



## Programming Guide

# Intel® Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide (Nehalem Core PMU)

## Table of Contents

1. About this document.....	8
2. Nehalem-based PMU Architecture.....	9
3. The Nehalem Core PMU.....	11
3.1. Global Control and Status Registers .....	13
3.1.1. IA32_PERF_CAPABILITIES.....	13
3.1.2. IA32_DEBUGCTL.....	15
3.1.3. IA32_PERF_GLOBAL_CTRL.....	17
3.1.4. IA32_PERF_GLOBAL_STATUS.....	19
3.1.5. IA32_PERF_GLOBAL_OVF_CTRL.....	20
3.1.6. IA32_FIXED_CTR_CTRL .....	22
3.2. Programmable Counter Control Registers .....	24
3.2.1. PerfEvtSelX .....	24
3.3. Counter Registers.....	27
3.3.1. PERF_FIXED_CTRX and IA32_PMCX Registers.....	27
3.4. Off-core Response Event Programming.....	28
3.5. The PEBS Facility .....	32
3.5.1. IA32_PEBS_ENABLE .....	32
3.5.2. PEBS Record Format.....	34
3.5.3. Programming the PEBS Facility.....	35
3.6. Counter Prioritization.....	38
3.7. The Load Latency Facility.....	39
3.8. The Last Branch Record Facility.....	43
3.8.1. LBR Filtering.....	44
3.8.2. Storing Branch Trace Messages in the Branch Trace Store.....	45
3.8.3. Storing Branch Trace Messages in the Last Branch Record Stack.....	47
3.9. The RDPMC Instruction .....	50
4. Detecting the Nehalem Processor .....	52
4.1. CPUID identification.....	52
4.2. Architectural PerfMon .....	52
4.3. Other Considerations.....	54

## List of Figures

<b>Figure 1: IA32_PERF_CAPABILITIES MSR Definition</b>	<b>13</b>
<b>Figure 2: IA32_DEBUGCTL MSR Definition</b>	<b>15</b>
<b>Figure 3: IA32_PERF_GLOBAL_CTRL MSR Definition</b>	<b>18</b>
<b>Figure 4: IA32_PERF_GLOBAL_STATUS MSR Definition</b>	<b>19</b>
<b>Figure 5: IA32_PERF_GLOBAL_OVF_CTRL MSR Definition</b>	<b>21</b>
<b>Figure 6: IA32_FIXED_CTR_CTRL MSR Definition</b>	<b>23</b>
<b>Figure 7: PerfEvtSelX MSR Definition</b>	<b>25</b>
<b>Figure 8: PERF_FIXED_CTRX and IA32_PMCX MSR Definition</b>	<b>27</b>
<b>Figure 9: OFFCORE_RSP_X MSR Definition</b>	<b>29</b>
<b>Figure 10: IA32_PEBS_ENABLE MSR Definition</b>	<b>33</b>
<b>Figure 11: PEBS Programming Overview</b>	<b>36</b>
<b>Figure 12: IA32_DS_AREA MSR Definition</b>	<b>38</b>
<b>Figure 13: LFSR for Tagging Memory Load Instructions</b>	<b>40</b>
<b>Figure 14: PEBS_LD_LAT_THRESHOLD MSR Definition</b>	<b>42</b>
<b>Figure 15: LBR_SELECT MSR Programming</b>	<b>44</b>
<b>Figure 16: Format of Branch Trace Message in Branch Trace Store</b>	<b>45</b>
<b>Figure 17: Branch Trace Store Programming Overview</b>	<b>47</b>
<b>Figure 18: MSR_LASTBRANCH_x_FROM_IP MSR Definition</b>	<b>48</b>
<b>Figure 19: MSR_LASTBRANCH_x_TO_IP MSR Definition</b>	<b>49</b>
<b>Figure 20: MSR_LASTBRANCH_TOS MSR Definition</b>	<b>50</b>

## List of Tables

Table 1: List of Terms.....	5
Table 2: Performance Monitoring Architecture Comparison.....	11
Table 3: IA32_PERF_CAPABILITIES Programming .....	13
Table 4: IA32_DEBUGCTL Programming.....	15
Table 5: IA32_PERF_GLOBAL_CTRL Programming .....	18
Table 6: IA32_PERF_GLOBAL_STATUS Programming .....	19
Table 7: IA32_PERF_GLOBAL_OVF_CTRL Programming .....	21
Table 8: IA32_FIXED_CTR_CTRL Programming.....	23
Table 9: Summary of CTL bit Programming for Fixed Counters .....	24
Table 10: PerfEvtSelX Programming .....	25
Table 11: Offcore Response Events Encoding .....	28
Table 12: OFFCORE_RSP_X Event Programming .....	29
Table 13: Offcore Response Programming Example .....	31
Table 14: IA32_PEBS_ENABLE Programming .....	33
Table 15: PEBS Record Format.....	34
Table 16: Data Souce Encodings in Load Latency Records.....	41
Table 17: PEBS_LD_LAT_THRESHOLD Programming.....	42
Table 18: Programming the BTS Facility .....	43
Table 19: LBR_SELECT Programming.....	45
Table 20: MSR_LASTBRANCH_x_FROM_IP Programming .....	48
Table 21: MSR_LASTBRANCH_x_TO_IP Programming .....	49
Table 22: MSR_LASTBRANCH_TOS Programming .....	50
Table 23: Legal Index (ECX) Values for Executing the RDPMC Instruction .....	51
Table 38: Nehalem CPUID Processor Signatures .....	52
Table 39: CPUID.A Information.....	53
Table 40: NHM Core PMU Register List.....	57

## Preface

This document contains advance information. While every effort has been made to ensure the accuracy of the information contained herein, some errors may occur. Please contact [thomas.m.johnson@intel.com](mailto:thomas.m.johnson@intel.com) if you have questions or comments.

This document describes the programming interface to the performance monitoring hardware on the Nehalem processor core. This document does not exhaustively describe all of the performance monitoring events which may be counted in the Nehalem. A detailed description of these events may be released separately.

## Terms

Table 1: List of Terms

Term	Definition
BTM	Branch Trace Message. A message sent on the system bus which external hardware can capture and thereby develop a reconstruction of program control flow.
BTS	Branch Trace Store. A memory buffer containing a collection of branch trace messages.
Clear	In reference to register programming, this means a bit is programmed to binary zero (0).
CPL	Current Privilege Level. The current privilege level at which the processor is executing (Ring 0, 1, 2, or 3).
DCU	Data cache. The cache closest to the processor core. This cache provides data to the core with the minimum latency.
EBS	Event Based Sampling. A technique in which counters are pre-loaded with a large negative count, and they are configured to interrupt the processor on overflow. When the counter overflows the interrupt service routine capture profiling data.
GO	Globally Observable. The point in time at which data in the machine is architecturally observable.
GP	General Protection (fault).
ISR	Interrupt Service Routine.
LBR	Last Branch Record. A facility which provides branch trace information either through special bus cycles on the system bus, or through records written to a user defined

	memory buffer (the BTS).
LLC	Last-level cache. The lowest level of cache, after which memory requests must be satisfied by system memory.
MLC	Mid-level cache. This is the intermediate level cache which lies between the DCU and LLC.
MSR	Model Specific Register. PMU counter and counter control registers are implemented as MSR registers. They are accessed via the <code>rdmsr</code> and <code>wrmsr</code> instruction. Certain counter registers can be accessed via the <code>rdpmc</code> instruction.
NHM	Nehalem. Specifically the Nehalem processor core.
PEBS	Precise Event Based Sampling. A special counting mode in which counters can be configured to overflow, interrupt the processor, and capture machine state at that point.
PerfMon	Short for Performance Monitoring
PMI	Performance Monitoring Interrupt. This interrupt is generated when a counter overflows and has been programmed to generate an interrupt, or when the PEBS interrupt threshold has been reached. The interrupt vector for this interrupt is controlled through the Local Vector Table in the Local APIC.
PMU	Performance Monitoring Unit
RFO	Read for ownership. When a cache line is written that misses in the cache, it must first be read into the cache so that the line exists in cache and can then be modified.
RO	A bit is read-only.
RW	A bit is readable and writeable.
Set	In reference to register programming, this means a bit is programmed to binary one (1).
SMM	System Management Mode.
SMT	Simultaneous Multi-threading.
Supervisor (SUP), or privilege level 0	Supervisor state is the most privileged state of execution. Typically operating system code executes in privilege level 0.
TBS	Time Based Sampling. A technique in which a time base is used to determine when to capture profiling data. This

	time base can be a timer interrupt or the occurrence of a certain number of other events, such clock ticks or instructions retired.
Thread	A hardware thread of execution. In other words, Hyper-Threading Technology.
Uop	Micro-operation. Macro instruction are broken down into uops within the machine, and these uops are executed by the execution units.
User (USR), or privilege level 1, 2, or 3	Intel processors operate in privilege levels zero through three, where lower numbered privilege levels operate in a more privileged state. User (or privilege levels 1, 2, or 3) refers to less privileged states of execution. User code typically executes at privilege level 3.
W0	A bit is write-only.
W01	A bit is write-only, and should be written to a '1' (set).

## 1. About this document

This is a programmer's reference manual for the Nehalem core performance monitoring units (PMU). This is targeted for current tool owners requiring documentation updates for Nehalem based platforms. It is not intended for first time tool developers or as a user analysis guide. Additional documents will be available at a later date targeted at providing that information.

## 2. Nehalem-based PMU Architecture

Intel processor cores for many years included a Performance Monitoring Unit (PMU). This unit provided the ability to count the occurrence of micro-architectural events which expose some of the inner workings of the processor core as it executes code.

One usage of this capability is to create a list of events from which certain performance metrics can be calculated. Software configures the PMU to count events over an interval of time and report the resulting event counts. Using this methodology, performance analysts can characterize overall system performance.

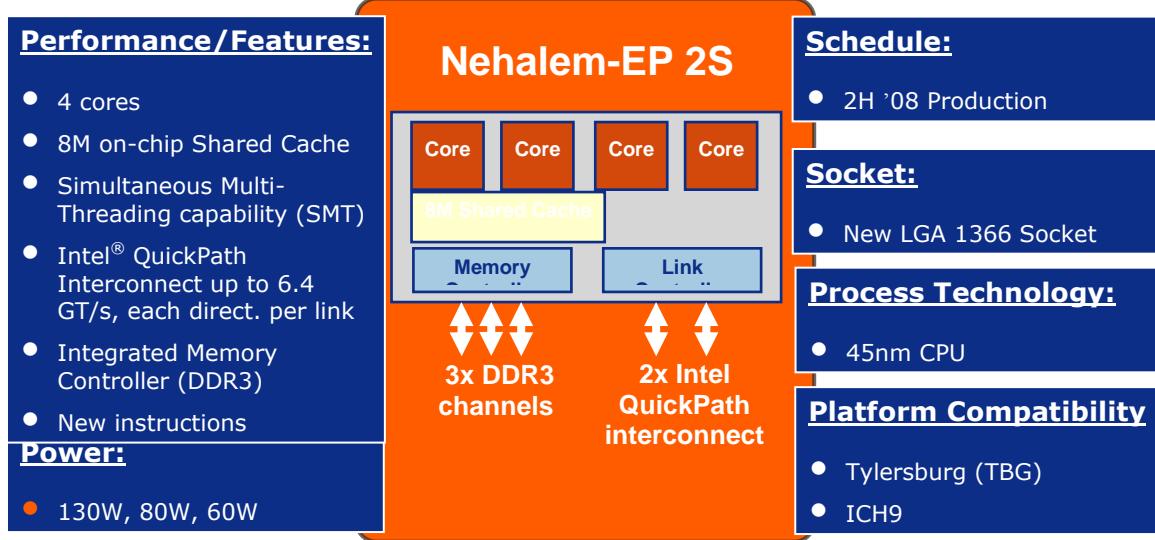
The PMU also provides facilities to generate a hardware interrupt through the Local APIC integrated within the processor core or logical thread. In this case software can pre-load event counter registers with a "sample after value," in which case a hardware interrupt is generated after the occurrence of N number of events. In the interrupt handler software collects additional architectural state which provides analysts with information regarding the performance of specific areas of application code. This methodology is sometimes referred to as profiling the execution of an application.

Products based on the Nehalem processor core include the capability to collect event data under both of these scenarios. In addition, these products include various platform features (uncore) integrated on the same die as the processor core. The uncore is essentially everything on the processor chip that is not part of the core. This includes point-to-point interconnect logic, memory controllers, and last-level caches, among other things. The uncores also provide an additional PMU facility that has the ability to interrupt the processor core in order that profiling information may be collected. This document describes the Nehalem processor core PMU.

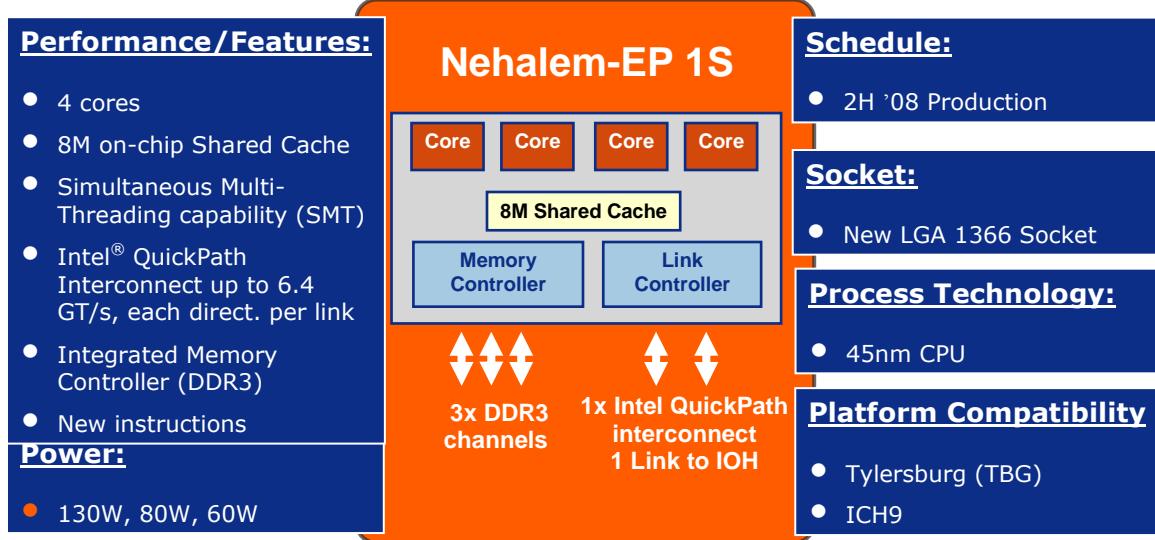
Platform Name	Processor	CPU Core	Processor Code Name	Chipset
Tylersburg-EP 2S	Nehalem-EP 2S	Nehalem	Gainstwon	Tylersburg
Tylersburg-EP 1S	Nehalem-EP 1S	Nehalem		Tylersburg
Boxboro-EX	Nehalem-EX	Nehalem	Beckton	Boxboro

The PMU differences between Nehalem-EP 2S and Nehalem-EP 1S will solely be reflected in the event list. The programming and architecture list will not differ. Supplemental documentation will be published for Nehalem-EX. Nehalem-EX uncore has a significantly different architecture and therefore programming.

## Nehalem-EP 2S CPU Summary



## Nehalem-EP 1S CPU Summary



### 3. The Nehalem Core PMU

The table below summarizes the differences between the Nehalem core (NHM) PMU features and that of previous products in the Intel® Core™ and Pentium 4® processor families. Nehalem adheres to Architectural Performance Monitoring Version 3.

The table includes architectural and non-architectural features. Architectural features are listed in the top of the table with intercepts highlighted.

Table 2: Performance Monitoring Architecture Comparison

Feature	Description	P4	Yonah (V1*)	Merom (V2*)	Penryn (V2*)	Nehalem (V3*)
Number of General Counters	Number of general counters per logical CPU.	18	2	2	2	4
Number of Fixed Counters	Examples include instructions retired, un-halted core clock ticks, un-halted reference clock ticks, etc.	0	0	3	3	3
Number of Selection and Control Registers	Used to select events and configure the general counters.	45 ESCR, 18 CCCR	2	2	2	4
Architectural Events	Architectural events retain a consistent definition and encoding across product implementations.	0	7	7	7	7
Global Counter Controls	Single MSR to globally enable and disable all counters.	No	Yes	Yes	Yes	Yes
Freeze on Overflow Interrupt	Ability to freeze (disable) counters on occurrence of overflow interrupt. Freeze occurs before PMI delivery.	No	No	Yes	Yes	Yes
VT VMCS	Add global enable MSR to guest and host VMCS.	No	No	No	Yes	Yes
Freeze Counters on SMI	Add support for save/disable/restore on SMI.	No	No	No	Yes	Yes
AnyThread	Enable bit for counting when an event is active across any hardware thread for fixed and general counters.	Yes Thread specific programming	No	No	No	Yes Thread independent programming

PEBS	Precise Event Based Sampling. Includes mechanisms to store branch and machine state information on counter overflow.	Limited to certain counters.	No	Limited to certain counters.	Limited to certain counters.	Available on any counter.
Load Latency	A PEBS record extension which allows sampling load instruction latency, load target address, and instruction pointer.	No	No	No	No	Yes
Last Branch Record (LBR)	Circular ring of MSRs which record branch target and source instruction pointers.	8 LIP	4 EIP	4 EIP	4 LIP	16 EIP
Freeze LBR Stack on Overflow Interrupt	Ability to freeze (disable) capture of LBR information on occurrence of overflow interrupt.	No	No	Yes	Yes	Yes
SMI Counter	Free running read-only SMI counters (used like time stamp counter).	No	No	No	No	Yes
Branch Trace Store	Branch trace store feature uses LBR MSRs.	Yes	No	Yes	Yes	Yes

\*Denotes the version of Architectural Performance Monitoring supported by this product.

The Nehalem core supports event counting with seven event counters. Three of these counters are fixed function counters; the events counted by each of these counters are fixed in hardware. Software can determine whether counting is enabled during user or supervisor code execution, or both. The four remaining counters are programmable, and can be configured to count a variety of events. There are some restrictions on individual counters. Fixed counters are controlled by bit fields in a global control register. Programmable counters are controlled by a separate control register, one for each counter.

PMU resources are available and must be programmed for each hardware thread (logical processor), if threading is enabled. Otherwise they are programmed for each core. PMU resources available in each thread do not accumulate to the core when hardware threading is disabled. Thus the PMU programming model remains consistent in any case. To successfully program all PMU resources, software must affinize itself to each processor the operating system exposes. Counter registers are 48 bits in extent.

Writing a binary one to any reserved bit in any counter or counter control register is undefined and may cause a general protection fault.

The following sections describe the programming the counter and counter control registers used to program the NHM PMU.

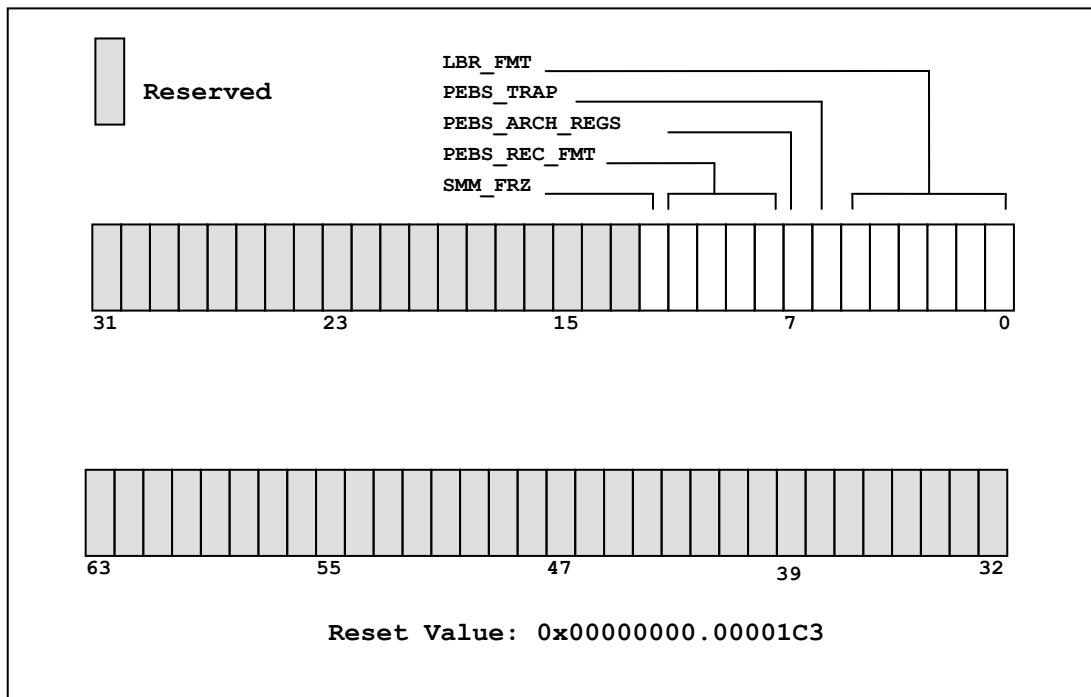
### 3.1. Global Control and Status Registers

There are a set of global control and status registers which control the fixed and programmable counters, and provide status indications of the NHM PMU in general. The following sections describe these global registers.

### 3.1.1. IA32\_PERF\_CAPABILITIES

This register provides availability status information regarding the capabilities supported by this implementation of the performance monitoring hardware. This is a read-only register.

Figure 1: IA32\_PERF\_CAPABILITIES MSR Definition



**Table 3: IA32\_PERF\_CAPABILITIES Programming**

Bit	Bit Position	Access	Description
-----	--------------	--------	-------------

LBR_FMT	5:0	RO	Branch to/from information contains: 00000b: 32-bit offset in current code segment. 00001b: 64-bit LIP 00010b: 64-bit EIP
PEBS_TRAP	6	RO	If clear indicates the PEBS assist is trap-like. The assist occurs at the end-of-macro (EOM) instruction boundary after machine state is committed. The PEBS record will reflect the state of the machine after this instruction executed. If set, indicates the PEBS assist is fault-like. The assist will occur before the current instruction retires, and the PEBS record will reflect the pre-retirement state of the machine. See section 3.5.2 for more information on the PEBS record.
PEBS_ARCH_REG	7	RO	If set, indicates that the PEBS record contains the architectural registers in addition to the return instruction pointer and flags. If clear, only return the instruction pointer and flags are recorded.
PEBS_REC_FMT	11:8	RO	0000b: Only the return instruction pointer, flags, and general purpose registers are recorded in the PEBS record. 0001b: In addition to the information contained in option 0 above, the PEBS record contains the global overflow status and load latency information. Other values are reserved.
SMM_FRZ	12	RO	If set, indicates this implementation has the ability to freeze the PMU on entry to SMM mode. This means all counters and profiling hardware will be disabled while in SMM mode. Upon exit from SMM mode, normal operation will resume. No freeze capability is supported otherwise.

### 3.1.2. IA32\_DEBUGCTL

This register controls tracing, single stepping, and last branch record (LBR) collection, and certain freeze functions related to the PMU. The programming for this register is summarized in the figure and table below.

Figure 2: IA32\_DEBUGCTL MSR Definition

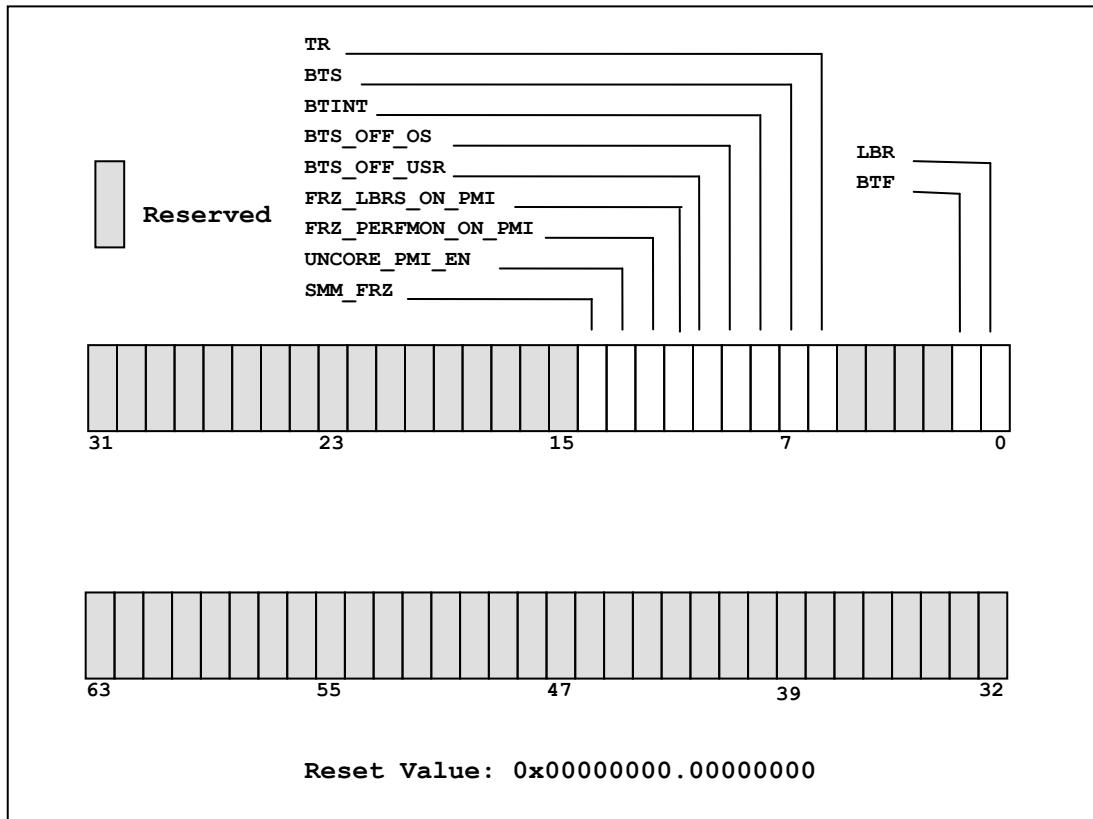


Table 4: IA32\_DEBUGCTL Programming

Bit	Bit Position	Access	Description
LBR	0	RW	When set, the processor records a running trace of the most recent branches, interrupts, and exceptions taken by the processor in the LBR stack. Because last branch recording and branch trace messages share common hardware, this bit should not be set at the same time as the TR bit.
BTF	1	RW	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 will cause a debug exception to be posted when a taken branch occurs while TF is set.
TR	6	RW		When set, branch trace messages are enabled. Upon detection of a branch, interrupt, or exception the processor sends a branch record to the system bus as a branch trace message (BTM).
BTS	7	RW		When set, the flag enables branch trace store (BTS) facilities to log BTMs to a memory buffer (the Branch Trace Store).
BTINT	8	RW		When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, the BTS buffer behaves as a circular buffer and older records are overwritten.
BTS_OFF_OS	9	RW		When set, BTMs will not be written to the system bus or the BTS when the CPL is equal to zero.
BTS_OFF_USR	10	RW		When set, BTMs will not be written to the system bus or the BTS when the CPL is not equal to zero.
FRZ_LBRS_ON_PMI	11	RW		When set, the LBR stack is frozen on a hardware PMI request. Software must explicitly re-enable the LBR capture mechanism.  Note: the freeze is currently triggered by counter overflow, and may result in freezing the LBR stack sooner than expected.
FRZ_PERFMON_ON_PMI	12	RW		When set, a PMI request clears each of the enable bits in the

			IA32_PERF_GLOBAL_CTRL MSR, disabling all event counting. Software must explicitly re-enable the counters by setting the enable bits in the IA32_PERF_GLOBAL_CTRL register before exiting the ISR.
UNCORE_PMI_EN	13	RW	When set, this logical processor is enabled to receive an uncore counter overflow interrupt.
SMM_FRZ	14	RW	When set, event counters are disabled while the processor is in system management mode.

### 3.1.3. IA32\_PERF\_GLOBAL\_CTRL

This register globally controls the fixed and programmable counters. If a control bit in this register is clear, all other control register programming for the corresponding counter will be ignored and the counter will not count.

Counters that are disabled by this register cannot count, overflow, or subsequently generate overflow interrupts. It is possible that a disabled counter may generate a PEBS assist. This can occur as follows. While the counter is enabled it can overflow (to zero) and arm the PEBS hardware. The next event (the counter transitions from zero to one) will cause the PEBS assist to occur. At this point the PEBS assist will occur even if an intervening write to this register disables the counter.

Note that the state of the IA32\_PERF\_GLOBAL\_CTRL register is preserved across entry and exit to probe mode (halting with an ITP). Writes to this register during probe mode will be lost upon exit from probe mode.

Figure 3: IA32\_PERF\_GLOBAL\_CTRL MSR Definition

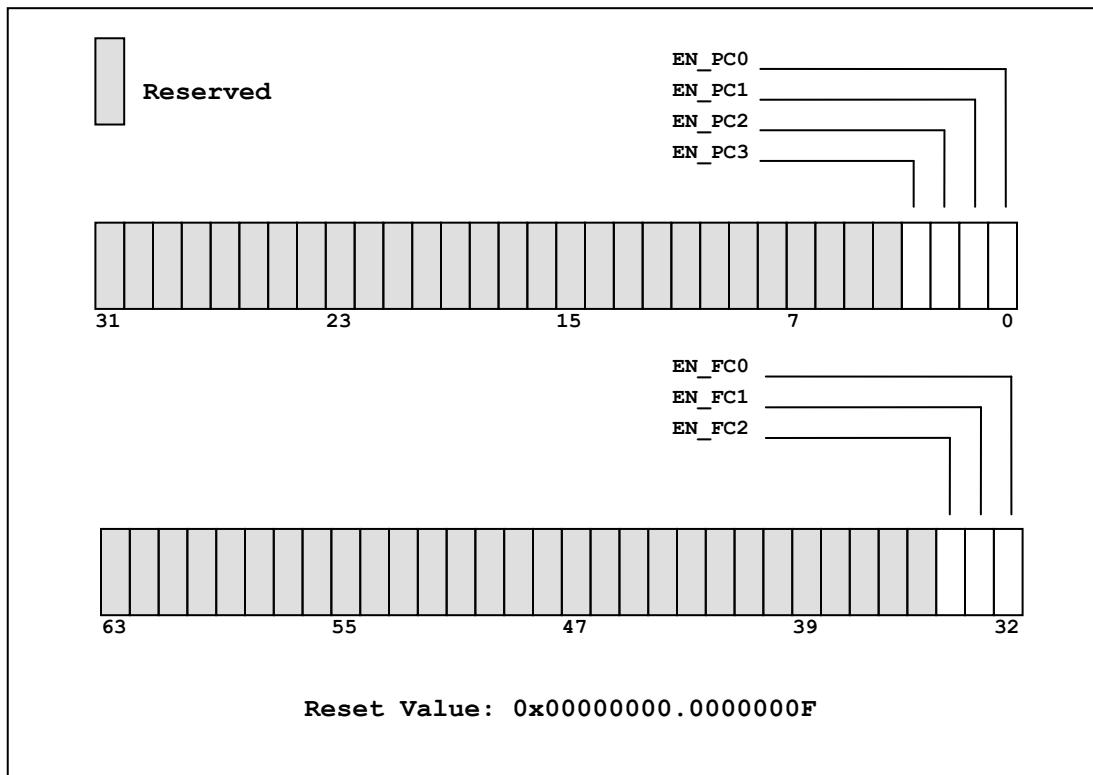


Table 5: IA32\_PERF\_GLOBAL\_CTRL Programming

Bit	Bit Position	Access	Description
EN_PC0	0	RW	Enable bit for programmable counter 0. If clear, disable the counter. If set, enable the counter.
EN_PC1	1	RW	Enable bit for programmable counter 1. If clear, disable the counter. If set, enable the counter.
EN_PC2	2	RW	Enable bit for programmable counter 2. If clear, disable the counter. If set, enable the counter.
EN_PC3	3	RW	Enable bit for programmable counter 3. If clear, disable the counter. If set, enable the counter.
EN_FC0	32	RW	Enable bit for fixed counter 0. If clear, disable the counter. If set, enable the counter.
EN_FC1	33	RW	Enable bit for fixed counter 1. If clear, disable the counter. If set, enable the counter.
EN_FC2	34	RW	Enable bit for fixed counter 2. If clear, disable the counter. If set, enable the counter.
Software must read-modify-write or explicitly clear reserved bits.			

### 3.1.4. IA32\_PERF\_GLOBAL\_STATUS

This register indicates the overflow status of each of the fixed and programmable counters. The upper bits provide additional status information of the PerfMon facilities. A set bit indicates an overflow has occurred in the corresponding counter. Overflow status bits in this register are cleared by writing the IA32\_PERF\_GLOBAL\_OVF\_CTRL register.

Status bit indications in this register have no affect on interrupts or pending interrupts.

Figure 4: IA32\_PERF\_GLOBAL\_STATUS MSR Definition

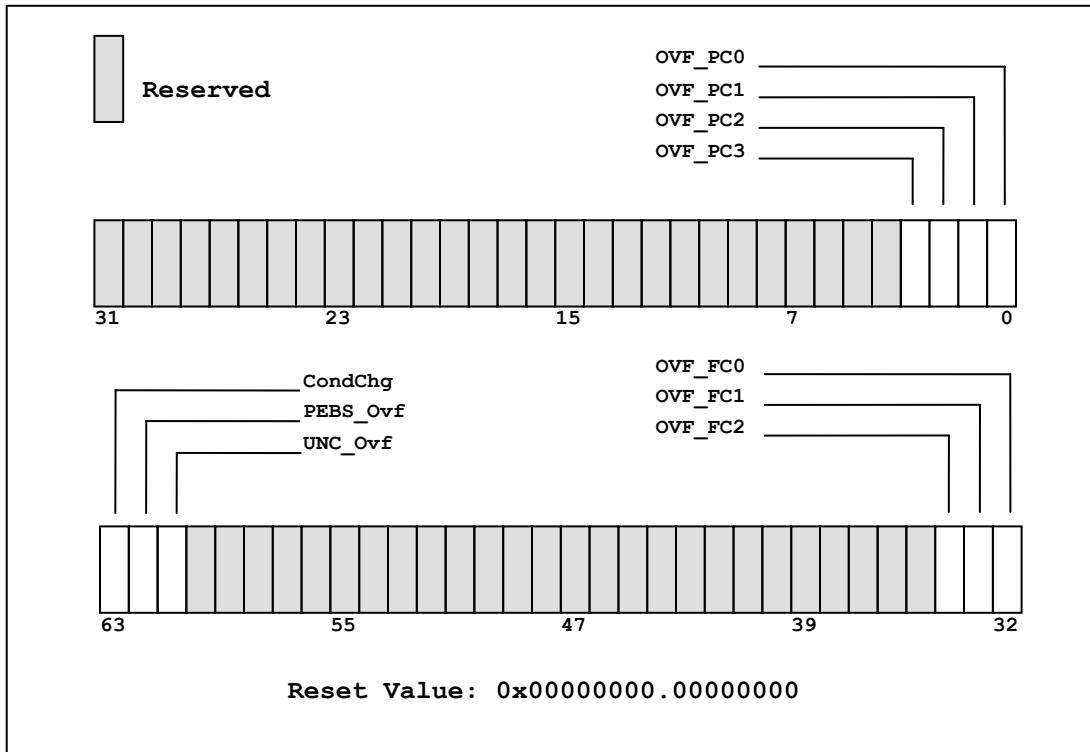


Table 6: IA32\_PERF\_GLOBAL\_STATUS Programming

Bit	Bit Position	Access	Description
OVF_PC0	0	RO	Indicates the overflow status of programmable counter 0. If this bit is clear no overflow occurred. If set then overflow occurred.
OVF_PC1	1	RO	Indicates the overflow status of programmable counter 1. If this bit is clear no overflow occurred. If set then overflow occurred.

OVF_PC2	2	RO	Indicates the overflow status of programmable counter 2. If this bit is clear no overflow occurred. If set then overflow occurred.
OVF_PC3	3	RO	Indicates the overflow status of programmable counter 3. If this bit is clear no overflow occurred. If set then overflow occurred.
OVF_FC0	32	RO	Indicates the overflow status of fixed counter 0. If this bit is clear no overflow occurred. If set then overflow occurred.
OVF_FC1	33	RO	Indicates the overflow status of fixed counter 1. If this bit is clear no overflow occurred. If set then overflow occurred.
OVF_FC2	34	RO	Indicates the overflow status of fixed counter 2. If this bit is clear no overflow occurred. If set then overflow occurred.
UNC_Ovf	61	RO	Indicates a counter in the uncore overflowed. Software must perform further queries to determine which specific counter overflowed.
PEBS_Ovf	62	RO	Indicates that the PEBS buffer threshold was reached and microcode scheduled a performance interrupt to indicate this condition.
CondChg	63	RO	Indicates that the state of the PerfMon hardware has changed. A change in this bit indicates the hardware has become available or unavailable. Software can execute CPUID leaf 0xA to confirm the availability of PerfMon hardware. This is a sticky bit, and must be explicitly cleared by writing to one the corresponding bit in the IA32_PERF_GLOBAL_OVF_CTRL MSR.
Software must read-modify-write or explicitly clear reserved bits.			

### 3.1.5. IA32\_PERF\_GLOBAL\_OVF\_CTRL

The IA32\_PERF\_GLOBAL\_OVF\_CTRL provides software the ability to clear status bits set in the IA32\_PERF\_GLOBAL\_STATUS register, described in the preceding

section. This is a write-only register. To clear overflow or condition change status in the global status register, software must write the corresponding bits in this register to binary one.

Figure 5: IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR Definition

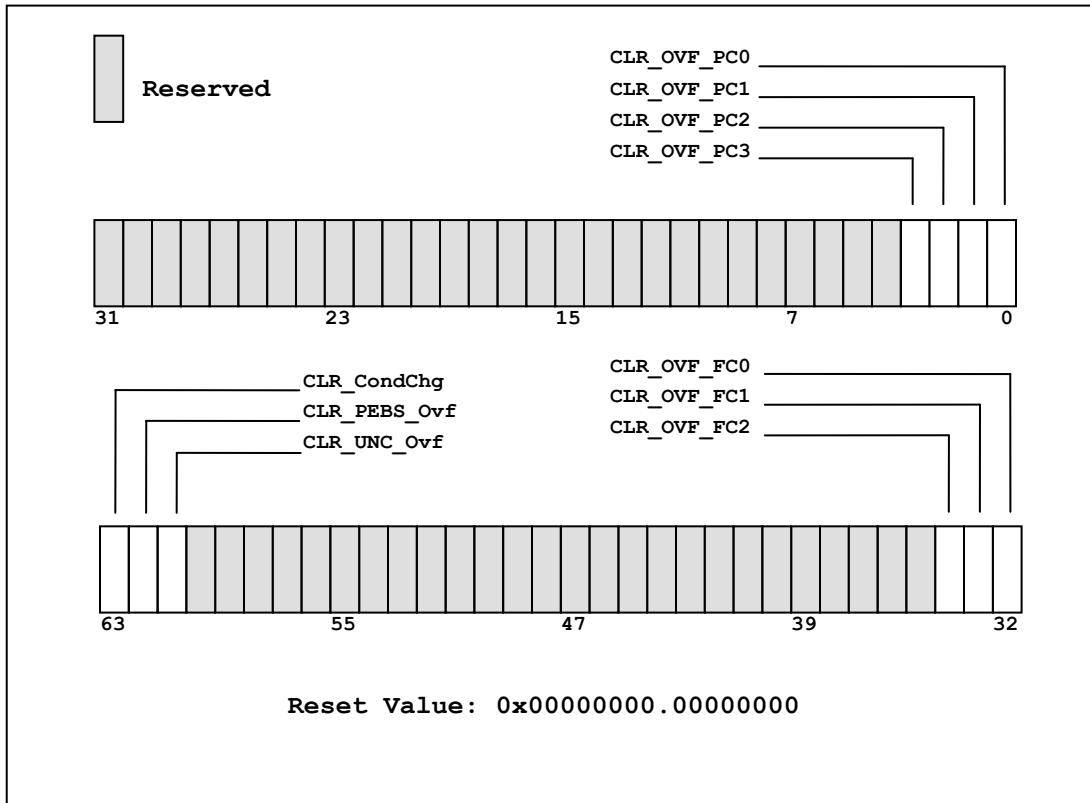


Table 7: IA32\_PERF\_GLOBAL\_OVF\_CTRL Programming

Bit	Bit Position	Access	Description
CLR_OVF_PC0	0	W01	Setting this bit will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for programmable counter 0.
CLR_OVF_PC1	1	W01	Setting this bit will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for programmable counter 1.
CLR_OVF_PC2	2	W01	Setting this bit will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for programmable counter 2.
CLR_OVF_PC3	3	W01	Setting this bit will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for programmable counter 3.

CLR_OVF_FC0	32	W01	Setting this bit will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for fixed counter 0.
CLR_OVF_FC1	33	W01	Setting this bit will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for fixed counter 1.
CLR_OVF_FC2	34	W01	Setting this bit will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for fixed counter 2.
CLR_UNC_Ovf	61	W01	Setting this bit clears the UNC_Ovf status bit in the IA32_PERF_GLOBAL_STATUS register.
CLR_PEBS_Ovf	62	W01	Setting this bit clears the PEBS_Ovf status bit in the IA32_PERF_GLOBAL_STATUS register.
CLR_CondChg	63	W01	Setting this bit clears the CondChg status bit in the IA32_PERF_GLOBAL_STATUS register.
Note: Writing any value other than '1' to these bits will be ignored.			

### 3.1.6. IA32\_FIXED\_CTR\_CTRL

This register controls the fixed counters, and determines whether these counters count in USR or SUP mode, or both, and whether these counters are enabled to generate performance interrupts.

Figure 6: IA32\_FIXED\_CTR\_CTRL MSR Definition

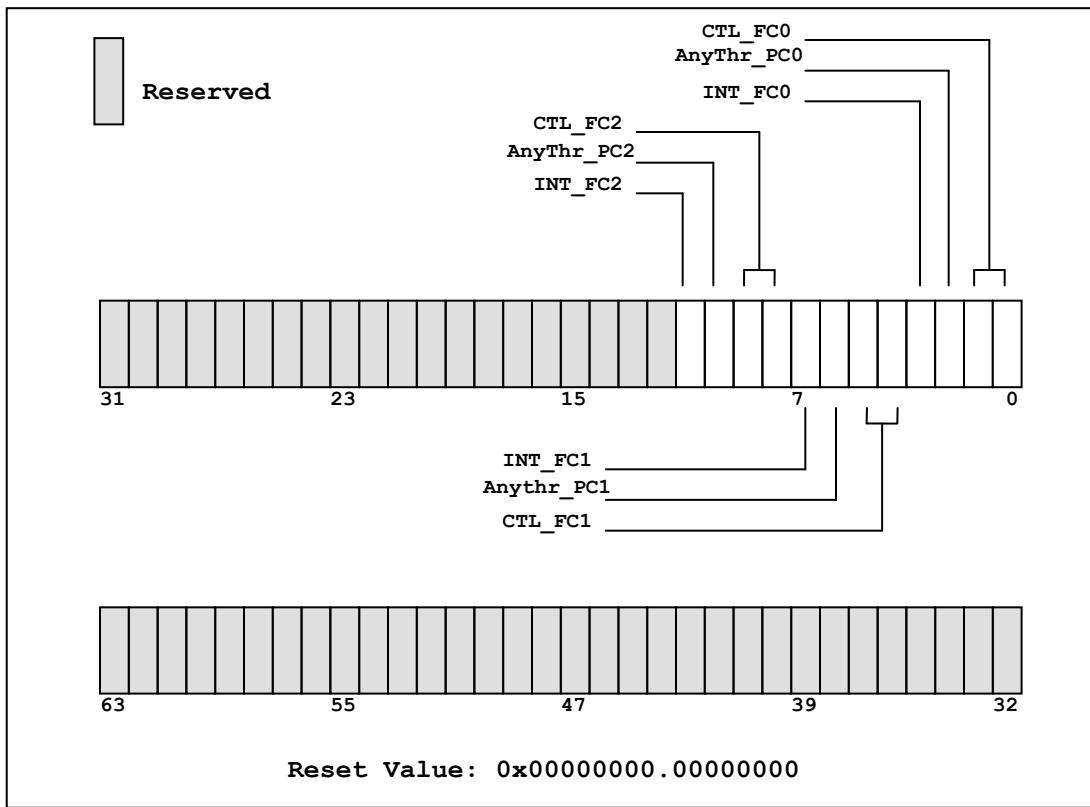


Table 8: IA32\_FIXED\_CTR\_CTRL Programming

Bit	Bit Position	Access	Description
CTL_FC0	1:0	RW	See Table 9.
AnyThr_FC0	2	RW	If clear the counter will count only events which occur on its own thread. If set, the counter will count events that occur on all threads in the core containing this thread (logical processor).
INT_FC0	3	RW	When set counter 0 is enabled to generate overflow interrupts. Interrupts are disabled if clear.
CTL_FC1	5:4	RW	See Table 9.
AnyThr_FC1	6	RW	If clear the counter will count only events which occur on its own thread. If set, the counter will count events that occur on all threads in the core containing this thread (logical processor).

INT_FC1	7	RW	When set counter 1 is enabled to generate overflow interrupts. Interrupts are disabled if clear.
CTL_FC2	9:8	RW	See Table 9.
AnyThr_FC2	10	RW	If clear the counter will count only events which occur on its own thread. If set, the counter will count events that occur on all threads.
INT_FC2	11	RW	When set counter 2 is enabled to generate overflow interrupts. Interrupts are disabled if clear.
Software must read-modify-write or explicitly clear reserved bits.			

Table 9: Summary of CTL bit Programming for Fixed Counters

CTL_FCx Bit Field <sup>1</sup>	Description
00b	Disable the counter.
01b	Count events when the logical processor is executing in privilege level 0.
10b	Count events when the logical processor is executing in privilege levels one, two, or three.
11b	Count events when the logical processor is executing at any privilege level.

<sup>1</sup> CTL\_FCO, CTL\_FC1, or CTL\_FC2.

The all/my thread control bit (AnyThr\_FCx) is always readable and writeable, even when Hyper-Threading Technology is disabled on the NHM processor. If Hyper-Threading Technology is disabled then setting these bits will have no effect.

## 3.2. Programmable Counter Control Registers

This section describes the control registers for the four programmable counters.

### 3.2.1. PerfEvtSelX

The PerfEvtSelX registers control the four programmable counters. Using these control registers, software can select the event to be counted, and the constraints under which those events are counted. Each counter must be locally enabled by this register, as well as globally enabled, in order to operate correctly.

The layout of this register is similar to previous Intel Core Architecture implementations. However, the pin control (PC) bit is now reserved. Also, NHM is the first Intel Core Architecture processor that implements SMT. NHM implements an additional event modifier bit, **AnyThr**, which controls counting events for the counter's logical processor only, or for all logical processors in the core which contains the counter.

Figure 7: PerfEvtSelX MSR Definition

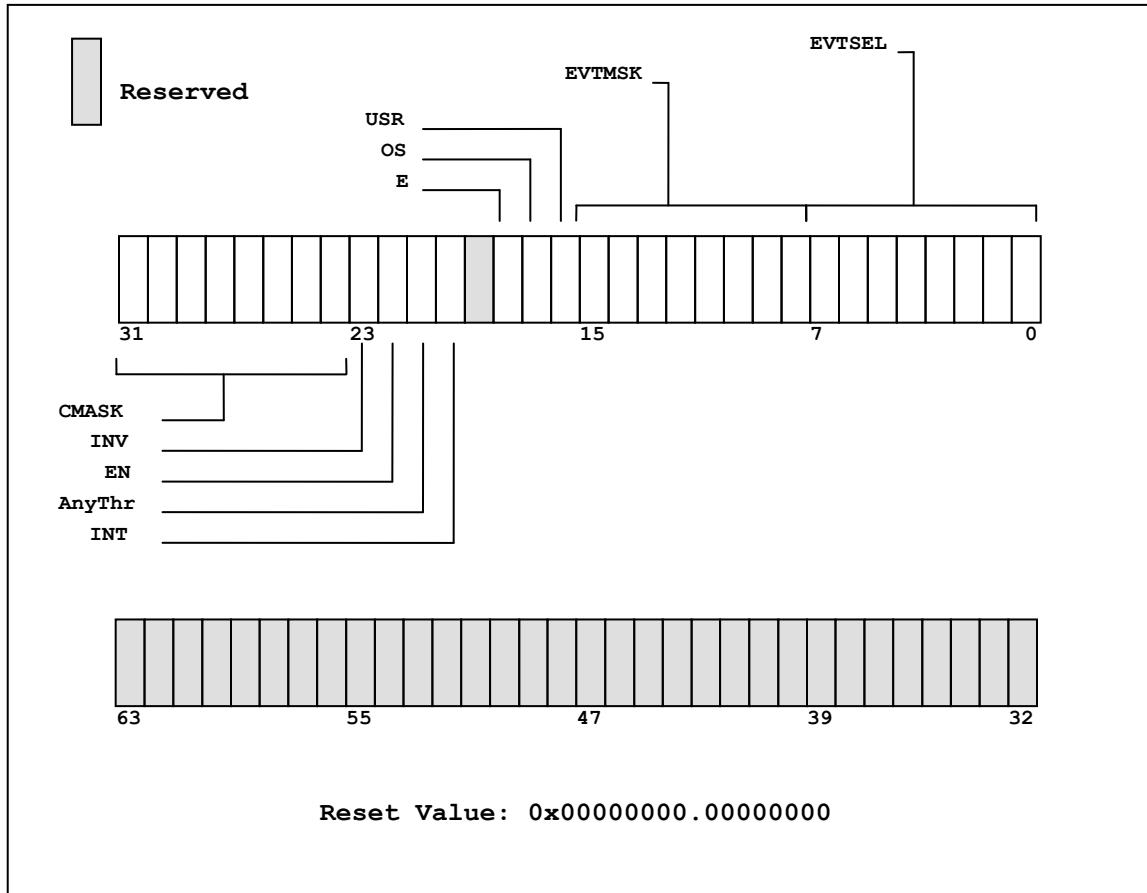


Table 10: PerfEvtSelX Programming

Bit	Bit Position	Access	Description
EVTSEL	7:0	RW	Selects the event logic unit used to detect micro-architectural conditions.
EVTMSK	15:8	RW	Condition qualifiers for the event selection logic specified in the EVTSEL field.
USR	16	RW	When set, indicates that the event specified by bit fields EVTSEL and EVTMSK is counted only when the logical processor is operating and privilege level 1, 2, or 3.

OS	17	Rw	When set, indicates that the event specified by bit fields EVTSEL and EVTMSK is counted only when the logical processor is operating and privilege level 0.
E	18	Rw	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.
INT	20	Rw	When set, the logical processor generates an exception through its local APIC on counter overflow. Counters only count up, and interrupts are generated on a transition from maximum count to zero. There will be some latency from the time the counter triggers the interrupt until the interrupt handler is invoked.
AnyThr	21	Rw	When clear, the counter increments only when event conditions are satisfied in its logical processor. When Set, the counter increments when event conditions are satisfied for any logical processor in the core in which this counter resides.
EN	22	Rw	When clear, this counter is locally disabled. When set, this counter is locally enabled.
INV	23	Rw	When clear, the CMASK field is interpreted as greater than or equal to. When set, the CMASK field is interpreted as less than.
CMASK	31:24	Rw	<p>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.</p> <p>Bits [31:29] are reserved to 0. Software must write these bits to zero.</p>
Software must read-modify-write or explicitly clear reserved bits.			

The all/my thread control bit (AnyThr) is always readable and writeable, even when Hyper-Threading Technology is disabled on the NHM processor. If Hyper-Threading Technology is disabled then setting this bit will have no effect.

### 3.3. Counter Registers

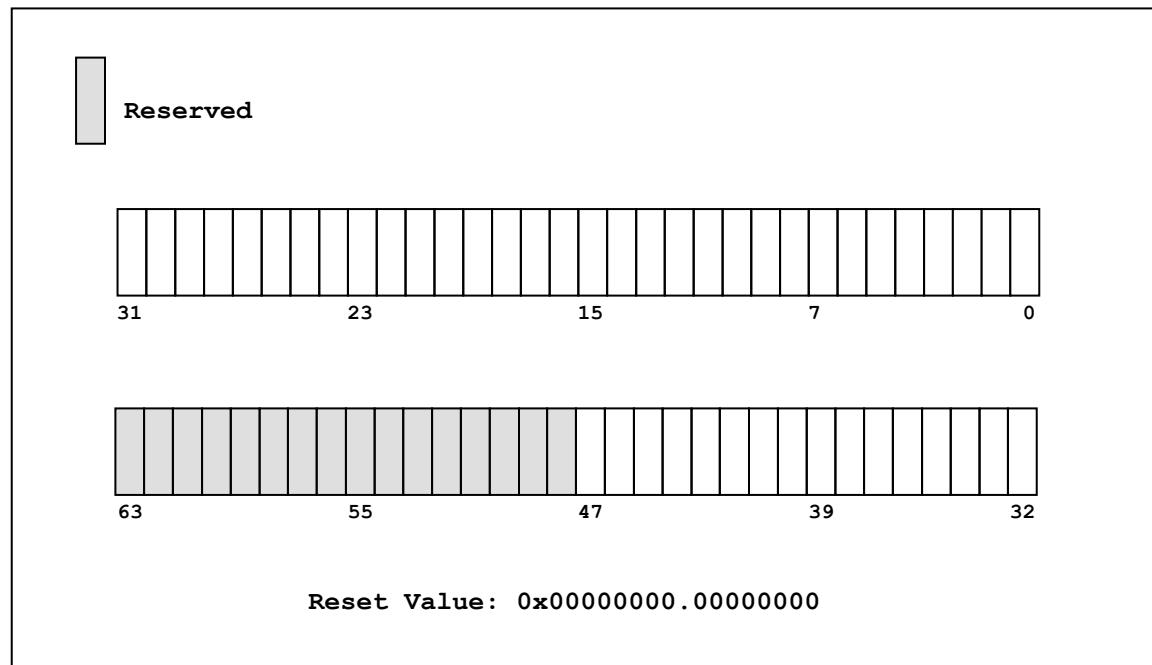
This section describes the three fixed and four programmable counter registers.

#### 3.3.1. `PERF_FIXED_CTRX` and `IA32_PMCX` Registers

Each counter register is 48-bits long. Counter registers can be cleared, or pre-loaded with count values as desired. This latter method is often used to set the point at which the counter will overflow, which is useful in event-based sampling. When writing the programmable counters using the `wrmsr` instruction, bits 32 through 47 cannot be written directly. They are sign-extended based on the value written to bit 31 of the counter register. When using the PEBS facility to re-load the programmable counters the entire 48-bit value is loaded from the DS Buffer Management area without any sign extension.

Previous implementations of Intel Core Architecture contained counters which were limited to 40 bits in length. Nehalem implements counters 48 bits in length. Counter width can be enumerated using the features in the `CPUD` instruction. See section 4.2 for more information.

Figure 8: `PERF_FIXED_CTRX` and `IA32_PMCX` MSR Definition



### 3.4. Off-core Response Event Programming

NHM provides the ability to program two counters to report off-core response counts. These counters are programmed to detect the occurrence of off-core data requests that return with a particular response. To enable these events, software programs the PerfEvtSelX register as shown in the following table. This capability is not available in processors previous to NHM.

Table 11: Offcore Response Events Encoding

Event Encoding PerfEvtSelX[7:0]	Event Mask PerfEvtSelX[15:8]	Extra MSR to Program
0xB7	0x01	0x1A6 (OFFCORE_RSP_0)
0xBB	0x01	0x1A7 (OFFCORE_RSP_1)

After programming the appropriate event in the PerfEvtSelX register, software must also program the “extra” register as shown in the figure below. To properly program this extra register, software must set at least one request type and one response type bit. Otherwise, the event count reported will be zero. It is legal and useful to set multiple request and response type bits in order to obtain various classes of off-core response events.

Figure 9: OFFCORE\_RSP\_X MSR Definition

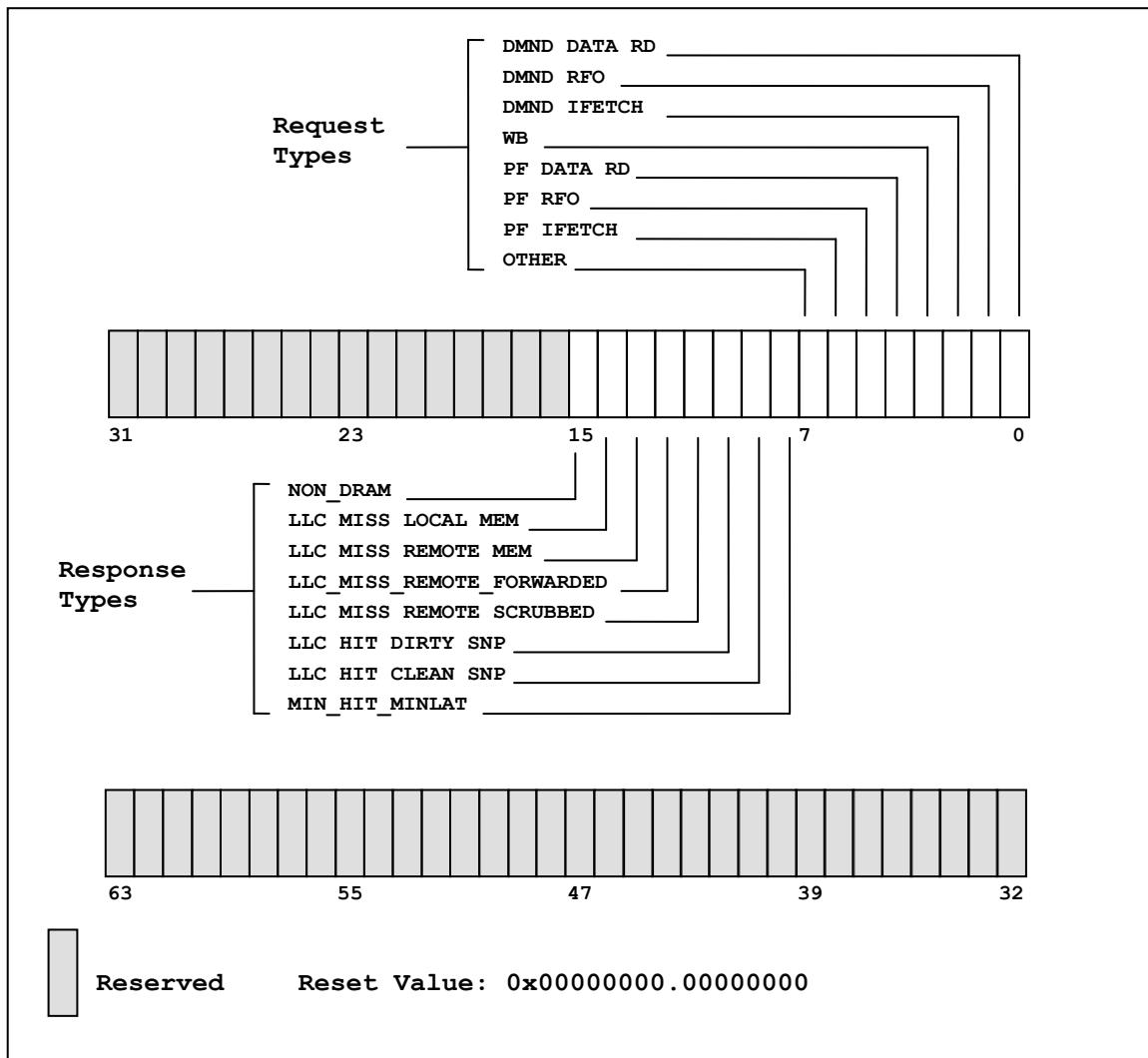


Table 12: OFFCORE\_RSP\_X Event Programming

Bit	Bit Position	Access	Description
DMND_DATA_RD	0	RW	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 MLC data read prefetches or instruction fetches.
DMND_RFO	1	RW	Counts the number of demand and DCU prefetch read for ownership (RFO) requests

				generated by a write to data cacheline. Does not count MLC RFO prefetches.
DMND_IFETCH	2	RW		Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count MLC code read prefetches.
WB	3	RW		Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	RW		Counts the number of data cacheline reads generated by the MLC prefetchers.
PF_RFO	5	RW		Counts the number of read for ownerships (RFO) requests generated by the MLC prefetchers.
PF_IFETCH	6	RW		Counts the number of code reads generated by the MLC prefetchers.
OTHER	7	RW		Counts these transactions: read code cacheline generated by monitor instruction, LLC invalidate, I/O, Write data partial or full cacheline, USWC stores and non-temporal stores, cache line flush instruction generated by a cache line flush, fence, lock, unlock, or split-lock.
UNCORE_HIT	8	RW		LLC Hit: local or remote home requests that hit last level cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HIT_SNP	9	RW		LLC Hit: local or remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where no modified copies were found. (clean).

OTHER_CORE_HITM	10	RW	LLC Hit: local or remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where modified copies were found. (HITM)
REMOTE_CACHE_HITM	11	RW	LLC Miss: local or remote home requests that missed the last level cache and was serviced by forwarded data following a cross package snoop where a modified copy was found and coherency actions were taken. (Not supported until WSM-T0)
REMOTE_CACHE_FWD	12	RW	LLC Miss: local homed requests that missed the last level 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	RW	LLC Miss: remote home requests that missed the last level cache and were serviced by remote DRAM.
LOCAL_DRAM	14	RW	LLC Miss: local home requests that missed the last level cache and were serviced by local DRAM.
IO_CSR_MMIO	15	RW	None: Non-DRAM requests that are serviced by IOH.

As an example, assume software wishes to count demand data read requests that are satisfied by the LLC. To accomplish this, software could program the PMU registers as shown in the table below.

Table 13: Offcore Response Programming Example

Register	MSR Address	Programming	Comments
PerfEvtSel0	0x186	0x4301B7	Enable counting of off-core

			responses occurring in user and supervisor code.
OFFCORE_RSP_0	0x1A6	0x17	Demand data reads satisfied by the LLC. Data could be owned by this core, or forwarded as clean or modified by another core on this package.
IA32_PMC0	0xC1	--	This counter will accumulate the event counts.

## 3.5. The PEBS Facility

This section details the PEBS facility as implemented in the NHM processor. The PEBS facility allows software to profile workload behavior relative to a limited set of events. Event counters are preloaded so they will reach an overflow condition after the occurrence of a predefined number of events. On overflow of a PEBS-enabled counter, the PEBS facility is armed. At the occurrence of the next precise (PEBS) event, the processor will take an assist and capture machine state in a predefined memory buffer.

### 3.5.1. IA32\_PEBS\_ENABLE

Counters can be configured to periodically capture machine state and load latency information as shown in Table 14. The IA32\_PEBS\_ENABLED register is used to enable this facility.

Previous implementations of Intel Core Architecture supported the PEBS facility in only one counter (counter 0). NHM supports PEBS mode in all four programmable counters. The ability to capture load latency information is also new to NHM and will be discussed in a subsequent section.

Figure 10: IA32\_PEBS\_ENABLE MSR Definition

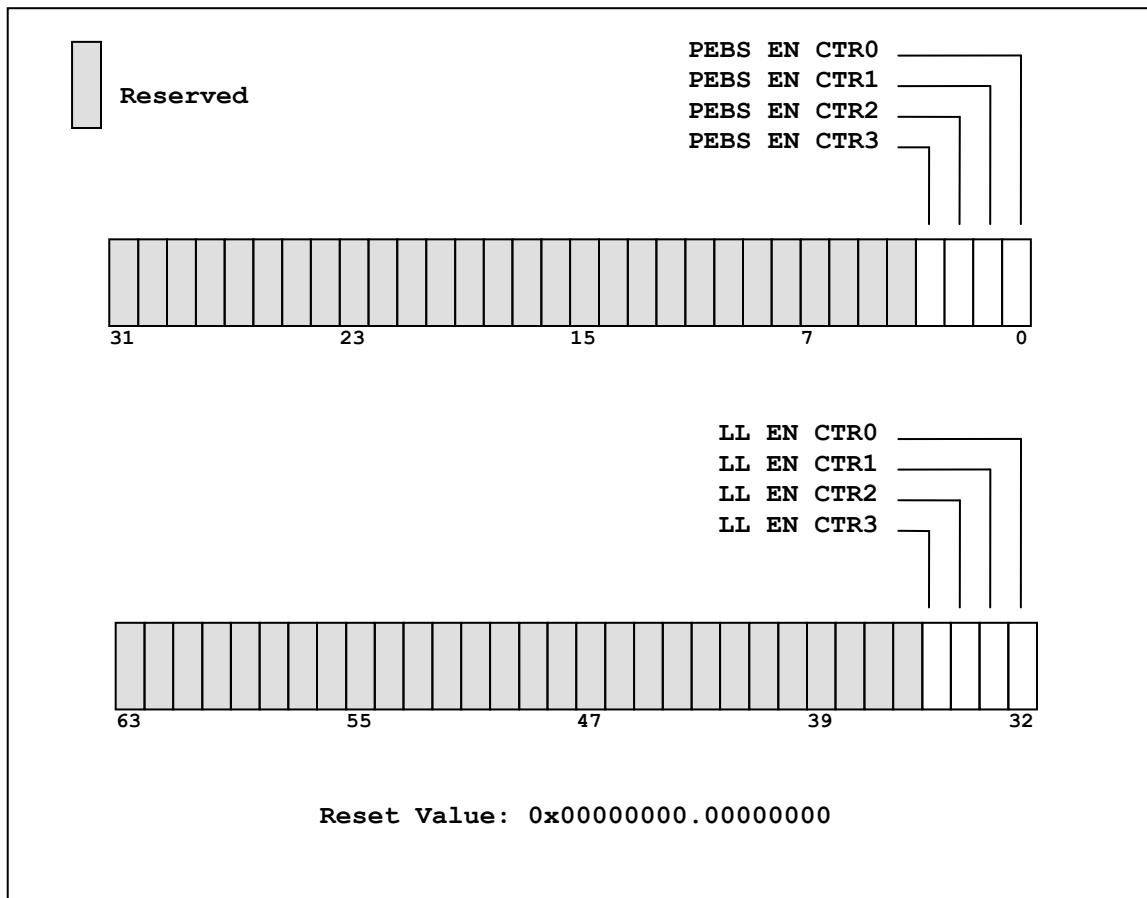


Table 14: IA32\_PEBS\_ENABLE Programming

Bit	Bit Position	Access	Description
PEBS_EN_CTR0	0	RW	Enable counter 0 to capture machine state on overflow.
PEBS_EN_CTR1	1	RW	Enable counter 1 to capture machine state on overflow.
PEBS_EN_CTR2	2	RW	Enable counter 2 to capture machine state on overflow.
PEBS_EN_CTR3	3	RW	Enable counter 3 to capture machine state on overflow.
LL_EN_CTR0	32	RW	Enable counter 0 to capture load latency information.
LL_EN_CTR1	33	RW	Enable counter 1 to capture load latency information.
LL_EN_CTR2	34	RW	Enable counter 2 to capture load latency information.

			latency information.
LL_EN_CTR3	35	RW	Enable counter 3 to capture load latency information.
Software must read-modify-write or explicitly clear reserved bits.			

### 3.5.2. PEBS Record Format

When a counter is enabled to capture machine state (PEBS\_EN\_CTRx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. In this mode, when the counter overflows from maximum count to zero, the PEBS hardware is armed. Upon occurrence of the next PEBS event, the PEBS hardware triggers 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].

PEBS assists on NHM are trap-like (see PEBS\_TRAP in Table 3). 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 below, and each field in the PEBS record is 64 bits long. This record format does not change regardless of IA32 or IA32e mode (compatibility or 64-bit mode). This behavior is different from previous implementations. Fields that are new to NHM are shaded.

Table 15: PEBS Record Format

Byte Offset	Field	Byte Offset	Field
0x00	R/EFLAGS	0x58	R9
0x08	R/EIP	0x60	R10
0x10	R/EAX	0x68	R11
0x18	R/EBX	0x70	R12
0x20	R/ECX	0x78	R13
0x28	R/EDX	0x80	R14

0x30	R/ESI	0x88	R15
0x38	R/EDI	0x90	IA32_PERF_GLOBAL_STATUS
0x40	R/EBP	0x98	Data Linear Address
0x48	R/ESP	0xA0	Data Source Encoding
0x50	R8	0xA8	Latency Value (core cycles)
<ol style="list-style-type: none"> <li>1. If the processor is operating in IA32 (32-bit) mode, then the 32-bit values of each register are written. The upper 32 bits in each field is written to zeroes. Fields for registers that are not defined in IA32 mode are written to zero.</li> <li>2. In IA32e mode, the full 64 bit value of each register is written.</li> <li>3. Shaded fields are new to NHM.</li> <li>4. Fields related to the load latency facility will be discussed in section 3.7.</li> </ol>			

- **IA32\_PERF\_GLOBAL\_STATUS**

The value written to this field 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.

- **Data Linear Address**
- **Data Source**
- **Latency Value**

These fields will be described in section 3.7.

Upon writing the PEBS record, microcode clears the overflow status bits in the IA32\_PERF\_GLOBAL\_STATUS corresponding to those counters that overflowed and were enabled in the IA32\_PEBS\_ENABLE register. The status bits of other counters remain unaffected.

### 3.5.3. Programming the PEBS Facility

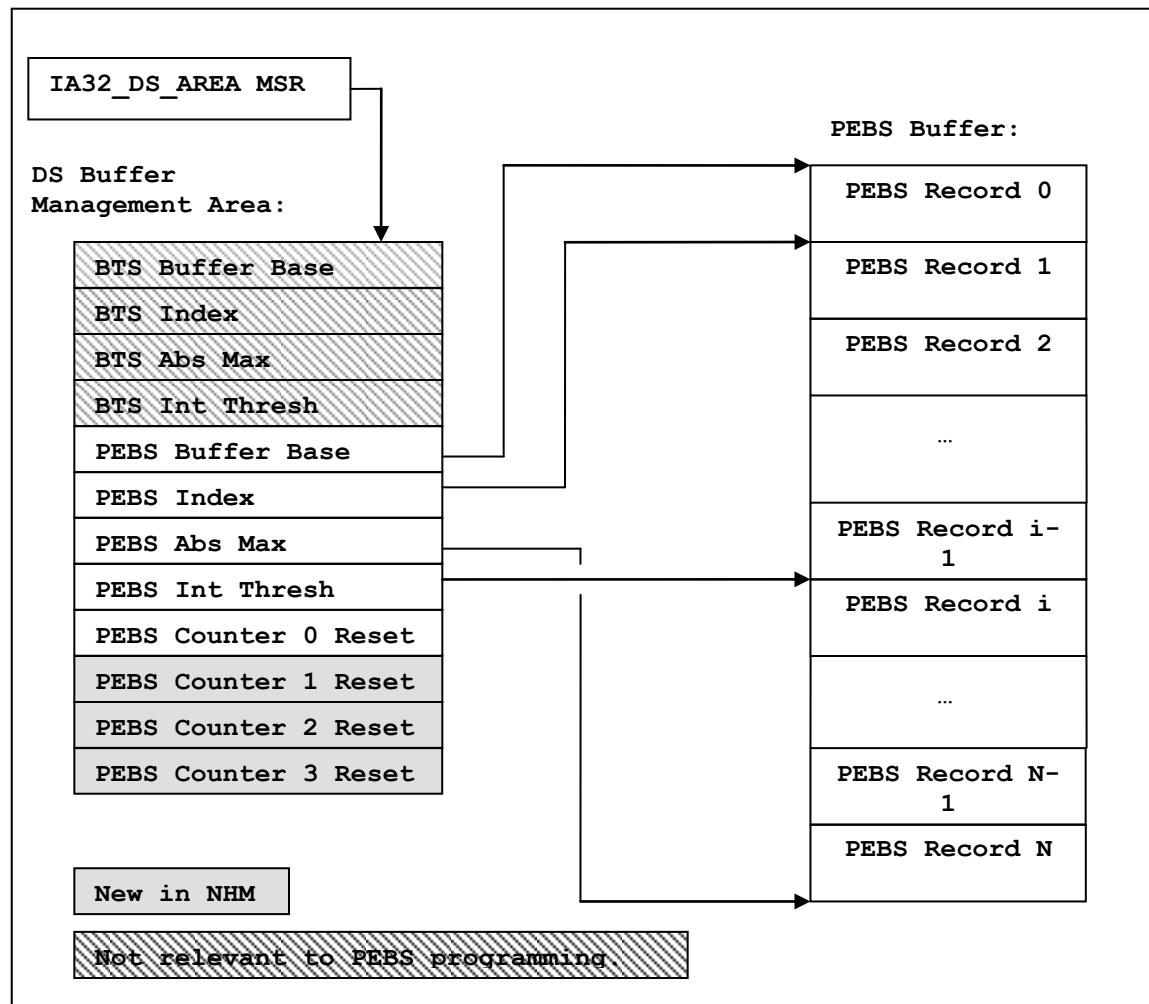
Software programs the PEBS facility by programming PEBS-enabled (precise) events in the PMU as described in section 3.2.1. Precise events are a subset of the total events supported by the PMU, and are listed in table 0. The PEBS hardware is enabled by setting the appropriate bit in the IA32\_PEBS\_ENABLE register for each counter programmed with a precise event.

Software must also initialize the DS\_BUFFER\_MANAGEMENT\_AREA data structure in memory which further describes the PEBS configuration. The PEBS-related fields of

this data structure are shown below. The beginning linear address of this data structure must be programmed into the IA32\_DS\_AREA register.

The overall relationship between the IA32\_DS\_AREA MSR, the DS\_BUFFER\_MANAGEMENT\_AREA, the PEBS record buffer, and the branch record buffer is shown below.

Figure 11: PEBS Programming Overview



- **PEBS Buffer Base**

This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. Microcode reads this field to determine the base address of the PEBS buffer. Software should allocate this memory from the non-paged pool.

- **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. Microcode reads this field to determine the location of the next PEBS record to write. After a PEBS record has been written, microcode 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 Abs Max**

This field represents the absolute maximum length of the PEBS buffer and is programmed with the linear address of the first byte past the end of the PEBS buffer. This indicates to microcode where the PEBS buffer ends.

- **PEBS Int Thresh**

This field represents the interrupt threshold and allows software to receive an interrupt notification indicating that the PEBS buffer is nearly exhausted. 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 writing a PEBS record, microcode checks the address of the next record to be written with the value of this field. If they are the same, microcode causes a performance interrupt. This is the same interrupt that is generated by a counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When this interrupt is generated the IA32\_PERF\_GLOBAL\_STATUS.PEBS\_Ovf bit will be set.

- **PEBS Counter X Reset**

This field allows software to set up PEBS counters to repeatedly trigger, generating multiple PEBS records. In this way, software can profile the execution of test code as desired. After each PEBS record is written, microcode checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32\_PEBS\_ENABLED is set). If these conditions are satisfied, then microcode reads the reset value for that counter from the DS Buffer Management Area and sets the counter to that value. For instance, 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. Software must specify the entire 48-bit value to be written to the counter register in this field. Unlike when using the `wrmsr` instruction, the value contained in this field is written to the counter register as is, and is not sign extended from bit 31.

When profiling test code, software typically desires to collect PEBS records or event data for every N events, where N is chosen to be a value that will provide statistically significant samples while not generating excessive intrusion. To accomplish this counters are typically pre-loaded with the value of negative N (-N), so that the counter will count up and overflow causing an interrupt for every N events detected.

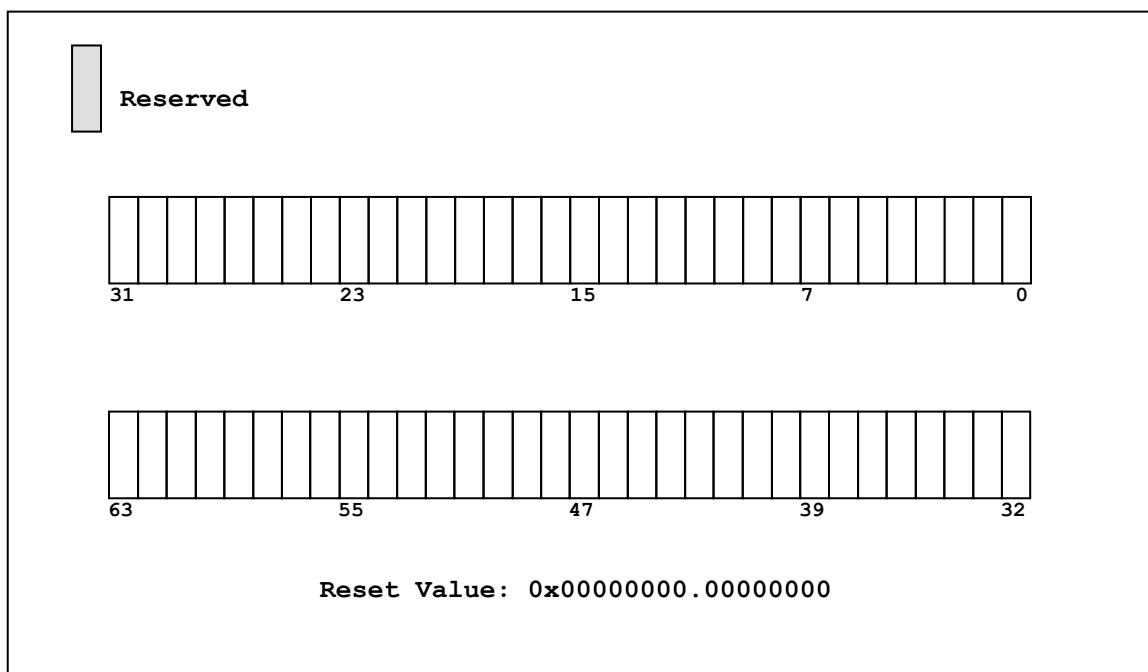
Note that the PEBS buffer is not treated as a circular buffer. Each time a PEBS record is written, microcode updates the “PEBS Index” field to the linear address of

the next PEBS record to write. Once this value becomes equal to that contained in the “PEBS Abs Max” field, microcode will simply stop writing PEBS records. No faults will be generated. To re-enable the PEBS buffer, software must reset the value of the “PEBS Index” field back to the base linear address of the PEBS buffer.

If software desires to take an interrupt for each PEBS record that is written, it may program the “PEBS Int Thresh” field with the linear address of the first byte of the second PEBS record in the PEBS buffer (PEBS Record 1 in the figure above). In this case, microcode will determine that the PEBS interrupt threshold was reached each time a PEBS record is written, and will trigger and PMI.

The definition of the IA32\_DS\_AREA MSR Definition is shown in the figure below. This MSR holds the linear address of the first byte of the IA32\_DS\_AREA memory buffer. All 64-bits must be programmed.

Figure 12: IA32\_DS\_AREA MSR Definition



### 3.6. Counter Prioritization

The interaction between counter overflows, PEBS, and interrupts is rather complex. This section addresses these complexities. Some background information is in order. Counter overflow interrupts are triggered by a counter transitioning from maximum count to zero (assuming 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 occur, which causes control to transition to microcode to handle the PEBS assist.

Counters (fixed and general) 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 counter. For instance, if IA32\_PMC1 and IA32\_PMC2 simultaneously cause an overflow interrupt and PEBS assist, respectively, 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 (programmable) 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).

Un-core counters may be programmed to interrupt one or more processor cores. It is possible for interrupt posted from the un-core to occur coincident with core counter overflow interrupts. Software must check core and un-core status registers to determine the precise source of counter overflow interrupts.

### 3.7. The Load Latency Facility

The load latency facility provides software a means to characterize load latency to different levels of the memory hierarchy. This facility is used in conjunction with the PEBS facility and is new in NHM processors. The facility measures latency from micro-operation (uop) dispatch to when data is globally observable (GO).

To use this feature software must assure:

- 1) A hardware counter is programmed with the MEM\_INST\_RETIRED event, and the LATENCY ABOVE THRESHOLD event mask must be specified (PerfEvtSelX[15:0] = 0x100B). The counter will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold. Stores are ignored when this event is programmed. Software must not program the CMASK or INV fields of the PerfEvtSelX register used for load latency (see Figure 7). Doing so will result in undefined behavior.
- 2) The 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).

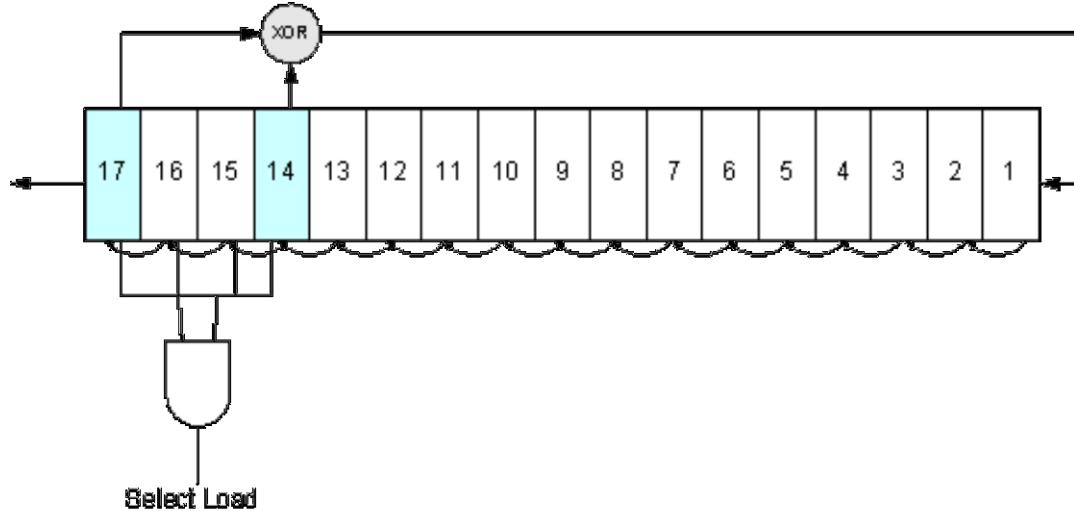
3) The PEBS enable bit in the IA32\_PEBS\_ENABLE register is set for the corresponding counter register (see Figure 10). This means that both the PEBS\_EN\_CTRX and LL\_EN\_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter zero, the IA32\_PEBS\_ENABLE register must be programmed with the 64-bit value 0x00000001.00000001.

When this facility is enabled hardware randomly tags load instructions to carry latency information, which is used to update internal data and source registers. These internal registers are continuously updated as each tagged load instruction retires.

When a PEBS assist occurs, the last value written to these internal registers will be read and subsequently written as part of the PEBS record. The PEBS sample after value (SAV) operates orthogonally to the instruction tagging mechanism. When enabled, the instruction tagging mechanism randomly tags memory load instructions from which to collect latency data. The SAV determines the number of qualified memory load instructions that will have to retire before a PEBS assist will be generated. The load latency data written to the PEBS record will be for the last qualified memory load instruction which retired just before the PEBS assist was invoked.

The hardware used to tag memory load instructions is shown in the figure below. It is a 17-bit, maximal length linear feedback shift register (LFSR) which generates a pseudo-random stream of 1's and 0's. Bits 14 and 17 are tapped to provide a pseudo-random sequence, and bits 14 through 17 are logically AND'ed together to generate the select signal.

Figure 13: LFSR for Tagging Memory Load Instructions



Given this configuration, each PEBS record written will contain load latency information, as shown in the shaded areas in Table 15.

- **Data Linear Address**

This is the linear address of the target of the load instruction. This address is non-canonical address. The lower 48 bits of the address are valid.

- **Latency Value**

This is the actual latency of the load in core clock cycles.

- **Data Source**

This value indicates the origin of the data obtained by the load instruction, and takes on the value of zero to fifteen, inclusive. The encoding for this value is shown in the table below. In the descriptions local memory refers to memory physically attached to this package, and remote memory refers to memory physically attached to another package. These descriptions also refer to cache line states exclusive (E), shared (S), modified (M), invalid (I), and forward (F). Details of the MESI protocol are beyond the scope of this document.

**Table 16: Data Souce Encodings in Load Latency Records**

Encoding	Description
0x0	Unknown LLC cache miss
0x1	Minimal latency core cache hit. This request was satisfied by the data cache.
0x2	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
0x3	This data request was satisfied by the MLC.
0x4	LLC HIT. Local or Remote home requests that hit last level cache in the uncore with no coherency actions required (snooping).
0x5	LLC HIT. Local or Remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where no modified copies found. (clean).
0x6	LLC HIT. Local or Remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where modified copies found. (HITM).
0x7	Reserved.
0x8	LLC MISS. Local homed requests that missed the last level cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
0x9	LLC MISS. Local or Remote home requests that missed the last level cache and was serviced by forwarded data following a cross package snoop where a modified copy was found and coherency actions taken.

	(Not supported until WSM-T0).
0xA	LLC MISS. Local home requests that missed the last level cache and was serviced by local DRAM (go to shared state).
0xB	LLC MISS. Remote home requests that missed the last level cache and was serviced by remote DRAM (go to shared state).
0xC	LLC MISS. Local home requests that missed the last level cache and was serviced by local DRAM (go to exclusive state).
0xD	LLC MISS. Remote home requests that missed the last level cache and was serviced by remote DRAM (go to exclusive state).
0xE	Reserved.
0xF	The request was to un-cacheable memory.

The figure below shows the programming for the PEBS\_LD\_LAT\_THRESHOLD register.

Figure 14: PEBS\_LD\_LAT\_THRESHOLD MSR Definition

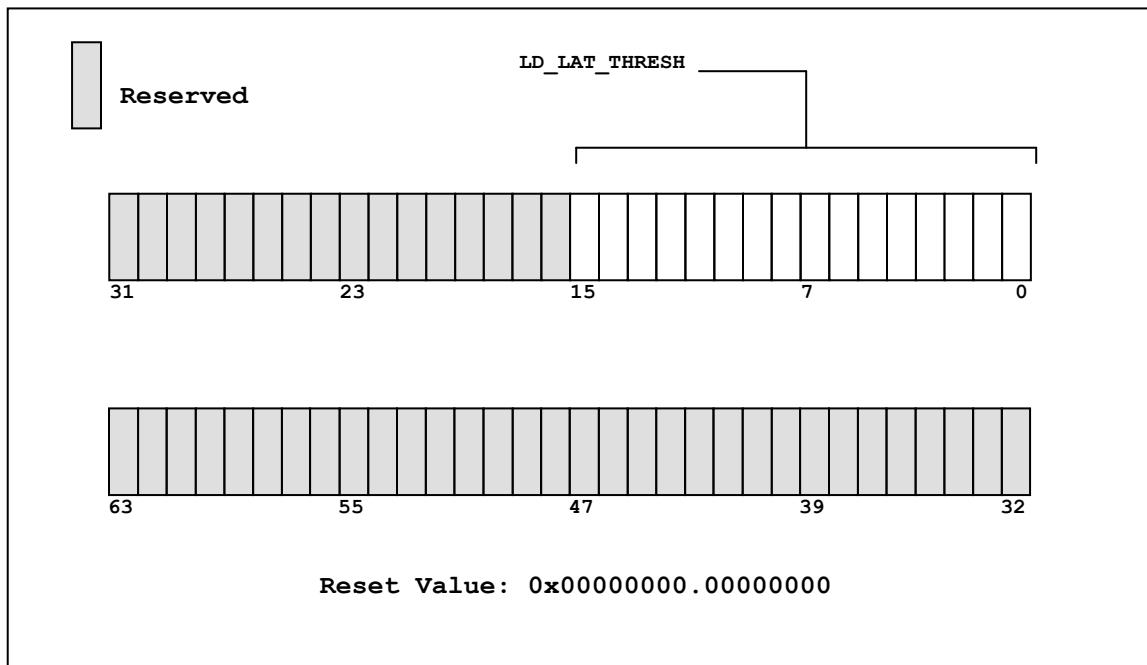


Table 17: PEBS\_LD\_LAT\_THRESHOLD Programming

Bit	Bit Position	Access	Description
LD_LAT_THRESH	15:0	RW	The threshold load latency in core clock cycles. Events with latencies greater than this value are counted 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 0x3.
--	--	--	---

### 3.8. The Last Branch Record Facility

This facility provides a means to capture the branch trace messages (BTM) in either an on-chip stack or memory buffer. Branch trace messages provide the source and destination addresses for executed branch instructions. The facility also provides a means for transmission of BTMs as a special cycle over the system bus, but details of that capability are beyond the scope of this document. Software directs the destination of the BTMs by programming certain bits in the IA32\_DEBUGCTL MSR, and by optionally programming certain fields of the DS Buffer Management Area, as will be shown below.

If IA32\_DEBUGCTL.BTS (bit 7) is set, enabling the BTS mechanism, then software must initialize the BTS fields in the DS Buffer Management Area. It is not possible to simultaneously store BTMs via the System Bus and to the BTS buffer.

Capturing BTMs in the on-chip LBR stack is enabled by setting IA32\_DEBUGCTL.LBR (bit 0). The on-chip LBR stack was extended on NHM to sixteen pairs of source and target addresses, as opposed to only four pairs in previous implementations. The details for programming this facility are summarized below.

Table 18: Programming the BTS Facility

IA32_DEBUGCTL					Description
TR	BTS	BTS_O FF_OS	BTS_ OFF_ USR	BTINT	
0	X	X	X	X	Branch trace messages (BTMs) off.
1	0	X	X	X	BTMs are transmitted via a special bus cycle on the system bus.
1	1	0	0	0 <sup>1</sup>	All branches are recorded in the BTS (memory buffer).
1	1	1	0	0 <sup>1</sup>	Branches occurring when CPL greater than zero are recorded in the BTS.
1	1	0	1	0 <sup>1</sup>	Branches occurring when CPL is equal to zero are recorded in the BTS.
1	1	1	1	X	BTMs are transmitted via a special bus cycle on the system bus.

1	1	0	0	1 <sup>2</sup>	All branches are recorded in the BTS.
1	1	1	0	1 <sup>2</sup>	Branches occurring when CPL greater than zero are recorded in the BTS.
1	1	0	1	1 <sup>2</sup>	Branches occurring when CPL is equal to zero are recorded in the BTS.

<sup>1</sup>The BTS is treated as a circular buffer.

<sup>2</sup>Upon reaching the BTS interrupt threshold a Performance Monitoring interrupt will be generated. If the BTS\_Index field in the DS Buffer Management Area is not reinitialized and the BTS\_Abs\_Max value is exceeded, the processor will stop writing BTMs.

### 3.8.1. LBR Filtering

Nehalem introduces the ability to filter the capture of LBR information based on various conditions. NHM allows software to allow or disallow capturing LBR data based on current privilege level, branches taken due to relative or indirect calls and jumps, and other conditions. When filtering is set, branch trace information is not captured in the branch trace store memory buffer or last branch record stack. Programming this filtering capability is detailed in the figure and table below.

Figure 15: LBR\_SELECT MSR Programming

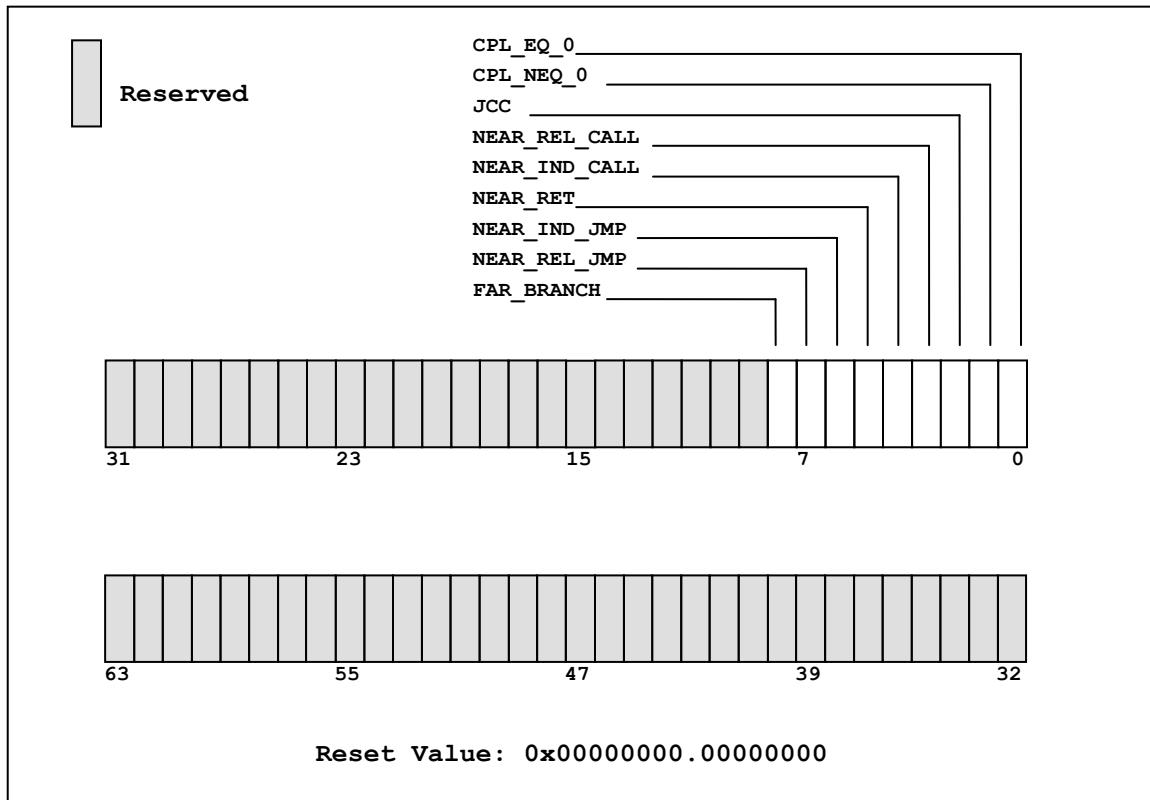


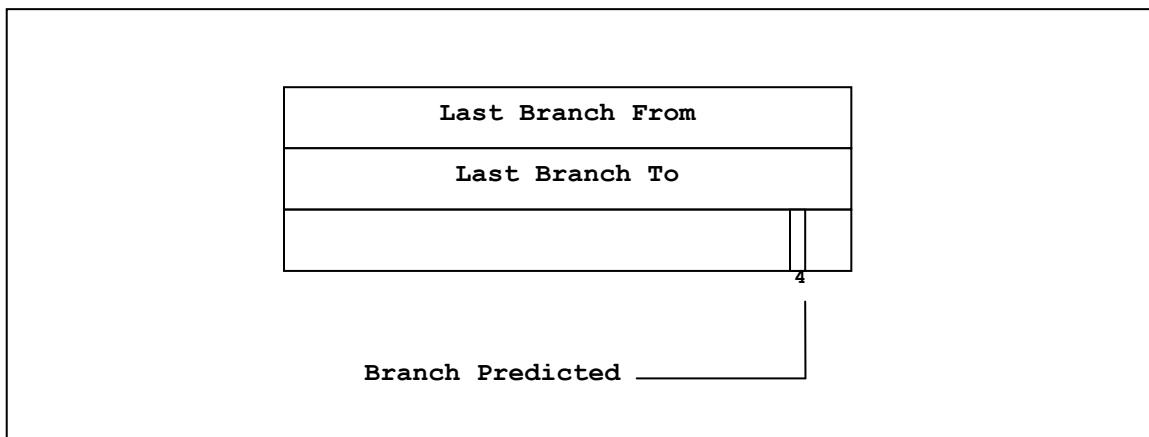
Table 19: LBR\_SELECT Programming

Bit	Bit Position	Access	Description
CPL_EQ_0	0	RW	When set do not capture branches occurring in ring 0.
CPL_NEQ_0	1	RW	When set do not capture branches occurring in CPL other than ring 0.
JCC	2	RW	When set do not capture conditional branches.
NEAR_REL_CALL	3	RW	When set do not capture near relative calls.
NEAR_IND_CALL	4	RW	When set do not capture near indirect calls.
NEAR_RET	5	RW	When set do not capture near returns.
NEAR_IND JMP	6	RW	When set do not capture near unconditional indirect jumps.
NEAR_REL JMP	7	RW	When set do not capture near unconditional relative branches.
FAR_BRANCH	8	RW	When set do not capture far branches.

### 3.8.2. Storing Branch Trace Messages in the Branch Trace Store

When the BTS mechanism is enabled BTMs are written to the BTS memory buffer. The format of BTMs written to the BTS is shown in the figure below.

Figure 16: Format of Branch Trace Message in Branch Trace Store



Microcode writes the source and destination linear addresses of the branch instruction for which this record was written. This record format does not change regardless of IA32 or IA32e mode, but in IA32 mode only the lower 32 bits of the linear addresses will be meaningful. The Branch Predicted bit indicates if this branch

was predicted or not. If this bit is set, the branch was predicted. Otherwise, the branch was not predicted (miss-predicted).

Software indicates the location of the BTS buffer by initializing the DS Buffer Management Area, which is also used in a similar fashion for the PEBS facility. Software initializes the following fields of the DS Buffer Management area:

- **BTS Buffer Base**

This is the base linear address of the first byte of the BTS memory buffer. Microcode reads this field to determine the base address of the BTS buffer. Software should allocate this memory from the non-paged pool.

- **BTS Index**

This field is initially programmed with the same value as the **BTS Buffer Base** field, or the beginning linear address of the BTS buffer. Microcode reads this field to determine the location of the next BTM record to write. After a BTM record has been written, microcode updates this field with the address of the next BTM record to be written. The figure above illustrates the state of **BTS Index** after the first BTM record is written.

- **BTS Abs Max**

This field represents the absolute maximum length of the BTS buffer and is programmed with the linear address of the first byte past the end of the BTS buffer. This indicates to microcode where the BTS buffer ends.

If IA32\_DEBUGCTL.BTINT (Bit 8) is clear, then the BTS buffer is treated as a circular buffer. When the BTS buffer is full, as defined by this field, microcode will wrap to the beginning of the buffer and overwrite the first record location with the latest BTM.

If IA32\_DEBUGCTL.BTINT is set, then BTMs will not be written beyond this limit, and data will potentially be lost.

- **BTS Int Thresh**

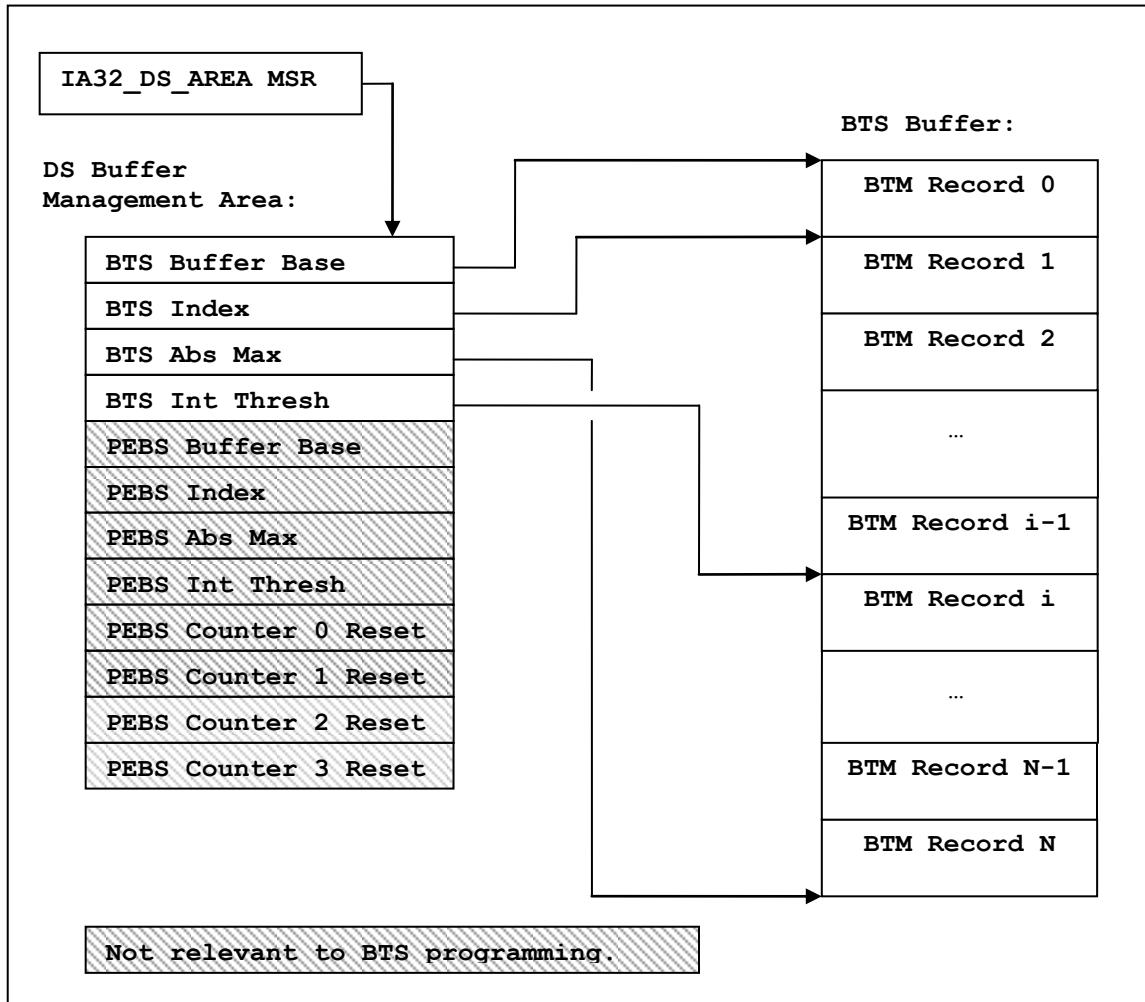
If IA32\_DEBUGCTL.BTINT is clear, this field is ignored and the BTS is treated as a circular buffer. No threshold interrupt is generated.

When IA32\_DEBUGCTL.BTINT is set, this field is used and represents the interrupt threshold, allowing software to receive an interrupt notification indicating that the BTS buffer is nearly exhausted. Software can detect this overflow condition if the value of the **BTS Index** field equals or exceeds the value of this field.

This field is programmed with the linear address of the first byte of the BTM record within the BTS buffer that represents the threshold record. After writing a BTM record, microcode checks the address of the next record to be written with the value of this field. If they are the same, microcode causes a performance interrupt.

This is the same interrupt that is generated by a counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC.

Figure 17: Branch Trace Store Programming Overview



### 3.8.3. Storing Branch Trace Messages in the Last Branch Record Stack

By setting the IA32\_DEBUGCTL.LBR (Bit 0) control bit, software enables storage of BTMs in the on-chip LBR stack. This stack consists of sixteen pairs of MSRs which hardware writes as branch instructions are executed. Hardware writes this stack in a circular fashion, from lowest to highest indexed last-branch registers, and MSR\_LASTBRANCH\_TOS register contains an index (0-15) which refers to the most recent BTM written to the LBR stack. Software reads these MSR pairs to obtain the source and target addresses of the branch instructions.

The bit field IA32\_PERF\_CAPABILITIES[5:0] indicates the format of the LBR stack records. The figures and tables below detail this format as contained by the “from” and “to” data registers. Details of the top-of-stack MSR are also shown.

Figure 18: MSR\_LASTBRANCH\_x\_FROM\_IP MSR Definition

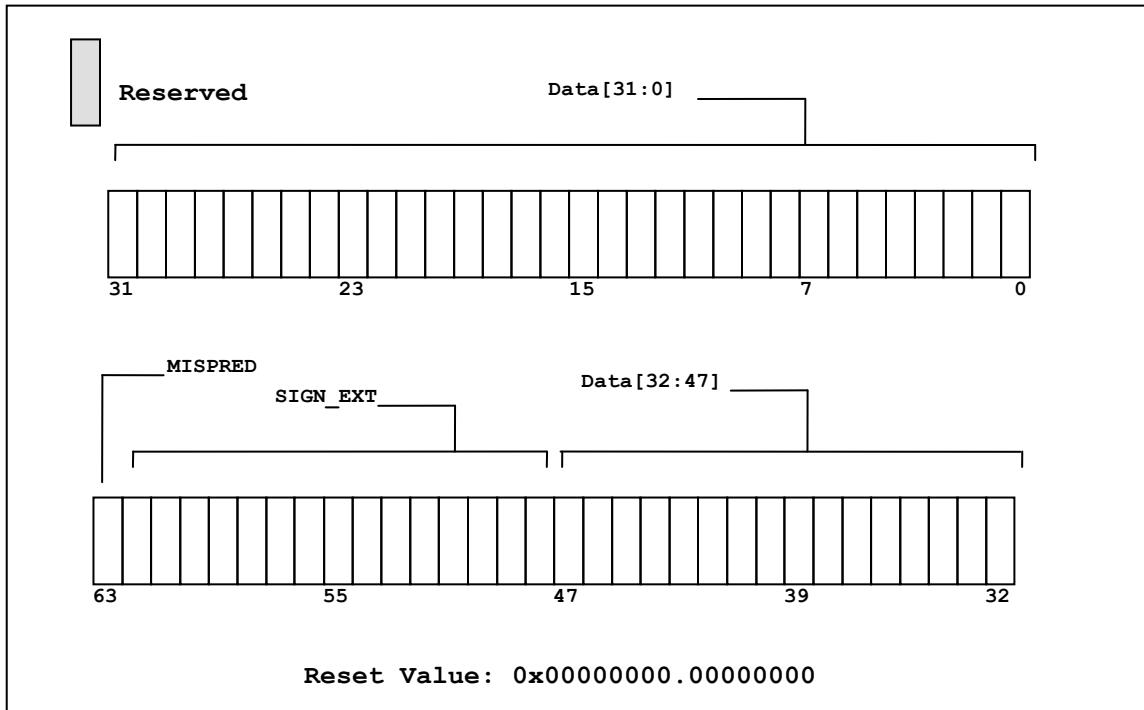


Table 20: MSR\_LASTBRANCH\_x\_FROM\_IP Programming

Bit	Bit Position	Access	Description
DATA	47:0	RO	The linear address of the branch instruction itself. This is the “branch from” address.
SIGN_EXT	62:48	RO	The sign extension of bit 47 of this register.
MISPRED	63	RO	When set, indicates the branch was predicted. Otherwise, the branch was not predicted (mispredicted).

Figure 19: MSR\_LASTBRANCH\_x\_TO\_IP MSR Definition

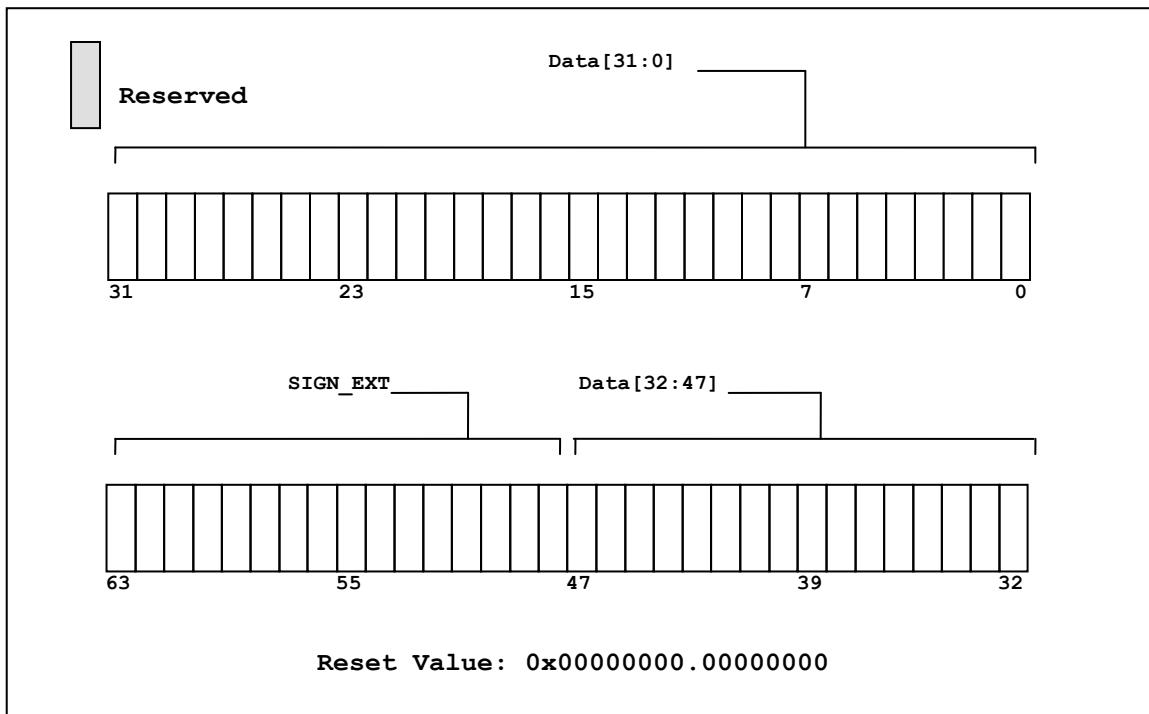


Table 21: MSR\_LASTBRANCH\_x\_TO\_IP Programming

Bit	Bit Position	Access	Description
DATA	47:0	RO	The linear address of the target of the branch instruction. This is the “branch from” address.
SIGN_EXT	63:48	RO	Sign extension of bit 47 of this register.

Figure 20: MSR\_LASTBRANCH\_TOS MSR Definition

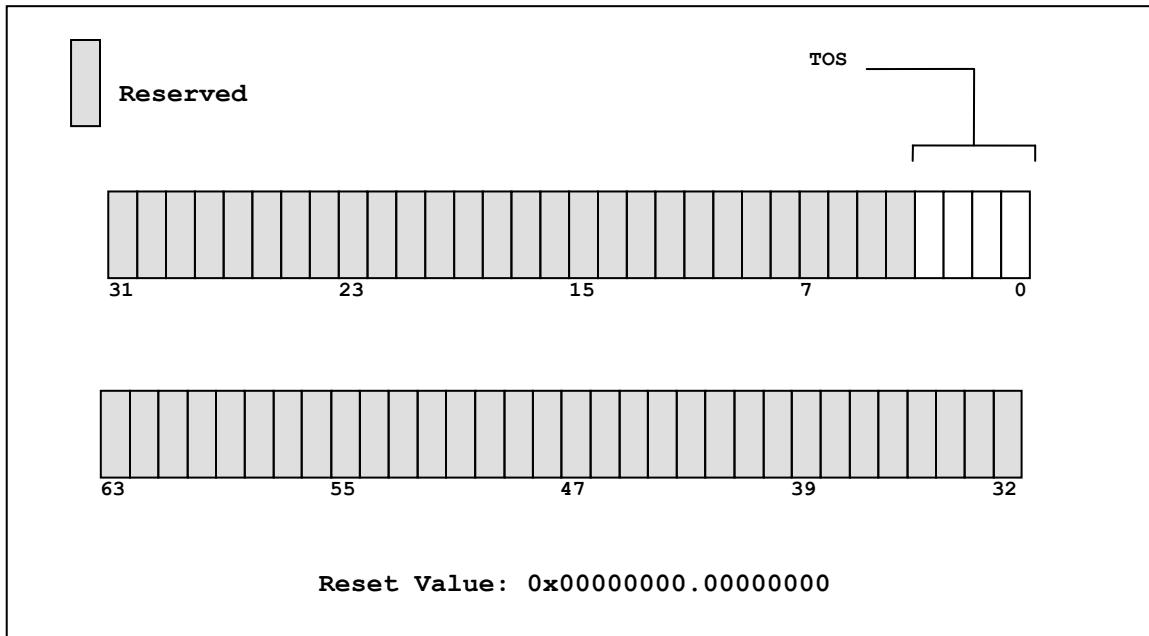


Table 22: MSR\_LASTBRANCH\_TOS Programming

Bit	Bit Position	Access	Description
TOS	3:0	RO	Contains the index of the register pair that contains the most recent BTM recorded in the LBR stack. To read the complete LBR stack, software will start reading BTMs from this location, to the end of the LBR stack (highest indexed register pair), then wrap and continue reading from the lowest indexed register pair to the register pair indexed by TOS - 1.

### 3.9. The RDPMC Instruction

All of the counter and counter control registers in the NHM core are implemented as model specific registers (MSR). There are two general instructions that allow software to access these registers: **rdmsr** (read an MSR) and **wrmsr** (write an MSR). The **rdmsr** instruction can be used to read the counter registers to obtain event counts as desired. However, due to constraints imposed on this more general instruction its latency is relatively high. Therefore, software is strongly encouraged to use the **rdpmc** instruction when reading counter registers, which improves read latency by approximately 50%. The **rdpmc** instruction may be used to read general and fixed counter registers by preloading the correct index value in the ECX register and executing the instruction.

Table 23: Legal Index (ECX) Values for Executing the RDPMC Instruction

ECX Value:	Reads:
0	IA32_PMC0
1	IA32_PMC1
2 <sup>1</sup>	IA32_PMC2
3 <sup>1</sup>	IA32_PMC3
0x40000000	PERF_FIXED_CTR0
0x40000001	PERF_FIXED_CTR1
0x40000002	PERF_FIXED_CTR2

<sup>1</sup> New in Nehalem.

Note: Using any values for ECX other than those listed above will cause a general protection fault when this instruction is executed.

## 4. Detecting the Nehalem Processor

To properly program the PMU facilities, software must obviously detect the presence of a Nehalem-based product. The following sections provide the information necessary for software to properly detect and configure for the Nehalem core and uncore PMUs.

### 4.1. CPUID identification

Software may use leaf one of the CPUID instruction (execute the **CPUID** instruction with the **EAX** register set to a value of **0x1**, or **CPUID.1** for short), and obtain the family, model, and stepping designations of the product.

While the Nehalem Core PMU architecture is consistent across all Nehalem implementations, uncore PMU architecture implementations may vary across the product line. However, the processor signature (family, model, and stepping) obtained from the **CPUID** instruction may still be used to uniquely identify the core and uncore PMU features. These details are summarized in the table below.

Table 24: Nehalem CPUID Processor Signatures

Processor Code Name	Extended Family [27:20]	Extended Model [19:16]	Type [13:12]	Family ID [11:8]	Model [7:4]	Stepping ID [3:0]	Composite Value (CPUID.1.EAX)
Nehalem-EP 2S (Gainstwon)	0x00	0x1	0x0	0x6	0xA	0xn	0x000106An
Nehalem-EP 1S (Bloomfield)	0x00	0x1	0x0	0x6	0xA	0xn	0x000106An
Nehalem-EX (Beckton)	0x00	0x2	0x0	0x6	0xE	0xn	0x000206En
Note: This information is subject to change without notice.							

### 4.2. Architectural PerfMon

Nehalem-based products conform to version 3.0 of the Architectural Core Performance Monitoring (ACPM) specification. This architecture enables software to confirm the presence of processor core PMU resources and enumerate their capabilities. It has no bearing on the uncore PMU resources. For more information refer to the ACPM specification or the Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B (reference the **CPUID** instruction).

ACPM provides a means by which software can detect the core PMU resources available on a particular product implementation. By executing leaf "A" of the **CPUID** instruction (**CPUID.A**), software can detect the version of the architectural

implementation of the PMU, the number and extent of general and fixed counters, and the architectural events which are available. After executing `CPUID.A` the registers `EAX`, `EBX`, and `EDX` contain the values described in the table below.

Table 25: CPUID.A Information

Register	Bit Field	Description	Comments
EAX	7:0	Version ID	Indicates the version of the Architectural Core Performance Monitoring specification that applies to this product.
	15:8	General Counters	Indicates the number of general (programmable) counters implemented in this product.
	23:16	Counter Width	Indicates the width in bits of each general counter. For instance, a value of 0x30 indicates a 48-bit counter.
	31:24	Vector Length	Indicates the length of the bit vector contained in the EBX register.
EBX	Variable (see length specified by EAX[31:24])	Architectural Events Vector	Indicates which architecturally defined events are supported in this product implementation. A value of zero in a bit position indicates that the event represented by that bit is supported. The following architectural events are defined: Bit 0 – Core Cycles Bit 1 – Instructions Retired Bit 2 – Reference Cycles Bit 3 – Last-level Cache References Bit 4 – Last-level Cache Misses Bit 5 – Branch Instructions Retired Bit 6 – Branch Miss-predicts Retired
EDX	4:0	Fixed Counters	Indicates the number of fixed function counters available on this product implementation.
	12:5	Counter Width	Indicates the width in bits of each fixed function counter.

### 4.3. Other Considerations

Software must first check the IA32\_MISC\_ENABLE MSR to verify that performance monitoring facilities and the PEBS facility are available. If bit 7 of this register is set, then the performance monitoring features of the processor core are available. If bit 12 of this register is clear, then the PEBS facility is available. Refer to the Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B for more details.

Software may also query the IA32\_PERF\_CAPABILITIES register to ascertain additional details of the PEBS facility. Please refer to section 3.1.1 for more information.

## Appendix A: Precise (PEBS-enabled) Events

Event Name	Event Encoding	Sub-event Name	Event Mask
MEM_INST_RETIR ED	0x0B	LOADS	0x01
		STORES	0x02
		LATENCY_ABOVE_THRESHOLD	0x10
MEM_STORE_RETIR ED	0x0C	STORE_MISS_IN_LAST_LEVEL_DTL B	0x01
		DROPPED_EVENTS	0x02
MEM_UNCORE_EVENT_R ETIRED	0x0F	LLC_DATA_MISS	0x01
		OTHER_CORE_L2_HIT	0x02
		OTHER_CORE_L2_HITM	0x04
		REMOTE_CACHE_HIT	0x08
		REMOTE_CACHE_HITM	0x10
		LOCAL_DRAM	0x20
		NON_LOCAL_DRAM	0x40
		IO	0x80
INST_RETIR ED	0xC0	ALL	0x01
		FP	0x02
		MMX	0x04
OTHER_ASSISTS	0xC1	PAGE_A/D_ASSISTS	0x01
UOPS_RETIR ED	0xC2	ALL_EXECUTED	0x01
		RETIRE_SLOTS	0x02
		MACRO_FUSED	0x04
BR_INST_RETIR ED	0xC4	CONDITIONAL	0x01
		NEAR_CALL	0x02
		ALL_BRANCHES	0x04
BR_MISP_RETIR ED	0xC5	CONDITIONAL	0x01

		NEAR_CALL	0x02
		ALL_BRANCHES	0x04
SSEX_UOPS_RETIRE	0xC7	PACKED_SINGLE	0x01
		SCALAR_SINGLE	0x02
		PACKED_DOUBLE	0x04
		SCALAR_DOUBLE	0x08
		VECTOR_INTEGER	0x10
ITLB_MISS_RETIRE	0xC8	ITLB_MISS	0x20
MEM_LOAD_RETIRE	0xCB	LOAD_HIT_L1	0x01
		LOAD_HIT_L2_MLC	0x02
		LOAD_HIT_L3_LLC	0x04
		LOAD_HIT_OTHER_PM_PKG_L2	0x08
		LLC_MISS	0x10
		DROPPED_EVENTS	0x20
		LOAD_HIT_LFB_BUT_MISSED_IN_L 1	0x40
		LOAD_MISS_IN_LAST_LEVEL_DTLB	0x80
BR_CND_MISPREDICTION	0xEB	BIMODAL	0x10
FP_ASSISTS	0xF7	ALL	0x01
		OUTPUT	0x02
		INPUT	0x04

## Appendix B: Core PMU MSR List

Table 26: NHM Core PMU Register List

Name	MSR Address
IA32_PMC0	0xC1
IA32_PMC1	0xC2
IA32_PMC2	0xC3
IA32_PMC3	0xC4
PerfEvtSel0	0x186
PerfEvtSel1	0x187
PerfEvtSel2	0x188
PerfEvtSel3	0x189
IA32_MISC_ENABLE	0x1A0
OFFCORE_RSP_0	0x1A6
OFFCORE_RSP_1	0x1A7
LBR_SELECT	0x1C8
MSR_LASTBRANCH_TOS	0x1C9
IA32_DEBUGCTL	0x1D9
PERF_FIXED_CTR0	0x309
PERF_FIXED_CTR1	0x30A
PERF_FIXED_CTR2	0x30B
IA32_PERF_CAPABILITIES	0x345
IA32_FIXED_CTR_CTRL	0X38D
IA32_PERF_GLOBAL_STATUS	0x38E
IA32_PERF_GLOBAL_CTRL	0x38F
IA32_PERF_GLOBAL_OVF_CTRL	0x390
IA32_PEBS_ENABLE	0x3F1
PEBS_LD_LAT_THRESHOLD	0x3F6
IA32_DS_AREA	0x600
MSR_LASTBRANCH_x_FROM_IP (0 ≤ x ≤ 15)	0x680-

	0x68F
MSR_LASTBRANCH_x_TO_IP (0 ≤ x ≤ 15)	0x6C0- 0x6CF

Copyright® 2010 Intel Corporation, all rights reserved

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: [http://www.intel.com/#/en\\_US\\_01](http://www.intel.com/#/en_US_01)

This document contains information on products in the design phase of development.

Intel®, Xeon® are trademarks of Intel Corporation in the U.S. and other countries.