Operation	CDP Operation
IA32_L3_QOS_Mask_0	CLOS0	CLOS0.Data
IA32_L3_QOS_Mask_1	CLOS1	CLOS0.Code
IA32_L3_QOS_Mask_2	CLOS2	CLOS1.Data
IA32_L3_QOS_Mask_3	CLOS3	CLOS1.Code
IA32_L3_QOS_Mask_4	CLOS4	CLOS2.Data
IA32_L3_QOS_Mask_5	CLOS5	CLOS2.Code
....	....	....
IA32_L3_QOS_Mask_’2n’	CLOS’2n’	CLOS’n’.Data
IA32_L3_QOS_Mask_’2n+1’	CLOS’2n+1’	CLOS’n’.Code

One can derive the MSR address for the data mask or code mask for a given CLOS number ‘n’ by:

- data_mask_address (n) = base + (n <<1), where base is the address of IA32_L3_QOS_MASK_0.
- code_mask_address (n) = base + (n <<1) +1.

When CDP is enabled, each CLOS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over code placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 18-29).

Mask MSR field definitions remain the same. Capacity masks must be formed of contiguous set bits, unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 0x00FF, 0x00F0, 0x0001, 0x0003, and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate #GP(0).

18.19.6 Code and Data Prioritization (CDP): Enumerating and Enabling L2 CDP Technology

L2 CDP is an extension of the L2 CAT feature. The presence of the L2 CDP feature is enumerated via CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] (see Figure 17-33). Most of the CPUID.(EAX=10H, ECX=2) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=2):EDX.CLOS_MAX_CAT specifies the maximum CLOS applicable to CAT-only operation. For CDP operations, CLOS_MAX_CDP is equal to (CPUID.(EAX=10H, ECX=2):EDX.CLOS_MAX_CAT >>1).

If CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] =1, the processor supports L2 CDP and provides a new MSR IA32_L2_QOS_CFG at address 0C82H. The layout of IA32_L2_QOS_CFG is shown in Figure 18-37. The bit field definition of IA32_L2_QOS_CFG are:

- Bit 0: L2 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into IA32_PQR_ASSOC.CLOS is CLOS_MAX_CDP.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a #GP(0).

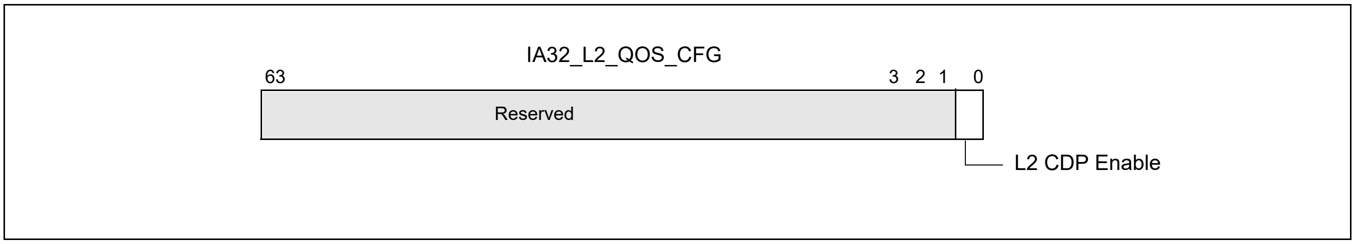


Figure 18-37. Layout of IA32_L2_QOS_CFG

IA32_L2_QOS_CFG default values are all 0s at RESET, and the mask MSRs are all 1s. Hence all logical processors are initialized in CLOS0 allocated with the entire L2 available and with CDP disabled, until software programs CAT and CDP. The IA32_L2_QOS_CFG MSR is defined at the same scope as the L2 cache, typically at the module level for Intel Atom processors for instance. In processors with multiple modules present it is recommended to program the IA32_L2_QOS_CFG MSR consistently across all modules for simplicity.

18.19.6.1 Mapping Between L2 CDP Masks and L2 CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per CLOS. This remapping is the same as the remapping shown in Table 18-19 for L3 CDP, but for the L2 MSR block (IA32_L2_QOS_MASK_n) instead of the L3 MSR block (IA32_L3_QOS_MASK_n). The same code / data mask mapping algorithm applies to remapping the MSR block between code and data masks.

As with L3 CDP, when L2 CDP is enabled, each CLOS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over code placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 18-29).

Mask MSR field definitions for L2 CDP remain the same as for L2 CAT. Capacity masks must be formed of contiguous set bits, unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 0x00FF, 0x00F0, 0x0001, 0x0003, and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate #GP(0).

18.19.6.2 Common L2 and L3 CDP Programming Considerations

Before enabling or disabling L2 or L3 CDP, software should write all 1's to all of the corresponding CAT/CDP masks to ensure proper behavior (e.g., the IA32_L3_QOS_Mask_n set of MSRs for the L3 CAT feature). When enabling CDP, software should also ensure that only CLOS number which are valid in CDP operation is used, otherwise undefined behavior may result. For instance in a case with 16 CAT CLOS, since CLOS are reduced by half when CDP is enabled, software should ensure that only CLOS 0-7 are in use before enabling CDP (along with writing 1's to all mask bits before enabling or disabling CDP).

Software should also account for the fact that mask interpretations change when CDP is enabled or disabled, meaning for instance that a CAT mask for a given CLOS may become a code mask for a different Class of Service when CDP is enabled. In order to simplify this behavior and prevent unintended remapping software should consider resetting all threads to CLOS[0] before enabling or disabling CDP.

18.19.6.3 Cache Allocation Technology Dynamic Configuration

All Intel Resource Director Technology (Intel RDT) interfaces including the IA32_PQR_ASSOC MSR, CAT/CDP masks, MBA delay values, and CQM/MBM registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,

- Accessing a QOS mask register outside the supported CLOS (the max CLOS number is specified in CPUID.(EAX=10H, ECX=ResID):EDX[15:0]), or
- Writing a CLOS greater than the supported maximum (specified as the maximum value of CPUID.(EAX=10H, ECX=ResID):EDX[15:0] for all valid ResID values) is written to the IA32_PQR_ASSOC.CLOS field.

When CDP is enabled, specifying a CLOS value in IA32_PQR_ASSOC.CLOS outside of the lower half of the CLOS space will cause undefined performance impact to code and data fetches due to MSR space re-indexing into code/data masks when CDP is enabled.

When reading the IA32_PQR_ASSOC register the currently programmed CLOS on the core will be returned.

When reading an IA32_*resourceType*_MASK_*n* register the current capacity bit mask for CLOS 'n' will be returned.

As noted previously, software should minimize migrations of CLOS across logical processors (across threads or cores), as a reduction in the accuracy of the Cache Allocation feature may result if CLOS are migrated frequently. This is aligned with the industry standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and CLOS migration across processor logical threads and processor cores.

18.19.6.4 Cache Allocation Technology Operation With Power Saving Features

Note that the Cache Allocation Technology feature cannot be used to enforce cache coherency, and that some advanced power management features such as C-states which may shrink or power off various caches within the system may interfere with CAT hints - in such cases the CAT bitmasks are ignored and the other features take precedence. If the highest possible level of CAT differentiation or determinism is required, disable any power-saving features which shrink the caches or power off caches. The details of the power management interfaces are typically implementation-specific, but can be found at Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

If software requires differentiation between threads but not absolute determinism then in many cases it is possible to leave power-saving cache shrink features enabled, which can provide substantial power savings and increase battery life in mobile platforms. In such cases when the caches are powered off (e.g., package C-states) the entire cache of a portion thereof may be powered off. Upon resuming an active state any new incoming data to the cache will be filled subject to the cache capacity bitmasks. Any data in the cache prior to the cache shrink or power off may have been flushed to memory during the process of entering the idle state, however, and is not guaranteed to remain in the cache. If differentiation between threads is the goal of system software then this model allows substantial power savings while continuing to deliver performance differentiation. If system software needs optimal determinism then power saving modes which flush portions of the caches and power them off should be disabled.

NOTE

IA32_PQR_ASSOC is saved and restored across C6 entry/exit. Similarly, the mask register contents are saved across package C-state entry/exit and are not lost.

18.19.6.5 Cache Allocation Technology Operation with Other Operating Modes

The states in IA32_PQR_ASSOC and mask registers are unmodified across an SMI delivery. Thus, the execution of SMM handler code can interact with the Cache Allocation Technology resource and manifest some degree of non-determinism to the non-SMM software stack. An SMM handler may also perform certain system-level or power management practices that affect CAT operation.

It is possible for an SMM handler to minimize the impact on data determinism in the cache by reserving a CLOS with a dedicated partition in the cache. Such an SMM handler can switch to the dedicated CLOS immediately upon entering SMM, and switching back to the previously running CLOS upon exit.

18.19.6.6 Associating Threads with CAT/CDP Classes of Service

Threads are associated with Classes of Service (CLOS) via the per-logical-processor IA32_PQR_ASSOC MSR. The same CLOS concept applies to both CAT and CDP (for instance, CLOS[5] means the same thing whether CAT or CDP is in use, and the CLOS has associated resource usage constraint attributes including cache capacity masks). The mapping of CLOS to mask MSRs does change when CDP is enabled, according to the following guidelines:

- In CAT-only Mode - one set of bitmasks in one mask MSR control both code and data.
  - Each CLOS number map 1:1 with a capacity mask on the applicable resource (e.g., L3 cache).
- When CDP is enabled:
  - Two mask sets exist for each CLOS number, one for code, one for data.
  - Masks for code/data are interleaved in the MSR address space (see Table 18-19).

18.19.7 Introduction to Memory Bandwidth Allocation

The Memory Bandwidth Allocation (MBA) feature provides indirect and approximate control over memory bandwidth available per-core. It was introduced in the Intel Xeon Scalable Processor Family. This feature provides a method to control applications that may be over-utilizing bandwidth relative to their priority in environments such as the data-center.

The MBA feature uses existing constructs from the Intel RDT feature set, including Classes of Service (CLOS). A given CLOS used for L3 CAT, for instance, means the same thing as a CLOS used for MBA. Infrastructure, such as the MSR used to associate a thread with a CLOS (the IA32_PQR_ASSOC_MSR) and some elements of the CPUID enumeration (such as CPUID leaf 10H), are shared. Certain generations include advanced hardware controllers for efficiency. For more information, refer to the “Intel® Resource Director Technology Architecture Specification.”

The following sections describe CPU interfaces to Memory Bandwidth Allocation, such as CPUID enumeration and configuration interfaces (MSRs).

18.19.7.1 Memory Bandwidth Allocation Enumeration

Similar to other Intel RDT features, enumeration of the presence and details of the MBA feature is provided via a sub-leaf of the CPUID instruction.

Key components of the enumeration are as follows.

- Support for the MBA feature on the processor, and if MBA is supported, the following details:
  - Number of supported Classes of Service (CLOS) for the processor.
  - The maximum MBA delay value supported (which also implicitly provides a definition of the granularity).
  - An indication of whether the delay values which can be programmed are linearly spaced or not.

The presence of any of the Intel RDT features which enable control over shared platform resources is enumerated by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software may then use CPUID leaf 10H to enumerate additional details on the specific controls provided.

Through CPUID leaf 10H software may determine whether MBA is supported on the platform. Specifically, as shown in Figure 18-31, bit 3 of the EBX register indicates whether MBA is supported on the processor, and the bit position (3) constitutes a Resource ID (ResID) which allows enumeration of MBA details. For instance, if bit 3 is supported this implies the presence of CPUID.10H.[ResID=3] as shown in Figure 18-38 which provides the following details.

- CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] reports the maximum MBA throttling value supported, minus one. For instance, a value of 89 indicates that a maximum throttling value of 90 is supported. Additionally, in cases where a linear interface (see below) is supported then one hundred minus the maximum throttling value indicates the granularity, 10% in this example.
- CPUID.(EAX=10H, ECX=ResID=3):EBX is reserved.
- CPUID.(EAX=10H, ECX=ResID=3):ECX[2] reports whether the response of the delay values is linear (see text).
- CPUID.(EAX=10H, ECX=ResID=3):EDX[15:0] reports the number of Classes of Service (CLOS) supported for the feature (minus one). For instance, a reported value of 15 implies a maximum of 16 supported MBA CLOS.

The number of CLOS supported for the MBA feature may or may not align with other resources such as L3 CAT. In cases where the Intel RDT features support different numbers of CLOS the lowest numerical CLOS support the common set of features, while higher CLOS may support a subset. For instance, if L3 CAT supports 8 CLOS while MBA supports 4 CLOS, all 8 CLOS would have L3 CAT masks available for cache control, but the upper 4 CLOS would not offer MBA support. In this case the upper 4 CLOS would not be subject to any throttling control. Software can manage supported resources / CLOS in order to either have consistent capabilities across CLOS by using the common subset or enable more flexibility by selectively applying resource control where needed based on careful CLOS and thread mapping. In all cases, CLOS[0] supports all Intel RDT resource control features present on the platform.

Discussion on the interpretation and usage of the MBA delay values is provided in Section 18.19.7.2 on MBA configuration.

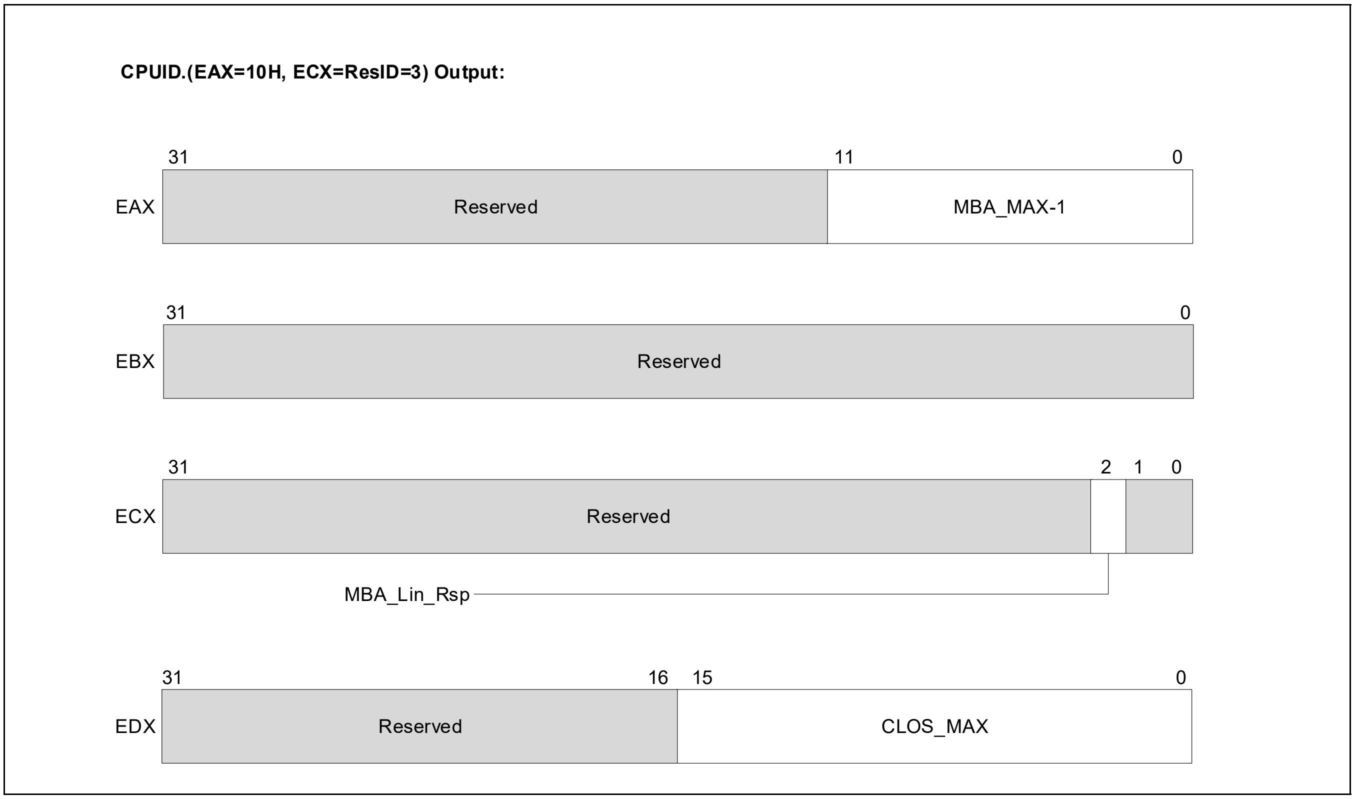


Figure 18-38. CPUID.(EAX=10H, ECX=3H) MBA Feature Details Identification

18.19.7.2 Memory Bandwidth Allocation Configuration

The configuration of MBA takes consists of two processes once enumeration is complete.

- Association of threads to Classes of Service (CLOS) - accomplished in a common fashion across Intel RDT features as described in Section 18.19.7.1 via the IA32_PQR_ASSOC MSR. As with features such as L3 CAT, software may update the CLOS field of the PQR MSR at context swap time in order to maintain the proper association of software threads to Classes of Service on the hardware. While logical processors may each be associated with independent CLOS, see Section 18.19.7.3 for important usage model considerations (initial versions of the MBA feature select the maximum delay value across threads).
- Configuration of the per-CLOS delay values, accomplished via the IA32_L2_QoS_Ext_BW_Thrtl_n MSR set shown in Table 18-20.

The MBA delay values which may be programmed range from zero (implying zero delay, and full bandwidth available) to the maximum (MBA_MAX) specified in CPUID as discussed in Section 18.19.7.1. The throttling values are approximate and do not sum to 100% across CLOS, rather they should be viewed as a maximum bandwidth “cap” per-CLOS.

Software may select an MBA delay value then write the value into one or more of the IA32_L2_QoS_Ext_BW_Thrtl_n MSRs to update the delay values applied for a specific CLOS. As shown in Table 18-20 the base address of the MSRs is at D50H, and the range corresponds to the maximum supported CLOS from CPUID.(EAX=10H, ECX=ResID=1):EDX[15:0] as described in Section 18.19.7.1. For instance, if 16 CLOS are supported then the valid MSR range will extend from D50H through D5F inclusive.

Table 18-20. MBA Delay Value MSRs

Delay Value MSR	Address
IA32_L2_QoS_Ext_BW_Thrtl_0	D50H
IA32_L2_QoS_Ext_BW_Thrtl_1	D51H
IA32_L2_QoS_Ext_BW_Thrtl_2	D52H
….	….
IA32_L2_QoS_Ext_BW_Thrtl_'CLOS_MAX'	D50H + CLOS_MAX from CPUID.10H.3

The definition for the MBA delay value MSRs is provided in Figure 17.39. The lower 16 bits are used for MBA delay values, and values from zero to the maximum from the CPUID MBA_MAX-1 value are supported. Values outside this range will generate #GP(0).

If linear input throttling values are indicated by CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] then values from zero through the MBA_MAX field from CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] are supported as inputs. In the linear mode the input precision is defined as 100-(MBA_MAX). For instance, if the MBA_MAX value is 90, the input precision is 10%. Values not an even multiple of the precision (e.g., 12%) will be rounded down (e.g., to 10% delay applied).

- If linear values are not supported (CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] = 0) then input delay values are powers-of-two from zero to the MBA_MAX value from CPUID. In this case any values not a power of two will be rounded down the next nearest power of two.

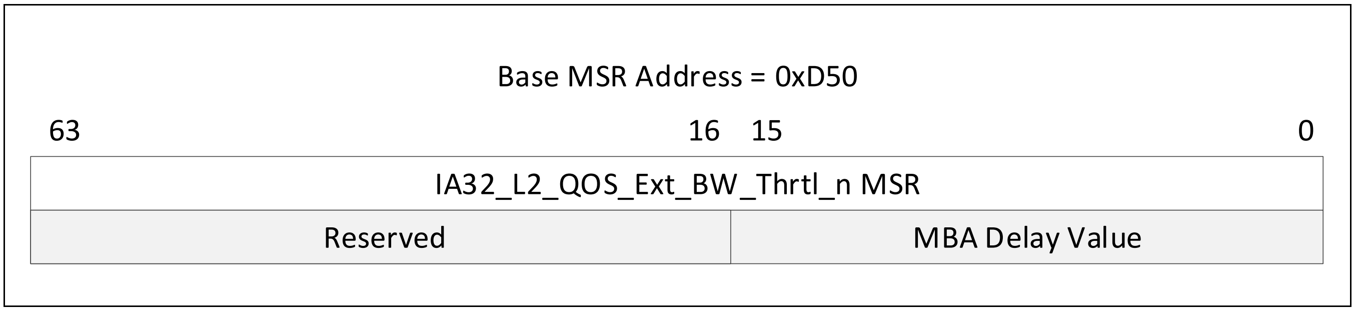


Figure 18-39. IA32_L2_QoS_Ext_BW_Thrtl_n MSR Definition

Note that the throttling values provided to software are calibrated through specific traffic patterns, however as workload characteristics may vary the response precision and linearity of the delay values will vary across products and should be treated as approximate values only.

18.19.7.3 Memory Bandwidth Allocation Usage Considerations

Different versions of Memory Bandwidth Allocation have various usage considerations and improving efficiency over time. See the “Intel® Resource Director Technology Architecture Specification” for additional details.

18.20 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FOR NON-CPU AGENTS

This section describes Intel RDT features for non-CPU agents. CPU agents are threads running on IA cores. Non-CPU agents include PCIe and CXL devices and integrated accelerators, thus broadly encompassing the set of agents that read from and write to either caches or memory, excluding IA cores. The non-CPU agent Intel RDT features enable monitoring of I/O device shared cache and memory bandwidth and cache allocation control. This provides features for I/O devices equivalent to the CPU agent Intel RDT capabilities CMT, MBM, and CAT (discussed in Section 18.18 and Section 18.19). Refer to the "Intel® Resource Director Technology Architecture Specification" regarding design goals, use cases, software architecture, ACPI enumeration, and MMIO register interfaces.

"Non-CPU agent Intel RDT" refers to capabilities that monitor and control non-CPU agents' resource utilization, including PCIe and CXL devices and integrated accelerators. Non-CPU agent Intel RDT may be called I/O RDT in some literature. In this document, the term "non-CPU agent Intel RDT" is used.

18.20.1 Non-CPU Agent Intel® RDT Features Enumeration Details

CPU agent Intel RDT features use the CPUID instruction to enumerate supported features and the level of support. Architectural Model-Specific Registers (MSRs) are interfaces to the monitoring and allocation features, as described in Sections 18.18 and 18.19.

Non-CPU agent Intel RDT builds on CPU agent Intel RDT by extending CPUID to indicate the presence and integration of non-CPU agent Intel RDT and by providing rich enumeration information in vendor-specific extensions to the Advanced Configuration and Power Interface (ACPI), in particular in the I/O RDT (IRDT) table. The ACPI extensions detailed in the "Intel® Resource Director Technology Architecture Specification" provide mechanisms to comprehend the structure of devices attached behind I/O blocks to particular links and what forms of tagging are supported on a per-link basis.

It is recommended that software parse CPUID and ACPI to obtain a detailed understanding of platform support and capabilities before attempting to use non-CPU agent Intel RDT.

18.20.1.1 CPUID-Based Enumeration for Non-CPU Agent Intel® RDT Feature

CPUID-based enumeration provides a method by which all architectural Intel RDT features may be enumerated.

For CPU agent Intel RDT, monitoring details are enumerated in a CPUID sub-leaf denoted as CPUID.(EAX=0FH, ECX=ResID), where ResID corresponds to a resource ID bit index from the CPUID.(EAX=0FH, ECX=0) sub-leaf. Similarly, Intel RDT allocation features are described in CPUID.(EAX=10H, ECX=ResID). (Note that the ResID bit positions are not guaranteed to be symmetric or have the same encodings.)

No CPUID leaves or sub-leaves are created for non-CPU agent Intel RDT. Rather, non-CPU agent Intel RDT extends the existing Intel RDT CPUID sub-leaves with a bit per resource type, indicating whether non-CPU agent Intel RDT monitoring or control is present. CPUID.(EAX=0FH, ECX=ResID=1):EAX[bits 9, 10] represents the presence of CMT and MBM features for non-CPU agents. CPUID.(EAX=10H, ECX=ResID=1):ECX[bit 1] represents the presence of the CAT feature for non-CPU agents.

Specifically, for non-CPU Agent Intel RDT Monitoring (see Figure 18-21):

- Bits are added in the CPU Agent Intel RDT CMT/MBM leaf: CPUID.(EAX=0FH, ECX=ResID=1):EAX[bits 9, 10].
    - EAX[bit 9]: If set, indicates the presence of non-CPU Agent Cache Occupancy Monitoring (the equivalent of CPU Agent Intel RDT's CMT feature).
    - EAX[bit 10]: If set, indicates the presence of non-CPU Agent memory L3 external BW monitoring (the equivalent of CPU Agent Intel RDT's MBM feature).

For non-CPU Agent Intel RDT Allocation (see Figure 18-32):

- New bit in L3 CAT leaf: CPUID.(EAX=10H, ECX=ResID=1):ECX[bit 1].
    - ECX[bit 1]: If set, indicates the presence of non-CPU Agent Cache Allocation Technology (the equivalent of CPU Agent Intel RDT's L3 CAT feature).
- As before, ECX[bit 2] indicates that L3 CDP is supported if set.

Note that no equivalent bits are defined in CPUID.(EAX=10H, ECX=ResID=2) as there is no ability for devices to fill into core L2 caches.

If any of these non-CPU agent Intel RDT enumeration bits are set, indicating that a monitoring feature or allocation feature is present, it also indicates the presence of the IA32_L3_IO_RDT_CFG architectural MSR. This MSR may be used to enable the non-CPU agent Intel RDT features. See Section 18.20.2 for MSR details.

The presence of Intel RDT is a prerequisite for using the equivalent non-CPU agent Intel RDT feature. If a particular CPU agent Intel RDT feature is absent, any attempt to use non-CPU agent Intel RDT equivalents will result in general protection faults in the MSR interface. Attempts to enable unsupported features in the I/O complex will result in writes to the corresponding MMIO enable or configuration interfaces being ignored.

Software may use the existing CPUID leaves to gather the maximum number of RMID and CLOS tags for each resource level (e.g., L3 cache), and non-CPU agent Intel RDT is also subject to these limits.

Some platforms may support a mix of features, for instance, supporting L3 CAT architectural controls and the non-CPU agent Intel RDT equivalent, but no CMT/MBM monitoring or non-CPU agent monitoring equivalent, and these capabilities should be enumerated on a per-platform basis.

18.20.1.2 ACPI Enumeration

When support for non-CPU agent Intel RDT features is detected using CPUID, ACPI may be consulted for further details on the level of feature support, device structures behind various I/O ports, and the specific MMIO interfaces used to control a given device.

Non-CPU agent Intel RDT enumeration is via the “IRDT” ACPI table. For more information, refer to the “Intel® Resource Director Technology Architecture Specification.”

18.20.2 Non-CPU Agent Intel® RDT Feature Enable MSR

Before configuring non-CPU agent Intel RDT through MMIO, the feature should be enabled using the non-CPU agent Intel RDT Feature Enable MSR, IA32_L3_IO_RDT_CFG (MSR address 0C83H). As described in Section 18.20.1.1, the presence of one or more CPUID bits indicating support for one or more non-CPU agent Intel RDT features also indicates the presence of this MSR. This MSR may be used to enable the non-CPU agent Intel RDT features.

Two bits are defined in this MSR. Bit 0, when set, enables non-CPU agent RDT resource allocation features. Bit 1, when set, enables non-CPU agent Intel RDT monitoring features.

The L3 Non-CPU agent Intel RDT Monitoring Enable bit is supported if CPUID indicates that one or more non-CPU agent Intel RDT resource monitoring features are present.

The L3 Non-CPU agent Intel RDT Allocation Enable bit is supported if CPUID indicates that one or more non-CPU agent Intel RDT resource allocation features are present.

The default value is 0x0, so both classes of features are disabled by default. All bits not defined are reserved. Writing a non-zero value to any reserved bit will generate a General Protection Fault (#GP(0)).

This MSR is scoped at the L3 cache level and is cleared on system reset. It is expected that the software will configure this MSR consistently across all L3 caches that may be present on that package.

The definition of the IA32_L3_IO_RDT_CFG MSR is shown in Figure 18-40.

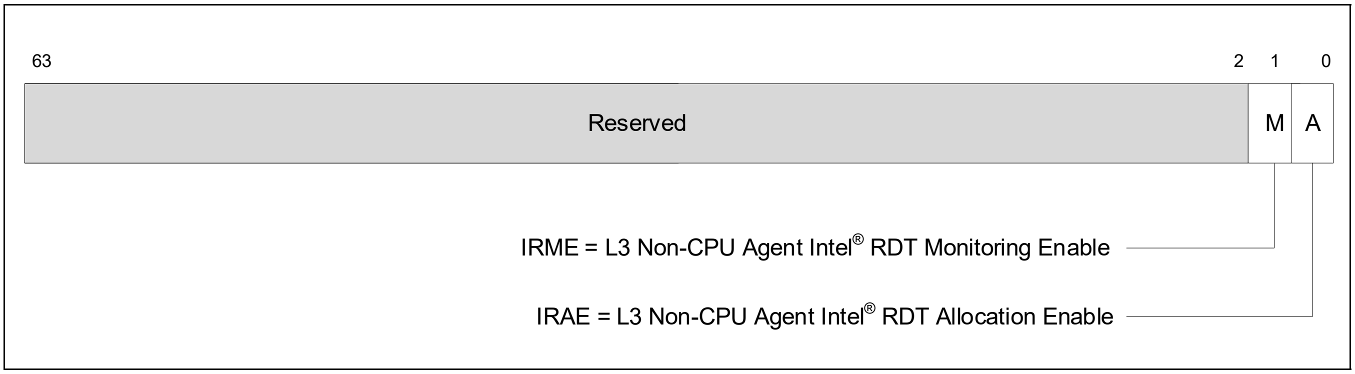

63
2
1
0
Reserved
M
A
IRME = L3 Non-CPU Agent Intel® RDT Monitoring Enable
IRAE = L3 Non-CPU Agent Intel® RDT Allocation Enable

Figure 18-40. Layout of the IA32_L3_IO_RDT_CFG MSR for Enabling Non-CPU Agent Intel® RDT

CHAPTER 19
LAST BRANCH RECORDS

NOTE

This chapter defines a last-branch recording (LBR) facility that is architectural and part of the Intel 64 architecture. This facility is an enhancement of but distinct from earlier LBR facilities that were not architectural. Those earlier facilities are documented in Chapter 18.

Support of the architectural LBR feature in a logical processor is reported in CPUID.(EAX=07H, ECX=0H):EDX[19]=1. When the architectural LBR feature is supported, capability details like the number of LBR records that are available is indicated in CPUID.1CH:EAX[7:0]. The number of LBR records available varies across processor generations, so software should only access the available LBR records indicated by CPUID.1CH:EAX[7:0].

Last Branch Records (LBRs) enable recording of software path history by logging taken branches and other control flow transfers within processor registers. Each LBR record or entry is comprised of three MSRs:

- IA32_LBR_x_FROM_IP — Holds the source IP of the operation.
- IA32_LBR_x_TO_IP — Holds the destination IP of the operation.
- IA32_LBR_x_INFO — Holds metadata for the operation, including mispredict, TSX, and elapsed cycle time information.

LBR records are stored in age order. The most recent LBR entry is stored in IA32_LBR_0_*, the next youngest in IA32_LBR_1_*, and so on. When an operation to be recorded completes (retires) with LBRs enabled (IA32_LBR_CTL.LBREn=1), older LBR entries are shifted in the LBR array by one entry, then a record of the new operation is written into entry 0. See Section 19.1.1 for the list of recorded operations.

The number of LBR entries available for recording operations is dictated by the value in IA32_LBR_DEPTH.DEPTH. By default, the DEPTH value matches the maximum number of LBRs supported by the processor, but software may opt to use fewer in order to achieve reduced context switch latency.

In addition to the LBRs, there is a single Last Event Record (LER). It records the last taken branch preceding the last exception, hardware interrupt, or software interrupt. Like LBRs, the LER is comprised of three MSRs (IA32_LER_FROM_IP, IA32_LER_TO_IP, IA32_LER_INFO), and is subject to the same dependencies on enabling and filtering.

Which operations are recorded in LBRs depends upon a series of factors:

- Branch Type Filtering — Software must opt in to the types of branches to be logged; see Section 19.1.2.3.
- Current Privilege Level (CPL) — LBRs can be filtered based on CPL; see Section 19.1.2.5.
- LBR Freeze — LBR and LER recording can be suspended by setting IA32_PERF_GLOBAL_STATUS.LBR_FRZ to 1. See Section 18.4.7 for details on LBR_FRZ.

On some implementations, recording LBRs may require constraining the number of operations that can complete in a cycle. As a result, on these implementations, enabling LBRs may have some performance overhead.

19.1 BEHAVIOR

19.1.1 Logged Operations

LBRs can log most control flow transfer operations.

The source IP recorded for a branch instruction is the IP of that instruction. For events that take place between instructions, the source IP recorded is the IP of the next sequential instruction.

The destination IP recorded is always the target of the branch or event, the next instruction that will execute.

The full list of operations and the respective IPs recorded is shown in Table 19-1.

Table 19-1. LBR IP Values for Various Operations

Operation	FROM_IP	TO_IP
Taken Branch[1], Exception, INT3, INTn, INTO, TSX Abort	Current IP	Target IP
Interrupt	Next IP	Target IP
INIT (BSP)	Next IP	Reset Vector
INIT (AP) + SIPI	Next IP	SIPI Vector
EENTER/ERESUME + EEXIT/AEX	Current IP	Target or Trampoline IP
RSM[2]	Target IP	Target IP
#DB, #SMI, VM exit, VM entry	None	None

NOTES:

1. Direct CALLs with displacement zero, for which the target is typically the next sequential IP, are not treated as taken branches by LBRs.
2. RSM is only recorded in LBRs when IA32_DEBUGCTL.FREEZE_WHILE_SMM is set to 0.

19.1.2 Configuration

19.1.2.1 Enabling and Disabling

LBRs are enabled by setting IA32_LBR_CTL.LBREn to 1.

Some operations, such as entry to a secure mode like SMM or Intel SGX, can cause LBRs to be temporarily disabled. Other operations, such as debug exceptions or some SMX operations, disable LBRs and require software to re-enable them. Details on these interactions can be found in Section 19.1.4.

19.1.2.2 LBR Depth

The number of LBRs used by the processor can be constrained by modifying the IA32_LBR_DEPTH.DEPTH value. DEPTH defaults to the maximum number of LBRs supported by the processor. Allowed DEPTH values can be found in CPUID.1CH:EAX[7:0].

Reducing the LBR depth can result in improved performance, by reducing the number of LBRs that need to be read and/or context switched.

On a software write to IA32_LBR_DEPTH, all LBR entries are reset to 0. LERs are not impacted.

A RDMSR or WRMSR to any IA32_LBR_x_* MSRs, such that x ≥ DEPTH, will generate a #GP exception. Note that the XSAVES and XRSTORS instructions access only the LBRs associated with entries 0 to DEPTH-1.

By clearing the LBR entries on writes to IA32_LBR_DEPTH, and forbidding any software writes to LBRs ≥ DEPTH, it is thereby guaranteed that any LBR entries equal to or above DEPTH will have value 0.

19.1.2.3 Branch Type Enabling and Filtering

Software must opt in to the types of branches that are desired to be recorded. These elections are made in IA32_LBR_CTL; see Section 19.2. Branch type options are listed in Table 19-2; only those enabled will be recorded.

Table 19-2. Branch Type Filtering Details

Branch Type	Operations Recorded
COND	Jcc, J*CXZ, and LOOP*
NEAR_IND_JMP	JMP r/m*
NEAR_REL_JMP	JMP rel*
NEAR_IND_CALL	CALL r/m*
NEAR_REL_CALL	CALL rel* (excluding CALLs to the next sequential IP)
NEAR_RET	RET (0C3H)
OTHER_BRANCH	JMP/CALL ptr*, JMP/CALL m*, RET (0C8H), SYS*, interrupts, exceptions (other than debug exceptions), IRET, INT3, INTn, INTO, TSX Abort, EENTER, ERESUME, EEXIT, AEX, INIT, SIPI, RSM

These encodings match those in IA32_LBR_x_INFO.BR_TYPE.

Control flow transfers that are not recorded include #DB, VM exit, VM entry, and #SMI.

19.1.2.4 Call-Stack Mode

The LBR array is, by default, treated as a ring buffer that captures control flow transitions. However, the finite depth of the LBR array can be limiting when profiling certain high-level languages (e.g., C++), where a transition of the execution flow is accompanied by a large number of leaf function calls. These calls to leaf functions, and their returns, are likely to displace the main execution context from the LBRs.

When call-stack mode is enabled, the LBR array can capture unfiltered call data normally, but as return instructions are executed the last captured branch (call) record is flushed from the LBRs in a last-in first-out (LIFO) manner. Thus, branch information pertaining to completed leaf functions will not be retained, while preserving the call stack information of the main line execution path.

Call-stack mode is enabled by setting IA32_LBR_CTL.CALL_STACK to 1. When enabled, near RET instructions receive special treatment. Rather than adding a new record in LBR_0, a near RET will instead “pop” the CALL entry at LBR_0 by shifting entries LBR_1..LBR_[DEPTH-1] up to LBR_0..LBR_[DEPTH-2], and clearing LBR_[DEPTH-1] to 0. Thus, LBR processing software can consume only valid call-stack entries by reading until finding an entry that is all zeros.

Call-stack mode should be used with branch type enabling configured to capture only CALLs (NEAR_REL_CALL and NEAR_IND_CALL) and RETs (NEAR_RET). When configured in this manner, the LBR array emulates a call stack, where CALLs are “pushed” and RETs “pop” them off the stack. If other branch types (JCC, NEAR_*_JMP, or OTHER_BRANCH) are enabled for recording with call-stack mode, LBR behavior may be undefined.

It is recommended that call-stack mode be used along with CPL filtering, by setting at most one of the OS and USR bits in the IA32_LBR_CTL MSR. Call-stack mode does not emulate the stack switch that can occur on CPL transitions, and hence monitoring all CPLs may result in a corrupted LBR call stack.

Call-Stack Mode and LBR Freeze

When IA32_DEBUGCTL.FREEZE_LBRS_ON_PMI=1, IA32_PERF_GLOBAL_STATUS.LBR_FRZ will be set to 1 when a PMI is pended. That will cause LBRs and LERs to cease recording branches until LBR_FRZ is cleared. Because there may be some “skid”, or instructions retiring, in between the PMI being pended and the PMI being taken, it is possible that some branches may be missing from the LBRs. In the case of call-stack mode, if a CALL or RET is missed, that can lead to confusing results where CALL entries fail to get “popped” off the stack, and RETs “pop” the wrong CALLs.

An alternative is to utilize CPL filtering to limit LBR recording to less privileged modes only (CPL>3) instead of using the FREEZE_LBRS_ON_PMI=1 feature. This will record branches in the “skid”, but avoid recording any branches in the privilege level 0 handler.

19.1.2.5 CPL Filtering

Software must opt in to which CPL(s) will have branches recorded. If IA32_LBR_CTL.OS=1, then branches in CPL=0 can be recorded. If IA32_LBR_CTL.USR=1, then branches in CPL>0 can be recorded. For operations which change the CPL, the operation is recorded in LBRs only if the CPL at the end of the operation is enabled for LBR recording. In cases where the CPL transitions from a value that is filtered out to a value that is enabled for LBR recording, the FROM_IP address for the recorded CPL transition branch or event will be 0FFFFFFFFFFFFFFFFH.

19.1.3 Record Data

19.1.3.1 IP Fields

The source and destination IP values in IA32_LBR_x_[FROM|TO]_IP and IA32_LER_x_[FROM|TO]_IP may hold effective IPs or linear IPs (LIPs), depending on the processor generation. The effective IP is the offset from the CS base address, while LIP includes the CS base address. Which IP type is used is indicated in CPUID.1CH:EAX[bit 31].

The value read from this field will always be canonical. Note that this includes the case where a canonical violation (#GP) results from executing sequential code that runs precisely to the end of the lower canonical address space (where IP[63:MAXLINADDR-1] is 0, but IP[MAXLINADDR-2:0] is all ones). In this case, the FROM_IP will hold the lowest canonical address in the upper canonical space, such that IP[63:MAXLINADDR-1] is all ones, and IP[MAXLINADDR-2:0] is 0.

In some cases, due to CPL filtering, the FROM_IP of the recorded operation may be filtered out. In this case 0FFFFFFFFFFFFFFFFH will be recorded. See Section 19.1.2.5 for details.

Writes of these fields will be forced canonical, such that the processor ignores the value written to the upper bits (IP[63:MAXLINADDR-1]).

19.1.3.2 Branch Types

The IA32_LBR_x_INFO.BR_TYPE and IA32_LER_INFO.BR_TYPE fields encode the branch types as shown in Table 19-3.

Table 19-3. IA32_LBR_x_INFO and IA32_LER_INFO Branch Type Encodings

Encoding	Branch Type
0000B	COND
0001B	NEAR_IND_JMP
0010B	NEAR_REL_JMP
0011B	NEAR_IND_CALL
0100B	NEAR_REL_CALL
0101B	NEAR_RET
011xB	Reserved
1xxxB	OTHER_BRANCH

For a list of branch operations that fall into the categories above, see Table 19-2. In future generations, BR_TYPE bits 2:0 may be used to distinguish between differing types of OTHER_BRANCH.

19.1.3.3 Cycle Time

Each time an operation is recorded in an LBR, the value of the LBR cycle timer is recorded in IA32_LBR_x_INFO.CYC_CNT. The LBR cycle timer is a saturating counter that counts at the processor clock rate. Each time an operation is recorded in an LBR, the counter is reset but continues counting.

There is an LBR cycle counter valid bit, IA32_LBR_x_INFO.CYC_CNT_VALID. When set, the CYC_CNT field holds a valid value, the number of elapsed cycles since the last operation recorded in an LBR (up to 0FFFFH).

Some implementations may opt to reduce the granularity of the CYC_CNT field for larger values. The implication of this is that the least significant bits may be forced to 1 in cases where the count has reached some minimum threshold. It is guaranteed that this reduced granularity will never result in an inaccuracy of more than 10%.

19.1.3.4 Mispredict Information

IA32_LBR_x_INFO.MISPRED provides an indication of whether the recorded branch was predicted incorrectly by the processor. The bit is set if either the taken/not-taken direction of a conditional branch was mispredicted, or if the target of an indirect branch was mispredicted.

19.1.3.5 Intel® TSX Information

IA32_LBR_x_INFO.IN_TSX indicates whether the operation recorded retired during a TSX transaction. IA32_LBR_x_INFO.TSX_ABORT indicates that the operation is a TSX Abort.

19.1.4 Interaction with Other Processor Features

19.1.4.1 SMM

IA32_LBR_CTL.LBREn is saved and cleared on #SMI, and restored on RSM. As a result of disabling LBRs, the #SMI is not recorded. RSM is recorded only if IA32_DEBUGCTL.FREEZE_WHILE_SMM is set to 0, and the FROM_IP will be set to the same value as the TO_IP.

19.1.4.2 SMM Transfer Monitor (STM)

LBREn is not cleared on #SMI when it causes SMM VM exit. Instead, the STM should use the VMCS controls described in Section 19.1.4.3 to disable LBRs while in SMM, and to restore them on VM entries that exit SMM.

On VMCALL to configure STM, IA32_LBR_CTL is cleared.

19.1.4.3 VMX

By default, LBR operation persists across VMX transitions. However, VMCS fields have been added to enable constraining LBR usage to within non-root operation only. See details in Table 19-4.

Table 19-4. LBR VMCS Fields

Name	Type	Bit Position	Behavior
Guest IA32_LBR_CTL	Guest State Field	NA	The guest value of IA32_LBR_CTL is written to this field on all VM exits.
Load Guest IA32_LBR_CTL	Entry Control	21	When set, VM entry will write the value from the "Guest IA32_LBR_CTL" guest state field to IA32_LBR_CTL.
Clear IA32_LBR_CTL	Exit Control	26	When set, VM exit will clear IA32_LBR_CTL after the value has been saved to the "Guest IA32_LBR_CTL" guest state field.

To enable "guest-only" LBR use, a VMM should set both the "Load Guest IA32_LBR_CTL" entry control and the "Clear IA32_LBR_CTL" exit control. For "system-wide" LBR use, where LBRs remain enabled across host and guest(s), a VMM should keep both new VMCS controls clear.

VM entry checks that, if the "Load Guest IA32_LBR_CTL" entry control is 1, bits reserved in the IA32_LBR_CTL MSR must be 0 in the field for that register.

For additional information relating to VMX transitions, see Chapter 25, Chapter 27, and Chapter 28 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C.

19.1.4.4 Intel® SGX

On entry to an enclave, via EENTER or ERESUME, logging of LBR entries is suspended. On enclave exit, via EEXIT or AEX, logging resumes. The cycle counter will continue to run during enclave execution.

An exception to the above is made for opt-in debug enclaves. For such enclaves, LBR logging is not impacted.

19.1.4.5 Debug Exceptions

When a branch happens because of a #DB exception, IA32_LBR_CTL.LBREn is cleared. As a result, the operation is not recorded.

19.1.4.6 SMX

On GETSEC leaves SENTER or ENTERACCS, IA32_LBR_CTL is cleared. As a result, the operation is not recorded.

19.1.4.7 MWAIT

On an MWAIT that requests a C-state deeper than C1, IA32_LBR_x_* MSRs may be cleared to 0. IA32_LBR_CTL, IA32_LBR_DEPTH, and IA32_LER_* MSRs will be preserved.

For an MWAIT that enters a C-state equal to or less deep than C1, and all C-states that enter as a result of Hardware Duty Cycling (HDC), all LBR MSRs are preserved.

19.1.4.8 Processor Event-Based Sampling (PEBS)

PEBS records can be configured to include LBRs, by setting PEBS_DATA_CFG.LBREn[3]=1. The number of LBRs to include in the record is also configurable, via PEBS_DATA_CFG.NUM_LBRS[28:24]. For details on PEBS, see Section 20.9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

If NUM_LBRS is set to a value greater than LBR_DEPTH, then only LBR_DEPTH entries will be written into the PEBS record. Further, the Record Size field will be decreased to match the actual size of the record to be written, and the Record Format field will replace the value of NUM_LBRS with the value of LBR_DEPTH. These adjustments ensure that software is able to properly interpret the PEBS record.

19.2 MSRS

The MSRs that represent the LBR entries (IA32_LBR_x_[TO|FROM|INFO]) and the LER entry (IA32_LER_[TO|FROM|INFO]) do not fault on writes. Any address field written will force sign-extension based on the maximum linear address width supported by the processor, and any non-zero value written to undefined bits may be ignored such that subsequent reads return 0.

On a warm reset, all LBR MSRs, including IA32_LBR_DEPTH, have their values preserved. However, IA32_LBR_CTL.LBREn is cleared to 0, disabling LBRs. If a warm reset is triggered while the processor is in the C6 idle state, also known as warm init, all LBR MSRs will be reset to their initial values.

See Table 2-2 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for details on LBR MSRs.

19.3 FAST LBR READ ACCESS

XSAVES provides a faster means than RDMSR for software to read all LBRs. When using XSAVES for reading LBRs rather than for context switch, software should take care to ensure that XSAVES does not write LBR state to an area of memory that has been or will be used by XRSTORS. This could corrupt INIT tracking.

19.4 OTHER IMPACTS

19.4.1 Branch Trace Store on Intel Atom® Processors

Branch Trace Store (BTS) on Intel Atom processors that support the architectural form of the LBR feature has dependencies on the LBR configuration. BTS will store out the LBR_0 (TOS) record each time a taken branch or event retires. If any filtering of LBRs is employed, or if LBRs are disabled, some duplicate entries may be stored by BTS. Like LBRs and LERs, BTS is suspended when IA32_PERF_GLOBAL_STATUS.LBR_FRZ is set to 1.

BTS will change to cease issuing branch records for direct near CALLs with displacement zero to align with LBR behavior.

19.4.2 IA32_DEBUGCTL

On processors that do not support model-specific LBRs, IA32_DEBUGCTL[bit 0] has no meaning. It can be written to 0 or 1, but reads will always return 0.

19.4.3 IA32_PERF_CAPABILITIES

On processors that do not support model-specific LBRs, IA32_PERF_CAPABILITIES.LBR_FMT will have the value 03FH.

CHAPTER 20
PERFORMANCE MONITORING

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

NOTE

Performance monitoring events can be found here: https://perfmon-events.intel.com/.

Additionally, performance monitoring event files for Intel processors are hosted by the Intel Open Source Technology Center. These files can be downloaded here: https://download.01.org/perfmon/.

20.1 PERFORMANCE MONITORING OVERVIEW

Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

The performance monitoring mechanisms and performance events defined for the Pentium, P6 family, and Intel processors based on Intel NetBurst microarchitecture are not architectural. They are all model specific (not compatible among processor families). Intel Core Solo and Intel Core Duo processors support a set of architectural performance events and a set of non-architectural performance events. Newer Intel processor generations support enhanced architectural performance events and non-architectural performance events.

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 20.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)." Non-architectural events for a given microarchitecture cannot be enumerated using CPUID; and they can be found at: https://perfmon-events.intel.com/.

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and Interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 20.2.

See also:

- Section 20.2, "Architectural Performance Monitoring."
- Section 20.3, "Performance Monitoring (Intel® Core™ Processors and Intel® Xeon® Processors)."
  - Section 20.3.1, "Performance Monitoring for Processors Based on Nehalem Microarchitecture."
  - Section 20.3.2, "Performance Monitoring for Processors Based on Westmere Microarchitecture."
  - Section 20.3.3, "Intel® Xeon® Processor E7 Family Performance Monitoring Facility."
  - Section 20.3.4, "Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture."
  - Section 20.3.5, "3rd Generation Intel® Core™ Processor Performance Monitoring Facility."

  - Section 20.3.6, "4th Generation Intel® Core™ Processor Performance Monitoring Facility."
  - Section 20.3.7, "5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility."
  - Section 20.3.8, "6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility."
  - Section 20.3.9, "10th Generation Intel® Core™ Processor Performance Monitoring Facility."
  - Section 20.3.10, "12th and 13th Generation Intel® Core™ Processors, and 4th Generation Intel® Xeon® Scalable Processor Family Performance Monitoring Facility."
- Section 20.4, "Performance monitoring (Intel® Xeon™ Phi Processors)."
  - Section 20.4.1, "Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring."
- Section 20.5, "Performance Monitoring (Intel Atom® Processors)."
  - Section 20.5.1, "Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)."
  - Section 20.5.2, "Performance Monitoring for Silvermont Microarchitecture."
  - Section 20.5.3, "Performance Monitoring for Goldmont Microarchitecture."
  - Section 20.5.4, "Performance Monitoring for Goldmont Plus Microarchitecture."
  - Section 20.5.5, "Performance Monitoring for Tremont Microarchitecture."
- Section 20.6, "Performance Monitoring (Legacy Intel Processors)."
  - Section 20.6.1, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)."
  - Section 20.6.2, "Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)."
  - Section 20.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)."
  - Section 20.6.4, "Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture."
    - Section 20.6.4.5, "Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture."
  - Section 20.6.5, "Performance Monitoring and Dual-Core Technology."
  - Section 20.6.6, "Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache."
  - Section 20.6.7, "Performance Monitoring on L3 and Caching Bus Controller Sub-Systems."
  - Section 20.6.8, "Performance Monitoring (P6 Family Processor)."
  - Section 20.6.9, "Performance Monitoring (Pentium Processors)."
- Section 20.7, "Counting Clocks."
- Section 20.8, "IA32_PERF_CAPABILITIES MSR Enumeration."
- Section 20.9, "PEBS Facility."

20.2 ARCHITECTURAL PERFORMANCE MONITORING

Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T

7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture. Intel Atom processors based on the Goldmont and Goldmont Plus microarchitectures support versionID 4. Intel Atom processors starting with processors based on the Tremont microarchitecture support versionID 5.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microarchitectures support version ID 3. Intel processors based on the Skylake through Coffee Lake microarchitectures support versionID 4. Intel processors starting with processors based on the Ice Lake microarchitecture support versionID 5.

20.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- The bit field layout of IA32_PERFEVTSELx is consistent across microarchitectures. A non-zero write of a field that is introduced after the initial implementation of architectural performance monitoring (Version 1) results in #GP if that field is not supported.
- Addresses of IA32_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32_PMC MSRs remain the same across microarchitectures.
- Each logical processor has its own set of IA32_PERFEVTSELx and IA32_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available to software in a logical processor (each IA32_PERFEVTSELx MSR is paired to the corresponding IA32_PMCx MSR).
- Number of bits supported in each IA32_PMCx.
- Number of architectural performance monitoring events supported in a logical processor.

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32_PERFEVTSELx/ IA32_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

20.2.1.1 Architectural Performance Monitoring Version 1 Facilities

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8]. Note that this may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

- IA32_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block. Note the number of IA32_PERFEVTSELx MSRs may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.
- The bit width of an IA32_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.
- Bit field layout of IA32_PERFEVTSELx MSRs is defined architecturally.

See Figure 20-1 for the bit field layout of IA32_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7) —** Selects the event logic unit used to detect microarchitectural conditions (see Table 20-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

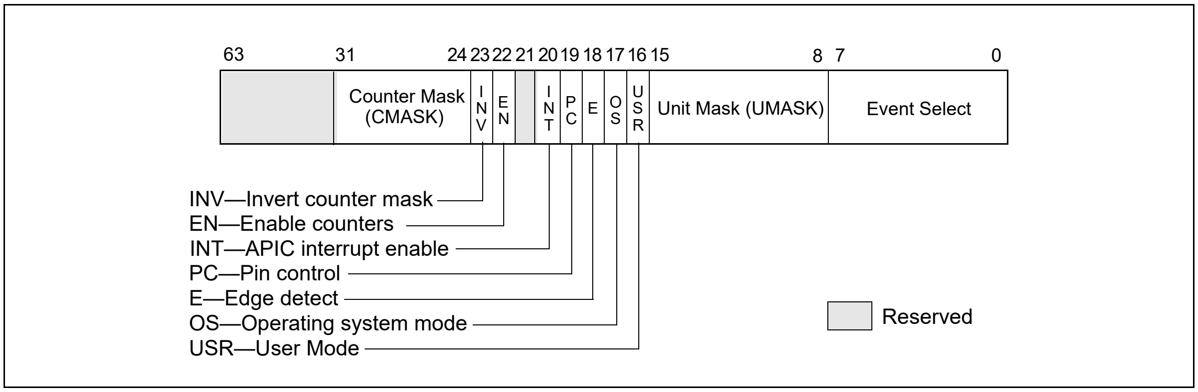


Figure 20-1. Layout of IA32_PERFEVTSELx MSRs

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

  A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 20-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX.
- **USR (user mode) flag (bit 16) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.
- **OS (operating system mode) flag (bit 17) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- **E (edge detect) flag (bit 18) —** Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

  This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
- **PC (pin control) flag (bit 19) —** Beginning with Sandy Bridge microarchitecture, this bit is reserved (not writeable). On processors based on previous microarchitectures, the logical processor toggles the PM*i* pins and

increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.

- **INT (APIC interrupt enable) flag (bit 20) —** When set, the logical processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32_PERFEVTSELx[bit 22] = 0, before writing to IA32_PMCx.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31) —** When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

  This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

20.2.1.2 Pre-defined Architectural Performance Events

Table 20-1 lists architecturally defined events.

Table 20-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

Bit Position CPUID.AH.EBX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles[1]	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H
7	Topdown Slots	01H	A4H

NOTES:

[1] Implementations prior to the 12th generation Intel® Core™ processor P-cores count at core crystal clock, TSC, or bus clock frequency.

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 20-1). The number of architectural events is reported through CPUID.0AH:EAX[31:24], while non-zero bits in CPUID.0AH:EBX indicate any architectural events that are not available.

The behavior of each architectural performance event is expected to be consistent on all processors that support that event. Minor variations between microarchitectures are noted below:

- **UnHalted Core Cycles —** Event select 3CH, Umask 00H

  This event counts core clock cycles when the clock signal on a specific core is running (not halted). The counter does not advance in the following conditions:

  — An ACPI C-state other than C0 for normal operation.

  — HLT.

  — STPCLK# pin asserted.

- — Being throttled by TM1.
- — During the frequency switching phase of a performance state transition (see Chapter 15, "Power and Thermal Management").

The performance counter for this event counts across performance state transitions using different core clock frequencies.

- **Instructions Retired —** Event select C0H, Umask 00H

  This event counts the number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. An instruction with a REP prefix counts as one instruction (not per iteration). Faults before the retirement of the last micro-op of a multi-ops instruction are not counted.

  This event does not increment under VM-exit conditions. Counters continue counting during hardware interrupts, traps, and inside interrupt handlers.

- **UnHalted Reference Cycles —** Event select 3CH, Umask 01H

  This event counts reference clock cycles at a fixed frequency while the clock signal on the core is running. The event counts at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. Current implementations use the core crystal clock, TSC or the bus clock. Because the rate may differ between implementations, software should calibrate it to a time source with known frequency.

- **Last Level Cache References —** Event select 2EH, Umask 4FH

  This event counts requests originating from the core that reference a cache line in the last level on-die cache. The event count includes speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Last Level Cache Misses —** Event select 2EH, Umask 41H

  This event counts each cache miss condition for references to the last level on-die cache. The event count may include speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Branch Instructions Retired —** Event select C4H, Umask 00H

  This event counts branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction.

- **All Branch Mispredict Retired —** Event select C5H, Umask 00H

  This event counts mispredicted branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction in the branch prediction hardware.

  Branch prediction hardware is implementation-specific across microarchitectures; value comparison to estimate performance differences is not recommended.

- **Topdown Slots —** Event select A4H, Umask 01H

  This event counts the total number of available slots for an unhalted logical processor.

  The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core, in processors that support Intel Hyper-Threading Technology.

  Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

NOTE

Programming decisions or software precisians on functionality should not be based on the event values or dependent on the existence of performance monitoring events.

20.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32_FIXED_CTR0 through IA32_-FIXED_CTR2). Each of the fixed-function PMC can count only one architectural performance event.

  Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32_FIXED_CTR_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32_PMCx) via UMASK field in (IA32_PERFEVTSELx), configuring, programming IA32_FIXED_CTR_CTRL for fixed-function PMCs do not require any UMASK.
- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - IA32_PERF_GLOBAL_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or any general-purpose PMCs via a single WRMSR.
  - IA32_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - IA32_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - IA32_DEBUGCTL.Freeze_LBR_On_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 18.4.7 for details of the legacy Freeze LBRs on PMI control.
  - IA32_DEBUGCTL.Freeze_PerfMon_On_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 18.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

NOTE

> Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32_FIXED_CTR_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 20-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

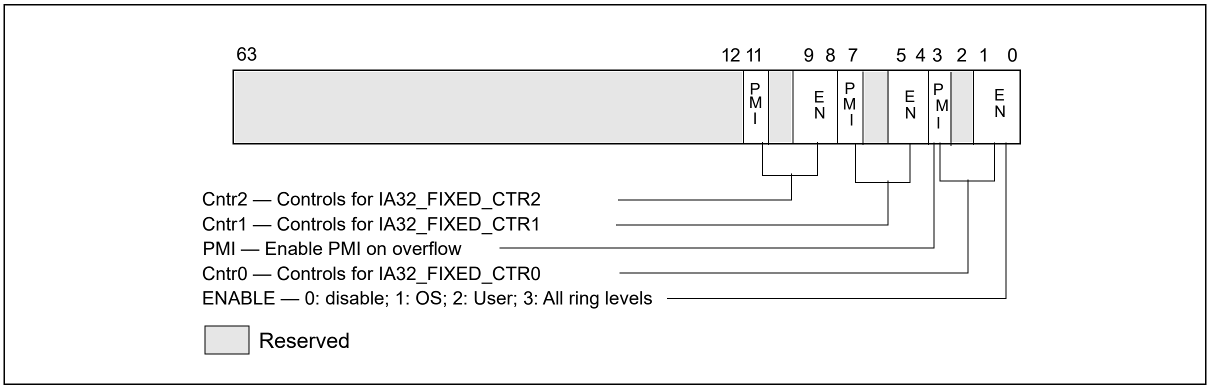

63
12 11
9 8 7
5 4 3 2 1 0
P M I
E N
P M I
E N
P M I
E N
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow
Cntr0 — Controls for IA32_FIXED_CTR0
ENABLE — 0: disable; 1: OS; 2: User; 3: All ring levels
Reserved

Figure 20-2. Layout of IA32_FIXED_CTR_CTRL MSR

- **Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- **PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 20-3 shows the layout of IA32_PERF_GLOBAL_CTRL. Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_C-TR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

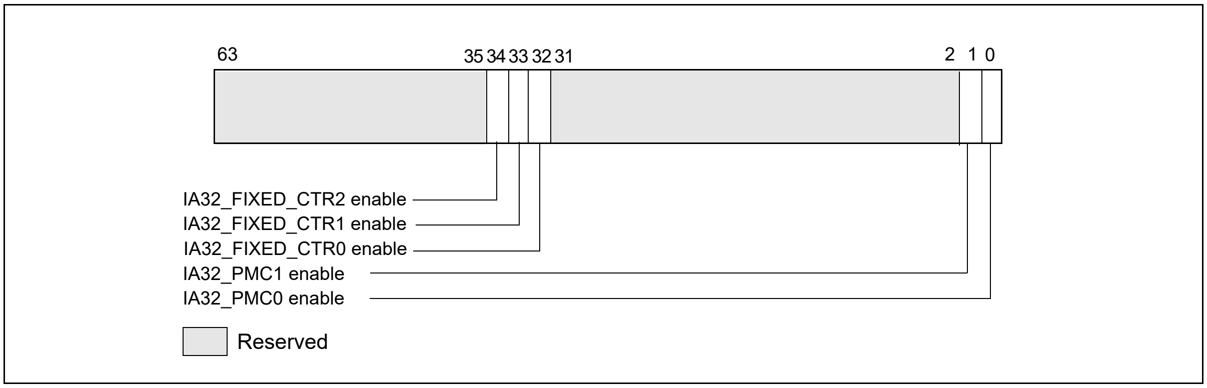


Figure 20-3. Layout of IA32_PERF_GLOBAL_CTRL MSR

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.

Table 20-2. Association of Fixed-Function Performance Counters with Architectural Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_FIXED_CTR0	309H	INST_RETIRED.ANY	This event counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The counter continues counting during hardware interrupts, traps, and in-side interrupt handlers.
IA32_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD CPU_CLK_UNHALTED.CORE	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.
IA32_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.
IA32_FIXED_CTR3	30CH	TOPDOWN.SLOTS	This event counts the number of available slots for an unhalted logical processor. The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core. Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 20-4 shows the layout of IA32_PERF_GLOBAL_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status., and sets the "OvfBuffer" bit in IA32_PERF_-GLOBAL_STATUS.

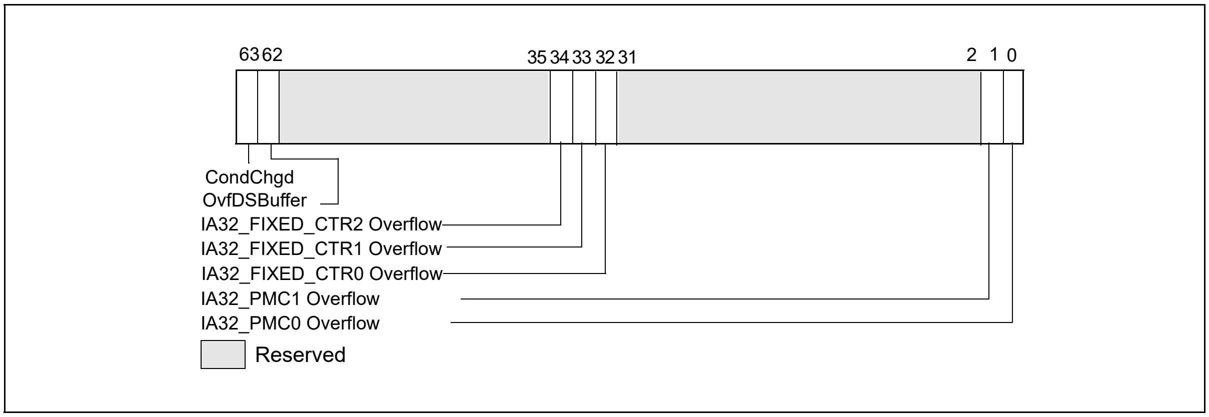


Figure 20-4. Layout of IA32_PERF_GLOBAL_STATUS MSR

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

The layout of IA32_PERF_GLOBAL_OVF_CTL is shown in Figure 20-5.

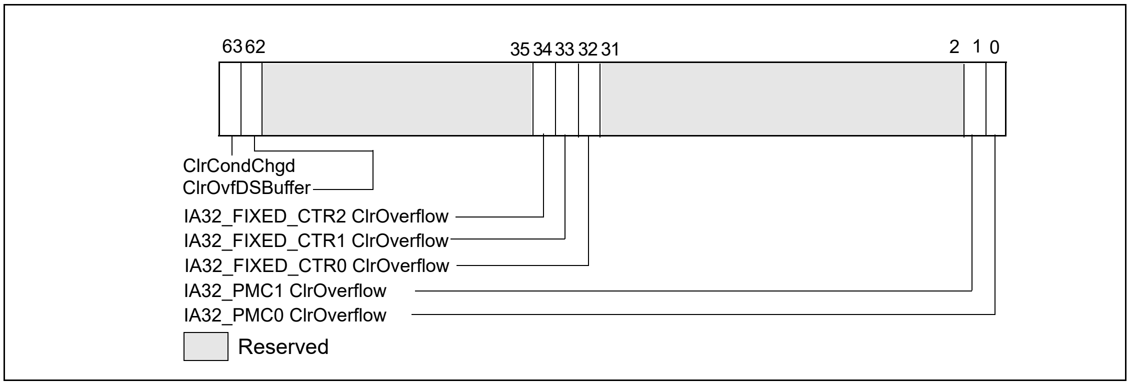


Figure 20-5. Layout of IA32_PERF_GLOBAL_OVF_CTRL MSR

20.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e., a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- AnyThread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  — Each IA32_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 20-6.

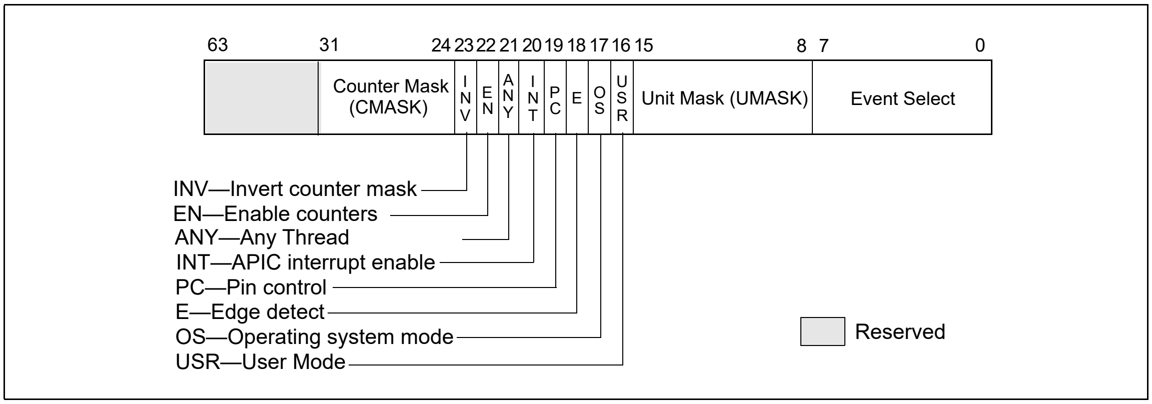


Figure 20-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

Bit 21 (AnyThread) of IA32_PERFEVTSELx is supported in architectural performance monitoring version 3 for processor core comprising of two or more logical processors. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring across all logical processors sharing a processor core. When bit 21 is 0, the counter only increments the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFE-VTSELx) occurring in the logical processor which programmed the IA32_PERFEVTSELx MSR.

— Each fixed-function performance counter IA32_FIXED_CTRx (starting at MSR address 309H) is configured by a 4-bit control block in the IA32_PERF_FIXED_CTR_CTRL MSR. The control block also allows thread-specificity configuration using an AnyThread bit for fixed-function counters 0, 1, and 2. The layout of IA32_PERF_FIXED_CTR_CTRL MSR is shown.

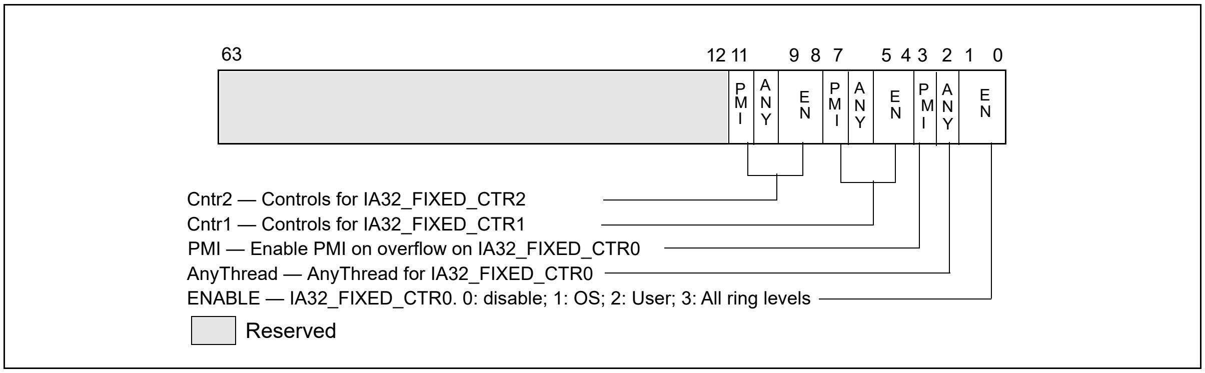

63
12
11
9
8
7
5
4
3
2
1
0
PMI
ANY
EN
PMI
ANY
EN
PMI
ANY
EN
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow on IA32_FIXED_CTR0
AnyThread — AnyThread for IA32_FIXED_CTR0
ENABLE — IA32_FIXED_CTR0. 0: disable; 1: OS; 2: User; 3: All ring levels
Reserved

Figure 20-7. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

Each control block for a fixed-function performance counter provides an **AnyThread** (bit position 2 + 4*N, N= 0, 1, etc.) bit. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the ENABLE setting of the corresponding control block of IA32_PERF_FIXED_CTR_CTRL) occurring across all logical processors sharing a processor core. When an **AnyThread** bit is 0 in IA32_PERF_-FIXED_CTR_CTRL, the corresponding fixed counter only increments the associated event conditions occurring in the logical processor which programmed the IA32_PERF_FIXED_CTR_CTRL MSR.

- The IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, IA32_PERF_GLOBAL_OVF_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 20-8 and Figure 20-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.

NOTE

The number of general-purpose performance monitoring counters (i.e., N in Figure 20-9) can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology, (e.g., N=2 for 45 nm Intel Atom processors; N =4 for processors based on the Nehalem microarchitecture; for processors based on the Sandy Bridge microarchitecture, N = 4 if Intel Hyper Threading Technology is active and N=8 if not active). In addition, the number of counters may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

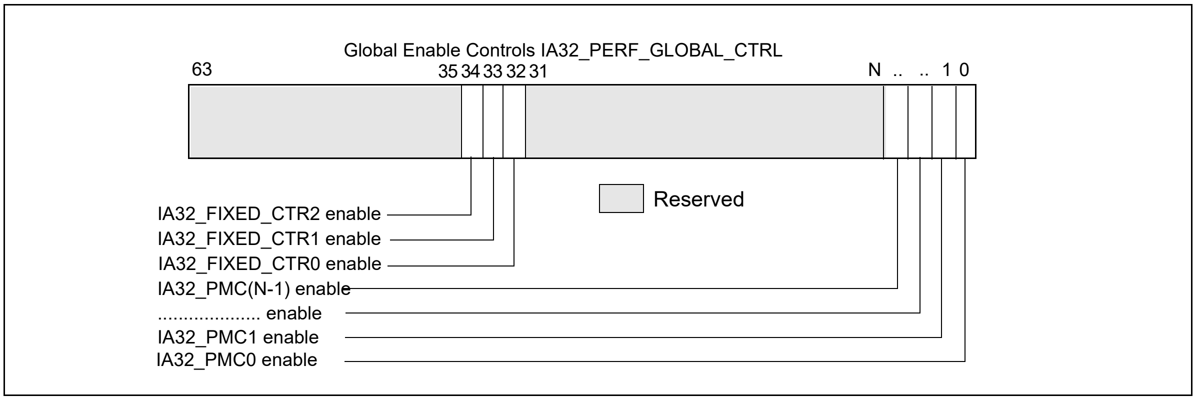


Figure 20-8. Layout of Global Performance Monitoring Control MSR

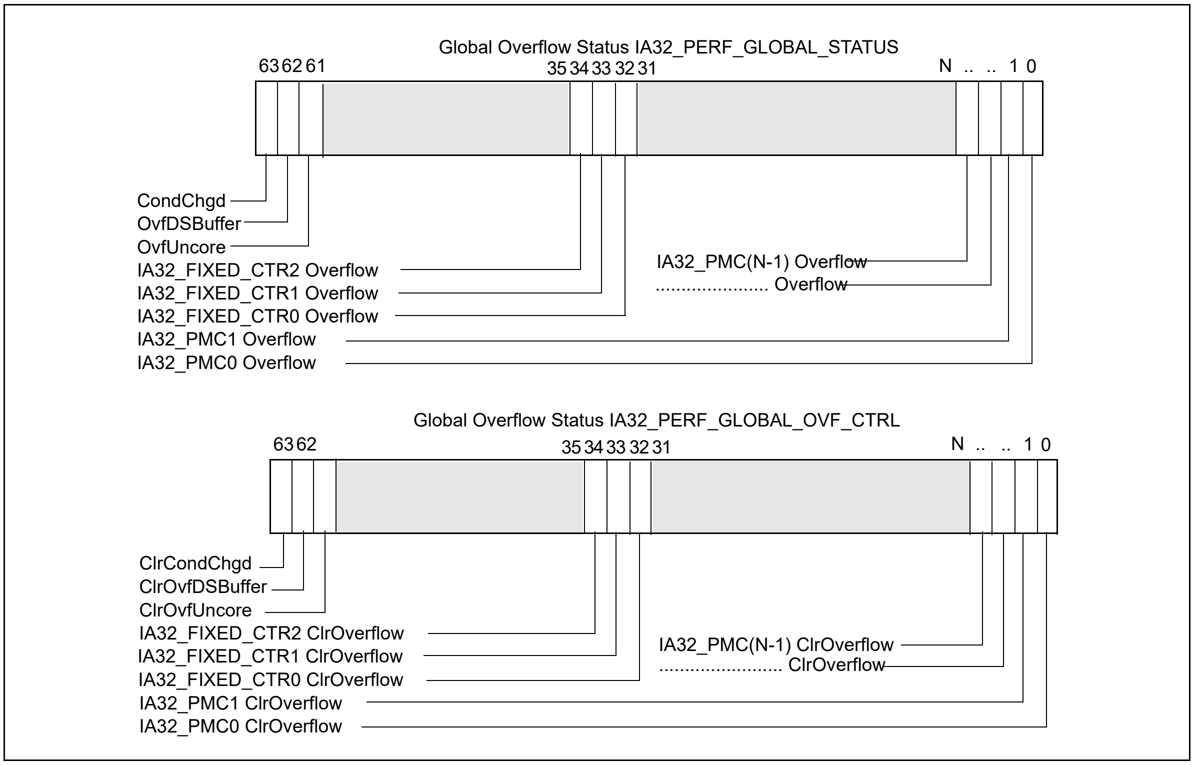


Figure 20-9. Global Performance Monitoring Overflow Status and Control MSRs

20.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL. While AnyThread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.

One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 24, “Introduction to Virtual Machine Extensions”) where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow an individual VM to employ performance monitoring facilities to profiles the performance characteristics of a workload. The use of the Anythread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- Configure the MSR bitmap to cause VM-exits for WRMSR to IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in VMX non-Root operation (see Chapter 25 for additional information),
- Clear the AnyThread bit of IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in the MSR-load lists for VM exits and VM entries (see Chapter 25, Chapter 27, and Chapter 28).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted.

20.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32_DEBUGCTL.Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI. Specifically version 4 provides the following enhancements:

- New indicators (LBR_FRZ, CTR_FRZ) in IA32_PERF_GLOBAL_STATUS, see Section 20.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32_DEBUGCTL.Freeze_LBRs_On_PMI and IA32_DEBUGCTL.Freeze_PerfMon_On_PMI: see Section 20.2.4.1. Legacy semantics of Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32_PERF_GLOBAL_STATUS and read-only performance counter enabling interface in IA32_PERF_GLOBAL_STATUS: see Section 20.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

20.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS

The IA32_PERF_GLOBAL_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32_PERF_GLOBAL_STATUS.LBR_FRZ[bit 58]: This bit is set due to the following conditions:
  - — IA32_DEBUGCTL.FREEZE_LBR_ON_PMI has been set by the profiling agent, and
  - — A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.LBR_FRZ bit also serves as a control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural perfmon version 4 or higher:

  - — (IA32_DEBUGCTL.LBR & (!IA32_PERF_GLOBAL_STATUS.LBR_FRZ)) =1
- IA32_PERF_GLOBAL_STATUS.CTR_FRZ[bit 59]: This bit is set due to the following conditions:

— IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI has been set by the profiling agent, and

— A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

Effectively, the IA32_PERF_GLOBAL_STATUS.CTR_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:

- (IA32_PERFEVTSELn.EN & IA32_PERF_GLOBAL_CTRL.PMCn & (!IA32_PERF_-GLOBAL_STATUS.CTR_FRZ)) = 1 for programmable counter ‘n’, or
- (IA32_PERF_FIXED_CRTL.ENi & IA32_PERF_GLOBAL_CTRL.FCi & (!IA32_PERF_-GLOBAL_STATUS.CTR_FRZ)) = 1 for fixed counter ‘i’

The read-only enable interface IA32_PERF_GLOBAL_STATUS.CTR_FRZ provides a more efficient flow for a PMI handler to use IA32_DEBUGCTL.Freeze_Perfmon_On_PMI to filter out data that may distort target workload analysis, see Table 18-3. It should be noted the IA32_PERF_GLOBAL_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32_PERF_-GLOBAL_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls, and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g., race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32_PERF_GLOBAL_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support it (for additional information about Intel SGX, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D):

- IA32_PERF_GLOBAL_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e., IA32_PMCx or IA32_FIXED_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32_PERF_GLOBAL_STATUS.ASCI was cleared).

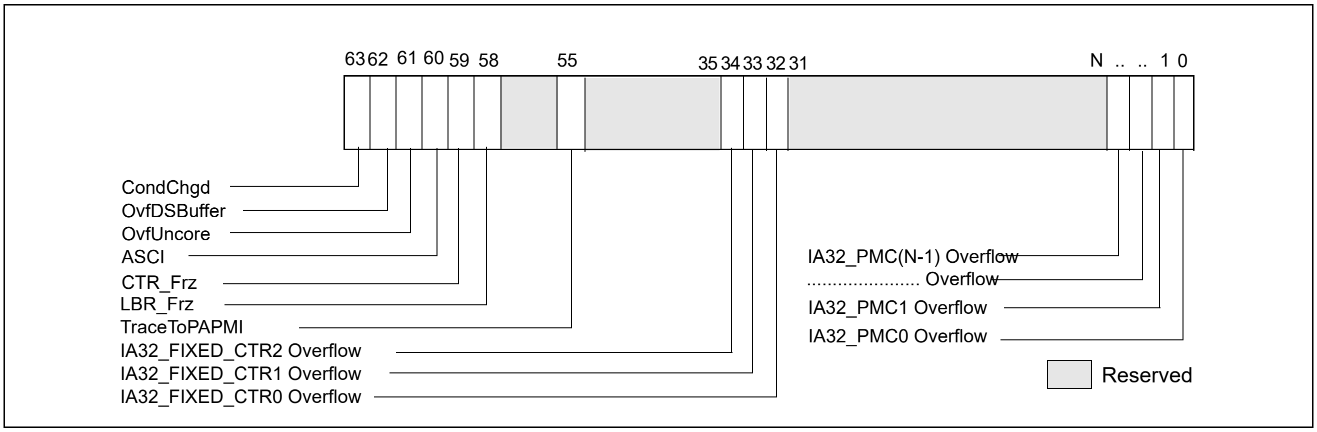


Figure 20-10. IA32_PERF_GLOBAL_STATUS MSR and Architectural Perfmon Version 4

Note, a processor’s support for IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32_PERF_GLOBAL_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.

20.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32_PERF_GLOBAL_STATUS MSR by software is done via IA32_PERF_GLOBAL_OVF_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32_PERF_GLOBAL_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32_PERF_GLOBAL_OVF_CTRL is inherited by IA32_PERF_GLOBAL_STATUS_RESET (see Figure 20-11). Further, IA32_PERF_GLOBAL_STATUS_RESET provides additional bit fields to clear the new indicators in IA32_PERF_GLOBAL_STATUS described in Section 20.2.4.1.

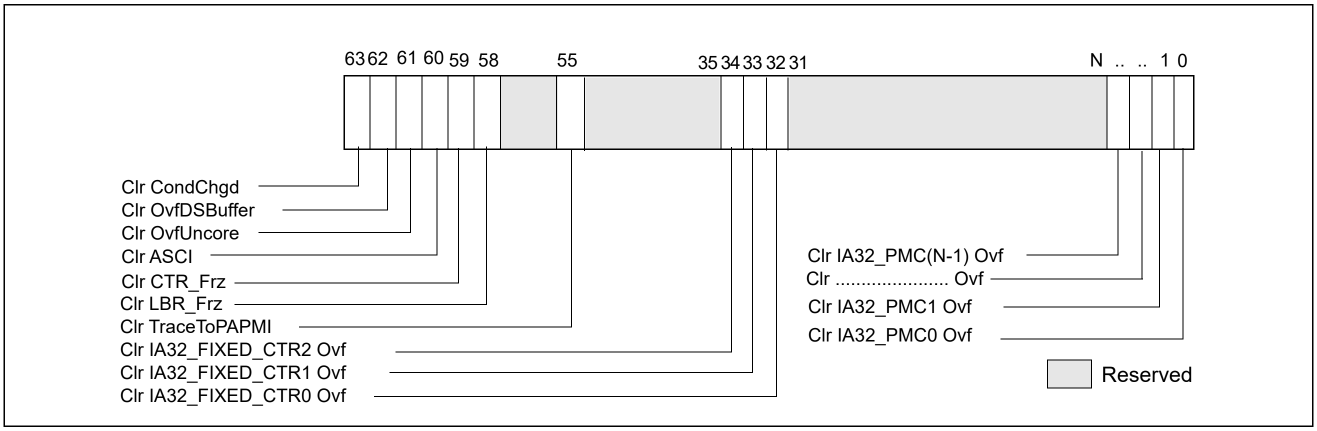


Figure 20-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural Perfmon Version 4

The IA32_PERF_GLOBAL_STATUS_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32_PERF_GLOBAL_STATUS. The IA32_PERF_GLOBAL_STATUS_SET interface can be used by a VMM to virtualize the state of IA32_PERF_GLOBAL_STATUS across VMs.

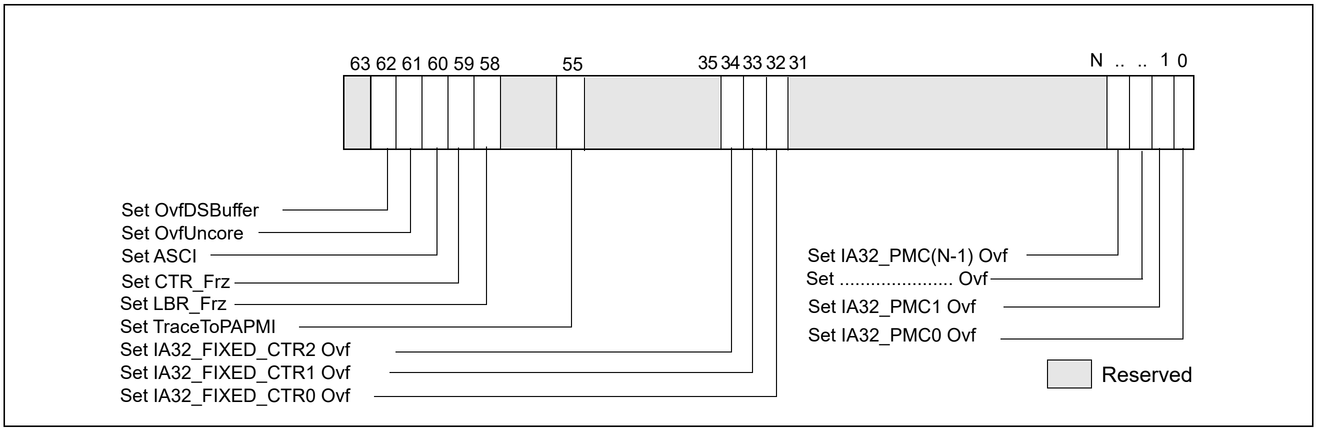


Figure 20-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural Perfmon Version 4

20.2.4.3 IA32_PERF_GLOBAL_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor's performance monitoring facilities. The IA32_MISC_ENABLE.PERFMON_AVAILABLE[bit 7] interface could not serve the need of multiple agent adequately. A white paper, “Performance Monitoring Unit Sharing Guideline”[1], proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32_PERF_GLOBAL_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32_PERF_GLOBAL_INUSE is shown in Figure 20-13.

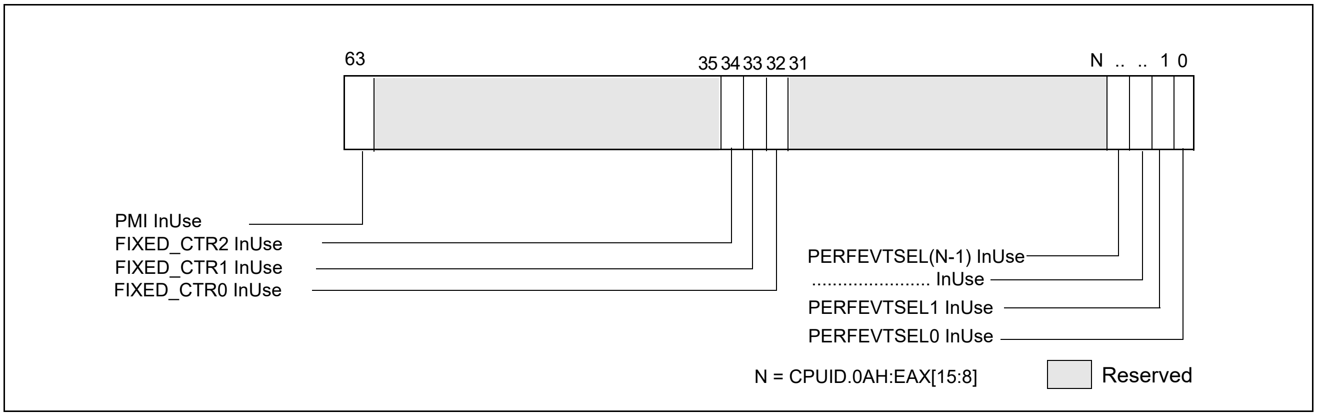


Figure 20-13. IA32_PERF_GLOBAL_INUSE MSR and Architectural Perfmon Version 4

The IA32_PERF_GLOBAL_INUSE MSR provides an “InUse” bit for each programmable performance counter and fixed counter in the processor. Additionally, it includes an indicator if the PMI mechanism has been configured by a profiling agent.

- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL0_InUse[bit 0]: This bit reflects the logical state of (IA32_PERFEVTSEL0[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL1_InUse[bit 1]: This bit reflects the logical state of (IA32_PERFEVTSEL1[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL2_InUse[bit 2]: This bit reflects the logical state of (IA32_PERFEVTSEL2[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSELn_InUse[bit n]: This bit reflects the logical state of (IA32_PERFEVTSELn[7:0] != 0), n < CPUID.0AH:EAX[15:8].
- IA32_PERF_GLOBAL_INUSE.FC0_InUse[bit 32]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[1:0] != 0).
- IA32_PERF_GLOBAL_INUSE.FC1_InUse[bit 33]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[5:4] != 0).
- IA32_PERF_GLOBAL_INUSE.FC2_InUse[bit 34]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[9:8] != 0).
- IA32_PERF_GLOBAL_INUSE.PMI_InUse[bit 63]: This bit is set if any one of the following bit is set:
  - IA32_PERFEVTSELn.INT[bit 20], n < CPUID.0AH:EAX[15:8].
  - IA32_FIXED_CTR_CTRL.ENi_PMI, i = 0, 1, 2.
  - Any IA32_PEBS_ENABLES bit which enables PEBS for a general-purpose or fixed-function performance counter.

[1] Available at http://www.intel.com/sdm

20.2.5 Architectural Performance Monitoring Version 5

Processors supporting architectural performance monitoring version 5 also support versions 1, 2, 3, and 4, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 5 provides the following enhancements:

- Deprecation of AnyThread mode, see Section 20.2.5.1.
- Individual enumeration of Fixed counters in CPUID.0AH, see Section 20.2.5.2.
- Domain separation, see Section 20.2.5.3.

20.2.5.1 AnyThread Mode Deprecation

With Architectural Performance Monitoring Version 5, a processor that supports AnyThread mode deprecation is enumerated by CPUID.0AH.EDX[15]. If set, software will not have to follow guidelines in Section 20.2.3.1.

20.2.5.2 Fixed Counter Enumeration

With Architectural Performance Monitoring Version 5, register CPUID.0AH.ECX indicates Fixed Counter enumeration. It is a bit mask which enumerates the supported Fixed Counters in a processor. If bit 'i' is set, it implies that Fixed Counter 'i' is supported. Software is recommended to use the following logic to check if a Fixed Counter is supported on a given processor:

FxCtr[i]_is_supported := ECX[i] || (EDX[4:0] > i);

20.2.5.3 Domain Separation

When the INV flag in IA32_PERFEVTSELx is used, a counter stops counting when the logical processor exits the C0 ACPI C-state.

20.2.6 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32_PMCx are writable via WRMSR instruction. However, the value written into IA32_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32_PERF_CAPABILITIES[13] to enumerate its full-width-write capability See Figure 20-65.

If IA32_PERF_CAPABILITIES.FW_WRITE[bit 13] =1, each IA32_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32_A_PMC0.

The bit width of the performance monitoring counters is specified in CPUID.0AH:EAX[23:16].

If IA32_A_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32_A_PMCi will cause IA32_PMCi to be updated by:

```
COUNTERWIDTH = CPUID.0AH:EAX[23:16] bit width of the performance monitoring counter
IA32_PMCi[COUNTERWIDTH-1:32] := EDX[COUNTERWIDTH-33:0]);
IA32_PMCi[31:0] := EAX[31:0];
EDX[63:COUNTERWIDTH] are reserved
```

20.3 PERFORMANCE MONITORING (INTEL® CORE™ PROCESSORS AND INTEL® XEON® PROCESSORS)

20.3.1 Performance Monitoring for Processors Based on Nehalem Microarchitecture

Intel Core i7 processor family[1] supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Nehalem microarchitecture, and provides four general-purpose performance counters (IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_-FIXED_CTR1, IA32_FIXED_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as "uncore".
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3.

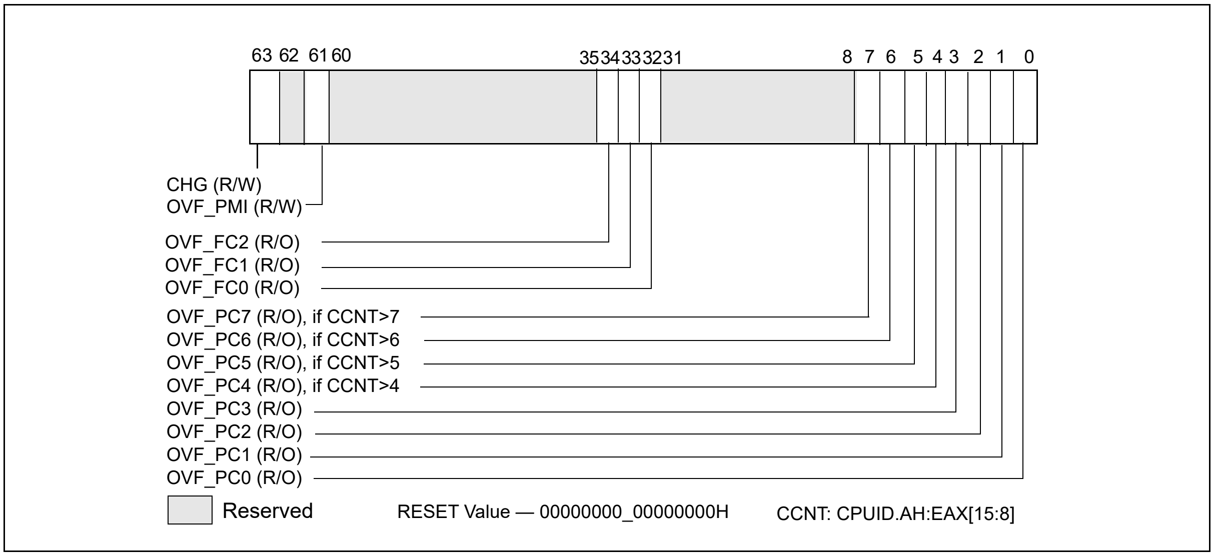


Figure 20-14. IA32_PERF_GLOBAL_STATUS MSR

20.3.1.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32_PMCx, associated counter configuration MSRs, IA32_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Nehalem

[1] Intel Xeon processor 5500 series and 3400 series are also based on Nehalem microarchitecture; the performance monitoring facilities described in this section generally also apply.

microarchitecture has been enhanced to include new data format to capture additional information, such as load latency.

- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Nehalem microarchitecture. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.

NOTE

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 20.2.1.

20.3.1.1.1 Processor Event Based Sampling (PEBS)

All general-purpose performance counters, IA32_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32_MISC_ENABLE[7] and IA32_MISC_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32_PEBS_ENABLE provides 4 bits that software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32_PEBS_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEBS_ENABLE for processors based on Nehalem microarchitecture is shown in Figure 20-15.

When a counter is enabled to capture machine state (PEBS_EN_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

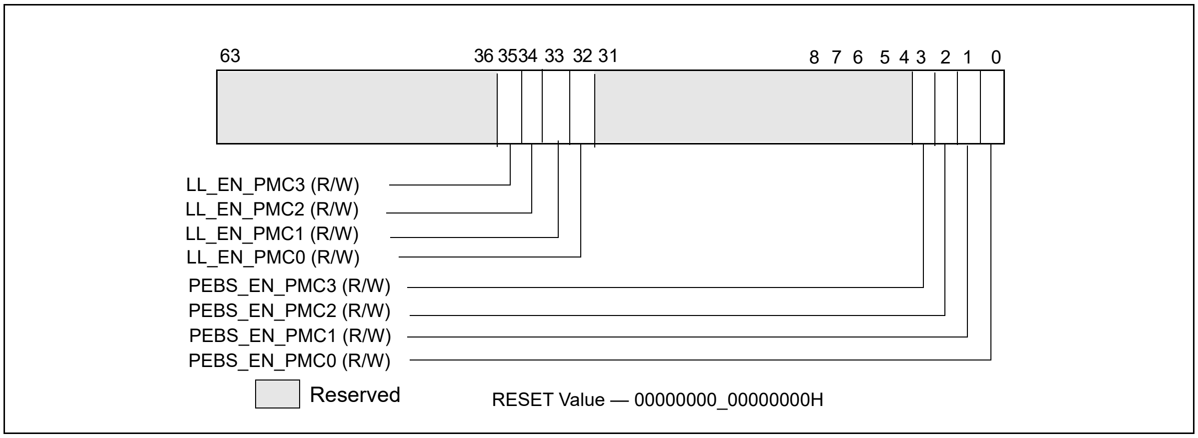


Figure 20-15. Layout of IA32_PEBS_ENABLE MSR

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32_PERF_CAPABILITIES[11:8] (see Figure 20-65).

The behavior of PEBS assists is reported by IA32_PERF_CAPABILITIES[6] (see Figure 20-65). The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

mov eax, [eax] ; causes PEBS assist

nop

are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

The PEBS record format is shown in Table 20-3, and each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 20-3. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	58H	R9
08H	R/EIP	60H	R10
10H	R/EAX	68H	R11
18H	R/EBX	70H	R12
20H	R/ECX	78H	R13
28H	R/EDX	80H	R14
30H	R/ESI	88H	R15
38H	R/EDI	90H	IA32_PERF_GLOBAL_STATUS
40H	R/EBP	98H	Data Linear Address
48H	R/ESP	A0H	Data Source Encoding
50H	R8	A8H	Latency value (core cycles)

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA-32e mode are written to zero.

Bytes AFH:90H are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32_PERF_CAPABILITIES[11:8] encoding of 0001B.

The value written to bytes 97H:90H is the state of the IA32_PERF_GLOBAL_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 20-84. In addition to using IA32_PERFEVTSELx to specify event unit/mask settings and setting the EN_PMCx bit in the IA32_PEBS_ENABLE register for the respective counter, the software must also initialize the DS_BUFFER_MANAGEMENT_AREA data structure in memory to support capturing PEBS records for precise events.

The recording of PEBS records may not operate properly if accesses to the linear addresses in the DS buffer management area or in the PEBS buffer (see below) cause page faults, VM exits, or the setting of accessed or dirty flags in the paging structures (ordinary or EPT). For that reason, system software should establish paging structures (both ordinary and EPT) to prevent such occurrences. Implications of this may be that an operating system should allocate this memory from a non-paged pool and that system software cannot do “lazy” page-table entry propagation for these pages. A virtual-machine monitor may choose to allow use of PEBS by guest software only if EPT maps all guest-physical memory as present and read/write.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

The beginning linear address of the DS_BUFFER_MANAGEMENT_AREA data structure must be programmed into the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 20-16.

- **PEBS Buffer Base**: This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. The processor reads this field to determine the base address of the PEBS buffer.
- **PEBS Index**: This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. The processor reads this field to determine the location of the next PEBS record to write to. After a PEBS record has been written, the processor also updates this field with the address of the next PEBS record to be written. The figure above illustrates the state of PEBS Index after the first PEBS record is written.
- **PEBS Absolute Maximum**: This field represents the absolute address of the maximum length of the allocated PEBS buffer plus the starting address of the PEBS buffer. The processor will not write any PEBS record beyond the end of PEBS buffer, when **PEBS Index** equals **PEBS Absolute Maximum**. No signaling is generated when PEBS buffer is full. Software must reset the **PEBS Index** field to the beginning of the PEBS buffer address to continue capturing PEBS records.

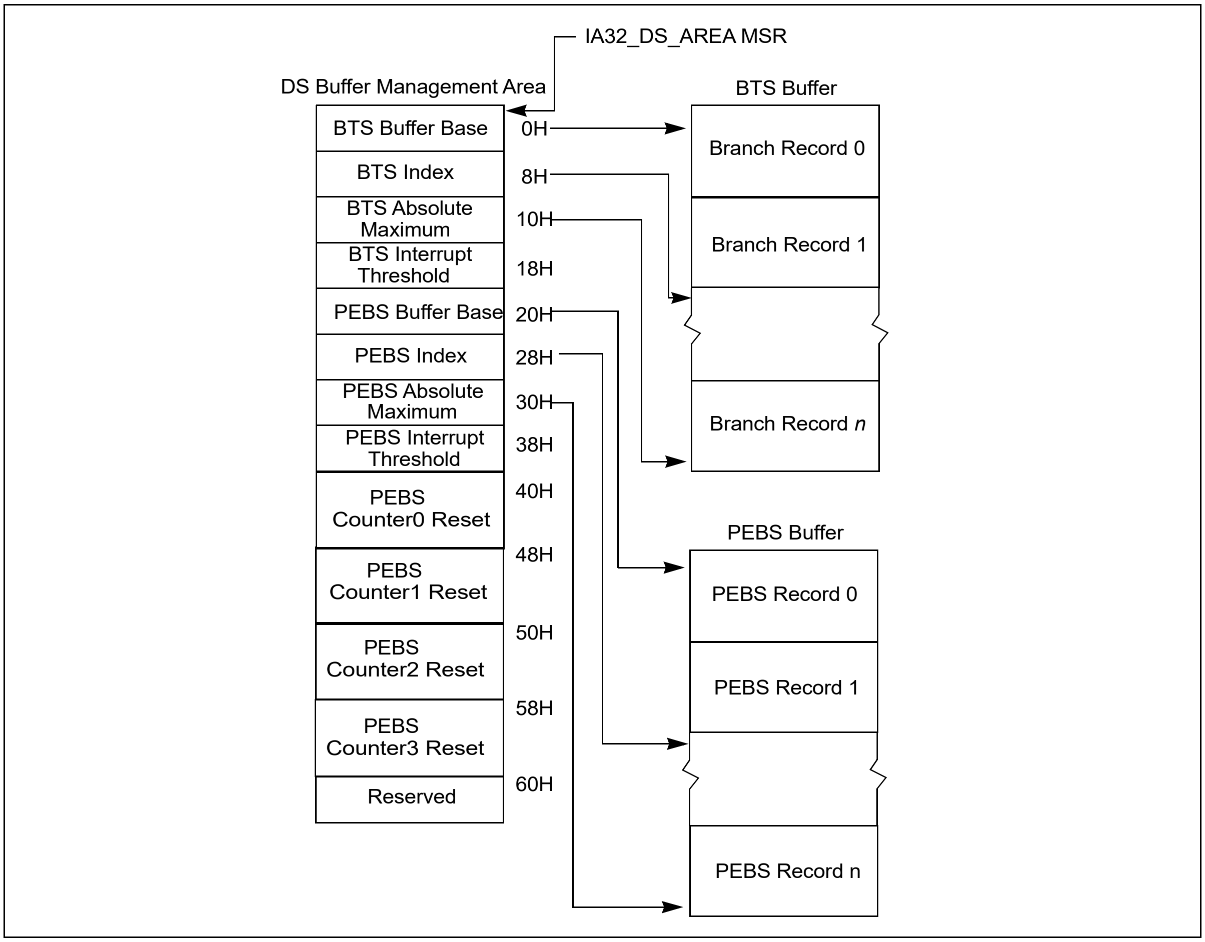


Figure 20-16. PEBS Programming Environment

- **PEBS Interrupt Threshold**: This field specifies the threshold value to trigger a performance interrupt and notify software that the PEBS buffer is nearly full. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After the processor writes a PEBS record and updates **PEBS Index**, if the **PEBS Index** reaches the threshold value of this field, the processor will generate a performance interrupt. This is the same interrupt that is generated by a performance counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When a performance interrupt due to PEBS buffer full is generated, the IA32_PERF_-GLOBAL_STATUS.PEBS_Ovf bit will be set.
- **PEBS CounterX Reset**: This field allows software to set up PEBS counter overflow condition to occur at a rate useful for profiling workload, thereby generating multiple PEBS records to facilitate characterizing the profile the execution of test code. After each PEBS record is written, the processor checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32_PEBS_ENABLED was set). If these conditions are met, then the reset value for each overflowed counter is loaded from the DS Buffer Management Area. For example, if counter IA32_PMC0 caused a PEBS record to be written, then the value of "PEBS Counter 0 Reset" would be written to counter IA32_PMC0. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist.

Performance Counter Prioritization

Performance monitoring interrupts are triggered by a counter transitioning from maximum count to zero (assuming IA32_PerfEvtSelX.INT is set). This same transition will cause PEBS hardware to arm, but not trigger. PEBS hardware triggers upon detection of the first PEBS event after the PEBS hardware has been armed (a 0 to 1 transition of the counter). At this point, a PEBS assist will be undertaken by the processor.

Performance counters (fixed and general-purpose) are prioritized in index order. That is, counter IA32_PMC0 takes precedence over all other counters. Counter IA32_PMC1 takes precedence over counters IA32_PMC2 and IA32_PMC3, and so on. This means that if simultaneous overflows or PEBS assists occur, the appropriate action will be taken for the highest priority performance counter. For example, if IA32_PMC1 cause an overflow interrupt and IA32_PMC2 causes an PEBS assist simultaneously, then the overflow interrupt will be serviced first.

The PEBS threshold interrupt is triggered by the PEBS assist, and is by definition prioritized lower than the PEBS assist. Hardware will not generate separate interrupts for each counter that simultaneously overflows. General-purpose performance counters are prioritized over fixed counters.

If a counter is programmed with a precise (PEBS-enabled) event and programmed to generate a counter overflow interrupt, the PEBS assist is serviced before the counter overflow interrupt is serviced. If in addition the PEBS interrupt threshold is met, the

threshold interrupt is generated after the PEBS assist completes, followed by the counter overflow interrupt (two separate interrupts are generated).

Uncore counters may be programmed to interrupt one or more processor cores (see Section 20.3.1.2). It is possible for interrupts posted from the uncore facility to occur coincident with counter overflow interrupts from the processor core. Software must check core and uncore status registers to determine the exact origin of counter overflow interrupts.

20.3.1.1.2 Load Latency Performance Monitoring Facility

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 20-3. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_INST_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 100H). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.

- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001_00000001H.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 20-3, bytes AFH:98H) consists of:

- **Data Linear Address**: This is the linear address of the target of the load operation.
- **Latency Value**: This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.
- **Data Source:** The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 20-4. In the descriptions, local memory refers to system memory physically attached to a processor package, and remote memory refers to system memory physically attached to another processor package.

Table 20-4. Data Source Encoding for Load Latency Record

Encoding	Description
00H	Unknown L3 cache miss.
01H	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
02H	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
03H	This data request was satisfied by the L2.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
06H	L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found.
07H[1]	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and were serviced by another core with a cross core snoop where modified copies were found.
08H	Reserved/L3 MISS. Local homed requests that missed the L3 cache and were serviced by forwarded data following a cross package snoop where no modified copies were found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to exclusive state).
0EH	I/O, Request of input/output operation.
0FH	The request was to un-cacheable memory.

NOTES:

1. Bit 7 is supported only for processors with a CPUID DisplayFamily_DisplayModel signature of 06_2A, and 06_2E; otherwise it is reserved.

The layout of MSR_PEBS_LD_LAT_THRESHOLD is shown in Figure 20-17.

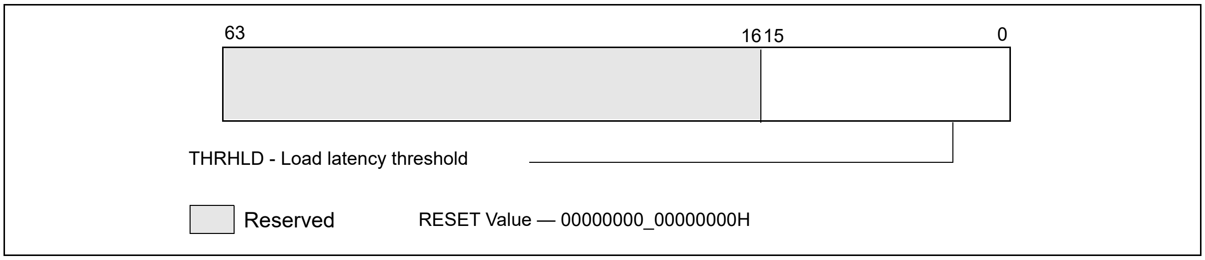

63
1615
0
THRHLD - Load latency threshold
Reserved
RESET Value — 00000000_00000000H

Figure 20-17. Layout of MSR_PEBS_LD_LAT MSR

Bits 15:0 specifies the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32_PMCx and their latency information is reported in the PEBS record. Otherwise, they are ignored. The minimum value that may be programmed in this field is 3.

20.3.1.1.3 Off-core Response Performance Monitoring in the Processor Core

Programming a performance event using the off-core response facility can choose any of the four IA32_PERFEVTSELx MSR with specific event codes and predefine mask bit value. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_0. There is only one off-core response configuration MSR. Table 20-5 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 20-5. Off-Core Response Event Encoding

Event code in IA32_PERFEVTSELx	**Mask Value in** IA32_PERFEVTSELx	**Required Off-core Response MSR**
B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)

The layout of MSR_OFFCORE_RSP_0 is shown in Figure 20-18. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.

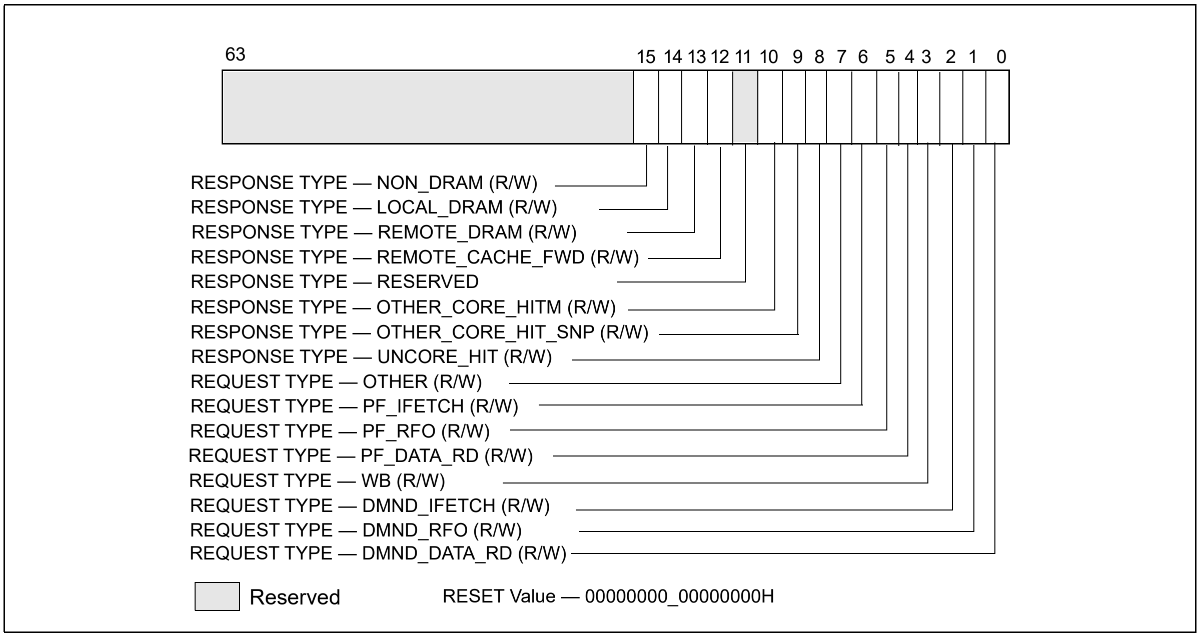

63
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
RESPONSE TYPE — NON_DRAM (R/W)
RESPONSE TYPE — LOCAL_DRAM (R/W)
RESPONSE TYPE — REMOTE_DRAM (R/W)
RESPONSE TYPE — REMOTE_CACHE_FWD (R/W)
RESPONSE TYPE — RESERVED
RESPONSE TYPE — OTHER_CORE_HITM (R/W)
RESPONSE TYPE — OTHER_CORE_HIT_SNP (R/W)
RESPONSE TYPE — UNCORE_HIT (R/W)
REQUEST TYPE — OTHER (R/W)
REQUEST TYPE — PF_IFETCH (R/W)
REQUEST TYPE — PF_RFO (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — WB (R/W)
REQUEST TYPE — DMND_IFETCH (R/W)
REQUEST TYPE — DMND_RFO (R/W)
REQUEST TYPE — DMND_DATA_RD (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 20-18. Layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 to Configure Off-core Response Events

Table 20-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
OTHER	7	Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HIT_SNP	9	L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).
OTHER_CORE_HITM	10	L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_FWD	12	L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.

Table 20-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition (Contd.)

Bit Name	Offset	Description
NON_DRAM	15	Non-DRAM requests that were serviced by IOH.

20.3.1.2 Performance Monitoring Facility in the Uncore

The "uncore" in Nehalem microarchitecture refers to subsystems in the physical processor package that are shared by multiple processor cores. Some of the sub-systems in the uncore include the L3 cache, Intel QuickPath Interconnect link logic, and integrated memory controller. The performance monitoring facilities inside the uncore operates in the same clock domain as the uncore (U-clock domain), which is usually different from the processor core clock domain. The uncore performance monitoring facilities described in this section apply to Intel Xeon processor 5500 series and processors with the following CPUID signatures: 06_1AH, 06_1EH, 06_1FH (see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4). An overview of the uncore performance monitoring facilities is described separately.

The performance monitoring facilities available in the U-clock domain consist of:

- Eight General-purpose counters (MSR_UNCORE_PerfCntr0 through MSR_UNCORE_PerfCntr7). The counters are 48 bits wide. Each counter is associated with a configuration MSR, MSR_UNCORE_PerfEvtSelx, to specify event code, event mask and other event qualification fields. A set of global uncore performance counter enabling/overflow/status control MSRs are also provided for software.
- Performance monitoring in the uncore provides an address/opcode match MSR that provides event qualification control based on address value or QPI command opcode.
- One fixed-function counter, MSR_UNCORE_FixedCntr0. The fixed-function uncore counter increments at the rate of the U-clock when enabled.

  The frequency of the uncore clock domain can be determined from the uncore clock ratio which is available in the PCI configuration space register at offset C0H under device number 0 and Function 0.

20.3.1.2.1 Uncore Performance Monitoring Management Facility

MSR_UNCORE_PERF_GLOBAL_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 20-19 shows the layout of MSR_UNCORE_PERF_GLOBAL_CTRL for an uncore that is shared by four processor cores in a physical package.

- EN_PCn (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR_UNCORE_PerfCntr n.
- EN_FC0 (bit 32): When set, enables counting for the fixed-function uncore counter MSR_UNCORE_FixedCntr0.
- EN_PMI_COREn (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32_DEBUGCTL.Offcore_PMI_EN to 1.
- PMI_FRZ (bit 63): When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in MSR_UNCORE_PERF_GLOBAL_CTRL upon exit from the ISR.

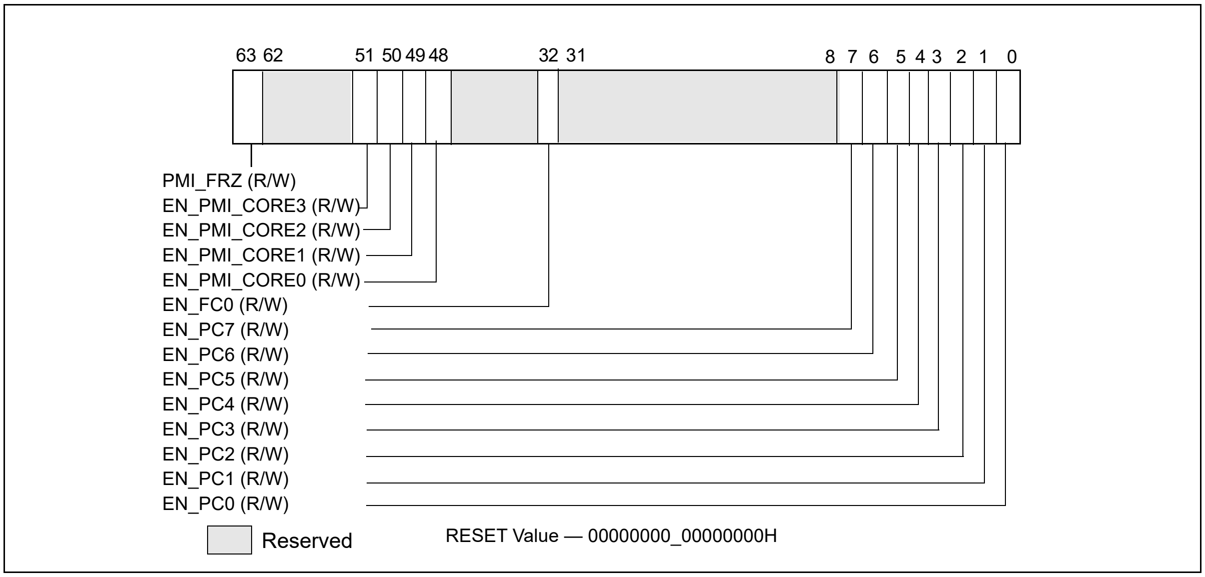


Figure 20-19. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR

MSR_UNCORE_PERF_GLOBAL_STATUS provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 20-20 shows the layout of MSR_UNCORE_PERF_GLOBAL_STATUS.

- OVF_PCn (bit n, n = 0, 7): When set, indicates general-purpose uncore counter MSR_UNCORE_PerfCntr n has overflowed.
- OVF_FC0 (bit 32): When set, indicates the fixed-function uncore counter MSR_UNCORE_FixedCntr0 has overflowed.
- OVF_PMI (bit 61): When set indicates that an uncore counter overflowed and generated an interrupt request.
- CHG (bit 63): When set indicates that at least one status bit in MSR_UNCORE_PERF_GLOBAL_STATUS register has changed state.

MSR_UNCORE_PERF_GLOBAL_OVF_CTRL allows software to clear the status bits in the UNCORE_PERF_GLOBAL_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.

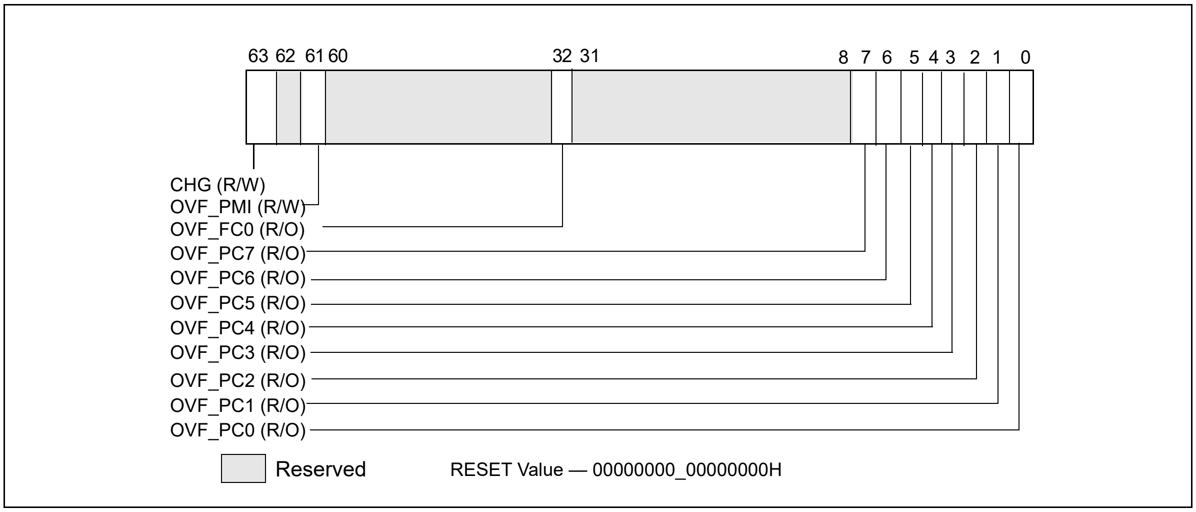


Figure 20-20. Layout of MSR_UNCORE_PERF_GLOBAL_STATUS MSR

Figure 20-21 shows the layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL.

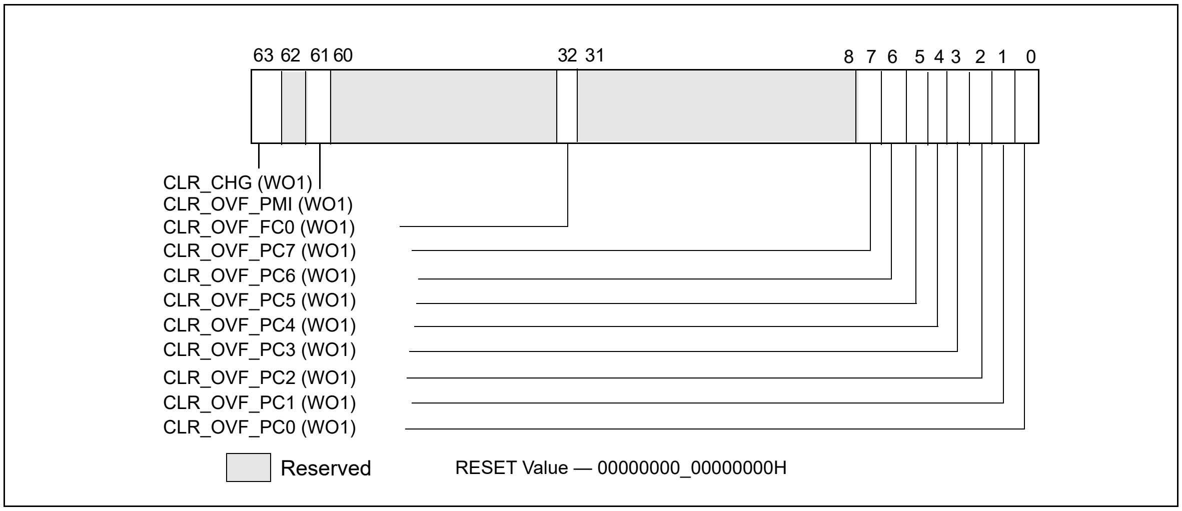


Figure 20-21. Layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL MSR

- CLR_OVF_PCn (bit n, n = 0, 7): Set this bit to clear the overflow status for general-purpose uncore counter MSR_UNCORE_PerfCntr n. Writing a value other than 1 is ignored.
- CLR_OVF_FC0 (bit 32): Set this bit to clear the overflow status for the fixed-function uncore counter MSR_UNCORE_FixedCntr0. Writing a value other than 1 is ignored.
- CLR_OVF_PMI (bit 61): Set this bit to clear the OVF_PMI flag in MSR_UNCORE_PERF_GLOBAL_STATUS. Writing a value other than 1 is ignored.
- CLR_CHG (bit 63): Set this bit to clear the CHG flag in MSR_UNCORE_PERF_GLOBAL_STATUS register. Writing a value other than 1 is ignored.

20.3.1.2.2 Uncore Performance Event Configuration Facility

MSR_UNCORE_PerfEvtSel0 through MSR_UNCORE_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corresponding MSR_UNCORE_PerfEvtSelx and counting must be enabled using the respective EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL. Figure 20-22 shows the layout of MSR_UNCORE_PERFEVTSELx.

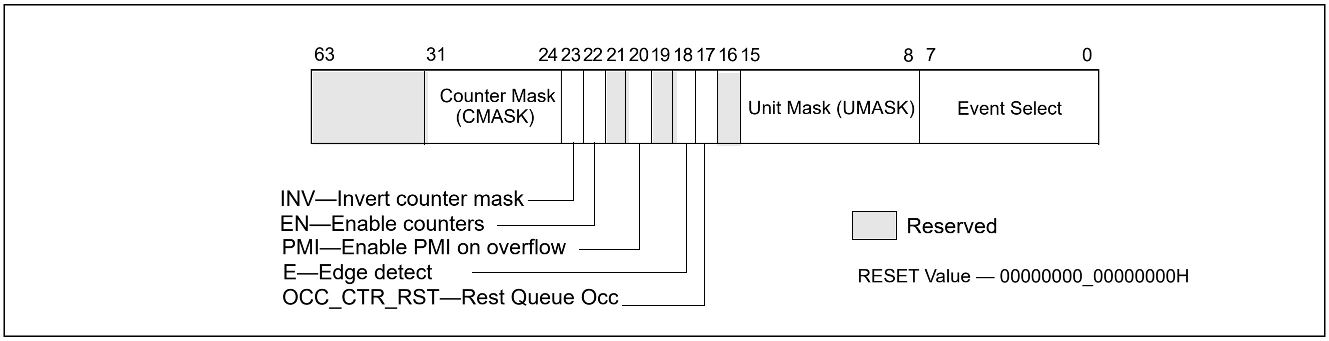


Figure 20-22. Layout of MSR_UNCORE_PERFEVTSELx MSRs

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8) : Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC_CTR_RST (bit17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 20-23 shows the layout of MSR_UNCORE_FIXED_CTR_CTRL.

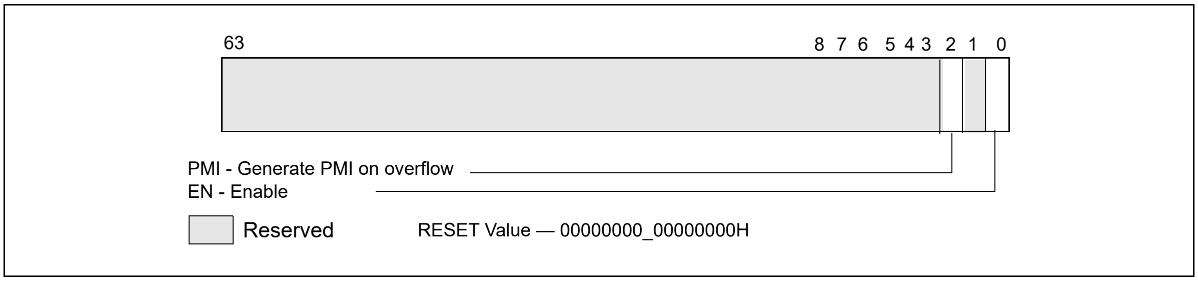

63
8
7
6
5
4
3
2
1
0
PMI - Generate PMI on overflow
EN - Enable
Reserved
RESET Value — 00000000_00000000H

Figure 20-23. Layout of MSR_UNCORE_FIXED_CTR_CTRL MSR

- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN_FC0 bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.

Both the general-purpose counters (MSR_UNCORE_PerfCntr) and the fixed-function counter (MSR_UNCORE_-FixedCntr0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

20.3.1.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR_UNCORE_PERFEVTSELx is used to select different uncore event logic unit. When the event "ADDR_OPCODE_MATCH" is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR_UNCORE_ADDR_OPCODE_MATCH register. The layout is shown in Figure 20-24.

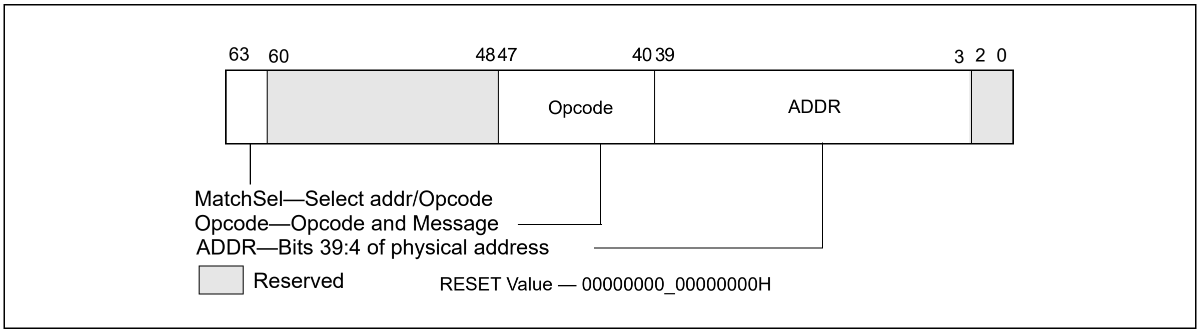

63
60
48
47
40
39
3
2
0
Opcode
ADDR
MatchSel—Select addr/Opcode
Opcode—Opcode and Message
ADDR—Bits 39:4 of physical address
▨ Reserved
RESET Value — 00000000_00000000H

Figure 20-24. Layout of MSR_UNCORE_ADDR_OPCODE_MATCH MSR

- Addr (bits 39:3): The physical address to match if "MatchSel" field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- Opcode (bits 47:40) : Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - — Bits 43:40 specify the QPI packet header opcode.
  - — Bits 47:44 specify the QPI message classes.

  Table 20-7 lists the encodings supported in the opcode field.

Table 20-7. Opcode Field Encoding for MSR_UNCORE_ADDR_OPCODE_MATCH

Opcode [43:40]	QPI Message Class		
	Home Request [47:44] = 0000B	Snoop Response [47:44] = 0001B	Data Response [47:44] = 1110B
		1	
DMND_IFETCH	2	2	
WB	3	3	
PF_DATA_RD	4	4	
PF_RFO	5	5	
PF_IFETCH	6	6	
OTHER	7	7	
NON_DRAM	15	15	

- MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  - 000B: Disable addr_opcode match hardware.
  - 100B: Count if only the address field matches.
  - 010B: Count if only the opcode field matches.
  - 110B: Count if either opcode field matches or the address field matches.
  - 001B: Count only if both opcode and address field match.
  - Other encoding are reserved.

20.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 20-25.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

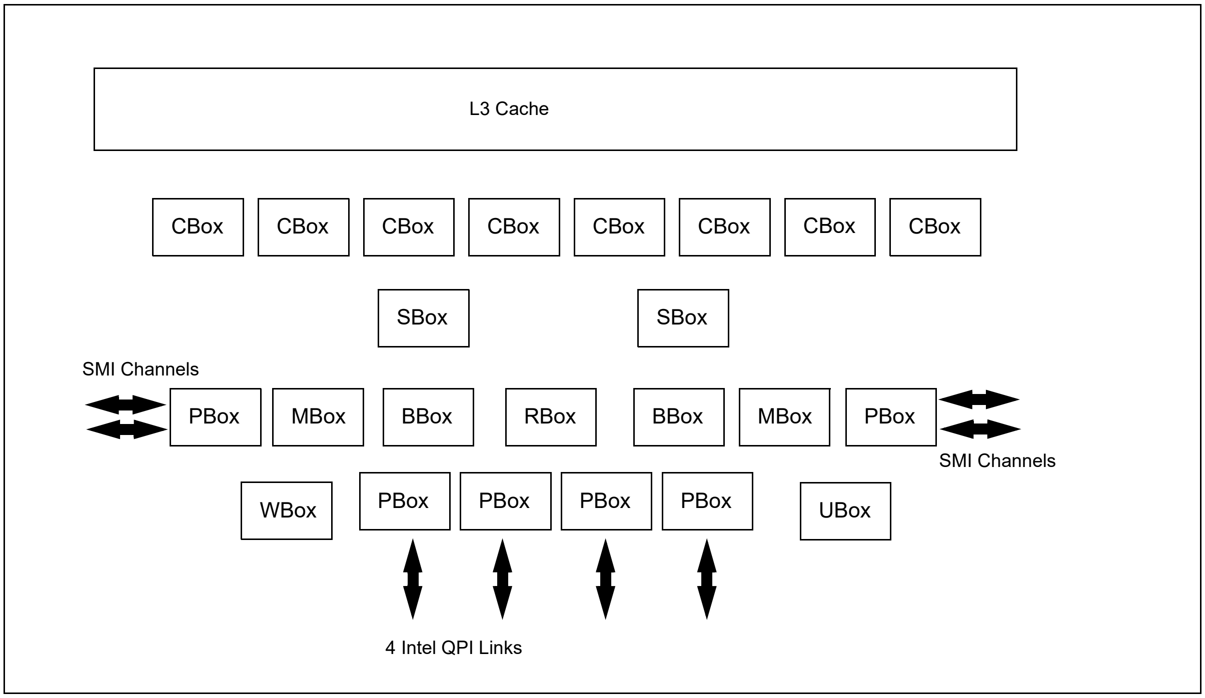


Figure 20-25. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 20-8 summarizes the number MSRs for uncore PMU for each box.

Table 20-8. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 (2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar the "global control" programming interface, IA32_PERF_-GLOBAL_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR_U_PMON_GLOBAL_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual MSRs that provide uncore PMU interfaces are listed in Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, Table 2-17 under the general naming style of MSR_%box#%_PMON_%scope_function%, where %box#% designates the type of box and zero-based index if there are more the one box of the same type, %scope_function% follows the examples below:

- Multi-counter enabling MSRs: MSR_U_PMON_GLOBAL_CTL, MSR_S0_PMON_BOX_CTL, MSR_C7_PMON_-BOX_CTL, etc.
- Multi-counter status MSRs: MSR_U_PMON_GLOBAL_STATUS, MSR_S0_PMON_BOX_STATUS, MSR_C7_P-MON_BOX_STATUS, etc.
- Multi-counter overflow control MSRs: MSR_U_PMON_GLOBAL_OVF_CTL, MSR_S0_PMON_BOX_OVF_CTL, MSR_C7_PMON_BOX_OVF_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g., MSR_U_PMON_CTR, MSR_S0_P-MON_CTR0, MSR_C7_PMON_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g., MSR_U_PMON_EVNT_SEL, MSR_S0_P-MON_EVNT_SEL0, MSR_C7_PMON_EVNT_SEL5, etc.
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g., MSR_M0_PMON_TIMESTAMP, MSR_R0_P-MON_IPERF0_P1, MSR_S1_PMON_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document "Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide".

20.3.2 Performance Monitoring for Processors Based on Westmere Microarchitecture

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 20.6.3 also apply to processors based on Westmere microarchitecture.

Table 20-5 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR_OFFCORE_RSP_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore response event using event code 0BBH and MSR_OFFCORE_RSP_1 (address 1A7H) are supported in processors based on Westmere microarchitecture. The event code and event mask definitions of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

The load latency facility is the same as described in Section 20.3.1.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB_MISS and LOCK, see Table 20-13.

20.3.3 Intel® Xeon® Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel® Xeon® processor E7 family is the same as those supported in the Intel Xeon processor 5600 series[1]. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore subsystem is similar to that shown in Figure 20-25, with the additional capability that up to 10 C-Box units are supported.

Table 20-9 summarizes the number MSRs for uncore PMU for each box.

Table 20-9. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	10	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 (2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E7 family is available in the “Intel® Xeon® Processor E7 Uncore Performance Monitoring Programming Reference Manual”.

20.3.4 Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture

Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel® Xeon® processor E3-1200 family are based on Sandy Bridge microarchitecture; this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 20.2.3.

The core PMU’s capability is similar to those described in Section 20.3.1.1 and Section 20.6.3, with some differences and enhancements relative to Westmere microarchitecture summarized in Table 20-10.

[1] Exceptions are indicated for event code 0FH in the event list for this processor (https://perfmon-events.intel.com/); and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 20-13.

Table 20-10. Core PMU Comparison

Box	Sandy Bridge Microarchitecture	Westmere Microarchitecture	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 20.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 20.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 20.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to determine # of counters. See Section 20.2.1.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 18.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 20-12.	See Table 20-84.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 20.3.4.4.2; ▪ Data source encoding ▪ STLB miss encoding ▪ Lock transaction encoding	Data source encoding	
PEBS-Precise Store	Section 20.3.4.4.3	No	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL).	No	
Off-core Response Event	MSR 1A6H and 1A7H, extended request and response types.	MSR 1A6H and 1A7H, limited response types.	Nehalem supports 1A6H only.

20.3.4.1 Global Counter Control Facilities in Sandy Bridge Microarchitecture

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Sandy Bridge microarchitecture. Software must use CPUID to determine the number performance counters/event select registers (See Section 20.2.1.1).

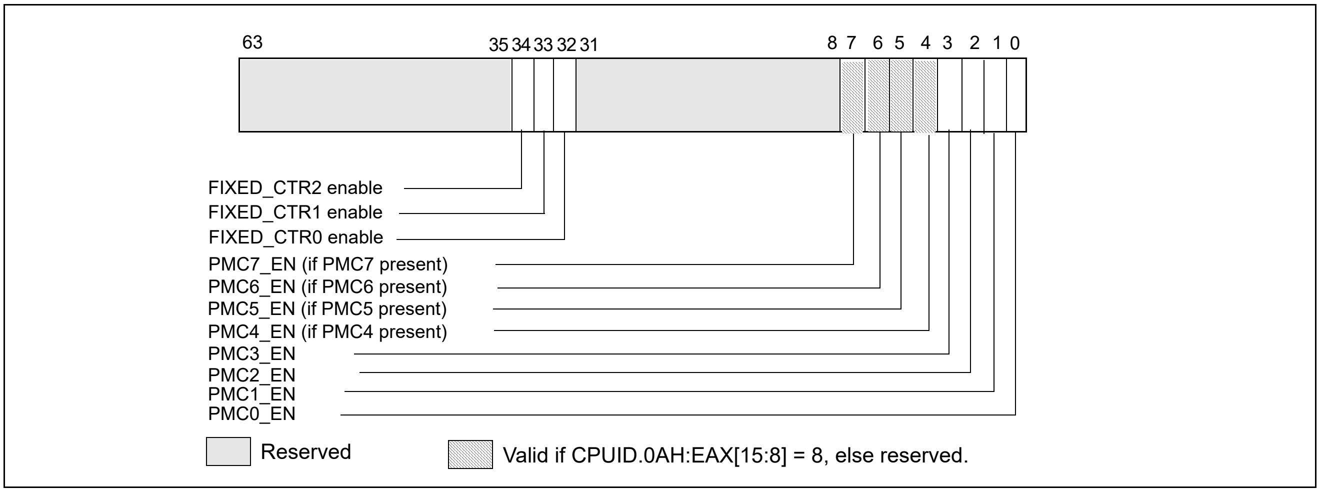

63
35
34
33
32
31
8
7
6
5
4
3
2
1
0
FIXED_CTR2 enable
FIXED_CTR1 enable
FIXED_CTR0 enable
PMC7_EN (if PMC7 present)
PMC6_EN (if PMC6 present)
PMC5_EN (if PMC5 present)
PMC4_EN (if PMC4 present)
PMC3_EN
PMC2_EN
PMC1_EN
PMC0_EN
Reserved
Valid if CPUID.0AH:EAX[15:8] = 8, else reserved.

Figure 20-26. IA32_PERF_GLOBAL_CTRL MSR in Sandy Bridge Microarchitecture

Figure 20-44 depicts the layout of IA32_PERF_GLOBAL_CTRL MSR. The enable bits (PMC4_EN, PMC5_EN, PMC6_EN, PMC7_EN) corresponding to IA32_PMC4-IA32_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of '8'. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.
IA32_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer (see Figure 20-27). A value of 1 in each bit of the PMCx_OVF field indicates an overflow condition has occurred in the associated counter.

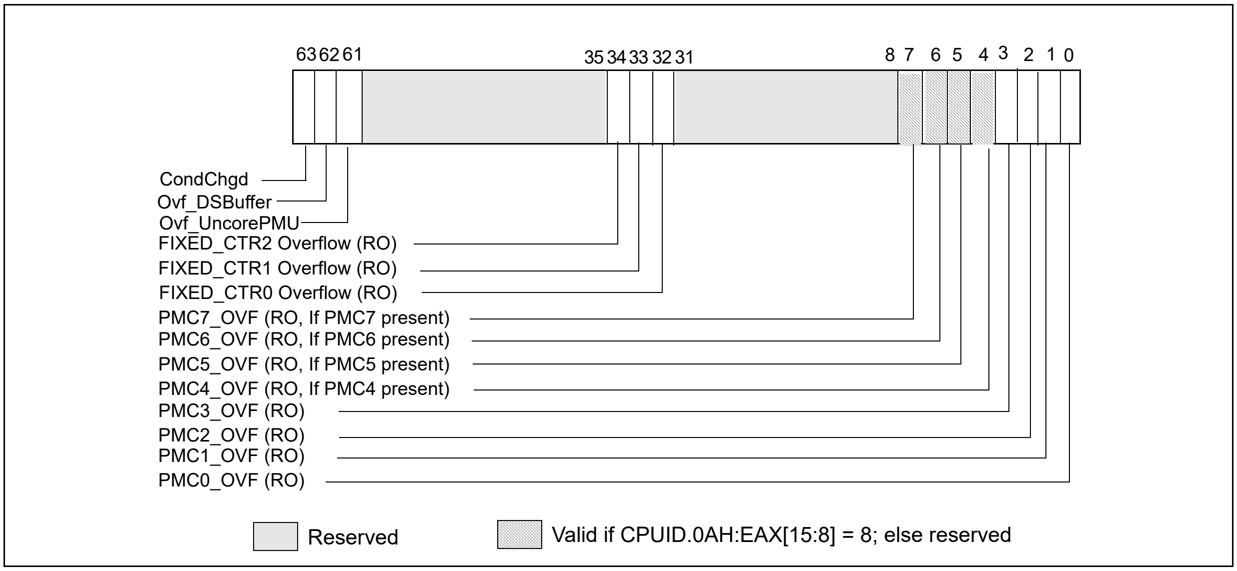


Figure 20-27. IA32_PERF_GLOBAL_STATUS MSR in Sandy Bridge Microarchitecture

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 18.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 20-28). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt based sampling.
- Reloading counter values to continue sampling.
- Disabling event counting or interrupt based sampling.

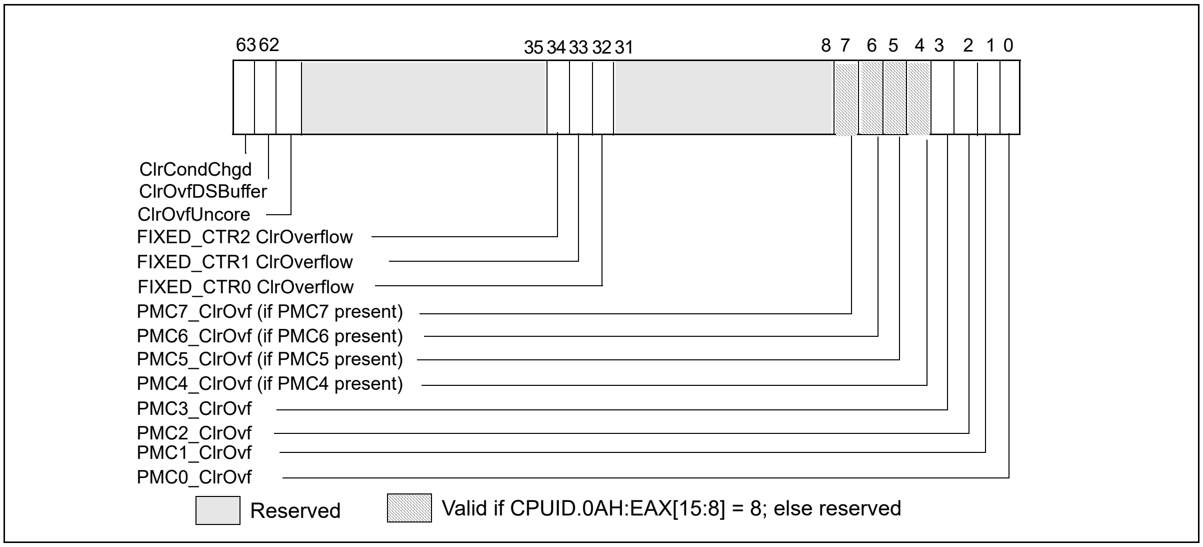


Figure 20-28. IA32_PERF_GLOBAL_OVF_CTRL MSR in Sandy Bridge Microarchitecture

20.3.4.2 Counter Coalescence

In processors based on Sandy Bridge microarchitecture, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report the number of counters visible to software.

If a processor core is shared by two logical processors, each logical processors can access up to four counters (IA32_PMC0-IA32_PMC3). This is the same as in the prior generation for processors based on Nehalem microarchitecture.

If a processor core is not shared by two logical processors, up to eight general-purpose counters are visible. If CPUID.0AH:EAX[15:8] reports 8 counters, then IA32_PMC4-IA32_PMC7 would occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32_PERFEVTSEL4-IA32_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32_PMC4-IA32_PMC7, IA32_PMC4-IA32_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, or IA32_PERF_-GLOBAL_OVF_CTL will also cause #GP.

20.3.4.3 Full Width Writes to Performance Counters

Processors based on Sandy Bridge microarchitecture support full-width writes to the general-purpose counters, IA32_PMCx. Support of full-width writes are enumerated by IA32_PERF_CAPABILITIES.FW_WRITES[13] (see Section 20.2.4).

The default behavior of IA32_PMCx is unchanged, i.e., WRMSR to IA32_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32_A_PMCx by testing IA32_PERF_CAPABILITIES[13].

20.3.4.4 PEBS Support in Sandy Bridge Microarchitecture

Processors based on Sandy Bridge microarchitecture support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Westmere microarchitecture is summarized in Table 20-11.

Table 20-11. PEBS Facility Comparison

Box	Sandy Bridge Microarchitecture	Westmere Microarchitecture	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 20.3.1.1.1	Section 20.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 20-29	Figure 20-15	
PEBS record layout	Physical Layout same as Table 20-3.	Table 20-3	Enhanced fields at offsets 98H, A0H, A8H.
PEBS Events	See Table 20-12.	See Table 20-84.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 20-13.	Table 20-4	
PEBS-Precise Store	Yes; see Section 20.3.4.4.3.	No	IA32_PMC3 only
PEBS-PDIR	Yes	No	IA32_PMC1 only
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

In IA32_PEBS_ENABLE MSR, bit 63 is defined as PS_ENABLE: When set, this enables IA32_PMC3 to capture precise store information. Only IA32_PMC3 supports the precise store facility. In typical usage of PEBS, the bit fields in IA32_PEBS_ENABLE are written to when the agent software starts PEBS operation; the enabled bit fields should be modified only when re-programming another PEBS event or cleared when the agent uses the performance counters for non-PEBS operations.

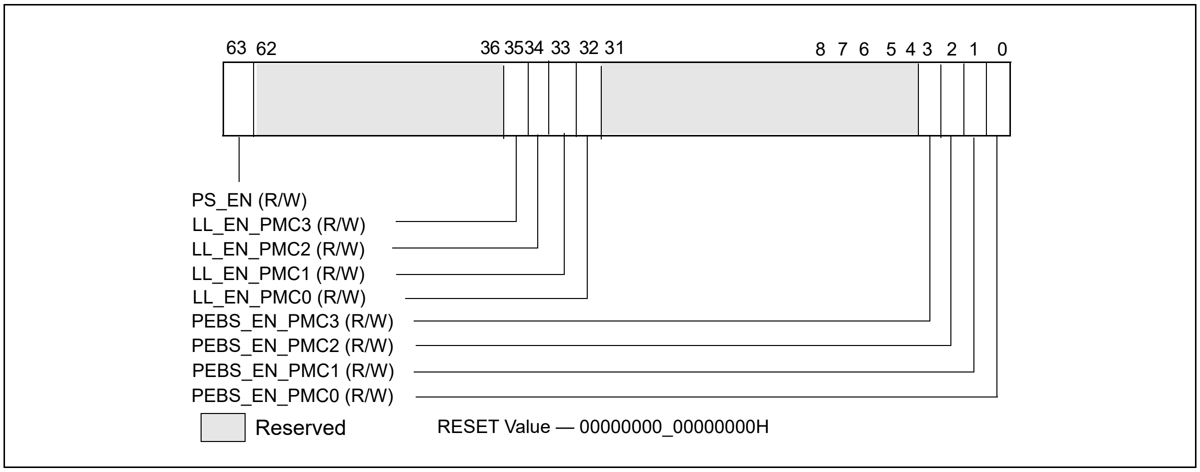


Figure 20-29. Layout of IA32_PEBS_ENABLE MSR

20.3.4.4.1 PEBS Record Format

The layout of PEBS records physically identical to those shown in Table 20-3, but the fields at offsets 98H, A0H, and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H): When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 20-4. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32_PERF_GLOBAL_STATUS corresponding to those counters that both overflowed and were enabled in the IA32_PEBS_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Sandy Bridge microarchitecture is shown in Table 20-12.

Table 20-12. PEBS Performance Events for Sandy Bridge Microarchitecture

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST	01H[1]
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Near_Taken	20H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H

Table 20-12. PEBS Performance Events for Sandy Bridge Microarchitecture (Contd.)

Event Name	Event Select	Sub-event	UMask
MEM_UOPS_RETIRED	D0H	STLB_MISS_LOADS	11H
		STLB_MISS_STORE	12H
		LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H

NOTES:

1. Only available on IA32_PMC1.

20.3.4.4.2 Load Latency Performance Monitoring Facility

The load latency facility in Sandy Bridge microarchitecture is similar to that in prior microarchitectures. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 20-3 and Section 20.3.4.4.1. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_TRANS_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001.00000001H.
- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM_TRANS_RETIRED event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 20-3. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.

Table 20-13. Layout of Data Source Field of Load Latency Record

Field	Position	Description
Source	3:0	See Table 20-4
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

The layout of MSR_PEBS_LD_LAT_THRESHOLD is the same as shown in Figure 20-17.

20.3.4.4.3 Precise Store Facility

Processors based on Sandy Bridge microarchitecture offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.
- Program the MEM_TRANS_RETIRED.PRECISE_STORE event in IA32_PERFEVTSEL3. Only counter 3 (IA32_PMC3) supports collection of precise store information.
- Set IA32_PEBS_ENABLE[3] and IA32_PEBS_ENABLE[63]. This enables IA32_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offsets 98H, A0H, and A8H of Table 20-3. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 20-14. Layout of Precise Store Information In PEBS Record

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	**L1D Hit** (Bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. **STLB Miss** (bit 4): The store missed the STLB if set, otherwise the store hit the STLB **Locked Access** (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.
Reserved	A8H	Reserved

20.3.4.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable "skids" in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Sandy Bridge microarchitecture include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the "skid" problem by providing an early indication of when the INST_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow. On processors based on Sandy Bridge microarchitecture, skid is significantly reduced and can be as little as one instruction. On future implementations, PDIR may eliminate skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and processors based on Sandy Bridge microarchitecture must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily_DisplayModel signatures of 06_2A and 06_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

20.3.4.5 Off-core Response Performance Monitoring

The core PMU in processors based on Sandy Bridge microarchitecture provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Table 20-15 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 20-15. Off-Core Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMC0-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 are shown in Figure 20-30 and Figure 20-31. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

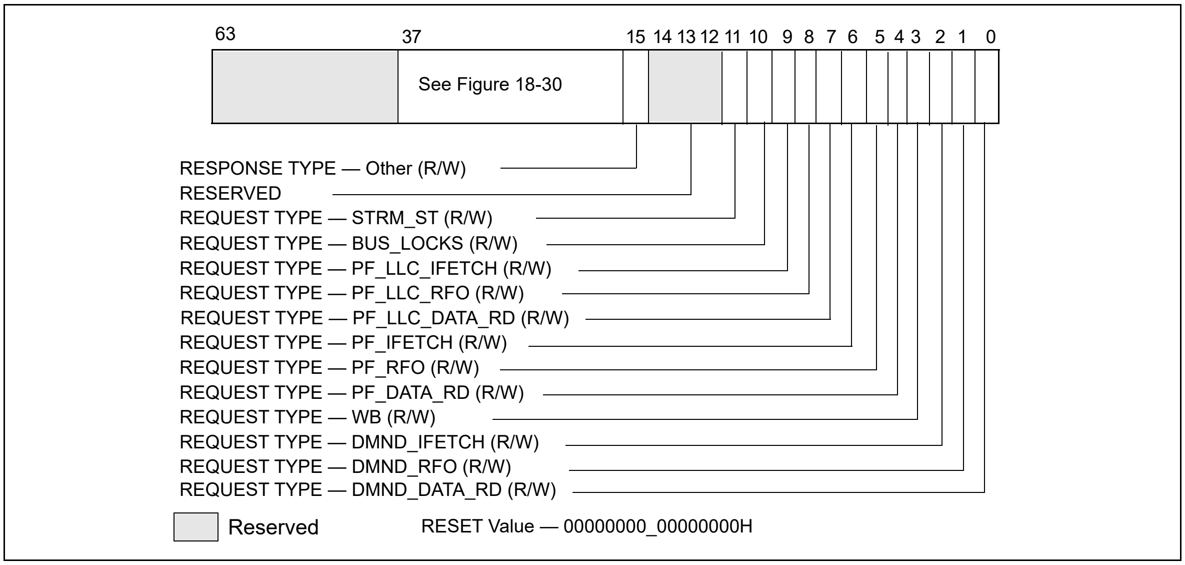

63
37
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
See Figure 18-30
RESPONSE TYPE — Other (R/W)
RESERVED
REQUEST TYPE — STRM_ST (R/W)
REQUEST TYPE — BUS_LOCKS (R/W)
REQUEST TYPE — PF_LLC_IFETCH (R/W)
REQUEST TYPE — PF_LLC_RFO (R/W)
REQUEST TYPE — PF_LLC_DATA_RD (R/W)
REQUEST TYPE — PF_IFETCH (R/W)
REQUEST TYPE — PF_RFO (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — WB (R/W)
REQUEST TYPE — DMND_IFETCH (R/W)
REQUEST TYPE — DMND_RFO (R/W)
REQUEST TYPE — DMND_DATA_RD (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 20-30. Request_Type Fields for MSR_OFFCORE_RSP_x

Table 20-16. MSR_OFFCORE_RSP_x Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests
STRM_ST	11	Streaming store requests
OTHER	15	Any other request that crosses IDI, including I/O.

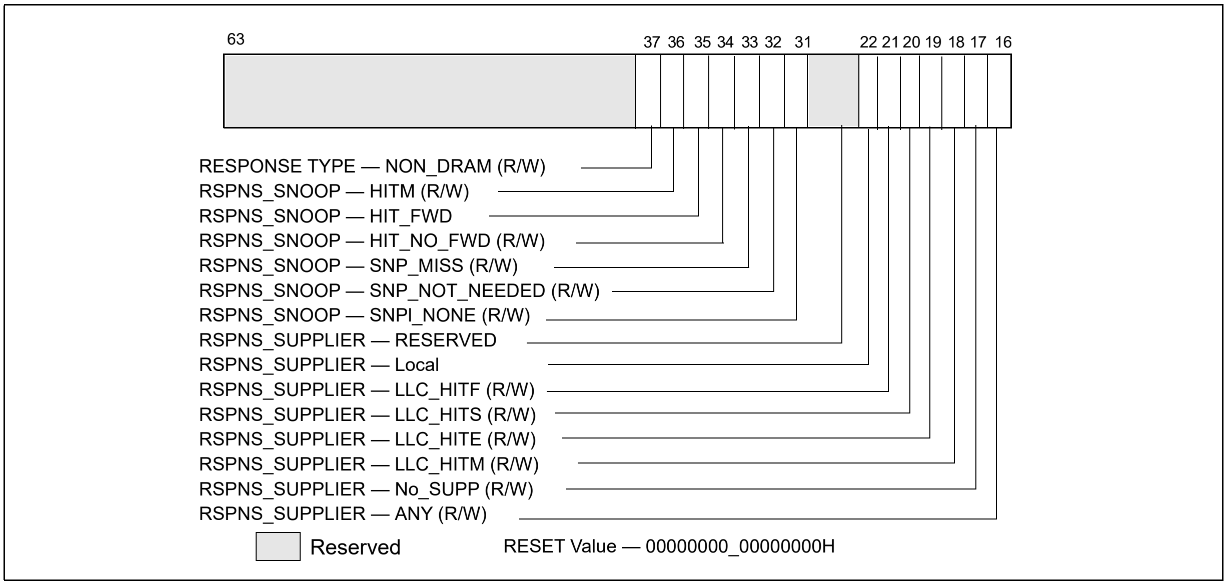

63
37 36 35 34 33 32 31
22 21 20 19 18 17 16
RESPONSE TYPE — NON_DRAM (R/W)
RSPNS_SNOOP — HITM (R/W)
RSPNS_SNOOP — HIT_FWD
RSPNS_SNOOP — HIT_NO_FWD (R/W)
RSPNS_SNOOP — SNP_MISS (R/W)
RSPNS_SNOOP — SNP_NOT_NEEDED (R/W)
RSPNS_SNOOP — SNPl_NONE (R/W)
RSPNS_SUPPLIER — RESERVED
RSPNS_SUPPLIER — Local
RSPNS_SUPPLIER — LLC_HITF (R/W)
RSPNS_SUPPLIER — LLC_HITS (R/W)
RSPNS_SUPPLIER — LLC_HITE (R/W)
RSPNS_SUPPLIER — LLC_HITM (R/W)
RSPNS_SUPPLIER — No_SUPP (R/W)
RSPNS_SUPPLIER — ANY (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 20-31. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSP_x

To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSP_x allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 20-17. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Reserved	30:23	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY“ bit is set, the supplier and snoop info bits are ignored.

Table 20-18. MSR_OFFCORE_RSP_x Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	SNP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNP_MISS	33	A snoop was needed and it missed all snooped caches: -For LLC Hit, ResIHitI was returned by all cores -For LLC Miss, RspI was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.

20.3.4.6 Uncore Performance Monitoring Facilities in the Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, and Intel® Core™ i3-2xxx Processor Series

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 20.3.1.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 20-32.

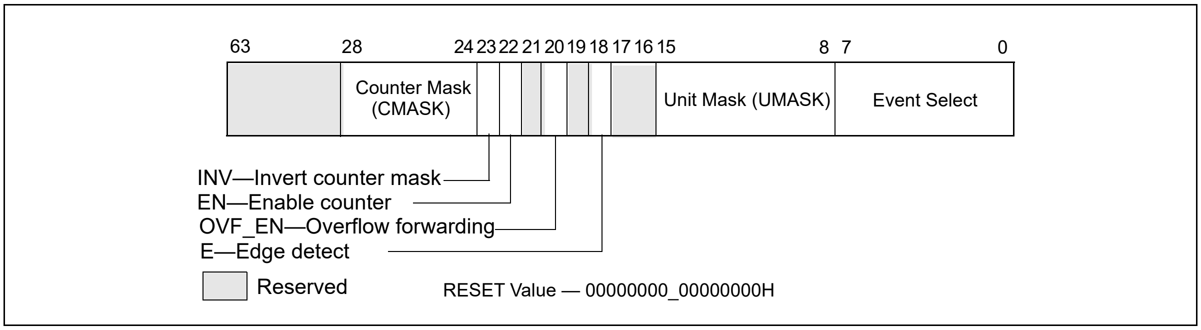


Figure 20-32. Layout of Uncore PERFEVTSEL MSR for a C-Box Unit or the ARB Unit

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see the event list at: https://perfmon-events.intel.com/.
- E (bit 18): Enables edge detection filtering, if 1.
- OVF_EN (bit 20): Enables the overflow indicator from the uncore counter forwarded to MSR_UNC_PERF_-GLOBAL_CTRL, if 1.
- EN (bit 22): Enables the local counter associated with this event select MSR.
- INV (bit 23): Event count increments with non-negative value if 0, with negated value if 1.
- CMASK (bits 28:24): Specifies a positive threshold value to filter raw event count input.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 20-33 shows the layout of the uncore domain global control.

When an uncore counter overflows, a PMI can be routed to a processor core. Bits 3:0 of MSR_UNC_PERF_-GLOBAL_CTRL can be used to select which processor core to handle the uncore PMI. Software must then write to bit 13 of IA32_DEBUGCTL (at address 1D9H) to enable this capability.

- PMI_SEL_Core#: Enables the forwarding of an uncore PMI request to a processor core, if 1. If bit 30 (WakePMI) is '1', a wake request is sent to the respective processor core prior to sending the PMI.
- EN: Enables the fixed uncore counter, the ARB counters, and the CBO counters in the uncore PMU, if 1. This bit is cleared if bit 31 (FREEZE) is set and any enabled uncore counters overflow.
- WakePMI: Controls sending a wake request to any halted processor core before issuing the uncore PMI request. If a processor core was halted and not sent a wake request, the uncore PMI will not be serviced by the processor core.
- FREEZE: Provides the capability to freeze all uncore counters when an overflow condition occurs in a unit counter. When this bit is set, and a counter overflow occurs, the uncore PMU logic will clear the global enable bit (bit 29).

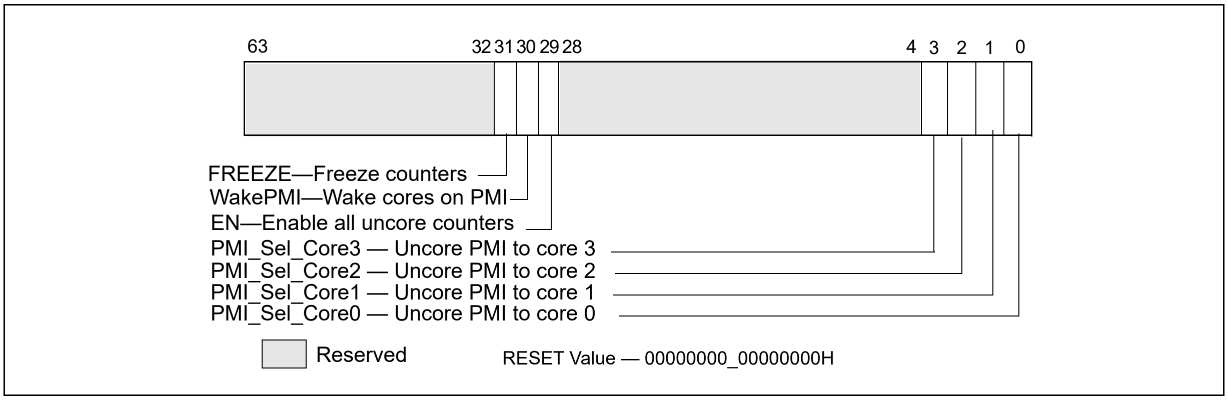

63
32
31
30
29
28
4
3
2
1
0
FREEZE—Freeze counters
WakePMI—Wake cores on PMI
EN—Enable all uncore counters
PMI_Sel_Core3 — Uncore PMI to core 3
PMI_Sel_Core2 — Uncore PMI to core 2
PMI_Sel_Core1 — Uncore PMI to core 1
PMI_Sel_Core0 — Uncore PMI to core 0
Reserved
RESET Value — 00000000_00000000H

Figure 20-33. Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 20-19 summarizes the number MSRs for uncore PMU for each box.

Table 20-19. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-21 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	

Table 20-19. Uncore PMU MSR Summary (Contd.)

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
Fixed Counter	N.A.	N.A.	48	No	Uncore	

20.3.4.6.1 Uncore Performance Monitoring Events

There are certain restrictions on the uncore performance counters in each C-Box. Specifically,

- Occupancy events are supported only with counter 0 but not counter 1.
- Other uncore C-Box events can be programmed with either counter 0 or 1.

The C-Box uncore performance events can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE_RESPONSE and qualified on a per-thread basis.

On the other hand, uncore performance event logic cannot associate its counts with the same level of per-thread qualification attributes as the core PMU events can. Therefore, whenever similar event programming capabilities are available from both core PMU and uncore PMU, the recommendation is that utilizing the core PMU events may be less affected by artifacts, complex interactions and other factors.

20.3.4.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility

The Intel® Xeon® Processor E5 Family (and Intel® Core™ i7-3930K Processor) are based on Sandy Bridge-E microarchitecture. While the processor cores share the same microarchitecture as those of the Intel® Xeon® Processor E3 Family and 2nd generation Intel Core i7-2xxx, Intel Core i5-2xxx, Intel Core i3-2xxx processor series, the uncore subsystems are different. An overview of the uncore performance monitoring facilities of the Intel Xeon processor E5 family (and Intel Core i7-3930K processor) is described in Section 20.3.4.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 20.6.3 through Section 20.3.4.5. However, the MSR_OFFCORE_RSP_0/MSR_OFFCORE_RSP_1 Response Supplier Info field shown in Table 20-17 applies to Intel Core Processors with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2AH; Intel Xeon processor with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2DH supports an additional field for remote DRAM controller shown in Table 20-20. Additionally, there are some small differences in the non-architectural performance monitoring events (see event list available at: https://perfmon-events.intel.com/).

Table 20-20. MSR_OFFCORE_RSP_x Supplier Info Field Definitions

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Remote	30:23	Remote DRAM Controller (either all 0s or all 1s).

20.3.4.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5-2600 product family has some similarities with those of the Intel Xeon processor E7 family. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope. For example, each Cbox caching agent has a set of local performance counters, and the power controller unit (PCU) has its own local performance counters. Up to 8 C-Box units are supported in the uncore sub-system.

Table 20-21 summarizes the uncore PMU facilities providing MSR interfaces.

Table 20-21. Uncore PMU MSR Summary for Intel® Xeon® Processor E5 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	4	44	Yes	per-box	None
PCU	1	4	48	Yes	per-box	Match/Mask
U-Box	1	2	44	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 family is available in “Intel® Xeon® Processor E5 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-24.

20.3.5 3rd Generation Intel® Core™ Processor Performance Monitoring Facility

The 3rd generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v2 product family are based on the Ivy Bridge microarchitecture. The performance monitoring facilities in the processor core generally are the same as those described in Section 20.6.3 through Section 20.3.4.5. The non-architectural performance monitoring events supported by the processor core can be found at: https://perfmon-events.intel.com/.

20.3.5.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5 v2 and Intel Xeon Processor E7 v2 product families are based on the Ivy Bridge-E microarchitecture. There are some similarities with those of the Intel Xeon processor E5 family based on the Sandy Bridge microarchitecture. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope.

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v2 and Intel Xeon Processor E7 v2 families are available in the “Intel® Xeon® Processor E5 v2 and E7 v2 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-28.

20.3.6 4th Generation Intel® Core™ Processor Performance Monitoring Facility

The 4th generation Intel® Core™ processor and Intel® Xeon® processor E3-1200 v3 product family are based on the Haswell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 20.2.3.

The core PMU’s capability is similar to those described in Section 20.6.3 through Section 20.3.4.5, with some differences and enhancements summarized in Table 20-22. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 20.3.6.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Table 20-22. Core PMU Comparison

Box	Haswell Microarchitecture	Sandy Bridge Microarchitecture	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 20.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 20.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 20.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to determine # of counters. See Section 20.2.1.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 18.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 20-12 and Section 20.3.6.5.1.	See Table 20-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 20.3.4.4.2.	See Section 20.3.4.4.2.	
PEBS-Precise Store	No, replaced by Data Address profiling.	Section 20.3.4.4.3	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL)	Yes (using precise INST_RETIRED.ALL)	
PEBS-EventingIP	Yes	No	
Data Address Profiling	Yes	No	
LBR Profiling	Yes	Yes	
Call Stack Profiling	Yes, see Section 18.11.	No	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; extended request and response types.	MSR 1A6H and 1A7H; extended request and response types.	
Intel TSX support for Perfmon	See Section 20.3.6.5.	No	

20.3.6.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 4th Generation Intel Core processor is similar to those in processors based on Sandy Bridge microarchitecture, with several enhanced features. The key components and differences of PEBS facility relative to Sandy Bridge microarchitecture is summarized in Table 20-23.

Table 20-23. PEBS Facility Comparison

Box	Haswell Microarchitecture	Sandy Bridge Microarchitecture	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7
PEBS Buffer Programming	Section 20.3.1.1.1	Section 20.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 20-15	Figure 20-29	
PEBS record layout	Table 20-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	Table 20-3; enhanced fields at offsets 98H, A0H, A8H.	

Table 20-23. PEBS Facility Comparison

Box	Haswell Microarchitecture	Sandy Bridge Microarchitecture	Comment
Precise Events	See Table 20-12.	See Table 20-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 20-13.	Table 20-13	
PEBS-Precise Store	No, replaced by data address profiling.	Yes; see Section 20.3.4.4.3.	
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

20.3.6.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 20-24. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 20-24. PEBS Record Format for 4th Generation Intel Core Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Data Linear Address
40H	R/EBP	A0H	Data Source Encoding
48H	R/ESP	A8H	Latency value (core cycles)
50H	R8	B0H	EventingIP
58H	R9	B8H	TX Abort Information (Section 20.3.6.5.1)

The layout of PEBS records are almost identical to those shown in Table 20-3. Offset B0H is a new field that records the eventing IP address of the retired instruction that triggered the PEBS assist.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 20.3.4.4.2), PDIR (Section 20.3.4.4.4), and the equivalent capability of precise store in prior generation (see Section 20.3.6.3).

In the core PMU of the 4th generation Intel Core processor, load latency facility and PDIR capabilities are unchanged. However, precise store is replaced by an enhanced capability, data address profiling, that is not restricted to store address. Data address profiling also records information in PEBS records at offsets 98H, A0H, and ABH.

20.3.6.3 PEBS Data Address Profiling

The Data Linear Address facility is also abbreviated as DataLA. The facility is a replacement or extension of the precise store facility in previous processor generations. The DataLA facility complements the load latency facility by providing a means to profile load and store memory references in the system, leverages the PEBS facility, and provides additional information about sampled loads and stores. Having precise memory reference events with linear address information for both loads and stores provides information to improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

The DataLA facility in the 4th generation processor supports the following events configured to use PEBS:

Table 20-25. Precise Events That Supports Data Linear Address Profiling

Event Name	Event Name
MEM_UOPS_RETIRED.STLB_MISS_LOADS	MEM_UOPS_RETIRED.STLB_MISS_STORES
MEM_UOPS_RETIRED.LOCK_LOADS	MEM_UOPS_RETIRED.SPLIT_STORES
MEM_UOPS_RETIRED.SPLIT_LOADS	MEM_UOPS_RETIRED.ALL_STORES
MEM_UOPS_RETIRED.ALL_LOADS	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM
MEM_LOAD_UOPS_RETIRED.L1_HIT	MEM_LOAD_UOPS_RETIRED.L2_HIT
MEM_LOAD_UOPS_RETIRED.L3_HIT	MEM_LOAD_UOPS_RETIRED.L1_MISS
MEM_LOAD_UOPS_RETIRED.L2_MISS	MEM_LOAD_UOPS_RETIRED.L3_MISS
MEM_LOAD_UOPS_RETIRED.HIT_LFB	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS
MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM
UOPS_RETIRED.ALL (if load or store is tagged)	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE

DataLA can use any one of the IA32_PMC0-IA32_PMC3 counters. Counter overflows will initiate the generation of PEBS records. Upon counter overflow, hardware captures the linear address and possible other status information of the retiring memory uop. This information is then written to the PEBS record that is subsequently generated.

To enable the DataLA facility, software must complete the following steps. Please note that the DataLA facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture DataLA information.

- Complete the PEBS configuration steps.
- Program an event listed in Table 20-25 using any one of IA32_PERFEVTSEL0-IA32_PERFEVTSEL3.
- Set the corresponding IA32_PEBS_ENABLE.PEBS_EN_CTRx bit. This enables the corresponding IA32_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H, and A8H, as shown in Table 20-26.

Table 20-26. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_UOPS_RETIRED.STLB_MISS_STORES, MEM_UOPS_RETIRED.SPLIT_STORES, MEM_UOPS_RETIRED.ALL_STORES ▪ Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 20-25.
Reserved	A8H	Always zero.

20.3.6.3.1 EventingIP Record

The PEBS record layout for processors based on Haswell microarchitecture adds a new field at offset 0B0H. This is the eventingIP field that records the IP address of the retired instruction that triggered the PEBS assist. The EIP/RIP field at offset 08H records the IP address of the next instruction to be executed following the PEBS assist.

20.3.6.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 20.3.4.5. The event codes are listed in Table 20-15. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 20-27.
- Supplier information (bits 30:16): see Table 20-28.
- Snoop response information (bits 37:31): see Table 20-18.

Table 20-27. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell Microarchitecture)

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
COREWB	3	Counts the number of modified cachelines written back.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_L3_DATA_RD	7	Counts the number of data cacheline reads generated by L3 prefetchers.
PF_L3_RFO	8	Counts the number of RFO requests generated by L3 prefetchers.
PF_L3_CODE_RD	9	Counts the number of code reads generated by L3 prefetchers.
SPLIT_LOCK_UC_LOCK	10	Counts the number of lock requests that split across two cachelines or are to UC memory.
STRM_ST	11	Counts the number of streaming store requests electronically.
Reserved	14:12	Reserved

Table 20-27. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell Microarchitecture) (Contd.)

Bit Name	Offset	Description
OTHER	15	Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 20-28. The fields vary across products (according to CPUID signatures) and is noted in the description.

Table 20-28. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_3CH, 06_46H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	LOCAL	22	Local DRAM Controller.
	Reserved	30:23	Reserved

Table 20-29. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_45H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	L4 Cache
	Reserved	30:26	Reserved

20.3.6.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 20-28 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CPUID signature 06_3FH).

Table 20-30. MSR_OFFCORE_RSP_x Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	L3_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

20.3.6.5 Performance Monitoring and Intel® TSX

Chapter 16 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the details of Intel® Transactional Synchronization Extensions (Intel® TSX). This section describes performance monitoring support for Intel TSX.

If a processor supports Intel TSX, the core PMU enhances its IA32_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32_PERFEVTSELx is shown in Figure 20-34. The two additional bit fields are:

- **IN_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32_PERFEVTSEL2.

When the IA32_PERFEVTSELx MSR is programmed with both IN_TX=0 and IN_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN_TXCP bit. See Table 2-29.

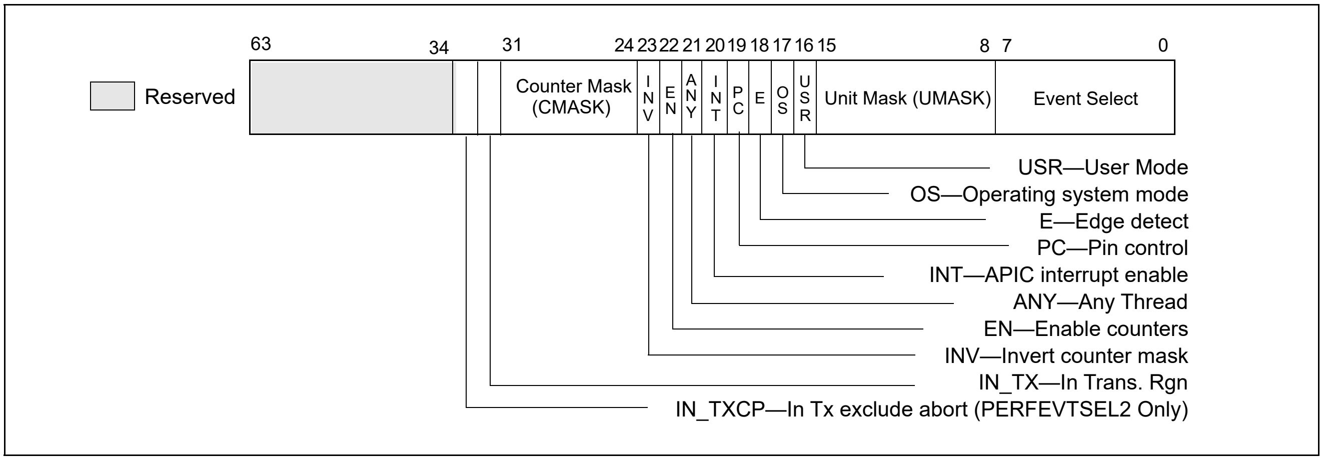


Figure 20-34. Layout of IA32_PERFEVTSELx MSRs Supporting Intel TSX

A common usage of setting IN_TXCP=1 is to capture the number of events that were discarded due to a transactional abort. With IA32_PMC2 configured to count in such a manner, then when a transactional region aborts, the value for that counter is restored to the value it had prior to the aborted transactional region. As a result, any updates performed to the counter during the aborted transactional region are discarded.

On the other hand, setting IN_TX=1 can be used to drill down on the performance characteristics of transactional code regions. When a PMCx is configured with the corresponding IA32_PERFEVTSELx.IN_TX=1, only eventing conditions that occur inside transactional code regions are propagated to the event logic and reflected in the counter result. Eventing conditions specified by IA32_PERFEVTSELx but occurring outside a transactional region are discarded.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they can be found at: https://perfmon-events.intel.com/.

20.3.6.5.1 Intel® TSX and PEBS Support

If a PEBS event would have occurred inside a transactional region, then the transactional region first aborts, and then the PEBS event is processed.

Two of the TSX performance monitoring events also support using the PEBS facility to capture additional information. They are:

- HLE_RETIRED.ABORTED (encoding C8H mask 04H),
- RTM_RETIRED.ABORTED (encoding C9H mask 04H).

A transactional abort (HLE_RETIRED.ABORTED,RTM_RETIRED.ABORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.

Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pended at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 20-31.

Table 20-31. TX Abort Information Field Definition

Bit Name	Offset	Description
Cycles_Last_TX	31:0	The number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
HLE_Abort	32	If set, the abort information corresponds to an aborted HLE execution
RTM_Abort	33	If set, the abort information corresponds to an aborted RTM execution
Instruction_Abort	34	If set, the abort was associated with the instruction corresponding to the eventing IP (offset 0B0H) within the transactional region.
Non_Instruction_Abort	35	If set, the instruction corresponding to the eventing IP may not necessarily be related to the transactional abort.
Retry	36	If set, retrying the transactional execution may have succeeded.
Data_Conflict	37	If set, another logical processor conflicted with a memory address that was part of the transactional region that aborted.
Capacity Writes	38	If set, the transactional region aborted due to exceeding resources for transactional writes.
Capacity Reads	39	If set, the transactional region aborted due to exceeding resources for transactional reads.
In_Suspend	40	Transaction was aborted while in a suspend region. This is an Intel Xeon processor only feature, available beginning with 4th generation Intel Xeon Scalable Processor Family; otherwise reserved.
Reserved	63:41	Reserved

20.3.6.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors

The uncore sub-system in the 4th Generation Intel® Core™ processors provides its own performance monitoring facility. The uncore PMU facility provides dedicated MSRs to select uncore performance monitoring events in a similar manner as those described in Section 20.3.4.6.

The ARB unit and each C-Box provide local pairs of event select MSR and counter register. The layout of the event select MSRs in the C-Boxes are identical as shown in Figure 20-32.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 20-33 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 20-19 summarizes the number MSRs for uncore PMU for each box.

Table 20-32. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-21 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

The uncore performance events for the C-Box and ARB units can be found at: https://perfmon-events.intel.com/.

20.3.6.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in "Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-33.

20.3.7 5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility

The 5th Generation Intel® Core™ processor and the Intel® Core™ M processor families are based on the Broadwell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 20.2.3.

The core PMU has the same capability as those described in Section 20.3.6. IA32_PERF_GLOBAL_STATUS provide a bit indicator (bit 55) for PMI handler to distinguish PMI due to output buffer overflow condition due to accumulating packet data from Intel Processor Trace.

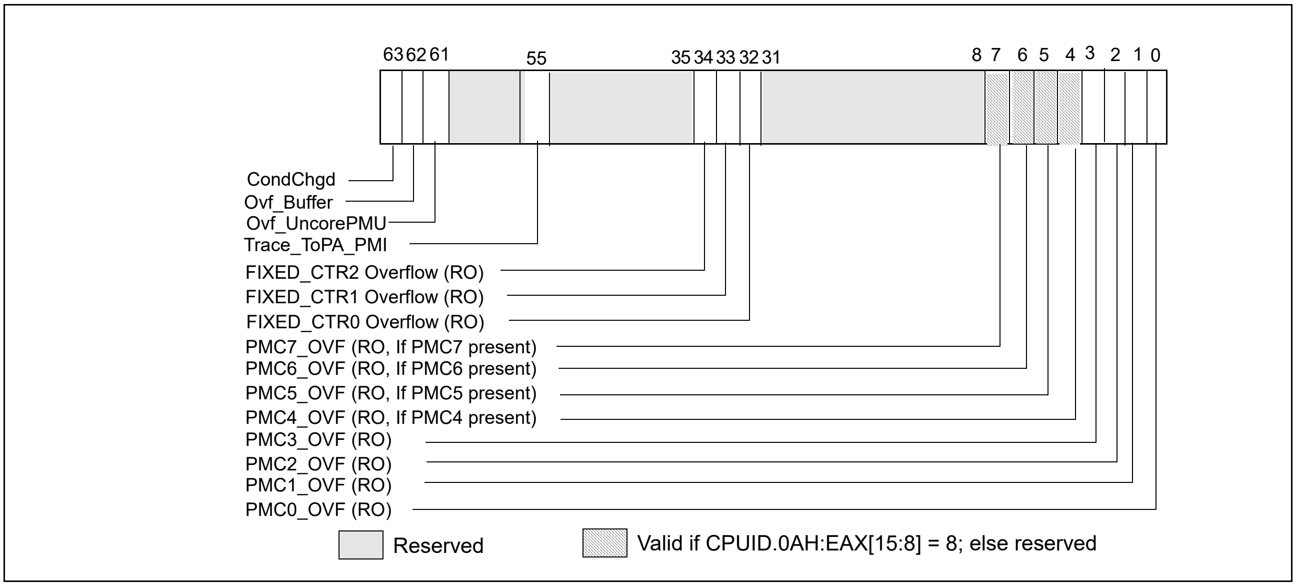


Figure 20-35. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture

Details of Intel Processor Trace is described in Chapter 33, “Intel® Processor Trace.” The IA32_PERF_GLOBAL_OVF_CTRL MSR provides a corresponding reset control bit.

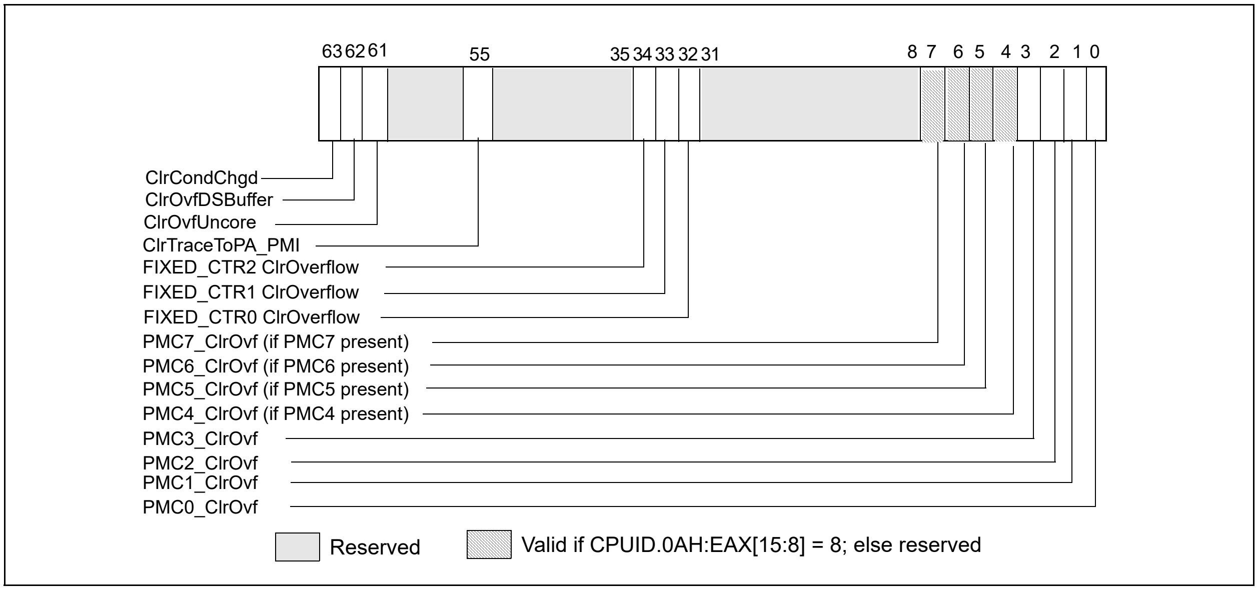


Figure 20-36. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events can be found at: https://perfmon-events.intel.com/.

20.3.8 6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture. For these microarchitectures, the core PMU supports architectural performance monitoring capability with version ID 4 (see Section 20.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 20.2.4.

The core PMU's capability is similar to those described in Section 20.6.3 through Section 20.3.4.5, with some differences and enhancements summarized in Table 20-33. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 20.3.6.5. For details of Intel TSX, see Chapter 16, "Programming with Intel® Transactional Synchronization Extensions," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 40, "Enclave Code Debug and Profiling."

Table 20-33. Core PMU Comparison

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 20.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 20.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 20.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to determine # of counters. See Section 20.2.1.
Architectural Perfmon version	4	3	See Section 20.2.4
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 18.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 20.2.4.

Table 20-33. Core PMU Comparison (Contd.)

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz, ASCI	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow (applicable to Broadwell microarchitecture)	See Section 20.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz ▪ LBR_Frz	NA	See Section 20.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 20.2.4.3.
Precise Events	See Table 20-36.	See Table 20-12.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 20.3.8.2.	No	
LBR Record Format Encoding	000101b	000100b	Section 18.4.8.1
LBR Size	32 entries	16 entries	
LBR Entry	From_IP/To_IP/LBR_Info triplet	From_IP/To_IP pair	Section 18.12
LBR Timing	Yes	No	Section 18.12.1
Call Stack Profiling	Yes, see Section 18.11	Yes, see Section 18.11	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types.	MSR 1A6H and 1A7H; Extended request and response types.	
Intel TSX support for Perfmon	See Section 20.3.6.5.	See Section 20.3.6.5.	

20.3.8.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th generation, 7th generation and 8th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 20-34.

Table 20-34. PEBS Facility Comparison

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 20.3.1.1.1	Section 20.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 20-15	Figure 20-15	
PEBS-EventingIP	Yes	Yes	
PEBS record format encoding	0011b	0010b	
PEBS record layout	Table 20-35; enhanced fields at offsets 98H- B8H; and TSC record field at C0H.	Table 20-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	
Multi-counter PEBS resolution	PEBS record 90H resolves the eventing counter overflow.	PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS.	
Precise Events	See Table 20-36.	See Table 20-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.

Table 20-34. PEBS Facility Comparison (Contd.)

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 20.3.4.4.2.	See Section 20.3.4.4.2.	
Data Address Profiling	Yes	Yes	
FrontEnd event support	FrontEnd_Retried event and MSR_PEBS_FRONTEND.	No	IA32_PMC0-PMC3 only.

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTES

Precise events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

20.3.8.1.1 PEBS Data Format

The PEBS record format for the 6th generation, 7th generation and 8th generation Intel Core processors is reporting with encoding 0011b in IA32_PERF_CAPABILITIES[11:8]. The lay out is shown in Table 20-35. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 20-35. PEBS Record Format for the 6th Generation, 7th Generation, and 8th Generation Intel Core Processor Families

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 20.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 20-24.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 20.3.4.4.2), PDIR (Section 20.3.4.4.4), and data address profiling (Section 20.3.6.3).

In the core PMU of the 6th generation, 7th generation and 8th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th generation and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32_PERF_GLOBAL_STATUS may be useful to resolve the situations when more than one of IA32_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the "applicable counter" field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

20.3.8.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake, Kaby Lake and Coffee Lake microarchitectures is shown in Table 20-36.

Table 20-36. Precise Events for the Skylake, Kaby Lake, and Coffee Lake Microarchitectures

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST[1]	01H
		ALL_CYCLES[2]	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRED	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRED	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRED	C6H	<Programmable[3]>	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED[2]	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H

Table 20-36. Precise Events for the Skylake, Kaby Lake, and Coffee Lake Microarchitectures (Contd.)

Event Name	Event Select	Sub-event	UMask
MEM_LOAD_RETIRED[4]	D1H	L1_HIT	01H
		L2_HIT	02H
		L3_HIT	04H
		L1_MISS	08H
		L2_MISS	10H
		L3_MISS	20H
		HIT_LFB	40H
MEM_LOAD_L3_HIT_RETIRED[2]	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

NOTES:

1. Only available on IA32_PMC1.
2. INST_RETIRED.ALL_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR_PEBS_FRONTEND, see Section 20.3.8.3
4. Instruction with at least one load uop experiencing the condition specified in the UMask.

20.3.8.1.3 Data Address Profiling

The PEBS Data address profiling on the 6th generation, 7th generation and 8th generation Intel Core processors is largely unchanged from the prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H, and A8H, as shown in Table 20-26.

Table 20-37. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES. ▪ Other bits are zero.
Reserved	A8H	Always zero.

20.3.8.2 Frontend Retired Facility

The Skylake Core PMU has been extended to cover common microarchitectural conditions related to the front end pipeline in addition to providing a generic latency mechanism that can locate fetch bubbles without necessarily attributing them to a particular condition. The facility counts the events if the associated instruction reaches retirement (architecturally committed). Additionally, the user may opt to enable the PEBS facility to obtain precise information on the context of the event, e.g., EventingIP.

The supported frontend microarchitectural conditions require the following interfaces:

- The IA32_PERFEVTSELx MSR must select the FRONTEND_RETIRED event, EventSelect = C6H and UMASK = 01H.

- This event employs a new MSR, MSR_PEBS_FRONTEND, to specify the supported frontend event details, see Table 20-38.
- If precise information is desired, program the PEBS_EN_PMCx field of IA32_PEBS_ENABLE MSR as required.

Note the AnyThread field of IA32_PERFEVTSELx is ignored by the processor for the "FRONTEND_RETIRED" event.

The sub-event encodings supported by MSR_PEBS_FRONTEND.EVTSEL is given in Table 20-38.

Table 20-38. FrontEnd_Retired Sub-Event Encodings Supported by MSR_PEBS_FRONTEND.EVTSEL

Sub-Event Name	EVTSEL	Description
ANY_DSB_MISS	1H	Retired Instructions which experienced any decode stream buffer (DSB) miss.
DSB_MISS	11H	Retired Instructions which experienced a DSB miss that caused a fetch starvation cycle.
L1I_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss[1]. Additional requests to the same cache line as an in-flight L1I cache miss will not be counted.
L2_MISS	13H	The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.
ITLB_MISS	14H	The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.
STLB_MISS	15H	The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.
IDQ_READ_BUBBLES	6H	An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles. The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.

NOTES:

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR_PEBS_FRONTEND is given in Table 20-39.

Table 20-39. MSR_PEBS_FRONTEND Layout

Bit Name	Offset	Description
EVTSEL	7:0	Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 20-38.
IDQ_Bubble_Length	19:8	Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event.
IDQ_Bubble_Width	22:20	Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event.
Reserved	63:23	Reserved

The FRONTEND_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

- The event counts only retired (non-speculative) front-end events, i.e., events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.

- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a front-end (miss) event occurs outside instruction boundary (e.g., due to processor handling of architectural event), it may be reported for the next instruction to retire.

20.3.8.3 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 20.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFF-CORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 20-40.
- Supplier information (bits 29:16): see Table 20-41.
- Snoop response information (bits 37:30): see Table 20-42.

Table 20-40. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake, Kaby Lake, and Coffee Lake Microarchitectures)

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	14:3	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and uncacheable accesses.

Table 20-41 lists the supplier information field that applies to 6th generation, 7th generation and 8th generation Intel Core processors. (6th generation Intel Core processor CPUID signatures: 06_4EH and 06_5EH; 7th generation and 8th generation Intel Core processor CPUID signatures: 06_8EH and 06_9EH).

Table 20-41. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9EH)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT	22	L4 Cache (if L4 is present in the processor).
	Reserved	25:23	Reserved
	DRAM	26	Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).

Table 20-42 lists the snoop information field that apply to processors with CPUID signatures 06_4EH, 06_5EH, 06_8EH, 06_9E, and 06_55H.

Table 20-42. MSR_OFFCORE_RSP_x Snoop Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9E, 06_55H)

Subtype	Bit Name	Offset	Description
Snoop Info	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).
	SNOOP_NONE	31	No details on snoop-related information.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores. -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO). -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD). -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S). In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD). -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO). -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	SNOOP_NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.

20.3.8.3.1 Off-core Response Performance Monitoring for the Intel® Xeon® Scalable Processor Family

The following tables list the requestor and supplier information fields that apply to the Intel® Xeon® Scalable Processor Family.

- Transaction request type encoding (bits 15:0): see Table 20-43.
- Supplier information (bits 29:16): see Table 20-44.
- Supplier information (bits 29:16) with support for Intel® Optane™ DC Persistent Memory support: see Table 20-45.
- Snoop response information has not been changed and is the same as in (bits 37:30): see Table 20-42.

Table 20-43. MSR_OFFCORE_RSP_x Request_Type Definition (Intel® Xeon® Scalable Processor Family)

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DEMAND_RFO	1	Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DEMAND_CODE_RD	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	3	Reserved.
PF_L2_DATA_RD	4	Counts the number of prefetch data reads into L2.
PF_L2_RFO	5	Counts the number of RFO Requests generated by the MLC prefetches to L2.
Reserved	6	Reserved.
PF_L3_DATA_RD	7	Counts the number of MLC data read prefetches into L3.
PF_L3_RFO	8	Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	9	Reserved.
PF_L1D_AND_SW	10	Counts data cacheline reads generated by hardware L1 data cache prefetcher or software prefetch requests.
Reserved	14:11	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

Table 20-44 lists the supplier information field that applies to the Intel Xeon Scalable Processor Family (CPUID signature: 06_55H).

Table 20-44. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_55H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	Reserved	25:22	Reserved
	L3_MISS_LOCAL_DRAM	26	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

Table 20-45 lists the supplier information field that applies to the Intel Xeon Scalable Processor Family (CPUID signature: 06_55H, Steppings 0x5H - 0xFH).

Table 20-45. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_55H, Steppings 0x5H - 0xFH)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	LOCAL_PMM	22	Local home requests that were serviced by local PMM.
	REMOTE_HOP0_PMM	23	Hop 0 Remote supplier.
	REMOTE_HOP1_PMM	24	Hop 1 Remote supplier.
	REMOTE_HOP2P_PMM	25	Hop 2 or more Remote supplier.
	L3_MISS_LOCAL_DRAM	26	L3 Miss: Local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

20.3.8.4 Uncore Performance Monitoring Facilities on Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Cannon Lake microarchitecture introduces LLC support of up to six processor cores. To support six processor cores and eight LLC slices, existing MSRs have been rearranged and new CBo MSRs have been added. Uncore performance monitoring software drivers from prior generations of Intel Core processors will need to update the MSR addresses. The new MSRs and updated MSR addresses have been added to the Uncore PMU listing in Section 2.17.2, “MSRs Specific to 8th Generation Intel® Core™ i3 Processors,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

20.3.9 10th Generation Intel® Core™ Processor Performance Monitoring Facility

Some 10th generation Intel® Core™ processors and some 3rd generation Intel® Xeon® Scalable Processor Family are based on Ice Lake microarchitecture. Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture. For these processors, the core PMU supports architectural performance monitoring capability with version Id 5 (see Section 20.2.5) and a host of non-architectural monitoring capabilities.

The core PMU's capability is similar to those described in Section 20.3.1 through Section 20.3.8, with some differences and enhancements summarized in Table 20-46.

Table 20-46. Core PMU Summary of the Ice Lake Microarchitecture

Box	Ice Lake Microarchitecture	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Comment
Architectural Perfmon version	5	4	See Section 20.2.5.
Number of programmable counters per thread	8	4	Use CPUID to determine number of counters. See Section 20.2.1.
PEBS: Basic functionality	Yes	Yes	See Section 20.3.9.1.
PEBS record format encoding	0100b	0011b	See Section 20.6.2.4.2.
Extended PEBS	PEBS is extended to all Fixed and General Purpose counters and to all performance monitoring events.	No	See Section 20.9.1.
Adaptive PEBS	Yes	No	See Section 20.9.2.
Performance Metrics	Yes (4)	No	See Section 20.3.9.3.
PEBS-PDIR	IA32_FIXED0 only (Corresponding counter control MSRs must be enabled.)	IA32_PMC1 only.	

20.3.9.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 10th generation Intel Core processors provides a number of enhancements relative to PEBS in processors based on the Skylake, Kaby Lake, and Coffee Lake microarchitectures. Enhancement of the PEBS facility with Extended PEBS and Adaptive PEBS features is described in detail in Section 20.9.

The 3rd generation Intel Xeon Scalable Family of processors based on the Ice Lake microarchitecture introduce EPT-friendly PEBS. This allows EPT violations and other VM Exits to be taken on PEBS accesses to the DS Area. See Section 20.9.5 for details.

20.3.9.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 20.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFF-CORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-[N1].
- Response type encoding (bits 16-37) of
  - Supplier information: see Table [18-N2].
  - Snoop response information: see Table [18-N3].
- All transactions are tracked at cacheline granularity except some in request type OTHER.

Table 20-47. MSR_OFFCORE_RSP_x Request_Type Definition (Processors Based on Ice Lake Microarchitecture)

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data and page table entry reads.
DEMAND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DEMAND_CODE_RD	2	Counts demand instruction fetches and instruction prefetches targeting the L1 instruction cache.
Reserved	3	Reserved

Table 20-47. MSR_OFFCORE_RSP_x Request_Type Definition (Processors Based on Ice Lake Microarchitecture)

Bit Name	Offset	Description
HWPF_L2_DATA_RD	4	Counts hardware generated data read prefetches targeting the L2 cache.
HWPF_L2_RFO	5	Counts hardware generated prefetches for exclusive ownership (RFO) targeting the L2 cache.
Reserved	6	Reserved
HWPF_L3	9:7 and 13[1]	Counts hardware generated prefetches of any type targeting the L3 cache.
HWPF_L1D_AND_SWPF	10	Counts hardware generated data read prefetches targeting the L1 data cache and the following software prefetches (PREFETCHNTA, PREFETCHT0/1/2).
STREAMING_WR	11	Counts streaming stores.
Reserved	12	Reserved
Reserved	14	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

NOTES:

1. All bits need to be set to 1 to count this type.

Ice Lake microarchitecture has added a new category of Response subtype, called a Combined Response Info. To count a feature in this type, all the bits specified must be set to 1.

A valid response type must be a non-zero value of the following expression:

Any | ['OR' of Combined Response Info Bits | [('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]]

If "ANY" bit[16] is set, other response type bits [17-39] are ignored.

Table 20-48 lists the supplier information field that applies to processors based on Ice Lake microarchitecture.

Table 20-48. MSR_OFFCORE_RSP_x Supplier Info Field Definition (Processors Based on Ice Lake Microarchitecture)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Combined Response Info	DRAM	26, 31, 32[1]	Requests that are satisfied by DRAM.
	NON_DRAM	26, 37[1]	Requests that are satisfied by a NON_DRAM system component. This includes MMIO transactions.
	L3_MISS	22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37[1]	Requests that were not supplied by the L3 Cache. The event includes some currently reserved bits in anticipation of future memory designs.
Supplier Info	L3_HIT	18,19, 20[1]	Requests that hit in L3 cache. Depending on the snoop response the L3 cache may have retrieved the cacheline from another core's cache.
Reserved		17, 21:25, 27:29	Reserved.

NOTES:

1. All bits need to be set to 1 to count this type.

Table 20-49 lists the snoop information field that applies to processors based on Ice Lake microarchitecture.

Table 20-49. MSR_OFFCORE_RSP_x Snoop Info Field Definition (Processors Based on Ice Lake Microarchitecture)

Subtype	Bit Name	Offset	Description
Snoop Info	Reserved	30	Reserved.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	A snoop was sent and none of the snooped caches contained the cacheline.
	SNOOP_HIT_NO_FWD	34	A snoop was sent and hit in at least one snooped cache. The unmodified cacheline was not forwarded back, because the L3 already has a valid copy.
	Reserved	35	Reserved.
	SNOOP_HITM	36	A snoop was sent and the cacheline was found modified in another core's caches. The modified cacheline was forwarded to the requesting core.

20.3.9.3 Performance Metrics

The Ice Lake core PMU provides built-in support for Top-down Microarchitecture Analysis (TMA) method level 1 metrics. These metrics are always available to cross-validate performance observations, freeing general purpose counters to count other events in high counter utilization scenarios. For more details about the method, refer to Top-Down Analysis Method chapter (Appendix B.1) of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

A new MSR called MSR_PERF_METRICS reports the metrics directly. Software can check (and/or expose to its guests) the availability of the PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15). For additional details on this MSR, refer to Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

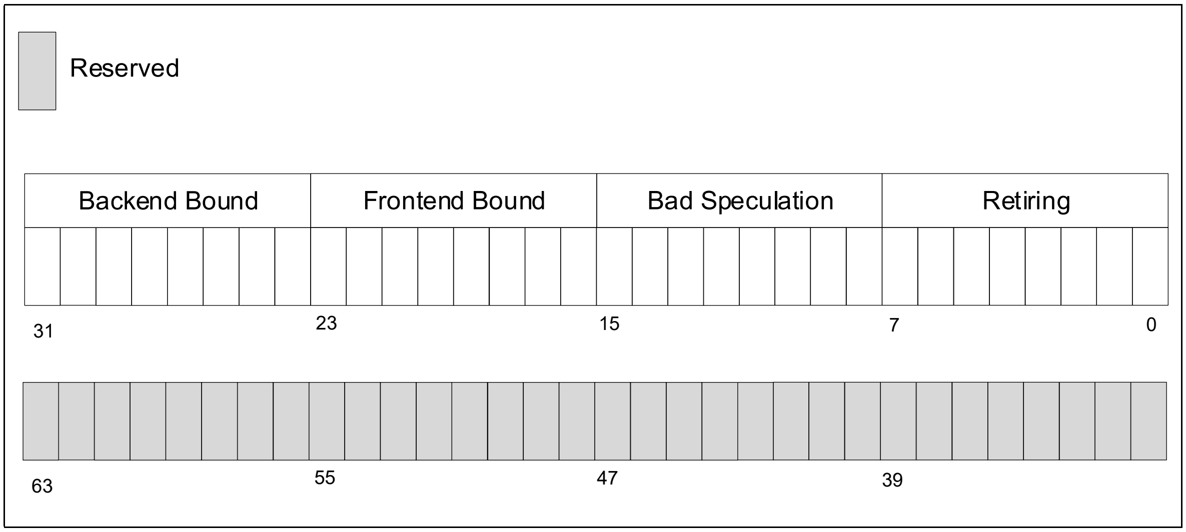


Figure 20-37. MSR_PERF_METRICS Definition

This register exposes the four TMA Level 1 metrics. The lower 32 bits are divided into four 8-bit fields, as shown by the above figure, each of which is an integer fraction of 255.

To support built-in performance metrics, new bits have been added to the following MSRs:

- IA32_PERF_GLOBAL_CTRL. EN_PERF_METRICS[48]: If this bit is set and fixed-function performance-monitoring counter 3 is enabled, built-in performance metrics are enabled.
- IA32_PERF_GLOBAL_STATUS_SET. SET_OVF_PERF_METRICS[48]: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.
- IA32_PERF_GLOBAL_STATUS_RESET. RESET_OVF_PERF_METRICS[48]: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.
- IA32_PERF_GLOBAL_STATUS. OVF_PERF_METRICS[48]: If this bit is set, it indicates that a PERF_METRICS-related resource has overflowed and a PMI is triggered[1]. If this bit is clear, no such overflow has occurred.

NOTE

Software has to synchronize, e.g., re-start, fixed-function performance-monitoring counter 3 as well as PERF_METRICS when either bit 35 or 48 in IA32_PERF_GLOBAL_STATUS is set. Otherwise, PERF_METRICS may return undefined values.

The values in MSR_PERF_METRICS are derived from fixed-function performance-monitoring counter 3. Software should start both registers, PERF_METRICS and fixed-function performance-monitoring counter 3, from zero. Additionally, software is recommended to periodically clear both registers in order to maintain accurate measurements for certain scenarios that involve sampling metrics at high rates.

In order to save/restore PERF_METRICS, software should follow these guidelines:

- PERF_METRICS and fixed-function performance-monitoring counter 3 should be saved and restored together.
- To ensure that PERF_METRICS and fixed-function performance-monitoring counter 3 remain synchronized, both should be disabled during both save and restore. Software should enable/disable them atomically, with a single write to IA32_PERF_GLOBAL_CTRL to set/clear both EN_PERF_METRICS[bit 48] and EN_FIXED_CTR3[bit 35].
- On state restore, fixed-function performance-monitoring counter 3 must be restored **before** PERF_METRICS, otherwise undefined results may be observed.

20.3.10 12th and 13th Generation Intel® Core™ Processors, and 4th Generation Intel® Xeon® Scalable Processor Family Performance Monitoring Facility

The 12th generation Intel® Core™ processor supports Alder Lake performance hybrid architecture. These processors offer a unique combination of Performance and Efficient-cores (P-core and E-core). The P-core is based on Golden Cove microarchitecture and the E-core is based on Gracemont microarchitecture. The 13th generation Intel® Core™ processor supports Raptor Lake performance hybrid architecture, utilizing both Raptor Cove cores and enhanced Gracemont cores. The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture utilizing Golden Cove cores. These processors all report architectural performance monitoring version ID = 5 and support non-architectural monitoring capabilities described in this section.

20.3.10.1 P-core Performance Monitoring Unit

The P-core PMU's capability is similar to those described in Section 20.3.1 through Section 20.3.9, with some differences and enhancements summarized in Table 20-50.

[1] An overflow of fixed-function performance-monitoring counter 3 should normally happen first if software follows Intel's recommendations.

Table 20-50. Core PMU Summary of the Golden Cove Microarchitecture

Box	Golden Cove Microarchitecture	Ice Lake Microarchitecture	Comment
Architectural Perfmon version	5	5	See Section 20.2.5.
Event-Counter Restrictions	Simplified identification		Counters 4-7 support a subset of events. See Section 20.3.10.1.2.
Performance Metrics	Yes (12)	Yes (4)	See Section 20.3.9.3.
PEBS: Baseline, record format	Yes 0100b	Yes 0100b	See Section 20.3.9.
PEBS: EPT-friendly	Yes	No; debuts in Ice Lake server microarchitecture	See Section 20.6.2.4.2.
PEBS: Precise Distribution	IA32_FIXED0 instruction-granularity PDist on IA32_PMC0	IA32_FIXED0 cycle-granularity No PDist	See Section 20.9.6.
PEBS: Load Latency	Instruction latency Cache latency Access info fields (5)	Instruction latency Access info fields (3)	See Section 20.9.7.
PEBS: Store Latency	Cache latency Access info fields (3)	None	See Section 20.9.8.
PEBS: Intel TSX support	Abort info fields (9)	Abort info fields (8)	See Section 20.3.6.5.1. (Intel Xeon processor only feature.)

20.3.10.1.1 P-core Perf Metrics Extensions

For 12th generation Intel Core processor P-cores, the core PMU supports the built-in metrics that were introduced in the Ice Lake microarchitecture PMU. This core PMU extends the PERF_METRICS MSR to feature TMA method level 2 metrics, as shown in Figure 20-38.

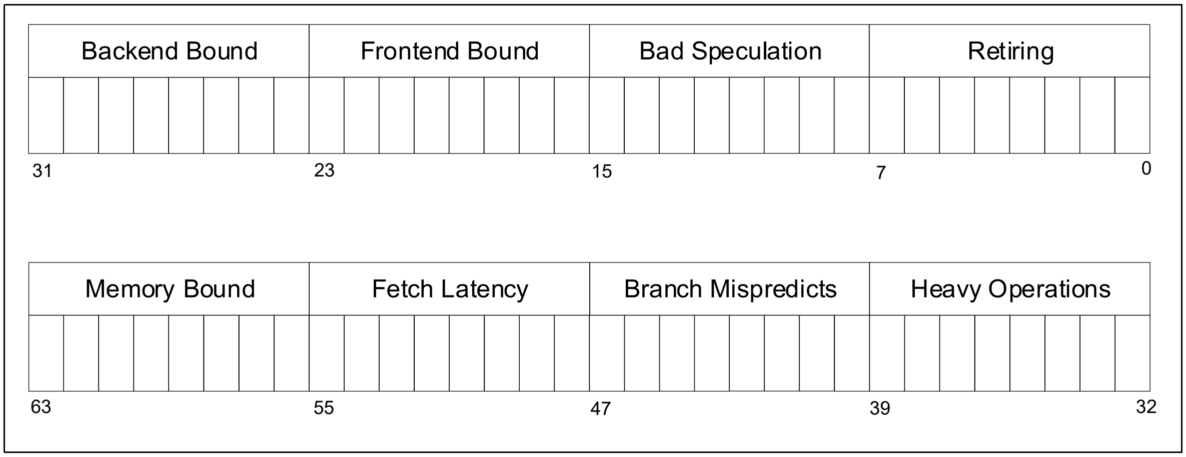


Figure 20-38. PERF_METRICS MSR Definition for 12th Generation Intel® Core™ Processor P-core

The lower half of the register is the TMA level 1 metrics (legacy). The upper half is also divided into four 8-bit fields, each of which is an integer fraction of 255. Additionally, each of the new level 2 metrics in the upper half is a subset of the corresponding level 1 metric in the lower half (that is, its parent node per the TMA hierarchy). This enables software to deduce the other four level 2 metrics by subtracting corresponding metrics as shown in Figure 20-39.

Light_Operations = Retiring - Heavy_Operations
Machine_Clears = Bad_Speculation - Branch_Mispredicts
Fetch_Bandwidth = Frontend_Bound - Fetch_Latency
Core_Bound = Backend_Bound - Memory_Bound

Figure 20-39. Deducing Implied Level 2 Metrics in the Core PMU for12th Generation Intel® Core™ Processor P-core

The PERF_METRICS MSR and fixed-function performance-monitoring counter 3 of the core PMU feature 12 metrics in total that cover all level 1 and level 2 nodes of the TMA hierarchy.

20.3.10.1.2 P-core Counter Restrictions Simplification

The 12th generation Intel Core processor P-core allows identification of performance monitoring events with counter restrictions based on event encodings. The general rule is: Event Codes < 0x90 are restricted to general-purpose performance-monitoring counters 0-3. Event Codes ≥ 0x90 are likely to have no restrictions. Table 20-51 lists the exceptions to this rule.

Table 20-51. Special Performance Monitoring Events with Counter Restrictions

Event Encoding[1]	Event Name	Counter Restriction
xx3C	CPU_CLK_UNHALTED.*	0-7 (No restriction for all architectural events.)
xx2E	LONGEST_LAT_CACHE.*	
xxDx	MEM_*_RETIRED.*	0-3
01A3, 02A3, 08A3	Some CYCLE_ACTIVITY sub-events	0-3
02CD	MEM_TRANS_RETIRED.STORE_SAMPLE	0
04A4	TOPDOWN.BAD_SPEC_SLOTS	0
08A4	TOPDOWN.BR_MISPREDICT_SLOTS	
xxCE	AMX_OPS_RETIRED	0

NOTES:

1. Linux perf rUUEE syntax, where UU is the Unit Mask field and EE is the Event Select (also known as Event Code) field in the IA32_PERFEVTSELx MSRs.

20.3.10.1.3 P-core Off-core Response Facility

For the 12th generation Intel Core processor P-core, the Off-core Response (OCR) Facility is similar to that described in Section 20.3.9.2.

The following enhancements are introduced for the Request_Type of MSR_OFFCORE_RSP_x:

- WB (bits 3 and 12): Count writeback (modified or non-modified) transactions by core caches.
- HWPF_L1D (bit 10): Counts hardware generated data read prefetches targeting the L1 data cache (only).
- SWPF_READ (bit 14): Counts software generated data read prefetches by the PREFETCHNTA and PREFETCHT0/1/2 instructions.

20.3.10.2 E-core Performance Monitoring Unit

The core PMU capabilities on the 12th generation Intel Core processor E-core are summarized in Table 20-52 below.

Table 20-52. Core PMU Summary of the Gracemont Microarchitecture

Box	Gracemont Microarchitecture	Tremont Microarchitecture	Comment
Number of fixed-function performance-monitoring counters per core	3	3	Use CPUID to enumerate number of counters. See Section 20.2.1.
Number of general-purpose counters per core	6	4	Use CPUID to enumerate number of counters. See Section 20.2.1.
Architectural Performance Monitoring version ID	5	5	See Section 20.2.5.
PEBS record format encoding	0100b	0100b	See Section 20.5.5.
EPT-friendly PEBS support	Yes	No	See Section 20.9.5.
Extended PEBS	Yes	Yes	See Section 20.9.1.
Adaptive PEBS	Yes	Yes	See Section 20.9.2.
Precise distribution (PDist) PEBS	IA32_PMC0 and IA32_FIXED_CTR0	IA32_PMC0 and IA32_FIXED_CTR0	PDist eliminates skid, see Section 20.9.3, Section 20.9.4, and Section 20.9.6.
PEBS Latency	Load and Store Latency	No	See Section 20.3.10.2.1, Section 20.3.10.2.2, Section 20.9.7, and Section 20.9.8.
PEBS Output	DS Save Area or Intel® Processor Trace	DS Save Area or Intel® Processor Trace	See Section 20.5.5.2.1.
Offcore Response	MSR 01A6H and 01A7H, each core has its own register, extended request and response types.	MSR 1A6H and 1A7H, each core has its own register, extended request and response types.	See Section 20.5.5.4.

20.3.10.2.1 E-core PEBS Load Latency

The 12th generation Intel Core processor E-core includes PEBS Load Latency support similar to that described in Section 20.9.7.

When a programmable counter is configured to count MEM_UOPS_RETIRED.LOAD_LATENCY_ABOVE_THRESHOLD (IA32_PERFEVTSELx[15:0] = 0xD005, with CMASK=0 and INV=0), selected load operations whose latency exceeds the threshold provided in MSR_PEBS_LD_LAT_THRESHOLD (MSR 03F6H) will be counted. If a PEBS record is generated on overflow of this counter, the Memory Access Latency and Memory Auxiliary Info data is reported in the Memory Access Info group (Section 20.9.2.2.2). The formats of these fields are shown in Table 20-53 and Table 20-94.

Table 20-53. E-core PEBS Memory Access Info Encoding

Bit(s)	Field	Description
3:0	Data Source	The source of the data; see Table 20-54.
4	Lock	0: The operation was not part of a locked transaction. 1: The operation was part of a locked transaction.

Table 20-53. E-core PEBS Memory Access Info Encoding (Contd.)

Bit(s)	Field	Description
5	STLB_MISS	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
6	ST_FWD_BLK	0: Load did not get a store forward block. 1: Load got a store forward block.
63:7	Reserved	Reserved

For details on E-core PEBS memory access latency encoding, see the Access Latency Field in Table 20-94.

Table 20-54. E-core PEBS Data Source Encodings

Encoding	Description
00H	Unknown Data Source (the processor could not retrieve the origin of this request) and MMIO. Memory mapped I/O hit.
01H	L1 HIT. This request was satisfied by the L1 data cache. (Minimal latency core cache hit.)
02H	FB HIT. Outstanding core cache miss to same cache-line address was already underway. (Pending core cache hit.)
03H	L2 HIT. This request was satisfied by the L2 cache.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HITE. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found (clean).
06H	L3 HITM. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where a modified copy was found.
07H	Reserved.
08H	L3 HITF. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where a shared or forwarding copy was found.
09H	Reserved.
0AH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).
0BH	Reserved.
0CH	Reserved.
0DH	Reserved.
0EH	I/O. Request of input/output operation.
0FH	The request was to un-cacheable memory.

20.3.10.2.2 E-core PEBS Store Latency

The 12th generation Intel Core processor E-core includes PEBS Store Latency support. When a programmable counter is configured to count MEM_UOPS_RETIRED.STORE_LATENCY (IA32_PERFEVTSELx[15:0] = 0xD006, with CMASK=0 and INV=0), all store operations will be counted. If a PEBS record is generated on overflow of this counter, the Memory Access Latency and Memory Auxiliary Info data is reported in the Memory Access Info group (Section 18.9.2.2.2). The formats of these fields are shown in Table 20-53 and Table 20-94.

20.3.10.2.3 E-core Precise Distribution (PDist) Support

The 12th generation Intel Core processor E-core supports PEBS with Precise Distribution (PDist) on IA32_PMC0 and IA32_FIXED_CTR0. All precise events support PDist save for UOPS_RETIRED. See Section 20.9.6 for additional details on PDist.

20.3.10.2.4 E-core Enhanced Off-core Response

Event number 0B7H support off-core response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. There are unique pairs of MSR_OFFCORE_RSPx registers per core. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as follows:

- Bits 15:0 and bits 49:44 specify the request type of a transaction request to the uncore. This is described in Table 20-55.
- Bits 30:16 specify Response Type information or an L2 Hit, and is described in Table 20-75.
- If L2 misses, then bits 37:31 can be used to specify snoop response information and is described in Table 20-76.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 20.5.2.3 for details.

Table 20-55. MSR_OFFCORE_RSPx Request Type Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data reads.
DEMAND_RFO	1	Counts all demand reads for ownership (RFO) requests and software based prefteches for exclusive ownership (prefetchw).
DEMAND_CODE_RD	2	Counts demand instruction fetches and L1 instruction cache prefetches.
COREWB_M	3	Counts modified write backs from L1 and L2.
HWPF_L2_DATA_RD	4	Counts prefetch (that bring data to L2) data reads.
HWPF_L2_RFO	5	Counts all prefetch (that bring data to L2) RFOs.
HWPF_L2_CODE_RD	6	Counts all prefetch (that bring data to MLC only) code reads.
HWPF_L3_DATA_RD	7	Counts L3 cache hardware prefetch data reads (written to the L3 cache only).
HWPF_L3_RFO	8	Counts L3 cache hardware prefetch RFOs (written to the L3 cache only) .
HWPF_L3_CODE_RD	9	Counts L3 cache hardware prefetch code reads (written to the L3 cache only).
HWPF_L1D_AND_SWPF	10	Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).
STREAMING_WR	11	Counts all streaming stores.
COREWB_NONM	12	Counts non-modified write backs from L2.
RSVD	14:13	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O accesses that have any response type.
UC_RD	44	Counts uncached memory reads (PRd, UCRdF).
UC_WR	45	Counts uncached memory writes (WiL).
PARTIAL_STREAMING_WR	46	Counts partial (less than 64 byte) streaming stores (WCiL).
FULL_STREAMING_WR	47	Counts full, 64 byte streaming stores (WCiLF).
L1WB_M	48	Counts modified WriteBacks from L1 that miss the L2.
L2WB_M	49	Counts modified WriteBacks from L2.

20.3.10.3 Unhalted Reference Cycles

The Unhalted Reference Cycles architectural performance monitoring event is enhanced to count at TSC-rate in the 12th generation Intel Core processor P-core when used on a general-purpose PMC. This enhancement makes it consistent with the fixed-function counter 2 and the E-core. As a result, this event is kept enumerated in CPUID leaf 0AH.EBX (unlike prior hybrid parts).

20.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS)

NOTE

This section also applies to the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture.

20.4.1 Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.

A complete description of the tile and untile PMU programming interfaces for Intel Xeon Phi processors based on the Knights Landing microarchitecture can be found in the Technical Document section at http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

A tile contains a pair of cores attached to a shared L2 cache and is similar to those found in Intel Atom® processors based on the Silvermont microarchitecture. The processor provides several new capabilities on top of the Silvermont performance monitoring facilities.

The processor supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural performance monitoring capabilities. The processor provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in the processor also uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3. The processor supports AnyThread counting in three architectural performance monitoring events.

20.4.1.1 Enhancements of Performance Monitoring in the Intel® Xeon Phi™ Processor Tile

The Intel® Xeon Phi™ processor tile includes the following enhancements to the Silvermont microarchitecture.

- AnyThread support. This facility is limited to following three architectural events: Instructions Retired, Unhalted Core Cycles, Unhalted Reference Cycles using IA32_FIXED_CTR0-2 and Unhalted Core Cycles, Unhalted Reference Cycles using IA32_PERFEVTSELx.
- PEBS-DLA (Processor Event-Based Sampling-Data Linear Address) fields. The processor provides memory address in addition to the Silvermont PEBS record support on select events. The PEBS recording format as reported by IA32_PERF_CAPABILITIES [11:8] is 2.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor tile to subsystems outside the tile (untile). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx. Two cores do not share the off-core response MSRs. Knights Landing expands off-core response capability to match the processor untile changes.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests. This facility is updated to match the processor untile changes.

20.4.1.1.1 Processor Event-Based Sampling

The processor supports processor event based sampling (PEBS). PEBS is supported using IA32_PMC0 (see also Section 18.4.9, “BTS and DS Save Area”).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 20.6.2.4).

The list of PEBS events supported in the processor is shown in the following table.

Table 20-56. PEBS Performance Events for Knights Landing Microarchitecture

Event Name	Event Select	Sub-event	UMask	Data Linear Address Support
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		CALL	F9H	No
		REL_CALL	FDH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		FAR_BRANCH	BFH	No
		RETURN	F7H	No
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		RETURN	F7H	No
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H	Yes
		L2_MISS_LOADS	04H	Yes
		DLTB_MISS_LOADS	08H	Yes
RECYCLEQ	03H	LD_BLOCK_ST_FORWARD	01H	Yes
		LD_SPLITS	08H	Yes

The PEBS record format 2 supported by processors based on the Knights Landing microarchitecture is shown in Table 20-57, and each field in the PEBS record is 64 bits long.

Table 20-57. PEBS Record Format for Knights Landing Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	PSDLA
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP

Table 20-57. PEBS Record Format for Knights Landing Microarchitecture (Contd.)

Byte Offset	Field	Byte Offset	Field
58H	R9	B8H	Reserved

20.4.1.1.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 20-58 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 20-58. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

Some of the MSR_OFFCORE_RESP [0,1] register bits are not valid in this processor and their use is reserved. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 registers are defined in Table 20-59. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 20.5.2.3 for details.

Table 20-59. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers

Main	Sub-field	Bit	Name	Description
Request Type		0	DEMAND_DATA_RD	Demand cacheable data and L1 prefetch data reads.
		1	DEMAND_RFO	Demand cacheable data writes.
		2	DEMAND_CODE_RD	Demand code reads and prefetch code reads.
		3	Reserved	Reserved.
		4	Reserved	Reserved.
		5	PF_L2_RFO	L2 data RFO prefetches (includes PREFETCHW instruction).
		6	PF_L2_CODE_RD	L2 code HW prefetches.
		7	PARTIAL_READS	Partial reads (UC or WC).
		8	PARTIAL_WRITES	Partial writes (UC or WT or WP). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		9	UC_CODE_READS	UC code reads.
		10	BUS_LOCKS	Bus locks and split lock requests.
		11	FULL_STREAMING_STORES	Full streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		12	SW_PREFETCH	Software prefetches.
		13	PF_L1_DATA_RD	L1 data HW prefetches.
		14	PARTIAL_STREAMING_STORES	Partial streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		15	ANY_REQUEST	Account for any requests.

Table 20-59. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers (Contd.)

Main	Sub-field	Bit	Name	Description
Response Type	Any	16	ANY_RESPONSE	Account for any response.
	Data Supply from Untile	17	NO_SUPP	No Supplier Details.
		18	Reserved	Reserved.
		19	L2_HIT_OTHER_TILE_NEAR	Other tile L2 hit E Near.
		20	Reserved	Reserved.
		21	MCDRAM_NEAR	MCDRAM Local.
		22	MCDRAM_FAR_OR_L2_HIT_OTHER_TILE_FAR	MCDRAM Far or Other tile L2 hit far.
		23	DRAM_NEAR	DRAM Local.
		24	DRAM_FAR	DRAM Far.
	Data Supply from within same tile	25	L2_HITM_THIS_TILE	M-state.
		26	L2_HITE_THIS_TILE	E-state.
		27	L2_HITS_THIS_TILE	S-state.
		28	L2_HITF_THIS_TILE	F-state.
		29	Reserved	Reserved.
		30	Reserved	Reserved.
	Snoop Info; Only Valid in case of Data Supply from Untile	31	SNOOP_NONE	None of the cores were snooped.
		32	NO_SNOOP_NEEDED	No snoop was needed to satisfy the request.
		33	Reserved	Reserved.
		34	Reserved	Reserved.
		35	HIT_OTHER_TILE_FWD	Snoop request hit in the other tile with data forwarded.
		36	HITM_OTHER_TILE	A snoop was needed and it HitM-ed in other core's L1 cache. HitM denotes a cache-line was in modified state before effect as a result of snoop.
		37	NON_DRAM	Target was non-DRAM system address. This includes MMIO transactions.
Outstanding requests	Weighted cycles	38	OUTSTANDING (Valid only for MSR_OFFCORE_RESP0. Should only be used on PMC0. This bit is reserved for MSR_OFFCORE_RESP1).	If set, counts total number of weighted cycles of any outstanding offcore requests with data response. Valid only for OFFCORE_RESP_0 event. Should only be used on PMC0. This bit is reserved for OFFCORE_RESP_1 event.

20.4.1.1.3 Average Offcore Request Latency Measurement

Measurement of average latency of offcore transaction requests can be enabled using MSR_OFFCORE_RSP0.[bit 38] with the choice of request type specified in MSR_OFFCORE_RSP0.[bit 15:0].

Refer to Section 20.5.2.3, "Average Offcore Request Latency Measurement," for typical usage. Note that MSR_OFFCORE_RESPx registers are not shared between cores in Knights Landing. This allows one core to measure average latency while other core is measuring different offcore response events.

20.5 PERFORMANCE MONITORING (INTEL ATOM® PROCESSORS)

20.5.1 Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 20.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

NOTE

> The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 20.2.1.

Non-architectural performance monitoring in Intel Atom processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

Architectural and non-architectural performance monitoring events in 45 nm and 32 nm Intel Atom processors support thread qualification using bit 21 (AnyThread) of IA32_PERFEVTSELx MSR, i.e., if IA32_PERFEVTSELx.AnyThread =1, event counts include monitored conditions due to either logical processors in the same processor core.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3.

Valid event mask (Umask) bits can be found at: https://perfmon-events.intel.com/. The UMASK field may contain sub-fields that provide the same qualifying actions like those listed in Table 20-77, Table 20-78, Table 20-79, and Table 20-80. One or more of these sub-fields may apply to specific events on an event-by-event basis. Precise Event Based Monitoring is supported using IA32_PMC0 (see also Section 18.4.9, “BTS and DS Save Area”).

20.5.2 Performance Monitoring for Silvermont Microarchitecture

Intel processors based on the Silvermont microarchitecture report architectural performance monitoring versionID = 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities. Intel processors based on the Silvermont microarchitecture provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2). Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Non-architectural performance monitoring in the Silvermont microarchitecture uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32_PERFEVTSELx MSR.

20.5.2.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- The width of counter reported by CPUID.0AH:EAX[23:16] is 40 bits.

- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

20.5.2.1.1 Processor Event Based Sampling (PEBS)

In the Silvermont microarchitecture, the PEBS facility can be used with precise events. PEBS is supported using IA32_PMC0 (see also Section 18.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 20.6.2.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 20-60.

Table 20-60. PEBS Performance Events for the Silvermont Microarchitecture

Event Name	Event Select	Sub-event	UMask
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H
		L2_MISS_LOADS	04H
		DLTB_MISS_LOADS	08H
		HITM	20H
REHABQ	03H	LD_BLOCK_ST_FORWARD	01H
		LD_SPLITS	08H

PEBS Record Format The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 20-61, and each field in the PEBS record is 64 bits long.

Table 20-61. PEBS Record Format for the Silvermont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Reserved
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

20.5.2.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 20-62 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

In the Silvermont microarchitecture, each MSR_OFFCORE_RSPx is shared by two processor cores.

Table 20-62. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are shown in Figure 20-40 and Figure 20-41. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 20.5.2.3 for details.

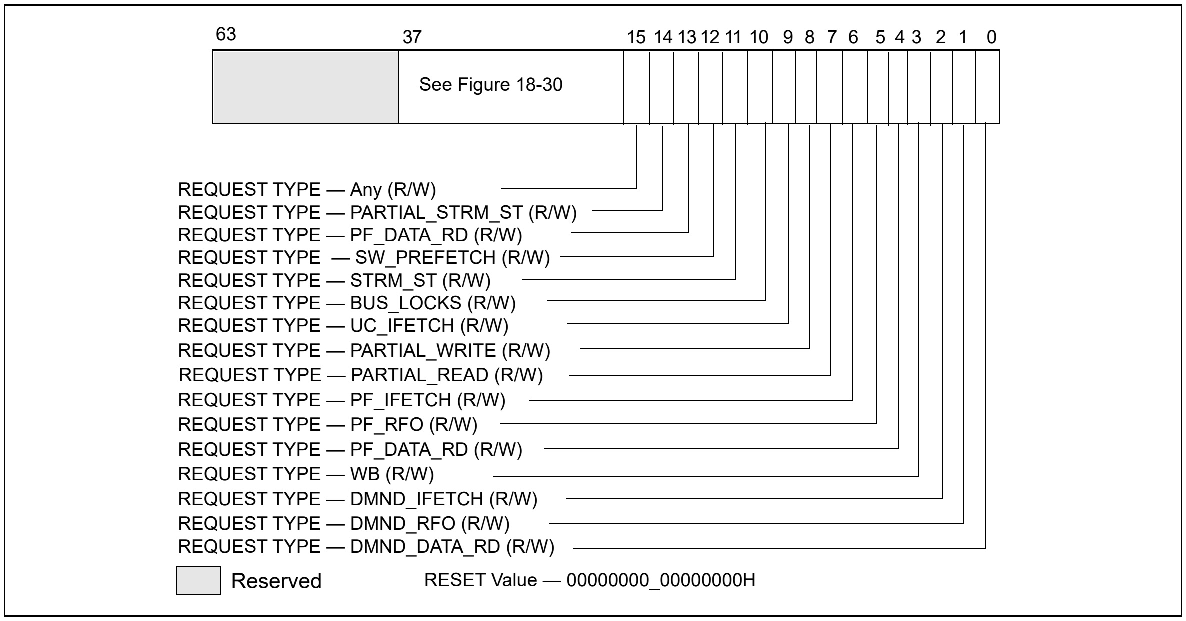

63
37
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
See Figure 18-30
REQUEST TYPE — Any (R/W)
REQUEST TYPE — PARTIAL_STRM_ST (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — SW_PREFETCH (R/W)
REQUEST TYPE — STRM_ST (R/W)
REQUEST TYPE — BUS_LOCKS (R/W)
REQUEST TYPE — UC_IFETCH (R/W)
REQUEST TYPE — PARTIAL_WRITE (R/W)
REQUEST TYPE — PARTIAL_READ (R/W)
REQUEST TYPE — PF_IFETCH (R/W)
REQUEST TYPE — PF_RFO (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — WB (R/W)
REQUEST TYPE — DMND_IFETCH (R/W)
REQUEST TYPE — DMND_RFO (R/W)
REQUEST TYPE — DMND_DATA_RD (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 20-40. Request_Type Fields for MSR_OFFCORE_RSPx

Table 20-63. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT, and WP).
UC_IFETCH	9	Counts the number of UC instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests.
STRM_ST	11	Streaming store requests.
SW_PREFETCH	12	Counts software prefetch requests.
PF_DATA_RD	13	Counts DCU hardware prefetcher data read requests.
PARTIAL_STRM_ST	14	Streaming store requests.
ANY	15	Any request that crosses IDI, including I/O.

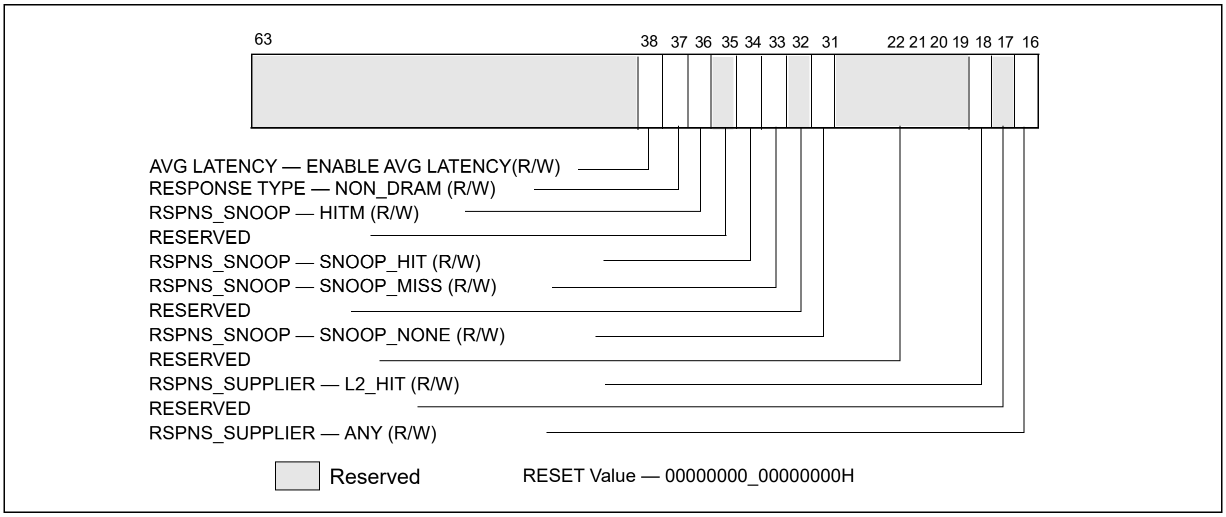

63
38
37
36
35
34
33
32
31
22
21
20
19
18
17
16
AVG LATENCY — ENABLE AVG LATENCY(R/W)
RESPONSE TYPE — NON_DRAM (R/W)
RSPNS_SNOOP — HITM (R/W)
RESERVED
RSPNS_SNOOP — SNOOP_HIT (R/W)
RSPNS_SNOOP — SNOOP_MISS (R/W)
RESERVED
RSPNS_SNOOP — SNOOP_NONE (R/W)
RESERVED
RSPNS_SUPPLIER — L2_HIT (R/W)
RESERVED
RSPNS_SUPPLIER — ANY (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 20-41. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSPx

To properly program this extra register, software must set at least one request type bit (Table 20-63) and a valid response type pattern (Table 20-64, Table 20-65). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 20-64. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	Cache reference hit L2 in either M/E/S states.
	Reserved	30:19	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY” bit is set, the supplier and snoop info bits are ignored.

Table 20-65. MSR_OFFCORE_RSPx Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	Reserved	32	Reserved
	SNOOP_MISS	33	Counts the number of snoop misses when L2 misses.
	SNOOP_HIT	34	Counts the number of snoops hit in the other module where no modified copies were found.
	Reserved	35	Reserved

Table 20-65. MSR_OFFCORE_RSPx Snoop Info Field Definition (Contd.)

Subtype	Bit Name	Offset	Description
	HITM	36	Counts the number of snoops hit in the other module where modified copies were found in other core's L1 cache.
	NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	Enable average latency measurement by counting weighted cycles of outstanding offcore requests of the request type specified in bits 15:0 and any response (bits 37:16 cleared to 0). This bit is available in MSR_OFFCORE_RESP0. The weighted cycles is accumulated in the specified programmable counter IA32_PMCx and the occurrence of specified requests are counted in the other programmable counter.

20.5.2.3 Average Offcore Request Latency Measurement

Average latency for offcore transactions can be determined by using both MSR_OFFCORE_RSP registers. Using two performance monitoring counters, program the two OFFCORE_RESPONSE event encodings into the corresponding IA32_PERFEVTSELx MSRs. Count the weighted cycles via MSR_OFFCORE_RSP0 by programming a request type in MSR_OFFCORE_RSP0.[15:0] and setting MSR_OFFCORE_RSP0.OUTSTANDING[38] to 1, white setting the remaining bits to 0. Count the number of requests via MSR_OFFCORE_RSP1 by programming the same request type from MSR_OFFCORE_RSP0 into MSR_OFFCORE_RSP1[bit 15:0], and setting MSR_OFFCORE_RSP1.ANY_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be obtained by dividing the value of the IA32_PMCx register that counted weight cycles by the register that counted requests.

20.5.3 Performance Monitoring for Goldmont Microarchitecture

Intel Atom processors based on the Goldmont microarchitecture report architectural performance monitoring versionID = 4 (see Section 20.2.4) and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 20.2.4.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3. The Goldmont microarchitecture does not support Hyper-Threading and thus architectural and non-architectural performance monitoring events ignore the AnyThread qualification regardless of its setting in the IA32_PERFEVTSELx MSR. However, Goldmont does not set the AnyThread deprecation bit (CPUID.0AH:EDX[15]).

The core PMU's capability is similar to that of the Silvermont microarchitecture described in Section 20.5.2, with some differences and enhancements summarized in Table 20-66.

Table 20-66. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	Goldmont Microarchitecture	Silvermont Microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 20.2.1.
# of general-purpose counters per core	4	2	Use CPUID to determine # of counters. See Section 20.2.1.
Counter width (R,W)	R:48, W: 32/48	R:40, W:32	See Section 20.2.2.
Architectural Performance Monitoring version ID	4	3	Use CPUID to determine # of counters. See Section 20.2.1.

Table 20-66. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	Goldmont Microarchitecture	Silvermont Microarchitecture	Comment
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics for branch profiling.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.	See Section 18.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 20.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz	▪ Individual counter overflow ▪ PEBS buffer overflow	See Section 20.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz, ▪ LBR_Frz	No	See Section 20.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	No	See Section 20.2.4.3.
Processor Event Based Sampling (PEBS) Events	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 20-67.	See Section 20.5.2.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 20-60).	IA32_PMC0 only.
PEBS record format encoding	0011b	0010b	
Reduce skid PEBS	IA32_PMC0 only	No	
Data Address Profiling	Yes	No	
PEBS record layout	Table 20-68; enhanced fields at offsets 90H- 98H; and TSC record field at C0H.	Table 20-61.	
PEBS EventingIP	Yes	Yes	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register.	MSR 1A6H and 1A7H, shared by a pair of cores.	Nehalem supports 1A6H only.

20.5.3.1 Processor Event Based Sampling (PEBS)

Processor event based sampling (PEBS) on the Goldmont microarchitecture is enhanced over prior generations with respect to sampling support of precise events and non-precise events. In the Goldmont microarchitecture, PEBS is supported using IA32_PMC0 for all events (see Section 18.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor at the time the sample was generated.

Precise events work the same way on Goldmont microarchitecture as on the Silvermont microarchitecture. The record will be generated after an instruction that causes the event when the counter is already overflowed and will capture the architectural state at this point (see Section 20.6.2.4 and Section 18.4.9). The eventingIP in the record will indicate the instruction that caused the event. The list of precise events supported in the Goldmont microarchitecture is shown in Table 20-67.

In the Goldmont microarchitecture, the PEBS facility also supports the use of non-precise events to record processor state information into PEBS records with the same format as with precise events.

However, a non-precise event may not be attributable to a particular retired instruction or the time of instruction execution. When the counter overflows, a PEBS record will be generated at the next opportunity. Consider the event ICACHE.HIT. When the counter overflows, the processor is fetching future instructions. The PEBS record will be generated at the next opportunity and capture the state at the processor's current retirement point. It is likely that the instruction fetch that caused the event to increment was beyond that current retirement point. Other examples of non-precise events are CPU_CLK_UNHALTED.CORE_P and HARDWARE_INTERRUPTS.RECEIVED. CPU_CLK_UNHALTED.CORE_P will increment each cycle that the processor is awake. When the counter over-flows, there may be many instructions in various stages of execution. Additionally, zero, one or multiple instructions may be retired the cycle that the counter overflows. HARDWARE_INTERRUPTS.RECEIVED increments independent of any instructions being executed. For all non-precise events, the PEBS record will be generated at the next opportunity, after the counter has overflowed. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record for a non-precise event, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked and without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32_PERF_GLOBAL_STATUS_SET.

Table 20-67. Precise Events Supported by the Goldmont Microarchitecture

Event Name	Event Select	Sub-event	UMask
LD_BLOCKS	03H	DATA_UNKNOWN	01H
		STORE_FORWARD	02H
		4K_ALIAS	04H
		UTLB_MISS	08H
		ALL_BLOCK	10H
MISALIGN_MEM_REF	13H	LOAD_PAGE_SPLIT	02H
		STORE_PAGE_SPLIT	04H
INST_RETIRED	C0H	ANY	00H
UOPS_RETITRED	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H

Table 20-67. Precise Events Supported by the Goldmont Microarchitecture (Contd.)

Event Name	Event Select	Sub-event	UMask
MEM_UOPS_RETIRED	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRED	D1H	L1_HIT	01H
		L2_HIT	02H
		L1_MISS	08H
		L2_MISS	10H
		HITM	20H
		WCB_HIT	40H
		DRAM_HIT	80H

The PEBS record format supported by processors based on the Goldmont microarchitecture is shown in Table 20-68, and each field in the PEBS record is 64 bits long.

Table 20-68. PEBS Record Format for the Goldmont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counters
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Reserved
40H	R/EBP	A8H	Reserved
48H	R/ESP	B0H	EventingRIP
50H	R8	B8H	Reserved
58H	R9	C0H	TSC
60H	R10		

On Goldmont microarchitecture, all 64 bits of architectural registers are written into the PEBS record regardless of processor mode.

With PEBS record format encoding 0011b, offset 90H reports the “Applicable Counter” field, which indicates which counters actually requested generating a PEBS record. This allows software to correlate the PEBS record entry properly with the instruction that caused the event even when multiple counters are configured to record PEBS records and multiple bits are set in the field. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

20.5.3.1.1 PEBS Data Linear Address Profiling

Goldmont supports the Data Linear Address field introduced in Haswell. It does not support the Data Source Encoding or Latency Value fields that are also part of Data Address Profiling; those fields are present in the record but are reserved.

For Goldmont microarchitecture, the Data Linear Address field will record the linear address of memory accesses in the previous instruction (e.g., the one that triggered a precise event that caused the PEBS record to be generated). Goldmont microarchitecture may record a Data Linear Address for the instruction that caused the event even for events not related to memory accesses. This may differ from other microarchitectures.

20.5.3.1.2 Reduced Skid PEBS

Processors based on Goldmont Plus microarchitecture support the Reduced Skid PEBS feature described in Section 20.9.4 on the IA32_PMC0 counter. Although Extended PEBS adds support for generating PEBS records for precise events on additional general-purpose and fixed-function performance counters, those counters do not support the Reduced Skid PEBS feature.

20.5.3.1.3 Enhancements to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62]

In addition to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62] being set when PEBS_Index reaches the PEBS_Interrupt_Theshold, the bit is also set when PEBS_Index is out of bounds. That is, the bit will be set when PEBS_Index < PEBS_Buffer_Base or PEBS_Index > PEBS_Absolute_Maximum. Note that when an out of bound condition is encountered, the overflow bits in IA32_PERF_GLOBAL_STATUS will be cleared according to Applicable Counters, however the IA32_PMCx values will not be reloaded with the Reset values stored in the DS_AREA.

20.5.3.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 20-62 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

The Goldmont microarchitecture provides unique pairs of MSR_OFFCORE_RSPx registers per core.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as follows:

- Bits 15:0 specifies the request type of a transaction request to the uncore. This is described in Table 20-69.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 20-64.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 20-70.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 20.5.2.3 for details.

Table 20-69. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	Counts demand instruction cacheline and I-side prefetch requests that miss the instruction cache.
COREWB	3	Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.

Table 20-69. MSR_OFFCORE_RSPx Request_Type Field Definition (Contd.)

Bit Name	Offset	Description
PARTIAL_READS	7	Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	Counts cacheline requests due to software prefetch instructions.
PF_L1_DATA_RD	13	Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	Counts partial cacheline writes due to streaming stores.
ANY_REQUEST	15	Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 20-63) and a valid response type pattern (either Table 20-64 or Table 20-70). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 20-70. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests

Subtype	Bit Name	Offset	Description
L2_MISS (Snoop Info)	Reserved	32:31	Reserved
	L2_MISS.SNOOP_MISS_OR_NO_SNOOP_NEEDED	33	A true miss to this module, for which a snoop request missed the other module or no snoop was performed/needed.
	L2_MISS.HIT_OTHER_CORE_NO_FWD	34	A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_CORE	36	Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	Target was a non-DRAM system address. This includes MMIO transactions.
Outstanding requests[1]	OUTSTANDING	38	Counts weighted cycles of outstanding offcore requests of the request type specified in bits 15:0, from the time the XQ receives the request and any response is received. Bits 37:16 must be set to 0. This bit is only available in MSR_OFFCORE_RESP0.

NOTES:

1. See Section 20.5.2.3, "Average Offcore Request Latency Measurement," for details on how to use this bit to extract average latency.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

Any_Response Bit | L2 Hit | 'OR' of Snoop Info Bits | Outstanding Bit

20.5.3.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 20.5.2.3.

20.5.4 Performance Monitoring for Goldmont Plus Microarchitecture

Intel Atom processors based on the Goldmont Plus microarchitecture report architectural performance monitoring versionID = 4 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 20.2.4.

Goldmont Plus performance monitoring capabilities are similar to Goldmont capabilities. The differences are in specific events and in which counters support PEBS. Goldmont Plus introduces the ability for fixed performance monitoring counters to generate PEBS records.

Goldmont Plus will set the AnyThread deprecation CPUID bit (CPUID.0AH:EDX[15]) to indicate that the Any-Thread bits in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL have no effect.

The core PMU's capability is similar to that of the Goldmont microarchitecture described in Section 20.6.3, with some differences and enhancements summarized in Table 20-71.

Table 20-71. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures

Box	Goldmont Plus Microarchitecture	Goldmont Microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 20.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 20.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change.
Architectural Performance Monitoring version ID	4	4	No change.
Processor Event Based Sampling (PEBS) Events	All General-Purpose and Fixed counters. Each General-Purpose counter supports all events (precise and non-precise).	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 20-67.	Goldmont Plus supports PEBS on all counters.
PEBS record format encoding	0011b	0011b	No change.

20.5.4.1 Extended PEBS

The PEBS facility in Goldmont Plus microarchitecture provides a number of enhancements relative to PEBS in processors from previous generations. Enhancement of PEBS facility with the Extended PEBS feature are described in detail in section 18.9.

20.5.5 Performance Monitoring for Tremont Microarchitecture

Intel Atom processors based on the Tremont microarchitecture report architectural performance monitoring versionID = 5 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 5 capabilities are described in Section 20.2.5.

Tremont performance monitoring capabilities are similar to Goldmont Plus capabilities, with the following extensions:

- Support for Adaptive PEBS.
- Support for PEBS output to Intel® Processor Trace.
- Precise Distribution support on Fixed Counter0.
- Compatibility enhancements to off-core response MSRs, MSR_OFFCORE_RSPx.

The differences and enhancements between Tremont microarchitecture and Goldmont Plus microarchitecture are summarized in Table 20-72.

Table 20-72. Core PMU Comparison Between the Tremont and Goldmont Plus Microarchitectures

Box	Tremont Microarchitecture	Goldmont Plus Microarchitecture	Comment
# of fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 20.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 20.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change. See Section 20.2.2.
Architectural Performance Monitoring version ID	5	4	
PEBS record format encoding	0100b	0011b	See Section 20.6.2.4.2.
Reduce skid PEBS	IA32_PMC0 and IA32_FIXED_CTR0	IA32_PMC0 only	
Extended PEBS	Yes	Yes	See Section 20.5.4.1.
Adaptive PEBS	Yes	No	See Section 20.9.2.
PEBS output	DS Save Area or Intel® Processor Trace	DS Save Area only	See Section 20.5.5.2.1.
PEBS record layout	See Section 20.9.2.3 for output to DS, Section 20.5.5.2.2 for output to Intel PT.	Table 20-68; enhanced fields at offsets 90H- 98H; and TSC record field at C0H.	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register, extended request and response types.	MSR 1A6H and 1A7H, each core has its own register.	

20.5.5.1 Adaptive PEBS

The PEBS record format and configuration interface has changed versus Goldmont Plus, as the Tremont microarchitecture includes support for the configurable Adaptive PEBS records; see Section 20.9.2.

20.5.5.2 PEBS output to Intel® Processor Trace

Intel Atom processors based on the Tremont microarchitecture introduce the following Precise Event-Based Sampling (PEBS) extensions:

- A mechanism to direct PEBS output into the Intel® Processor Trace (Intel® PT) output stream. In this scenario, the PEBS record is written in packetized form, in order to co-exist with other Intel PT trace data.
- New Performance Monitoring counter reload MSRs, which are used by PEBS in place of the counter reload values stored in the DS Management area when PEBS output is directed into the Intel PT output stream.

Processors that indicate support for Intel PT by setting CPUID.07H.0.EBX[25]=1, and set the new IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16] bit, support these extensions.

20.5.5.2.1 PEBS Configuration

PEBS output to Intel Processor Trace includes support for two new fields in IA32_PEBS_ENABLE.

Table 20-73. New Fields in IA32_PEBS_ENABLE

Field	Description
PMI_AFTER_EACH_RECORD[60]	Pend a PerfMon Interrupt (PMI) after each PEBS event.
PEBS_OUTPUT[62:61]	Specifies PEBS output destination. Encodings: 00B: DS Save Area. Matches legacy PEBS behavior, output location defined by IA32_DS_AREA. 01B: Intel PT trace output. 10B: Reserved. 11B: Reserved.

When PEBS_OUTPUT is set to 01B, the DS Management Area is not used and need not be configured. Instead, the output mechanism is configured through IA32_RTIT_CTL and other Intel PT MSRs, while counter reload values are configured in the MSR_RELOAD_PMCx MSRs. Details on configuring Intel PT can be found in Section 33.2.7.

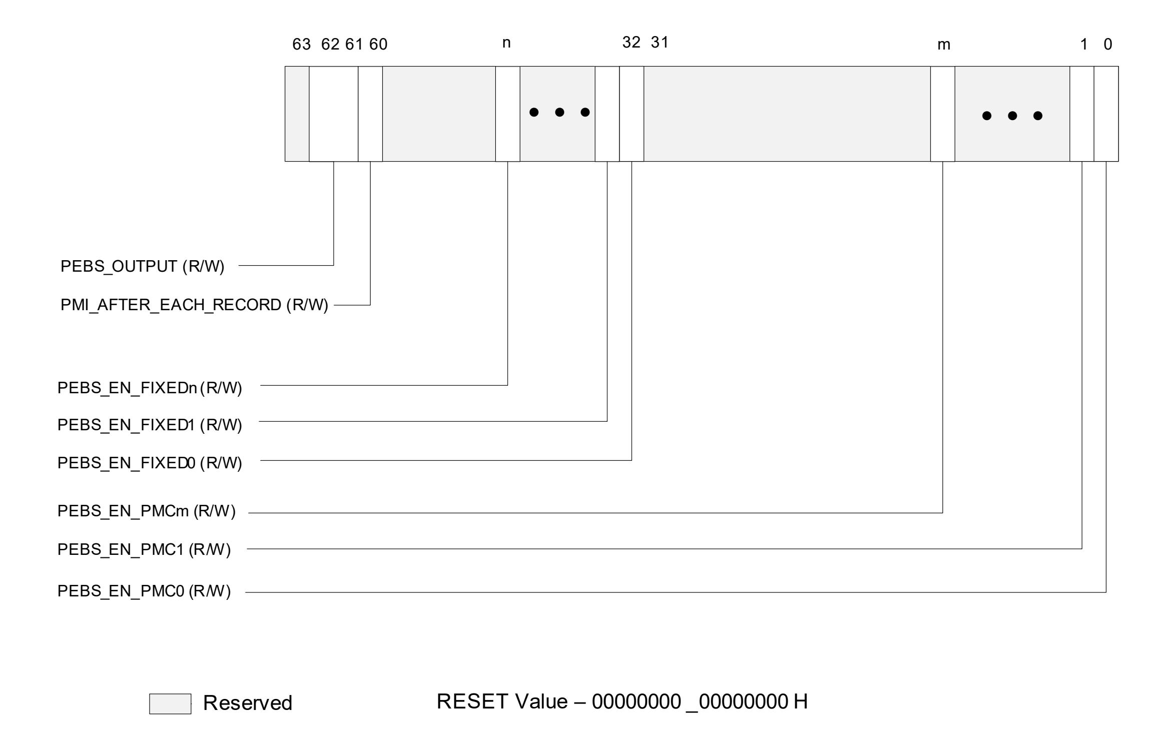


Figure 20-42. IA32_PEBS_ENABLE MSR with PEBS Output to Intel® Processor Trace

20.5.5.2.2 PEBS Record Format in Intel® Processor Trace

The format of the PEBS record changes when output to Intel PT, as the PEBS state is packetized. Each PEBS grouping is emitted as a Block Begin (BBP) and following Block Item (BIP) packets. A PEBS grouping ends when either a new PEBS grouping begins (indicated by a BBP packet) or a Block End (BEP) packet is encountered. See Section 33.4.1.1 for details of these Intel PT packets.

Because the packet headers describe the state held in the packet payload, PEBS state ordering is not fixed. PEBS state groupings may be emitted in any order, and the PEBS state elements within those groupings may be emitted in any order. Further, there is no packet that provides indication of “Record Format” or “Record Size”.

If Intel PT tracing is not enabled (IA32_RTIT_STATUS.TriggerEn=0), any PEBS records triggered will be dropped. PEBS packets do not depend on ContextEn or FilterEn in IA32_RTIT_STATUS, any filtering of PEBS must be enabled from within the PerfMon configuration. Counter reload will occur in all scenarios where PEBS is triggered, regardless of TriggerEn.

The PEBS threshold mechanism for generating PerfMon Interrupts (PMIs) is not available in this mode. However, there exist other means to generate PMIs based on PEBS output. When the Intel PT ToPA output mechanism is chosen, a PMI can optionally be pended when a ToPA region is filled; see Section 33.2.7.2 for details. Further, software can opt to generate a PMI on each PEBS record by setting the new IA32_PEBS_ENABLE.PMI_AFTER_EACH_RECORD[60] bit.

The IA32_PERF_GLOBAL_STATUS.OvfDSBuffer bit will not be set in this mode.

20.5.5.2.3 PEBS Counter Reload

When PEBS output is directed into Intel PT (IA32_PEBS_ENABLE.PEBS_OUTPUT = 01B), new MSR_RELOAD_PMCx MSRs are used by the PEBS routine to reload PerfMon counters. The value from the associated reload MSR will be loaded to the appropriate counter on each PEBS event.

20.5.5.3 Precise Distribution Support on Fixed Counter 0

The Tremont microarchitecture supports the PDIR (Precise Distribution of Retired Instructions) facility, as described in Section 20.3.4.4.4, on Fixed Counter 0. Fixed Counter 0 counts the INST_RETIRED.ALL event. PEBS skid for Fixed Counter 0 will be precisely one instruction.

This is in addition to the reduced skid PEBS behavior on IA32_PMC0; see Section 20.5.3.1.2.

20.5.5.4 Compatibility Enhancements to Offcore Response MSRs

The Off-core Response facility is similar to that described in Section 20.5.3.2.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as shown below. RequestType bits are defined in Table 20-74, ResponseType bits in Table 20-75, and SnoopInfo bits in Table 20-76.

Table 20-74. MSR_OFFCORE_RSPx Request Type Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data reads.
DEMAND_RFO	1	Counts all demand reads for ownership (RFO) requests and software based prefetches for exclusive ownership (prefetchw).
DEMAND_CODE_RD	2	Counts demand instruction fetches and L1 instruction cache prefetches.
COREWB_M	3	Counts modified write backs from L1 and L2.
HWPF_L2_DATA_RD	4	Counts prefetch (that bring data to L2) data reads.
HWPF_L2_RFO	5	Counts all prefetch (that bring data to L2) RFOs.
HWPF_L2_CODE_RD	6	Counts all prefetch (that bring data to L2 only) code reads.
Reserved	9:7	Reserved.
HWPF_L1D_AND_SWPF	10	Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).
STREAMING_WR	11	Counts all streaming stores.
COREWB_NONM	12	Counts non-modified write backs from L2.
Reserved	14:13	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O accesses that have any response type.
UC_RD	44	Counts uncached memory reads (PRd, UCRdF).
UC_WR	45	Counts uncached memory writes (WiL).
PARTIAL_STREAMING_WR	46	Counts partial (less than 64 byte) streaming stores (WCiL).
FULL_STREAMING_WR	47	Counts full, 64 byte streaming stores (WCiLF).

Table 20-74. MSR_OFFCORE_RSPx Request Type Definition (Contd.)

Bit Name	Offset	Description
L1WB_M	48	Counts modified WriteBacks from L1 that miss the L2.
L2WB_M	49	Counts modified WriteBacks from L2.

Table 20-75. MSR_OFFCORE_RSPx Response Type Definition

Bit Name	Offset	Description
ANY_RESPONSE	16	Catch all value for any response types.
L3_HIT_M	18	LLC/L3 Hit - M-state.
L3_HIT_E	19	LLC/L3 Hit - E-state.
L3_HIT_S	20	LLC/L3 Hit - S-state.
L3_HIT_F	21	LLC/L3 Hit - I-state.
LOCAL_DRAM	26	LLC/L3 Miss, DRAM Hit.
OUTSTANDING	63	Average latency of outstanding requests with the other counter counting number of occurrences; can also can be used to count occupancy.

Table 20-76. MSR_OFFCORE_RSPx Snoop Info Definition

Bit Name	Offset	Description
SNOOP_NONE	31	None of the cores were snooped. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_NOT_NEEDED	32	No snoop needed to satisfy the request. ▪ LLC hit and CV bit(s) (core valid) was not set. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_MISS	33	A snoop was sent but missed. ▪ LLC hit and CV bit(s) was set but snoop missed (silent data drop in core), data returned from LLC. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_HIT_NO_FWD	34	A snoop was sent but no data forward. ▪ LLC hit and CV bit(s) was set but no data forward from the core, data returned from LLC. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_HIT_WITH_FWD	35	A snoop was sent and non-modified data was forward. ▪ LLC hit and CV bit(s) was set, non-modified data was forward from core.
SNOOP_HITM	36	A snoop was sent and modified data was forward. ▪ LLC hit E or M and the CV bit(s) was set, modified data was forward from core.
NON_DRAM_BIT	37	Target was non-DRAM system address, MMIO access. ▪ LLC miss and Non-Dram data returned.

The Off-core Response capability behaves as follows:

- To specify a complete offcore response filter, software must properly program at least one RequestType and one ResponseType. A valid request type must have at least one bit set in the non-reserved bits of 15:0 or 49:44. A valid response type must be a non-zero value of one the following expressions:
  - Read requests:

    Any_Response Bit | (‘OR’ of Supplier Info Bits) ‘AND’ (‘OR’ of Snoop Info Bits) | Outstanding Bit
  - Write requests:

    Any_Response Bit | (‘OR’ of Supplier Info Bits) | Outstanding Bit
- When the ANY_RESPONSE bit in the ResponseType is set, all other response type bits will be ignored.
- True Demand Cacheable Loads include neither L1 Prefetches nor Software Prefetches.
- Bits 15:0 and Bits 49:44 specifies the request type of a transaction request to the uncore. This is described in Table 20-74.
- Bits 30:16 specifies common supplier information.
- “Outstanding Requests” (bit 63) is only available on MSR_OFFCORE_RSP0; a #GP fault will occur if software attempts to write a 1 to this bit in MSR_OFFCORE_RSP1. It is mutually exclusive with any ResponseType. Software must guarantee that all other ResponseType bits are set to 0 when the “Outstanding Requests” bit is set.
- “Outstanding Requests” bit 63 can enable measurement of the average latency of a specific type of off-core transaction; two programmable counters must be used simultaneously and the RequestType programming for MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 must be the same when using this Average Latency feature. See Section 20.5.2.3 for further details.

20.6 PERFORMANCE MONITORING (LEGACY INTEL PROCESSORS)

20.6.1 Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)

In Intel Core Solo and Intel Core Duo processors, non-architectural performance monitoring events are programmed using the same facilities (see Figure 20-1) used for architectural performance events.

Non-architectural performance events use event select values that are model-specific. Event mask (Umask) values are also specific to event logic units. Some microarchitectural conditions detectable by a Umask value may have specificity related to processor topology (see Section 9.6, “Detecting Hardware Multi-Threading Support and Topology,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). As a result, the unit mask field (for example, IA32_PERFEVTSELx[bits 15:8]) may contain sub-fields that specify topology information of processor cores.

The sub-field layout within the Umask field may support two-bit encoding that qualifies the relationship between a microarchitectural condition and the originating core. This data is shown in Table 20-77. The two-bit encoding for core-specificity is only supported for a subset of Umask values (see: https://perfmon-events.intel.com/) and for Intel Core Duo processors. Such events are referred to as core-specific events.

Table 20-77. Core Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 15:14 Encoding	**Description**
11B	All cores
10B	Reserved
01B	This core
00B	Reserved

Some microarchitectural conditions allow detection specificity only at the boundary of physical processors. Some bus events belong to this category, providing specificity between the originating physical processor (a bus agent) versus other agents on the bus. Sub-field encoding for agent specificity is shown in Table 20-78.

Table 20-78. Agent Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13 Encoding	**Description**
0	This agent
1	Include all agents

Some microarchitectural conditions are detectable only from the originating core. In such cases, unit mask does not support core-specificity or agent-specificity encodings. These are referred to as core-only conditions.

Some microarchitectural conditions allow detection specificity that includes or excludes the action of hardware prefetches. A two-bit encoding may be supported to qualify hardware prefetch actions. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 20-79.

Table 20-79. HW Prefetch Qualification Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13:12 Encoding	**Description**
11B	All inclusive
10B	Reserved
01B	Hardware prefetch only
00B	Exclude hardware prefetch

Some performance events may (a) support none of the three event-specific qualification encodings (b) may support core-specificity and agent specificity simultaneously (c) or may support core-specificity and hardware prefetch qualification simultaneously. Agent-specificity and hardware prefetch qualification are mutually exclusive.

In addition, some L2 events permit qualifications that distinguish cache coherent states. The sub-field definition for cache coherency state qualification is shown in Table 20-80. If no bits in the MESI qualification sub-field are set for an event that requires setting MESI qualification bits, the event count will not increment.

Table 20-80. MESI Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	Counts modified state
Bit 10	Counts exclusive state
Bit 9	Counts shared state
Bit 8	Counts Invalid state

20.6.2 Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)

In addition to architectural performance monitoring, processors based on the Intel Core microarchitecture support non-architectural performance monitoring events.

Architectural performance events can be collected using general-purpose performance counters. Non-architectural performance events can be collected using general-purpose performance counters (coupled with two IA32_PERFEVTSELx MSRs for detailed event configurations), or fixed-function performance counters (see Section 20.6.2.1). IA32_PERFEVTSELx MSRs are architectural; their layout is shown in Figure 20-1. Starting with Intel Core 2 processor T 7700, fixed-function performance counters and associated counter control and status MSR becomes part of architectural performance monitoring version 2 facilities (see also Section 20.2.2).

Non-architectural performance events in processors based on Intel Core microarchitecture use event select values that are model-specific. Valid event mask (Umask) bits can be found at: https://perfmon-events.intel.com/. The UMASK field may contain sub-fields identical to those listed in Table 20-77, Table 20-78, Table 20-79, and Table 20-80. One or more of these sub-fields may apply to specific events on an event-by-event basis.

In addition, the UMASK filed may also contain a sub-field that allows detection specificity related to snoop responses. Bits of the snoop response qualification sub-field are defined in Table 20-81.

Table 20-81. Bus Snoop Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	HITM response
Bit 10	Reserved
Bit 9	HIT response
Bit 8	CLEAN response

There are also non-architectural events that support qualification of different types of snoop operation. The corresponding bit field for snoop type qualification are listed in Table 20-82.

Table 20-82. Snoop Type Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 9:8	**Description**
Bit 9	CMP2I snoops
Bit 8	CMP2S snoops

No more than one sub-field of MESI, snoop response, and snoop type qualification sub-fields can be supported in a performance event.

NOTE

Software must write known values to the performance counters prior to enabling the counters. The content of general-purpose counters and fixed-function counters are undefined after INIT or RESET.

20.6.2.1 Fixed-function Performance Counters

Processors based on Intel Core microarchitecture provide three fixed-function performance counters. Bits beyond the width of the fixed counter are reserved and must be written as zeros. Model-specific fixed-function performance counters on processors that support Architectural Perfmon version 1 are 40 bits wide.

Each of the fixed-function counter is dedicated to count a pre-defined performance monitoring events. See Table 20-2 for details of the PMC addresses and what these events count.

Programming the fixed-function performance counters does not involve any of the IA32_PERFEVTSELx MSRs, and does not require specifying any event masks. Instead, the MSR IA32_FIXED_CTR_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 20-43. Two sub-fields are defined for each control. See Figure 20-43; bit fields are:

- **Enable field (low 2 bits in each 4-bit control) —** When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring 0.

When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring greater than 0.

Writing 0 to both bits stops the performance counter. Writing 11B causes the counter to increment irrespective of privilege levels.

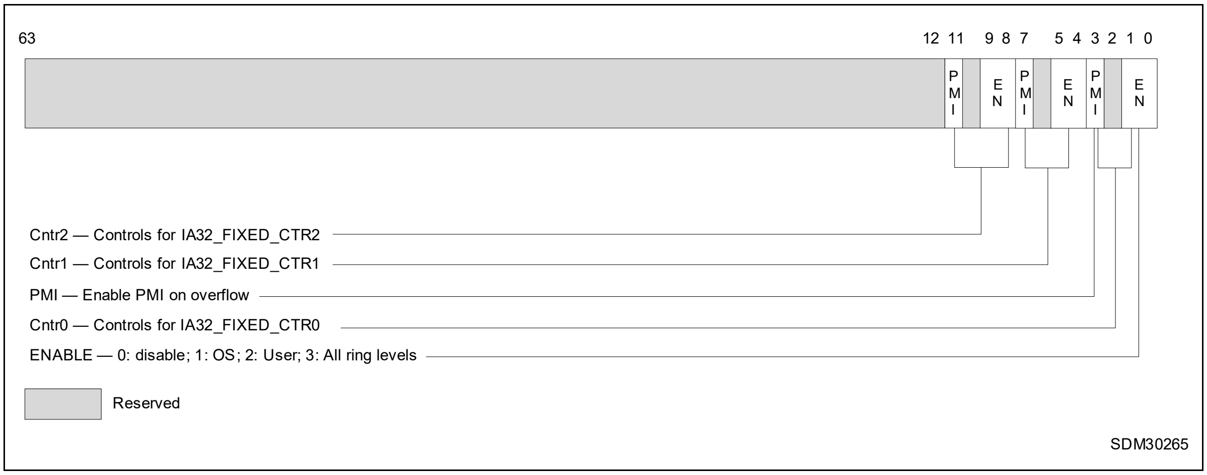

63
12 11
9 8 7
5 4 3 2 1 0
PMI
EN
PMI
EN
PMI
EN
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow
Cntr0 — Controls for IA32_FIXED_CTR0
ENABLE — 0: disable; 1: OS; 2: User; 3: All ring levels
Reserved
SDM30265

Figure 20-43. Layout of IA32_FIXED_CTR_CTRL MSR

- **PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

20.6.2.2 Global Counter Control Facilities

Processors based on Intel Core microarchitecture provides simplified performance counter control that simplifies the most frequent operations in programming performance events, i.e., enabling/disabling event counting and checking the status of counter overflows. This is done by the following three MSRs:

- MSR_PERF_GLOBAL_CTRL enables/disables event counting for all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
- MSR_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single RDMSR.
- MSR_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.

MSR_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 20-44). Each enable bit in MSR_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

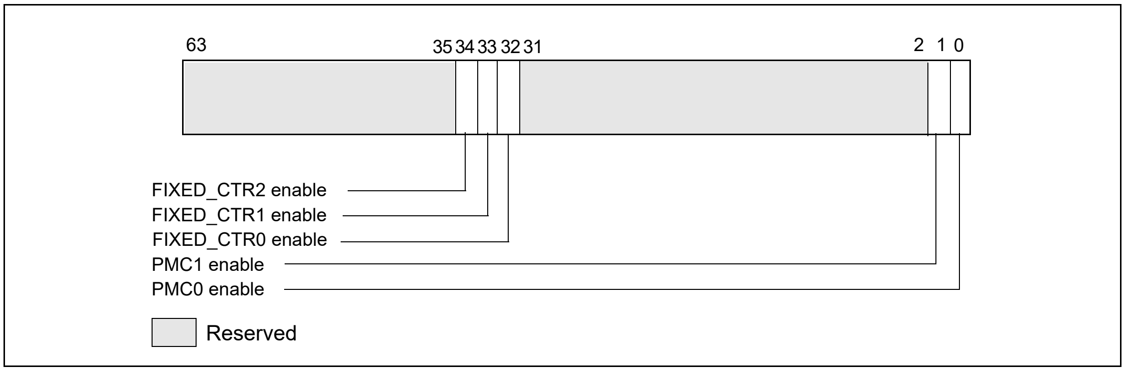


Figure 20-44. Layout of MSR_PERF_GLOBAL_CTRL MSR

MSR_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. MSR_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. MSR_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware (see Figure 20-45). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has occurred in the associated counter.

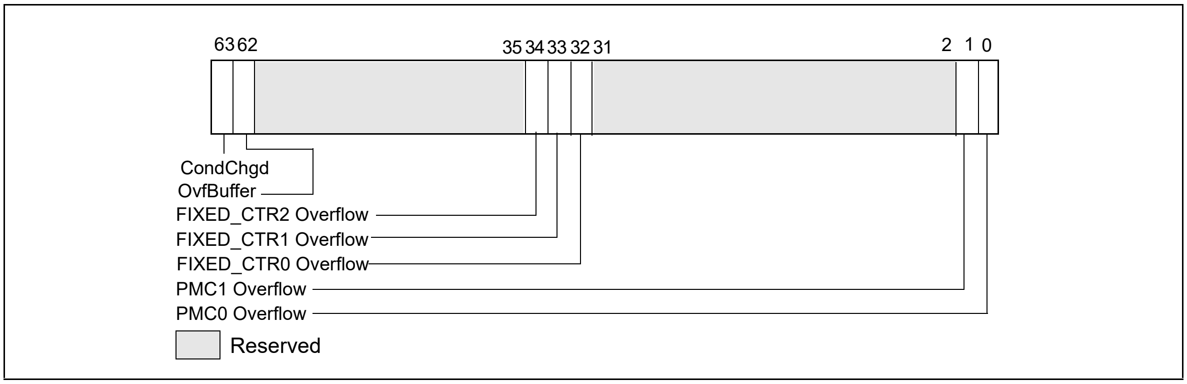


Figure 20-45. Layout of MSR_PERF_GLOBAL_STATUS MSR

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 18.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

MSR_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 20-46). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

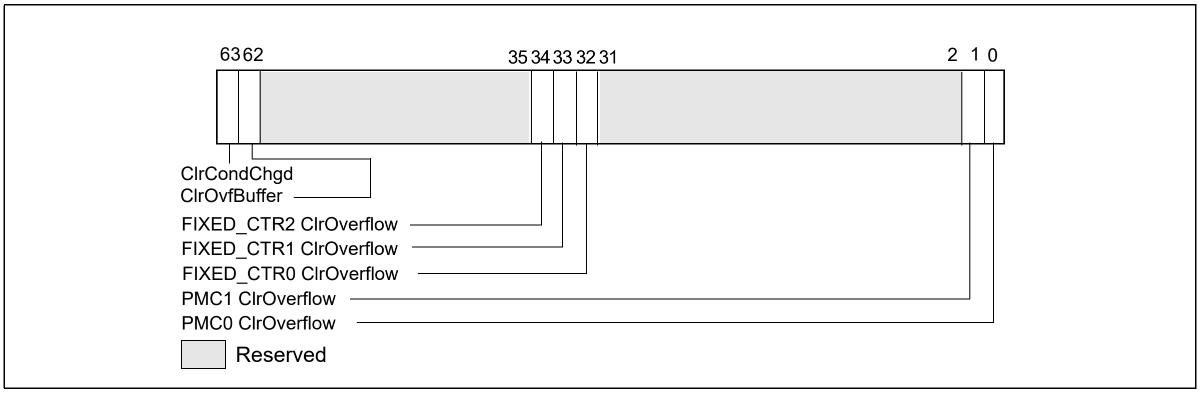


Figure 20-46. Layout of MSR_PERF_GLOBAL_OVF_CTRL MSR

20.6.2.3 At-Retirement Events

Many non-architectural performance events are impacted by the speculative nature of out-of-order execution. A subset of non-architectural performance events on processors based on Intel Core microarchitecture are enhanced with a tagging mechanism (similar to that found in Intel NetBurst® microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core microarchitecture does not require special MSR programming control (see Section 20.6.3.6, “At-Retirement Counting”), but is limited to IA32_PMC0. See Table 20-83 for a list of events available to processors based on Intel Core microarchitecture.

Table 20-83. At-Retirement Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
ITLB_MISS_RETIRED	00H	C9H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

20.6.2.4 Processor Event Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support processor event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 20.6.2.4.2 and Section 18.4.9).

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, precise events that can be used with PEBS are listed in Table 20-84. The procedure for detecting availability of PEBS is the same as described in Section 20.6.3.8.1.

Table 20-84. PEBS Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

20.6.2.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32_PMC0 only. Use the following procedure to set up the processor and IA32_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 18-8 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32_PEBS_ENABLE MSR.
3. Set up the IA32_PMC0 performance counter and IA32_PERFEVTSEL0 for an event listed in Table 20-84.

20.6.2.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32_PERF_CAPABILITES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version ID equals 2 or higher. The bit fields of IA32_PERF_CAPABILITES are defined in Table 2-2 of Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. The relevant bit fields that governs PEBS are:

- PEBSTrap [bit 6]: When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.
- PEBSSaveArchRegs [bit 7]: When set, PEBS will save architectural register and state information according to the encoded value of the PEBSRecordFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1.
- PEBSRecordFormat [bits 11:8]: Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (See Section 20.6.3.8).
  - 0001B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS and load latency data. (See Section 20.3.1.1.1).
  - 0010B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS, load latency data, and TSX tuning information. (See Section 20.3.6.2).
  - 0011B: PEBS record includes additional information of load latency data, TSX tuning information, TSC data, and the applicable counter field replaces IA32_PERF_GLOBAL_STATUS at offset 90H. (See Section 20.3.8.1.1).
  - 0100B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG. (See Section 20.9.2.3). The PEBS Configuration Buffer is defined as shown in Figure 20-64 with Counter Reset fields allocation for 8 general-purpose counters followed by 4 fixed-function counters.

— 0101B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG. (See Section 20.9.2.3). The PEBS Configuration Buffer is defined as shown in Figure 20-64 with Counter Reset fields allocation for 32 general-purpose counters followed by 16 fixed-function counters.

20.6.2.4.3 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the Interrupt-based event sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 18.4.9.1, "64 Bit Format of the DS Save Area," for guidelines when writing the DS ISR.

The service routine can query MSR_PERF_GLOBAL_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR_PERF_GLOBAL_OVF_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 20-85.

Table 20-85. Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS.	▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled.	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0. On subsequent entries: ▪ Clear all counters if "Counter Freeze on PMI" is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[1]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write "sample-after" values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable.	▪ Set appropriate OVF_PMI bits - 1. ▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

NOTES:

[1] Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

20.6.2.4.4 Re-configuring PEBS Facilities

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32_DS_AREA).

20.6.3 Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMC instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMC instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32_MISC_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCRs sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32_DS_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR_PEBS_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 20-86 lists the performance counters and their associated CCCRs, along with the ESCRs that select events to be counted for each performance counter. Predefined event metrics and events can be found at: https://perfmon-events.intel.com/.

Table 20-86. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_BPU_COUNTER0	0	300H	MSR_BPU_CCCR0	360H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H

Table 20-86. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCR0	364H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCR0	368H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER1	9	309H	MSR_FLAME_CCCR1	369H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H

Table 20-86. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER3	15	30FH	MSR_IQ_CCCR3	36FH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH

NOTES:

[1] MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- Non-retirement events are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

  The at-retirement counting mechanism includes facilities for tagging µops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting —** A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.
- **Interrupt-based event sampling —** A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted.

  When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.
- **Processor event-based sampling (PEBS) —** In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

20.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 20-86) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 20-86) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 20-47 shows the layout of an ESCR MSR. The functions of the flags and fields are:

- **USR flag, bit 2 —** When set, events are counted when the processor is operating at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.
- **OS flag, bit 3 —** When set, events are counted when the processor is operating at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the OS and USR flags are set, events are counted at all privilege levels.)

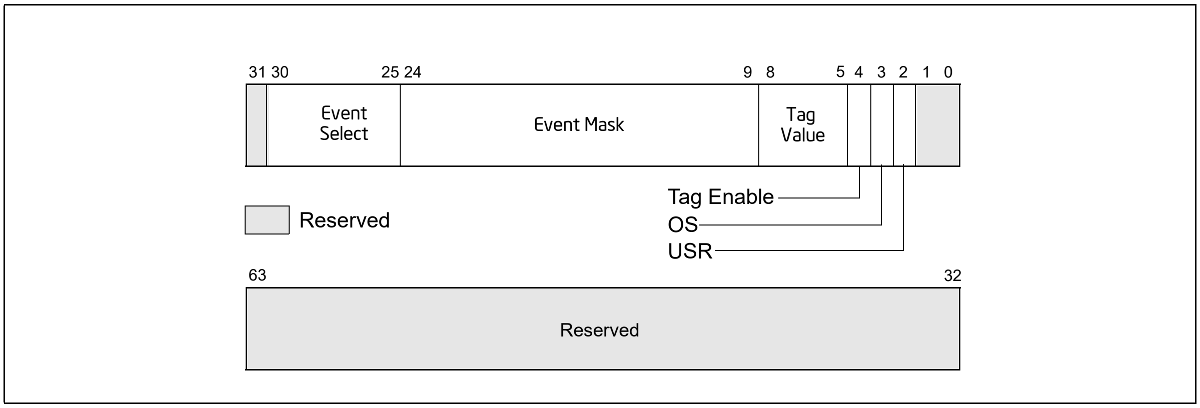


Figure 20-47. Event Selection Control Register (ESCR) for Pentium 4 and Intel® Xeon® Processors without Intel HT Technology Support

- **Tag enable, bit 4 —** When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 20.6.3.6, “At-Retirement Counting.”

- **Tag value field, bits 5 through 8 —** Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 20-86 gives the addresses of the ESCR MSRs.

Writing to an ESCR MSR does not enable counting with its associated performance counter; it only selects the event or events to be counted. The CCCR for the selected performance counter must also be configured. Configuration of the CCCR includes selecting the ESCR and enabling the counter.

20.6.3.2 Performance Counters

The performance counters in conjunction with the counter configuration control registers (CCCRs) are used for filtering and counting the events selected by the ESCRs. Processors based on Intel NetBurst microarchitecture provide 18 performance counters organized into 9 pairs. A pair of performance counters is associated with a particular subset of events and ESCR's (see Table 20-86). The counter pairs are partitioned into four groups:

- The BPU group, includes two performance counter pairs:
  - MSR_BPU_COUNTER0 and MSR_BPU_COUNTER1.
  - MSR_BPU_COUNTER2 and MSR_BPU_COUNTER3.
- The MS group, includes two performance counter pairs:
  - MSR_MS_COUNTER0 and MSR_MS_COUNTER1.
  - MSR_MS_COUNTER2 and MSR_MS_COUNTER3.
- The FLAME group, includes two performance counter pairs:
  - MSR_FLAME_COUNTER0 and MSR_FLAME_COUNTER1.
  - MSR_FLAME_COUNTER2 and MSR_FLAME_COUNTER3.
- The IQ group, includes three performance counter pairs:
  - MSR_IQ_COUNTER0 and MSR_IQ_COUNTER1.
  - MSR_IQ_COUNTER2 and MSR_IQ_COUNTER3.
  - MSR_IQ_COUNTER4 and MSR_IQ_COUNTER5.

The MSR_IQ_COUNTER4 counter in the IQ group provides support for the PEBS.

Alternate counters in each group can be cascaded: the first counter in one pair can start the first counter in the second pair and vice versa. A similar cascading is possible for the second counters in each pair. For example, within the BPU group of counters, MSR_BPU_COUNTER0 can start MSR_BPU_COUNTER2 and vice versa, and MSR_BPU_COUNTER1 can start MSR_BPU_COUNTER3 and vice versa (see Section 20.6.3.5.6, "Cascading Counters"). The cascade flag in the CCCR register for the performance counter enables the cascading of counters.

Each performance counter is 40-bits wide (see Figure 20-48). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or, if ECX[31] is set to 1, the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits. In such cases, counter bits 31:0 are written to EAX, while 0 is written to EDX.

The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

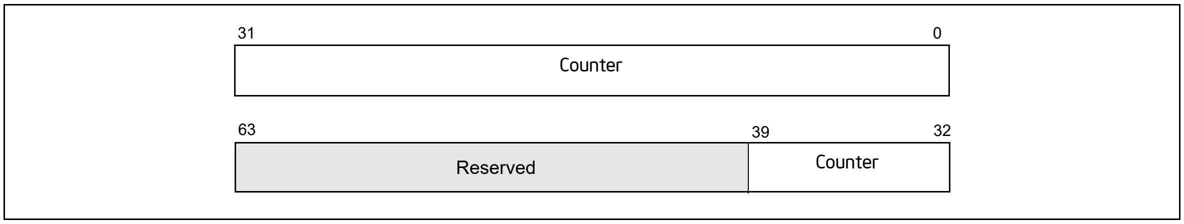


Figure 20-48. Performance Counter (Pentium 4 and Intel® Xeon® Processors)

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement negative integer in the counter. The counter will then count from the preset value up to -1 and overflow. Writing to a performance counter in a Pentium 4 or Intel Xeon processor with the WRMSR instruction causes all 40 bits of the counter to be written.

20.6.3.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 20-86). The CCCRs control the filtering and counting of events as well as interrupt generation. Figure 20-49 shows the layout of an CCCR MSR. The functions of the flags and fields are as follows:

- **Enable flag, bit 12 —** When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset.
- **ESCR select field, bits 13 through 15 —** Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Compare flag, bit 18 —** When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.
- **Complement flag, bit 19 —** Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 20.6.3.5.2, “Filtering Events”). The complement flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23 —** Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 20.6.3.5.2, “Filtering Events”).
- **Edge flag, bit 24 —** When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.

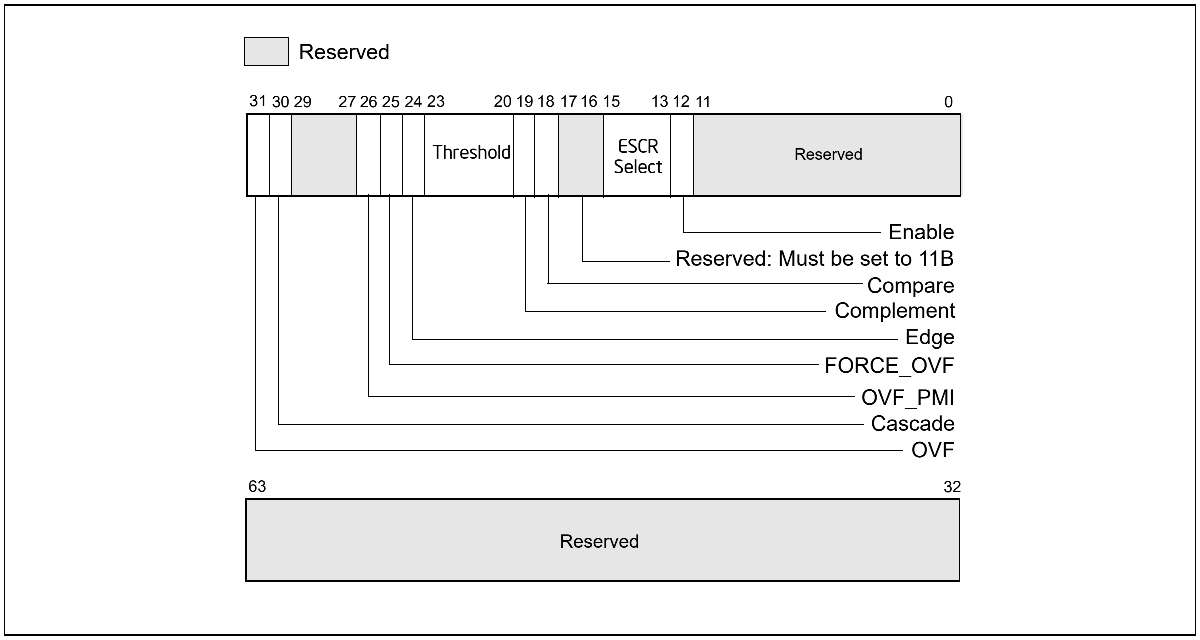


Figure 20-49. Counter Configuration Control Register (CCCR)

- **FORCE_OVF flag, bit 25 —** When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI flag, bit 26 —** When set, causes a performance monitor interrupt (PMI) to be generated when the counter overflows occurs; when clear, disables PMI generation. Note that the PMI is generated on the next event count after the counter has overflowed.
- **Cascade flag, bit 30 —** When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 20.6.3.2, "Performance Counters," for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31 —** Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

The CCCRs are initialized to all 0s on reset.

The events that an enabled performance counter actually counts are selected and filtered by the following flags and fields in the ESCR and CCCR registers and in the qualification order given:

1. The event select and event mask fields in the ESCR select a class of events to be counted and one or more event types within the class, respectively.
2. The OS and USR flags in the ESCR selected the privilege levels at which events will be counted.
3. The ESCR select field of the CCCR selects the ESCR. Since each counter has several ESCRs associated with it, one ESCR must be chosen to select the classes of events that may be counted.
4. The compare and complement flags and the threshold field of the CCCR select an optional threshold to be used in qualifying an event count.
5. The edge flag in the CCCR allows events to be counted only on rising-edge transitions.

The qualification order in the above list implies that the filtered output of one "stage" forms the input for the next. For instance, events filtered using the privilege level flags can be further qualified by the compare and complement flags and the threshold field, and an event that matched the threshold criteria, can be further qualified by edge detection.

The uses of the flags and fields in the CCCRs are discussed in greater detail in Section 20.6.3.5, "Programming the Performance Counters for Non-Retirement Events."

20.6.3.4 Debug Store (DS) Mechanism

The debug store (DS) mechanism was introduced with processors based on Intel NetBurst microarchitecture to allow various types of information to be collected in memory-resident buffers for use in debugging and tuning programs. The DS mechanism can be used to collect two types of information: branch records and processor event-based sampling (PEBS) records. The availability of the DS mechanism in a processor is indicated with the DS feature flag (bit 21) returned by the CPUID instruction.

See Section 18.4.5, "Branch Trace Store (BTS)," and Section 20.6.3.8, "Processor Event-Based Sampling (PEBS)," for a description of these facilities. Records collected with the DS mechanism are saved in the DS save area. See Section 18.4.9, "BTS and DS Save Area."

20.6.3.5 Programming the Performance Counters for Non-Retirement Events

The basic steps to program a performance counter and to count events include the following:

1. Select the event or events to be counted.
2. For each event, select an ESCR that supports the event.
3. Match the CCCR Select value and ESCR name to a value listed in Table 20-86; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which they are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

20.6.3.5.1 Selecting Events to Count

There is a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event, setup information is provided. Table 20-87 gives an example of one of the events.

Table 20-87. Event Example

Event Name	Event Parameters	Parameter Value	Description
branch_retired			Counts the retirement of a branch. Specify one or more mask bits to select any combination of branch taken, not-taken, predicted, and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 15-3 for the addresses of the ESCR MSRs.
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 15-3.
	ESCR Event Select	06H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bit 0: MMNP	Branch Not-taken Predicted
		1: MMNM	Branch Not-taken Mispredicted
		2: MMTP	Branch Taken Predicted
		3: MMTM	Branch Taken Mispredicted
	CCCR Select	05H	CCCR[15:13]

Table 20-87. Event Example (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

Event Parameters are described below.

- **ESCR restrictions —** Lists the ESCRs that can be used to program the event. Typically only one ESCR is needed to count an event.
- **Counter numbers per ESCR —** Lists which performance counters are associated with each ESCR. Table 20-86 gives the name of the counter and CCCR for each counter number. Typically only one counter is needed to count the event.
- **ESCR event select —** Gives the value to be placed in the event select field of the ESCR to select the event.
- **ESCR event mask —** Gives the value to be placed in the Event Mask field of the ESCR to select sub-events to be counted. The parameter value column defines the documented bits with relative bit position offset starting from 0, where the absolute bit position of relative offset 0 is bit 9 of the ESCR. All undocumented bits are reserved and should be set to 0.
- **CCCR select —** Gives the value to be placed in the ESCR select field of the CCCR associated with the counter to select the ESCR to be used to define the event. This value is not the address of the ESCR; it is the number of the ESCR from the Number column in Table 20-86.
- **Event specific notes —** Gives additional information about the event, such as the name of the same or a similar event defined for the P6 family processors.
- **Can support PEBS —** Indicates if PEBS is supported for the event (only supplied for at-retirement events).
- **Requires additional MSR for tagging —** Indicates which if any additional MSRs must be programmed to count the events (only supplied for the at-retirement events).

NOTE

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The following procedure shows how to set up a performance counter for basic counting; that is, the counter is set up to count a specified event indefinitely, wrapping around whenever it reaches its maximum count. This procedure is continued through the following four sections.

An event to be counted can be selected as follows:

1. Select the event to be counted.
2. Select the ESCR to be used to select events to be counted from the ESCRs field.
3. Select the number of the counter to be used to count the event from the Counter Numbers Per ESCR field.
4. Determine the name of the counter and the CCCR associated with the counter, and determine the MSR addresses of the counter, CCCR, and ESCR from Table 20-86.
5. Use the WRMSR instruction to write the ESCR Event Select and ESCR Event Mask values into the appropriate fields in the ESCR. At the same time set or clear the USR and OS flags in the ESCR as desired.
6. Use the WRMSR instruction to write the CCCR Select value into the appropriate field in the CCCR.

NOTE

> Typically all the fields and flags of the CCCR will be written with one WRMSR instruction; however, in this procedure, several WRMSR writes are used to more clearly demonstrate the uses of the various CCCR fields and flags.

This setup procedure is continued in the next section, Section 20.6.3.5.2, "Filtering Events."

20.6.3.5.2 Filtering Events

Each counter receives up to 4 input lines from the processor hardware from which it is counting events. The counter treats these inputs as binary inputs (input 0 has a value of 1, input 1 has a value of 2, input 3 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counters threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a "greater than" or a "less than or equal to" comparison of the input value vs. a threshold value can be made. The complement flag selects "less than or equal to" (flag set) or "greater than" (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, than any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a "rising edge" event; that is, a false-to-true transition. Figure 20-50 illustrates rising edge filtering.

The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 20.6.3.5.1, "Selecting Events to Count."

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
   - Set the compare flag.
   - Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
   - Enter a value from 0 to 15 in the threshold field.
8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 20.6.3.5.3, "Starting Event Counting."

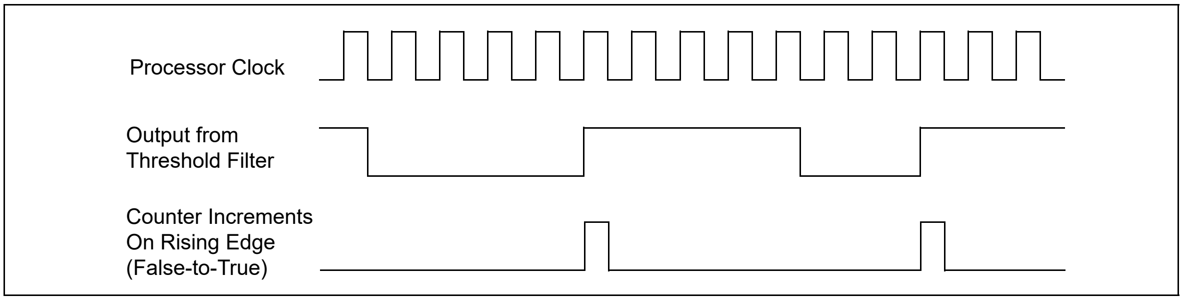

Processor Clock
Output from Threshold Filter
Counter Increments On Rising Edge (False-to-True)

Figure 20-50. Effects of Edge Filtering

20.6.3.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter's CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 20.6.3.5.5, "Halting Event Counting").

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 20.6.3.5.2, "Filtering Events."

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 20.6.3.5.4, "Reading a Performance Counter's Count."

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 20.6.3.5.6, "Cascading Counters").

20.6.3.5.4 Reading a Performance Counter's Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 20.6.3.2, "Performance Counters." These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 20.6.3.5.3, "Starting Event Counting."

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 20-86 used as an operand.

This setup procedure is continued in the next section, Section 20.6.3.5.5, "Halting Event Counting."

20.6.3.5.5 Halting Event Counting

After a performance counter has been started (enabled), it continues counting indefinitely. If the counter overflows (goes one count past its maximum count), it wraps around and continues counting. When the counter wraps around, it sets its OVF flag to indicate that the counter has overflowed. The OVF flag is a sticky flag that indicates that the counter has overflowed at least once since the OVF bit was last cleared.

To halt counting, the CCCR enable flag for the counter must be cleared.

The following procedural step shows how to stop event counting. This step is a continuation of the setup procedure introduced in Section 20.6.3.5.4, "Reading a Performance Counter's Count."

11. To stop event counting, execute a WRMSR instruction to clear the CCCR enable flag for the performance counter.

To halt a cascaded counter (a counter that was started when its alternate counter overflowed), either clear the Cascade flag in the cascaded counter's CCCR MSR or clear the OVF flag in the alternate counter's CCCR MSR.

20.6.3.5.6 Cascading Counters

As described in Section 20.6.3.2, "Performance Counters," eighteen performance counters are implemented in pairs. Nine pairs of counters and associated CCCRs are further organized as four blocks: BPU, MS, FLAME, and IQ (see Table 20-86). The first three blocks contain two pairs each. The IQ block contains three pairs of counters (12 through 17) with associated CCCRs (MSR_IQ_CCCR0 through MSR_IQ_CCCR5).

The first 8 counter pairs (0 through 15) can be programmed using ESCRs to detect performance monitoring events. Pairs of ESCRs in each of the four blocks allow many different types of events to be counted. The cascade flag in the CCCR MSR allows nested monitoring of events to be performed by cascading one counter to a second counter located in another pair in the same block (see Figure 20-49 for the location of the flag).

Counters 0 and 1 form the first pair in the BPU block. Either counter 0 or 1 can be programmed to detect an event via MSR_MO B_ESCR0. Counters 0 and 2 can be cascaded in any order, as can counters 1 and 3. It's possible to set up 4 counters in the same block to cascade on two pairs of independent events. The pairing described also applies to subsequent blocks. Since the IQ PUB has two extra counters, cascading operates somewhat differently if 16 and 17 are involved. In the IQ block, counter 16 can only be cascaded from counter 14 (not from 12); counter 14 cannot be cascaded from counter 16 using the CCCR cascade bit mechanism. Similar restrictions apply to counter 17.

Example 20-1. Counting Events

Assume a scenario where counter X is set up to count 200 occurrences of event A; then counter Y is set up to count 400 occurrences of event B. Each counter is set up to count a specific event and overflow to the next counter. In the above example, counter X is preset for a count of -200 and counter Y for a count of -400; this setup causes the counters to overflow on the 200th and 400th counts respectively.

Continuing this scenario, counter X is set up to count indefinitely and wraparound on overflow. This is described in the basic performance counter setup procedure that begins in Section 20.6.3.5.1, “Selecting Events to Count.” Counter Y is set up with the cascade flag in its associated CCCR MSR set to 1 and its enable flag set to 0.

To begin the nested counting, the enable bit for the counter X is set. Once enabled, counter X counts until it overflows. At this point, counter Y is automatically enabled and begins counting. Thus counter X overflows after 200 occurrences of event A. Counter Y then starts, counting 400 occurrences of event B before overflowing. When performance counters are cascaded, the counter Y would typically be set up to generate an interrupt on overflow. This is described in Section 20.6.3.5.8, “Generating an Interrupt on Overflow.”

The cascading counters mechanism can be used to count a single event. The counting begins on one counter then continues on the second counter after the first counter overflows. This technique doubles the number of event counts that can be recorded, since the contents of the two counters can be added together.

20.6.3.5.7 EXTENDED CASCADING

Extended cascading is a model-specific feature in the Intel NetBurst microarchitecture with CPUID DisplayFamily_DisplayModel 0F_02, 0F_03, 0F_04, 0F_06. This feature uses bit 11 in CCCRs associated with the IQ block. See Table 20-88.

Table 20-88. CCR Names and Bit Positions

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1|2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INTO0	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INTO3	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INTO4	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INTO5	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily_DisplayModel signature of 0F_02. For processors with CPUID DisplayFamily_DisplayModel signature of 0F_00 and 0F_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINTOy bits is used.

Example 20-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR_CRU_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR_IQ_CCCR0. This enables CASCNT4INTO0 and OVF_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR_CRU_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR_IQ_CCCR4 (set ENABLE bit, but not OVF_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INTOy bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

20.6.3.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter overflows. The PMI interrupt service routine can then collect information about the state of the processor or program when overflow occurred. This information can then be used with a tool like the Intel® VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR_PMI flag in the counter's associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 20.6.3.5.6, "Cascading Counters." Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.

20.6.3.5.9 Counter Usage Guideline

There are some instances where the user must take care to configure counting logic properly, so that it is not powered down. To use any ESCR, even when it is being used just for tagging, (any) one of the counters that the particular ESCR (or its paired ESCR) can be connected to should be enabled. If this is not done, 0 counts may result. Likewise, to use any counter, there must be some event selected in a corresponding ESCR (other than no_event, which generally has a select value of 0).

20.6.3.6 At-Retirement Counting

At-retirement counting provides a means counting only events that represent work committed to architectural state and ignoring work that was performed speculatively and later discarded.

One example of this speculative activity is branch prediction. When a branch misprediction occurs, the results of instructions that were decoded and executed down the mispredicted path are canceled. If a performance counter was set up to count all executed instructions, the count would include instructions whose results were canceled as well as those whose results committed to architectural state.

To provide finer granularity in event counting in these situations, the performance monitoring facilities provided in the Pentium 4 and Intel Xeon processors provide a mechanism for tagging events and then counting only those tagged events that represent committed results. This mechanism is called "at-retirement counting."

There are predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire —** In at-retirement event descriptions, the term "bogus" refers to instructions or μops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or μops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and μops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, Instruction_Retired and Uops_Retired) can count instructions or μops that are retired based on the characterization of bogus" versus non-bogus.
- **Tagging —** Tagging is a means of marking μops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per μop and a direct count of the event would not provide an indication of how many μops encountered that event.

  The tagging mechanisms allow a μop to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per μop, rather than once per event. For example, a μop may encounter a cache miss more than once during its life time, but a "Miss Retired" metric (that counts the number of retired μops that encountered a cache miss) will increment only once for that μop. A "Miss Retired" metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the Intel® 64 and IA-32 Architectures Optimization Reference Manual (see Section 1.4, "Related Literature").
- **Replay —** To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules μops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied, μops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of μops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist —** When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of μops before they begin and are costly.

20.6.3.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and μops that encountered a specified event. For a subset of the at-retirement events, a μop may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count μops tagged with that mechanism:

- **Front-end tagging —** This mechanism pertains to the tagging of μops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the Front_end_event event.
- **Execution tagging —** This mechanism pertains to the tagging of μops that encountered execution events (for example, instruction types) and are counted with the Execution_Event event.

- **Replay tagging —** This mechanism pertains to tagging of μops whose retirement is replayed (for example, a cache miss) and are counted with the Replay_event event. Branch mispredictions are also tagged with this mechanism.
- **No tags —** This mechanism does not use tags. It uses the Instr_retired and the Uops_ retired events.

Each tagging mechanism is independent from all others; that is, a μop that has been tagged using one mechanism will not be detected with another mechanism's tagged-μop detector. For example, if μops are tagged using the front-end tagging mechanisms, the Replay_event will not count those as tagged μops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of μops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of μops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

There are performance monitoring events that support at-retirement counting: specifically the Front_end_event, Execution_event, Replay_event, Inst_retired, and Uops_retired events. The following sections describe the tagging mechanisms for using these events to tag μop and count tagged μops.

20.6.3.6.2 Tagging Mechanism for Front_end_event

The Front_end_event counts μops that have been tagged as encountering any of the following events:

- **μop decode events —** Tagging μops for μop decode events requires specifying bits in the ESCR associated with the performance-monitoring event, Uop_type.
- **Trace cache events —** Tagging μops for trace cache events may require specifying certain bits in the MSR_TC_PRECISE_EVENT MSR.

The MSRs that are supported by the front-end tagging mechanism must be set and one or both of the NBOGUS and BOGUS bits in the Front_end_event event mask must be set to count events. None of the events currently supported requires the use of the MSR_TC_PRECISE_EVENT MSR.

20.6.3.6.3 Tagging Mechanism For Execution_event

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* ESCR is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* ESCR is used to detect μops that have been tagged with that tag value identifier using Execution_event for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular μop. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of NBOGUS0 should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged μops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. The tag enable and tag value bits are irrelevant for the downstream ESCR used to select the Execution_event.

The four separate tag bits allow the user to simultaneously but distinctly count up to four execution events at retirement. (This applies for interrupt-based event sampling. There are additional restrictions for PEBS as noted in Section 20.6.3.8.3, “Setting Up the PEBS Buffer.”) It is also possible to detect or count combinations of events by setting multiple tag value bits in the upstream ESCR or multiple mask bits in the downstream ESCR. For example, use a tag value of 3H in the upstream ESCR and use NBOGUS0/NBOGUS1 in the downstream ESCR event mask.

20.6.3.7 Tagging Mechanism for Replay_event

The replay mechanism enables tagging of μops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of μop that may experience the replay in the MSR_PEBS_MATRIX_VERT MSR and selecting the type of event in the MSR_PEBS_ENABLE MSR. Replay tagging must also be enabled with the UOP_Tag flag (bit 24) in the MSR_PEBS_ENABLE MSR.

The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 18.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA_32_PEBS_ENABLE_MSR. Each of these metrics requires that the Replay_Event be used to count the tagged μops.

20.6.3.8 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 18.4.5, "Branch Trace Store (BTS)," for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 18.4.9, "BTS and DS Save Area"). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and EFLAGS registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can only be carried out using the one performance counter, the MSR_IQ_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32_PMC0 and IA32_PERFEVTSEL0 MSRs (See Section 20.6.2.4).

20.6.3.8.1 Detection of the Availability of the PEBS Facilities

The DS feature flag (bit 21) returned by the CPUID instruction indicates (when set) the availability of the DS mechanism in the processor, which supports the PEBS (and BTS) facilities. When this bit is set, the following PEBS facilities are available:

- The PEBS_UNAVAILABLE flag in the IA32_MISC_ENABLE MSR indicates (when clear) the availability of the PEBS facilities, including the MSR_PEBS_ENABLE MSR.
- The enable PEBS flag (bit 24) in the MSR_PEBS_ENABLE MSR allows PEBS to be enabled (set) or disabled (clear).
- The IA32_DS_AREA MSR can be programmed to point to the DS save area.

20.6.3.8.2 Setting Up the DS Save Area

Section 18.4.9.2, "Setting Up the DS Save Area," describes how to set up and enable the DS save area. This procedure is common for PEBS and BTS.

20.6.3.8.3 Setting Up the PEBS Buffer

Only the MSR_IQ_COUNTER4 performance counter can be used for PEBS. Use the following procedure to set up the processor and this counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, and precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area (see Figure 18-5) to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS flag (bit 24) in MSR_PEBS_ENABLE MSR.
3. Set up the MSR_IQ_COUNTER4 performance counter and its associated CCCR and one or more ESCRs for PEBS.

20.6.3.8.4 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 18.4.9.5, “Writing the DS Interrupt Service Routine,” for guidelines for writing the DS ISR.

20.6.3.8.5 Other DS Mechanism Implications

The DS mechanism is not available in the SMM. It is disabled on transition to the SMM mode. Similarly the DS mechanism is disabled on the generation of a machine check exception and is cleared on processor RESET and INIT.

The DS mechanism is available in real address mode.

20.6.3.9 Operating System Implications

The DS mechanism can be used by the operating system as a debugging extension to facilitate failure analysis. When using this facility, a 25 to 30 times slowdown can be expected due to the effects of the trace store occurring on every taken branch.

Depending upon intended usage, the instruction pointers that are part of the branch records or the PEBS records need to have an association with the corresponding process. One solution requires the ability for the DS specific operating system module to be chained to the context switch. A separate buffer can then be maintained for each process of interest and the MSR pointing to the configuration area saved and setup appropriately on each context switch.

If the BTS facility has been enabled, then it must be disabled and state stored on transition of the system to a sleep state in which processor context is lost. The state must be restored on return from the sleep state.

It is required that an interrupt gate be used for the DS interrupt as opposed to a trap gate to prevent the generation of an endless interrupt loop.

Pages that contain buffers must have mappings to the same physical address for all processes/logical processors, such that any change to CR3 will not change DS addresses. If this requirement cannot be satisfied (that is, the feature is enabled on a per thread/process basis), then the operating system must ensure that the feature is enabled/disabled appropriately in the context switch code.

20.6.4 Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

The performance monitoring capability of processors based on Intel NetBurst microarchitecture and supporting Intel Hyper-Threading Technology is similar to that described in Section 20.6.3. However, the capability is extended so that:

- Performance counters can be programmed to select events qualified by logical processor IDs.
- Performance monitoring interrupts can be directed to a specific logical processor within the physical processor.

The sections below describe performance counters, event qualification by logical processor ID, and special purpose bits in ESCRs/CCCRs. They also describe MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_TC_PRECISE_EVENT.

20.6.4.1 ESCR MSRs

Figure 20-51 shows the layout of an ESCR MSR in processors supporting Intel Hyper-Threading Technology.

The functions of the flags and fields are as follows:

- **T1_USR flag, bit 0 —** When set, events are counted when thread 1 (logical processor 1) is executing at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.

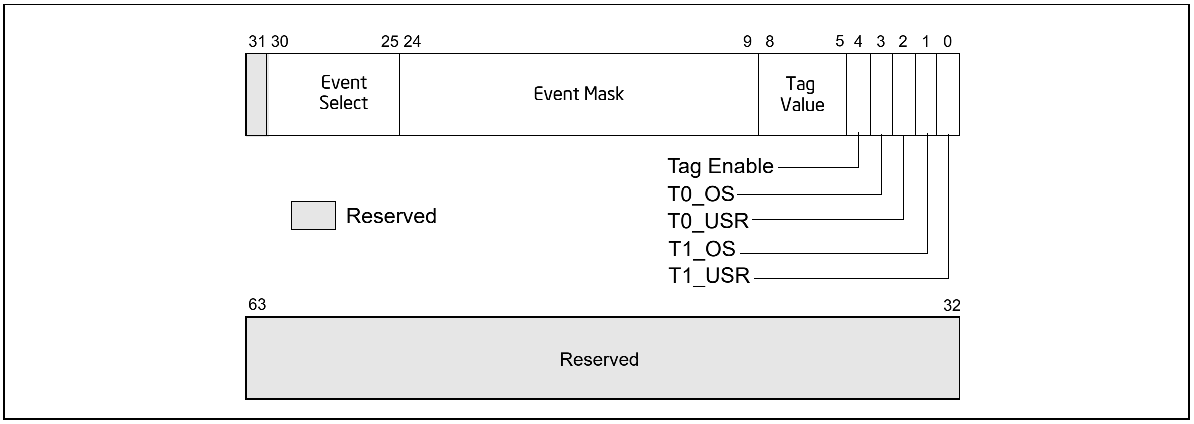


Figure 20-51. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel® Xeon® Processor, and Intel® Xeon® Processor MP Supporting Hyper-Threading Technology

- **T1_OS flag, bit 1 —** When set, events are counted when thread 1 (logical processor 1) is executing at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the T1_OS and T1_USR flags are set, thread 1 events are counted at all privilege levels.)
- **T0_USR flag, bit 2 —** When set, events are counted when thread 0 (logical processor 0) is executing at a CPL of 1, 2, or 3.
- **T0_OS flag, bit 3 —** When set, events are counted when thread 0 (logical processor 0) is executing at CPL of 0. (When both the T0_OS and T0_USR flags are set, thread 0 events are counted at all privilege levels.)
- **Tag enable, bit 4 —** When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 20.6.3.6, “At-Retirement Counting.”
- **Tag value field, bits 5 through 8 —** Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0_OS and T0_USR flags and the T1_OS and T1_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 9.4.5, “Identifying Logical Processors in an MP System,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 20.6.4.4, “Performance Monitoring Events”). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

20.6.4.2 CCCR MSRs

Figure 20-52 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

- **Enable flag, bit 12 —** When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset
- **ESCR select field, bits 13 through 15 —** Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Active thread field, bits 16 and 17 —** Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:

  00 — None. Count only when neither logical processor is active.

01 — Single. Count only when one logical processor is active (either 0 or 1).

10 — Both. Count only when both logical processors are active.

11 — Any. Count when either logical processor is active.

A halted logical processor or a logical processor in the "wait for SIPI" state is considered inactive.

- **Compare flag, bit 18 —** When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.

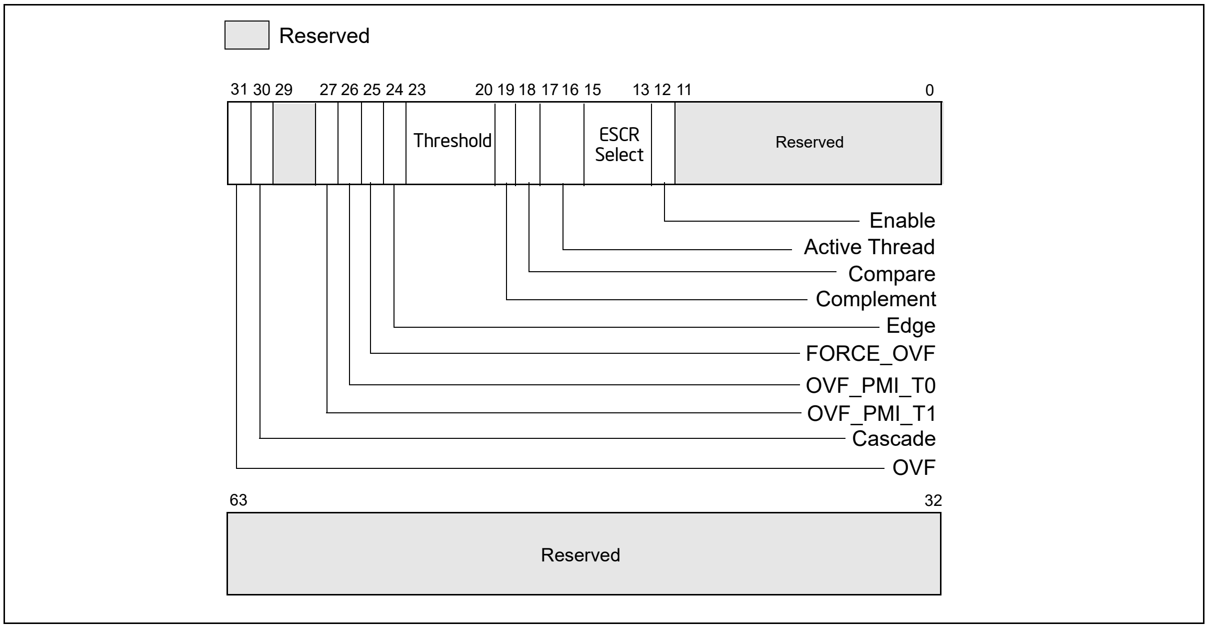


Figure 20-52. Counter Configuration Control Register (CCCR)

- **Complement flag, bit 19 —** Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 20.6.3.5.2, "Filtering Events"). The compare flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23 —** Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 20.6.3.5.2, "Filtering Events").
- **Edge flag, bit 24 —** When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
- **FORCE_OVF flag, bit 25 —** When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI_T0 flag, bit 26 —** When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.
- **OVF_PMI_T1 flag, bit 27 —** When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.
- **Cascade flag, bit 30 —** When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 20.6.3.2, "Performance Counters," for further details); when clear, disables cascading of counters.

- **OVF flag, bit 31 —** Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

20.6.4.3 IA32_PEBS_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR_PEBS_ENABLE MSR: bit 25 (ENABLE_PEBS_MY_THR) and 26 (ENABLE_PEBS_OTH_THR) respectively. These bits do not explicitly identify a specific logical processor by logic processor ID(T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running ("my thread") or for the other logical processor in the physical package on which the agent is not running ("other thread").

PEBS is supported for only a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can be carried out only with two performance counters: MSR_IQ_CCCR4 (MSR address 370H) for logical processor 0 and MSR_IQ_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE_PEBS_MY_THR and ENABLE_PEBS_OTH_THR bits in the MSR_PEBS_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

20.6.4.4 Performance Monitoring Events

When Intel Hyper-Threading Technology is active, many performance monitoring events can be can be qualified by the logical processor ID, which corresponds to bit 0 of the initial APIC ID. This allows for counting an event in any or all of the logical processors. However, not all the events have this logic processor specificity, or thread specificity.

Here, each event falls into one of two categories:

- **Thread specific (TS) —** The event can be qualified as occurring on a specific logical processor.
- **Thread independent (TI) —** The event cannot be qualified as being associated with a specific logical processor.

If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 20-89) depends only on the setting of the T0_USR and T0_OS flags in the ESCR being used to set up the event counter. The T1_USR and T1_OS flags have no effect on the count.

Table 20-89. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while T1 in USR	Counts while T1 in OS or USR	Counts while T1 in OS
T0_OS/T0_USR = 01	Counts while T0 in USR	Counts while T0 in USR or T1 in USR	Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 11	Counts while T0 in OS or USR	Counts while (a) T0 in OS or (b) T0 in USR or (c) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) or T0 in USR or (c) T1 in OS
T0_OS/T0_USR = 10	Counts T0 in OS	Counts T0 in OS or T1 in USR	Counts while (a)T0 in Os or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS

When a bit in the event mask field is TI, the effect of specifying bit-0-3 of the associated ESCR are described in Table 15-6. For events that are marked as TI, the effect of selectively specifying T0_USR, T0_OS, T1_USR, T1_OS bits is shown in Table 20-90.

Table 20-90. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 01	Counts while (a) T0 in USR or (b) T1 in USR	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 11	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 0	Counts while (a) T0 in OS or (b) T1 in OS	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS

20.6.4.5 Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

20.6.4.5.1 Non-Halted Clockticks

Use the following procedure to program ESCRs and CCCRs to obtain non-halted clockticks on processors based on Intel NetBurst microarchitecture:

1. Select an ESCR for the global_power_events and specify the RUNNING sub-event mask and the desired T0_OS/T0_USR/T1_OS/T1_USR bits for the targeted processor.
2. Select an appropriate counter.
3. Enable counting in the CCCR for that counter by setting the enable bit.

20.6.4.5.2 Non-Sleep Clockticks

Performance monitoring counters can be configured to count clockticks whenever the performance monitoring hardware is not powered-down. To count Non-sleep Clockticks with a performance-monitoring counter, do the following:

1. Select one of the 18 counters.
2. Select any of the ESCRs whose events the selected counter can count. Set its event select to anything other than "no_event"; the counter may be disabled if this is not done.
3. Turn threshold comparison on in the CCCR by setting the compare bit to "1".
4. Set the threshold to "15" and the complement to "1" in the CCCR. Since no event can exceed this threshold, the threshold condition is met every cycle and the counter counts every cycle. Note that this overrides any qualification (e.g., by CPL) specified in the ESCR.
5. Enable counting in the CCCR for the counter by setting the enable bit.

In most cases, the counts produced by the non-halted and non-sleep metrics are equivalent if the physical package supports one logical processor and is not placed in a power-saving state. Operating systems may execute an HLT instruction and place a physical processor in a power-saving state.

On processors that support Intel Hyper-Threading Technology (Intel HT Technology), each physical package can support two or more logical processors. Current implementation of Intel HT Technology provides two logical processors for each physical processor. While both logical processors can execute two threads simultaneously, one logical processor may halt to allow the other logical processor to execute without sharing execution resources between two logical processors.

Non-halted Clockticks can be set up to count the number of processor clock cycles for each logical processor whenever the logical processor is not halted (the count may include some portion of the clock cycles for that logical processor to complete a transition to a halted state). Physical processors that support Intel HT Technology enter into a power-saving state if all logical processors halt.

The Non-sleep Clockticks mechanism uses a filtering mechanism in CCCRs. The mechanism will continue to increment as long as one logical processor is not halted or in a power-saving state. Applications may cause a processor to enter into a power-saving state by using an OS service that transfers control to an OS's idle loop. The idle loop then may place the processor into a power-saving state after an implementation-dependent period if there is no work for the processor.

20.6.5 Performance Monitoring and Dual-Core Technology

The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

In the case of Pentium D processor, each logical processor is associated with dedicated resources for performance monitoring. In the case of Pentium processor Extreme edition, each processor core has dedicated resources, but two logical processors in the same core share performance monitoring resources (see Section 20.6.4, “Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”).

20.6.6 Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache

The 64-bit Intel Xeon processor MP with up to 8-MByte L3 cache has a CPUID signature of family [0FH], model [03H or 04H]. Performance monitoring capabilities available to Pentium 4 and Intel Xeon processors with the same values (see Section 20.1 and Section 20.6.4) apply to the 64-bit Intel Xeon processor MP with an L3 cache.

The level 3 cache is connected between the system bus and IOQ through additional control logic. See Figure 20-53.

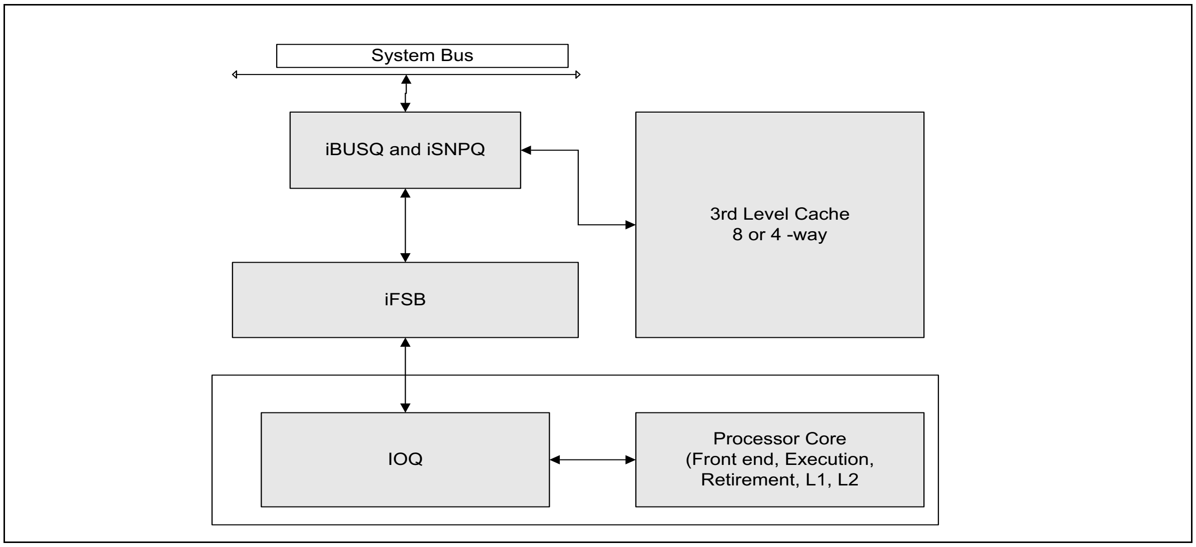


Figure 20-53. Block Diagram of 64-bit Intel® Xeon® Processor MP with 8-MByte L3

Additional performance monitoring capabilities and facilities unique to 64-bit Intel Xeon processor MP with an L3 cache are described in this section. The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs), each dedicated to a specific event. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values.

The lower 32-bits of the MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers. These performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:

- **IBUSQ event —** This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR_IFSB_IBUSQ0 and MSR_IFSB_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 20-54.

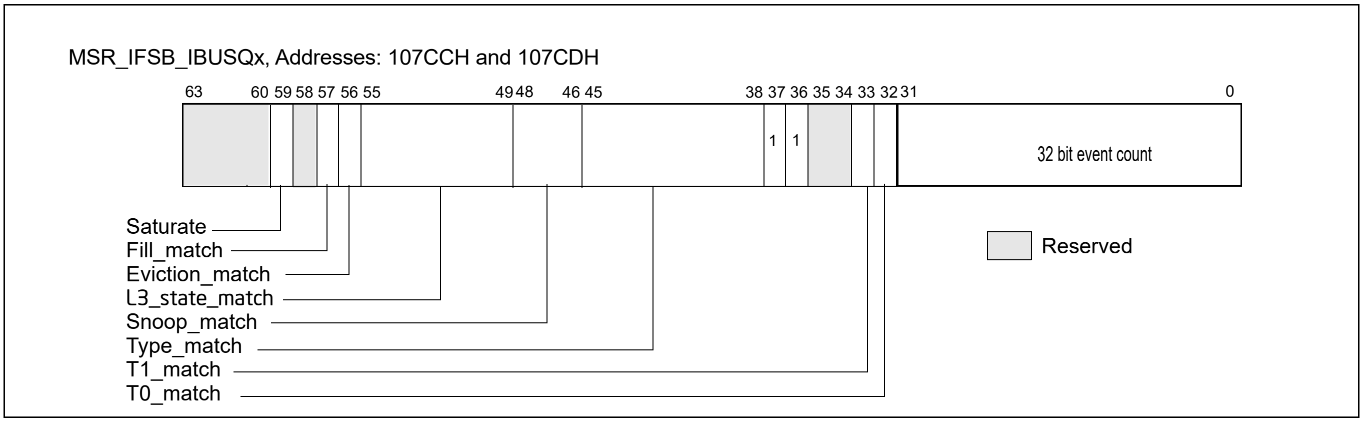


Figure 20-54. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH

- **ISNPQ event —** This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR_IFSB_ISNPQ0 and MSR_IFSB_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 20-55.

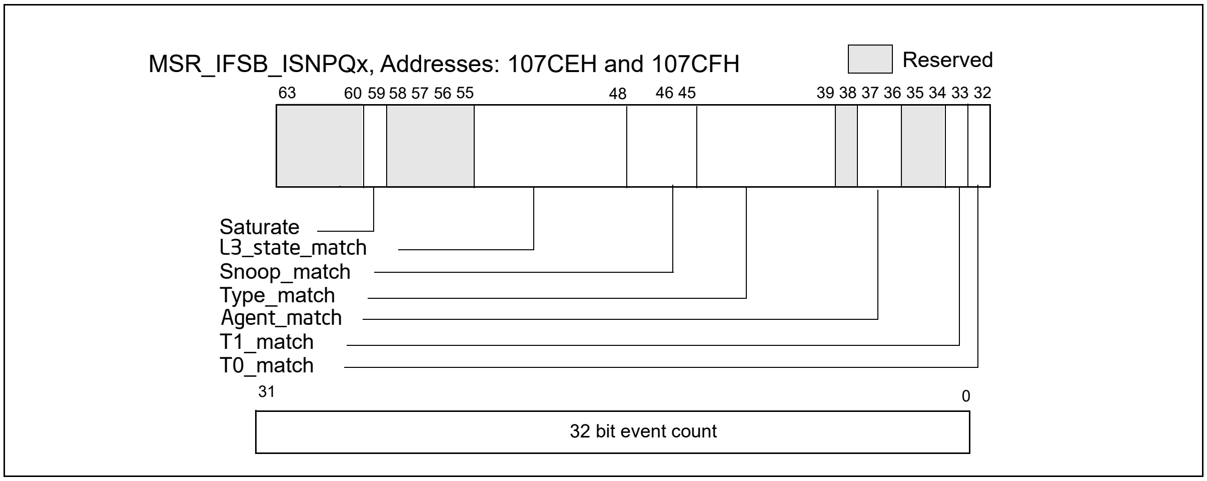


Figure 20-55. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH

- **EFSB event —** This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR_EFSB_DRDY0 and MSR_EFSB_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 20-56.

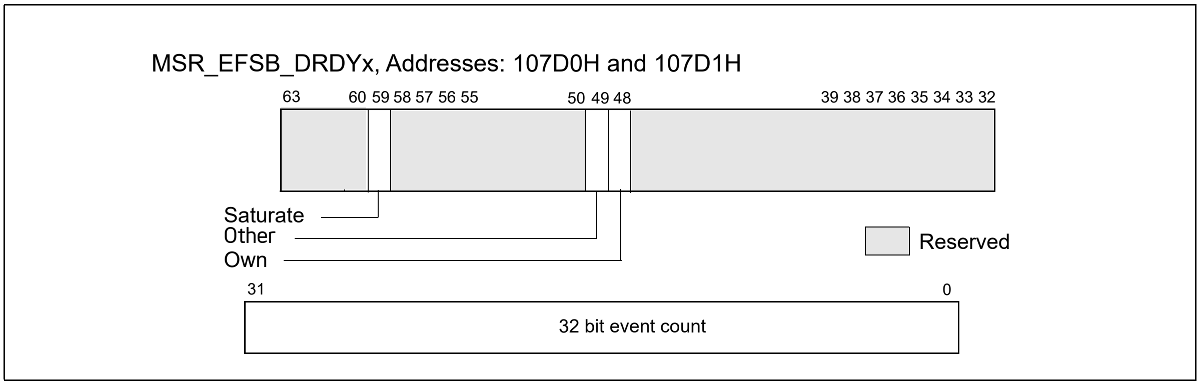


Figure 20-56. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H

- **IBUSQ Latency event —** This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR_IFSB_CTRL6[bit 26] to 1; the count freezes after software sets MSR_IFSB_CTRL6[bit 26] to 0. MSR_IFSB_CNTR7 acts as a 64-bit event counter for this event. See Figure 20-57.

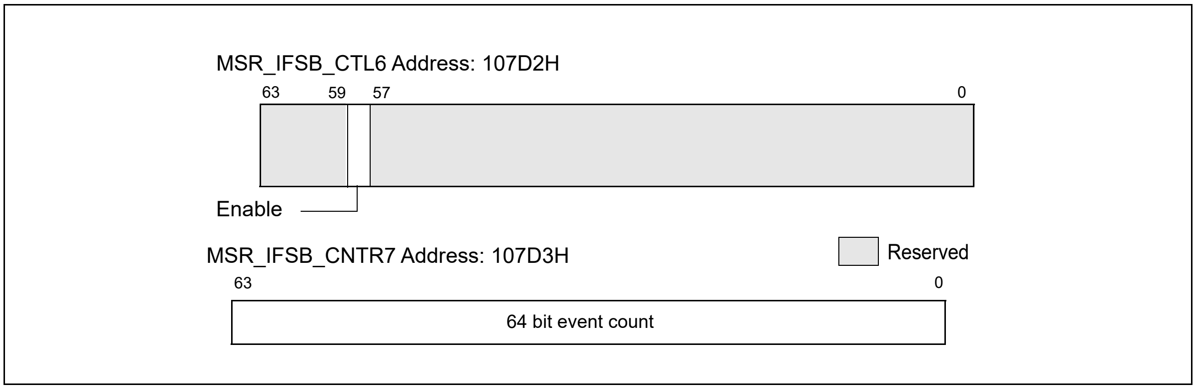


Figure 20-57. MSR_IFSB_CTL6, Address: 107D2H; MSR_IFSB_CNTR7, Address: 107D3H

20.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems

The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily_DisplayModel value of 06_1DH (see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through additional control logic. See Figure 20-58 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 20-58 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.

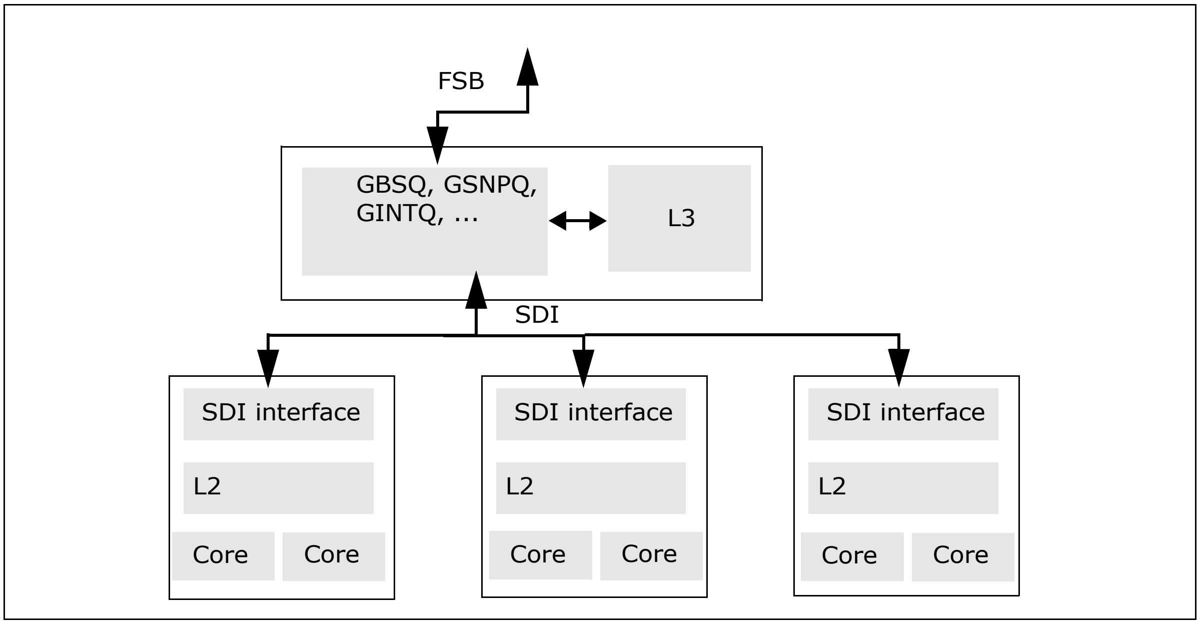


Figure 20-58. Block Diagram of the Intel® Xeon® Processor 7400 Series

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 20.1 and Section 20.6.4) apply to Intel Xeon processor 7100 series. The MSRs used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily_DisplayModel signature 06_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ_allocation and IOQ_active_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.

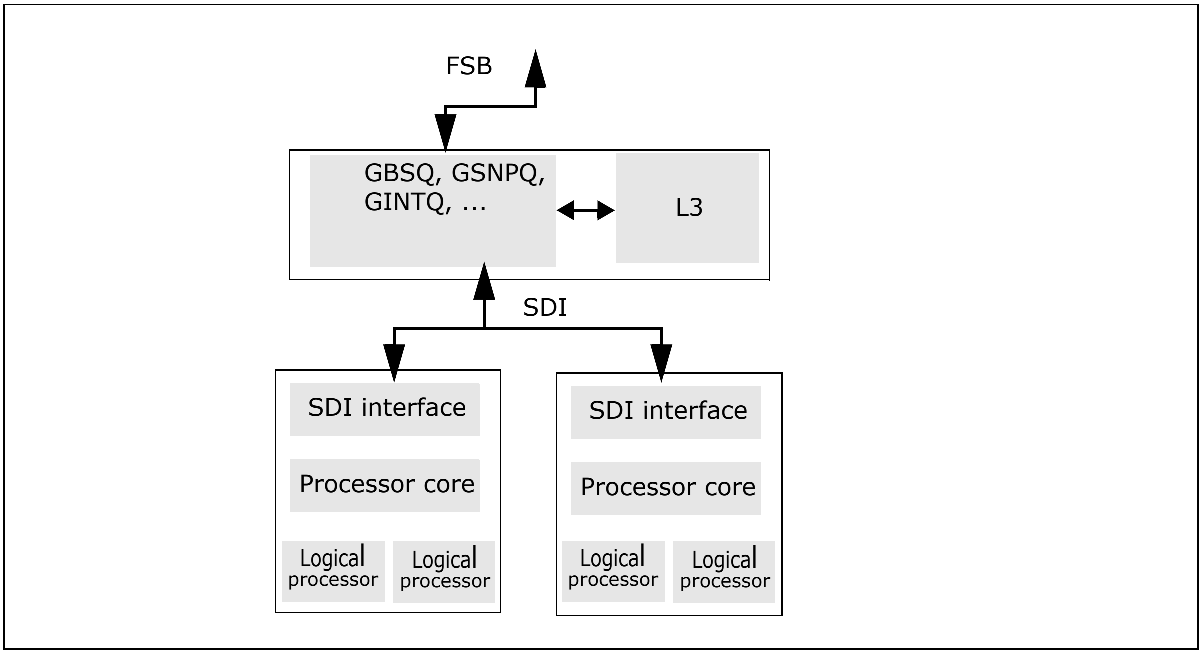


Figure 20-59. Block Diagram of the Intel® Xeon® Processor 7100 Series

20.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR_EMON_L3_GL_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3) are dedicated to counting GSNPQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Four MSRs (MSR_EMON_L3_CTR_CTL4, MSR_EMON_L3_CTR_CTL5, MSR_EMON_L3_CTR_CTL6, and MSR_EMON_L3_CTR_CTL7) are dedicated to counting external bus operations.

The bit fields in each of eight MSRs share the following common characteristics:

- Bits 63:32 is the event control field that includes an event mask and other bit fields that control counter operation. The event mask field specifies details of the microarchitectural condition, and its definition differs across GBSQ, GSNPQ, FSB.
- Bits 31:0 is the event count field. If the specified condition is met during each relevant clock domain of the event logic, the matched condition signals the counter logic to increment the associated event count field. The lower 32-bits of these 8 MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers.

In Dual-Core Intel Xeon processor 7100 series, the uncore performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

In Intel Xeon processor 7400 series, RDPMC with ECX between 2 and 9 can be used to access the eight uncore performance counter/control registers.

20.6.7.2 GBSQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1 is given in Figure 20-60. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following eight attributes:

- Agent_Select (bits 35:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, each bit specifies a logical processor in the physical package. The lower two bits corresponds to two logical processors in the first processor core, the upper two bits corresponds to two logical processors in the second processor core. 0FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each bit of [34:32] specifies the SDI logic of a dual-core module as the originator of the transaction. A value of 0111B in bits [35:32] specifies transaction from any processor core.

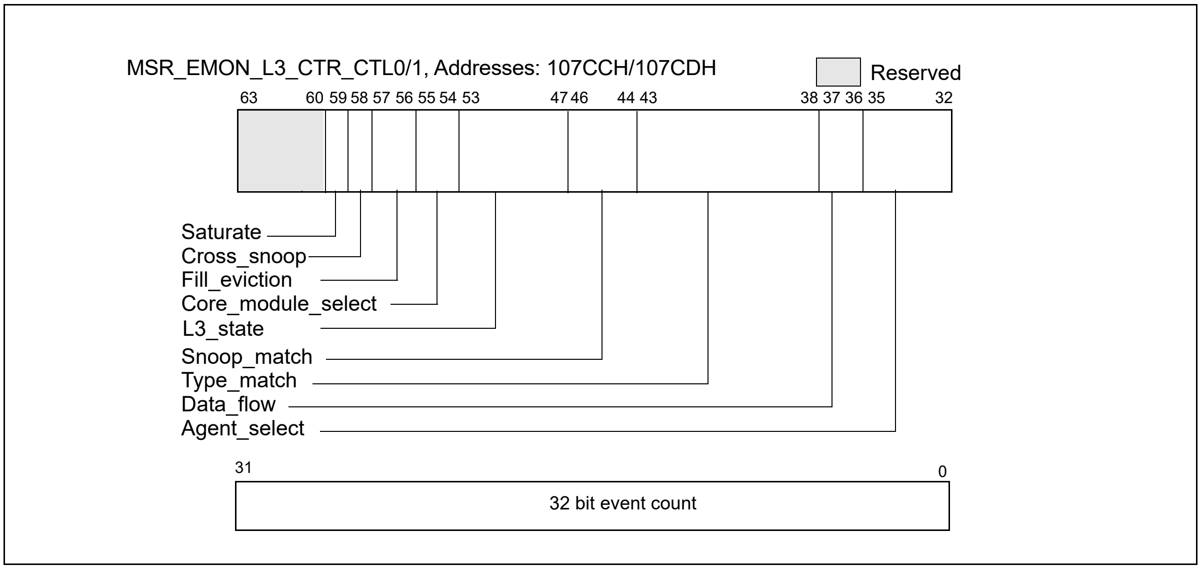


Figure 20-60. MSR_EMON_L3_CTR_CTL0/1, Addresses: 107CCH/107CDH

- Data_Flow (bits 37:36): Bit 36 specifies demand transactions, bit 37 specifies prefetch transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include all transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L3_State (bits 53:47): Each bit specifies an L2 coherency state.
- Core_Module_Select (bits 55:54): The valid encodings for L3 lookup differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series,

  — 00B: Match transactions from any core in the physical package

  — 01B: Match transactions from this core only

  — 10B: Match transactions from the other core in the physical package

  — 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series,

  — 00B: Match transactions from any dual-core module in the physical package

  — 01B: Match transactions from this dual-core module only

  — 10B: Match transactions from either one of the other two dual-core modules in the physical package

  - 11B: Match transaction from more than one dual-core modules in the physical package
- Fill_Eviction (bits 57:56): The valid encodings are
  - 00B: Match any transactions
  - 01B: Match transactions that fill L3
  - 10B: Match transactions that fill L3 without an eviction
  - 11B: Match transaction fill L3 with an eviction
- Cross_Snoop (bit 58): The encodings are
  - 0B: Match any transactions
  - 1B: Match cross snoop transactions

For each counting clock domain, if all eight attributes match, event logic signals to increment the event count field.

20.6.7.3 GSNPQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3 is given in Figure 20-61. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following six attributes:

- Agent_Select (bits 37:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
- For Intel Xeon processor 7100 series, each of the lowest 4 bits specifies a logical processor in the physical package. The lowest two bits corresponds to two logical processors in the first processor core, the next two bits corresponds to two logical processors in the second processor core. Bit 36 specifies other symmetric agent transactions. Bit 37 specifies central agent transactions. 3FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each of the lowest 3 bits specifies a dual-core module in the physical package. Bit 37 specifies central agent transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include any transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L2_State (bits 53:47): Each bit specifies an L3 coherency state.
- Core_Module_Select (bits 56:54): Bit 56 enables Core_Module_Select matching. If bit 56 is clear, Core_Module_Select encoding is ignored. The valid encodings for the lower two bits (bit 55, 54) differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one core (irrespective which core) in the physical package
  - 01B: Match transactions from this core and not the other core
  - 10B: Match transactions from the other core in the physical package, but not this core
  - 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one dual-core module (irrespective which module) in the physical package.
  - 01B: Match transactions from one or more dual-core modules.
  - 10B: Match transactions from two or more dual-core modules.
  - 11B: Match transaction from all three dual-core modules in the physical package.
- Block_Snoop (bit 57): specifies blocked snoop.

For each counting clock domain, if all six attributes match, event logic signals to increment the event count field.

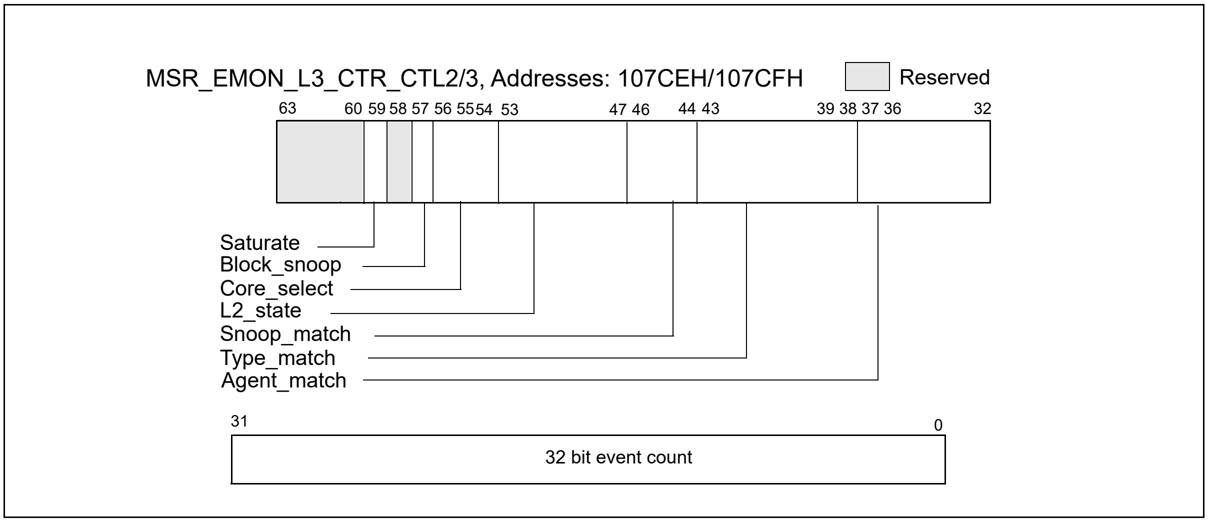


Figure 20-61. MSR_EMON_L3_CTR_CTL2/3, Addresses: 107CEH/107CFH

20.6.7.4 FSB Event Interface

The layout of MSR_EMON_L3_CTR_CTL4 through MSR_EMON_L3_CTR_CTL7 is given in Figure 20-62. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) is organized as follows:

- Bit 58: must set to 1.
- FSB_Submask (bits 57:32): Specifies FSB-specific sub-event mask.

The FSB sub-event mask defines a set of independent attributes. The event logic signals to increment the associated event count field if one of the attribute matches. Some of the sub-event mask bit counts durations. A duration event increments at most once per cycle.

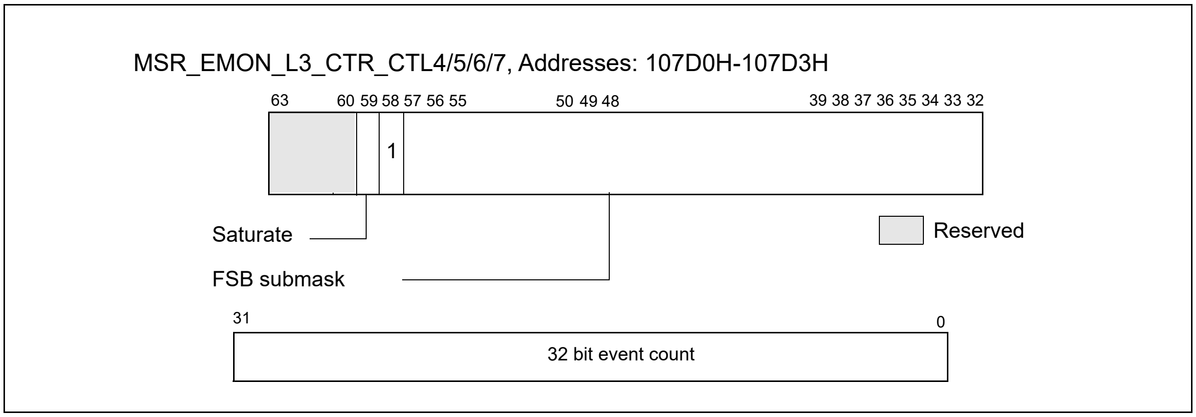


Figure 20-62. MSR_EMON_L3_CTR_CTL4/5/6/7, Addresses: 107D0H-107D3H

20.6.7.4.1 FSB Sub-Event Mask Interface

- FSB_type (bit 37:32): Specifies different FSB transaction types originated from this physical package.
- FSB_L_clear (bit 38): Count clean snoop results from any source for transaction originated from this physical package.
- FSB_L_hit (bit 39): Count HIT snoop results from any source for transaction originated from this physical package.

- FSB_L_hitm (bit 40): Count HITM snoop results from any source for transaction originated from this physical package.
- FSB_L_defer (bit 41): Count DEFER responses to this processor’s transactions.
- FSB_L_retry (bit 42): Count RETRY responses to this processor’s transactions.
- FSB_L_snoop_stall (bit 43): Count snoop stalls to this processor’s transactions.
- FSB_DBSY (bit 44): Count DBSY assertions by this processor (without a concurrent DRDY).
- FSB_DRDY (bit 45): Count DRDY assertions by this processor.
- FSB_BNR (bit 46): Count BNR assertions by this processor.
- FSB_IOQ_empty (bit 47): Counts each bus clocks when the IOQ is empty.
- FSB_IOQ_full (bit 48): Counts each bus clocks when the IOQ is full.
- FSB_IOQ_active (bit 49): Counts each bus clocks when there is at least one entry in the IOQ.
- FSB_WW_data (bit 50): Counts back-to-back write transaction’s data phase.
- FSB_WW_issue (bit 51): Counts back-to-back write transaction request pairs issued by this processor.
- FSB_WR_issue (bit 52): Counts back-to-back write-read transaction request pairs issued by this processor.
- FSB_RW_issue (bit 53): Counts back-to-back read-write transaction request pairs issued by this processor.
- FSB_other_DBSY (bit 54): Count DBSY assertions by another agent (without a concurrent DRDY).
- FSB_other_DRDY (bit 55): Count DRDY assertions by another agent.
- FSB_other_snoop_stall (bit 56): Count snoop stalls on the FSB due to another agent.
- FSB_other_BNR (bit 57): Count BNR assertions from another agent.

20.6.7.5 Common Event Control Interface

The MSR_EMON_L3_GL_CTL MSR provides simplified access to query overflow status of the GBSQ, GSNPQ, FSB event counters. It also provides control bit fields to freeze, unfreeze, or reset those counters. The following bit fields are supported:

- GL_freeze_cmd (bit 0): Freeze the event counters specified by the GL_event_select field.
- GL_unfreeze_cmd (bit 1): Unfreeze the event counters specified by the GL_event_select field.
- GL_reset_cmd (bit 2): Clear the event count field of the event counters specified by the GL_event_select field. The event select field is not affected.
- GL_event_select (bit 23:16): Selects one or more event counters to subject to specified command operations indicated by bits 2:0. Bit 16 corresponds to MSR_EMON_L3_CTR_CTL0, bit 23 corresponds to MSR_EMON_L3_CTR_CTL7.
- GL_event_status (bit 55:48): Indicates the overflow status of each event counters. Bit 48 corresponds to MSR_EMON_L3_CTR_CTL0, bit 55 corresponds to MSR_EMON_L3_CTR_CTL7.

In the event control field (bits 63:32) of each MSR, if the saturate control (bit 59, see Figure 20-60 for example) is set, the event logic forces the value FFFF_FFFFH into the event count field instead of incrementing it.

20.6.8 Performance Monitoring (P6 Family Processor)

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These can either count events or measure duration. When counting events, a counter increments each time a specified event takes place or a specified number of events takes place. When measuring duration, it counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.

NOTE

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The performance-monitoring counters are supported by four MSRs: the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

NOTE

The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed for P6 family processors are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

20.6.8.1 PerfEvtSel0 and PerfEvtSel1 MSRs

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Figure 20-63 shows the flags and fields in these MSRs.

The functions of the flags and fields in the PerfEvtSel0 and PerfEvtSel1 MSRs are as follows:

- **Event select field (bits 0 through 7) —** Selects the event logic unit to detect certain microarchitectural conditions.
- **Unit mask (UMASK) field (bits 8 through 15) —** Further qualifies the event logic unit selected in the event select field to detect a specific microarchitectural condition. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states.

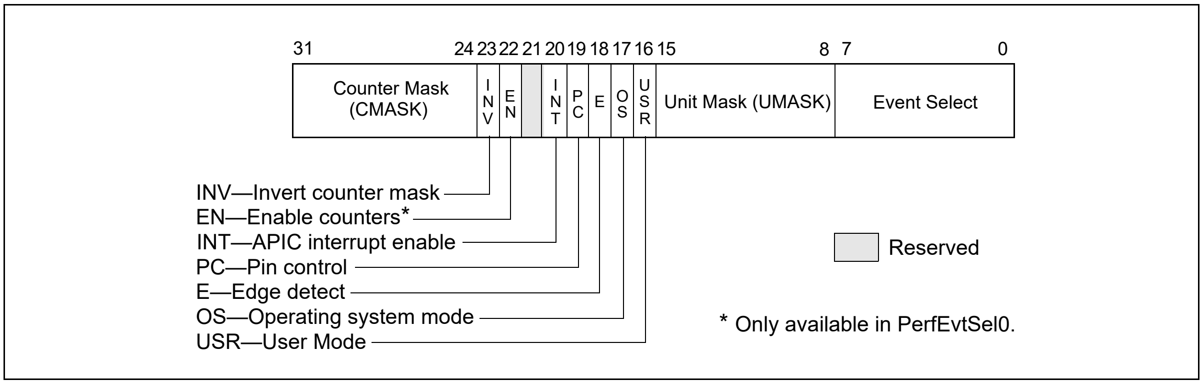


Figure 20-63. PerfEvtSel0 and PerfEvtSel1 MSRs

- **USR (user mode) flag (bit 16) —** Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.
- **OS (operating system mode) flag (bit 17) —** Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.
- **E (edge detect) flag (bit 18) —** Enables (when set) edge detection of events. The processor counts the number of deasserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19) —** When set, the processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20) —** When set, the processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31) —** When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

20.6.8.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value, and the high-order 8 bits are sign-extended according to the value of bit 31. This operation allows writing both positive and negative values to the performance counters.

20.6.8.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

20.6.8.4 Event and Time-Stamp Monitoring Software

To use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking.
- Initialize and start counters.
- Stop counters.
- Read the event counters.
- Read the time-stamp counter.

The event monitor feature determination procedure must check whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. (The Pentium and P6 family processors support performance counters.) The procedure also checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialize and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be counted and the method used to count them and initializes the counter MSRs (PerfCtr0 and PerfCtr1) to starting counts. The stop counters procedure stops the performance counters (see Section 20.6.8.3, “Starting and Stopping the Performance-Monitoring Counters”).

The read counters procedure reads the values in the PerfCtr0 and PerfCtr1 MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures would be provided in lieu of enabling the RDTSC and RDPMC instructions that allow application code to read the counters.

20.6.8.5 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

20.6.9 Performance Monitoring (Pentium Processors)

The Pentium processor provides two 40-bit performance counters, which can be used to count events or measure duration. The counters are supported by three MSRs: the control and event select MSR (CESR) and the performance counter MSRs (CTR0 and CTR1). These can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0.

Each counter has an associated external pin (PM0/BP0 and PM1/BP1), which can be used to indicate the state of the counter to external hardware.

NOTES

The CESR, CTR0, and CTR1 MSRs and the events listed for Pentium processors are model-specific for the Pentium processor.

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

20.6.9.1 Control and Event Select Register (CESR)

The 32-bit control and event select MSR (CESR) controls the operation of performance-monitoring counters CTR0 and CTR1 and the associated pins (see Figure 20-64). To control each counter, the CESR register contains a 6-bit event select field (ES0 and ES1), a pin control flag (PC0 and PC1), and a 3-bit counter control field (CC0 and CC1). The functions of these fields are as follows:

- **ES0 and ES1 (event select) fields (bits 0-5, bits 16-21) —** Selects (by entering an event code in the field) up to two events to be monitored.

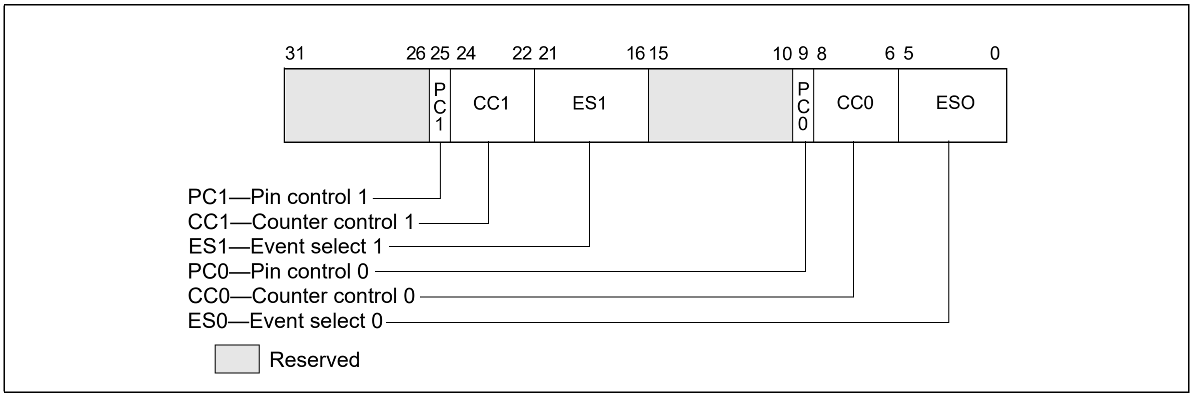


Figure 20-64. CESR MSR (Pentium Processor Only)

- **CC0 and CC1 (counter control) fields (bits 6-8, bits 22-24) —** Controls the operation of the counter. Control codes are as follows:

  000 — Count nothing (counter disabled).

  001 — Count the selected event while CPL is 0, 1, or 2.

  010 — Count the selected event while CPL is 3.

  011 — Count the selected event regardless of CPL.

  100 — Count nothing (counter disabled).

  101 — Count clocks (duration) while CPL is 0, 1, or 2.

  110 — Count clocks (duration) while CPL is 3.

  111 — Count clocks (duration) regardless of CPL.

  The highest order bit selects between counting events and counting clocks (duration); the middle bit enables counting when the CPL is 3; and the low-order bit enables counting when the CPL is 0, 1, or 2.

- **PC0 and PC1 (pin control) flags (bits 9, 25) —** Selects the function of the external performance-monitoring counter pin (PM0/BP0 and PM1/BP1). Setting one of these flags to 1 causes the processor to assert its associated pin when the counter has overflowed; setting the flag to 0 causes the pin to be asserted when the counter has been incremented. These flags permit the pins to be individually programmed to indicate the overflow or incremented condition. The external signaling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

20.6.9.2 Use of the Performance-Monitoring Pins

When performance-monitor pins PM0/BP0 and/or PM1/BP1 are configured to indicate when the performance-monitor counter has incremented and an “occurrence event” is being counted, the associated pin is asserted (high) each time the event occurs. When a “duration event” is being counted, the associated PM pin is asserted for the entire duration of the event. When the performance-monitor pins are configured to indicate when the counter has overflowed, the associated PM pin is asserted when the counter has overflowed.

When the PM0/BP0 and/or PM1/BP1 pins are configured to signal that a counter has incremented, it should be noted that although the counters may increment by 1 or 2 in a single clock, the pins can only indicate that the event occurred. Moreover, since the internal clock frequency may be higher than the external clock frequency, a single external clock may correspond to multiple internal clocks.

A "count up to" function may be provided when the event pin is programmed to signal an overflow of the counter. Because the counters are 40 bits, a carry out of bit 39 indicates an overflow. A counter may be preset to a specific value less then $2^{40} – 1$. After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

20.6.9.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).
- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

20.7 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.
- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor's bus clock frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions.
- Enhanced Intel SpeedStep Technology transitions (P-state transitions).

For Intel processors that support TM2, the processor core clocks may operate at a frequency that differs from the Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See Section 20.7.2 for more detail.

Due to the above considerations there are several important clocks referenced in this manual:

- **Base Clock —** The frequency of this clock is the frequency of the processor when the processor is not in turbo mode, and not being throttled via Intel SpeedStep.
- **Maximum Clock —** This is the maximum frequency of the processor when turbo mode is at the highest point.
- **Bus Clock —** These clockticks increment at a fixed frequency and help coordinate the bus on some systems.

- **Core Crystal Clock —** This is a clock that runs at fixed frequency; it coordinates the clocks on all packages across the system.
- **Non-halted Clockticks —** Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.
- **Non-sleep Clockticks —** Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.
- **Time-stamp Counter —** See Section 18.17, "Time-Stamp Counter."
- **Reference Clockticks —** TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 18.17, "Time-Stamp Counter," for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI —** Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.
- **Nominal CPI —** Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

20.7.1 Non-Halted Reference Clockticks

Software can use UnHalted Reference Cycles on either a general purpose performance counter using event mask 0x3C and UMASK 0x01 or on fixed function performance counter 2 to count at a constant rate. These events count at a consistent rate irrespective of P-state, TM2, or frequency transitions that may occur to the processor. The UnHalted Reference Cycles event may count differently on the general purpose event and fixed counter.

20.7.2 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 15, "Power and Thermal Management"), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

- The time stamp counter operates at a fixed-rate frequency of the processor.
- The IA32_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.
- The IA32_FIXED_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.
- The Local APIC timer operation is unaffected by opportunistic processor operation.
- The TSC, IA32_MPERF, and IA32_FIXED_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.

20.7.3 Determining the Processor Base Frequency

For Intel processors in which the nominal core crystal clock frequency is enumerated in CPUID.15H.ECX and the core crystal clock ratio is encoded in CPUID.15H (see Table 3-8 "Information Returned by CPUID Instruction"), the nominal TSC frequency can be determined by using the following equation:

Nominal TSC frequency = (CPUID.15H.ECX[31:0] * CPUID.15H.EBX[31:0]) ÷ CPUID.15H.EAX[31:0]

For Intel processors in which CPUID.15H.EBX[31:0] ÷ CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 20-91 can be used to look up the nominal core crystal clock frequency.

Table 20-91. Nominal Core Crystal Clock Frequency

Processor Families/Processor Number Series[1]	Nominal Core Crystal Clock Frequency
Intel® Xeon® Scalable Processor Family with CPUID signature 06_55H.	25 MHz
6th and 7th generation Intel® Core™ processors and Intel® Xeon® W Processor Family.	24 MHz
Next Generation Intel Atom® processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors).	19.2 MHz

NOTES:

[1] For any processor in which CPUID.15H is enumerated and MSR_PLATFORM_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

20.7.3.1 For Intel® Processors Based on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell Microarchitectures

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 100 MHz.

20.7.3.2 For Intel® Processors Based on Nehalem Microarchitecture

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 133.33 MHz.

20.7.3.3 For Intel Atom® Processors Based on Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by the scalable bus frequency. The scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] for Intel Atom processors based on the Silvermont microarchitecture, and in bit field MSR_FSB_FREQ[3:0] for processors based on the Airmont microarchitecture; see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

20.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] at (0CDH), see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR_PLATFORM_ID[12:8]. It corresponds to the Processor Base frequency.

- IF XE operation is enabled, the maximum resolved bus ratio is given in MSR_PERF_STATUS[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR_PERF_STATUS[31] is set, XE operation is enabled. The MSR_PERF_STATUS[31] field is read-only.

20.8 IA32_PERF_CAPABILITIES MSR ENUMERATION

The layout of IA32_PERF_CAPABILITIES MSR is shown in Figure 20-65; it provides enumeration of a variety of interfaces:

- IA32_PERF_CAPABILITIES.LBR_FMT[bits 5:0]: encodes the LBR format, details are described in Section 18.4.8.1.
- IA32_PERF_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist; see Section 20.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers; see Section 20.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBS_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records; see Section 20.6.2.4.2.
- IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[12]: Indicates IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if 1, see Section 20.8.1.
- IA32_PERF_CAPABILITIES.FULL_WRITE[13]: Indicates the processor supports IA32_A_PMCx interface for updating bits 32 and above of IA32_PMCx; see Section 20.2.6.
- IA32_PERF_CAPABILITIES.PEBS_BASELINE [bit 14]: If set, the following is true:
  - The IA32_PEBS_ENABLE MSR (address 3F1H) exists and all architecturally enumerated fixed and general-purpose counters have corresponding bits in IA32_PEBS_ENABLE that enable generation of PEBS records. The general-purpose counter bits start at bit IA32_PEBS_ENABLE[0], and the fixed counter bits start at bit IA32_PEBS_ENABLE[32].
  - The format of the PEBS record is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT; see Section 20.6.2.4.2.
  - Extended PEBS is supported. All counters support the PEBS facility, and all events (both precise and non-precise) can generate PEBS records when PEBS is enabled for that counter. Note that not all events may be available on all counters.
  - Adaptive PEBS is supported. The PEBS_DATA_CFG MSR (address 3F2H) and adaptive record enable bits (IA32_PERFEVTSELx.Adaptive_Record and IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record) are supported. The definition of the PEBS_DATA_CFG MSR, including which bits are supported and how they affect the record, is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT; see Section 20.9.2.3.
  - NOTE: Software is recommended to feature PEBS Baseline when the following is true: IA32_PERF_CAPABILITIES.PEBS_BASELINE[14] && IA32_PERF_CAPABILITIES.PEBS_FMT[11:8] ≥ 4.
- IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE[15]: If set, indicates that the architecture provides built in support for TMA L1 metrics through the PERF_METRICS MSR, see Section 20.3.9.3.
- IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16]: If set on parts that enumerate support for Intel PT (CPUID.0x7.0.EBX[25]=1), setting IA32_PEBS_ENABLE.PEBS_OUTPUT to 01B will result in PEBS output being written into the Intel PT trace stream. See Section 20.5.5.2.

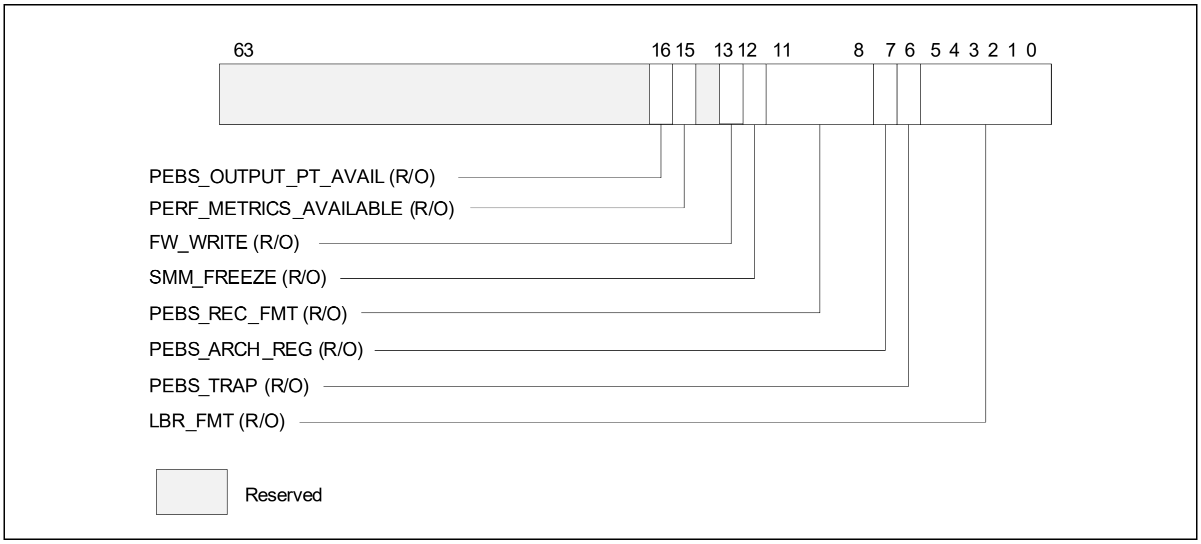


Figure 20-65. Layout of IA32_PERF_CAPABILITIES MSR

20.8.1 Filtering of SMM Handler Overhead

When performance monitoring facilities and/or branch profiling facilities (see Section 18.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel Atom® Processors)”) are enabled, these facilities capture event counts, branch records and branch trace messages occurring in a logical processor. The occurrence of interrupts, instruction streams due to various interrupt handlers all contribute to the results recorded by these facilities.

If CPUID.01H:ECX.PDCM[bit 15] is 1, the processor supports the IA32_PERF_CAPABILITIES MSR. If IA32_PERF_-CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is 1, the processor supports the ability for system software using performance monitoring and/or branch profiling facilities to filter out the effects of servicing system management interrupts.

If the FREEZE_WHILE_SMM capability is enabled on a logical processor and after an SMI is delivered, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler.

The enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored , after the SMI handler issues RSM to complete its servicing.

It is the responsibility of the SMM code to ensure the state of the performance monitoring and branch profiling facilities are preserved upon entry or until prior to exiting the SMM. If any of this state is modified due to actions by the SMM code, the SMM code is required to restore such state to the values present at entry to the SMM handler.

System software is allowed to set IA32_DEBUGCTL.FREEZE_WHILE_SMM[bit 14] to 1 only supported as indicated by IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] reporting 1.

20.9 PEBS FACILITY

20.9.1 Extended PEBS

- The Extended PEBS feature supports Processor Event Based Sampling (PEBS) on all counters, both fixed function and general purpose; and all performance monitoring events, both precise and non-precise. PEBS can be enabled for the general purpose counters using PEBS_EN_PMCi bits of IA32_PEBS_ENABLE (i = 0, 1,..m). PEBS can be enabled for 'i' fixed function counters using the PEBS_EN_FIXEDi bits of IA32_PEBS_ENABLE (i = 0, 1, ...n).

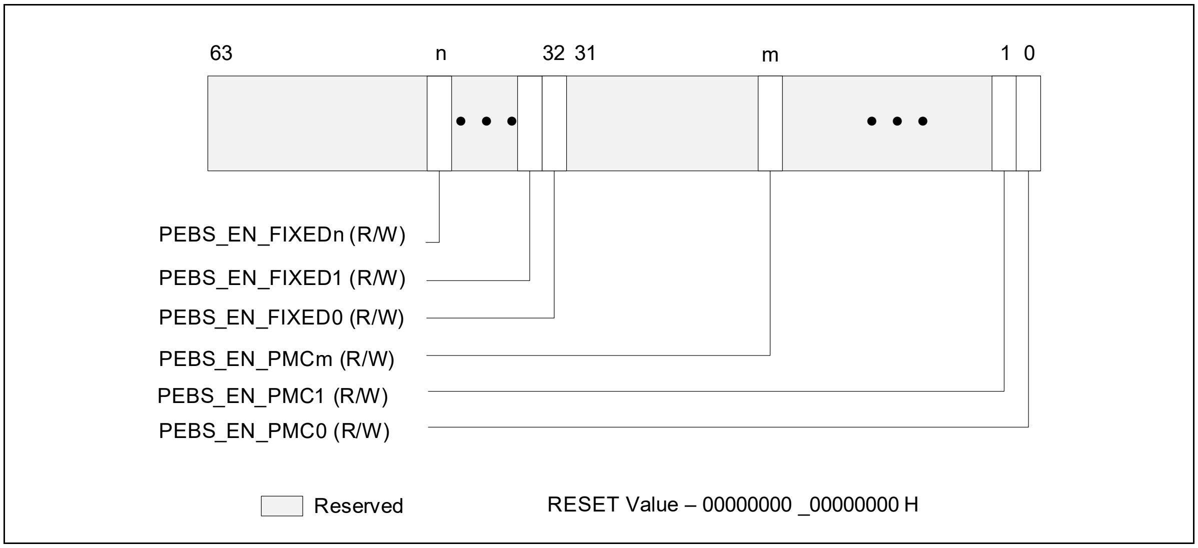


Figure 20-66. Layout of IA32_PEBS_ENABLE MSR

A PEBS record due to a precise event will be generated after an instruction that causes the event when the counter has already overflowed. A PEBS record due to a non-precise event will occur at the next opportunity after the counter has overflowed, including immediately after an overflow is set by an MSR write.

Currently, IA32_FIXED_CTR0 counts instructions retired and is a precise event. IA32_FIXED_CTR1, IA32_-FIXED_CTR2 … IA32_FIXED_CTRm count as non-precise events.

The Applicable Counter field in the Basic Info Group of the PEBS record indicates which counters caused the PEBS record to be generated. It is in the same format as the enable bits for each counter in IA32_PEBS_ENABLE. As an example, an Applicable Counter field with bits 2 and 32 set would indicate that both general purpose counter 2 and fixed function counter 0 generated the PEBS record.

- To properly use PEBS for the additional counters, software will need to set up the counter reset values in PEBS portion of the DS_BUFFER_MANAGEMENT_AREA data structure that is indicated by the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 20-67. When a counter generates a PEBS records, the appropriate counter reset values will be loaded into that counter. In the above example where general purpose counter 2 and fixed function counter 0 generated the PEBS record, general purpose counter 2 would be reloaded with the value contained in PEBS GP Counter 2 Reset (offset 50H) and fixed function counter 0 would be reloaded with the value contained in PEBS Fixed Counter 0 Reset (offset 80H).

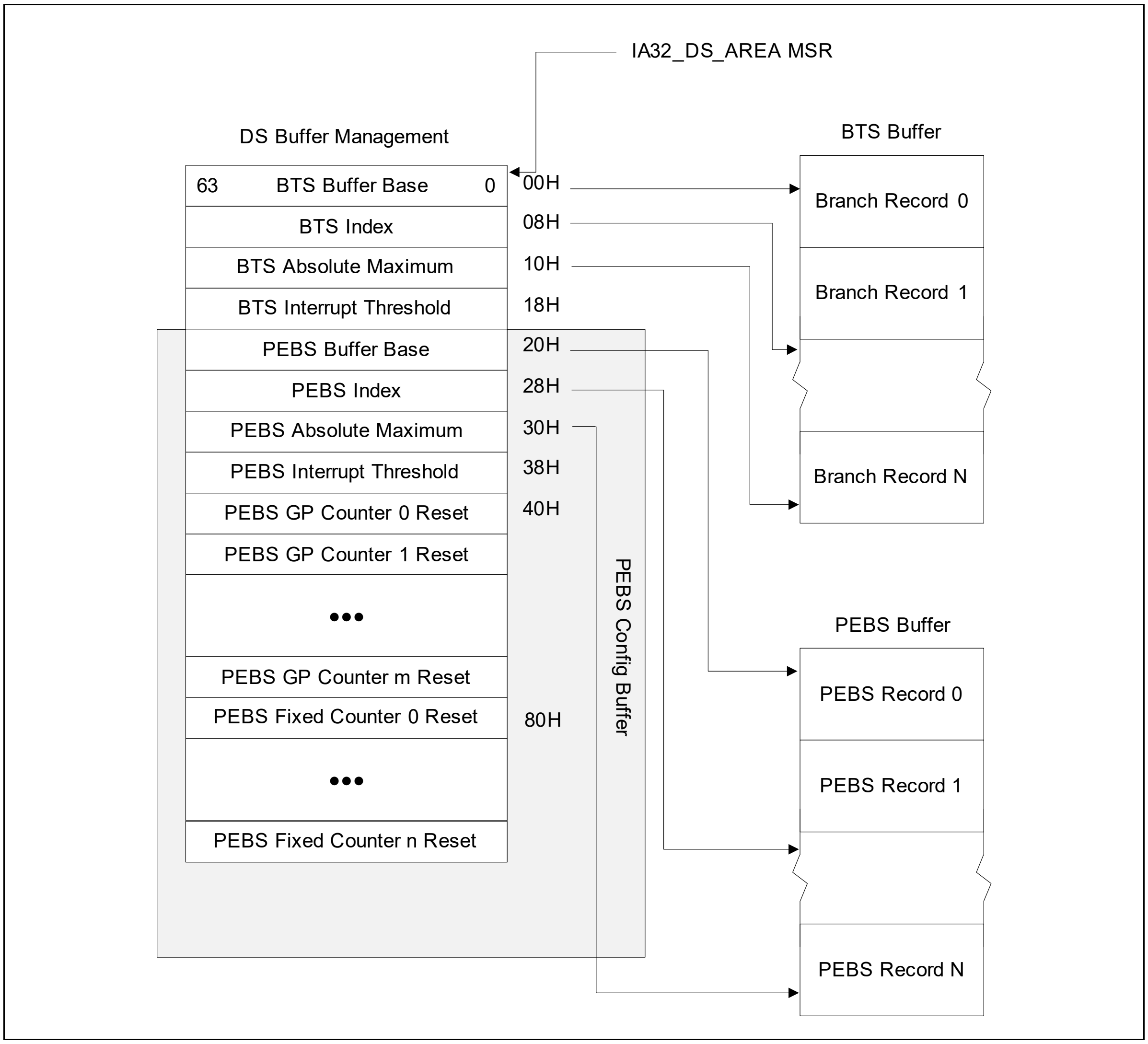


Figure 20-67. PEBS Programming Environment

Extended PEBS support debuts on Intel Atom® processors based on the Goldmont Plus microarchitecture and future Intel® Core™ processors based on the Ice Lake microarchitecture.

20.9.2 Adaptive PEBS

The PEBS facility has been enhanced to collect the following CPU state in addition to GPRs, EventingIP, TSC, and memory access related information collected by legacy PEBS:

- XMM registers
- LBR records (TO/FROM/INFO)

The PEBS record is restructured where fields are grouped into Basic group, Memory group, GPR group, XMM group, and LBR group. A new register MSR_PEBS_DATA_CFG provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.

By default, the PEBS record will only contain the Basic group. Optionally, each counter can be configured to generate a PEBS records with the groups specified in MSR_PEBS_DATA_CFG.

Details and examples for the Adaptive PEBS capability follow below.

20.9.2.1 Adaptive_Record Counter Control

- IA32_PERFEVTSELx.Adaptive_Record[34]: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_PMCx is set for the corresponding GP counter, an overflow of PMCx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.

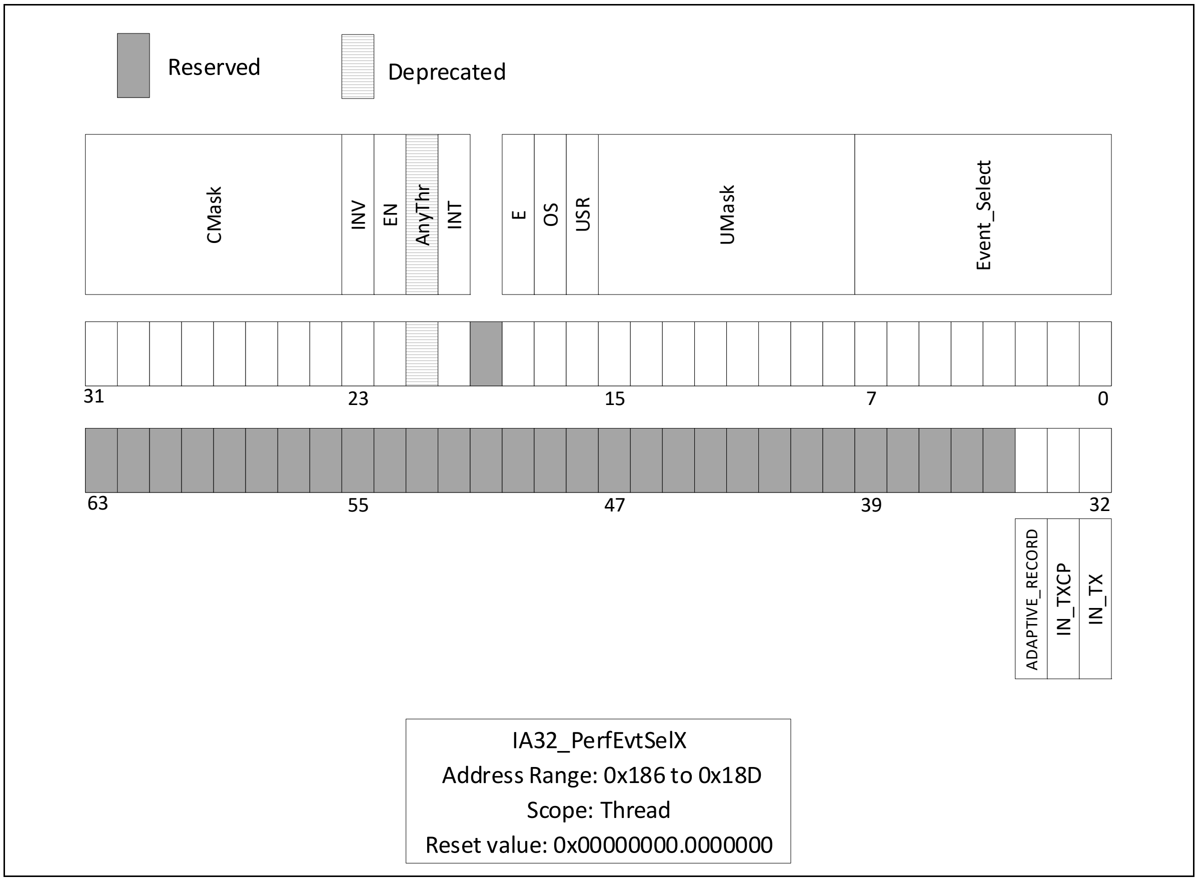


Figure 20-68. Layout of IA32_PerfEvtSelX MSR Supporting Adaptive PEBS

- IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_FIXEDx is set for the corresponding Fixed counter, an overflow of FixedCtrx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.

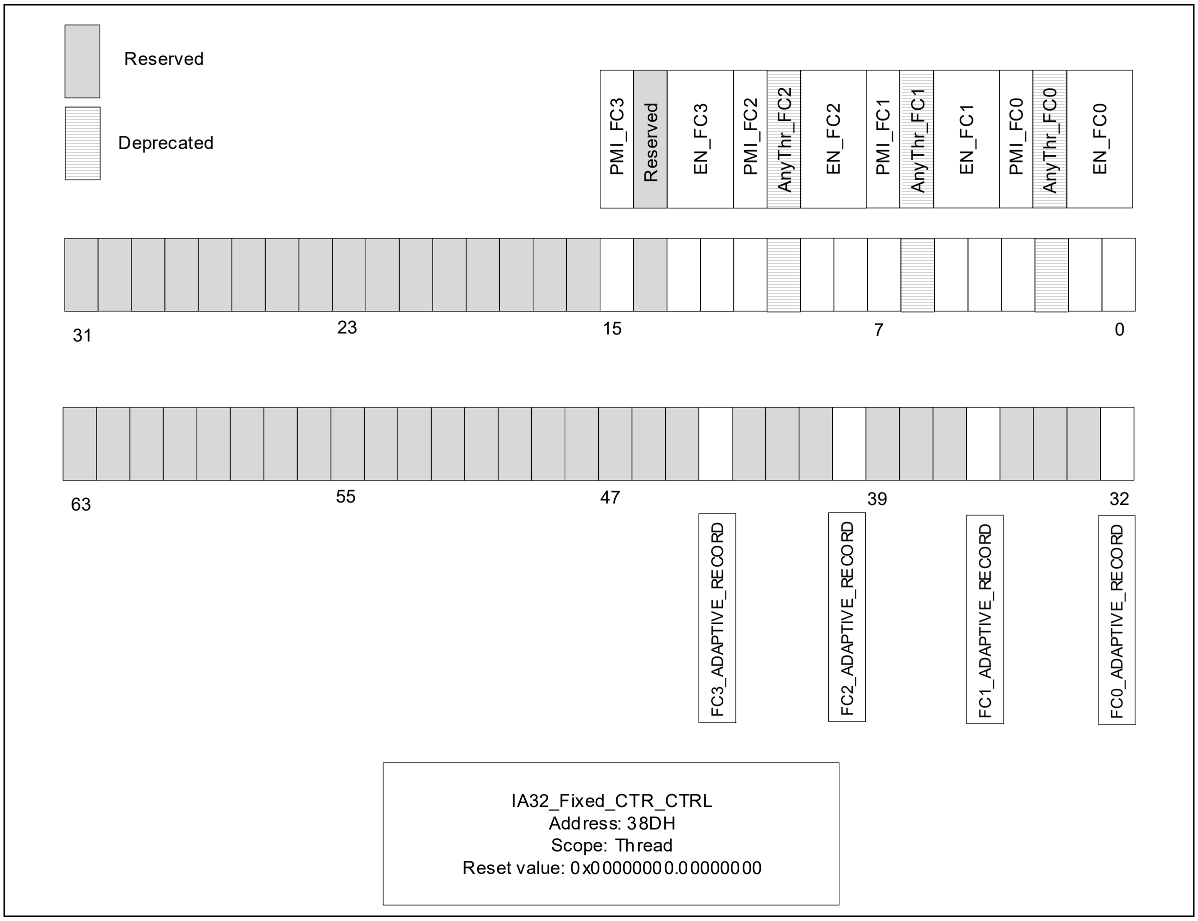


Figure 20-69. Layout of IA32_FIXED_CTR_CTRL MSR Supporting Adaptive PEBS

20.9.2.2 PEBS Record Format

The data fields in the PEBS record are aggregated into five groups which are described in the sub-sections below. Processors that support Adaptive PEBS implement a new MSR called MSR_PEBS_DATA_CFG which allows software to select the data groups to be captured. The data groups are not placed at fixed locations in the PEBS record, but are positioned immediately after one another, thus making the record format/size variable based on the groups selected.

20.9.2.2.1 Basic Info

The Basic group contains essential information for software to parse a record along with several critical fields. It is always collected.

Table 20-92. Basic Info Group

Field Name	Bit Width	Description
Record Format	[47:0]	This field indicates which data groups are included in the record. The field is zero if none of the counters that triggered the current PEBS record have their Adaptive_Record bit set. Otherwise it contains the value of MSR_PEBS_DATA_CFG.
	[63:48]	This field provides the size of the current record in bytes. Selected groups are packed back-to-back in the record without gaps or padding for unselected groups.

Table 20-92. Basic Info Group (Contd.)

Instruction Pointer	[63:0]	This field reports the Eventing Instruction Pointer (EventingIP) of the retired instruction that triggered the PEBS record generation. Note that this field is different than R/EIP which records the instruction pointer of the next instruction to be executed after record generation. The legacy R/EIP field has been removed.
Applicable Counters	[63:0]	The Applicable Counters field indicates which counters triggered the generation of the PEBS record, linking the record to specific events. This allows software to correlate the PEBS record entry properly with the instruction that caused the event, even when multiple counters are configured to generate PEBS records and multiple bits are set in the field.
TSC	[63:0]	This field provides the time stamp counter value when the PEBS record was generated.

20.9.2.2.2 Memory Access Info

This group contains the legacy PEBS memory-related fields; see Section 20.3.1.1.2.

Table 20-93. Memory Access Info Group

Field Name	Bit Width	Description
Memory Access Address	[63:0]	This field contains the linear address of the source of the load, or linear address of the destination (target) of the store. This value is written as a 64-bit address in canonical form.
Memory Auxiliary Info	[63:0]	When a MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains an encoded value indicating the memory hierarchy source which satisfied the load. These encodings are detailed in Table 20-4 and Table 20-13. If the PEBS assist was triggered for a store uop, this field will contain information indicating the status of the store, as detailed in Table 20-14.
Memory Access Latency[1]	[63:0]	When a MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains the latency to service the load in core clock cycles.
TSX Auxiliary Info	[31:0]	This field contains the number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
	[63:32]	This field contains the abort details. Refer to Section 20.3.6.5.1.

NOTES:

1. In certain conditions, high latencies in fields under “Memory Access Latency” may be observed even when the Data Src of the “Memory Auxiliary Info” field indicates a close source.

Beginning with 12th generation Intel Core processors, the memory access information group has been updated. New fields added are shaded gray in Table 20-94.

Table 20-94. Updated Memory Access Info Group

Field Name	Sub-field Name	Bits	Description
Access Address (offset 0H)	DLA	[63:0]	This field reports the data linear address (DLA) of the memory access in canonical form. A zero value indicates the processor could not retrieve the address of the particular access.
Access Info (offset 8H)	Data Src	[3:0]	An encoded value indicating the memory hierarchy source which satisfied the access. These encodings are detailed in Table 20-4. A zero value indicates the processor could not retrieve the data source of the particular access.
	STLB-miss	[4]	A value of 1 indicates the access has missed the Second-level TLB (STLB).
	Is-Lock	[5]	A value of 1 indicates the access was part of a locked (atomic) memory transaction.
	Data-Blk	[6]	A value of 1 indicates the load was blocked since its data could not be forwarded from a preceding store.
	Address-Blk	[7]	A value of 1 indicates the load was blocked due to potential address conflict with a preceding store.
Access Latency (offset 10H)	Instruction Latency	[15:0]	Measured instruction latency in core cycles. For loads, the latency starts by the dispatch of the load operation for execution and lasts until completion of the instruction it belongs to. This field includes the entire latency including time for data-dependency resolution or TLB lookups.
	Cache Latency	[47:32]	Measured cache access latency in core cycles. For loads, the latency starts by the actual cache access until the data is returned by the memory subsystem. For stores, the latency starts when the demand write accesses the L1 data-cache and lasts until the cacheline write is completed in the memory subsystem. This field does not include non-data-cache latency such as memory ordering checks or TLB lookups.
TSX (offset 18H)	Transaction Latency	[31:0]	This field contains the number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
	Abort Info	[63:32]	This field contains the abort details. Refer to Section 20.3.6.5.1.

To determine which fields are supported for certain performance monitoring events, consult the Memory Info attribute in the event lists at https://download.01.org/perfmon/.

NOTE

There may be additional block reasons, even if Data-Blk and Address-Blk are both clear, e.g., non-optimal instruction latency.

On P-core, the new Data-Blk and Address-Blk bits require the event LD_BLOCKS.STORE_FORWARD (r8203) to be configured in a programmable counter.

20.9.2.2.3 GPRs

This group is captured when the GPR bit is enabled in MSR_PEBS_DATA_CFG. GPRs are always 64 bits wide. If they are selected for non 64-bit mode, the upper 32-bit of the legacy RAX - RDI and all contents of R8-15 GPRs will be filled with 0s. In 64bit mode, the full 64 bit value of each register is written.

The order differs from legacy. The table below shows the order of the GPRs in Ice Lake microarchitecture.

Table 20-95. GPRs in Ice Lake Microarchitecture

Field Name	Bit Width
RFLAGS	[63:0]
RIP	[63:0]
RAX	[63:0]
RCX*	[63:0]
RDX*	[63:0]
RBX*	[63:0]
RSP*	[63:0]
RBP*	[63:0]
RSI*	[63:0]
RDI*	[63:0]
R8	[63:0]
...	...
R15	[63:0]

The machine state reported in the PEBS record is the committed machine state immediately after the instruction that triggers PEBS completes.

For instance, consider the following instruction sequence:

```
MOV eax, [eax]; triggers PEBS record generation
NOP
```

If the mov instruction triggers PEBS record generation, the EventingIP field in the PEBS record will report the address of the mov, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation. And the value of RIP will contain the linear address of the nop.

20.9.2.2.4 XMMs

This group is captured when the XMM bit is enabled in MSR_PEBS_DATA_CFG and SSE is enabled. If SSE is not enabled, the fields will contain zeroes. XMM8-XMM15 will also contain zeroes if not in 64-bit mode.

Table 20-96. XMMs

Field Name	Bit Width
XMM0	[127:0]
...	...
XMM15	[127:0]

20.9.2.2.5 LBRs

To capture LBR data in the PEBS record, the LBR bit in MSR_PEBS_DATA_CFG must be enabled. The number of LBR entries included in the record can be configured in the LBR_entries field of MSR_PEBS_DATA_CFG.

Table 20-97. LBRs

Field Name	Bit Width	Description
LBR[].FROM	[63:0]	Branch from address.
LBR[].TO	[63:0]	Branch to address.
LBR[].INFO	[63:0]	Other LBR information, like timing. This field is described in more detail in Section 18.12.1, “MSR_LBR_INFO_x MSR.”

LBR entries are recorded into the record starting at LBR[TOS] and proceeding to LBR[TOS-1] and following. Note that LBR index is modulo the number of LBRs supporting on the processor.

20.9.2.3 MSR_PEBS_DATA_CFG

Bits in MSR_PEBS_DATA_CFG can be set to include data field blocks/groups into adaptive records. The Basic Info group is always included in the record. Additionally, the number of LBR entries included in the record is configurable.

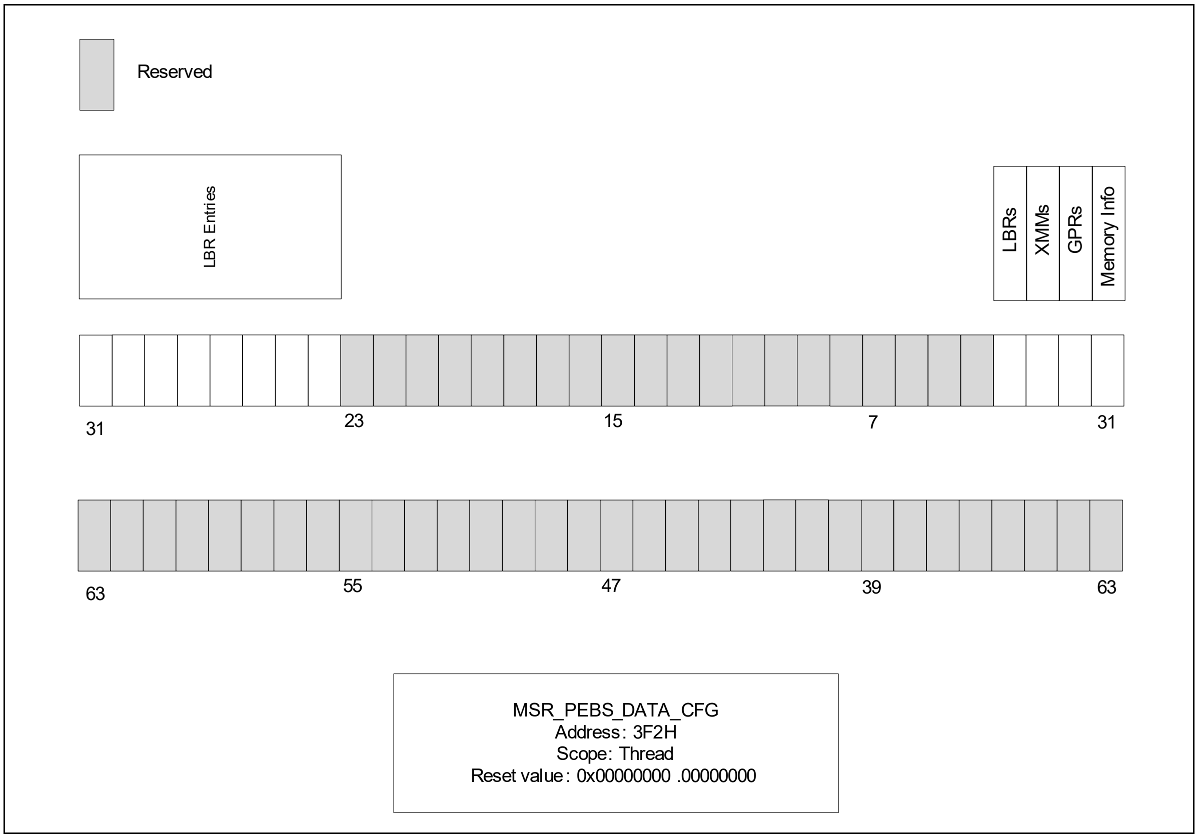


Figure 20-70. MSR_PEBS_DATA_CFG

Table 20-98. MSR_PEBS_CFG Programming[1]

Bit	Bit Index	Access	Description
Memory Info	0	R/W	Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.
GPRs	1	R/W	Setting this bit will capture the contents of the General Purpose registers in the PEBS record.
XMMs	2	R/W	Setting this bit will capture the contents of the XMM registers in the PEBS record.
LBRs	3	R/W	Setting this bit will capture LBR TO, FROM, and INFO in the PEBS record.
Reserved[2]	23:4	NA	Reserved
LBR Entries	31:24	R/W	Set the field to the desired number of entries minus 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, it is recommended to select an even number of LBR entries (programmed into LBR_entries as an odd number).

NOTES:

1. A write to the MSR will be ignored when IA32_MISC_ENABLE.PERFMON_AVAILABLE is zero (default).
2. Writing to the reserved bits will cause a GP fault.

20.9.2.4 PEBS Record Examples

The following example shows the layout of the PEBS record when all data groups are selected (all valid bits in MSR_PEBS_DATA_CFG are set) and maximum number of LBRs are selected. There are no gaps in the PEBS record when a subset of the groups are selected, thus keeping the layout compact. Implementations that do not support some features will have to pad zeroes in the corresponding fields.

Table 20-99. PEBS Record Example 1

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	
0x20	Memory Info	Memory Access Address	DLA
0x28		Memory Auxiliary Info	DATA_SRC
0x30		Memory Access Latency	Load Latency
0x38		TSX Auxiliary Info	HLE Information

Table 20-99. PEBS Record Example 1

0x40	GPRs	RFLAGS	
0x48		RIP	
0x50		RAX	
...		...	
0x88		RDI	
0x90		R8	
...		...	
0xC8		R15	
0xD0	XMMs	XMM0	New
...		...	
0x1C0		XMM15	
0x1D0	LBRs	LBR[TOS].FROM	New
0x1D8		LBR[TOS].TO	
0x1E0		LBR[TOS].INFO	
...		...	
0x4B8		LBR[TOS +1].FROM	
0x4C0		LBR[TOS +1].TO	
0x4C8		LBR[TOS +1].INFO	

The following example shows the layout of the PEBS record when Basic, GPR, and LBR group with 3 LBR entries are selected.

Table 20-100. PEBS Record Example 2

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	

Table 20-100. PEBS Record Example 2

0x20	GPRs	RFLAGS	
0x28		RIP	
0x30		RAX	
...		...	
0x68		RDI	
0x70		R8	
...		...	
0xA8		R15	
0xB0	LBRs	LBR[TOS].FROM	New
0xB8		LBR[TOS].TO	
0xC0		LBR[TOS].INFO	
...		...	
0xE0		LBR[TOS +1].FROM	
0xE8		LBR[TOS +1].TO	
0xF0		LBR[TOS +1].INFO	

20.9.3 Precise Distribution of Instructions Retired (PDIR) Facility

Precise Distribution of Instructions Retired Facility is available via PEBS on some microarchitectures. Refer to Section 20.3.4.4.4. Counters that support PDIR also vary. See the processor specific sections for availability.

20.9.4 Reduced Skid PEBS

For precise events, upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. The Reduced Skid mechanism mitigates the “skid” problem by providing an early indication of when the counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus greatly reducing skid.

This mechanism is a superset of the PDIR mechanism available in the Sandy Bridge microarchitecture. See Section 20.3.4.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including, INST_RETIRED, except for UOPS_RETIRED. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

With Reduced Skid PEBS, the skid is precisely one event occurrence. Hence if counting INST_RETIRED, PEBS will indicate the instruction that follows that which caused the counter to overflow.

For the Reduced Skid mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g., via IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g., what event is specified in IA32_PERFEVTSELx or whether PEBS is enabled for that counter via IA32_PEBS_ENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

20.9.5 EPT-Friendly PEBS

The 3rd generation Intel Xeon Scalable Family of processors based on Ice Lake microarchitecture (and later processors) and the 12th generation Intel Core processor (and later processors) support VMX guest use of PEBS when the DS Area (including the PEBS Buffer and DS Management Area) is allocated from a paged pool of EPT pages. In such a configuration PEBS DS Area accesses may result in VM exits (e.g., EPT violations due to "lazy" EPT page-table entry propagation), and in such cases the PEBS record will not be lost but instead will "skid" to after the subsequent VM Entry back to the guest. For precise events the guest will observe that the record skid by one event occurrence, while for non-precise events the record will skid by one instruction.

20.9.6 PDist: Precise Distribution

PDist eliminates any skid or shadowing effects from PEBS. With PDist, the PEBS record will be generated precisely upon completion of the instruction or operation that causes the counter to overflow (there is no "wait for next occurrence" by default).

PDist is supported by selected counters, and is only supported when those counters are programmed to count select precise events[1]. The legacy PEBS behavior applies to counters that do not support PDist, unless specified otherwise. PDist requires that the INV, ANY, E, and CMASK fields are cleared. Which counters support PDist, and which events are supported for PDist, is model-specific. Further, the counter reload value must not be lesser than 127 for PDist to operate.

For the PDist mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g., via IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g., what event is specified in IA32_PERFEVTSELx or whether PEBS is enabled for that counter via IA32_PEBS_ENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

20.9.7 Load Latency Facility

The load latency facility provides software a means to characterize the latencies of memory load operations to different levels of cache/memory hierarchy. This facility requires a processor supporting the enhanced PEBS record format in the PEBS buffer.

Beginning with 12th generation Intel Core processors, the load latency facility supports all fields in Table 20-94, "Updated Memory Access Info Group," in addition to the Memory Access Address field:

- The **Instruction Latency** field measures the load latency from the load's first dispatch until final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches and data dependencies).
- The **Cache Latency** field measures the subset of cache access latency in core cycles. It starts from the actual cache access until the data is returned by the memory subsystem The latency is reported for retired demand load operations in core cycles (it does not account for memory ordering blocks).
- The **Data Source** field is an encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 20-101. In the descriptions, local memory refers to system memory physically attached to a processor package, and remote memory refers to system memory or cache physically attached to another processor package (in a server product).
- Through the **Access Info** field, load latency features binary indications on certain blocks that the load operation may have encountered. Refer to STLB-miss, Is-Lock, Data-Blk and Address-Blk fields in Table 20-94.

NOTE

> For loads triggered by software prefetch instructions, the cache related fields including Data Source and Cache Latency, report values as if the load was an L1 cache hit (the prefetch completes without waiting for data return, for performance reasons).

[1] To determine whether an event is precise or supports PDist, consult the relevant attribute in the event lists at https://download.01.org/perfmon/.

Table 20-101. Data Source Encoding for Memory Accesses (Ice Lake and Later Microarchitectures)

Encoding	Description
00H	Unknown Data Source (the processor could not retrieve the origin of this request).
01H	L1 HIT. This request was satisfied by the L1 data cache. (Minimal latency core cache hit.)
02H	FB HIT. This request was merged into an outstanding cache miss to same cache-line address.
03H	L2 HIT. This request was satisfied by the L2 cache.
04H	L3 HIT. This request was satisfied by the L3 cache with no coherency actions performed (snooping).
05H	XCORE MISS. This request was satisfied by the L3 cache but involved a coherency check in some sibling core(s).
06H	XCORE HIT. This request was satisfied by the L3 cache but involved a coherency check that hit a non-modified copy in a sibling core.
07H	XCORE FWD. This request was satisfied by a sibling core where either a modified (cross-core HITM) or a non-modified (cross-core FWD) cache-line copy was found.
08H	Local Far Memory. This request has missed the L3 cache and was serviced by local far memory.
09H	Remote Far Memory. This request has missed the L3 cache and was serviced by remote far memory.
0AH	Local Near Memory. This request has missed the L3 cache and was serviced by local near memory.
0BH	Remote Near Memory. This request has missed the L3 cache and was serviced by remote near memory.
0CH	Remote FWD. This request has missed the L3 cache and a non-modified cache-line copy was forwarded from a remote cache.
0DH	Remote HITM. This request has missed the L3 cache and a modified cache-line was forwarded from a remote cache.
0EH	I/O. Request of input/output operation.
0FH	UC. The request was to uncacheable memory.

To use this feature, software must complete the following steps:

- Complete the PEBS configuration steps.
- Set the Memory Info bit in the PEBS_DATA_CFG MSR.
- One of the relevant IA32_PERFEVTSELx MSRs is programmed to specify the event unit MEM_TRANS_RETIRED.LOAD_LATENCY (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter, IA32_PMCx, will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in an MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with instruction latency greater than this value are eligible for counting and PEBS data reporting. The minimum value that may be programmed in this register is 1.
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register.

Refer to Section 20.3.4.4.2 for further implementation details of Load Latency.

20.9.8 Store Latency Facility

Store latency support is available on the 12th generation Intel Core processor. Store latency is a PEBS extension that provides a means to profile store memory accesses in the system. It complements the load latency facility.

Store latency leverages the PEBS facility where it can provide additional information about sampled stores. The additional information includes the data address, memory auxiliary information, and the cache latency of the store access. Normal stores (those preceded with a read-for-ownership) as well as streaming stores are supported by the store latency facility.

Memory store operations typically do not limit performance since they update the memory with no operation that directly depends on them. Thus, data out of this facility should be carefully used once stores are suspected as a performance limiter; for example, once the TMA node of Backend_Bound.Memory_Bound.Store_Bound is flagged[1].

To enable the store latency facility, software must complete the following steps:

- Complete the PEBS configuration steps.
- Set the Memory Info bit in the PEBS_DATA_CFG MSR.
- Program the MEM_TRANS_RETIRED.STORE_SAMPLE event on general-purpose performance-monitoring counter 0 (IA32_PERFEVTSEL0[15:0] = 2CDH).
- Setup the PEBS buffer to hold at least two records, setting both 'PEBS Absolute Maximum' and 'PEBS Interrupt Threshold', should any other counter be used by PEBS (that is whenever IA32_PEBS_ENABLE[x] ≠ 0 for x ≠ 0).
- Set IA32_PEBS_ENABLE[0].

The store latency information is written into a PEBS record as shown in Table 20-48.

The store latency relies on the PEBS facility, so the PEBS configuration must be completed first. Unlike load latency, there is no option to filter on a subset of stores that exceed a certain threshold.

[1] For more details about the method, refer to Section B.1, "Top-Down Analysis Method" of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

CHAPTER 21
8086 EMULATION

IA-32 processors (beginning with the Intel386 processor) provide two ways to execute new or legacy programs that are assembled and/or compiled to run on an Intel 8086 processor:

- Real-address mode.
- Virtual-8086 mode.

Figure 2-3 shows the relationship of these operating modes to protected mode and system management mode (SMM).

When the processor is powered up or reset, it is placed in the real-address mode. This operating mode almost exactly duplicates the execution environment of the Intel 8086 processor, with some extensions. Virtually any program assembled and/or compiled to run on an Intel 8086 processor will run on an IA-32 processor in this mode.

When running in protected mode, the processor can be switched to virtual-8086 mode to run 8086 programs. This mode also duplicates the execution environment of the Intel 8086 processor, with extensions. In virtual-8086 mode, an 8086 program runs as a separate protected-mode task. Legacy 8086 programs are thus able to run under an operating system (such as Microsoft Windows*) that takes advantage of protected mode and to use protected-mode facilities, such as the protected-mode interrupt- and exception-handling facilities. Protected-mode multitasking permits multiple virtual-8086 mode tasks (with each task running a separate 8086 program) to be run on the processor along with other non-virtual-8086 mode tasks.

This section describes both the basic real-address mode execution environment and the virtual-8086-mode execution environment, available on the IA-32 processors beginning with the Intel386 processor.

21.1 REAL-ADDRESS MODE

The IA-32 architecture’s real-address mode runs programs written for the Intel 8086, Intel 8088, Intel 80186, and Intel 80188 processors, or for the real-address mode of the Intel 286, Intel386, Intel486, Pentium, P6 family, Pentium 4, and Intel Xeon processors.

The execution environment of the processor in real-address mode is designed to duplicate the execution environment of the Intel 8086 processor. To an 8086 program, a processor operating in real-address mode behaves like a high-speed 8086 processor. The principal features of this architecture are defined in Chapter 3, “Basic Execution Environment,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

The following is a summary of the core features of the real-address mode execution environment as would be seen by a program written for the 8086:

- The processor supports a nominal 1-MByte physical address space (see Section 21.1.1, “Address Translation in Real-Address Mode,” for specific details). This address space is divided into segments, each of which can be up to 64 KBytes in length. The base of a segment is specified with a 16-bit segment selector, which is shifted left by 4 bits to form a 20-bit offset from address 0 in the address space. An operand within a segment is addressed with a 16-bit offset from the base of the segment. A physical address is thus formed by adding the offset to the 20-bit segment base (see Section 21.1.1, “Address Translation in Real-Address Mode”).
- All operands in “native 8086 code” are 8-bit or 16-bit values. (Operand size override prefixes can be used to access 32-bit operands.)
- Eight 16-bit general-purpose registers are provided: AX, BX, CX, DX, SP, BP, SI, and DI. The extended 32 bit registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI) are accessible to programs that explicitly perform a size override operation.
- Four segment registers are provided: CS, DS, SS, and ES. (The FS and GS registers are accessible to programs that explicitly access them.) The CS register contains the segment selector for the code segment; the DS and ES registers contain segment selectors for data segments; and the SS register contains the segment selector for the stack segment.
- The 8086 16-bit instruction pointer (IP) is mapped to the lower 16-bits of the EIP register. Note this register is a 32-bit register and unintentional address wrapping may occur.

- The 16-bit FLAGS register contains status and control flags. (This register is mapped to the 16 least significant bits of the 32-bit EFLAGS register.)
- All of the Intel 8086 instructions are supported (see Section 21.1.3, "Instructions Supported in Real-Address Mode").
- A single, 16-bit-wide stack is provided for handling procedure calls and invocations of interrupt and exception handlers. This stack is contained in the stack segment identified with the SS register. The SP (stack pointer) register contains an offset into the stack segment. The stack grows down (toward lower segment offsets) from the stack pointer. The BP (base pointer) register also contains an offset into the stack segment that can be used as a pointer to a parameter list. When a CALL instruction is executed, the processor pushes the current instruction pointer (the 16 least-significant bits of the EIP register and, on far calls, the current value of the CS register) onto the stack. On a return, initiated with a RET instruction, the processor pops the saved instruction pointer from the stack into the EIP register (and CS register on far returns). When an implicit call to an interrupt or exception handler is executed, the processor pushes the EIP, CS, and EFLAGS (low-order 16-bits only) registers onto the stack. On a return from an interrupt or exception handler, initiated with an IRET instruction, the processor pops the saved instruction pointer and EFLAGS image from the stack into the EIP, CS, and EFLAGS registers.
- A single interrupt table, called the "interrupt vector table" or "interrupt table," is provided for handling interrupts and exceptions (see Figure 21-2). The interrupt table (which has 4-byte entries) takes the place of the interrupt descriptor table (IDT, with 8-byte entries) used when handling protected-mode interrupts and exceptions. Interrupt and exception vector numbers provide an index to entries in the interrupt table. Each entry provides a pointer (called a "vector") to an interrupt- or exception-handling procedure. See Section 21.1.4, "Interrupt and Exception Handling," for more details. It is possible for software to relocate the IDT by means of the LIDT instruction on IA-32 processors beginning with the Intel386 processor.
- The x87 FPU is active and available to execute x87 FPU instructions in real-address mode. Programs written to run on the Intel 8087 and Intel 287 math coprocessors can be run in real-address mode without modification.

The following extensions to the Intel 8086 execution environment are available in the IA-32 architecture's real-address mode. If backwards compatibility to Intel 286 and Intel 8086 processors is required, these features should not be used in new programs written to run in real-address mode.

- Two additional segment registers (FS and GS) are available.
- Many of the integer and system instructions that have been added to later IA-32 processors can be executed in real-address mode (see Section 21.1.3, "Instructions Supported in Real-Address Mode").
- The 32-bit operand prefix can be used in real-address mode programs to execute the 32-bit forms of instructions. This prefix also allows real-address mode programs to use the processor's 32-bit general-purpose registers.
- The 32-bit address prefix can be used in real-address mode programs, allowing 32-bit offsets.

The following sections describe address formation, registers, available instructions, and interrupt and exception handling in real-address mode. For information on I/O in real-address mode, see Chapter 19, "Input/Output," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

21.1.1 Address Translation in Real-Address Mode

In real-address mode, the processor does not interpret segment selectors as indexes into a descriptor table; instead, it uses them directly to form linear addresses as the 8086 processor does. It shifts the segment selector left by 4 bits to form a 20-bit base address (see Figure 21-1). The offset into a segment is added to the base address to create a linear address that maps directly to the physical address space.

When using 8086-style address translation, it is possible to specify addresses larger than 1 MByte. For example, with a segment selector value of FFFFH and an offset of FFFFH, the linear (and physical) address would be 10FFEFH (1 megabyte plus 64 KBytes). The 8086 processor, which can form addresses only up to 20 bits long, truncates the high-order bit, thereby "wrapping" this address to FFEFH. When operating in real-address mode, however, the processor does not truncate such an address and uses it as a physical address. (Note, however, that for IA-32 processors beginning with the Intel486 processor, the A20M# signal can be used in real-address mode to mask address line A20, thereby mimicking the 20-bit wrap-around behavior of the 8086 processor.) Care should be take to ensure that A20M# based address wrapping is handled correctly in multiprocessor based system.

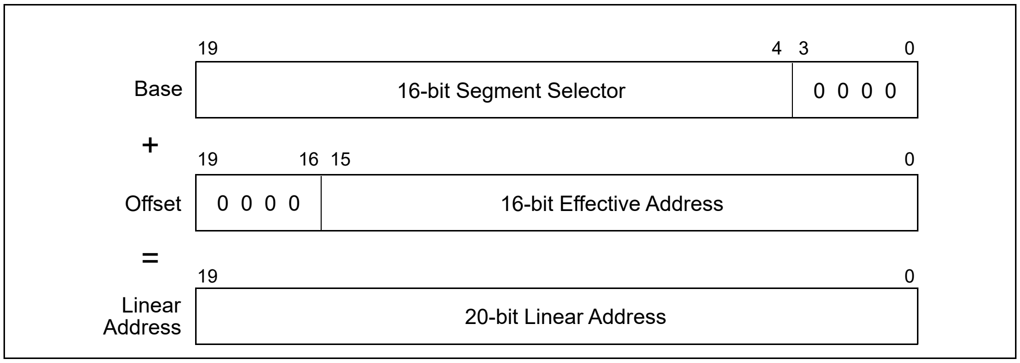


Figure 21-1. Real-Address Mode Address Translation

The IA-32 processors beginning with the Intel386 processor can generate 32-bit offsets using an address override prefix; however, in real-address mode, the value of a 32-bit offset may not exceed FFFFH without causing an exception.

For full compatibility with Intel 286 real-address mode, pseudo-protection faults (interrupt 12 or 13) occur if a 32-bit offset is generated outside the range 0 through FFFFH.

21.1.2 Registers Supported in Real-Address Mode

The register set available in real-address mode includes all the registers defined for the 8086 processor plus the new registers introduced in later IA-32 processors, such as the FS and GS segment registers, the debug registers, the control registers, and the floating-point unit registers. The 32-bit operand prefix allows a real-address mode program to use the 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI).

21.1.3 Instructions Supported in Real-Address Mode

The following instructions make up the core instruction set for the 8086 processor. If backwards compatibility to the Intel 286 and Intel 8086 processors is required, only these instructions should be used in a new program written to run in real-address mode.

- Move (MOV) instructions that move operands between general-purpose registers, segment registers, and between memory and general-purpose registers.
- The exchange (XCHG) instruction.
- Load segment register instructions LDS and LES.
- Arithmetic instructions ADD, ADC, SUB, SBB, MUL, IMUL, DIV, IDIV, INC, DEC, CMP, and NEG.
- Logical instructions AND, OR, XOR, and NOT.
- Decimal instructions DAA, DAS, AAA, AAS, AAM, and AAD.
- Stack instructions PUSH and POP (to general-purpose registers and segment registers).
- Type conversion instructions CWD, CDQ, CBW, and CWDE.
- Shift and rotate instructions SAL, SHL, SHR, SAR, ROL, ROR, RCL, and RCR.
- TEST instruction.
- Control instructions JMP, J*cc*, CALL, RET, LOOP, LOOPE, and LOOPNE.
- Interrupt instructions INT *n*, INTO, and IRET.
- EFLAGS control instructions STC, CLC, CMC, CLD, STD, LAHF, SAHF, PUSHF, and POPF.
- I/O instructions IN, INS, OUT, and OUTS.
- Load effective address (LEA) instruction, and translate (XLATB) instruction.

- LOCK prefix.
- Repeat prefixes REP, REPE, REPZ, REPNE, and REPNZ.
- Processor halt (HLT) instruction.
- No operation (NOP) instruction.

The following instructions, added to later IA-32 processors (some in the Intel 286 processor and the remainder in the Intel386 processor), can be executed in real-address mode, if backwards compatibility to the Intel 8086 processor is not required.

- Move (MOV) instructions that operate on the control and debug registers.
- Load segment register instructions LSS, LFS, and LGS.
- Generalized multiply instructions and multiply immediate data.
- Shift and rotate by immediate counts.
- Stack instructions PUSHA, PUSHAD, POPA, POPAD, and PUSH immediate data.
- Move with sign extension instructions MOVSX and MOVZX.
- Long-displacement J*cc* instructions.
- Exchange instructions CMPXCHG, CMPXCHG8B, and XADD.
- String instructions MOVS, CMPS, SCAS, LODS, and STOS.
- Bit test and bit scan instructions BT, BTS, BTR, BTC, BSF, and BSR; the byte-set-on condition instruction SET*cc*; and the byte swap (BSWAP) instruction.
- Double shift instructions SHLD and SHRD.
- EFLAGS control instructions PUSHF and POPF.
- ENTER and LEAVE control instructions.
- BOUND instruction.
- CPU identification (CPUID) instruction.
- System instructions CLTS, INVD, WINVD, INVLPG, LGDT, SGDT, LIDT, SIDT, LMSW, SMSW, RDMSR, WRMSR, RDTSC, and RDPMC.

Execution of any of the other IA-32 architecture instructions (not given in the previous two lists) in real-address mode result in an invalid-opcode exception (#UD) being generated.

21.1.4 Interrupt and Exception Handling

When operating in real-address mode, software must provide interrupt and exception-handling facilities that are separate from those provided in protected mode. Even during the early stages of processor initialization when the processor is still in real-address mode, elementary real-address mode interrupt and exception-handling facilities must be provided to ensure reliable operation of the processor, or the initialization code must ensure that no interrupts or exceptions will occur.

The IA-32 processors handle interrupts and exceptions in real-address mode similar to the way they handle them in protected mode. When a processor receives an interrupt or generates an exception, it uses the vector number of the interrupt or exception as an index into the interrupt table. (In protected mode, the interrupt table is called the **interrupt descriptor table (IDT)**, but in real-address mode, the table is usually called the **interrupt vector table**, or simply the **interrupt table**.) The entry in the interrupt vector table provides a pointer to an interrupt- or exception-handler procedure. (The pointer consists of a segment selector for a code segment and a 16-bit offset into the segment.) The processor performs the following actions to make an implicit call to the selected handler:

1. Pushes the current values of the CS and EIP registers onto the stack. (Only the 16 least-significant bits of the EIP register are pushed.)
2. Pushes the low-order 16 bits of the EFLAGS register onto the stack.
3. Clears the IF flag in the EFLAGS register to disable interrupts.
4. Clears the TF, RF, and AC flags, in the EFLAGS register.

5. Transfers program control to the location specified in the interrupt vector table.

An IRET instruction at the end of the handler procedure reverses these steps to return program control to the interrupted program. Exceptions do not return error codes in real-address mode.

The interrupt vector table is an array of 4-byte entries (see Figure 21-2). Each entry consists of a far pointer to a handler procedure, made up of a segment selector and an offset. The processor scales the interrupt or exception vector by 4 to obtain an offset into the interrupt table. Following reset, the base of the interrupt vector table is located at physical address 0 and its limit is set to 3FFH. In the Intel 8086 processor, the base address and limit of the interrupt vector table cannot be changed. In the later IA-32 processors, the base address and limit of the interrupt vector table are contained in the IDTR register and can be changed using the LIDT instruction.

(For backward compatibility to Intel 8086 processors, the default base address and limit of the interrupt vector table should not be changed.)

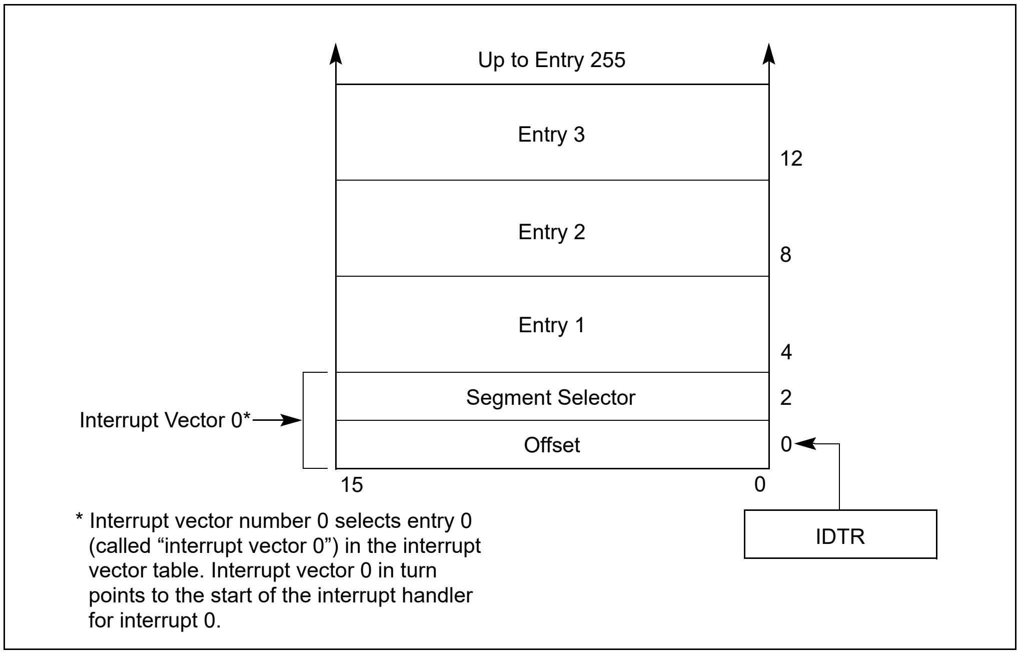

Up to Entry 255
Entry 3
12
Entry 2
8
Entry 1
4
Segment Selector
2
Offset
0
Interrupt Vector 0*
15
0
IDTR
* Interrupt vector number 0 selects entry 0 (called "interrupt vector 0") in the interrupt vector table. Interrupt vector 0 in turn points to the start of the interrupt handler for interrupt 0.

Figure 21-2. Interrupt Vector Table in Real-Address Mode

Table 21-1 shows the interrupt and exception vectors that can be generated in real-address mode and virtual-8086 mode, and in the Intel 8086 processor. See Chapter 6, "Interrupt and Exception Handling," for a description of the exception conditions.

21.2 VIRTUAL-8086 MODE

Virtual-8086 mode is actually a special type of a task that runs in protected mode. When the operating-system or executive switches to a virtual-8086-mode task, the processor emulates an Intel 8086 processor. The execution environment of the processor while in the 8086-emulation state is the same as is described in Section 21.1, "Real-Address Mode," for real-address mode, including the extensions. The major difference between the two modes is that in virtual-8086 mode the 8086 emulator uses some protected-mode services (such as the protected-mode interrupt and exception-handling and paging facilities).

As in real-address mode, any new or legacy program that has been assembled and/or compiled to run on an Intel 8086 processor will run in a virtual-8086-mode task. And several 8086 programs can be run as virtual-8086-mode tasks concurrently with normal protected-mode tasks, using the processor's multitasking facilities.

Table 21-1. Real-Address Mode Exceptions and Interrupts

Vector No.	Description	Real-Address Mode	Virtual-8086 Mode	Intel 8086 Processor
0	Divide Error (#DE)	Yes	Yes	Yes
1	Debug Exception (#DB)	Yes	Yes	No
2	NMI Interrupt	Yes	Yes	Yes
3	Breakpoint (#BP)	Yes	Yes	Yes
4	Overflow (#OF)	Yes	Yes	Yes
5	BOUND Range Exceeded (#BR)	Yes	Yes	Reserved
6	Invalid Opcode (#UD)	Yes	Yes	Reserved
7	Device Not Available (#NM)	Yes	Yes	Reserved
8	Double Fault (#DF)	Yes	Yes	Reserved
9	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
10	Invalid TSS (#TS)	Reserved	Yes	Reserved
11	Segment Not Present (#NP)	Reserved	Yes	Reserved
12	Stack Fault (#SS)	Yes	Yes	Reserved
13	General Protection (#GP)*	Yes	Yes	Reserved
14	Page Fault (#PF)	Reserved	Yes	Reserved
15	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
16	Floating-Point Error (#MF)	Yes	Yes	Reserved
17	Alignment Check (#AC)	Reserved	Yes	Reserved
18	Machine Check (#MC)	Yes	Yes	Reserved
19-31	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
32-255	User Defined Interrupts	Yes	Yes	Yes

NOTE:

* In the real-address mode, vector 13 is the segment overrun exception. In protected and virtual-8086 modes, this exception covers all general-protection error conditions, including traps to the virtual-8086 monitor from virtual-8086 mode.

21.2.1 Enabling Virtual-8086 Mode

The processor runs in virtual-8086 mode when the VM (virtual machine) flag in the EFLAGS register is set. This flag can only be set when the processor switches to a new protected-mode task or resumes virtual-8086 mode via an IRET instruction.

System software cannot change the state of the VM flag directly in the EFLAGS register (for example, by using the POPFD instruction). Instead it changes the flag in the image of the EFLAGS register stored in the TSS or on the stack following a call to an interrupt- or exception-handler procedure. For example, software sets the VM flag in the EFLAGS image in the TSS when first creating a virtual-8086 task.

The processor tests the VM flag under three general conditions:

- When loading segment registers, to determine whether to use 8086-style address translation.
- When decoding instructions, to determine which instructions are not supported in virtual-8086 mode and which instructions are sensitive to IOPL.

- When checking privileged instructions, on page accesses, or when performing other permission checks. (Virtual-8086 mode always executes at CPL 3.)

21.2.2 Structure of a Virtual-8086 Task

A virtual-8086-mode task consists of the following items:

- A 32-bit TSS for the task.
- The 8086 program.
- A virtual-8086 monitor.
- 8086 operating-system services.

The TSS of the new task must be a 32-bit TSS, not a 16-bit TSS, because the 16-bit TSS does not load the most-significant word of the EFLAGS register, which contains the VM flag. All TSS's, stacks, data, and code used to handle exceptions when in virtual-8086 mode must also be 32-bit segments.

The processor enters virtual-8086 mode to run the 8086 program and returns to protected mode to run the virtual-8086 monitor.

The virtual-8086 monitor is a 32-bit protected-mode code module that runs at a CPL of 0. The monitor consists of initialization, interrupt- and exception-handling, and I/O emulation procedures that emulate a personal computer or other 8086-based platform. Typically, the monitor is either part of or closely associated with the protected-mode general-protection (#GP) exception handler, which also runs at a CPL of 0. As with any protected-mode code module, code-segment descriptors for the virtual-8086 monitor must exist in the GDT or in the task's LDT. The virtual-8086 monitor also may need data-segment descriptors so it can examine the IDT or other parts of the 8086 program in the first 1 MByte of the address space. The linear addresses above 10FFEFH are available for the monitor, the operating system, and other system software.

The 8086 operating-system services consists of a kernel and/or operating-system procedures that the 8086 program makes calls to. These services can be implemented in either of the following two ways:

- They can be included in the 8086 program. This approach is desirable for either of the following reasons:
  - The 8086 program code modifies the 8086 operating-system services.
  - There is not sufficient development time to merge the 8086 operating-system services into main operating system or executive.
- They can be implemented or emulated in the virtual-8086 monitor. This approach is desirable for any of the following reasons:
  - The 8086 operating-system procedures can be more easily coordinated among several virtual-8086 tasks.
  - Memory can be saved by not duplicating 8086 operating-system procedure code for several virtual-8086 tasks.
  - The 8086 operating-system procedures can be easily emulated by calls to the main operating system or executive.

The approach chosen for implementing the 8086 operating-system services may result in different virtual-8086-mode tasks using different 8086 operating-system services.

21.2.3 Paging of Virtual-8086 Tasks

Even though a program running in virtual-8086 mode can use only 20-bit linear addresses, the processor converts these addresses into 32-bit linear addresses before mapping them to the physical address space. If paging is being used, the 8086 address space for a program running in virtual-8086 mode can be paged and located in a set of pages in physical address space. If paging is used, it is transparent to the program running in virtual-8086 mode just as it is for any task running on the processor.

Paging is not necessary for a single virtual-8086-mode task, but paging is useful or necessary in the following situations:

- When running multiple virtual-8086-mode tasks. Here, paging allows the lower 1 MByte of the linear address space for each virtual-8086-mode task to be mapped to a different physical address location.
- When emulating the 8086 address-wraparound that occurs at 1 MByte. When using 8086-style address translation, it is possible to specify addresses larger than 1 MByte. These addresses automatically wraparound in the Intel 8086 processor (see Section 21.1.1, “Address Translation in Real-Address Mode”). If any 8086 programs depend on address wraparound, the same effect can be achieved in a virtual-8086-mode task by mapping the linear addresses between 100000H and 110000H and linear addresses between 0 and 10000H to the same physical addresses.
- When sharing the 8086 operating-system services or ROM code that is common to several 8086 programs running as different 8086-mode tasks.
- When redirecting or trapping references to memory-mapped I/O devices.

21.2.4 Protection within a Virtual-8086 Task

Protection is not enforced between the segments of an 8086 program. Either of the following techniques can be used to protect the system software running in a virtual-8086-mode task from the 8086 program:

- Reserve the first 1 MByte plus 64 KBytes of each task’s linear address space for the 8086 program. An 8086 processor task cannot generate addresses outside this range.
- Use the U/S flag of page-table entries to protect the virtual-8086 monitor and other system software in the virtual-8086 mode task space. When the processor is in virtual-8086 mode, the CPL is 3. Therefore, an 8086 processor program has only user privileges. If the pages of the virtual-8086 monitor have supervisor privilege, they cannot be accessed by the 8086 program.

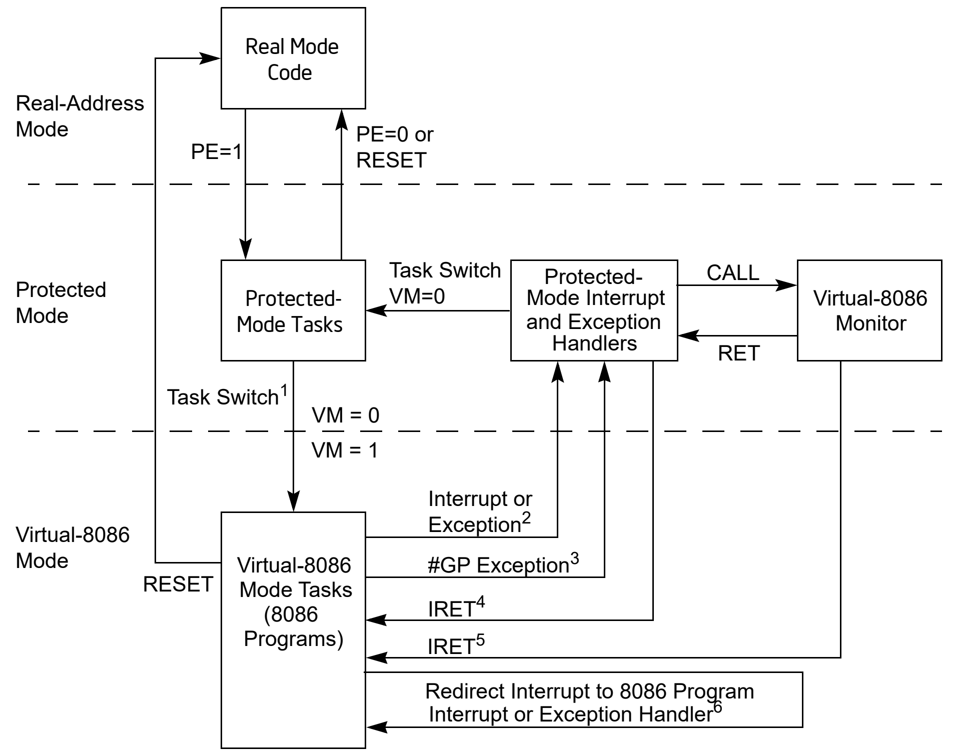
21.2.5 Entering Virtual-8086 Mode

Figure 21-3 summarizes the methods of entering and leaving virtual-8086 mode. The processor switches to virtual-8086 mode in either of the following situations:

- Task switch when the VM flag is set to 1 in the EFLAGS register image stored in the TSS for the task. Here the task switch can be initiated in either of two ways:
    - A CALL or JMP instruction.
    - An IRET instruction, where the NT flag in the EFLAGS image is set to 1.
- Return from a protected-mode interrupt or exception handler when the VM flag is set to 1 in the EFLAGS register image on the stack.

When a task switch is used to enter virtual-8086 mode, the TSS for the virtual-8086-mode task must be a 32-bit TSS. (If the new TSS is a 16-bit TSS, the upper word of the EFLAGS register is not in the TSS, causing the processor to clear the VM flag when it loads the EFLAGS register.) The processor updates the VM flag prior to loading the segment registers from their images in the new TSS. The new setting of the VM flag determines whether the processor interprets the contents of the segment registers as 8086-style segment selectors or protected-mode segment selectors. When the VM flag is set, the segment registers are loaded from the TSS, using 8086-style address translation to form base addresses.

See Section 21.3, “Interrupt and Exception Handling in Virtual-8086 Mode,” for information on entering virtual-8086 mode on a return from an interrupt or exception handler.



NOTES:

1. Task switch carried out in either of two ways:
   - CALL or JMP where the VM flag in the EFLAGS image is 1.
   - IRET where VM is 1 and NT is 1.
2. Hardware interrupt or exception; software interrupt (INT *n*) when IOPL is 3.
3. General-protection exception caused by software interrupt (INT *n*), IRET, POPF, PUSHF, IN, or OUT when IOPL is less than 3.
4. Normal return from protected-mode interrupt or exception handler.
5. A return from the 8086 monitor to redirect an interrupt or exception back to an interrupt or exception handler in the 8086 program running in virtual-8086 mode.
6. Internal redirection of a software interrupt (INT *n*) when VME is 1, IOPL is <3, and the redirection bit is 1.

Figure 21-3. Entering and Leaving Virtual-8086 Mode

21.2.6 Leaving Virtual-8086 Mode

The processor can leave the virtual-8086 mode only through an interrupt or exception. The following are situations where an interrupt or exception will lead to the processor leaving virtual-8086 mode (see Figure 21-3):

- The processor services a hardware interrupt generated to signal the suspension of execution of the virtual-8086 application. This hardware interrupt may be generated by a timer or other external mechanism. Upon receiving the hardware interrupt, the processor enters protected mode and switches to a protected-mode (or another virtual-8086 mode) task either through a task gate in the protected-mode IDT or through a trap or interrupt gate that points to a handler that initiates a task switch. A task switch from a virtual-8086 task to another task loads the EFLAGS register from the TSS of the new task. The value of the VM flag in the new EFLAGS determines if the new task executes in virtual-8086 mode or not.
- The processor services an exception caused by code executing the virtual-8086 task or services a hardware interrupt that "belongs to" the virtual-8086 task. Here, the processor enters protected mode and services the

exception or hardware interrupt through the protected-mode IDT (normally through an interrupt or trap gate) and the protected-mode exception- and interrupt-handlers. The processor may handle the exception or interrupt within the context of the virtual 8086 task and return to virtual-8086 mode on a return from the handler procedure. The processor may also execute a task switch and handle the exception or interrupt in the context of another task.

- The processor services a software interrupt generated by code executing in the virtual-8086 task (such as a software interrupt to call a MS-DOS* operating system routine). The processor provides several methods of handling these software interrupts, which are discussed in detail in Section 21.3.3, "Class 3—Software Interrupt Handling in Virtual-8086 Mode." Most of them involve the processor entering protected mode, often by means of a general-protection (#GP) exception. In protected mode, the processor can send the interrupt to the virtual-8086 monitor for handling and/or redirect the interrupt back to the application program running in virtual-8086 mode task for handling.

  IA-32 processors that incorporate the virtual mode extension (enabled with the VME flag in control register CR4) are capable of redirecting software-generated interrupts back to the program's interrupt handlers without leaving virtual-8086 mode. See Section 21.3.3.4, "Method 5: Software Interrupt Handling," for more information on this mechanism.
- A hardware reset initiated by asserting the RESET or INIT pin is a special kind of interrupt. When a RESET or INIT is signaled while the processor is in virtual-8086 mode, the processor leaves virtual-8086 mode and enters real-address mode.
- Execution of the HLT instruction in virtual-8086 mode will cause a general-protection (GP#) fault, which the protected-mode handler generally sends to the virtual-8086 monitor. The virtual-8086 monitor then determines the correct execution sequence after verifying that it was entered as a result of a HLT execution.

See Section 21.3, "Interrupt and Exception Handling in Virtual-8086 Mode," for information on leaving virtual-8086 mode to handle an interrupt or exception generated in virtual-8086 mode.

21.2.7 Sensitive Instructions

When an IA-32 processor is running in virtual-8086 mode, the CLI, STI, PUSHF, POPF, INT *n*, and IRET instructions are sensitive to IOPL. The IN, INS, OUT, and OUTS instructions, which are sensitive to IOPL in protected mode, are not sensitive in virtual-8086 mode.

The CPL is always 3 while running in virtual-8086 mode; if the IOPL is less than 3, an attempt to use the IOPL-sensitive instructions listed above triggers a general-protection exception (#GP). These instructions are sensitive to IOPL to give the virtual-8086 monitor a chance to emulate the facilities they affect.

21.2.8 Virtual-8086 Mode I/O

Many 8086 programs written for non-multitasking systems directly access I/O ports. This practice may cause problems in a multitasking environment. If more than one program accesses the same port, they may interfere with each other. Most multitasking systems require application programs to access I/O ports through the operating system. This results in simplified, centralized control.

The processor provides I/O protection for creating I/O that is compatible with the environment and transparent to 8086 programs. Designers may take any of several possible approaches to protecting I/O ports:

- Protect the I/O address space and generate exceptions for all attempts to perform I/O directly.
- Let the 8086 program perform I/O directly.
- Generate exceptions on attempts to access specific I/O ports.
- Generate exceptions on attempts to access specific memory-mapped I/O ports.

The method of controlling access to I/O ports depends upon whether they are I/O-port mapped or memory mapped.

21.2.8.1 I/O-Port-Mapped I/O

The I/O permission bit map in the TSS can be used to generate exceptions on attempts to access specific I/O port addresses. The I/O permission bit map of each virtual-8086-mode task determines which I/O addresses generate exceptions for that task. Because each task may have a different I/O permission bit map, the addresses that generate exceptions for one task may be different from the addresses for another task. This differs from protected mode in which, if the CPL is less than or equal to the IOPL, I/O access is allowed without checking the I/O permission bit map. See Chapter 19, "Input/Output," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about the I/O permission bit map.

21.2.8.2 Memory-Mapped I/O

In systems which use memory-mapped I/O, the paging facilities of the processor can be used to generate exceptions for attempts to access I/O ports. The virtual-8086 monitor may use paging to control memory-mapped I/O in these ways:

- Map part of the linear address space of each task that needs to perform I/O to the physical address space where I/O ports are placed. By putting the I/O ports at different addresses (in different pages), the paging mechanism can enforce isolation between tasks.
- Map part of the linear address space to pages that are not-present. This generates an exception whenever a task attempts to perform I/O to those pages. System software then can interpret the I/O operation being attempted.

Software emulation of the I/O space may require too much operating system intervention under some conditions. In these cases, it may be possible to generate an exception for only the first attempt to access I/O. The system software then may determine whether a program can be given exclusive control of I/O temporarily, the protection of the I/O space may be lifted, and the program allowed to run at full speed.

21.2.8.3 Special I/O Buffers

Buffers of intelligent controllers (for example, a bit-mapped frame buffer) also can be emulated using page mapping. The linear space for the buffer can be mapped to a different physical space for each virtual-8086-mode task. The virtual-8086 monitor then can control which virtual buffer to copy onto the real buffer in the physical address space.

21.3 INTERRUPT AND EXCEPTION HANDLING IN VIRTUAL-8086 MODE

When the processor receives an interrupt or detects an exception condition while in virtual-8086 mode, it invokes an interrupt or exception handler, just as it does in protected or real-address mode. The interrupt or exception handler that is invoked and the mechanism used to invoke it depends on the class of interrupt or exception that has been detected or generated and the state of various system flags and fields.

In virtual-8086 mode, the interrupts and exceptions are divided into three classes for the purposes of handling:

- **Class 1** — All processor-generated exceptions and all hardware interrupts, including the NMI interrupt and the hardware interrupts sent to the processor's external interrupt delivery pins. All class 1 exceptions and interrupts are handled by the protected-mode exception and interrupt handlers.
- **Class 2** — Special case for maskable hardware interrupts (Section 6.3.2, "Maskable Hardware Interrupts") when the virtual mode extensions are enabled.
- **Class 3** — All software-generated interrupts, that is interrupts generated with the INT *n* instruction[1].

The method the processor uses to handle class 2 and 3 interrupts depends on the setting of the following flags and fields:

- **IOPL field (bits 12 and 13 in the EFLAGS register)** — Controls how class 3 software interrupts are handled when the processor is in virtual-8086 mode (see Section 2.3, "System Flags and Fields in the EFLAGS

[1] The INT 3 instruction is a special case (see the description of the INT *n* instruction in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A).

Register”). This field also controls the enabling of the VIF and VIP flags in the EFLAGS register when the VME flag is set. The VIF and VIP flags are provided to assist in the handling of class 2 maskable hardware interrupts.

- **VME flag (bit 0 in control register CR4)** — Enables the virtual mode extension for the processor when set (see Section 2.5, “Control Registers”).
- **Software interrupt redirection bit map (32 bytes in the TSS, see Figure 21-5)** — Contains 256 flags that indicates how class 3 software interrupts should be handled when they occur in virtual-8086 mode. A software interrupt can be directed either to the interrupt and exception handlers in the currently running 8086 program or to the protected-mode interrupt and exception handlers.
- **The virtual interrupt flag (VIF) and virtual interrupt pending flag (VIP) in the EFLAGS register** — Provides **virtual interrupt support** for the handling of class 2 maskable hardware interrupts (see Section 21.3.2, “Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism”).

NOTE

The VME flag, software interrupt redirection bit map, and VIF and VIP flags are only available in IA-32 processors that support the virtual mode extensions. These extensions were introduced in the IA-32 architecture with the Pentium processor.

The following sections describe the actions that processor takes and the possible actions of interrupt and exception handlers for the two classes of interrupts described in the previous paragraphs. These sections describe three possible types of interrupt and exception handlers:

- **Protected-mode interrupt and exceptions handlers** — These are the standard handlers that the processor calls through the protected-mode IDT.
- **Virtual-8086 monitor interrupt and exception handlers** — These handlers are resident in the virtual-8086 monitor, and they are commonly accessed through a general-protection exception (#GP, interrupt 13) that is directed to the protected-mode general-protection exception handler.
- **8086 program interrupt and exception handlers** — These handlers are part of the 8086 program that is running in virtual-8086 mode.

The following sections describe how these handlers are used, depending on the selected class and method of interrupt and exception handling.

21.3.1 Class 1—Hardware Interrupt and Exception Handling in Virtual-8086 Mode

In virtual-8086 mode, the Pentium, P6 family, Pentium 4, and Intel Xeon processors handle hardware interrupts and exceptions in the same manner as they are handled by the Intel486 and Intel386 processors. They invoke the protected-mode interrupt or exception handler that the interrupt or exception vector points to in the IDT. Here, the IDT entry must contain either a 32-bit trap or interrupt gate or a task gate. The following sections describe various ways that a virtual-8086 mode interrupt or exception can be handled after the protected-mode handler has been invoked.

See Section 21.3.2, “Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism,” for a description of the virtual interrupt mechanism that is available for handling maskable hardware interrupts while in virtual-8086 mode. When this mechanism is either not available or not enabled, maskable hardware interrupts are handled in the same manner as exceptions, as described in the following sections.

21.3.1.1 Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate

When an interrupt or exception vector points to a 32-bit trap or interrupt gate in the IDT, the gate must in turn point to a nonconforming, privilege-level 0, code segment. When accessing this code segment, processor performs the following steps.

1. Switches to 32-bit protected mode and privilege level 0.
2. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 21-4).

3. Clears the segment registers. Saving the DS, ES, FS, and GS registers on the stack and then clearing the registers lets the interrupt or exception handler safely save and restore these registers regardless of the type segment selectors they contain (protected-mode or 8086-style). The interrupt and exception handlers, which may be called in the context of either a protected-mode task or a virtual-8086-mode task, can use the same code sequences for saving and restoring the registers for any task. Clearing these registers before execution of the IRET instruction does not cause a trap in the interrupt handler. Interrupt procedures that expect values in the segment registers or that return values in the segment registers must use the register images saved on the stack for privilege level 0.
4. Clears VM, NT, RF, and TF flags (in the EFLAGS register). If the gate is an interrupt gate, clears the IF flag.
5. Begins executing the selected interrupt or exception handler.

If the trap or interrupt gate references a procedure in a conforming segment or in a segment at a privilege level other than 0, the processor generates a general-protection exception (#GP). Here, the error code is the segment selector of the code segment to which a call was attempted.

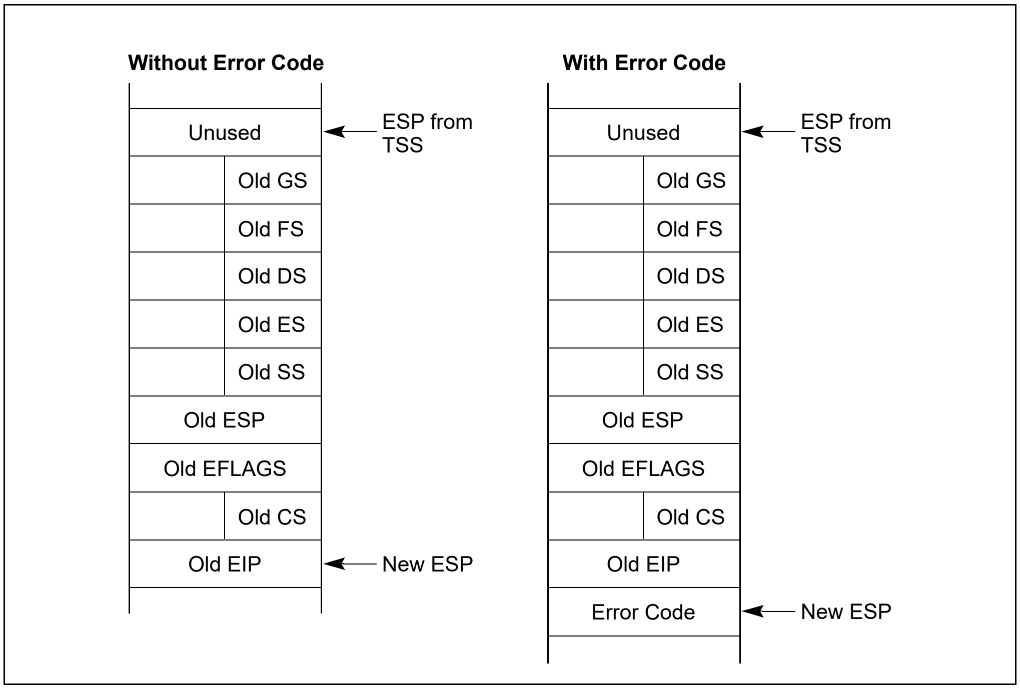


Figure 21-4. Privilege Level 0 Stack After Interrupt or Exception in Virtual-8086 Mode

Interrupt and exception handlers can examine the VM flag on the stack to determine if the interrupted procedure was running in virtual-8086 mode. If so, the interrupt or exception can be handled in one of three ways:

- The protected-mode interrupt or exception handler that was called can handle the interrupt or exception.
- The protected-mode interrupt or exception handler can call the virtual-8086 monitor to handle the interrupt or exception.
- The virtual-8086 monitor (if called) can in turn pass control back to the 8086 program’s interrupt and exception handler.

If the interrupt or exception is handled with a protected-mode handler, the handler can return to the interrupted program in virtual-8086 mode by executing an IRET instruction. This instruction loads the EFLAGS and segment registers from the images saved in the privilege level 0 stack (see Figure 21-4). A set VM flag in the EFLAGS image causes the processor to switch back to virtual-8086 mode. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.

The virtual-8086 monitor runs at privilege level 0, like the protected-mode interrupt and exception handlers. It is commonly closely tied to the protected-mode general-protection exception (#GP, vector 13) handler. If the protected-mode interrupt or exception handler calls the virtual-8086 monitor to handle the interrupt or exception, the return from the virtual-8086 monitor to the interrupted virtual-8086 mode program requires two return instructions: a RET instruction to return to the protected-mode handler and an IRET instruction to return to the interrupted program.

The virtual-8086 monitor has the option of directing the interrupt and exception back to an interrupt or exception handler that is part of the interrupted 8086 program, as described in Section 21.3.1.2, “Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler.”

21.3.1.2 Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler

Because it was designed to run on an 8086 processor, an 8086 program running in a virtual-8086-mode task contains an 8086-style interrupt vector table, which starts at linear address 0. If the virtual-8086 monitor correctly directs an interrupt or exception vector back to the virtual-8086-mode task it came from, the handlers in the 8086 program can handle the interrupt or exception. The virtual-8086 monitor must carry out the following steps to send an interrupt or exception back to the 8086 program:

1. Use the 8086 interrupt vector to locate the appropriate handler procedure in the 8086 program interrupt table.
2. Store the EFLAGS (low-order 16 bits only), CS and EIP values of the 8086 program on the privilege-level 3 stack. This is the stack that the virtual-8086-mode task is using. (The 8086 handler may use or modify this information.)
3. Change the return link on the privilege-level 0 stack to point to the privilege-level 3 handler procedure.
4. Execute an IRET instruction to pass control to the 8086 program handler.
5. When the IRET instruction from the privilege-level 3 handler triggers a general-protection exception (#GP) and thus effectively again calls the virtual-8086 monitor, restore the return link on the privilege-level 0 stack to point to the original, interrupted, privilege-level 3 procedure.
6. Copy the low order 16 bits of the EFLAGS image from the privilege-level 3 stack to the privilege-level 0 stack (because some 8086 handlers modify these flags to return information to the code that caused the interrupt).
7. Execute an IRET instruction to pass control back to the interrupted 8086 program.

Note that if an operating system intends to support all 8086 MS-DOS-based programs, it is necessary to use the actual 8086 interrupt and exception handlers supplied with the program. The reason for this is that some programs modify their own interrupt vector table to substitute (or hook in series) their own specialized interrupt and exception handlers.

21.3.1.3 Handling an Interrupt or Exception Through a Task Gate

When an interrupt or exception vector points to a task gate in the IDT, the processor performs a task switch to the selected interrupt- or exception-handling task. The following actions are carried out as part of this task switch:

1. The EFLAGS register with the VM flag set is saved in the current TSS.
2. The link field in the TSS of the called task is loaded with the segment selector of the TSS for the interrupted virtual-8086-mode task.
3. The EFLAGS register is loaded from the image in the new TSS, which clears the VM flag and causes the processor to switch to protected mode.
4. The NT flag in the EFLAGS register is set.
5. The processor begins executing the selected interrupt- or exception-handler task.

When an IRET instruction is executed in the handler task and the NT flag in the EFLAGS register is set, the processors switches from a protected-mode interrupt- or exception-handler task back to a virtual-8086-mode task. Here, the EFLAGS and segment registers are loaded from images saved in the TSS for the virtual-8086-mode task. If the VM flag is set in the EFLAGS image, the processor switches back to virtual-8086 mode on the task switch. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.

21.3.2 Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism

Maskable hardware interrupts are those interrupts that are delivered through the INTR# pin or through an interrupt request to the local APIC (see Section 6.3.2, “Maskable Hardware Interrupts”). These interrupts can be inhibited (masked) from interrupting an executing program or task by clearing the IF flag in the EFLAGS register.

When the VME flag in control register CR4 is set and the IOPL field in the EFLAGS register is less than 3, two additional flags are activated in the EFLAGS register:

- VIF (virtual interrupt) flag, bit 19 of the EFLAGS register.
- VIP (virtual interrupt pending) flag, bit 20 of the EFLAGS register.

These flags provide the virtual-8086 monitor with more efficient control over handling maskable hardware interrupts that occur during virtual-8086 mode tasks. They also reduce interrupt-handling overhead, by eliminating the need for all IF related operations (such as PUSHF, POPF, CLI, and STI instructions) to trap to the virtual-8086 monitor. The purpose and use of these flags are as follows.

NOTE

> The VIF and VIP flags are only available in IA-32 processors that support the virtual mode extensions. These extensions were introduced in the IA-32 architecture with the Pentium processor. When this mechanism is either not available or not enabled, maskable hardware interrupts are handled as class 1 interrupts. Here, if VIF and VIP flags are needed, the virtual-8086 monitor can implement them in software.

Existing 8086 programs commonly set and clear the IF flag in the EFLAGS register to enable and disable maskable hardware interrupts, respectively; for example, to disable interrupts while handling another interrupt or an exception. This practice works well in single task environments, but can cause problems in multitasking and multiple-processor environments, where it is often desirable to prevent an application program from having direct control over the handling of hardware interrupts. When using earlier IA-32 processors, this problem was often solved by creating a virtual IF flag in software. The IA-32 processors (beginning with the Pentium processor) provide hardware support for this virtual IF flag through the VIF and VIP flags.

The VIF flag is a virtualized version of the IF flag, which an application program running from within a virtual-8086 task can used to control the handling of maskable hardware interrupts. When the VIF flag is enabled, the CLI and STI instructions operate on the VIF flag instead of the IF flag. When an 8086 program executes the CLI instruction, the processor clears the VIF flag to request that the virtual-8086 monitor inhibit maskable hardware interrupts from interrupting program execution; when it executes the STI instruction, the processor sets the VIF flag requesting that the virtual-8086 monitor enable maskable hardware interrupts for the 8086 program. But actually the IF flag, managed by the operating system, always controls whether maskable hardware interrupts are enabled. Also, if under these circumstances an 8086 program tries to read or change the IF flag using the PUSHF or POPF instructions, the processor will change the VIF flag instead, leaving IF unchanged.

The VIP flag provides software a means of recording the existence of a deferred (or pending) maskable hardware interrupt. This flag is read by the processor but never explicitly written by the processor; it can only be written by software.

If the IF flag is set and the VIF and VIP flags are enabled, and the processor receives a maskable hardware interrupt (interrupt vector 0 through 255), the processor performs and the interrupt handler software should perform the following operations:

1. The processor invokes the protected-mode interrupt handler for the interrupt received, as described in the following steps. These steps are almost identical to those described for method 1 interrupt and exception handling in Section 21.3.1.1, “Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate”:
   a. Switches to 32-bit protected mode and privilege level 0.
   b. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 21-4).
   c. Clears the segment registers.

   d. Clears the VM flag in the EFLAGS register.

   e. Begins executing the selected protected-mode interrupt handler.

2. The recommended action of the protected-mode interrupt handler is to read the VM flag from the EFLAGS image on the stack. If this flag is set, the handler makes a call to the virtual-8086 monitor.

3. The virtual-8086 monitor should read the VIF flag in the EFLAGS register.

   — If the VIF flag is clear, the virtual-8086 monitor sets the VIP flag in the EFLAGS image on the stack to indicate that there is a deferred interrupt pending and returns to the protected-mode handler.

   — If the VIF flag is set, the virtual-8086 monitor can handle the interrupt if it "belongs" to the 8086 program running in the interrupted virtual-8086 task; otherwise, it can call the protected-mode interrupt handler to handle the interrupt.

4. The protected-mode handler executes a return to the program executing in virtual-8086 mode.

5. Upon returning to virtual-8086 mode, the processor continues execution of the 8086 program.

When the 8086 program is ready to receive maskable hardware interrupts, it executes the STI instruction to set the VIF flag (enabling maskable hardware interrupts). Prior to setting the VIF flag, the processor automatically checks the VIP flag and does one of the following, depending on the state of the flag:

- If the VIP flag is clear (indicating no pending interrupts), the processor sets the VIF flag.
- If the VIP flag is set (indicating a pending interrupt), the processor generates a general-protection exception (#GP).

The recommended action of the protected-mode general-protection exception handler is to then call the virtual-8086 monitor and let it handle the pending interrupt. After handling the pending interrupt, the typical action of the virtual-8086 monitor is to clear the VIP flag and set the VIF flag in the EFLAGS image on the stack, and then execute a return to the virtual-8086 mode. The next time the processor receives a maskable hardware interrupt, it will then handle it as described in steps 1 through 5 earlier in this section.

If the processor finds that both the VIF and VIP flags are set at the beginning of an instruction, it generates a general-protection exception. This action allows the virtual-8086 monitor to handle the pending interrupt for the virtual-8086 mode task for which the VIF flag is enabled. Note that this situation can only occur immediately following execution of a POPF or IRET instruction or upon entering a virtual-8086 mode task through a task switch.

Note that the states of the VIF and VIP flags are not modified in real-address mode or during transitions between real-address and protected modes.

NOTE

The virtual interrupt mechanism described in this section is also available for use in protected mode, see Section 21.4, "Protected-Mode Virtual Interrupts."

21.3.3 Class 3—Software Interrupt Handling in Virtual-8086 Mode

When the processor receives a software interrupt (an interrupt generated with the INT *n* instruction) while in virtual-8086 mode, it can use any of six different methods to handle the interrupt. The method selected depends on the settings of the VME flag in control register CR4, the IOPL field in the EFLAGS register, and the software interrupt redirection bit map in the TSS. Table 21-2 lists the six methods of handling software interrupts in virtual-8086 mode and the respective settings of the VME flag, IOPL field, and the bits in the interrupt redirection bit map for each method. The table also summarizes the various actions the processor takes for each method.

The VME flag enables the virtual mode extensions for the Pentium and later IA-32 processors. When this flag is clear, the processor responds to interrupts and exceptions in virtual-8086 mode in the same manner as an Intel386 or Intel486 processor does. When this flag is set, the virtual mode extension provides the following enhancements to virtual-8086 mode:

- Speeds up the handling of software-generated interrupts in virtual-8086 mode by allowing the processor to bypass the virtual-8086 monitor and redirect software interrupts back to the interrupt handlers that are part of the currently running 8086 program.
- Supports virtual interrupts for software written to run on the 8086 processor.

The IOPL value interacts with the VME flag and the bits in the interrupt redirection bit map to determine how specific software interrupts should be handled.

The software interrupt redirection bit map (see Figure 21-5) is a 32-byte field in the TSS. This map is located directly below the I/O permission bit map in the TSS. Each bit in the interrupt redirection bit map is mapped to an interrupt vector. Bit 0 in the interrupt redirection bit map (which maps to vector zero in the interrupt table) is located at the I/O base map address in the TSS minus 32 bytes. When a bit in this bit map is set, it indicates that the associated software interrupt (interrupt generated with an INT *n* instruction) should be handled through the protected-mode IDT and interrupt and exception handlers. When a bit in this bit map is clear, the processor redirects the associated software interrupt back to the interrupt table in the 8086 program (located at linear address 0 in the program's address space).

NOTE

The software interrupt redirection bit map does not affect hardware generated interrupts and exceptions. Hardware generated interrupts and exceptions are always handled by the protected-mode interrupt and exception handlers.

Table 21-2. Software Interrupt Handling Methods While in Virtual-8086 Mode

Method	VME	IOPL	Bit in Redir. Bitmap*	Processor Action
1	0	3	X	Interrupt directed to a protected-mode interrupt handler: ▪ Switches to privilege-level 0 stack. ▪ Pushes GS, FS, DS, and ES onto privilege-level 0 stack. ▪ Pushes SS, ESP, EFLAGS, CS, and EIP of interrupted task onto privilege-level 0 stack. ▪ Clears VM, RF, NT, and TF flags. ▪ If serviced through interrupt gate, clears IF flag. ▪ Clears GS, FS, DS, and ES to 0. ▪ Sets CS and EIP from interrupt gate.
2	0	< 3	X	Interrupt directed to protected-mode general-protection exception (#GP) handler.
3	1	< 3	1	Interrupt directed to a protected-mode general-protection exception (#GP) handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts.
4	1	3	1	Interrupt directed to protected-mode interrupt handler: (see method 1 processor action).
5	1	3	0	Interrupt redirected to 8086 program interrupt handler: ▪ Pushes EFLAGS. ▪ Pushes CS and EIP (lower 16 bits only). ▪ Clears IF flag. ▪ Clears TF flag. ▪ Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task.
6	1	< 3	0	Interrupt redirected to 8086 program interrupt handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts: ▪ Pushes EFLAGS with IOPL set to 3 and VIF copied to IF. ▪ Pushes CS and EIP (lower 16 bits only). ▪ Clears the VIF flag. ▪ Clears TF flag. ▪ Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task.

NOTE:

* When set to 0, software interrupt is redirected back to the 8086 program interrupt handler; when set to 1, interrupt is directed to protected-mode handler.

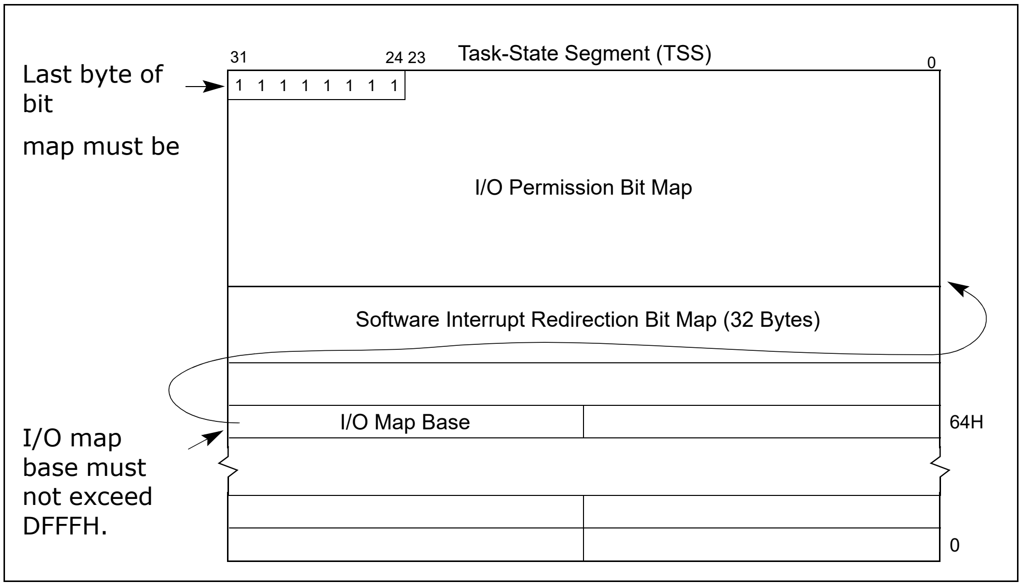


Figure 21-5. Software Interrupt Redirection Bit Map in TSS

Redirecting software interrupts back to the 8086 program potentially speeds up interrupt handling because a switch back and forth between virtual-8086 mode and protected mode is not required. This latter interrupt-handling technique is particularly useful for 8086 operating systems (such as MS-DOS) that use the INT *n* instruction to call operating system procedures.

The CPUID instruction can be used to verify that the virtual mode extension is implemented on the processor. Bit 1 of the feature flags register (EDX) indicates the availability of the virtual mode extension (see "CPUID—CPU Identification" in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A).

The following sections describe the six methods (or mechanisms) for handling software interrupts in virtual-8086 mode. See Section 21.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism," for a description of the use of the VIF and VIP flags in the EFLAGS register for handling maskable hardware interrupts.

21.3.3.1 Method 1: Software Interrupt Handling

When the VME flag in control register CR4 is clear and the IOPL field is 3, a Pentium or later IA-32 processor handles software interrupts in the same manner as they are handled by an Intel386 or Intel486 processor. It executes an implicit call to the interrupt handler in the protected-mode IDT pointed to by the interrupt vector. See Section 21.3.1, "Class 1—Hardware Interrupt and Exception Handling in Virtual-8086 Mode," for a complete description of this mechanism and its possible uses.

21.3.3.2 Methods 2 and 3: Software Interrupt Handling

When a software interrupt occurs in virtual-8086 mode and the method 2 or 3 conditions are present, the processor generates a general-protection exception (#GP). Method 2 is enabled when the VME flag is set to 0 and the IOPL value is less than 3. Here the IOPL value is used to bypass the protected-mode interrupt handlers and cause any software interrupt that occurs in virtual-8086 mode to be treated as a protected-mode general-protection exception (#GP). The general-protection exception handler calls the virtual-8086 monitor, which can then emulate an 8086-program interrupt handler or pass control back to the 8086 program's handler, as described in Section 21.3.1.2, "Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler."

Method 3 is enabled when the VME flag is set to 1, the IOPL value is less than 3, and the corresponding bit for the software interrupt in the software interrupt redirection bit map is set to 1. Here, the processor performs the same operation as it does for method 2 software interrupt handling. If the corresponding bit for the software interrupt in the software interrupt redirection bit map is set to 0, the interrupt is handled using method 6 (see Section 21.3.3.5, "Method 6: Software Interrupt Handling").

21.3.3.3 Method 4: Software Interrupt Handling

Method 4 handling is enabled when the VME flag is set to 1, the IOPL value is 3, and the bit for the interrupt vector in the redirection bit map is set to 1. Method 4 software interrupt handling allows method 1 style handling when the virtual mode extension is enabled; that is, the interrupt is directed to a protected-mode handler (see Section 21.3.3.1, "Method 1: Software Interrupt Handling").

21.3.3.4 Method 5: Software Interrupt Handling

Method 5 software interrupt handling provides a streamlined method of redirecting software interrupts (invoked with the INT *n* instruction) that occur in virtual 8086 mode back to the 8086 program's interrupt vector table and its interrupt handlers. Method 5 handling is enabled when the VME flag is set to 1, the IOPL value is 3, and the bit for the interrupt vector in the redirection bit map is set to 0. The processor performs the following actions to make an implicit call to the selected 8086 program interrupt handler:

1. Pushes the low-order 16 bits of the EFLAGS register onto the stack.
2. Pushes the current values of the CS and EIP registers onto the current stack. (Only the 16 least-significant bits of the EIP register are pushed and no stack switch occurs.)
3. Clears the IF flag in the EFLAGS register to disable interrupts.
4. Clears the TF flag, in the EFLAGS register.
5. Locates the 8086 program interrupt vector table at linear address 0 for the 8086-mode task.
6. Loads the CS and EIP registers with values from the interrupt vector table entry pointed to by the interrupt vector number. Only the 16 low-order bits of the EIP are loaded and the 16 high-order bits are set to 0. The interrupt vector table is assumed to be at linear address 0 of the current virtual-8086 task.
7. Begins executing the selected interrupt handler.

An IRET instruction at the end of the handler procedure reverses these steps to return program control to the interrupted 8086 program.

Note that with method 5 handling, a mode switch from virtual-8086 mode to protected mode does not occur. The processor remains in virtual-8086 mode throughout the interrupt-handling operation.

The method 5 handling actions are virtually identical to the actions the processor takes when handling software interrupts in real-address mode. The benefit of using method 5 handling to access the 8086 program handlers is that it avoids the overhead of methods 2 and 3 handling, which requires first going to the virtual-8086 monitor, then to the 8086 program handler, then back again to the virtual-8086 monitor, before returning to the interrupted 8086 program (see Section 21.3.1.2, "Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler").

NOTE

Methods 1 and 4 handling can handle a software interrupt in a virtual-8086 task with a regular protected-mode handler, but this approach requires all virtual-8086 tasks to use the same software interrupt handlers, which generally does not give sufficient latitude to the programs running in the virtual-8086 tasks, particularly MS-DOS programs.

21.3.3.5 Method 6: Software Interrupt Handling

Method 6 handling is enabled when the VME flag is set to 1, the IOPL value is less than 3, and the bit for the interrupt or exception vector in the redirection bit map is set to 0. With method 6 interrupt handling, software interrupts are handled in the same manner as was described for method 5 handling (see Section 21.3.3.4, "Method 5: Software Interrupt Handling").

Method 6 differs from method 5 in that with the IOPL value set to less than 3, the VIF and VIP flags in the EFLAGS register are enabled, providing virtual interrupt support for handling class 2 maskable hardware interrupts (see Section 21.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism"). These flags provide the virtual-8086 monitor with an efficient means of handling maskable hardware interrupts that occur during a virtual-8086 mode task. Also, because the IOPL value is less than 3 and the VIF flag is enabled, the information pushed on the stack by the processor when invoking the interrupt handler is slightly different between methods 5 and 6 (see Table 21-2).

21.4 PROTECTED-MODE VIRTUAL INTERRUPTS

The IA-32 processors (beginning with the Pentium processor) also support the VIF and VIP flags in the EFLAGS register in protected mode by setting the PVI (protected-mode virtual interrupt) flag in the CR4 register. Setting the PVI flag allows applications running at privilege level 3 to execute the CLI and STI instructions without causing a general-protection exception (#GP) or affecting hardware interrupts.

When the PVI flag is set to 1, the CPL is 3, and the IOPL is less than 3, the STI and CLI instructions set and clear the VIF flag in the EFLAGS register, leaving IF unaffected. In this mode of operation, an application running in protected mode and at a CPL of 3 can inhibit interrupts in the same manner as is described in Section 21.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism," for a virtual-8086 mode task. When the application executes the CLI instruction, the processor clears the VIF flag. If the processor receives a maskable hardware interrupt, the processor invokes the protected-mode interrupt handler. This handler checks the state of the VIF flag in the EFLAGS register. If the VIF flag is clear (indicating that the active task does not want to have interrupts handled now), the handler sets the VIP flag in the EFLAGS image on the stack and returns to the privilege-level 3 application, which continues program execution. When the application executes a STI instruction to set the VIF flag, the processor automatically invokes the general-protection exception handler, which can then handle the pending interrupt. After handing the pending interrupt, the handler typically sets the VIF flag and clears the VIP flag in the EFLAGS image on the stack and executes a return to the application program. The next time the processor receives a maskable hardware interrupt, the processor will handle it in the normal manner for interrupts received while the processor is operating at a CPL of 3.

If the protected-mode virtual interrupt extension is enabled, CPL = 3, and the processor finds that both the VIF and VIP flags are set at the beginning of an instruction, a general-protection exception is generated.

Because the protected-mode virtual interrupt extension changes only the treatment of EFLAGS.IF (by having CLI and STI update EFLAGS.VIF instead), it affects only the masking of maskable hardware interrupts (interrupt vectors 32 through 255). NMI interrupts and exceptions are handled in the normal manner.

(When protected-mode virtual interrupts are disabled (that is, when the PVI flag in control register CR4 is set to 0, the CPL is less than 3, or the IOPL value is 3), then the CLI and STI instructions execute in a manner compatible with the Intel486 processor. That is, if the CPL is greater (less privileged) than the I/O privilege level (IOPL), a general-protection exception occurs. If the IOPL value is 3, CLI and STI clear or set the IF flag, respectively.)

PUSHF, POPF, IRET, and INT are executed like in the Intel486 processor, regardless of whether protected-mode virtual interrupts are enabled.

It is only possible to enter virtual-8086 mode through a task switch or the execution of an IRET instruction, and it is only possible to leave virtual-8086 mode by faulting to a protected-mode interrupt handler (typically the general-protection exception handler, which in turn calls the virtual 8086-mode monitor). In both cases, the EFLAGS register is saved and restored. This is not true, however, in protected mode when the PVI flag is set and the processor is not in virtual-8086 mode. Here, it is possible to call a procedure at a different privilege level, in which case the EFLAGS register is not saved or modified. However, the states of VIF and VIP flags are never examined by the processor when the CPL is not 3.

CHAPTER 22
MIXING 16-BIT AND 32-BIT CODE

Program modules written to run on IA-32 processors can be either 16-bit modules or 32-bit modules. Table 22-1 shows the characteristic of 16-bit and 32-bit modules.

Table 22-1. Characteristics of 16-Bit and 32-Bit Program Modules

Characteristic	16-Bit Program Modules	32-Bit Program Modules
Segment Size	0 to 64 KBytes	0 to 4 GBytes
Operand Sizes	8 bits and 16 bits	8 bits and 32 bits
Pointer Offset Size (Address Size)	16 bits	32 bits
Stack Pointer Size	16 Bits	32 Bits
Control Transfers Allowed to Code Segments of This Size	16 Bits	32 Bits

The IA-32 processors function most efficiently when executing 32-bit program modules. They can, however, also execute 16-bit program modules, in any of the following ways:

- In real-address mode.
- In virtual-8086 mode.
- System management mode (SMM).
- As a protected-mode task, when the code, data, and stack segments for the task are all configured as a 16-bit segments.
- By integrating 16-bit and 32-bit segments into a single protected-mode task.
- By integrating 16-bit operations into 32-bit code segments.

Real-address mode, virtual-8086 mode, and SMM are native 16-bit modes. A legacy program assembled and/or compiled to run on an Intel 8086 or Intel 286 processor should run in real-address mode or virtual-8086 mode without modification. Sixteen-bit program modules can also be written to run in real-address mode for handling system initialization or to run in SMM for handling system management functions. See Chapter 21, "8086 Emulation," for detailed information on real-address mode and virtual-8086 mode; see Chapter 32, "System Management Mode," for information on SMM.

This chapter describes how to integrate 16-bit program modules with 32-bit program modules when operating in protected mode and how to mix 16-bit and 32-bit code within 32-bit code segments.

22.1 DEFINING 16-BIT AND 32-BIT PROGRAM MODULES

The following IA-32 architecture mechanisms are used to distinguish between and support 16-bit and 32-bit segments and operations:

- The D (default operand and address size) flag in code-segment descriptors.
- The B (default stack size) flag in stack-segment descriptors.
- 16-bit and 32-bit call gates, interrupt gates, and trap gates.
- Operand-size and address-size instruction prefixes.
- 16-bit and 32-bit general-purpose registers.

The D flag in a code-segment descriptor determines the default operand-size and address-size for the instructions of a code segment. (In real-address mode and virtual-8086 mode, which do not use segment descriptors, the default is 16 bits.) A code segment with its D flag set is a 32-bit segment; a code segment with its D flag clear is a 16-bit segment.

The B flag in the stack-segment descriptor specifies the size of stack pointer (the 32-bit ESP register or the 16-bit SP register) used by the processor for implicit stack references. The B flag for all data descriptors also controls upper address range for expand down segments.

When transferring program control to another code segment through a call gate, interrupt gate, or trap gate, the operand size used during the transfer is determined by the type of gate used (16-bit or 32-bit), (not by the D-flag or prefix of the transfer instruction). The gate type determines how return information is saved on the stack (or stacks).

For most efficient and trouble-free operation of the processor, 32-bit programs or tasks should have the D flag in the code-segment descriptor and the B flag in the stack-segment descriptor set, and 16-bit programs or tasks should have these flags clear. Program control transfers from 16-bit segments to 32-bit segments (and vice versa) are handled most efficiently through call, interrupt, or trap gates.

Instruction prefixes can be used to override the default operand size and address size of a code segment. These prefixes can be used in real-address mode as well as in protected mode and virtual-8086 mode. An operand-size or address-size prefix only changes the size for the duration of the instruction.

22.2 MIXING 16-BIT AND 32-BIT OPERATIONS WITHIN A CODE SEGMENT

The following two instruction prefixes allow mixing of 32-bit and 16-bit operations within one segment:

- The operand-size prefix (66H)
- The address-size prefix (67H)

These prefixes reverse the default size selected by the D flag in the code-segment descriptor. For example, the processor can interpret the (MOV *mem*, *reg*) instruction in any of four ways:

- In a 32-bit code segment:
  - Moves 32 bits from a 32-bit register to memory using a 32-bit effective address.
  - If preceded by an operand-size prefix, moves 16 bits from a 16-bit register to memory using a 32-bit effective address.
  - If preceded by an address-size prefix, moves 32 bits from a 32-bit register to memory using a 16-bit effective address.
  - If preceded by both an address-size prefix and an operand-size prefix, moves 16 bits from a 16-bit register to memory using a 16-bit effective address.
- In a 16-bit code segment:
  - Moves 16 bits from a 16-bit register to memory using a 16-bit effective address.
  - If preceded by an operand-size prefix, moves 32 bits from a 32-bit register to memory using a 16-bit effective address.
  - If preceded by an address-size prefix, moves 16 bits from a 16-bit register to memory using a 32-bit effective address.
  - If preceded by both an address-size prefix and an operand-size prefix, moves 32 bits from a 32-bit register to memory using a 32-bit effective address.

The previous examples show that any instruction can generate any combination of operand size and address size regardless of whether the instruction is in a 16- or 32-bit segment. The choice of the 16- or 32-bit default for a code segment is normally based on the following criteria:

- **Performance** — Always use 32-bit code segments when possible. They run much faster than 16-bit code segments on P6 family processors, and somewhat faster on earlier IA-32 processors.
- **The operating system the code segment will be running on** — If the operating system is a 16-bit operating system, it may not support 32-bit program modules.
- **Mode of operation** — If the code segment is being designed to run in real-address mode, virtual-8086 mode, or SMM, it must be a 16-bit code segment.

- **Backward compatibility to earlier IA-32 processors** — If a code segment must be able to run on an Intel 8086 or Intel 286 processor, it must be a 16-bit code segment.

22.3 SHARING DATA AMONG MIXED-SIZE CODE SEGMENTS

Data segments can be accessed from both 16-bit and 32-bit code segments. When a data segment that is larger than 64 KBytes is to be shared among 16- and 32-bit code segments, the data that is to be accessed from the 16-bit code segments must be located within the first 64 KBytes of the data segment. The reason for this is that 16-bit pointers by definition can only point to the first 64 KBytes of a segment.

A stack that spans less than 64 KBytes can be shared by both 16- and 32-bit code segments. This class of stacks includes:

- Stacks in expand-up segments with the G (granularity) and B (big) flags in the stack-segment descriptor clear.
- Stacks in expand-down segments with the G and B flags clear.
- Stacks in expand-up segments with the G flag set and the B flag clear and where the stack is contained completely within the lower 64 KBytes. (Offsets greater than FFFFH can be used for data, other than the stack, which is not shared.)

See Section 3.4.5, "Segment Descriptors," for a description of the G and B flags and the expand-down stack type.

The B flag cannot, in general, be used to change the size of stack used by a 16-bit code segment. This flag controls the size of the stack pointer only for implicit stack references such as those caused by interrupts, exceptions, and the PUSH, POP, CALL, and RET instructions. It does not control explicit stack references, such as accesses to parameters or local variables. A 16-bit code segment can use a 32-bit stack only if the code is modified so that all explicit references to the stack are preceded by the 32-bit address-size prefix, causing those references to use 32-bit addressing and explicit writes to the stack pointer are preceded by a 32-bit operand-size prefix.

In 32-bit, expand-down segments, all offsets may be greater than 64 KBytes; therefore, 16-bit code cannot use this kind of stack segment unless the code segment is modified to use 32-bit addressing.

22.4 TRANSFERRING CONTROL AMONG MIXED-SIZE CODE SEGMENTS

There are three ways for a procedure in a 16-bit code segment to safely make a call to a 32-bit code segment:

- Make the call through a 32-bit call gate.
- Make a 16-bit call to a 32-bit interface procedure. The interface procedure then makes a 32-bit call to the intended destination.
- Modify the 16-bit procedure, inserting an operand-size prefix before the call, to change it to a 32-bit call.

Likewise, there are three ways for procedure in a 32-bit code segment to safely make a call to a 16-bit code segment:

- Make the call through a 16-bit call gate. Here, the EIP value at the CALL instruction cannot exceed FFFFH.
- Make a 32-bit call to a 16-bit interface procedure. The interface procedure then makes a 16-bit call to the intended destination.
- Modify the 32-bit procedure, inserting an operand-size prefix before the call, changing it to a 16-bit call. Be certain that the return offset does not exceed FFFFH.

These methods of transferring program control overcome the following architectural limitations imposed on calls between 16-bit and 32-bit code segments:

- Pointers from 16-bit code segments (which by default can only be 16 bits) cannot be used to address data or code located beyond FFFFH in a 32-bit segment.
- The operand-size attributes for a CALL and its companion RETURN instruction must be the same to maintain stack coherency. This is also true for implicit calls to interrupt and exception handlers and their companion IRET instructions.
- A 32-bit parameters (particularly a pointer parameter) greater than FFFFH cannot be squeezed into a 16-bit parameter location on a stack.

- The size of the stack pointer (SP or ESP) changes when switching between 16-bit and 32-bit code segments.

These limitations are discussed in greater detail in the following sections.

22.4.1 Code-Segment Pointer Size

For control-transfer instructions that use a pointer to identify the next instruction (that is, those that do not use gates), the operand-size attribute determines the size of the offset portion of the pointer. The implications of this rule are as follows:

- A JMP, CALL, or RET instruction from a 32-bit segment to a 16-bit segment is always possible using a 32-bit operand size, providing the 32-bit pointer does not exceed FFFFH.
- A JMP, CALL, or RET instruction from a 16-bit segment to a 32-bit segment cannot address a destination greater than FFFFH, unless the instruction is given an operand-size prefix.

See Section 22.4.5, "Writing Interface Procedures," for an interface procedure that can transfer program control from 16-bit segments to destinations in 32-bit segments beyond FFFFH.

22.4.2 Stack Management for Control Transfer

Because the stack is managed differently for 16-bit procedure calls than for 32-bit calls, the operand-size attribute of the RET instruction must match that of the CALL instruction (see Figure 22-1). On a 16-bit call, the processor pushes the contents of the 16-bit IP register and (for calls between privilege levels) the 16-bit SP register. The matching RET instruction must also use a 16-bit operand size to pop these 16-bit values from the stack into the 16-bit registers.

A 32-bit CALL instruction pushes the contents of the 32-bit EIP register and (for inter-privilege-level calls) the 32-bit ESP register. Here, the matching RET instruction must use a 32-bit operand size to pop these 32-bit values from the stack into the 32-bit registers. If the two parts of a CALL/RET instruction pair do not have matching operand sizes, the stack will not be managed correctly and the values of the instruction pointer and stack pointer will not be restored to correct values.

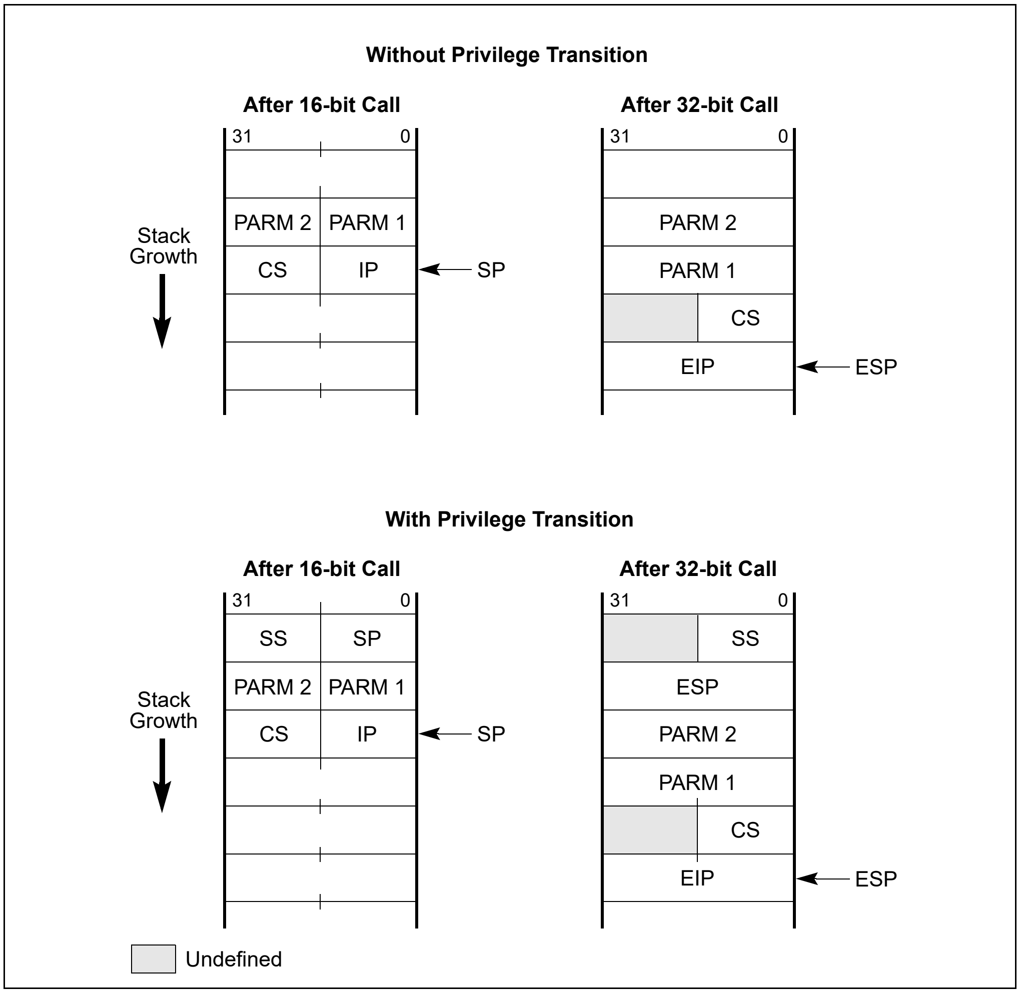


Figure 22-1. Stack after Far 16- and 32-Bit Calls

While executing 32-bit code, if a call is made to a 16-bit code segment which is at the same or a more privileged level (that is, the DPL of the called code segment is less than or equal to the CPL of the calling code segment) through a 16-bit call gate, then the upper 16-bits of the ESP register may be unreliable upon returning to the 32-bit code segment (that is, after executing a RET in the 16-bit code segment).

When the CALL instruction and its matching RET instruction are in code segments that have D flags with the same values (that is, both are 32-bit code segments or both are 16-bit code segments), the default settings may be used. When the CALL instruction and its matching RET instruction are in segments which have different D-flag settings, an operand-size prefix must be used.

22.4.2.1 Controlling the Operand-Size Attribute For a Call

Three things can determine the operand-size of a call:

- The D flag in the segment descriptor for the calling code segment.
- An operand-size instruction prefix.
- The type of call gate (16-bit or 32-bit), if a call is made through a call gate.

When a call is made with a pointer (rather than a call gate), the D flag for the calling code segment determines the operand-size for the CALL instruction. This operand-size attribute can be overridden by prepending an operand-size prefix to the CALL instruction. So, for example, if the D flag for a code segment is set for 16 bits and the operand-size prefix is used with a CALL instruction, the processor will cause the information stored on the stack to be stored in 32-bit format. If the call is to a 32-bit code segment, the instructions in that code segment will be able to read the stack coherently. Also, a RET instruction from the 32-bit code segment without an operand-size prefix will maintain stack coherency with the 16-bit code segment being returned to.

When a CALL instruction references a call-gate descriptor, the type of call is determined by the type of call gate (16-bit or 32-bit). The offset to the destination in the code segment being called is taken from the gate descriptor; therefore, if a 32-bit call gate is used, a procedure in a 16-bit code segment can call a procedure located more than 64 KBytes from the base of a 32-bit code segment, because a 32-bit call gate uses a 32-bit offset.

Note that regardless of the operand size of the call and how it is determined, the size of the stack pointer used (SP or ESP) is always controlled by the B flag in the stack-segment descriptor currently in use (that is, when B is clear, SP is used, and when B is set, ESP is used).

An unmodified 16-bit code segment that has run successfully on an 8086 processor or in real-mode on a later IA-32 architecture processor will have its D flag clear and will not use operand-size override prefixes. As a result, all CALL instructions in this code segment will use the 16-bit operand-size attribute. Procedures in these code segments can be modified to safely call procedures to 32-bit code segments in either of two ways:

- Relink the CALL instruction to point to 32-bit call gates (see Section 22.4.2.2, “Passing Parameters With a Gate”).
- Add a 32-bit operand-size prefix to each CALL instruction.

22.4.2.2 Passing Parameters With a Gate

When referencing 32-bit gates with 16-bit procedures, it is important to consider the number of parameters passed in each procedure call. The count field of the gate descriptor specifies the size of the parameter string to copy from the current stack to the stack of a more privileged (numerically lower privilege level) procedure. The count field of a 16-bit gate specifies the number of 16-bit words to be copied, whereas the count field of a 32-bit gate specifies the number of 32-bit doublewords to be copied. The count field for a 32-bit gate must thus be half the size of the number of words being placed on the stack by a 16-bit procedure. Also, the 16-bit procedure must use an even number of words as parameters.

22.4.3 Interrupt Control Transfers

A program-control transfer caused by an exception or interrupt is always carried out through an interrupt or trap gate (located in the IDT). Here, the type of the gate (16-bit or 32-bit) determines the operand-size attribute used in the implicit call to the exception or interrupt handler procedure in another code segment.

A 32-bit interrupt or trap gate provides a safe interface to a 32-bit exception or interrupt handler when the exception or interrupt occurs in either a 32-bit or a 16-bit code segment. It is sometimes impractical, however, to place exception or interrupt handlers in 16-bit code segments, because only 16-bit return addresses are saved on the stack. If an exception or interrupt occurs in a 32-bit code segment when the EIP was greater than FFFFH, the 16-bit handler procedure cannot provide the correct return address.

22.4.4 Parameter Translation

When segment offsets or pointers (which contain segment offsets) are passed as parameters between 16-bit and 32-bit procedures, some translation is required. If a 32-bit procedure passes a pointer to data located beyond 64 KBytes to a 16-bit procedure, the 16-bit procedure cannot use it. Except for this limitation, interface code can perform any format conversion between 32-bit and 16-bit pointers that may be needed.

Parameters passed by value between 32-bit and 16-bit code also may require translation between 32-bit and 16-bit formats. The form of the translation is application-dependent.

22.4.5 Writing Interface Procedures

Placing interface code between 32-bit and 16-bit procedures can be the solution to the following interface problems:

- Allowing procedures in 16-bit code segments to call procedures with offsets greater than FFFFH in 32-bit code segments.
- Matching operand-size attributes between companion CALL and RET instructions.
- Translating parameters (data), including managing parameter strings with a variable count or an odd number of 16-bit words.
- The possible invalidation of the upper bits of the ESP register.

The interface procedure is simplified where these rules are followed.

1. The interface procedure must reside in a 32-bit code segment (the D flag for the code-segment descriptor is set).
2. All procedures that may be called by 16-bit procedures must have offsets not greater than FFFFH.
3. All return addresses saved by 16-bit procedures must have offsets not greater than FFFFH.

The interface procedure becomes more complex if any of these rules are violated. For example, if a 16-bit procedure calls a 32-bit procedure with an entry point beyond FFFFH, the interface procedure will need to provide the offset to the entry point. The mapping between 16- and 32-bit addresses is only performed automatically when a call gate is used, because the gate descriptor for a call gate contains a 32-bit address. When a call gate is not used, the interface code must provide the 32-bit address.

The structure of the interface procedure depends on the types of calls it is going to support, as follows:

- **Calls from 16-bit procedures to 32-bit procedures** — Calls to the interface procedure from a 16-bit code segment are made with 16-bit CALL instructions (by default, because the D flag for the calling code-segment descriptor is clear), and 16-bit operand-size prefixes are used with RET instructions to return from the interface procedure to the calling procedure. Calls from the interface procedure to 32-bit procedures are performed with 32-bit CALL instructions (by default, because the D flag for the interface procedure’s code segment is set), and returns from the called procedures to the interface procedure are performed with 32-bit RET instructions (also by default).
- **Calls from 32-bit procedures to 16-bit procedures** — Calls to the interface procedure from a 32-bit code segment are made with 32-bit CALL instructions (by default), and returns to the calling procedure from the interface procedure are made with 32-bit RET instructions (also by default). Calls from the interface procedure to 16-bit procedures require the CALL instructions to have the operand-size prefixes, and returns from the called procedures to the interface procedure are performed with 16-bit RET instructions (by default).

CHAPTER 23
ARCHITECTURE COMPATIBILITY

Intel 64 and IA-32 processors are binary compatible. Compatibility means that, within limited constraints, programs that execute on previous generations of processors will produce identical results when executed on later processors. The compatibility constraints and any implementation differences between the Intel 64 and IA-32 processors are described in this chapter.

Each new processor has enhanced the software visible architecture from that found in earlier Intel 64 and IA-32 processors. Those enhancements have been defined with consideration for compatibility with previous and future processors. This chapter also summarizes the compatibility considerations for those extensions.

23.1 PROCESSOR FAMILIES AND CATEGORIES

IA-32 processors are referred to in several different ways in this chapter, depending on the type of compatibility information being related, as described in the following:

- **IA-32 Processors** — All the Intel processors based on the Intel IA-32 Architecture, which include the 8086/88, Intel 286, Intel386, Intel486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
- **32-bit Processors** — All the IA-32 processors that use a 32-bit architecture, which include the Intel386, Intel486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
- **16-bit Processors** — All the IA-32 processors that use a 16-bit architecture, which include the 8086/88 and Intel 286 processors.
- **P6 Family Processors** — All the IA-32 processors that are based on the P6 microarchitecture, which include the Pentium Pro, Pentium II, and Pentium III processors.
- **Pentium® 4 Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst® microarchitecture.
- **Intel® Pentium® M Processors** — A family of IA-32 processors that are based on the Intel Pentium M processor microarchitecture.
- **Intel® Core™ Duo and Solo Processors** — Families of IA-32 processors that are based on an improved Intel Pentium M processor microarchitecture.
- **Intel® Xeon® Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst microarchitecture. This family includes the Intel Xeon processor and the Intel Xeon processor MP based on the Intel NetBurst microarchitecture. Intel Xeon processors 3000, 3100, 3200, 3300, 3200, 5100, 5200, 5300, 5400, 7200, 7300 series are based on Intel Core microarchitectures and support Intel 64 architecture.
- **Pentium® D Processors** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture.
- **Pentium® Processor Extreme Editions** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture and supports Intel Hyper-Threading Technology.
- **Intel® Core™ 2 Processor family**— A family of Intel 64 processors that are based on the Intel Core microarchitecture. Intel Pentium Dual-Core processors are also based on the Intel Core microarchitecture.
- **Intel Atom® Processors** — A family of IA-32 and Intel 64 processors. 45 nm Intel Atom processors are based on the Intel Atom microarchitecture. 32 nm Intel Atom processors are based on newer microarchitectures including the Silvermont microarchitecture and the Airmont microarchitecture. Each generation of Intel Atom processors can be identified by the CPUID's DisplayFamily_DisplayModel signature; see Table 2-1 "CPUID Signature Values of DisplayFamily_DisplayModel" in Chapter 2, "Model-Specific Registers (MSRs)," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

23.2 RESERVED BITS

Throughout this manual, certain bits are marked as reserved in many register and memory layout descriptions. When bits are marked as undefined or reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown effect. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers or memory locations that contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing them to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

Software written for existing IA-32 processor that handles reserved bits correctly will port to future IA-32 processors without generating protection exceptions.

23.3 ENABLING NEW FUNCTIONS AND MODES

Most of the new control functions defined for the P6 family and Pentium processors are enabled by new mode flags in the control registers (primarily register CR4). This register is undefined for IA-32 processors earlier than the Pentium processor. Attempting to access this register with an Intel486 or earlier IA-32 processor results in an invalid-opcode exception (#UD). Consequently, programs that execute correctly on the Intel486 or earlier IA-32 processor cannot erroneously enable these functions. Attempting to set a reserved bit in register CR4 to a value other than its original value results in a general-protection exception (#GP). So, programs that execute on the P6 family and Pentium processors cannot erroneously enable functions that may be implemented in future IA-32 processors.

The P6 family and Pentium processors do not check for attempts to set reserved bits in model-specific registers; however these bits may be checked on more recent processors. It is the obligation of the software writer to enforce this discipline. These reserved bits may be used in future Intel processors.

23.4 DETECTING THE PRESENCE OF NEW FEATURES THROUGH SOFTWARE

Software can check for the presence of new architectural features and extensions in either of two ways:

1. Test for the presence of the feature or extension. Software can test for the presence of new flags in the EFLAGS register and control registers. If these flags are reserved (meaning not present in the processor executing the test), an exception is generated. Likewise, software can attempt to execute a new instruction, which results in an invalid-opcode exception (#UD) being generated if it is not supported.
2. Execute the CPUID instruction. The CPUID instruction (added to the IA-32 in the Pentium processor) indicates the presence of new features directly.

See Chapter 20, “Processor Identification and Feature Determination,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for detailed information on detecting new processor features and extensions.

23.5 INTEL MMX TECHNOLOGY

The Pentium processor with MMX technology introduced the MMX technology and a set of MMX instructions to the IA-32. The MMX instructions are described in Chapter 9, “Programming with Intel® MMX™ Technology,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D. The MMX technology and MMX instructions are also included in the Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.

23.6 STREAMING SIMD EXTENSIONS (SSE)

The Streaming SIMD Extensions (SSE) were introduced in the Pentium III processor. The SSE extensions consist of a new set of instructions and a new set of registers. The new registers include the eight 128-bit XMM registers and the 32-bit MXCSR control and status register. These instructions and registers are designed to allow SIMD computations to be made on single precision floating-point numbers. Several of these new instructions also operate in the MMX registers. SSE instructions and registers are described in Section 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

23.7 STREAMING SIMD EXTENSIONS 2 (SSE2)

The Streaming SIMD Extensions 2 (SSE2) were introduced in the Pentium 4 and Intel Xeon processors. They consist of a new set of instructions that operate on the XMM and MXCSR registers and perform SIMD operations on double precision floating-point values and on integer values. Several of these new instructions also operate in the MMX registers. SSE2 instructions and registers are described in Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

23.8 STREAMING SIMD EXTENSIONS 3 (SSE3)

The Streaming SIMD Extensions 3 (SSE3) were introduced in Pentium 4 processors supporting Intel Hyper-Threading Technology and Intel Xeon processors. SSE3 extensions include 13 instructions. Ten of these 13 instructions support the single instruction multiple data (SIMD) execution model used with SSE/SSE2 extensions. One SSE3 instruction accelerates x87 style programming for conversion to integer. The remaining two instructions (MONITOR and MWAIT) accelerate synchronization of threads. SSE3 instructions are described in Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

23.9 ADDITIONAL STREAMING SIMD EXTENSIONS

The Supplemental Streaming SIMD Extensions 3 (SSSE3) were introduced in the Intel Core 2 processor and Intel Xeon processor 5100 series. Streaming SIMD Extensions 4 provided 54 new instructions introduced in 45 nm Intel Xeon processors and Intel Core 2 processors. SSSE3, SSE4.1 and SSE4.2 instructions are described in Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

23.10 INTEL HYPER-THREADING TECHNOLOGY

Intel Hyper-Threading Technology provides two logical processors that can execute two separate code streams (called *threads*) concurrently by using shared resources in a single processor core or in a physical package.

This feature was introduced in the Intel Xeon processor MP and later steppings of the Intel Xeon processor, and Pentium 4 processors supporting Intel Hyper-Threading Technology. The feature is also found in the Pentium processor Extreme Edition. See also: Section 9.7, "Intel® Hyper-Threading Technology Architecture."

45 nm and 32 nm Intel Atom processors support Intel Hyper-Threading Technology.

Intel Atom processors based on Silvermont and Airmont microarchitectures do not support Intel Hyper-Threading Technology.

23.11 MULTI-CORE TECHNOLOGY

The Pentium D processor and Pentium processor Extreme Edition provide two processor cores in each physical processor package. See also: Section 9.5, “Intel® Hyper-Threading Technology and Intel® Multi-Core Technology,” and Section 9.8, “Multi-Core Architecture.” Intel Core 2 Duo, Intel Pentium Dual-Core processors, Intel Xeon processors 3000, 3100, 5100, 5200 series provide two processor cores in each physical processor package. Intel Core 2 Extreme, Intel Core 2 Quad processors, Intel Xeon processors 3200, 3300, 5300, 5400, 7300 series provide two processor cores in each physical processor package.

23.12 SPECIFIC FEATURES OF DUAL-CORE PROCESSOR

Dual-core processors may have some processor-specific features. Use CPUID feature flags to detect the availability features. Note the following:

- **CPUID Brand String** — On Pentium processor Extreme Edition, the process will report the correct brand string only after the correct microcode updates are loaded.
- **Enhanced Intel SpeedStep Technology** — This feature is supported in Pentium D processor but not in Pentium processor Extreme Edition.

23.13 NEW INSTRUCTIONS IN THE PENTIUM AND LATER IA-32 PROCESSORS

Table 23-1 identifies the instructions introduced into the IA-32 in the Pentium processor and later IA-32 processors.

23.13.1 Instructions Added Prior to the Pentium Processor

The following instructions were added in the Intel486 processor:

- BSWAP (byte swap) instruction.
- XADD (exchange and add) instruction.
- CMPXCHG (compare and exchange) instruction.
- INVD (invalidate cache) instruction.
- WBINVD (write-back and invalidate cache) instruction.
- INVLPG (invalidate TLB entry) instruction.

Table 23-1. New Instruction in the Pentium Processor and Later IA-32 Processors

Instruction	CPUID Identification Bits	Introduced In
CMOV*cc* (conditional move)	EDX, Bit 15	Pentium Pro processor
FCMOV*cc* (floating-point conditional move)	EDX, Bits 0 and 15	
FCOMI (floating-point compare and set EFLAGS)	EDX, Bits 0 and 15	
RDPMC (read performance monitoring counters)	EAX, Bits 8-11, set to 6H; see Note 1	
UD2 (undefined)	EAX, Bits 8-11, set to 6H	

Table 23-1. New Instruction in the Pentium Processor and Later IA-32 Processors (Contd.)

Instruction	CPUID Identification Bits	Introduced In
CMPXCHG8B (compare and exchange 8 bytes)	EDX, Bit 8	Pentium processor
CPUID (CPU identification)	None; see Note 2	
RDTSC (read time-stamp counter)	EDX, Bit 4	
RDMSR (read model-specific register)	EDX, Bit 5	
WRMSR (write model-specific register)	EDX, Bit 5	
MMX Instructions	EDX, Bit 23	

NOTES:

1. The RDPMC instruction was introduced in the P6 family of processors and added to later model Pentium processors. This instruction is model specific in nature and not architectural.
2. The CPUID instruction is available in all Pentium and P6 family processors and in later models of the Intel486 processors. The ability to set and clear the ID flag (bit 21) in the EFLAGS register indicates the availability of the CPUID instruction.

The following instructions were added in the Intel386 processor:

- LSS, LFS, and LGS (load SS, FS, and GS registers).
- Long-displacement conditional jumps.
- Single-bit instructions.
- Bit scan instructions.
- Double-shift instructions.
- Byte set on condition instruction.
- Move with sign/zero extension.
- Generalized multiply instruction.
- MOV to and from control registers.
- MOV to and from test registers (now obsolete).
- MOV to and from debug registers.
- RSM (resume from SMM). This instruction was introduced in the Intel386 SL and Intel486 SL processors.

The following instructions were added in the Intel 387 math coprocessor:

- FPREM1.
- FUCOM, FUCOMP, and FUCOMPP.

23.14 OBSOLETE INSTRUCTIONS

The MOV to and from test registers instructions were removed from the Pentium processor and future IA-32 processors. Execution of these instructions generates an invalid-opcode exception (#UD).

23.15 UNDEFINED OPCODES

All new instructions defined for Intel 64 and IA-32 processors use binary encodings that were reserved on earlier-generation processors. Generally, attempting to execute a reserved opcode results in an invalid-opcode (#UD) exception being generated. Consequently, programs that execute correctly on earlier-generation processors cannot erroneously execute these instructions and thereby produce unexpected results when executed on later Intel 64 processors.

For compatibility with prior generations, there are a few reserved opcodes which do not result in a #UD but rather result in the same behavior as certain defined instructions. In the interest of standardization, it is recommended that software not use the opcodes given below but instead use those defined in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

The following items enumerate those reserved opcodes (referring in some cases to opcode groups as defined in Appendix A, "Opcode Map," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D).

- **Immediate Group 1** - When not in 64-bit mode, instructions encoded with opcode 82H result in the behavior of the corresponding instructions encoded with opcode 80H. Depending on the Op/Reg field of the ModR/M Byte, these opcodes are the byte forms of ADD, OR, ADC, SBB, AND, SUB, XOR, CMP. (In 64-bit mode, these opcodes cause a #UD.)
- **Shift Group 2 /6** - Instructions encoded with opcodes C0H, C1H, D0H, D1H, D2H, and D3H with value 110B in the Op/Reg field (/6) of the ModR/M Byte result in the behavior of the corresponding instructions with value 100B in the Op/Reg field (/4). These are various forms of the SAL/SHL instruction.
- **Unary Group 3 /1** - Instructions encoded with opcodes F6H and F7H with value 001B in the Op/Reg field (/01) of the ModR/M Byte result in the behavior of the corresponding instructions with value 000B in the Op/Reg field (/0). These are various forms of the TEST instruction.
- **Reserved NOP** - Instructions encoded with the opcode 0F0DH or with the opcodes 0F18H through 0F1FH result in the behavior of the NOP (No Operation) instruction, except for those opcodes defined in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D. The opcodes not so defined are considered "Reserved NOP" and may be used for future instructions which have no defined impact on existing architectural state. These reserved NOP opcodes are decoded with a ModR/M byte and typical instruction prefix options but still result in the behavior of the NOP instruction.
- **x87 Opcodes** - There are several groups of x87 opcodes which provide the same behavior as other x87 instructions. See Section 23.18.9 for the complete list.

There are a few reserved opcodes that provide unique behavior but do not provide capabilities that are not already available in the main instructions defined in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

- **D6H** - When not in 64-bit mode SALC - Set AL to Cary flag. IF (CF=1), AL=FF, ELSE, AL=0 (#UD in 64-bit mode)
- **x87 Opcodes** - There are a few x87 opcodes with subtly different behavior from existing x87 instructions. See Section 23.18.9 for details.

23.16 NEW FLAGS IN THE EFLAGS REGISTER

The section titled "EFLAGS Register" in Chapter 3, "Basic Execution Environment," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, shows the configuration of flags in the EFLAGS register for the P6 family processors. No new flags have been added to this register in the P6 family processors. The flags added to this register in the Pentium and Intel486 processors are described in the following sections.

The following flags were added to the EFLAGS register in the Pentium processor:

- VIF (virtual interrupt flag), bit 19.
- VIP (virtual interrupt pending), bit 20.
- ID (identification flag), bit 21.

The AC flag (bit 18) was added to the EFLAGS register in the Intel486 processor.

23.16.1 Using EFLAGS Flags to Distinguish Between 32-Bit IA-32 Processors

The following bits in the EFLAGS register that can be used to differentiate between the 32-bit IA-32 processors:

- Bit 18 (the AC flag) can be used to distinguish an Intel386 processor from the P6 family, Pentium, and Intel486 processors. Since it is not implemented on the Intel386 processor, it will always be clear.
- Bit 21 (the ID flag) indicates whether an application can execute the CPUID instruction. The ability to set and clear this bit indicates that the processor is a P6 family or Pentium processor. The CPUID instruction can then be used to determine which processor.

- Bits 19 (the VIF flag) and 20 (the VIP flag) will always be zero on processors that do not support virtual mode extensions, which includes all 32-bit processors prior to the Pentium processor.

See Chapter 20, "Processor Identification and Feature Determination," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on identifying processors.

23.17 STACK OPERATIONS AND USER SOFTWARE

This section identifies the differences in stack implementation between the various IA-32 processors.

23.17.1 PUSH SP

The P6 family, Pentium, Intel486, Intel386, and Intel 286 processors push a different value on the stack for a PUSH SP instruction than the 8086 processor. The 32-bit processors push the value of the SP register before it is decremented as part of the push operation; the 8086 processor pushes the value of the SP register after it is decremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

```
PUSH BP
MOV  BP, SP
XCHG BP, [BP]
```

This code functions as the 8086 processor PUSH SP instruction on the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors.

23.17.2 EFLAGS Pushed on the Stack

The setting of the stored values of bits 12 through 15 (which includes the IOPL field and the NT flag) in the EFLAGS register by the PUSHF instruction, by interrupts, and by exceptions is different with the 32-bit IA-32 processors than with the 8086 and Intel 286 processors. The differences are as follows:

- 8086 processor—bits 12 through 15 are always set.
- Intel 286 processor—bits 12 through 15 are always cleared in real-address mode.
- 32-bit processors in real-address mode—bit 15 (reserved) is always cleared, and bits 12 through 14 have the last value loaded into them.

23.18 X87 FPU

This section addresses the issues that must be faced when porting floating-point software designed to run on earlier IA-32 processors and math coprocessors to a Pentium 4, Intel Xeon, P6 family, or Pentium processor with integrated x87 FPU. To software, a Pentium 4, Intel Xeon, or P6 family processor looks very much like a Pentium processor. Floating-point software which runs on a Pentium or Intel486 DX processor, or on an Intel486 SX processor/Intel 487 SX math coprocessor system or an Intel386 processor/Intel 387 math coprocessor system, will run with at most minor modifications on a Pentium 4, Intel Xeon, or P6 family processor. To port code directly from an Intel 286 processor/Intel 287 math coprocessor system or an Intel 8086 processor/8087 math coprocessor system to a Pentium 4, Intel Xeon, P6 family, or Pentium processor, certain additional issues must be addressed.

In the following sections, the term "32-bit x87 FPUs" refers to the P6 family, Pentium, and Intel486 DX processors, and to the Intel 487 SX and Intel 387 math coprocessors; the term "16-bit IA-32 math coprocessors" refers to the Intel 287 and 8087 math coprocessors.

23.18.1 Control Register CR0 Flags

The ET, NE, and MP flags in control register CR0 control the interface between the integer unit of an IA-32 processor and either its internal x87 FPU or an external math coprocessor. The effect of these flags in the various IA-32 processors are described in the following paragraphs.

The ET (extension type) flag (bit 4 of the CR0 register) is used in the Intel386 processor to indicate whether the math coprocessor in the system is an Intel 287 math coprocessor (flag is clear) or an Intel 387 DX math coprocessor (flag is set). This bit is hardwired to 1 in the P6 family, Pentium, and Intel486 processors.

The NE (Numeric Exception) flag (bit 5 of the CR0 register) is used in the P6 family, Pentium, and Intel486 processors to determine whether unmasked floating-point exceptions are reported internally through interrupt vector 16 (flag is set) or externally through an external interrupt (flag is clear). On a hardware reset, the NE flag is initialized to 0, so software using the automatic internal error-reporting mechanism must set this flag to 1. This flag is nonexistent on the Intel386 processor.

As on the Intel 286 and Intel386 processors, the MP (monitor coprocessor) flag (bit 1 of register CR0) determines whether the WAIT/FWAIT instructions or waiting-type floating-point instructions trap when the context of the x87 FPU is different from that of the currently-executing task. If the MP and TS flag are set, then a WAIT/FWAIT instruction and waiting instructions will cause a device-not-available exception (interrupt vector 7). The MP flag is used on the Intel 286 and Intel386 processors to support the use of a WAIT/FWAIT instruction to wait on a device other than a math coprocessor. The device reports its status through the BUSY# pin. Since the P6 family, Pentium, and Intel486 processors do not have such a pin, the MP flag has no relevant use and should be set to 1 for normal operation.

23.18.2 x87 FPU Status Word

This section identifies differences to the x87 FPU status word for the different IA-32 processors and math coprocessors, the reason for the differences, and their impact on software.

23.18.2.1 Condition Code Flags (C0 through C3)

The following information pertains to differences in the use of the condition code flags (C0 through C3) located in bits 8, 9, 10, and 14 of the x87 FPU status word.

After execution of an FINIT instruction or a hardware reset on a 32-bit x87 FPU, the condition code flags are set to 0. The same operations on a 16-bit IA-32 math coprocessor leave these flags intact (they contain their prior value). This difference in operation has no impact on software and provides a consistent state after reset.

Transcendental instruction results in the core range of the P6 family and Pentium processors may differ from the Intel486 DX processor and Intel 487 SX math coprocessor by 2 to 3 units in the last place (ulps)—(see "Transcendental Instruction Accuracy" in Chapter 8, "Programming with the x87 FPU," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). As a result, the value saved in the C1 flag may also differ.

After an incomplete FPREM/FPREM1 instruction, the C0, C1, and C3 flags are set to 0 on the 32-bit x87 FPUs. After the same operation on a 16-bit IA-32 math coprocessor, these flags are left intact.

On the 32-bit x87 FPUs, the C2 flag serves as an incomplete flag for the FTAN instruction. On the 16-bit IA-32 math coprocessors, the C2 flag is undefined for the FPTAN instruction. This difference has no impact on software, because Intel 287 or 8087 programs do not check C2 after an FPTAN instruction. The use of this flag on later processors allows fast checking of operand range.

23.18.2.2 Stack Fault Flag

When unmasked stack overflow or underflow occurs on a 32-bit x87 FPU, the IE flag (bit 0) and the SF flag (bit 6) of the x87 FPU status word are set to indicate a stack fault and condition code flag C1 is set or cleared to indicate overflow or underflow, respectively. When unmasked stack overflow or underflow occurs on a 16-bit IA-32 math coprocessor, only the IE flag is set. Bit 6 is reserved on these processors. The addition of the SF flag on a 32-bit x87 FPU has no impact on software. Existing exception handlers need not change, but may be upgraded to take advantage of the additional information.

23.18.3 x87 FPU Control Word

Only affine closure is supported for infinity control on a 32-bit x87 FPU. The infinity control flag (bit 12 of the x87 FPU control word) remains programmable on these processors, but has no effect. This change was made to conform to the IEEE Standard 754 for Floating-Point Arithmetic. On a 16-bit IA-32 math coprocessor, both affine and projective closures are supported, as determined by the setting of bit 12. After a hardware reset, the default value of bit 12 is projective. Software that requires projective infinity arithmetic may give different results.

23.18.4 x87 FPU Tag Word

When loading the tag word of a 32-bit x87 FPU, using an FLDENV, FRSTOR, or FXRSTOR (Pentium III processor only) instruction, the processor examines the incoming tag and classifies the location only as empty or non-empty. Thus, tag values of 00, 01, and 10 are interpreted by the processor to indicate a non-empty location. The tag value of 11 is interpreted by the processor to indicate an empty location. Subsequent operations on a non-empty register always examine the value in the register, not the value in its tag. The FSTENV, FSAVE, and FXSAVE (Pentium III processor only) instructions examine the non-empty registers and put the correct values in the tags before storing the tag word.

The corresponding tag for a 16-bit IA-32 math coprocessor is checked before each register access to determine the class of operand in the register; the tag is updated after every change to a register so that the tag always reflects the most recent status of the register. Software can load a tag with a value that disagrees with the contents of a register (for example, the register contains a valid value, but the tag says special). Here, the 16-bit IA-32 math coprocessors honor the tag and do not examine the register.

Software written to run on a 16-bit IA-32 math coprocessor may not operate correctly on a 16-bit x87 FPU, if it uses the FLDENV, FRSTOR, or FXRSTOR instructions to change tags to values (other than to empty) that are different from actual register contents.

The encoding in the tag word for the 32-bit x87 FPUs for unsupported data formats (including pseudo-zero and unnormal) is special (10B), to comply with IEEE Standard 754. The encoding in the 16-bit IA-32 math coprocessors for pseudo-zero and unnormal is valid (00B) and the encoding for other unsupported data formats is special (10B). Code that recognizes the pseudo-zero or unnormal format as valid must therefore be changed if it is ported to a 32-bit x87 FPU.

23.18.5 Data Types

This section discusses the differences of data types for the various x87 FPUs and math coprocessors.

23.18.5.1 NaNs

The 32-bit x87 FPUs distinguish between signaling NaNs (SNaNs) and quiet NaNs (QNaNs). These x87 FPUs only generate QNaNs and normally do not generate an exception upon encountering a QNaN. An invalid-operation exception (#I) is generated only upon encountering a SNaN, except for the FCOM, FIST, and FBSTP instructions, which also generates an invalid-operation exceptions for a QNaNs. This behavior matches IEEE Standard 754.

The 16-bit IA-32 math coprocessors only generate one kind of NaN (the equivalent of a QNaN), but the raise an invalid-operation exception upon encountering any kind of NaN.

When porting software written to run on a 16-bit IA-32 math coprocessor to a 32-bit x87 FPU, uninitialized memory locations that contain QNaNs should be changed to SNaNs to cause the x87 FPU or math coprocessor to fault when uninitialized memory locations are referenced.

23.18.5.2 Pseudo-zero, Pseudo-NaN, Pseudo-infinity, and Unnormal Formats

The 32-bit x87 FPUs neither generate nor support the pseudo-zero, pseudo-NaN, pseudo-infinity, and unnormal formats. Whenever they encounter them in an arithmetic operation, they raise an invalid-operation exception. The 16-bit IA-32 math coprocessors define and support special handling for these formats. Support for these formats was dropped to conform with IEEE Standard 754 for Floating-Point Arithmetic.

This change should not impact software ported from 16-bit IA-32 math coprocessors to 32-bit x87 FPUs. The 32-bit x87 FPUs do not generate these formats, and therefore will not encounter them unless software explicitly loads them in the data registers. The only affect may be in how software handles the tags in the tag word (see also: Section 23.18.4, “x87 FPU Tag Word”).

23.18.6 Floating-Point Exceptions

This section identifies the implementation differences in exception handling for floating-point instructions in the various x87 FPUs and math coprocessors.

23.18.6.1 Denormal Operand Exception (#D)

When the denormal operand exception is masked, the 32-bit x87 FPUs automatically normalize denormalized numbers when possible; whereas, the 16-bit IA-32 math coprocessors return a denormal result. A program written to run on a 16-bit IA-32 math coprocessor that uses the denormal exception solely to normalize denormalized operands is redundant when run on the 32-bit x87 FPUs. If such a program is run on 32-bit x87 FPUs, performance can be improved by masking the denormal exception. Floating-point programs run faster when the FPU performs normalization of denormalized operands.

The denormal operand exception is not raised for transcendental instructions and the FXTRACT instruction on the 16-bit IA-32 math coprocessors. This exception is raised for these instructions on the 32-bit x87 FPUs. The exception handlers ported to these latter processors need to be changed only if the handlers gives special treatment to different opcodes.

23.18.6.2 Numeric Overflow Exception (#O)

On the 32-bit x87 FPUs, when the numeric overflow exception is masked and the rounding mode is set to chop (toward 0), the result is the largest positive or smallest negative number. The 16-bit IA-32 math coprocessors do not signal the overflow exception when the masked response is not ∞; that is, they signal overflow only when the rounding control is not set to round to 0. If rounding is set to chop (toward 0), the result is positive or negative ∞. Under the most common rounding modes, this difference has no impact on existing software.

If rounding is toward 0 (chop), a program on a 32-bit x87 FPU produces, under overflow conditions, a result that is different in the least significant bit of the significand, compared to the result on a 16-bit IA-32 math coprocessor. The reason for this difference is IEEE Standard 754 compatibility.

When the overflow exception is not masked, the precision exception is flagged on the 32-bit x87 FPUs. When the result is stored in the stack, the significand is rounded according to the precision control (PC) field of the FPU control word or according to the opcode. On the 16-bit IA-32 math coprocessors, the precision exception is not flagged and the significand is not rounded. The impact on existing software is that if the result is stored on the stack, a program running on a 32-bit x87 FPU produces a different result under overflow conditions than on a 16-bit IA-32 math coprocessor. The difference is apparent only to the exception handler. This difference is for IEEE Standard 754 compatibility.

23.18.6.3 Numeric Underflow Exception (#U)

When the underflow exception is masked on the 32-bit x87 FPUs, the underflow exception is signaled when the result is tiny and inexact (see Section 4.9.1.5, “Numeric Underflow Exception (#U),” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). When the underflow exception is unmasked and the instruction is supposed to store the result on the stack, the significand is rounded to the appropriate precision (according to the PC flag in the FPU control word, for those instructions controlled by PC, otherwise to extended precision), after adjusting the exponent.

23.18.6.4 Exception Precedence

There is no difference in the precedence of the denormal-operand exception on the 32-bit x87 FPUs, whether it be masked or not. When the denormal-operand exception is not masked on the 16-bit IA-32 math coprocessors, it takes precedence over all other exceptions. This difference causes no impact on existing software, but some unneeded normalization of denormalized operands is prevented on the Intel486 processor and Intel 387 math coprocessor.

23.18.6.5 CS and EIP For FPU Exceptions

On the Intel 32-bit x87 FPUs, the values from the CS and EIP registers saved for floating-point exceptions point to any prefixes that come before the floating-point instruction. On the 8087 math coprocessor, the saved CS and IP registers points to the floating-point instruction.

23.18.6.6 FPU Error Signals

The floating-point error signals to the P6 family, Pentium, and Intel486 processors do not pass through an interrupt controller; an INT# signal from an Intel 387, Intel 287 or 8087 math coprocessors does. If an 8086 processor uses another exception for the 8087 interrupt, both exception vectors should call the floating-point-error exception handler. Some instructions in a floating-point-error exception handler may need to be deleted if they use the interrupt controller. The P6 family, Pentium, and Intel486 processors have signals that, with the addition of external logic, support reporting for emulation of the interrupt mechanism used in many personal computers.

On the P6 family, Pentium, and Intel486 processors, an undefined floating-point opcode will cause an invalid-opcode exception (#UD, interrupt vector 6). Undefined floating-point opcodes, like legal floating-point opcodes, cause a device not available exception (#NM, interrupt vector 7) when either the TS or EM flag in control register CR0 is set. The P6 family, Pentium, and Intel486 processors do not check for floating-point error conditions on encountering an undefined floating-point opcode.

23.18.6.7 Assertion of the FERR# Pin

When using the MS-DOS compatibility mode for handing floating-point exceptions, the FERR# pin must be connected to an input to an external interrupt controller. An external interrupt is then generated when the FERR# output drives the input to the interrupt controller and the interrupt controller in turn drives the INTR pin on the processor.

For the P6 family and Intel386 processors, an unmasked floating-point exception always causes the FERR# pin to be asserted upon completion of the instruction that caused the exception. For the Pentium and Intel486 processors, an unmasked floating-point exception may cause the FERR# pin to be asserted either at the end of the instruction causing the exception or immediately before execution of the next floating-point instruction. (Note that the next floating-point instruction would not be executed until the pending unmasked exception has been handled.) See Appendix D, “Guidelines for Writing SIMD Floating-Point Exception Handlers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a complete description of the required mechanism for handling floating-point exceptions using the MS-DOS compatibility mode.

Using FERR# and IGNNE# to handle floating-point exception is deprecated by modern operating systems; this approach also limits newer processors to operate with one logical processor active.

23.18.6.8 Invalid Operation Exception On Denormals

An invalid-operation exception is not generated on the 32-bit x87 FPUs upon encountering a denormal value when executing a FSQRT, FDIV, or FPREM instruction or upon conversion to BCD or to integer. The operation proceeds by first normalizing the value. On the 16-bit IA-32 math coprocessors, upon encountering this situation, the invalid-operation exception is generated. This difference has no impact on existing software. Software running on the 32-bit x87 FPUs continues to execute in cases where the 16-bit IA-32 math coprocessors trap. The reason for this change was to eliminate an exception from being raised.

23.18.6.9 Alignment Check Exceptions (#AC)

If alignment checking is enabled, a misaligned data operand on the P6 family, Pentium, and Intel486 processors causes an alignment check exception (#AC) when a program or procedure is running at privilege-level 3, except for the stack portion of the FSAVE/FNSAVE, FXSAVE, FRSTOR, and FXRSTOR instructions.

23.18.6.10 Segment Not Present Exception During FLDENV

On the Intel486 processor, when a segment not present exception (#NP) occurs in the middle of an FLDENV instruction, it can happen that part of the environment is loaded and part not. In such cases, the FPU control word is left with a value of 007FH. The P6 family and Pentium processors ensure the internal state is correct at all times by attempting to read the first and last bytes of the environment before updating the internal state.

23.18.6.11 Device Not Available Exception (#NM)

The device-not-available exception (#NM, interrupt 7) will occur in the P6 family, Pentium, and Intel486 processors as described in Section 2.5, "Control Registers," Table 2-2, and Chapter 6, "Interrupt 7—Device Not Available Exception (#NM)."

23.18.6.12 Coprocessor Segment Overrun Exception

The coprocessor segment overrun exception (interrupt 9) does not occur in the P6 family, Pentium, and Intel486 processors. In situations where the Intel 387 math coprocessor would cause an interrupt 9, the P6 family, Pentium, and Intel486 processors simply abort the instruction. To avoid undetected segment overruns, it is recommended that the floating-point save area be placed in the same page as the TSS. This placement will prevent the FPU environment from being lost if a page fault occurs during the execution of an FLDENV, FRSTOR, or FXRSTOR instruction while the operating system is performing a task switch.

23.18.6.13 General Protection Exception (#GP)

A general-protection exception (#GP, interrupt 13) occurs if the starting address of a floating-point operand falls outside a segment's size. An exception handler should be included to report these programming errors.

23.18.6.14 Floating-Point Error Exception (#MF)

In real mode and protected mode (not including virtual-8086 mode), interrupt vector 16 must point to the floating-point exception handler. In virtual-8086 mode, the virtual-8086 monitor can be programmed to accommodate a different location of the interrupt vector for floating-point exceptions.

23.18.7 Changes to Floating-Point Instructions

This section identifies the differences in floating-point instructions for the various Intel FPU and math coprocessor architectures, the reason for the differences, and their impact on software.

23.18.7.1 FDIV, FPREM, and FSQRT Instructions

The 32-bit x87 FPUs support operations on denormalized operands and, when detected, an underflow exception can occur, for compatibility with the IEEE Standard 754. The 16-bit IA-32 math coprocessors do not operate on denormalized operands or return underflow results. Instead, they generate an invalid-operation exception when they detect an underflow condition. An existing underflow exception handler will require change only if it gives different treatment to different opcodes. Also, it is possible that fewer invalid-operation exceptions will occur.

23.18.7.2 FSCALE Instruction

With the 32-bit x87 FPUs, the range of the scaling operand is not restricted. If (0 < | ST(1) < 1), the scaling factor is 0; therefore, ST(0) remains unchanged. If the rounded result is not exact or if there was a loss of accuracy (masked underflow), the precision exception is signaled. With the 16-bit IA-32 math coprocessors, the range of the scaling operand is restricted. If (0 < | ST(1) | < 1), the result is undefined and no exception is signaled. The impact of this difference on exiting software is that different results are delivered on the 32-bit and 16-bit FPUs and math coprocessors when (0 < | ST(1) | < 1).

23.18.7.3 FPREM1 Instruction

The 32-bit x87 FPUs compute a partial remainder according to IEEE Standard 754. This instruction does not exist on the 16-bit IA-32 math coprocessors. The availability of the FPREM1 instruction has is no impact on existing software.

23.18.7.4 FPREM Instruction

On the 32-bit x87 FPUs, the condition code flags C0, C3, C1 in the status word correctly reflect the three low-order bits of the quotient following execution of the FPREM instruction. On the 16-bit IA-32 math coprocessors, the quotient bits are incorrect when performing a reduction of $(64^N + M)$ when $(N \geq 1)$ and M is 1 or 2. This difference does not affect existing software; software that works around the bug should not be affected.

23.18.7.5 FUCOM, FUCOMP, and FUCOMPP Instructions

When executing the FUCOM, FUCOMP, and FUCOMPP instructions, the 32-bit x87 FPUs perform unordered compare according to IEEE Standard 754. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these new instructions has no impact on existing software.

23.18.7.6 FPTAN Instruction

On the 32-bit x87 FPUs, the range of the operand for the FPTAN instruction is much less restricted $(| ST(0) | < 2^{63})$ than on earlier math coprocessors. The instruction reduces the operand internally using an internal $\pi/4$ constant that is more accurate. The range of the operand is restricted to $(| ST(0) | < \pi/4)$ on the 16-bit IA-32 math coprocessors; the operand must be reduced to this range using FPREM. This change has no impact on existing software. See also sections 8.3.8 and section 8.3.10 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on the accuracy of the FPTAN instruction.

23.18.7.7 Stack Overflow

On the 32-bit x87 FPUs, if an FPU stack overflow occurs when the invalid-operation exception is masked, the FPU returns the real, integer, or BCD-integer indefinite value to the destination operand, depending on the instruction being executed. On the 16-bit IA-32 math coprocessors, the original operand remains unchanged following a stack overflow, but it is loaded into register ST(1). This difference has no impact on existing software.

23.18.7.8 FSIN, FCOS, and FSINCOS Instructions

On the 32-bit x87 FPUs, these instructions perform three common trigonometric functions. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these instructions has no impact on existing software, but using them provides a performance upgrade. See also sections 8.3.8 and section 8.3.10 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on the accuracy of the FSIN, FCOS, and FSINCOS instructions.

23.18.7.9 FPATAN Instruction

On the 32-bit x87 FPUs, the range of operands for the FPATAN instruction is unrestricted. On the 16-bit IA-32 math coprocessors, the absolute value of the operand in register ST(0) must be smaller than the absolute value of the operand in register ST(1). This difference has impact on existing software.

23.18.7.10 F2XM1 Instruction

The 32-bit x87 FPUs support a wider range of operands $(-1 < ST\ (0) < +\ 1)$ for the F2XM1 instruction. The supported operand range for the 16-bit IA-32 math coprocessors is $(0 \leq ST(0) \leq 0.5)$. This difference has no impact on existing software.

23.18.7.11 FLD Instruction

On the 32-bit x87 FPUs, when using the FLD instruction to load an extended-real value, a denormal-operand exception is not generated because the instruction is not arithmetic. The 16-bit IA-32 math coprocessors do report a denormal-operand exception in this situation. This difference does not affect existing software.

On the 32-bit x87 FPUs, loading a denormal value that is in single- or double-real format causes the value to be converted to extended-real format. Loading a denormal value on the 16-bit IA-32 math coprocessors causes the value to be converted to an unnormal. If the next instruction is FXTRACT or FXAM, the 32-bit x87 FPUs will give a different result than the 16-bit IA-32 math coprocessors. This change was made for IEEE Standard 754 compatibility.

On the 32-bit x87 FPUs, loading an SNaN that is in single- or double-real format causes the FPU to generate an invalid-operation exception. The 16-bit IA-32 math coprocessors do not raise an exception when loading a signaling NaN. The invalid-operation exception handler for 16-bit math coprocessor software needs to be updated to handle this condition when porting software to 32-bit FPUs. This change was made for IEEE Standard 754 compatibility.

23.18.7.12 FXTRACT Instruction

On the 32-bit x87 FPUs, if the operand is 0 for the FXTRACT instruction, the divide-by-zero exception is reported and –∞ is delivered to register ST(1). If the operand is +∞, no exception is reported. If the operand is 0 on the 16-bit IA-32 math coprocessors, 0 is delivered to register ST(1) and no exception is reported. If the operand is +∞, the invalid-operation exception is reported. These differences have no impact on existing software. Software usually bypasses 0 and ∞. This change is due to the IEEE Standard 754 recommendation to fully support the “logb” function.

23.18.7.13 Load Constant Instructions

On 32-bit x87 FPUs, rounding control is in effect for the load constant instructions. Rounding control is not in effect for the 16-bit IA-32 math coprocessors. Results for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions are the same as for the 16-bit IA-32 math coprocessors when rounding control is set to round to nearest or round to +∞. They are the same for the FLDL2T instruction when rounding control is set to round to nearest, round to –∞, or round to zero. Results are different from the 16-bit IA-32 math coprocessors in the least significant bit of the mantissa if rounding control is set to round to –∞ or round to 0 for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions; they are different for the FLDL2T instruction if round to +∞ is specified. These changes were implemented for compatibility with IEEE Standard 754 for Floating-Point Arithmetic recommendations.

23.18.7.14 FXAM Instruction

With the 32-bit x87 FPUs, if the FPU encounters an empty register when executing the FXAM instruction, it not generate combinations of C0 through C3 equal to 1101 or 1111. The 16-bit IA-32 math coprocessors may generate these combinations, among others. This difference has no impact on existing software; it provides a performance upgrade to provide repeatable results.

23.18.7.15 FSAVE and FSTENV Instructions

With the 32-bit x87 FPUs, the address of a memory operand pointer stored by FSAVE or FSTENV is undefined if the previous floating-point instruction did not refer to memory

23.18.8 Transcendental Instructions

The floating-point results of the P6 family and Pentium processors for transcendental instructions in the core range may differ from the Intel486 processors by about 2 or 3 ulps (see “Transcendental Instruction Accuracy” in Chapter 8, “Programming with the x87 FPU,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Condition code flag C1 of the status word may differ as a result. The exact threshold for underflow and overflow will vary by a few ulps. The P6 family and Pentium processors’ results will have a worst case error of less than 1 ulp when rounding to the nearest-even and less than 1.5 ulps when rounding in other modes. The transcendental instructions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

Transcendental instructions may generate different results in the round-up flag (C1) on the 32-bit x87 FPUs. The round-up flag is undefined for these instructions on the 16-bit IA-32 math coprocessors. This difference has no impact on existing software.

23.18.9 Obsolete Instructions and Undefined Opcodes

The 8087 math coprocessor instructions FENI and FDISI, and the Intel 287 math coprocessor instruction FSETPM are treated as integer NOP instructions in the 32-bit x87 FPUs. If these opcodes are detected in the instruction stream, no specific operation is performed and no internal states are affected. FSETPM informed the Intel 287 math coprocessor that the processor was in protected mode. The 32-bit x87 FPUs handle all addressing and exception-pointer information, whether in protected mode or not.

For compatibility with prior generations there are a few reserved x87 opcodes which do not result in an invalid-opcode (#UD) exception, but rather result in the same behavior as existing defined x87 instructions. In the interest of standardization, it is recommended that the opcodes defined in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, be used for these operations for standardization.

- DCD0H through DCD7H - Behaves the same as FCOM, D8D0H through D8D7H.
- DCD8H through DCDFH - Behaves the same as FCOMP, D8D8H through D8DFH.
- DDC8H through DDCFH - Behaves the same as FXCH, D9C8H through D9CFH.
- DED0H through DED7H - Behaves the same as FCOMP, D8D8H through D8DFH.
- DFD0H through DFD7H - Behaves the same as FSTP, DDD8H through DDDFH.
- DFC8H through DFCFH - Behaves the same as FXCH, D9C8H through D9CFH.
- DFD8H through DFDFH - Behaves the same as FSTP, DDD8H through DDDFH.

There are a few reserved x87 opcodes which provide unique behavior but do not provide capabilities which are not already available in the main instructions defined in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

- D9D8H through D9DFH - Behaves the same as FSTP (DDD8H through DDDFH) but won't cause a stack underflow exception.
- DFC0H through DFC7H - Behaves the same as FFREE (DDC0H through DDD7H) with the addition of an x87 stack POP.

23.18.10 WAIT/FWAIT Prefix Differences

On the Intel486 processor, when a WAIT/FWAIT instruction precedes a floating-point instruction (one which itself automatically synchronizes with the previous floating-point instruction), the WAIT/FWAIT instruction is treated as a no-op. Pending floating-point exceptions from a previous floating-point instruction are processed not on the WAIT/FWAIT instruction but on the floating-point instruction following the WAIT/FWAIT instruction. In such a case, the report of a floating-point exception may appear one instruction later on the Intel486 processor than on a P6 family or Pentium FPU, or on Intel 387 math coprocessor.

23.18.11 Operands Split Across Segments and/or Pages

On the P6 family, Pentium, and Intel486 processor FPUs, when the first half of an operand to be written is inside a page or segment and the second half is outside, a memory fault can cause the first half to be stored but not the second half. In this situation, the Intel 387 math coprocessor stores nothing.

23.18.12 FPU Instruction Synchronization

On the 32-bit x87 FPUs, all floating-point instructions are automatically synchronized; that is, the processor automatically waits until the previous floating-point instruction has completed before completing the next floating-point instruction. No explicit WAIT/FWAIT instructions are required to assure this synchronization. For the 8087 math coprocessors, explicit waits are required before each floating-point instruction to ensure synchronization. Although 8087 programs having explicit WAIT instructions execute perfectly on the 32-bit IA-32 processors without reassembly, these WAIT instructions are unnecessary.

23.19 SERIALIZING INSTRUCTIONS

Certain instructions have been defined to serialize instruction execution to ensure that modifications to flags, registers, and memory are completed before the next instruction is executed (or in P6 family processor terminology "committed to machine state"). Because the P6 family processors use branch-prediction and out-of-order execution techniques to improve performance, instruction execution is not generally serialized until the results of an executed instruction are committed to machine state (see Chapter 2, "Intel® 64 and IA-32 Architectures," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

As a result, at places in a program or task where it is critical to have execution completed for all previous instructions before executing the next instruction (for example, at a branch, at the end of a procedure, or in multiprocessor dependent code), it is useful to add a serializing instruction. See Section 9.3, "Serializing Instructions," for more information on serializing instructions.

23.20 FPU AND MATH COPROCESSOR INITIALIZATION

Table 10-1 shows the states of the FPUs in the P6 family, Pentium, Intel486 processors and of the Intel 387 math coprocessor and Intel 287 coprocessor following a power-up, reset, or INIT, or following the execution of an FINIT/FNINIT instruction. The following is some additional compatibility information concerning the initialization of x87 FPUs and math coprocessors.

23.20.1 Intel® 387 and Intel® 287 Math Coprocessor Initialization

Following an Intel386 processor reset, the processor identifies its coprocessor type (Intel® 287 or Intel® 387 DX math coprocessor) by sampling its ERROR# input some time after the falling edge of RESET# signal and before execution of the first floating-point instruction. The Intel 287 coprocessor keeps its ERROR# output in inactive state after hardware reset; the Intel 387 coprocessor keeps its ERROR# output in active state after hardware reset.

Upon hardware reset or execution of the FINIT/FNINIT instruction, the Intel 387 math coprocessor signals an error condition. The P6 family, Pentium, and Intel486 processors, like the Intel 287 coprocessor, do not.

23.20.2 Intel486 SX Processor and Intel 487 SX Math Coprocessor Initialization

When initializing an Intel486 SX processor and an Intel 487 SX math coprocessor, the initialization routine should check the presence of the math coprocessor and should set the FPU related flags (EM, MP, and NE) in control register CR0 accordingly (see Section 2.5, "Control Registers," for a complete description of these flags). Table 23-2 gives the recommended settings for these flags when the math coprocessor is present. The FSTCW instruction will give a value of FFFFH for the Intel486 SX microprocessor and 037FH for the Intel 487 SX math coprocessor.

Table 23-2. Recommended Values of the EM, MP, and NE Flags for Intel486 SX Microprocessor/Intel 487 SX Math Coprocessor System

CR0 Flags	Intel486 SX Processor Only	Intel 487 SX Math Coprocessor Present
EM	1	0
MP	0	1
NE	1	0, for MS-DOS* systems 1, for user-defined exception handler

The EM and MP flags in register CR0 are interpreted as shown in Table 23-3.

Table 23-3. EM and MP Flag Interpretation

EM	MP	Interpretation
0	0	Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions ignore TS.
0	1	Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions test TS.
1	0	Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions ignore TS.
1	1	Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions test TS.

Following is an example code sequence to initialize the system and check for the presence of Intel486 SX processor/Intel 487 SX math coprocessor.

```
fninit
fstcw mem_loc
mov ax, mem_loc
cmp ax, 037fh
jz Intel487_SX_Math_CoProcessor_present     ;ax=037fh
jmp Intel486_SX_microprocessor_present      ;ax=ffffh
```

If the Intel 487 SX math coprocessor is not present, the following code can be run to set the CR0 register for the Intel486 SX processor.

```
mov eax, cr0
and eax, fffffffdh  ;make MP=0
or eax, 0024h       ;make EM=1, NE=1
mov cr0, eax
```

This initialization will cause any floating-point instruction to generate a device not available exception (#NM), interrupt 7. The software emulation will then take control to execute these instructions. This code is not required if an Intel 487 SX math coprocessor is present in the system. In that case, the typical initialization routine for the Intel486 SX microprocessor will be adequate.

Also, when designing an Intel486 SX processor based system with an Intel 487 SX math coprocessor, timing loops should be independent of frequency and clocks per instruction. One way to attain this is to implement these loops in hardware and not in software (for example, BIOS).

23.21 CONTROL REGISTERS

The following sections identify the new control registers and control register flags and fields that were introduced to the 32-bit IA-32 in various processor families. See Figure 2-7 for the location of these flags and fields in the control registers.

The Pentium III processor introduced one new control flag in control register CR4:

- OSXMMEXCPT (bit 10) — The OS will set this bit if it supports unmasked SIMD floating-point exceptions.

The Pentium II processor introduced one new control flag in control register CR4:

- OSFXSR (bit 9) — The OS supports saving and restoring the Pentium III processor state during context switches.

The Pentium Pro processor introduced three new control flags in control register CR4:

- PAE (bit 5) — Physical address extension. Enables paging mechanism to reference extended physical addresses when set; restricts physical addresses to 32 bits when clear (see also: Section 23.22.1.1, "Physical Memory Addressing Extension").
- PGE (bit 7) — Page global enable. Inhibits flushing of frequently-used or shared pages on CR3 writes (see also: Section 23.22.1.2, "Global Pages").
- PCE (bit 8) — Performance-monitoring counter enable. Enables execution of the RDPMC instruction at any protection level.

The content of CR4 is 0H following a hardware reset.

Control register CR4 was introduced in the Pentium processor. This register contains flags that enable certain new extensions provided in the Pentium processor:

- VME — Virtual-8086 mode extensions. Enables support for a virtual interrupt flag in virtual-8086 mode (see Section 21.3, "Interrupt and Exception Handling in Virtual-8086 Mode").
- PVI — Protected-mode virtual interrupts. Enables support for a virtual interrupt flag in protected mode (see Section 21.4, "Protected-Mode Virtual Interrupts").
- TSD — Time-stamp disable. Restricts the execution of the RDTSC instruction to procedures running at privileged level 0.
- DE — Debugging extensions. Causes an undefined opcode (#UD) exception to be generated when debug registers DR4 and DR5 are references for improved performance (see Section 23.23.3, "Debug Registers DR4 and DR5").
- PSE — Page size extensions. Enables 4-MByte pages with 32-bit paging when set (see Section 4.3, "32-Bit Paging").
- MCE — Machine-check enable. Enables the machine-check exception, allowing exception handling for certain hardware error conditions (see Chapter 16, "Machine-Check Architecture").

The Intel486 processor introduced five new flags in control register CR0:

- NE — Numeric error. Enables the normal mechanism for reporting floating-point numeric errors.
- WP — Write protect. Write-protects read-only pages against supervisor-mode accesses.
- AM — Alignment mask. Controls whether alignment checking is performed. Operates in conjunction with the AC (Alignment Check) flag.
- NW — Not write-through. Enables write-throughs and cache invalidation cycles when clear and disables invalidation cycles and write-throughs that hit in the cache when set.
- CD — Cache disable. Enables the internal cache when clear and disables the cache when set.

The Intel486 processor introduced two new flags in control register CR3:

- PCD — Page-level cache disable. The state of this flag is driven on the PCD# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PCD# pin is used to control caching in an external cache on a cycle-by-cycle basis.
- PWT — Page-level write-through. The state of this flag is driven on the PWT# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PWT# pin is used to control write through in an external cache on a cycle-by-cycle basis.

23.22 MEMORY MANAGEMENT FACILITIES

The following sections describe the new memory management facilities available in the various IA-32 processors and some compatibility differences.

23.22.1 New Memory Management Control Flags

The Pentium Pro processor introduced three new memory management features: physical memory addressing extension, the global bit in page-table entries, and general support for larger page sizes. These features are only available when operating in protected mode.

23.22.1.1 Physical Memory Addressing Extension

The new PAE (physical address extension) flag in control register CR4, bit 5, may enable additional address lines on the processor, allowing extended physical addresses. This option can only be used when paging is enabled, using a new page-table mechanism provided to support the larger physical address range (see Section 4.1, "Paging Modes and Control Bits").

23.22.1.2 Global Pages

The new PGE (page global enable) flag in control register CR4, bit 7, provides a mechanism for preventing frequently used pages from being flushed from the translation lookaside buffer (TLB). When this flag is set, frequently used pages (such as pages containing kernel procedures or common data tables) can be marked global by setting the global flag in a page-directory or page-table entry.

On a task switch or a write to control register CR3 (which normally causes the TLBs to be flushed), the entries in the TLB marked global are not flushed. Marking pages global in this manner prevents unnecessary reloading of the TLB due to TLB misses on frequently used pages. See Section 4.10, "Caching Translation Information," for a detailed description of this mechanism.

23.22.1.3 Larger Page Sizes

The P6 family processors support large page sizes. For 32-bit paging, this facility is enabled with the PSE (page size extension) flag in control register CR4, bit 4. When this flag is set, the processor supports either 4-KByte or 4-MByte page sizes. PAE paging and 4-level paging[1] support 2-MByte pages regardless of the value of CR4.PSE (see Section 4.4, "PAE Paging," and Section 4.5, "4-Level Paging and 5-Level Paging"). See Chapter 4, "Paging," for more information about large page sizes.

23.22.2 CD and NW Cache Control Flags

The CD and NW flags in control register CR0 were introduced in the Intel486 processor. In the P6 family and Pentium processors, these flags are used to implement a writeback strategy for the data cache; in the Intel486 processor, they implement a write-through strategy. See Table 12-5 for a comparison of these bits on the P6 family, Pentium, and Intel486 processors. For complete information on caching, see Chapter 12, "Memory Cache Control."

23.22.3 Descriptor Types and Contents

Operating-system code that manages space in descriptor tables often contains an invalid value in the access-rights field of descriptor-table entries to identify unused entries. Access rights values of 80H and 00H remain invalid for the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors. Other values that were invalid on the Intel 286 processor may be valid on the 32-bit processors because uses for these bits have been defined.

[1] Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

23.22.4 Changes in Segment Descriptor Loads

On the Intel386 processor, loading a segment descriptor always causes a locked read and write to set the accessed bit of the descriptor. On the P6 family, Pentium, and Intel486 processors, the locked read and write occur only if the bit is not already set.

23.23 DEBUG FACILITIES

The P6 family and Pentium processors include extensions to the Intel486 processor debugging support for breakpoints. To use the new breakpoint features, it is necessary to set the DE flag in control register CR4.

23.23.1 Differences in Debug Register DR6

It is not possible to write a 1 to reserved bit 12 in debug status register DR6 on the P6 family and Pentium processors; however, it is possible to write a 1 in this bit on the Intel486 processor. See Table 10-1 for the different setting of this register following a power-up or hardware reset.

23.23.2 Differences in Debug Register DR7

The P6 family and Pentium processors determines the type of breakpoint access by the R/W0 through R/W3 fields in debug control register DR7 as follows:

00 Break on instruction execution only.

01 Break on data writes only.

10 Undefined if the DE flag in control register CR4 is cleared; break on I/O reads or writes but not instruction fetches if the DE flag in control register CR4 is set.

11 Break on data reads or writes but not instruction fetches.

On the P6 family and Pentium processors, reserved bits 11, 12, 14, and 15 are hard-wired to 0. On the Intel486 processor, however, bit 12 can be set. See Table 10-1 for the different settings of this register following a power-up or hardware reset.

23.23.3 Debug Registers DR4 and DR5

Although the DR4 and DR5 registers are documented as reserved, previous generations of processors aliased references to these registers to debug registers DR6 and DR7, respectively. When debug extensions are not enabled (the DE flag in control register CR4 is cleared), the P6 family and Pentium processors remain compatible with existing software by allowing these aliased references. When debug extensions are enabled (the DE flag is set), attempts to reference registers DR4 or DR5 will result in an invalid-opcode exception (#UD).

23.24 RECOGNITION OF BREAKPOINTS

For the Pentium processor, it is recommended that debuggers execute the LGDT instruction before returning to the program being debugged to ensure that breakpoints are detected. This operation does not need to be performed on the P6 family, Intel486, or Intel386 processors.

The implementation of test registers on the Intel486 processor used for testing the cache and TLB has been redesigned using MSRs on the P6 family and Pentium processors. (Note that MSRs used for this function are different on the P6 family and Pentium processors.) The MOV to and from test register instructions generate invalid-opcode exceptions (#UD) on the P6 family processors.

23.25 EXCEPTIONS AND/OR EXCEPTION CONDITIONS

This section describes the new exceptions and exception conditions added to the 32-bit IA-32 processors and implementation differences in existing exception handling. See Chapter 6, "Interrupt and Exception Handling," for a detailed description of the IA-32 exceptions.

The Pentium III processor introduced new state with the XMM registers. Computations involving data in these registers can produce exceptions. A new MXCSR control/status register is used to determine which exception or exceptions have occurred. When an exception associated with the XMM registers occurs, an interrupt is generated.

- SIMD floating-point exception (#XM, interrupt 19) — New exceptions associated with the SIMD floating-point registers and resulting computations.

No new exceptions were added with the Pentium Pro and Pentium II processors. The set of available exceptions is the same as for the Pentium processor. However, the following exception condition was added to the IA-32 with the Pentium Pro processor:

- Machine-check exception (#MC, interrupt 18) — New exception conditions. Many exception conditions have been added to the machine-check exception and a new architecture has been added for handling and reporting on hardware errors. See Chapter 16, "Machine-Check Architecture," for a detailed description of the new conditions.

The following exceptions and/or exception conditions were added to the IA-32 with the Pentium processor:

- Machine-check exception (#MC, interrupt 18) — New exception. This exception reports parity and other hardware errors. It is a model-specific exception and may not be implemented or implemented differently in future processors. The MCE flag in control register CR4 enables the machine-check exception. When this bit is clear (which it is at reset), the processor inhibits generation of the machine-check exception.
- General-protection exception (#GP, interrupt 13) — New exception condition added. An attempt to write a 1 to a reserved bit position of a special register causes a general-protection exception to be generated.
- Page-fault exception (#PF, interrupt 14) — New exception condition added. When a 1 is detected in any of the reserved bit positions of a page-table entry, page-directory entry, or page-directory pointer during address translation, a page-fault exception is generated.

The following exception was added to the Intel486 processor:

- Alignment-check exception (#AC, interrupt 17) — New exception. Reports unaligned memory references when alignment checking is being performed.

The following exceptions and/or exception conditions were added to the Intel386 processor:

- Divide-error exception (#DE, interrupt 0)
  - Change in exception handling. Divide-error exceptions on the Intel386 processors always leave the saved CS:IP value pointing to the instruction that failed. On the 8086 processor, the CS:IP value points to the next instruction.
  - Change in exception handling. The Intel386 processors can generate the largest negative number as a quotient for the IDIV instruction (80H and 8000H). The 8086 processor generates a divide-error exception instead.
- Invalid-opcode exception (#UD, interrupt 6) — New exception condition added. Improper use of the LOCK instruction prefix can generate an invalid-opcode exception.
- Page-fault exception (#PF, interrupt 14) — New exception condition added. If paging is enabled in a 16-bit program, a page-fault exception can be generated as follows. Paging can be used in a system with 16-bit tasks if all tasks use the same page directory. Because there is no place in a 16-bit TSS to store the PDBR register, switching to a 16-bit task does not change the value of the PDBR register. Tasks ported from the Intel 286 processor should be given 32-bit TSSs so they can make full use of paging.
- General-protection exception (#GP, interrupt 13) — New exception condition added. The Intel386 processor sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. A general-protection exception is generated if the limit on instruction length is violated. The 8086 processor has no instruction length limit.

23.25.1 Machine-Check Architecture

The Pentium Pro processor introduced a new architecture to the IA-32 for handling and reporting on machine-check exceptions. This machine-check architecture (described in detail in Chapter 16, “Machine-Check Architecture”) greatly expands the ability of the processor to report on internal hardware errors.

23.25.2 Priority of Exceptions

The priority of exceptions are broken down into several major categories:

1. Traps on the previous instruction
2. External interrupts
3. Faults on fetching the next instruction
4. Faults in decoding the next instruction
5. Faults on executing an instruction

There are no changes in the priority of these major categories between the different processors, however, exceptions within these categories are implementation dependent and may change from processor to processor.

23.25.3 Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers

MMX instructions and a subset of SSE, SSE2, SSSE3 instructions operate on MMX registers. The exception conditions of these instructions are described in the following tables.

Table 23-4. Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
#XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1
Applicable Instructions	CVTPD2PI, CVTTPD2PI				

Table 23-5. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1
Applicable Instructions	CVTPI2PS, CVTPS2PI, CVTTPS2PI				

Table 23-6. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF[1]	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	CVTPI2PD				

NOTES:

[1] Applies to "CVTPI2PD xmm, mm" but not "CVTPI2PD xmm, m64".

Table 23-7. Exception Conditions for SIMD/MMX Instructions with Memory Reference

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	PABSB, PABSD, PABSW, PACKSSWB, PACKSSDW, PACKUSWB, PADDB, PADDD, PADDQ, PADDW, PADDSB, PADDSW, PADDUSB, PADDUSW, PALIGNR, PAND, PANDN, PAVGB, PAVGW, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PHADDD, PHADDW, PHADDSW, PHSUBD, PHSUBW, PHSUBSW, PINSRW, PMADDUBSW, PMADDWD, PMAXSW, PMAXUB, PMINSW, PMINUB, PMULHRSW, PMULHUW, PMULHW, PMULLW, PMULUDQ, PSADBW, PSHUFB, PSHUFW, PSIGNB PSIGND PSIGNW, PSLLW, PSLLD, PSLLQ, PSRAD, PSRAW, PSRLW, PSRLD, PSRLQ, PSUBB, PSUBD, PSUBQ, PSUBW, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ, PXOR				

Table 23-8. Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1. If ModR/M.mod ≠ 11b[1]
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
#GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the destination operand is in a non-writable segment.[2] If the DS, ES, FS, or GS register contains a NULL segment selector.[3]
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
#AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	MASKMOVQ, MOVNTQ, "MOVQ (mmreg)"				

NOTES:

[1] Applies to MASKMOVQ only.

[2] Applies to MASKMOVQ and MOVQ (mmreg) only.

[3] Applies to MASKMOVQ only.

Table 23-9. Exception Conditions for Legacy SIMD/MMX Instructions without Memory Reference

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM			X	X	If CR0.TS[bit 3]=1
Applicable Instructions	PEXTRW, PMOVMSKB				

23.26 INTERRUPTS

The following differences in handling interrupts are found among the IA-32 processors.

23.26.1 Interrupt Propagation Delay

External hardware interrupts may be recognized on different instruction boundaries on the P6 family, Pentium, Intel486, and Intel386 processors, due to the superscaler designs of the P6 family and Pentium processors. Therefore, the EIP pushed onto the stack when servicing an interrupt may be different for the P6 family, Pentium, Intel486, and Intel386 processors.

23.26.2 NMI Interrupts

After an NMI interrupt is recognized by the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors, the NMI interrupt is masked until the first IRET instruction is executed, unlike the 8086 processor.

23.26.3 IDT Limit

The LIDT instruction can be used to set a limit on the size of the IDT. A double-fault exception (#DF) is generated if an interrupt or exception attempts to read a vector beyond the limit. Shutdown then occurs on the 32-bit IA-32 processors if the double-fault handler vector is beyond the limit. (The 8086 processor does not have a shutdown mode nor a limit.)

23.27 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

The Advanced Programmable Interrupt Controller (APIC), referred to in this book as the **local APIC**, was introduced into the IA-32 processors with the Pentium processor (beginning with the 735/90 and 815/100 models) and is included in the Pentium 4, Intel Xeon, and P6 family processors. The features and functions of the local APIC are derived from the Intel 82489DX external APIC, which was used with the Intel486 and early Pentium processors. Additional refinements of the local APIC architecture were incorporated in the Pentium 4 and Intel Xeon processors.

23.27.1 Software Visible Differences Between the Local APIC and the 82489DX

The following features in the local APIC features differ from those found in the 82489DX external APIC:

- When the local APIC is disabled by clearing the APIC software enable/disable flag in the spurious-interrupt vector MSR, the state of its internal registers are unaffected, except that the mask bits in the LVT are all set to block local interrupts to the processor. Also, the local APIC ceases accepting IPIs except for INIT, SMI, NMI, and start-up IPIs. In the 82489DX, when the local unit is disabled, all the internal registers including the IRR, ISR, and TMR are cleared and the mask bits in the LVT are set. In this state, the 82489DX local unit will accept only the reset deassert message.
- In the local APIC, NMI and INIT (except for INIT deassert) are always treated as edge triggered interrupts, even if programmed otherwise. In the 82489DX, these interrupts are always level triggered.
- In the local APIC, IPIs generated through the ICR are always treated as edge triggered (except INIT Deassert). In the 82489DX, the ICR can be used to generate either edge or level triggered IPIs.
- In the local APIC, the logical destination register supports 8 bits; in the 82489DX, it supports 32 bits.
- In the local APIC, the APIC ID register is 4 bits wide; in the 82489DX, it is 8 bits wide.
- The remote read delivery mode provided in the 82489DX and local APIC for Pentium processors is not supported in the local APIC in the Pentium 4, Intel Xeon, and P6 family processors.
- For the 82489DX, in the lowest priority delivery mode, all the target local APICs specified by the destination field participate in the lowest priority arbitration. For the local APIC, only those local APICs which have free interrupt slots will participate in the lowest priority arbitration.

23.27.2 New Features Incorporated in the Local APIC for the P6 Family and Pentium Processors

The local APIC in the Pentium and P6 family processors have the following new features not found in the 82489DX external APIC.

- Cluster addressing is supported in logical destination mode.
- Focus processor checking can be enabled/disabled.
- Interrupt input signal polarity can be programmed for the LINT0 and LINT1 pins.
- An SMI IPI is supported through the ICR and I/O redirection table.
- An error status register is incorporated into the LVT to log and report APIC errors.

In the P6 family processors, the local APIC incorporates an additional LVT register to handle performance monitoring counter interrupts.

23.27.3 New Features Incorporated in the Local APIC of the Pentium 4 and Intel Xeon Processors

The local APIC in the Pentium 4 and Intel Xeon processors has the following new features not found in the P6 family and Pentium processors and in the 82489DX.

- The local APIC ID is extended to 8 bits.
- An thermal sensor register is incorporated into the LVT to handle thermal sensor interrupts.
- The the ability to deliver lowest-priority interrupts to a focus processor is no longer supported.
- The flat cluster logical destination mode is not supported.

23.28 TASK SWITCHING AND TSS

This section identifies the implementation differences of task switching, additions to the TSS and the handling of TSSs and TSS segment selectors.

23.28.1 P6 Family and Pentium Processor TSS

When the virtual mode extensions are enabled (by setting the VME flag in control register CR4), the TSS in the P6 family and Pentium processors contain an interrupt redirection bit map, which is used in virtual-8086 mode to redirect interrupts back to an 8086 program.

23.28.2 TSS Selector Writes

During task state saves, the Intel486 processor writes 2-byte segment selectors into a 32-bit TSS, leaving the upper 16 bits undefined. For performance reasons, the P6 family and Pentium processors write 4-byte segment selectors into the TSS, with the upper 2 bytes being 0. For compatibility reasons, code should not depend on the value of the upper 16 bits of the selector in the TSS.

23.28.3 Order of Reads/Writes to the TSS

The order of reads and writes into the TSS is processor dependent. The P6 family and Pentium processors may generate different page-fault addresses in control register CR2 in the same TSS area than the Intel486 and Intel386 processors, if a TSS crosses a page boundary (which is not recommended).

23.28.4 Using A 16-Bit TSS with 32-Bit Constructs

Task switches using 16-bit TSSs should be used only for pure 16-bit code. Any new code written using 32-bit constructs (operands, addressing, or the upper word of the EFLAGS register) should use only 32-bit TSSs. This is due to the fact that the 32-bit processors do not save the upper 16 bits of EFLAGS to a 16-bit TSS. A task switch back to a 16-bit task that was executing in virtual mode will never re-enable the virtual mode, as this flag was not saved in the upper half of the EFLAGS value in the TSS. Therefore, it is strongly recommended that any code using 32-bit constructs use a 32-bit TSS to ensure correct behavior in a multitasking environment.

23.28.5 Differences in I/O Map Base Addresses

The Intel486 processor considers the TSS segment to be a 16-bit segment and wraps around the 64K boundary. Any I/O accesses check for permission to access this I/O address at the I/O base address plus the I/O offset. If the I/O map base address exceeds the specified limit of 0DFFFH, an I/O access will wrap around and obtain the permission for the I/O address at an incorrect location within the TSS. A TSS limit violation does not occur in this situation on the Intel486 processor. However, the P6 family and Pentium processors consider the TSS to be a 32-bit segment and a limit violation occurs when the I/O base address plus the I/O offset is greater than the TSS limit. By following the recommended specification for the I/O base address to be less than 0DFFFH, the Intel486 processor will not wrap around and access incorrect locations within the TSS for I/O port validation and the P6 family and Pentium processors will not experience general-protection exceptions (#GP). Figure 23-1 demonstrates the different areas accessed by the Intel486 and the P6 family and Pentium processors.

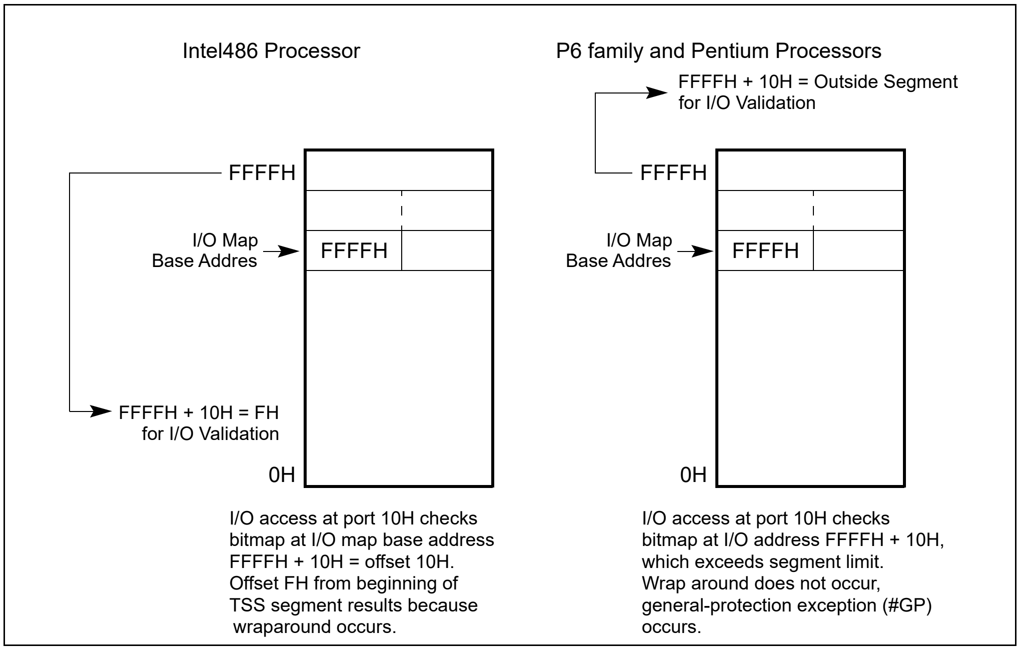

Intel486 Processor
P6 family and Pentium Processors
FFFFH + 10H = Outside Segment for I/O Validation
FFFFH
I/O Map Base Addres
FFFFH
FFFFH + 10H = FH for I/O Validation
0H
FFFFH
I/O Map Base Addres
FFFFH
0H
I/O access at port 10H checks bitmap at I/O map base address FFFFH + 10H = offset 10H. Offset FH from beginning of TSS segment results because wraparound occurs.
I/O access at port 10H checks bitmap at I/O address FFFFH + 10H, which exceeds segment limit. Wrap around does not occur, general-protection exception (#GP) occurs.

Figure 23-1. I/O Map Base Address Differences

23.29 CACHE MANAGEMENT

The P6 family processors include two levels of internal caches: L1 (level 1) and L2 (level 2). The L1 cache is divided into an instruction cache and a data cache; the L2 cache is a general-purpose cache. See Section 12.1, “Internal Caches, TLBs, and Buffers,” for a description of these caches. (Note that although the Pentium II processor L2 cache is physically located on a separate chip in the cassette, it is considered an internal cache.)

The Pentium processor includes separate level 1 instruction and data caches. The data cache supports a writeback (or alternatively write-through, on a line by line basis) policy for memory updates.

The Intel486 processor includes a single level 1 cache for both instructions and data.

The meaning of the CD and NW flags in control register CR0 have been redefined for the P6 family and Pentium processors. For these processors, the recommended value (00B) enables writeback for the data cache of the Pentium processor and for the L1 data cache and L2 cache of the P6 family processors. In the Intel486 processor, setting these flags to (00B) enables write-through for the cache.

External system hardware can force the Pentium processor to disable caching or to use the write-through cache policy should that be required. In the P6 family processors, the MTRRs can be used to override the CD and NW flags (see Table 12-6).

The P6 family and Pentium processors support page-level cache management in the same manner as the Intel486 processor by using the PCD and PWT flags in control register CR3, the page-directory entries, and the page-table entries. The Intel486 processor, however, is not affected by the state of the PWT flag since the internal cache of the Intel486 processor is a write-through cache.

23.29.1 Self-Modifying Code with Cache Enabled

On the Intel486 processor, a write to an instruction in the cache will modify it in both the cache and memory. If the instruction was prefetched before the write, however, the old version of the instruction could be the one executed. To prevent this problem, it is necessary to flush the instruction prefetch unit of the Intel486 processor by coding a jump instruction immediately after any write that modifies an instruction. The P6 family and Pentium processors, however, check whether a write may modify an instruction that has been prefetched for execution. This check is based on the linear address of the instruction. If the linear address of an instruction is found to be present in the prefetch queue, the P6 family and Pentium processors flush the prefetch queue, eliminating the need to code a jump instruction after any writes that modify an instruction.

Because the linear address of the write is checked against the linear address of the instructions that have been prefetched, special care must be taken for self-modifying code to work correctly when the physical addresses of the instruction and the written data are the same, but the linear addresses differ. In such cases, it is necessary to execute a serializing operation to flush the prefetch queue after the write and before executing the modified instruction. See Section 9.3, "Serializing Instructions," for more information on serializing instructions.

NOTE

> The check on linear addresses described above is not in practice a concern for compatibility. Applications that include self-modifying code use the same linear address for modifying and fetching the instruction. System software, such as a debugger, that might possibly modify an instruction using a different linear address than that used to fetch the instruction must execute a serializing operation, such as IRET, before the modified instruction is executed.

23.29.2 Disabling the L3 Cache

A unified third-level (L3) cache in processors based on Intel NetBurst microarchitecture (see Section 12.1, "Internal Caches, TLBs, and Buffers") provides the third-level cache disable flag, bit 6 of the IA32_MISC_ENABLE MSR. The third-level cache disable flag allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches (see Section 12.5.4, "Disabling and Enabling the L3 Cache"). The third-level cache disable flag applies only to processors based on Intel NetBurst microarchitecture. Processors with L3 and based on other microarchitectures do not support the third-level cache disable flag.

23.30 PAGING

This section identifies enhancements made to the paging mechanism and implementation differences in the paging mechanism for various IA-32 processors.

23.30.1 Large Pages

The Pentium processor extended the memory management/paging facilities of the IA-32 to allow large (4 MBytes) pages sizes (see Section 4.3, "32-Bit Paging"). The first P6 family processor (the Pentium Pro processor) added a 2 MByte page size to the IA-32 in conjunction with the physical address extension (PAE) feature (see Section 4.4, "PAE Paging").

The availability of large pages with 32-bit paging on any IA-32 processor can be determined via feature bit 3 (PSE) of register EDX after the CPUID instruction has been execution with an argument of 1. (Large pages are always available with PAE paging and 4-level paging.) Intel processors that do not support the CPUID instruction support only 32-bit paging and do not support page size enhancements. (See "CPUID—CPU Identification" in Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for more information on the CPUID instruction.)

23.30.2 PCD and PWT Flags

The PCD and PWT flags were introduced to the IA-32 in the Intel486 processor to control the caching of pages:

- PCD (page-level cache disable) flag—Controls caching on a page-by-page basis.
- PWT (page-level write-through) flag—Controls the write-through/writeback caching policy on a page-by-page basis. Since the internal cache of the Intel486 processor is a write-through cache, it is not affected by the state of the PWT flag.

23.30.3 Enabling and Disabling Paging

Paging is enabled and disabled by loading a value into control register CR0 that modifies the PG flag. For backward and forward compatibility with all IA-32 processors, Intel recommends that the following operations be performed when enabling or disabling paging:

1. Execute a MOV CR0, REG instruction to either set (enable paging) or clear (disable paging) the PG flag.
2. Execute a near JMP instruction.

The sequence bounded by the MOV and JMP instructions should be identity mapped (that is, the instructions should reside on a page whose linear and physical addresses are identical).

For the P6 family processors, the MOV CR0, REG instruction is serializing, so the jump operation is not required. However, for backwards compatibility, the JMP instruction should still be included.

23.31 STACK OPERATIONS AND SUPERVISOR SOFTWARE

This section identifies the differences in the stack mechanism for the various IA-32 processors.

23.31.1 Selector Pushes and Pops

When pushing a segment selector onto the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors decrement the ESP register by the operand size and then write 2 bytes. If the operand size is 32-bits, the upper two bytes of the write are not modified. The Pentium processor decrements the ESP register by the operand size and determines the size of the write by the operand size. If the operand size is 32-bits, the upper two bytes are written as 0s.

When popping a segment selector from the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors read 2 bytes and increment the ESP register by the operand size of the instruction. The Pentium processor determines the size of the read from the operand size and increments the ESP register by the operand size.

It is possible to align a 32-bit selector push or pop such that the operation generates an exception on a Pentium processor and not on an Pentium 4, Intel Xeon, P6 family, or Intel486 processor. This could occur if the third and/or fourth byte of the operation lies beyond the limit of the segment or if the third and/or fourth byte of the operation is locate on a non-present or inaccessible page.

For a POP-to-memory instruction that meets the following conditions:

- The stack segment size is 16-bit.
- Any 32-bit addressing form with the SIB byte specifying ESP as the base register.
- The initial stack pointer is FFFCH (32-bit operand) or FFFEH (16-bit operand) and will wrap around to 0H as a result of the POP operation.

The result of the memory write is implementation-specific. For example, in P6 family processors, the result of the memory write is SS:0H plus any scaled index and displacement. In Pentium processors, the result of the memory write may be either a stack fault (real mode or protected mode with stack segment size of 64 KByte), or write to SS:10000H plus any scaled index and displacement (protected mode and stack segment size exceeds 64 KByte).

23.31.2 Error Code Pushes

The Intel486 processor implements the error code pushed on the stack as a 16-bit value. When pushed onto a 32-bit stack, the Intel486 processor only pushes 2 bytes and updates ESP by 4. The P6 family and Pentium processors' error code is a full 32 bits with the upper 16 bits set to zero. The P6 family and Pentium processors, therefore, push 4 bytes and update ESP by 4. Any code that relies on the state of the upper 16 bits may produce inconsistent results.

23.31.3 Fault Handling Effects on the Stack

During the handling of certain instructions, such as CALL and PUSHA, faults may occur in different sequences for the different processors. For example, during far calls, the Intel486 processor pushes the old CS and EIP before a possible branch fault is resolved. A branch fault is a fault from a branch instruction occurring from a segment limit or access rights violation. If a branch fault is taken, the Intel486 and P6 family processors will have corrupted memory below the stack pointer. However, the ESP register is backed up to make the instruction restartable. The P6 family processors issue the branch before the pushes. Therefore, if a branch fault does occur, these processors do not corrupt memory below the stack pointer. This implementation difference, however, does not constitute a compatibility problem, as only values at or above the stack pointer are considered to be valid. Other operations that encounter faults may also corrupt memory below the stack pointer and this behavior may vary on different implementations.

23.31.4 Interlevel RET/IRET From a 16-Bit Interrupt or Call Gate

If a call or interrupt is made from a 32-bit stack environment through a 16-bit gate, only 16 bits of the old ESP can be pushed onto the stack. On the subsequent RET/IRET, the 16-bit ESP is popped but the full 32-bit ESP is updated since control is being resumed in a 32-bit stack environment. The Intel486 processor writes the SS selector into the upper 16 bits of ESP. The P6 family and Pentium processors write zeros into the upper 16 bits.

23.32 MIXING 16- AND 32-BIT SEGMENTS

The features of the 16-bit Intel 286 processor are an object-code compatible subset of those of the 32-bit IA-32 processors. The D (default operation size) flag in segment descriptors indicates whether the processor treats a code or data segment as a 16-bit or 32-bit segment; the B (default stack size) flag in segment descriptors indicates whether the processor treats a stack segment as a 16-bit or 32-bit segment.

The segment descriptors used by the Intel 286 processor are supported by the 32-bit IA-32 processors if the Intel-reserved word (highest word) of the descriptor is clear. On the 32-bit IA-32 processors, this word includes the upper bits of the base address and the segment limit.

The segment descriptors for data segments, code segments, local descriptor tables (there are no descriptors for global descriptor tables), and task gates are the same for the 16- and 32-bit processors. Other 16-bit descriptors (TSS segment, call gate, interrupt gate, and trap gate) are supported by the 32-bit processors.

The 32-bit processors also have descriptors for TSS segments, call gates, interrupt gates, and trap gates that support the 32-bit architecture. Both kinds of descriptors can be used in the same system.

For those segment descriptors common to both 16- and 32-bit processors, clear bits in the reserved word cause the 32-bit processors to interpret these descriptors exactly as an Intel 286 processor does, that is:

- Base Address — The upper 8 bits of the 32-bit base address are clear, which limits base addresses to 24 bits.
- Limit — The upper 4 bits of the limit field are clear, restricting the value of the limit field to 64 KBytes.
- Granularity bit — The G (granularity) flag is clear, indicating the value of the 16-bit limit is interpreted in units of 1 byte.
- Big bit — In a data-segment descriptor, the B flag is clear in the segment descriptor used by the 32-bit processors, indicating the segment is no larger than 64 KBytes.
- Default bit — In a code-segment descriptor, the D flag is clear, indicating 16-bit addressing and operands are the default. In a stack-segment descriptor, the D flag is clear, indicating use of the SP register (instead of the ESP register) and a 64-KByte maximum segment limit.

For information on mixing 16- and 32-bit code in applications, see Chapter 22, “Mixing 16-Bit and 32-Bit Code.”

23.33 SEGMENT AND ADDRESS WRAPAROUND

This section discusses differences in segment and address wraparound between the P6 family, Pentium, Intel486, Intel386, Intel 286, and 8086 processors.

23.33.1 Segment Wraparound

On the 8086 processor, an attempt to access a memory operand that crosses offset 65,535 or 0FFFFH or offset 0 (for example, moving a word to offset 65,535 or pushing a word when the stack pointer is set to 1) causes the offset to wrap around modulo 65,536 or 010000H. With the Intel 286 processor, any base and offset combination that addresses beyond 16 MBytes wraps around to the 1 MByte of the address space. The P6 family, Pentium, Intel486, and Intel386 processors in real-address mode generate an exception in these cases:

- A general-protection exception (#GP) if the segment is a data segment (that is, if the CS, DS, ES, FS, or GS register is being used to address the segment).
- A stack-fault exception (#SS) if the segment is a stack segment (that is, if the SS register is being used).

An exception to this behavior occurs when a stack access is data aligned, and the stack pointer is pointing to the last aligned piece of data that size at the top of the stack (ESP is FFFFFFFCH). When this data is popped, no segment limit violation occurs and the stack pointer will wrap around to 0.

The address space of the P6 family, Pentium, and Intel486 processors may wraparound at 1 MByte in real-address mode. An external A20M# pin forces wraparound if enabled. On Intel 8086 processors, it is possible to specify addresses greater than 1 MByte. For example, with a selector value FFFFH and an offset of FFFFH, the effective address would be 10FFEFH (1 MByte plus 65519 bytes). The 8086 processor, which can form addresses up to 20 bits long, truncates the uppermost bit, which “wraps” this address to FFEFH. However, the P6 family, Pentium, and Intel486 processors do not truncate this bit if A20M# is not enabled.

If a stack operation wraps around the address limit, shutdown occurs. (The 8086 processor does not have a shutdown mode or a limit.)

The behavior when executing near the limit of a 4-GByte selector (limit = FFFFFFFFH) is different between the Pentium Pro and the Pentium 4 family of processors. On the Pentium Pro, instructions which cross the limit -- for example, a two byte instruction such as INC EAX that is encoded as FFH C0H starting exactly at the limit faults for a segment violation (a one byte instruction at FFFFFFFFH does not cause an exception). Using the Pentium 4 microprocessor family, neither of these situations causes a fault.

Segment wraparound and the functionality of A20M# is used primarily by older operating systems and not used by modern operating systems. On newer Intel 64 processors, A20M# may be absent.

23.34 STORE BUFFERS AND MEMORY ORDERING

The Pentium 4, Intel Xeon, and P6 family processors provide a store buffer for temporary storage of writes (stores) to memory (see Section 12.10, “Store Buffer”). Writes stored in the store buffer(s) are always written to memory in program order, with the exception of “fast string” store operations (see Section 9.2.4, “Fast-String Operation and Out-of-Order Stores”).

The Pentium processor has two store buffers, one corresponding to each of the pipelines. Writes in these buffers are always written to memory in the order they were generated by the processor core.

It should be noted that only memory writes are buffered and I/O writes are not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors do not synchronize the completion of memory writes on the bus and instruction execution after a write. An I/O, locked, or serializing instruction needs to be executed to synchronize writes with the next instruction (see Section 9.3, “Serializing Instructions”).

The Pentium 4, Intel Xeon, and P6 family processors use processor ordering to maintain consistency in the order that data is read (loaded) and written (stored) in a program and the order the processor actually carries out the reads and writes. With this type of ordering, reads can be carried out speculatively and in any order, reads can pass buffered writes, and writes to memory are always carried out in program order. (See Section 9.2, “Memory Ordering,” for more information about processor ordering.) The Pentium III processor introduced a new instruction to serialize writes and make them globally visible. Memory ordering issues can arise between a producer and a consumer of data. The SFENCE instruction provides a performance-efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

No re-ordering of reads occurs on the Pentium processor, except under the condition noted in Section 9.2.1, “Memory Ordering in the Intel® Pentium® and Intel486™ Processors,” and in the following paragraph describing the Intel486 processor.

Specifically, the store buffers are flushed before the IN instruction is executed. No reads (as a result of cache miss) are reordered around previously generated writes sitting in the store buffers. The implication of this is that the store buffers will be flushed or emptied before a subsequent bus cycle is run on the external bus.

On both the Intel486 and Pentium processors, under certain conditions, a memory read will go onto the external bus before the pending memory writes in the buffer even though the writes occurred earlier in the program execution. A memory read will only be reordered in front of all writes pending in the buffers if all writes pending in the buffers are cache hits and the read is a cache miss. Under these conditions, the Intel486 and Pentium processors will not read from an external memory location that needs to be updated by one of the pending writes.

During a locked bus cycle, the Intel486 processor will always access external memory, it will never look for the location in the on-chip cache. All data pending in the Intel486 processor's store buffers will be written to memory before a locked cycle is allowed to proceed to the external bus. Thus, the locked bus cycle can be used for eliminating the possibility of reordering read cycles on the Intel486 processor. The Pentium processor does check its cache on a read-modify-write access and, if the cache line has been modified, writes the contents back to memory before locking the bus. The P6 family processors write to their cache on a read-modify-write operation (if the access does not split across a cache line) and does not write back to system memory. If the access does split across a cache line, it locks the bus and accesses system memory.

I/O reads are never reordered in front of buffered memory writes on an IA-32 processor. This ensures an update of all memory locations before reading the status from an I/O device.

23.35 BUS LOCKING

The Intel 286 processor performs the bus locking differently than the Intel P6 family, Pentium, Intel486, and Intel386 processors. Programs that use forms of memory locking specific to the Intel 286 processor may not run properly when run on later processors.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and Intel 286 configurations lock the entire physical memory space. Programmers should not depend on this.

On the Intel 286 processor, the LOCK prefix is sensitive to IOPL. If the CPL is greater than the IOPL, a general-protection exception (#GP) is generated. On the Intel386 DX, Intel486, and Pentium, and P6 family processors, no check against IOPL is performed.

The Pentium processor automatically asserts the LOCK# signal when acknowledging external interrupts. After signaling an interrupt request, an external interrupt controller may use the data bus to send the interrupt vector to the processor. After receiving the interrupt request signal, the processor asserts LOCK# to ensure that no other data appears on the data bus until the interrupt vector is received. This bus locking does not occur on the P6 family processors.

23.36 BUS HOLD

Unlike the 8086 and Intel 286 processors, but like the Intel386 and Intel486 processors, the P6 family and Pentium processors respond to requests for control of the bus from other potential bus masters, such as DMA controllers, between transfers of parts of an unaligned operand, such as two words which form a doubleword. Unlike the Intel386 processor, the P6 family, Pentium, and Intel486 processors respond to bus hold during reset initialization.

23.37 MODEL-SPECIFIC EXTENSIONS TO THE IA-32

Certain extensions to the IA-32 are specific to a processor or family of IA-32 processors and may not be implemented or implemented in the same way in future processors. The following sections describe these model-specific extensions. The CPUID instruction indicates the availability of some of the model-specific features.

23.37.1 Model-Specific Registers

The Pentium processor introduced a set of model-specific registers (MSRs) for use in controlling hardware functions and performance monitoring. To access these MSRs, two new instructions were added to the IA-32 architecture: read MSR (RDMSR) and write MSR (WRMSR). The MSRs in the Pentium processor are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

The P6 family processors greatly increased the number of MSRs available to software. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for a complete list of the available MSRs. The new registers control the debug extensions, the performance counters, the machine-check exception capability, the machine-check architecture, and the MTRRs. These registers are accessible using the RDMSR and WRMSR instructions. Specific information on some of these new MSRs is provided in the following sections. As with the Pentium processor MSR, the P6 family processor MSRs are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

23.37.2 RDMSR and WRMSR Instructions

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions recognize a much larger number of model-specific registers in the P6 family processors. (See "RDMSR—Read from Model Specific Register" and "WRMSR—Write to Model Specific Register" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D, for more information.)

23.37.3 Memory Type Range Registers

Memory type range registers (MTRRs) are a new feature introduced into the IA-32 in the Pentium Pro processor. MTRRs allow the processor to optimize memory operations for different types of memory, such as RAM, ROM, frame buffer memory, and memory-mapped I/O.

MTRRs are MSRs that contain an internal map of how physical address ranges are mapped to various types of memory. The processor uses this internal memory map to determine the cacheability of various physical memory locations and the optimal method of accessing memory locations. For example, if a memory location is specified in an MTRR as write-through memory, the processor handles accesses to this location as follows. It reads data from that location in lines and caches the read data or maps all writes to that location to the bus and updates the cache to maintain cache coherency. In mapping the physical address space with MTRRs, the processor recognizes five types of memory: uncacheable (UC), uncacheable, speculatable, write-combining (WC), write-through (WT), write-protected (WP), and writeback (WB).

Earlier IA-32 processors (such as the Intel486 and Pentium processors) used the KEN# (cache enable) pin and external logic to maintain an external memory map and signal cacheable accesses to the processor. The MTRR mechanism simplifies hardware designs by eliminating the KEN# pin and the external logic required to drive it.

See Chapter 10, "Processor Management and Initialization," and Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for more information on the MTRRs.

23.37.4 Machine-Check Exception and Architecture

The Pentium processor introduced a new exception called the machine-check exception (#MC, interrupt 18). This exception is used to detect hardware-related errors, such as a parity error on a read cycle.

The P6 family processors extend the types of errors that can be detected and that generate a machine-check exception. It also provides a new machine-check architecture for recording information about a machine-check error and provides extended recovery capability.

The machine-check architecture provides several banks of reporting registers for recording machine-check errors. Each bank of registers is associated with a specific hardware unit in the processor. The primary focus of the machine checks is on bus and interconnect operations; however, checks are also made of translation lookaside buffer (TLB) and cache operations.

The machine-check architecture can correct some errors automatically and allow for reliable restart of instruction execution. It also collects sufficient information for software to use in correcting other machine errors not corrected by hardware.

See Chapter 16, “Machine-Check Architecture,” for more information on the machine-check exception and the machine-check architecture.

23.37.5 Performance-Monitoring Counters

The P6 family and Pentium processors provide two performance-monitoring counters for use in monitoring internal hardware operations. The number of performance monitoring counters and associated programming interfaces may be implementation specific for Pentium 4 processors, Pentium M processors. Later processors may have implemented these as part of an architectural performance monitoring feature. The architectural and non-architectural performance monitoring interfaces for different processor families are described in Chapter 20, “Performance Monitoring.” https://perfmon-events.intel.com/ lists all the events that can be counted for architectural performance monitoring events and non-architectural events. The counters are set up, started, and stopped using two MSRs and the RDMSR and WRMSR instructions. For the P6 family processors, the current count for a particular counter can be read using the new RDPMC instruction.

The performance-monitoring counters are useful for debugging programs, optimizing code, diagnosing system failures, or refining hardware designs. See Chapter 20, “Performance Monitoring,” for more information on these counters.

23.38 TWO WAYS TO RUN INTEL 286 PROCESSOR TASKS

When porting 16-bit programs to run on 32-bit IA-32 processors, there are two approaches to consider:

- Porting an entire 16-bit software system to a 32-bit processor, complete with the old operating system, loader, and system builder. Here, all tasks will have 16-bit TSSs. The 32-bit processor is being used as if it were a faster version of the 16-bit processor.
- Porting selected 16-bit applications to run in a 32-bit processor environment with a 32-bit operating system, loader, and system builder. Here, the TSSs used to represent 286 tasks should be changed to 32-bit TSSs. It is possible to mix 16 and 32-bit TSSs, but the benefits are small and the problems are great. All tasks in a 32-bit software system should have 32-bit TSSs. It is not necessary to change the 16-bit object modules themselves; TSSs are usually constructed by the operating system, by the loader, or by the system builder. See Chapter 22, “Mixing 16-Bit and 32-Bit Code,” for more detailed information about mixing 16-bit and 32-bit code.

Because the 32-bit processors use the contents of the reserved word of 16-bit segment descriptors, 16-bit programs that place values in this word may not run correctly on the 32-bit processors.

23.39 INITIAL STATE OF PENTIUM, PENTIUM PRO AND PENTIUM 4 PROCESSORS

Table 23-10 shows the state of the flags and other registers following power-up for the Pentium, Pentium Pro and Pentium 4 processors. The state of control register CR0 is 60000010H (see Figure 10-1 “Contents of CR0 Register after Reset” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). This places the processor in real-address mode with paging disabled.

Table 23-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H

Table 23-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor

CS	Selector = F000H
	Base = FFFF0000H
	Limit = FFFFH
	AR = Present, R/W, Accessed
		Selector = F000H
		Base = FFFF0000H
		Limit = FFFFH
		AR = Present, R/W, Accessed
			Selector = F000H
			Base = FFFF0000H
			Limit = FFFFH
			AR = Present, R/W, Accessed

SS, DS, ES, FS, GS	Selector = 0000H
	Base = 00000000H
	Limit = FFFFH
	AR = Present, R/W, Accessed
		Selector = 0000H
		Base = 00000000H
		Limit = FFFFH
		AR = Present, R/W, Accessed
			Selector = 0000H
			Base = 00000000H
			Limit = FFFFH
			AR = Present, R/W, Accessed

EDX	00000FxxH	000n06xxH[3]	000005xxH
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7[5]	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged
x87 FPU Control Word[5]	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH
x87 FPU Status Word[5]	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Tag Word[5]	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors[5]	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers[5]	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H
MM0 through MM7[5]	Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium II and Pentium III Processors Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium with MMX Technology Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	Pwr up or Reset: 0H INIT: Unchanged	If CPUID.01H:SSE is 1 — Pwr up or Reset: 0H INIT: Unchanged	NA
MXCSR	Pwr up or Reset: 1F80H INIT: Unchanged	Pentium III processor only- Pwr up or Reset: 1F80H INIT: Unchanged	NA

GDTR, IDTR	Base = 00000000H
	Limit = FFFFH
	AR = Present, R/W
		Base = 00000000H
		Limit = FFFFH
		AR = Present, R/W
			Base = 00000000H
			Limit = FFFFH
			AR = Present, R/W

LDTR, Task Register	Selector = 0000H
	Base = 00000000H
	Limit = FFFFH
	AR = Present, R/W
		Selector = 0000H
		Base = 00000000H
		Limit = FFFFH
		AR = Present, R/W
			Selector = 0000H
			Base = 00000000H
			Limit = FFFFH
			AR = Present, R/W

DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H

Table 23-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
DR7	00000400H	00000400H	00000400H
Time-Stamp Counter	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
Perf. Counters and Event Select	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
All Other MSRs	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged
Data and Code Cache, TLBs	Invalid[6]	Invalid[6]	Invalid[6]
Fixed MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Variable MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Machine-Check Architecture	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Not Implemented
APIC	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged
R8-R15[7]	0000000000000000H	0000000000000000H	N.A.
XMM8-XMM15[7]	Pwr up or Reset: 0H INIT: Unchanged	Pwr up or Reset: 0H INIT: Unchanged	N.A.

NOTES:

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. Where "n" is the Extended Model Value for the respective processor.
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
7. If the processor supports IA-32e mode.