60- and 66-MHz Pentium® Processor Specification Update

Release Date: February 1997

Order Number 243326-001

The Pentium® processor may contain design defects or errors known as errata. Characterized errata that may cause the Pentium processor's behavior to deviate from published specifications are documented in this specification update.

CONTENTS

REVISION HISTORY

Date of Revision	Version	Description
February 15, 1997	-001	This document consolidates information previously published in the *Pentium Processor Specification Update* (Order Number 242480).

PREFACE

This document is an update to the Specifications contained in the *Pentium® Processor at iCOMP index 510\60 MHz, Pentium Processor at iCOMP Index 567\66 MHz* datasheet (Order Number 241595). It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools. It contains Specification Changes, Errata, Specification Clarifications, and Documentation Changes.

NOTE

The following manuals referenced in this document are archived and are available on Intel's web site at http://www.intel.com: *Pentium® Processor Family Developer's Manual, Volume 1: Pentium Processor* (Order Number 241428-004) and the *Pentium® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual* (Order Number 241430-004).

Nomenclature

Specification Changes are modifications to the current published specifications. These changes will be incorporated in the next release of the specifications.

Errata are design defects or errors. Errata may cause the Pentium processor's behavior to deviate from published specifications. Hardware and software designed to be used with any given stepping must assume that all errata documented for that stepping are present on all devices.

Specification Clarifications describe a specification in greater detail or further highlight a specification's impact to a complex design situation. These clarifications will be incorporated in the next release of the specifications.

Documentation Changes include typos, errors, or omissions from the current published specifications. These changes will be incorporated in the next release of the specifications.

Identification Information

The Pentium processor can be identified by the following register contents:

Family[1]	60- and 66-MHz Model 1[2]
05h	01h (described in Part I)

NOTES:

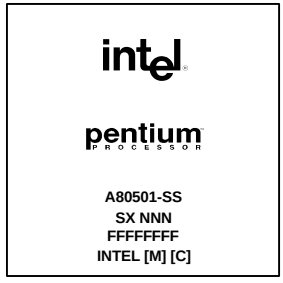
1. The Family corresponds to bits [11:8] of the EDX register after RESET, bits [11:8] of the EAX register after the CPUID instruction is executed, and the generation field of the Device ID register accessible through Boundary Scan.
2. The Model corresponds to bits [7:4] of the EDX register after RESET, bits [7:4] of the EAX register after the CPUID instruction is executed, and the model field of the Device ID register accessible through Boundary Scan.

Specification Update for 60- and 66-MHz Pentium® Processors

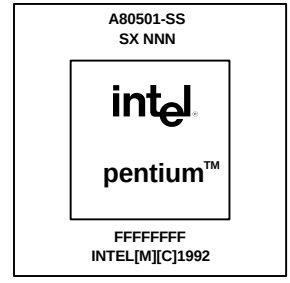
GENERAL INFORMATION

Top Markings

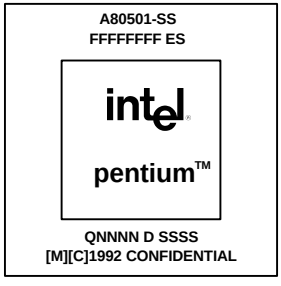
B1 Production Units – Top:



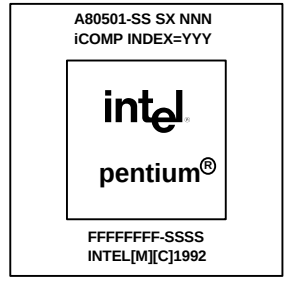
C1 Production Units – Top:



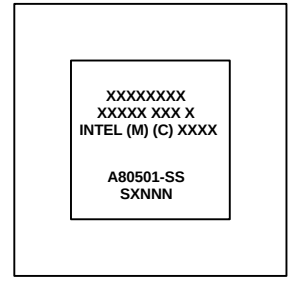
D1 Sample Units – Top:



D1 Production Units – Top:



D1 Production Units – Bottom:



NOTES:

- SS = Speed (MHz).
- SX NNN = S-Spec number.
- FFFFFFFF = FPO # (Test Lot Traceability #).
- For packages with heat spreaders, the inner line box defines the spreader edge.
- Ink Mark = All logo information on the heat spreader.
- Laser Mark = The two lines of information above and below the heat spreader. All bottomside information is laser mark.
- ES = Engineering Sample.
- QNNNN = Sample Specification number.
- SSSS = Serialization code.
- YYY = iCOMP® Index (510 for 60-MHz and 567 for 66-MHz product).
- On the bottomside, the inner line box defines the edge of the metallic package lid.
- Bottomside, first two lines = Reserved for Intel internal use.
- Bottomside, third line = Copyright information; the last four digits of this line are reserved for Intel internal use

Basic 60- and 66-MHz Pentium® Processor Identification Information

CPUID									
Type	Family	Model	Stepping	Mfg. Stepping	Speed (MHz) Core/Bus	S-Spec	V_{CC}	T_{CASE}	Notes
0	5	1	3	B1'	50/50	Q0399	4.75V-5.25V	0°C-85°C	1, 2
0	5	1	3	B1'	60/60	Q0352	4.75V-5.25V	0°C-85°C	1
0	5	1	3	B1'	60/60	Q0400	4.75V-5.25V	0°C-75°C	1, 2
0	5	1	3	B1'	60/60	Q0394	4.75V-5.25V	0°C-80°C	2, 3
0	5	1	3	B1'	66/66	Q0353	4.90V-5.25V	0°C-75°C	1
0	5	1	3	B1'	66/66	Q0395	4.90V-5.25V	0°C-70°C	2, 3
0	5	1	3	B1"	60/60	Q0412	4.75V-5.25V	0°C-85°C	1
0	5	1	3	B1"	60/60	SX753	4.75V-5.25V	0°C-85°C	1
0	5	1	3	B1"	66/66	Q0413	4.90V-5.40V	0°C-75°C	1
0	5	1	3	B1"	66/66	SX754	4.90V-5.40V	0°C-75°C	1, 4
0	5	1	5	C1	60/60	Q0466	4.75V-5.25V	0°C-80°C	3
0	5	1	5	C1	60/60	SX835	4.75V-5.25V	0°C-80°C	3
0	5	1	5	C1	60/60	SZ949	4.75V-5.25V	0°C-80°C	3,5
0	5	1	5	C1	66/66	Q0467	4.90V-5.40V	0°C-70°C	3
0	5	1	5	C1	66/66	SX837	4.90V-5.40V	0°C-70°C	3
0	5	1	5	C1	66/66	SZ950	4.90V-5.40V	0°C-70°C	3,5
0	5	1	7	D1	60/60	Q0625	4.75V-5.25V	0°C-80°C	3
0	5	1	7	D1	60/60	SX948	4.75V-5.25V	0°C-80°C	3
0	5	1	7	D1	60/60	SX974	5.15V-5.40V	0°C-70°C	3
0	5	1	7	D1	60/60	---	4.75V-5.25V	0°C-80°C	3, 5, 6
0	5	1	7	D1	66/66	Q0626	4.90V-5.40V	0°C-70°C	3
0	5	1	7	D1	66/66	SX950	4.90V-5.40V	0°C-70°C	3
0	5	1	7	D1	66/66	Q0627	5.15V-5.40V	0°C-70°C	3
0	5	1	7	D1	66/66	SX949	5.15V-5.40V	0°C-70°C	3
0	5	1	7	D1	66/66	---	4.90V-5.40V	0°C-70°C	3, 5, 6

NOTES:

1. Non-heat spreader package.
2. These are engineering samples only.
3. Heat spreader package (see Specification Change #2).
4. 66 MHz B1" shipped after work week 34 of 1993 were tested to V_{CC} = 4.90V - 5.40V
5. This is a boxed Pentium processor
6. This part is not marked with a S-Spec number

Summary Table of Changes

The following table indicates the Specification Changes, Errata, Specification Clarifications, or Documentation Changes which apply to the 60- and 66-MHz Pentium ® processor. Intel intends to fix some of the errata in a future stepping of the component, and to account for the other outstanding issues through documentation or specification changes as noted. This table uses the following notations:

CODES USED IN SUMMARY TABLE

X:	Erratum, Specification Change or Clarifica tion that applies to this stepping.
Doc:	Document change or update that will be implemented.
Fix:	This erratum is intended to be fixed in a future stepping of the component.
Fixed:	This erratum has been previously fixed.
NoFix	There are no plans to fix this erratum
(No mark) or (Blank Box):	This erratum is fixed in listed stepping or specification change does not apply to listed stepping.
Shaded:	This item is either new or modified from the previous version of the document.

NO.	B1	C1	D1	Plans	SPECIFICATION CHANGES
1	X	X	X	Doc	STI followed by FP instruction doesn't delay interrupt window
2	X	X	X	Doc	IDT limit violation causes GP fault, not interrupt 8
3	X	X	X	Doc	Leakage current I_{LI} /I_{LO} applies only in the valid logic states
NO.	**B1**	**C1**	**D1**	**Plans**	**ERRATA**
1	X			Fixed	BOFF# hold timing
2	X			Fixed	Incomplete initialization may flush the internal pipeline
3	X			Fixed	IV pin may not be asserted under certain conditions
4	X			Fixed	Testability writes to data TLB may store wrong parity
5	X			Fixed	LRU bits in the data cache TLBs are updated incorrectly
6	X			Fixed	A replacement writeback cycle may invade a locked sequence
7	X			Fixed	RUNBIST instruction generates incorrect BIST signature
8	X	X		Fixed	Data breakpoint mistakenly remembered on a faulty instruction
9	X	X	X	NoFix	RESET affects RUNBIST instruction execution in boundary scan
10	X	X		Fixed	Locked operation during instruction execution tracing may hang the processor

NO.	B1	C1	D1	Plans	ERRATA (Cont'd)
11	X	X		Fixed	Breakpoint or single-step may be missed for one instruction following STI
12	X	X		Fixed	Internal snoop problem due to reflection on address bus
13	X	X		Fixed	Internal parity error on uninitialized data cache entry
14	X	X		Fixed	Missing shutdown after an IERR#
15	X	X		Fixed	Processor core may not serialize on bus idle
16	X	X		Fixed	SMIACT# assertion during replacement writeback cycle
17	X	X	X	NoFix	Overflow undetected on some numbers on FIST
18	X	X	X	NoFix	Six operands result in unexpected FIST operation
19	X	X		Fixed	Snoop with table-walk violation may not invalidate snooped line
20	X	X		Fixed	Slight precision loss for floating-point divides on specific operand pairs
21	X	X	X	NoFix	Power-up BIST failure
22	X	X	X	NoFix	FLUSH#, INIT or Machine Check dropped due to floating-point exception
23	X	X	X	NoFix	Floating-point operations may clear Alignment Check bit
24	X	X	X	NoFix	CMPXCHG8B across page boundary may cause invalid opcode exception
25	X	X	X	NoFix	Single step debug exception breaks out of HALT
26	X	X	X	NoFix	EIP altered after specific FP operations followed by MOV Sreg, Reg
27	X	X	X	NoFix	WRMSR into illegal MSR does not generate GP fault
28	X	X	X	NoFix	Inconsistent data cache state from concurrent snoop and memory write
29	X	X	X	NoFix	Incorrect FIP after RESET
30	X	X	X	NoFix	Second assertion of FLUSH# not ignored
31	X	X	X	NoFix	Segment limit violation by FPU operand may corrupt FPU state
32	X	X	X	NoFix	FP exception inside SMM with pending NMI hangs system
33	X	X	X	NoFix	Incorrect decode of certain 0F instructions
34	X	X	X	NoFix	Data breakpoint deviations
35	X	X	X	NoFix	Event monitor counting discrepancies
36	X	X	X	NoFix	VERR type instructions causing page fault task switch with T bit set may corrupt CS:EIP
37	X	X	X	NoFix	BUSCHK# interrupt has wrong priority

NO.	B1	C1	D1	Plans	ERRATA (Cont'd)
38	X	X	X	NoFix	A fault causing a page fault can cause an instruction to execute twice
39	X	X	X	NoFix	Machine check exception pending, then HLT, can cause skipped or incorrect instruction, or CPU hang
40	X	X	X	NoFix	FBSTP stores BCD operand incorrectly If address wrap & FPU error both occur
41	X	X	X	NoFix	V86 interrupt routine at illegal privilege level can cause spurious pushes to stack
42	X	X	X	NoFix	Corrupted HLT flag can cause skipped or incorrect instruction, or CPU hang
43	X	X	X	NoFix	Benign exceptions can erroneously cause double fault
44	X	X	X	NoFix	Double fault counter may not increment correctly
45	X	X	X	NoFix	Short form of mov EAX/ AX/ AL may not pair
46	X	X	X	NoFix	Turning off paging may result in prefetch to random location
47	X	X	X	NoFix	TRST# not asynchronous
48	X	X	X	NoFix	Asserting TRST# pin or issuing JTAG instructions does not exit TAP Hi-Z state
49	X	X	X	NoFix	ADS# may be delayed after HLDA deassertion
50	X	X	X	NoFix	Stack underflow in IRET gives #GP, not #SS
51	X	X	X	NoFix	Performance monitoring pins PM[1:0] may count the events incorrectly
NO.	**B1**	**C1**	**D1**	**Plans**	**SPECIFICATION CLARIFICATIONS**
1	X	X	X	Doc	Only one SMI# can be latched during SMM
2	X	X	X	Doc	SMIACT# handling during snoop writeback
3	X	X	X	Doc	EADS# recognition
4	X	X	X	Doc	Event monitor counters
5	X	X	X	Doc	KEN# sets cacheability for restarted cycles
6	X	X	X	Doc	NMI signal description
7	X	X	X	Doc	BTB behavior when entering SMM
8	X	X	X	Doc	SMI# activation may cause a nested NMI handling
9	X	X	X	Doc	Exit from shutdown
10	X	X	X	Doc	Code breakpoints set on meaningless prefixes not guaranteed to be recognized
11	X	X	X	Doc	Resume flag should be set by software
12	X	X	X	Doc	Data breakpoints on INS delayed one iteration
13	X	X	X	Doc	CPUID feature flags

NO.	B1	C1	D1	Plans	SPECIFICATION CLARIFICATIONS (Cont'd)
14	X	X	X	Doc	When L1 cache disabled, inquire cycles are blocked
15	X	X	X	Doc	Serializing operation required when one CPU modifies another CPU's code
16	X	X	X	Doc	Cache test registers are modifies during FLUSH#
17	X	X	X	Doc	Extra code break can occur on I/O or HLT instruction if SMI coincides
18	X	X	X	Doc	4-Mbyte page extensions
19	X	X	X	Doc	Recognizing INTR after its INTA cycle
20	X	X	X	Doc	Multiple interrupt delaying instructions may not delay more than one instruction
21	X	X	X	Doc	FYL2XP1 does not generate exceptions for X out of range
22	X	X	X	Doc	Enabling NMI inside SMM
NO.	**B1**	**C1**	**D1**	**Plans**	**DOCUMENTATION CHANGES**
1	X	X	X	Doc	Flatness specification, Volume I, Table 9-12
2	X	X	X	Doc	JMP cannot do a nested task switch, Volume 3, page 13-12
3	X	X	X	Doc	Interrupt sampling window, Volume 3, page 23-39
4	X	X	X	Doc	Errors in the detailed descriptions of FSUB, FSUBR, FDIV, FDIVR and related instructions
5	X	X	X	Doc	PUSHA, PUSHF, POPA & POPF can cause alignment faults
6	X	X	X	Doc	Corrections to BT, BTC, BTR and BTS instruction descriptions
7	X	X	X	Doc	Corrections for description of a stack overflow on tnterrupt to inner privilege case
8	X	X	X	Doc	FSETPM is like NOP, not like FNOP
9	X	X	X	Doc	Corrections in the pseudocode descriptions of the instructions CALL, IRET(D) & RET
10	X	X	X	Doc	Errors in 3 tables of special descriptor types
11	X	X	X	Doc	INTR required to enable NMI inside SMM

SPECIFICATION CHANGES

The Specification Changes listed in this section apply to the 60- and 66-MHz Pentium ® Processor.

No Specification Changes at this time.

1. STI Followed by FP instruction Doesn't Delay Interrupt Window

In the *Pentium® Processor Family Developer's Manual*, Volume 3, page 25-301, the paragraph under "Description" should read as follows:

The STI instruction sets the IF. **STI enables interrupts after the next instruction executed following STI unless the next instruction clears the IF. Assuming interrupts are not already enabled, STI guarantees that interrupts will not be enabled until after the next instruction unless that next instruction is an FP instruction.** If external interrupts are disabled and the STI instruction is followed by the RET instruction (such as at the end of a subroutine), the RET instruction is allowed to execute before external interrupts are recognized. Also, if external interrupts are disabled and the STI instruction is followed by a CLI instruction which clears the IF flag, then external interrupts are not recognized because the CLI instruction clears the IF flag during its execution.

2. IDT Limit Violation Causes GP Fault, Not Interrupt 8

The last sentence in Section 9.3 of the *Pentium® Processor Family Developer's Manual*, Volume 3, says about exception handling in Real Mode: "If an interrupt occurs and its entry in the interrupt table is beyond the limit stored in the IDTR register, a double-fault exception is generated." In fact, in the Pentium processor, there is no difference between Real and Protected Mode when an IDT limit violation occurs. It generates interrupt 13: General Protection fault in both modes.

3. Leakage Current I_{LI}/I_{LO} Applies Only in the Valid Logic States

The leakage currents I_{LI} and I_{LO} only apply in the valid logic states, i.e. $0<V_{IN}<V_{IL}$ and $V_{CC}>V_{IN}>V_{IH}$, instead of $0<V_{IN}<V_{IH}$, which is the current specification. The *Pentium® Processor Family Developer's Manual*, Volume 1, Tables 7-2 and 23-4 will be updated accordingly

ERRATA

1. BOFF# Hold Timing

PROBLEM: Silicon characterization indicates that the processor does not meet the specified hold time, 1.5nS, for the BOFF# signal. Data indicates that a minimum hold time of 2.0nS is required.

IMPLICATION: If a minimum hold time of 2.0nS is not met for the BOFF# input, the processor may drive undefined or incorrect cycles on to the external bus.

WORKAROUND: System must guarantee a minimum hold time of 2.0nS at the BOFF# input to the Pentium processor.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

2. Incomplete Initialization May Flush the Internal Pipeline

PROBLEM: If a memory write occurs before the first branch instruction immediately after RESET, then the internal pipeline may get flushed unexpectedly. Assuming normal distribution of code space, this unexpected flush has the probability of 1 in 2^{26} of occurring.

IMPLICATION: The probability of this unexpected flush occurring is very low. When it happens, its only effect is to flush the internal pipeline and re-fetch the correct opcodes again. The instructions will still be executed correctly.

In the FRC (Functional Redundancy Checking) environment, where the clock-by-clock behavior of the processor needs to be checked deterministically, it may cause the system to report an error.

WORKAROUND: After RESET, ensure that the first write to memory occurs after a branch instruction.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section..

3. IV Pin May Not Be Asserted Under Certain Conditions

PROBLEM: The IV pin is driven active by the Pentium processor to indicate that an instruction in the v-pipe has completed execution. In the following case, the IV pin may not get asserted:

When a mispredicted instruction (pair) reaches the execution stage, it will cause a pipeline flush. If in this clock, a fault is detected on the instruction in the u-pipe, the IV pin will not be asserted for a v-pipe instruction of the next instruction pair which is executed next.

IMPLICATION: The fact that the IV pin is not asserted under certain conditions will affect the reliability of the execution tracing data. It will also affect the performance monitoring event count for instructions executed in the v-pipe.

WORKAROUND: Disabling the v-pipe will allow execution tracing to work properly.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

4. Testability Writes to Data TLB May Store Wrong Parity

PROBLEM: During testability writes to the data TLB, an incorrect tag parity may be computed and stored with the tag address. Subsequently, when this entry is read during a normal (non-testability) cycle, an internal parity error (IERR#) may be generated.

IMPLICATION: The internal parity error may occur only if a non-testability access is made to the same data TLB entry which had been previously written on a testability write. This problem will not show up if the data TLB is flushed after having been used for testability purposes using the TLB test registers.

WORKAROUND: Ensure that the data TLB is flushed after having been used for testability writes and before being used for normal operation.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

5. LRU Bits in the Data Cache TLBs Are Updated Incorrectly

PROBLEM: Due to a circuit problem, the LRU bits for the data cache TLBs are updated incorrectly when both the u and v pipes access the same set. As the TLBs are organized as 4-Way set associative, more specifically this problem occurs when a u-pipe match is found on Way 0 and a v-pipe match is found on Way 1, or the u-pipe match is found on Way 2 and the v-pipe match is found on Way 3 of the same set.

IMPLICATION: The LRU bits are used to handle replacements in the TLB. In this specific case, the pseudo LRU mechanism is not strictly adhered to. Any performance degradation resulting from this is expected to be negligible.

WORKAROUND: None identified at this time.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

6. A Replacement Writeback Cycle May Invade a Locked Sequence

PROBLEM: During a locked read-modify-write (RMW) sequence, if BOFF# or AHOLD is asserted before the write portion of the RMW sequence is completed, then the write cycle will be held off in the internal write buffer until the BOFF# or AHOLD signal is deasserted. During the time that the bus is backed off, if another locked instruction (i.e., with a LOCK prefix) enters the instruction pipeline and initiates a replacement writeback in the data cache, then as soon as the bus is freed, the writeback cycle due to the replacement writeback may be issued in front of the locked write cycle pending in the write buffer. After completion of the writeback cycle, the processor will issue the write cycle to complete the RMW sequence.

IMPLICATION: This problem will only affect those systems which do not expect a replacement writeback cycle in the middle of a locked RMW sequence. Furthermore, the timing of events needed for the above problem to manifest itself has a low probability of occurrence. Note that even though the bus cycles are reordered in this case, the correct bus cycles are run and should not cause any data coherency problems.

WORKAROUND: Do not assert BOFF# or AHOLD in between the read and the write portion of a locked read-modify-write sequence.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

7. RUNBIST Instruction Generates Incorrect BIST Signature

PROBLEM: The Pentium processor TAP instruction, RUNBIST, generates incorrect BIST signature if issued while the processor is in Probe mode. The BIST result is also incorrect if the RUNBIST instruction is issued during a repeated MOVS (move data from string to string) instruction.

IMPLICATION: The BIST may report an incorrect signature indicating self-test failure even though the processor may not be faulty.

WORKAROUND: Do not issue the (TAP) RUNBIST instruction when the processor is in Probe mode, or during a repeated MOVS instruction .

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

8. Data Breakpoint Mistakenly Remembered on a Faulty Instruction

PROBLEM: In the following two cases, an instruction that has data breakpoints enabled and also generates a fault before completing execution may cause an unexpected data breakpoint to occur later in the code or may cause the software to hang:

CASE 1: For the first failing case to occur, the data breakpoint must be set on an instruction that is **not** a simple instruction (NOTE: simple instructions are those that are entirely hardwired and do not require any microcode control) and includes multiple memory references (e.g., a read operation followed by a write operation). In addition, this instruction also generates a fault before completing execution. The data breakpoint must be set on one memory operation (e.g., a data read portion), and the fault occurs on a subsequent memory operation (e.g., a data write portion). If later in the code, a fault during a repeated string operation is encountered, then a spurious debug exception will be reported first, followed by the fault for the string operation. This unexpected debug exception during the string operation may only occur if no other debug exception is taken before the string operation is executed.

CASE 2: For the second failing case, the data breakpoint must be set on the data read of iteration X of a repeated string instruction, and a fault must occur on the data write of the same iteration X. In this case, the Pentium processor takes the debug exception first, before handling the fault. When the faulting iteration is restarted after the debug exception is handled, the data breakpoint is again detected when the fault is encountered, and the processor returns to the debug exception handler. This will cause repeated entries into the debug exception handler for the same iteration. This loop may occur forever, unless the debug exception handler modifies the data breakpoints or the return instruction pointer.

IMPLICATION: In the first case, a data breakpoint may occur when it is not expected. In the second case, a repeated string instruction has a data breakpoint set on the read portion and a fault occurs on the corresponding write. This may cause the software to hang.

WORKAROUND: When setting data breakpoints, be aware of the above failure cases. Note that code and I/O breakpoints can be set properly and are not affected by this erratum.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

9. RESET Affects RUNBIST Instruction Execution in Boundary S can

PROBLEM: The Boundary Scan TAP instruction, RUNBIST, is affected by the assertion of the RESET pin. If the RESET pin is asserted while the processor is executing the TAP instruction, RUNBIST (TAP command field = 0111, and the TAP controller is in the Run-Test-Idle state), then the processor indicates a BIST failure.

IMPLICATION: The IEEE 1149.1-1990 specification states "the design of the component shall ensure that results of the self-tests executed in response to the RUNBIST instruction are not affected by signals received at the non-clock system input pins". The Pentium processor does not meet this requirement.

WORKAROUND: Ensure that the RESET pin is deasserted while the RUNBIST (TAP) instruction is executing.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

10. Locked Operation During Instruction Execution Tracing May Hang the Processor

PROBLEM: During instruction execution tracing (TR12.TR bit set to '1') the processor can internally buffer up to two Branch Trace messages. If there is a possibility of a third Branch Trace message being delivered from the instruction being executed, the machine will stall in order to avoid overwriting either of the two messages that are already buffered. If this instruction is performing a "locked read-modify-write" operation, the processor can hang up due to internal service contention for the bus controller logic.

IMPLICATION: This problem does not affect normal operation (TR12.TR bit is not set). It only affects operation while the instruction execution tracing feature is enabled. A hardware RESET will be required to get the processor out of the deadlock condition if it occurs.

WORKAROUND: Do not enable instruction execution tracing.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

11. Breakpoint or Single-Step May Be Missed for One Instruction Following STI

PROBLEM: If the next instruction following STI is the target of a mispredicted branch, the processor may shut off the interrupt window for one instruction following STI. This will prevent breakpoints, single-step or other external interrupts from being recognized during this time.

IMPLICATION: The processor may not recognize NMI, SMI#, INIT, FLUSH#, BUSCHK#, R/S#, code/data breakpoint and single-step for one instruction after executing STI. This is not a problem unless breakpoints or single stepping is used. The only possible effect is that they may be missed.

WORKAROUND: Do not set a breakpoint on the next sequential instruction after STI. Alternatively, disabling branch prediction will prevent this problem from occurring.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

12. Internal Snoop Problem Due to Reflection on Address Bus

PROBLEM: An internal snoop occurs in the following three cases:

1. An access is made to the code cache, and that access is a miss.
2. An access is made to the data cache, and that access is a miss or a writethrough.
3. There is an access to the page table/directory entries.

In all of these cases, the address used for the internal snoop is obtained from the input buffer of the address I/O buffers inside the chip. If there is signal reflection on the address lines A[31:5] which causes the setup/hold time inside the chip to be violated, then the internal snoop may fail.

When the reflection on the address I/O buffer is above (or below) the trip point of an input buffer (i.e., 1.5V for a TTL input) for a high to low (or low to high) transition, then the internal snoop address will not be valid until after the address reflection falls below (or transitions above) the trip point in the clock ADS# and address is driven. If the reflection causes the wrong snoop address A[31:5] to be latched inside the chip due to setup/hold time violation, then an incorrect internal snoop may occur.

IMPLICATION: When the failure occurs, a wrong cache line may be snooped causing either a line to remain valid when it should not, a valid line to become invalid when it should not, or an invalid line to be snooped.

In the first case, when a valid line should be invalidated and is not, a cache coherency problem between the code and data cache is possible (e.g., self-modifying code). In the second case, when a valid cache line is incorrectly made invalid, an unexpected writeback cycle may occur if the line was in the modified state, and an unexpected bus cycle may occur to re-allocate the cache line. The third case, where an invalid line is snooped, will not cause any detectable failure.

An additional failure mechanism can be seen if address lines A[11:5] are transitioning while being sampled. In this case, the internal snoop may fail, causing a tag parity error during the snoop resulting in an IERR# assertion.

The magnitude of the reflection is dependent upon the I/O buffer vs. transmission line impedance mismatch and the length/layout of trace (transmission line). There is sufficient margin built into the external timing specification for the address bus and I/O buffer models, and it is not expected that any failures will be seen on existing systems. In a lab environment, a failure was forced by adding a 12 inch coaxial cable (with no termination) along with a 330 Ohm pullup resistor on the address line to induce excessive reflection. Removing either the pullup resistor or adding termination to the coaxial cable eliminated the failure.

WORKAROUND: To avoid any internal snoop failures, ensure that the address A[31:5] setup and hold times are not violated at the processor's input due to reflection in the clock in which the processor drives the address bus and ADS# is asserted.

NOTE:

Meeting the setup and hold times at the processor's input is not necessary in the clocks where ADS# is not asserted.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

13. Internal Parity Error on Uninitialized Data Cache Entry

PROBLEM: In the following case, an incorrect internal parity error (IERR#) may be reported due to an uninitialized entry in the data cache during a special qword (64 bit) read. This may occur if the qword read issued to the u-pipe is also followed by a v-pipe read, and the v-pipe read is to a different odd bank and different way than the u-pipe qword read. In addition, the u-pipe address corresponding to this different way must have invalid and uninitialized data. Under these conditions, the processor may check the invalid data for parity errors and incorrectly assert the IERR# pin.

IMPLICATION: After power-on reset, before the data cache is completely initialized, the processor may incorrectly report an IERR# and shutdown.

WORKAROUND: After power-on reset, initialize all entries in the data cache before the cache is enabled. The easiest way to do this is to invoke BIST (built-in self test) after reset. Alternatively, software can initialize the data cache by reading data from memory appropriately, or writing into all its locations through the cache test registers.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

14. Missing Shutdown After an IERR#

PROBLEM: If an internal parity error is reported to the IERR# pin and a mispredicted branch or a trap/fault/interrupt with a higher priority than shutdown occurs, then the processor may not shutdown.

IMPLICATIONS: During the reporting of an internal parity error, the IERR# pin may go active without a processor shutdown. Note that IERR# due to an internal parity error will not occur unless the parity error is induced through parity reversal testing or if the chip is defective.

WORKAROUND: The system can latch an IERR# assertion at the processor clock edge and force a shutdown by asserting NMI or initializing the processor through RESET or INIT.

NOTE: IERR# is a glitch free signal, so no spurious assertions of IERR# will occur.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

15. Processor Core May Not Serialize on Bus Idle

PROBLEM: Under rare circumstances, the processor may not serialize with the bus when the processor core is waiting for the bus to finish pending cycles and BOFF# is asserted. The processor will not reorder bus cycles; it may only start with the next event (fetching and executing subsequent instructions) before waiting for all pending bus cycles to complete. The following cases have been identified that may be affected by this:

SMI# pending	If BOFF# is used to back off a bus cycle while an SMI# is pending, the processor may assert SMIACT# before re-starting the aborted bus cycles.
Serializing instruction	If BOFF# is used to back off a bus cycle due to a serializing instruction, the processor may start executing the next instruction before restarting or completing the previous bus cycle. The processor, however, will not reorder any bus cycles for the new instruction in front of bus cycles for the previous instruction.
Invalidation during cache line fill	If BOFF# is used to back off a cache line fill and BOFF# occurs after the data has been returned to the processor but before the end of the line fill, an invalidation request during this time may result in the cache invalidation to occur before the line fill has completed, This may cause the cache line to remain in a valid state after the invalidation has completed. Note that if the invalidation request comes in via WBINVD or FLUSH#, the line fill would have to be backed off at least twice (or once for INVD) in order for the cache line to remain in a valid state after the invalidation has completed.
OUT instruction	If BOFF# is used to back off a bus cycle due to an OUT instruction, the processor may start executing the next instruction before the bus cycle due to OUT has completed (NOTE: The OUT instruction is similar to the serializing instructions except that it does not stop the prefetch of the subsequent instruction.) The processor, however, will not reorder any bus cycles for the new instruction in front of the OUT bus cycle.

IMPLICATION: This problem has only been observed in internal test vehicles. The events described above have different possible implications as follows:

SMI# pending	The processor may enter SMM before restarting the aborted bus cycle. The SMIACT# assertion may cause the restarted bus cycle to run to SMRAM space.
Serializing Instruction	Since the cycles are not reordered, a system should not encounter any problems unless it depends on the serializing instruction to cause an external event prior to execution of the next instruction.

Invalidation during cache line fill	In a rare instance, a cache line may remain in the valid state (E or S state) after the cache invalidation has completed.
OUT instruction	Since the cycles are not reordered, a system should not encounter any problems unless it depends on the OUT instruction to cause an external event prior to execution of the next instruction. For example, an OUT instruction may be used to assert the A20M# signal prior to the next instruction. In this case, observed code has followed the OUT with an I/O read (IN) to ensure the signal is properly asserted. A second case, could be using an OUT instruction to configure/initialize and interrupt controller and follow it with STI to enable interrupts. Once again no failure would be observed. The controller would respond with the spurious interrupt vector.

WORKAROUND: Restrict the use of BOFF# for the described events. In addition, the SMI# pending event can be eliminated by locating SMRAM so that it does not shadow standard memory and does not require SMIACT# for memory decode. The OUT or serializing instruction events are eliminated if the next instruction does not depend on the result of the event before executing the instruction.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

16. SMIACT# Assertion During Replacement Writeback Cycle

PROBLEM: If a data read cycle triggers a replacement writeback cycle and the SMI# signal is asserted prior to the first BRDY# of the read cycle, the processor may assert the SMIACT# signal prematurely.

Before the processor asserts SMIACT# in response to an SMI# request, it should complete all pending write cycles (including emptying the write buffers). However, if the appropriate conditions occur, the SMIACT# signal may get asserted during the replacement writeback cycle a few clocks after the last BRDY# of the read cycle.

IMPLICATION: If the SMIACT# signal is used by the system logic to decode SMRAM (e.g., SMRAM is located in an area that is overlaid on top of normal cacheable memory space), then the replacement writeback cycle with SMIACT# asserted could occur to SMM space. Systems that locate SMRAM in its own distinct memory space (non-overlaid) should not be affected by this.

WORKAROUND: Asserting FLUSH# simultaneously with SMI# will prevent this problem from occurring. In systems that overlay SMRAM over normal cacheable memory space, this is already a necessary requirement to maintain cache coherency.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

17. Overflow Undetected on Some Numbers on FIST

PROBLEM: On certain large positive input floating-point numbers, and only in two processor rounding modes, the instructions FIST[P] m16int and FIST[P] m32int fail to process integer overflow. As a consequence, the expected exception response for this situation is not correctly provided. Instead, a zero is returned to memory as the result.

The problem occurs only when all of the following conditions are met:

1. The FIST[P] instruction is either a 16- or 32-bit operation; 64-bit operations are unaffected.
2. Either the ‘nearest’ or ‘up’ rounding modes are being used. Both ‘chop’ and ‘down’ rounding modes are unaffected by this erratum.
3. The sign bit of the floating-point operand is positive.

4. The operand is one of a very limited number of operands. The table below lists the operands that are affected. For an operand to be affected, it must have an exponent equal to the value listed and a significand with the most significant bits equal to '1'. Additionally in the 'up' rounding mode, at least one of the remaining lower bits of the significand must be '1'. The Exponent and the two Significand columns describe the affected operands exactly, while the values in the column titled 'Approximate Affected Range' are only for clarity.

FIST[P] Operation	Rounding Mode	Unbiased Exponent	Upper Bits of Significand	Lower Bits of Significand	Approximate Affected Range
16 bit	Up	15	16 MSB's = '1'	At least one '1'	65,535.50 ± 0.50
	Nearest	15	17 MSB's = '1'	don't care	65,535.75 ± 0.25
32 bit	Up	31	32 MSB's = '1'	At least one '1'	4,294,967,295.50 ± 0.50
	Nearest	31	33 MSB's = '1'	don't care	4,294,967,295.75 ± 0.25
64 bit	Problem does not occur				

ACTUAL vs. EXPECTED RESPONSE

Actual Response

When the flaw is encountered, the processor provides the following response:

- A zero is returned as a result. This result gets stored to memory.
- The IE (Invalid Operation) bit in the FPU status word is not set to flag the overflow.
- The C1 (roundup) and PE bits in the FPU status word may be incorrectly updated.
- No event handler is ever invoked.

Expected Response

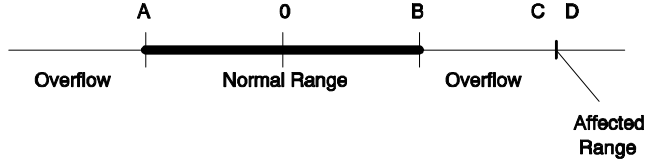
The expected processor response when the invalid operation exception is masked is:

- Return a value 10000....0000 to memory.
- Set the IE bit in the FPU status word.

The expected processor response when the invalid operation exception is unmasked is:

- Do not return a result to memory. Keep the original operand intact on the stack.
- Set the IE bit in the FPU status word.
- Appropriately vector to the user numeric exception handler.

IMPLICATION: An unexpected result is returned when an overflow condition occurs without being processed or flagged. Integer overflow occurs rarely in applications. If overflow does occur, the likelihood of the operand being in the range of affected numbers is even more rare. The range of numbers affected by this erratum is outside that which can be converted to an integer value. The figure below and corresponding table detail the normal range of numbers (between A and B) and the range affected by this erratum (between C and D):



16-bit Operation	A	B	C	D
Round Nearest	[-32,768.5]	(+32,767.5)	[+65,535.5]	(+65,536.0)
Round Up	(-32,769.0)	[+32,767.0]	(+65,535.0)	(+65,536.0)
32-bit Operation	**A**	**B**	**C**	**D**
Round Nearest	[-2,147,483,648.5]	(+2,147,483,647.5)	[+4,294,967,295.5]	(+4,294,967,296.0)
Round Up	(-2,147,483,649.0)	[+2,147,483,647.0]	(+4,294,967,295.0)	(+4,294,967,296.0)

NOTE:
[xxx.x] indicates the endpoint is included in the interval; (xxx.x) indicates the endpoint is **not** included in the interval.

Furthermore, given that the problem cannot occur in the 'chop' rounding mode, and given that the 'chop' rounding mode is the standard rounding mode in ANSI-C and ANSI-FORTRAN 77, it is unlikely that this condition will occur in most applications.

This erratum is not believed to affect application programs in general. Applications will need to handle exceptional behavior and take the appropriate actions to recover from exceptions. In order to do this applications will need to do either range checking prior to conversion or implement explicit exception handling. If an application relies on explicit exception handling the possibility of an error exists. It is, however, believed that applications written in languages that support exception handling will most likely do range checking, thereby allowing the application to be compiled with a no check option for performance reasons when the application has been debugged.

WORKAROUND: Any of three software workarounds will completely avoid occurrence of this erratum:

1. Range checking performed prior to execution of the FIST[P] instruction will avoid the overflow condition from occurring, and is likely to already be implemented.
2. Use the 'chop' or 'down' rounding modes. This erratum will never occur in these modes. By default, both ANSI-C and FORTRAN will use the 'chop' mode.
3. Use the FRNDINT (Round floating-point value to integer) instruction immediately preceding FIST[P]. FRNDINT will always round the value correctly.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

18. Six Operands Result in Unexpected FIST Operation

PROBLEM: For six possible operands and only in two processor rounding modes (up, down), the FIST or FISTP (floating-point to integer conversion and store) instructions return unexpected results to memory. Additionally, incorrect status information is returned under certain conditions in all 4 rounding modes.

The flaw occurs only on certain operands on the instructions FIST[P] m16int, FIST[P] m32int, and FIST[P] m64int. These operands are ±0.0625, ±0.125, and ±0.1875.

The following table details the conditions for the flaw and the results returned. For use of any of the six above-listed operands, refer to the left-hand side of the table in the column for a given combination of sign and rounding mode. The corresponding right-hand side of the table shows the results which will occur for the given conditions. These results include the C1 (Condition Code 1) and PE (Precision Exception) bits and, in two instances, storing of unexpected results.

Operand (any one of)	Rounding Mode	Result Expected / Actual	Status Bits	
			PE Expected / Actual	C1 Expected / Actual
+0.0625	nearest	SAME	1 / Unchanged	SAME
+0.1250	chop	SAME	1 / Unchanged	SAME
+0.1875	down	SAME	1 / Unchanged	SAME
	up	0001 / 0000	1 / Unchanged	1 / 0
–0.0625	nearest	SAME	1 / Unchanged	SAME
–0.1250	chop	SAME	1 / Unchanged	SAME
–0.1875	down	- 0001 / 0000	1 / Unchanged	1 / 0
	up	SAME	1 / Unchanged	SAME

IMPLICATION: An incorrect result (0 instead of +1 or -1) is returned to memory for the six previously listed operands. Incorrect results are returned to memory only when in the 'up' rounding mode with a positive sign or in the 'down' rounding mode with a negative sign. The majority of applications will be unaffected by this erratum since the standard rounding mode in ANSI-C and ANSI-FORTRAN 77 is 'chop', while BASIC and the ISO PASCAL intrinsic ROUND function use 'nearest' rounding mode. In addition, 'nearest' is also the default rounding mode in the processor.

Incorrect status information is also insignificant to most applications. Given that the PE bit in the numeric status register is 'sticky', and that most floating-point instructions will set this bit, PE will most likely have already been set to '1' before execution of the FIST[P] instruction and therefore the PE bit will not be seen to be incorrect. In addition, the unexpected values for the C1 bit are unlikely to be significant for most applications since the only usage of this information is in exception handling.

WORKAROUND: Either of two software workarounds will completely avoid this erratum.

1. Use the FRNDINT (Round floating-point value to integer) instruction immediately preceding FIST[P]. FRNDINT will always round the value correctly.
2. Use of 'nearest' or 'chop' modes will always avoid any incorrect result being stored (although the PE bit may have incorrect values).

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

19. Snoop with Table-Walk Violation May Not Invalidate Snooped Line

PROBLEM: If an internal snoop (as a result of ADS#) or external snoop (EADS#) with invalidation (INV) occurs coincident with a page table walk violation, the snoop may fail to invalidate the entry in the instruction cache. A page table walk violation occurs when the processor speculatively prefetches across a page boundary and this page is not accessible or not present. This violation results in a page fault if this code is executed. A page fault does not occur if the code is not executed.

For this erratum to occur, all the following conditions must be met:

1. A snoop with invalidation is run in the code cache. The snoop may be internal or external.
2. The Pentium processor is performing a page table walk to service an instruction TLB miss.
3. The page table walk results in a violation (this may or may not lead to a page or other fault due to a speculative fetch).
4. The EADS# of the external snoop or ADS# of the table update occur within the window of failure.

   The window is defined by:

   a. 2-4 clocks after BRDY# is returned for the page directory or table read.

   b. 2-n clocks after BRDY# is returned for the page directory or table read if the set address of a buffered write matches that of the instruction cache lookup. “n” is determined by the time to complete two new data write bus cycles from the data cache.

IMPLICATION: This erratum has not been observed on any system. It was found only through investigation of component schematics, and Intel has only duplicated it on a proprietary test system by forcing failure conditions using the internal test registers. The low frequency of occurrence is due to the way most systems operate; DMA devices snoop code 4 bytes at a time so that each line would get snooped and invalidated multiple times.

If this erratum occurs and a line is not invalidated in the instruction cache, then the instruction cache may have a coherency problem. As a result the processor may execute incorrect instructions leading to a GPF or an application error. This erratum affects only self-modifying code and bus masters/DMA devices. Due to necessary conditions, this erratum is expected to have an extremely low frequency of occurrence.

WORKAROUND: There are two workarounds. Because of the rarity of occurrence of this erratum, many OEMs may choose not to implement either workaround.

1. Rewrite the device driver for the DMA devices such that after DMA is complete, the instruction cache is invalidated using the TR5.cntl=11(flush) and CD=0 (code cache) bits.
2. Disable the L1 cache.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

20. Slight Precision Loss for Floating-point Divides on Specific Operand Pairs

PROBLEM: For certain input datum the divide, remaindering, tangent and arctangent floating-point instructions produce results with reduced precision.

The odds of encountering the erratum are highly dependent upon the input data. Statistical characterization yields a probability that about one in nine billion randomly fed operand pairs on the divide instruction will produce results with reduced precision. The statistical fraction of the total input number space prone to failure is 1.14×10^{-10}. The degree of inaccuracy of the result delivered depends upon the input data and upon the instruction involved. On the divide, tangent, and arctangent instructions, the worst-case inaccuracy occurs in the 13th significant binary digit (4th decimal digit). On the remainder instruction, the entire result could be imprecise.

This flaw can occur in all three precision modes (single, double, extended), and is independent of rounding mode. This flaw requires the binary divisor to have any of the following bit patterns (1.0001, 1.0100, 1.0111, 1.1010 or 1.1101) as the most significant bits, followed by at least six binary ones. This condition is necessary but not sufficient for the flaw to occur. The instructions that are affected by this erratum are: FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, FIDIVR, FPREM, FPREM1, FPTAN, FPATAN.

During execution, these instructions use a hardware divider that employs the SRT (Sweeney-Robertson-Tocher) algorithm which relies upon a quotient prediction PLA. Five PLA entries were omitted. As a result of the omission, a divisor/remainder pair that hits one of these missing entries during the lookup phase of the SRT division algorithm will incorrectly predict a intermediate quotient digit value. Subsequently, the iterative algorithm will return a quotient result with reduced precision.

The flaw will **not** occur when a floating-point divide is used to calculate the reciprocal of the input operand in single precision mode, nor will it occur on integer operand pairs that have a value of less than 100,000.

IMPLICATION: For certain input datum, there will be a loss in precision of the result that is produced. The loss in precision can occur between the 13th and 64th significant binary digit in extended precision. On the remainder instruction the entire result could be imprecise.

The occurrence of the anomaly depends upon all of the following:

1. The rate of use of the specific FP instructions in the Pentium processor.
2. The data fed to them.
3. The way in which the results are propagated for further computation by the application.
4. The way in which the final result of the application is interpreted.

Because of the low probability of the occurrence with respect to the input data (one in nine billion random operand pairs), this anomaly is of low significance to users unless they exercise the CPU with a very large number of divides and/or remainder instructions per day, or unless the data fed to the divisor is abnormally high in sensitive bit patterns.

WORKAROUND: Due to the extreme rarity of this flaw, a workaround is not necessary for almost all users. However, Intel is replacing components for end users and OEM's upon request. In addition, a software patch is available for compiler developers. If you are a compiler developer, contact your local Intel representative to obtain this, or download from the World Wide Web Intel support server (www.intel.com).

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

A white paper, *Statistical Analysis of Floating-point Flaw in the Pentium ä Processor,* (Order Number 242481), is available that includes a complete analysis performed by Intel.

21. Power-Up BIST Failure

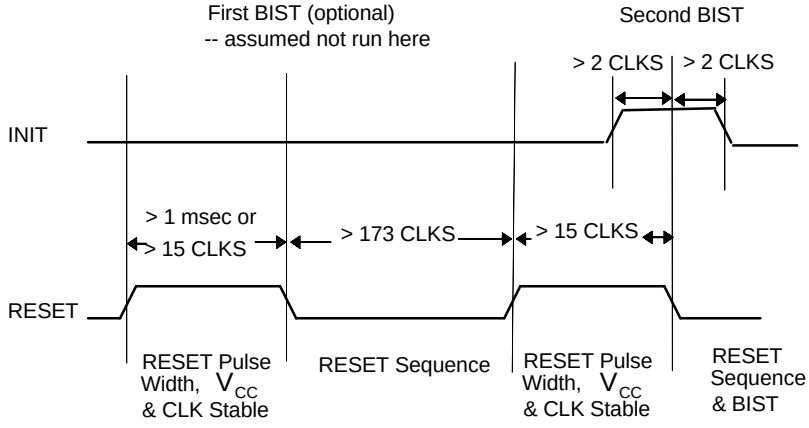
PROBLEM: The BIST (Built In Self Test) feature may fail.

IMPLICATION: Upon completion of BIST, a false failure may be reported. That is, a non-zero value may be returned in the EAX register, even though the part is operating correctly. As such, this erratum has no other product or system implications and is fully functional. To date, all observed false failures have returned a value of 20000H in EAX.

WORKAROUND: The root cause of the failure is not yet fully understood. As a result, workarounds suggested in this document are not guaranteed to fix the problem. To Intel's knowledge, however, no BIST failures have been observed with use of either of the workarounds.

There are two workarounds:

1. Invoke a double RESET — All false BIST failures have occurred during cold RESET. In order to accurately execute BIST, a double RESET sequence may be performed as shown in the figure below:



The first RESET pulse is used to ensure that all logic used in the BIST test is correctly initialized; optionally, INIT may be driven high to enter BIST at the falling edge of the first RESET. In this case the results of the first BIST should be ignored.

After deassertion of the first RESET pulse, a minimum of 173 clocks must elapse before assertion of the second RESET. (Note that if BIST was entered at the end of the first RESET, this 173-clock requirement is satisfied by the time required to complete the first BIST.) During the second RESET, INIT is toggled high in order to run the 'real' BIST. This BIST correctly returns results in EAX.

The RESET and INIT pulses must meet normal specifications; RESET must be active at least 15 clocks, as described in the AC specifications (Tables 7-3 and 7-4 of the *Pentium® Processor Family Developer's Manual*, Volume 1 and in the case of cold RESET must remain asserted for a minimum of 1 millisecond after V_{CC} and CLK have reached their proper AC/DC specifications. INIT must meet setup and hold times around the falling edge of RESET, and asynchronous INIT must be driven high at least 2 clocks before and held for at least 2 clocks after the falling edge of RESET.

2. Use RUNBIST — BIST may be run through the RUNBIST instruction, available through the Test Access Port as documented. In this fashion, BIST will return correct values.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

22. FLUSH#, INIT or Machine Check Dropped Due to Floating-point Exception

PROBLEM: HARDWARE FLUSH and INIT requests and Machine Check exceptions may be dropped if they occur coincidentally with a floating-point exception.

The following conditions are necessary to create this erratum:

1. Two floating-point instructions are paired and immediately followed by an integer instruction.
2. The floating-point instruction in the u-pipe causes a floating-point exception.
3. The FLUSH, INIT, or Machine Check occurs internally on the same clock as the integer instruction.

IMPLICATION: The processor caches will not be flushed, or the INIT request may be dropped.

WORKAROUND: None identified at this time.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

23. Floating-Point Operations May Clear Alignment Check Bit

PROBLEM: The Alignment Check bit (bit 18 in the EFLAGS register) may be inadvertently cleared.

This erratum occurs if a fault occurs during execution of the FNSAVE instruction. After servicing the fault and resuming and completing the FNSAVE, the AC bit will be '0'. Expected operation is that the contents of AC are unchanged.

IMPLICATION: The only known use being made of the AC bit, at this time, is to aid code developers in aligning data structures for performance optimizations. As a result, there are no hardware or system application implications known to Intel at this time. Operating systems and applications will not be affected by this erratum.

WORKAROUND: None identified at this time.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

24. CMPXCHG8B Across Page Boundary May Cause Invalid Opcode Exception

PROBLEM: Use of the new Pentium processor specific CMPXCHG8B instruction may result in an invalid opcode exception.

If the CMPXCHG8B opcode crosses a page boundary such that the first two bytes are located in a page which is present in physical memory and the remaining bytes are located in a non-present page, unexpected program execution results. In this case, the processor generates an Invalid Opcode exception and passes control to exception handler number 6. Normal execution would be for a Page Fault to occur when the non-present page access is attempted, followed by reading in of the requested page from a storage device and completion of the CMPXCHG8B instruction.

IMPLICATION: This erratum only affects existing software which is Pentium processor aware and uses the CMPXCHG8B instruction. Any occurrence would generate an invalid opcode exception and pass control to exception handler 6.

WORKAROUND: Any software which uses Pentium processor specific instructions should ensure that the CMPXCHG8B opcode does not cross a page boundary.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

25. Single Step Debug Exception Breaks Out of HALT

PROBLEM: When Single Stepping is enabled (i.e. the TF flag is set) and the HLT instruction is executed the processor does not stay in the HALT state as it should. Instead, it exits the HALT state and immediately begins servicing the Single Step exception.

IMPLICATION: The behavior described above is identical to Intel486 CPU behavior.

WORKAROUND: None identified at this time.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

26. EIP Altered After Specific FP Operations Followed by MOV Sreg, Reg

PROBLEM: A specific sequence of code may cause corruption of the EIP. This specific code sequence must contain **all** of the following characteristics:

1. Three consecutive FP instructions
2. The third FP instruction must be immediately followed by a MOV Sreg, Reg instruction. (Sreg means one of the 6 segment registers. Note that **all other** ways of changing an Sreg, such as "POP Sreg", "MOV Sreg, Mem", LES, LDS, LSS etc., and far JMPs, CALLs & RETs etc., will **not** cause this problem.)
3. The descriptor selected by the MOV Sreg, Reg must not be available in the on-chip cache of the most recently used descriptors.
4. In addition, there are specific restrictions on each of the 3 consecutive FP instructions:
5. The first instruction in the sequence must be an FP instruction which can be paired with FXCH (FADD, FSUB, FMUL, FDIV, FLD, FCOM, FUCOM, FCHS, FTST & FABS.)
6. The second FP instruction **must be** FXCH, since it's the only FP instruction that can go down the v-pipe (a pair of FP instructions must run together in u and v to allow this problem to occur).
7. The last (third) FP instruction must cause the CPU to test for an unsafe condition. Instructions that compare 2 numbers, and adjust FP flags as the result (FCOM(P), FICOM(P), FUCOM(P) & FTEST) are the usual sources of this test, but only with certain arguments (e.g. NANs and infinities). (In rare situations, FDIV (with a denominator of zero) and FSQRT (negative argument) will cause an unsafe condition test.)

```
Example:    FADD            ; is able to go into the u pipe & pair with FXCH in the v pipe
            FXCH            ; must be in the v pipe for this problem to occur
            FCOMP           ; can cause test for an unsafe condition
            MOV Sreg, Reg   ; selected descriptor must be a "miss' in the segment descriptor cache
```

IMPLICATION: If this erratum occurs, an erroneous value is written into the EIP (instruction pointer), causing unpredictable results. This will most frequently result in an invalid opcode exception.

Intel knows of no existing application or OS code which involves this sequence. First, FXCH is rarely used just before a compare type instruction. Usually a calculation would be done just before compare which would set up one of the numbers to be compared at the FP stack top. Further, to make use of the compare, the FP flags must first be stored in a location where they can be tested, or loaded into EFLAGS for testing, so the next instruction is typically FSTSW, not an Sreg load. Also, Sreg loads are typically not done by application code in a 32-bit environment.

WORKAROUND: Move any instruction between the compare and the Sreg load, such as FSTSW. Even NOP will work.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

27. WRMSR Into Illegal MSR Does Not Generate GP Fault

PROBLEM: The WRMSR and RDMSR instructions allow writing and reading of special MSRs (Model Specific Registers) based on the index number placed in ECX. The architecture was specified to reject access to

illegal MSRs by generating the fault GP(0) if WRMSR or RDMSR is executed with an illegal index. However, negative indices, all of which are illegal, do not trigger GP(0).

IMPLICATION: If RDMSR is used with negative indices, undefined values will be read into EAX. If WRMSR is used with negative indices, undefined processor behavior may result.

WORKAROUND: Do not use illegal indices with WRMSR and RDMSR.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

28. Inconsistent Data Cache State From Concurrent Snoop and Memory Write

PROBLEM: This is a generalization of the Dual Processing specific erratum 5DP. Although this erratum has not been verified in a real Pentium processor-based system without using the DP mode, detailed analysis with the simulation model indicates that it is possible for the erratum to occur as described here. The possible occurrence requires the following conditions:

1. The processor begins a WRITE cycle to a writeback (WB) line in its L1 cache that is in the (S) (shared) state.
2. A non-invalidating snoop using AHOLD/EADS# is generated by another bus master by reading the **same** cache line **before** the completion of the WRITE.
3. There is at least one more cache in the system that holds a copy of the cache line.

When the snoop occurs during this window, the snoop is mishandled and the cache line will transition to the exclusive (E) state instead of the shared (S) state. An additional write to the same cache line by the processor will cause a cache state transition from (E) to modified (M) and will not generate a bus cycle. Since a bus cycle is not generated, other caching agents in the system that hold the cache line in the shared (S) state will not be updated and will contain stale data.

This erratum occurs because the sequence of cycles completed inside the processor is different from the sequence of cycles started on the bus, which is a memory write by the processor followed by a snoop on the same address. Inside the processor, the snoop occurs, and then the memory write completes. This can only happen if the snoop occurs in a window between the ADS# assertion by the processor for its WRITE cycle, and up to 2 CPU clocks after the system assertion of BRDY# at the end of the WRITE cycle.

This erratum can only occur if AHOLD/EADS# is used for snoops; if HOLD or BOFF# are used to force a snoop before the processor WRITE is completed, it will be restarted after the snoop and handled correctly. In addition, this erratum cannot occur on memory updates where the LOCK# signal is asserted.

IMPLICATION: The processor's L1 cache may become incoherent with an external cache. A memory read cycle by an external bus master could read stale data.

WORKAROUND: Designs which do not snoop under AHOLD are not affected. Uniprocessor systems using a lookaside L2 cache, such as those built with either the 82430NX or 82430FX PCIsets, are not affected because a read by an external bus master will always snoop the L1 as well as L2. Intel knows of no uniprocessor implementation which is subject to this erratum. While this erratum is more likely to occur in a multi-processing environment, Intel is not aware of any designs which have demonstrated a susceptibility to this issue.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

29. Incorrect FIP After RESET

PROBLEM: After a RESET, the floating point instruction pointer (FIP) should be initialized to 00000000h. The FIP will instead retain the value it contained prior to the RESET. The FIP gets updated whenever the processor decodes a floating point instruction other than an administrative floating point instruction (FCLEX, FLDCW, FSTCW, FSTSW, FSTSWAX, FSTENV, FLDENV, FSAVE, FRSTOR and FWAIT). If an FSAVE or FSTENV is executed after a RESET and before any non-administrative floating point instruction caused the FIP to be updated, the old value contained in the FIP will be saved. If a non-administrative floating point instruction is the first floating point instruction executed after RESET, the old value in the FIP will be overwritten and any successive FSAVE of FSTENV will save the correct value.

The FIP is used by software exception handlers to determine which floating point instruction caused the exception. The only instructions that can cause an exception are non-administrative floating point instructions, so a non-administrative floating point instruction is usually executed before an FSAVE or FSTENV.

IMPLICATION: If an FSAVE or FSTENV is executed after a RESET and before any non-administrative floating point instruction, the incorrect FIP will be saved.

WORKAROUND: If an FSAVE or FSTENV is executed after a RESET and before a non-administrative floating point instruction is executed, perform an FINIT instruction after RESET as recommended in *Pentium® Processor Family Developer's Manual*, Volume 3, Section 16.2. This will set the FIP to 00000000h. Otherwise, no workaround is required.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

30. Second Assertion of FLUSH# Not Ignored

PROBLEM: If FLUSH# is asserted while the processor is servicing an existing flush request, a second flush operation will follow after the first one completes. Proper operation is for a second assertion of FLUSH# to be ignored between the time the first FLUSH# is asserted and completion of its Flush Acknowledge cycle.

IMPLICATION: A system that asserts FLUSH# during a flush that's already in progress will flush the cache a second time. Flushing the cache again is not necessary and results in a slight performance degradation.

WORKAROUND: For best performance, the system hardware should not assert any subsequent FLUSH# while a flush is already being serviced.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

31. Segment Limit Violation by FPU Operand May Corrupt FPU State

PROBLEM: On the Intel486, 80386 and earlier processors, if the operand of the FSTENV/FLDENV instructions, or the FSAVE/FRSTOR instructions, exceeds a segment limit during execution, the resulting General Protection fault blocks completion of the instruction. (Actually, interrupt #9 is generated in the 80386 and earlier.) This leaves the FPU state itself (with FLDENV, FRSTOR) or its image in memory (with FSTENV, FSAVE) partly updated, thus corrupted, and the instruction generally is non-restartable. In Section 23.3.3.9, Volume 1, the *Pentium® Processor Family Developer's Manual* states that the Pentium processor fixes this problem by starting these instructions with a test read of the first and last bytes of the operand. Thus if there is a segment limit violation, it is triggered before the actual data transfer begins, so partial updates cannot occur.

This improvement works as intended in the large majority of segment limit violations. There is however a special case in which the beginning and end of the FPU operand are within the segment, so the endpoints pass the initial test, but part of the operand exceeds the segment limit. Thus part way through the data

transfer, the limit is violated, the GP fault occurs, and thus the FPU state is corrupted. Note that this is a subset of the cases which will cause the same problem with Intel486 and earlier CPUs, so any code that executes correctly on those CPUs will run correctly on the Pentium processor.

This erratum will happen when both the segment limit and a 16 or 32 bit addressing wrap around boundary falls within the range of the FPU operand, with the segment limit below the wrap boundary. (To use a 16 bit wrap boundary of course, one must be executing code using 16 bit addressing.) The upper endpoint of the FPU operand wraps to near the bottom of the segment, so it passes the initial test. But part way through the data transfer the CPU tries to access memory above the segment limit but below the wrap boundary, causing the GP fault with the FPU state partly copied. This erratum can also happen if the segment limit is at or above a 16 bit addressing wrap boundary, with both straddled by an FPU operand that is **not aligned on an 8 byte boundary.** Test of the upper endpoint wraps and thus passes. But when the instruction is actually transferring data, the misalignment forces the CPU to calculate extra addresses for special bus cycles. This special address calculation does not support the 16 bit wrap, so the GP fault is triggered when the segment limit is crossed.

Note that in Section 23.3.6.2, Volume 1, the *Pentium® Processor Family Developer's Manual* warns in general that the Pentium processor may store only part of operands which generate a memory fault by crossing either a segment or page limit. This erratum is just one case of that general problem, and all cases will be avoided by following the recommended programming practice of never straddling segment or page boundaries with operands. Note also that the handling of operands which straddle such boundaries is processor specific, so code which uses such straddling will behave differently when run on different Intel Architecture processors.

IMPLICATION: This erratum can corrupt that state of the FPU and will cause a GP fault. This generally will require that the task using the FPU be restarted, but it will not cause unflagged errors in results. Code written following Intel recommendations, and any code which runs on the Intel486 (or earlier) CPUs, will not cause this erratum. The case where the Pentium processor will experience this erratum is a small subset of the cases in which the Intel486 (and earlier) CPUs will be corrupted.

WORKAROUNDS:

1. Do not use code in which FPU operands wrap around the top of their segments.
2. If one must use FPU operands which wrap at the top of their segments, make sure that they are aligned on an 8 byte boundary, **and** that the segment limit is not below the 16 or 32 bit wrap boundary.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

32. FP Exception Inside SMM with Pending NMI Hangs System

PROBLEM: If a previous FPU instruction has caused an unmasked exception, and an FP instruction is executed inside SMM with an NMI pending, the system will hang unless the system is both DOS compatible (CR0.NE=0), **and** external interrupts are enabled.

IMPLICATION: For standard PC-AT systems, NMI is typically used (if at all) to indicate a parity error, and the response required is a system reset, to preserve data integrity. So this erratum will only occur when the system has already suffered a parity error; the effect of the erratum is only to force reset inside SMM, instead of after the RSM when the NMI would normally be serviced. In a system where NMI is **not** used for an error that requires shutdown, the workaround should be implemented.

A properly designed system should not experience a hang-up. In such a system the SMM BIOS checks for pending interrupts before issuing an FSAVE or FRSTOR. If an interrupt is pending, the BIOS will exit SMM to handle the interrupt. If an interrupt is not present, the BIOS will disable interrupts (for example, it will disable NMI by writing to the chip set) and only then will issue the FP instruction.

WORKAROUND: If FPU instructions are used in SMM, **and** NMI is used for other than an error that requires shutdown, NMI should be blocked from outside the CPU during SMM.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

33. Incorrect Decode of Certain 0F Instructions

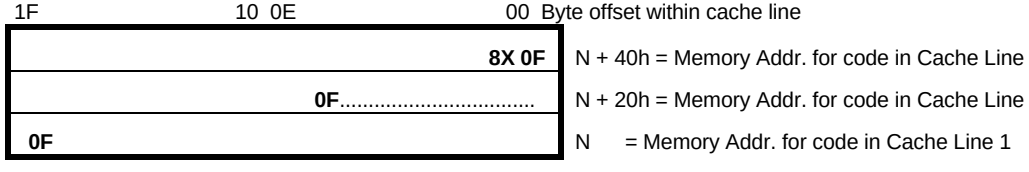
PROBLEM: With a specific arrangement of instructions in memory and certain asynchronous events, the processor may incorrectly decode certain 0F prefixed instructions.

In order for this erratum to occur there must be a very specific arrangement of instructions in memory. In conjunction, these instructions must be resident in the cache and an asynchronous cache line replacement must occur during execution of these instructions. The conditions for this erratum are as follows:

1. There **is a 0F prefix instruction (other than 0F80-0F8F) which can begin in the range of the most significant byte of the first cache line (1F) through the 0E byte of the second cache line** , followed by the rest of the 0F prefixed instruction. (See cache lines 1 & 2 in Fig. 1; the 0F byte is at offset 1F in line 1.)
2. The processor must **execute a branch to the second half of the first cache line shown below (10 - 1F**).
3. There is a **replacement or invalidation of cache line 2** shown below. This replacement or invalidation must complete **in a narrow window** (3 or less CPU clocks) between the decode of the 0F byte from the instruction queue, and the decode of the rest of the instruction.
4. The **third consecutive cache line must contain the bit pattern 0F80 - 0F8F offset by 33 bytes from the 0F byte of the first instruction.** There must be a spacing of exactly 33 bytes between the first and subsequent 0F bytes.
5. All 3 lines must already be resident in the instruction cache and must be from sequential linear memory addresses.

This erratum can occur only if **all** of the above conditions are met. After the first byte (0F) of the opcode is decoded, but before the rest of the instruction can be decoded, the second cache line gets replaced or invalidated. While the processor is waiting for the line containing the second byte of the 0F opcode to be read in again from memory, the 2 bytes offset by 33 bytes from the 0F byte of the stalled instruction are temporarily presented to the instruction decoder. Normally this data would be completely ignored, however if the bit pattern is in the range of 0F80 - 0F8F, then the decode of the 0F byte of the stalled instruction is discarded. (Note that this happens **only** with the 0F prefixed instructions.) When the cache line fill has returned the missing line, the second byte of the stalled instruction is incorrectly interpreted as the start of the instruction.

3 Consecutive L1 Cache Lines, Holding Consecutive Code



IMPLICATION: When this erratum occurs, the processor will execute invalid or erroneous instructions. Depending on software and system configuration, the user will typically see an application error message or system reset.

WORKAROUND: There are currently no workaround identified for existing code.

This erratum may be eliminated in code that is being created or recompiled as follows: The compiler must check for the occurrence of bytes 0F, not 8X, 31 other bytes, 0F and 8X. When such an occurrence is found, a NOP inserted or any other change in spacing will prevent the alignment required for this erratum to occur.

A loader based workaround can also be implemented as follows: At load time, scan the executable for the existence of a 0F instruction (other than 0F8X), check the cache alignment of the 0F instruction and check for the existence of a 0F8X bit pattern 33 bytes beyond the 0F byte of the first instruction. If these conditions are found, the page containing this code sequence can be marked as non-cacheable.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

34. Data Breakpoint Deviations

The following three problems are deviations from the data breakpoint specification when a fault occurs during an FP instruction while the data breakpoint is waiting to be serviced. They all share the same workaround. In the first case the breakpoint **is serviced,** incorrectly, before the actual data access that should trigger it takes place; in the other cases the breakpoint is **not serviced** when it should be.

PROBLEM A: **First**, the debug registers must be set up so that any of the FP instructions which read from memory (except for FRSTOR, FNRSTOR, FLDENV and FNLDENV) will trigger a data breakpoint upon accessing its memory operand. **Second**, there must be an unmasked FP exception pending from a previous FP instruction when the FP load or store instruction enters the execution stage. This so far would cause, per specification, a branch to the FP exception handler. The data breakpoint would not be triggered until/ unless the memory access is made after return from the exception handler. But if **third**, either of the external interrupts INTR or NMI is asserted after the FP instruction enters the execution stage, but before the branch to the FP exception handler occurs, this erratum is generated. In this situation, the processor should branch to the external interrupt handler, but instead it goes to the data breakpoint handler. This is incorrect because the data access that should trigger the breakpoint has not occurred yet.

PROBLEM B: Interrupts are blocked for the instruction after a MOV or POP to SS (to allow a MOV or POP to ESP to complete a stack switch before any interrupt). If the MOV or POP to SS triggers a data breakpoint, it normally is serviced after the following instruction is executed. However, if the following instruction is a FP instruction **and** there is a pending FP error from a preceding FP instruction (even if the error is masked), the delayed data breakpoint is forgotten.

PROBLEM C: If the sequence of memory accesses during execution of FSAVE or FSTENV (or their counterparts FNSAVE and FNSTENV) touches an enabled data breakpoint location, the data breakpoint exception (interrupt 1) occurs at the end of the FP instruction. If however the sequence of memory accesses cross a segment limit after touching the data breakpoint location, the General Protection (GP) fault will occur. This erratum is that as the processor branches to the GP fault handler, the valid data breakpoint is forgotten.

IMPLICATION: This erratum will only be seen by software or hardware developers using the data breakpoint feature of the debug registers. It can cause data breakpoints to be both lost, and asserted prematurely, as long as the contributing FP and GP errors remain uncorrected.

WORKAROUND: Use one of the following:

1. General solution: For problems A & B to occur, an FP error must be caused by a preceding FP instruction, and in problem C, the FP operand causes a segment limit violation. These errors are all indicated in the normal way, despite this erratum. Eliminate them and this erratum disappears, allowing

the data breakpoint debugging to proceed normally. Since debugging is usually done in successive stages, this workaround is usually performed as part of the debugging process.

2. Problem A may also be handled by blocking NMI and INTR during debugging.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

35. Event Monitor Counting Discrepancies

PROBLEM: The Pentium processor contains two registers which can count the occurrence of specific events used to measure and monitor various parameters which contribute to the performance of the processor. There are several conditions where the counters do not operate as specified.

In some cases it is possible for the same instruction to cause the “Breakpoint match” (event 100011, 100100, 100101 or 100110) event counter to be incremented multiple times for the same instruction. Instructions which generate FP exceptions may be stalled and restarted several times causing the counter to be incremented every time the instruction is restarted. In addition, if FLUSH# or STPCLK# is asserted during a matched breakpoint or if a data breakpoint is set on a POP SS instruction, the counter will be incremented twice. The counter will incorrectly not get incremented if the matched instruction generates an exception and the exception handler does an IRET which sets the resume flag. The counter will also not get incremented for a data breakpoint match on a u-pipe instruction if the paired instruction in the v-pipe generates an exception.

The “Hardware interrupts” (event 100111) event counter counts the number of **taken** INTR and NMIs. In the event that both INTR/NMI and a higher priority interrupt are present on the same instruction boundary, the higher priority interrupt correctly gets processed first. However, the counter prematurely counts the INTR/NMI as taken and the count incorrectly gets incremented.

The “Code breakpoint match” (event 100011, 100100, 100101 or 100110) event counter may also fail to be incremented in some cases. If there is a code breakpoint match on an instruction and there is also a single-step or data breakpoint interrupt pending, the code breakpoint match counter will not be incremented.

The “Non-cacheable memory reads” (event 0111110) event counter is defined to count non-cacheable instruction or data memory read bus cycles. Reads to I/O memory space are not supposed to be counted. However, the counter incorrectly gets incremented for reads to I/O memory space.

The “Instructions executed” (event 010110) and “Instructions executed in the v-pipe” (event 010111) event counters are both supposed to be incremented when any exception is recognized. However, if the instruction in the v-pipe generates an exception and a second exception occurs before execution of the first instruction of the exception handler for the first exception, the counter incorrectly does not get incremented for the first exception.

The “Stall on write to an E or M state line” (event 011011) event counter counts the number of clocks the processor is stalled on a data memory write hit to an E or M state line in the internal data cache while either the write buffers are not empty or EWBE# is not asserted. However, it does not count stalls while the write buffers are not empty, it only counts the number of clocks stalled while EWBE# is not asserted.

The “Code TLB miss” (event 001101) and “Data TLB miss” (event 000010) event counters incorrectly get incremented twice if the instruction that misses the code TLB or the data that misses the data TLB also causes an exception.

The “Data read miss” (event 000011) and “Data write miss” (event 000100) event counters incorrectly get incremented twice if the access to the cache is misaligned.

The “Bank conflicts” (event 001010) event counter may be incremented more than once if a v-pipe access takes more than 1 clock to execute.

The “Misaligned data memory or I/O References” (event 001011) incorrectly gets incremented twice if the access was caused by a FST or FSTP instruction.

The “Pipeline flushes” (event 010101) event counter may incorrectly be incremented for some segment descriptor loads and the VERR instruction.

The “Pipeline stalled waiting for data memory read” (event 011010) event counter incorrectly counts a misaligned access as 2 clocks instead of 3 clocks, unless it misses the TLB.

IMPLICATION: The event monitor counters report an inaccurate count for certain events.

WORKAROUND: None identified at this time.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

36. VERR Type Instructions Causing Page Fault Task Switch with T Bit Set May Corrupt CS:EIP

PROBLEM: This erratum can only occur during debugging with the T bit set in the Page Fault Handler’s TSS. It requires the following very specific sequence of events:

1. The descriptor read caused by a VERR type instruction must trigger a page fault. (These instructions are VERR, VERW, LAR and LSL. They each use a selector to access the selected descriptor and perform some checks on it.)
2. The OS must have the page fault handler set up as a separate task, so the page fault causes a task switch.
3. The T bit in the page fault handler’s TSS must be set, which would normally cause a branch to the interrupt 1 (debug exception) handler.
4. The interrupt 1 handler must be in a not present code segment.

The not present code segment should cause a branch to interrupt 11. However, because of this erratum, execution begins at an invalid location selected by the CS from the page fault handler TSS but with the EIP value pointing to the instruction just beyond the VERR type instruction.

IMPLICATION: This erratum will only be seen by software or hardware developers setting the T bit in the page fault handler’s TSS for debugging. It requires that the OS in use has the page fault handler set up as a separate task, which is not done in any standard OS. Even when these conditions are met, the other conditions will cause this erratum to occur only infrequently. When it does occur, the processor will execute invalid or erroneous instructions. Depending on software and system configuration, the developer will typically see an application error message or system reset.

WORKAROUND: If debugging a system in which the page fault handler is a separate task, use one of the following:

1. Do not set the T bit in the page fault handler’s TSS.
2. Ensure that the code segment where the debug exception handler starts is always present in the system memory during debugging.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

37. BUSCHK# Interrupt Has Wrong Priority

PROBLEM: Section 3.4 of the Pentium Processor Data Book lists the priorities of the external interrupts, with BUSCHK# as the highest (if the BUSCHK# interrupt, AKA the machine check exception, is enabled by setting the MCE bit in CR4), and INTR as the lowest. Section 30.5.2 further specifies that STPCLK# is the very lowest priority external interrupt for those Pentium processors provided with it (all CPUs with a core frequency of 75 MHz and above). Consistently with this specification, the CPU blocks all other external interrupts once execution of the BUSCHK# exception handler begins.

However this erratum can change the effective priority for a given assertion of BUSCHK# in the following cases:

CASE 1: An additional external interrupt (except INTR) or a debug exception occurs during a narrow window after the CPU begins to transfer control to the BUSCHK# handler, but before the first instruction of the handler begins execution.

In this case, the other interrupt may be serviced before BUSCHK# is serviced. Thus for other interrupts that occur during this narrow window, BUSCHK# is effectively treated as the next to lowest priority interrupt instead of the highest.

CASE 2: The following conditions must all apply for this case to cause an erratum:

1. A machine check request (INT 18) is pending
2. A FLUSH# or SMI# request is pending
3. A single step or data breakpoint exception (INT 1) is pending
4. The IO_Restart feature is enabled (i.e. TR12 bit 9 is set)

Given the above set of conditions, the interrupt priority logic does not recognize the machine check exception as the highest priority. The processor will not service the FLUSH#/SMI# nor the debug exception (INT 1). Instead, it will generate an illegal opcode exception (INT 6).

IMPLICATION: Most systems do not use BUSCHK# and thus are unaffected by this erratum. For those that do use BUSCHK#, the pin allows the system to signal an unsuccessful completion of a bus cycle. This would only occur in a defective system. (Since BUSCHK# is an “abort” type exception, it cannot be used to handle a problem from which the OS intends to recover; BUSCHK# always requires a system reset.)

Due to this erratum, the BUSCHK# interrupt would either occasionally be displaced by another interrupt (which incorrectly would be serviced first) or an unexpected illegal opcode exception (INT 6) would be generated and the pending machine check would be skipped.

Depending on the system and also the severity of the defect, this delay of the BUSCHK# interrupt (case #1 above) could cause a system hang or reset before a bus cycle error message is displayed by the BUSCHK# interrupt. In case #2 above where an illegal opcode exception (INT 6) is generated instead of the machine check exception, a properly architected INT 6 handler will usually require a reset since this handler was erroneously entered without an illegal opcode. But in any event, the normal outcome of a bus cycle error is to require a system reset, so the practical result of this erratum is just the occasional loss of the proper error message in a defective system.

Another problem can occur due to this erratum if the system is using the SMM I/O instruction restart feature. This problem requires an improbable coincidence: the SMI# signal caused by an I/O restart event must occur essentially simultaneously with BUSCHK#, such that the SMI# interrupt hits the narrow window (as described above) just before the first instruction of the BUSCHK# handler begins execution. This could happen if the same I/O instruction that triggers SMI# (usually to turn back on a device that’s been turned off to save power) also generates a bus failure due to the system suddenly going defective, thus signaling BUSCHK#. The result is that the SMI# interrupt is serviced after the EIP has already been switched to point to the first instruction of the BUSCHK# handler, instead of the I/O instruction. The SMM code that services the I/O restart feature may

well use the image of EIP in the SMRAM state save memory to inspect the I/O instruction, for example to determine what I/O address it's trying to access. In this case, the I/O restart part of SMM code will not find the correct instruction. If it is well written, it will execute RSM when it determines there is no valid I/O access to service. Then execution returns to the BUSCHK# handler with no deleterious impact. But less robust code might turn on the wrong I/O device, hang up, or begin executing from a random location.

WORKAROUND: Do not design a system which relies on BUSCHK# as the highest priority interrupt. If using SMM, do not use BUSCHK# at all.

Note that Case 2 does not apply to B1, C1 or D1 steppings of the 60- and 66-MHz Pentium processors.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

38. A Fault Causing a Page Fault Can Cause an Instruction To Execute Twice

PROBLEM: When the processor encounters an exception while trying to begin the handler for a prior exception, it should be able to handle the two serially (i.e. the second fault is handled and then the faulting instruction is restarted, which causes the first fault again, whose handler now should begin properly); if not, it signals the double-fault exception. A "contributory" exception followed by another contributory exception causes the double-fault, but a contributory exception followed by a page fault are both handled. (See Section 14.9.8 in the *Pentium® Processor Family Developer's Manual*, Volume 3 for the list of contributory exceptions and other details.) This erratum occurs under the following circumstances:

1. One of these three contributory faults: #12 (stack fault), #13 (General Protection), or #17 (alignment check), is caused by an instruction in the v-pipe.
2. Then a page fault occurs before the first instruction of the contributory fault handler is fetched. (This means that a page fault that occurs because the handler starts in a not present page will **not** cause this erratum.)

The result is that execution correctly branches to the page fault handler, but **an incorrect return address is pushed on the stack**: the address of the (immediately preceding) u-pipe instruction, instead of the v-pipe instruction that caused the faults. This causes the u-pipe instruction to be executed an extra time, after the page fault handler is finished.

IMPLICATION: When this erratum occurs, an instruction will be (incorrectly) executed, effectively, twice in a row. For many instructions (e.g. MOV, AND, OR) it will have no effect, but for some instructions it can cause an incorrect answer (e.g. ADD would increase the destination by double the correct amount). However, the page fault (during transfer to the handler for fault #12, #13 or #17) required for this erratum to occur can happen in only three unusual cases:

1. If the alignment check fault handler is placed at privilege level 3, the push of the return address could cause a page fault, thus causing this erratum. (Fault #17 can only be invoked from level 3, so it is legal to have its handler at level 3. Fault 12 and 13 handlers must always be at level 0 since they can be invoked from level 0. The push of a return address on the level 0 stack **must not** cause a page fault, because if the OS allowed that to happen, the push of return address for a regular page fault could cause a second page fault, which causes a double-fault and crashes the OS.)
2. If the descriptor for the fault handler's code segment (in either the GDT or the current LDT) is in a not present page, a page fault occurs which causes this erratum.
3. If the OS has defined the fault handler as a separate task, and a page fault occurs while bringing in the new LDT or initial segments, this erratum will occur.

WORKAROUND: All of the following steps must be taken (but 2 & 3 are part of normal OS strategy, done in order to optimize speed of access to key OS elements, and minimize chances for bugs): 1). If allowing the

alignment fault (#17), place its handler at level 0. 2). Do not allow any of the GDT or current LDT to be "swapped out" during virtual memory management by paging. 3). Do not use a separate task for interrupts 12, 13 or 17.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

39. Machine Check Exception Pending, then HLT, Can Cause Skipped or Incorrect Instruction, or CPU Hang

PROBLEM: This erratum can occur if a machine check exception is pending when the CPU encounters a HLT instruction, or occurs while the CPU is in the HLT state. (E.g. the BUSCHK# error could be caused by executing the previous instruction, or by a code prefetch.) Before checking for pending interrupts, the HLT instruction issues its special bus cycle, and sets an special internal flag to indicate that the CPU is in the HLT state. The machine check exception (MCE) can then be detected, and if it is present the CPU branches to the MCE handler, but **without clearing the special HLT flag - the source of this erratum** . As when other interrupts break into HLT, the return address is that of the next instruction after HLT, so execution continues there after return from the MCE handler.

Except for MCE (and some cases of the debug interrupt), interrupts clear the special HLT flag before executing their handlers. The erratum that causes the MCE logic to not clear the HLT flag in this case can have the following consequences:

1. If NMI, or INTR if enabled, occurs while the HLT flag is set, the CPU logic assumes the instruction immediately following the interrupt is an HLT. So it places the address of the instruction after that on the stack, which means that upon return from the interrupt, the instruction immediately following the interrupt occurrence is skipped over.

2. If FLUSH # is asserted while the HLT flag is set, the CPU flushes the L1 cache and then returns to the HLT state. If the CPU is extracted from the HLT state by NMI or INTR, as in 1), the CPU logic assumes that the current CS:EIP points to an HLT instruction, and pushes the address of the **next** instruction on the stack, so the instruction immediately following the FLUSH# assertion is skipped over.

3. If RSM is executed while the HLT flag is set, again the CPU logic assumes that the CPU must have been interrupted (by SMI, in this case) while in the HLT state. Normally, RSM would cause the CPU to branch back to the instruction that was aborted when entering SMM. But in this case, the CPU branches to the address of the **next** instruction minus one byte. If the aborted instruction is one byte long, this is fine. If it is longer, the CPU executes effectively a random opcode: the last byte of the aborted instruction is interpreted as the first byte of the next instruction.

IMPLICATION: In cases 1 and 2, skipping an instruction can have no noticeable effect, or it could cause some obvious error condition signaled by a system exception, or it could cause an error which is not easily detected. In case 3, executing a random opcode is most likely to cause a system exception like #6 (invalid opcode), but it could cause either of the other results as with cases 1 and 2. Case 2 can also cause an indefinite CPU hang, if the problem occurred when INTR was disabled. However, in order to encounter any of these problems, the system has to continue on with program execution after servicing the MCE. Since the MCE is an abort type exception, the handler for it cannot rely on a valid return address. Also MCE usually signals a serious system reliability problem. For both these reasons, the usual protocol is to require a system reset to terminate the MCE handler. If this usual protocol is followed successfully, it will clear the HLT flag and thus always prevent the above problems. However, there is an additional complication: the cases 1, 2 and 3 above can occur **inside** the MCE handler, possibly preventing its completion.

WORKAROUND: The problems caused by this erratum will be prevented if the Machine Check Exception handler (if invoked) always forces a CPU RESET or INIT (which it should do anyway, for reasons given

above). Since the problems can occur **inside** the MCE handler, the IF should be left zero to prevent INTR from interrupting. Also, NMI, SMI and FLUSH could be blocked inside the MCE handler. The most secure strategy is to force INIT immediately upon entrance to the MCE handler.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

40. FBSTP Stores BCD Operand Incorrectly If Address Wrap & FPU Error Both Occur

PROBLEM: This erratum occurs only if a program does all of the following:

1. The program uses 16 bit addressing inside a USE32 segment (requiring the 67H addressing override prefix) in order to wrap addresses at offsets above 64K back to the bottom of the segment.
2. The 10 byte BCD operand written to memory by the FBSTP instruction must actually straddle the 64K boundary. If all 10 bytes are either above or below 64K, the wrap works normally.
3. The FBSTP instruction whose operand straddles the boundary must also generate an FPU exception. (e.g. Overflow if the operand is too big, or Precision if the operand must be rounded, to fit the BCD format.)

The result is that some of the 10 bytes of the stored BCD number will be located incorrectly if there is an FPU exception. They will be nearby, in the same segment, so no protection violation occurs from this erratum.

The erratum is caused by the fact that when an FPU exception occurs due to FBSTP, a different internal logic sequence is used by the CPU, which incidentally sends the bytes to memory in different groupings. Normally this does not affect the result, but when address wrap occurs in the middle of the operand, the different groupings can cause different destination addresses to be calculated for some bytes.

IMPLICATION: Code which relies on this address wrap with a straddled FBSTP operand may not store the operand correctly if FBSTP also generates an FPU exception. Intel recommends not to straddle segment or addressing boundaries with operands for several reasons, including (see Section 23.3.6.2 of Volume 3 of the *Pentium® Processor Family Developer's Manual*) the chance of losing data if a memory fault interrupts an access to the operand. Also there is variation between generations of Intel processors in how straddled operands are handled.

WORKAROUND: Use one of the following:

1. Do not use 16 bit addressing to cause wraps at 64K inside a USE32 segment.
2. Follow Intel's recommendation and do not straddle an addressing boundary with an operand.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

41. V86 Interrupt Routine at Illegal Privelege Level Can Cau se Spurious Pushes to Stack

PROBLEM: By architectural definition, V86 mode interrupts must be executed at privilege level 0. If the target CPL (Current Privilege Level) in the interrupt gate in the IDT (Interrupt Descriptor Table) and the DPL (Descriptor Privilege Level) of the selected code segment are not 0 when an interrupt occurs in V86 mode, then interrupt 13 (GP fault) occurs. This is described on p. 25-176 in the *Pentium® Processor Family Developer's Manual*, Volume 3. The architectural definition says that execution transfers to the GP fault routine (which must be at level 0) with nothing done at the privilege level (call it level N) where the interrupt service routine is illegally located. In fact (this erratum) the Pentium ® Processor incorrectly pushes the segment registers GS and FS on the stack at level N, before correctly transferring to the GP fault routine at level 0 (and pushing GS and FS again, along with all the rest that's specified for a V86 interrupt).

IMPLICATION: When this erratum occurs, it will place a few additional bytes on the stack at the level (1, 2 or 3) where the interrupt service routine is illegally located. If the stack is full or does not exist, the erratum will cause an unexpected exception. But this problem will have to be fixed during the development process for a V86 mode OS or application, because otherwise the interrupt service routine can never be accessed by V86 code. Thus this erratum can only be seen during the debugging process, and only if the software violates V86 specifications.

WORKAROUND: Place all code for V86 mode interrupt service routines at privilege level 0, per specification.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

42. Corrupted HLT Flag Can Cause Skipped or Incorrect Instruction, or CPU Hang

PROBLEM: The Pentium processor sets an internal HLT flag while in the HLT state. There are some specific instances where this HLT flag can be incorrectly set when the CPU is not in the HLT state.

1. A POP SS which generates a data breakpoint, and is immediately followed by a HLT. Any interrupt which is pending during an instruction which changes the SS, is delayed until after the next instruction (to allow atomic modification of SS:ESP). In this case, the breakpoint is therefore correctly delayed until after the HLT instruction is executed. The processor waits until after the HLT cycle to honor the breakpoint, but in this case when the processors branches to the interrupt 1 handler, it fails to clear the HLT flag. The interrupt 1 handler will return to the instruction following the HLT, and execution will proceed, but with the HLT flag erroneously set.
2. A code breakpoint is placed on a HLT instruction, and an SMI# occurs while processor is in the HLT state (after servicing the code breakpoint). The SMI handler usually chooses to RSM to the HLT instruction, rather than the next one, in order to be transparent to the rest of the system. In this case, on returning from the SMI# handler, the code breakpoint is typically re-triggered (SMI# handler does not typically set the RF flag in the EFLAGS image in the SMM save area). The processor branches to the interrupt 1 handler again, but without clearing the HLT flag. The interrupt 1 handler will return to the instruction following the HLT, and execution will proceed, but with the HLT flag erroneously set.
3. A machine check exception just before, or during, a HLT instruction can leave the HLT flag erroneously set. This is described in detail in Erratum number 52: *Machine Check Exception Pending, then HLT, Can Cause Skipped or Incorrect Instruction, or CPU Hang.*

IMPLICATION: For cases 1 and 2, the CPU will proceed with the HLT flag erroneously set. The following problematic conditions may then occur.

a. If NMI, or INTR if enabled, occurs while the HLT flag is set, the CPU logic assumes the instruction immediately following the interrupt is a HLT. It therefore places the address of the instruction after that on the stack, which means that upon return from the interrupt, the instruction immediately following the interrupt occurrence is skipped over.

b. If FLUSH # is asserted while the HLT flag is set, the CPU flushes the L1 cache and then incorrectly returns to the HLT state, which will hang the system if INTR is blocked (IF = 0) and NMI does not occur. If the CPU is extracted from the HLT state by NMI or INTR, as in a), the CPU logic assumes that the current CS:EIP points to an HLT instruction, and pushes the address of the **next** instruction on the stack, so the instruction immediately following the FLUSH# assertion is skipped over.

c. If RSM is executed while the HLT flag is set, again the CPU logic assumes that the CPU must have been interrupted (by SMI#, in this case) while in the HLT state. Normally, RSM would cause the CPU

to branch back to the instruction that was aborted when entering SMM. But in this case, the CPU branches to the address of the **next** instruction minus one byte. If the aborted instruction is one byte long, this is fine. If it is longer, the CPU executes effectively a random opcode: the last byte of the aborted instruction is interpreted as the first byte of the next instruction.

d. If STPCLK# is asserted to the CPU while the HLT flag is incorrectly set, the CPU will hang such that a CPU reset is required to continue execution.

Cases 1 and 2 of this erratum occur only during code development work, and only with the unusual combination of data breakpoint triggered by POP SS followed by HLT or code breakpoint on HLT followed by SMI#.

WORKAROUND:

CASE 1: Avoid following POP SS with a HLT instruction. POP SS should always be followed by POP ESP anyway, to finish switching stacks without interruption. Following POP SS with HLT instead would normally be a program logic error (the interrupt that breaks the CPU out of HLT will not have a well defined stack to use).

CASE 2: Do not place code breakpoints on HLT instructions. Or: Modify the SMI# handler slightly for debugging purposes by adding instructions to set the RF flag in the EFLAGS image in the SMM save area.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

43. Benign Exceptions Can Erroneously Cause Double Fault

PROBLEM: The double-fault counter can be incorrectly incremented in the following cases:

CASE 1: An instruction generates a benign exception (for example, a FP instruction generates an INT 7) and this instruction causes a segment limit violation (or is paired with a v-pipe instruction which causes a segment limit violation)

CASE 2: A machine check exception (INT 18) is generated.

The initial benign exception will be serviced properly. However, if while trying to begin execution of the benign exception handler, the processor gets an additional contributory exception, the processor will trigger a double fault (and start to service the double fault handler) instead of servicing the new contributory fault. (See Table 14-3 of the *Pentium® Processor Family Developer's Manual* , Volume 3 for a complete list of benign/contributory exceptions).

IMPLICATION: Contributory exceptions generated while servicing benign exceptions can erroneously cause the processor to execute the double fault handler instead of the contributory exception handler.

WORKAROUND: Use benign exception handlers that do not generate additional exceptions. Operating systems designed such that benign exception handlers do not generate additional exceptions will be immune to this erratum. In general, most operating system exception handlers are architected accordingly.

Note that Case 2 does not apply to OverDrive processors.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

44. Double Fault Counter May Not Increment Correctly

PROBLEM: In some cases a double fault exception is not generated when it should have been because the internal double fault counter does not correctly get incremented.

When the processor encounters a contributory exception while attempting to begin execution of the handler for a prior contributory exception (for example, while fetching the interrupt vector from the IDT or accessing the GDT/LDT) it should signal the double fault exception. Due to this erratum, however, the CPU will incorrectly service the new exception instead of going to the double fault handler.

In addition, if the first contributory fault is the result of an instruction executed in the v-pipe, a second contributory fault will cause the processor to push an incorrect EIP onto the stack before entering the second exception handler. Upon completion of the second exception handler, this incorrect EIP gets popped from the stack and the processor resumes execution from the wrong address.

IMPLICATION: The processor could incorrectly service a second contributory fault instead of going to the double fault handler. The resulting system behavior will be operating system dependent. Additionally, an inconsistent EIP may be pushed on to the stack.

Robust operating systems should be immune to this erratum because their exception handlers are designed such that they do not generate additional contributory exceptions. This erratum is only of concern during operating system development and debug.

WORKAROUND: Use contributory exception handlers that do not generate additional contributory exceptions. Operating systems which are designed such that their contributory exception handlers do not generate additional contributory exceptions will not be affected by this erratum. In general, most operating system exception handlers are architected accordingly.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

45. Short Form of MOV EAX/ AX/ AL May Not Pair

PROBLEM: The MOV data instruction forms (excluding MOV using Control, Debug or Segment registers) are intended to be pairable, unless there is a register dependency between the two instructions considered for pairing. (e.g. MOV EAX, mem1 followed by MOV mem2, EAX: here the 2nd instruction cannot be completed until after the first has put the new value in EAX.) This pairing for MOV data is documented by the UV symbol in the Pairing column on page F-15 in the *Pentium® Processor Family Developer's Manual* , Volume 3. This erratum is that the instruction unit under some conditions fails to pair the special short forms of MOV mem, EAX /AX /AL, when no register dependency exists.

The Intel Architecture includes special instructions to MOV EAX /AX /AL to a memory offset (opcodes 0A2H & 0A3H). These instructions don't have a MOD/RM byte (and so are shortened by one byte). Instead, the opcode is followed immediately by 1/2/4 bytes giving the memory offset (displacement). This erratum occurs specifically when a MOV mem, EAX /AX /AL instruction using opcode 0A2H or 0A3H is followed by an instruction that uses the EAX /AX /AL register as a source (register source, or as base or index for the address of a memory source) or a destination register. Then the instruction unit detects a (false) dependency and it doesn't allow pairing. For example, the following two instructions are not paired:

```
A340000000  MOV DWORD PTR 40H, EAX ; memory DS:[40H] <- EAX [goes into u-pipe ]
A160000000  MOV EAX, DWORD PTR 60H ; EAX <- memory DS:[60H] [does NOT go into
v-pipe]
```

IMPLICATION: The only result of this erratum is a very small performance impact due to the non-pairing of the above instructions under the specified conditions. The impact was evaluated for SPECint92* and SPECfp92* and was estimated to be much smaller than run-to-run measurement variations.

WORKAROUND: For the Pentium processor, use the normal MOV instructions (with the normal MOD/RM byte) for EAX /AX /AL instead of the short forms, when writing optimizing compilers and assemblers or hand assembling code for maximum speed. However, as documented above, the performance improvement from avoiding this erratum will be quite small for most programs.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

Note *. Third-party brands and names are the property of their respective owners.

46. Turning Off Paging May Result In Prefetch To Random Location

PROBLEM: When paging is turned off a small window exists where the BTB has not been flushed and a speculative prefetch to a random location may be performed. The *Pentium® Processor Family Developer's Manual*, Volume 3, Section 16.6.2, lists a sequence of 9 steps for switching from protected mode to real-address mode. Listed here is step 1.

1. If paging is enabled, perform the following sequence:
   a. - Transfer control to linear addresses which have an identity mapping (i.e., linear addresses equal physical addresses). Ensure the GDT and IDT are identity mapped.
   b. - Clear the PG bit in the CR0 register.
   c. - Move zero into the CR3 register to flush the TLB.

With paging enabled, linear addresses are mapped to physical addresses using the paging system. In step a above the executing code transfers control to code located where the linear addresses are mapped directly to physical addresses. Step b turns off paging followed by step c which writes zero to CR3 which flushes the TLB (and BTB). A small window exists (after clearing the PG bit and before zeroing CR3) where the BTB has not been flushed, and a BTB hit may cause a prefetch to an unintended physical address.

IMPLICATION: A prefetch to an unintended physical address could potentially cause a problem if this prefetch was to a memory mapped I/O address. If reading a memory mapped I/O address changes the state of a memory mapped I/O device, this unintended access may cause a system problem.

WORKAROUND: Flush the BTB just before turning paging off. This can be done by reading the contents of CR3 and writing it back to CR3 prior to clearing the PG bit in CR0.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

47. TRST# Not Asynchronous

PROBLEM: TRST# is not an asynchronous input as specified in section 5.1.56 of the *Pentium® Processor Family Developer's Manual*, Volume 1.

IMPLICATION: TRST# will not be recognized in cases where it does not overlap a rising TCK# clock edge. This violates the IEEE 1149.1 specification on Boundary Scan.

WORKAROUND: TRST# should be asserted for a minimum of two TCK periods to ensure recognition by the processor.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

48. Asserting TRST# Pin or Issuing JTAG Instructions Does not Exit TAP Hi-Z State

PROBLEM: The *Pentium® Processor Family Developer's Manual,* Volume 1 states that the TAP Hi-Z state can be terminated by resetting the TAP with the TRST# pin, by issuing another TAP instruction, or by entering the Test_Logic_Reset state. However, the indication that the processor has entered the TAP Hi-Z state is maintained until the next RESET. Therefore by using the above methods alone, the TAP Hi-Z state can not be terminated.

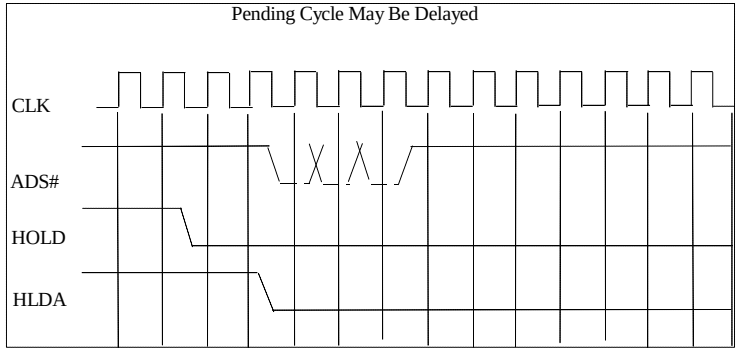
IMPLICATION: When the TAP Hi-Z instruction is enabled and executed, using the methods described in the *Pentium® Processor Family Developer's Manual,* Volume 1 may not terminate the Hi-Z state.

WORKAROUND: To exit TAP Hi-Z state, in addition to the methods described above, the processor needs to be RESET as well.

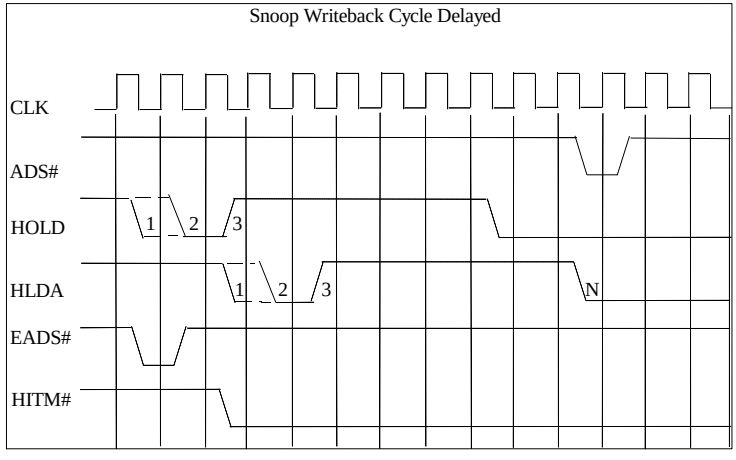
STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

49. ADS# May be Delayed After HLDA Deassertion

PROBLEM: The Pentium processor typically starts a pending bus cycle on the same clock that HLDA is deasserted. However, it may be delayed by as many as two clocks. See the diagram below.



In two cases, for example, if HOLD is deasserted for one clock (i.e., clock 2) or two clocks (i.e., clocks 1 & 2) and then reasserted, the window may not be large enough to start a pending snoop writeback cycle. The writeback cycle may be delayed until the HLDA is deasserted again (i.e., clock N). See diagram below.



IMPLICATION: If the system expects a cycle, for example a writeback cycle, and depends on this cycle to commence within the HLDA deassertion window, then the system may not complete the handshake and cause a hang.

WORKAROUND:

1. Deassert HOLD for at least 3 clocks (i.e., clocks 1, 2, and 3 shown in figure) before reasserting HOLD again. This ensures that the Pentium processor initiates any pending cycles before reasserting HLDA.
2. If the system is waiting for the snoop writeback cycle to commence, for instance if HITM# is asserted, the system should wait for the ADS# before reasserting HOLD.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

50. Stack Underflow in IRET Gives #GP, Not #SS

PROBLEM: The general Intel architecture rule about accessing the stack beyond either its top or bottom is that the stack fault error (#SS) will be generated. However, if during the execution of the IRET instruction there are insufficient bytes for an interlevel IRET to pop from the stack (stack underflow), the general protection (#GP) fault is generated instead of #SS.

IMPLICATIONS: This can only occur if the stack has been modified since the interrupt stored its return address, flags etc. such that there is no longer room on the stack for all of the stored information when IRET tries to access it. This would constitute a serious programming error that would cause problems more obvious than this erratum, and would normally be corrected during debugging. If this erratum did occur during regular execution of a program, the normal O/S response to a task causing either a #GP or #SS exception is to terminate the task, and so this erratum (#GP instead of #SS) would normally have no effect. If however the O/S is to be programmed to try to correct #GP and #SS problems and allow the task to continue execution, the workaround should be used.

WORKAROUND: In order for the O/S code to correctly analyze this case of stack limit violation, the #GP code must include a test for stack underflow when #GP occurs during the IRET instruction.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

51. Performance Monitoring Pins PM[1:0] May Count The Events Incorrectly

PROBLEM: The performance monitoring pins PM[1:0] can be used to indicate externally the status of event counters CTR1 and CTR0. While events are generated at the rate of the CPU clock, the PM[1:0] pins toggle at the rate of the I/O bus clock. However in some cases, the PM[1:0] pins may toggle twice when the event counters increment twice in one I/O clock, while in some cases, the PM[1:0] pins may toggle only once even when the event counters increment twice in two consecutive I/O clocks.

IMPLICATION: The performance monitoring pins PM[1:0] may not be relied upon to reflect the correct number of events that have occurred.

WORKAROUND: None identified at this time.

STATUS: For the steppings affected see the Summary Table of Changes at the beginning of this section.

SPECIFICATION CLARIFICATIONS

The Specification Clarifications listed in this section apply to the 60- and 66-MHz Pentium processor.

1. Only One SMI# Can Be Latched During SMM

Section 20.1.4.2 of Volume 3 of the *Pentium® Processor Family Developer's Manual* correctly states that only one SMI# can be latched by the CPU while it is in SMM (end of 2nd paragraph). However, Section 5.1.50 of Volume 1 of the manual in the SMI# pin definition incorrectly implies by the use of the plural that more than one SMI# request may be held pending during SMM. Thus the following changes will be implemented in the next revision of the Manual:

Section 20.1.4.2 of Volume 3, next to last sentence in the second paragraph, will have the underlined phrase added: "The first SMI# interrupt request that occurs while the processor is in SMM (i.e. after SMIACT# has been asserted) is latched, and serviced when the processor exits SMM with the RSM instruction."

Section 5.1.50 of Volume 1: The second paragraph of the Signal Description, that refers to SMI# request s held pending during SMM, will be replaced with the entire second paragraph of Section 20.1.4.2 of Volume 3.

2. SMIACT# Handling During Snoop Writeback

In Section 14.2.1 in the *Pentium® Processor Family Developer's Manual*, Volume 1, the following text should be added as item 5 in the SRAM Interface Section.

Inquire cycles are permitted during SMM, but it is the responsibility of the system to ensure that any snoop writeback completes to the correct memory space, irrespective of the state of the SMIACT# pin. Specifically, if SMM is overlaid, and SMM space is non cacheable, then any snoop writeback cycle occurring during SMM must complete to system memory, even though SMIACT# will remain active.

If an inquire cycle occurs after assertion of SMI# to the processor, but before SMIACT# is returned, note that SMIACT# could be returned at any point during the snoop writeback cycle. Depending on the timing of SMI# and the inquire cycle, SMIACT# could change states during the writeback cycle. Again, it is the responsibility of the system, if it supports snooping during SMM, to ensure that the snoop writeback cycle completes to the correct memory space, irrespective of the state of the SMIACT# pin.

3. EADS# Recognition

The following text replaces the "When Sampled" section of the *Pentium® Processor Family Developer's Manual*, Volume 1, section 5.1.19.

When sampled

To guarantee recognition, EADS# should be asserted two clocks after an assertion of AHOLD or BOFF#, or one clock after an assertion of HLDA. In addition, the Pentium processor will ignore an assertion of EADS# if the processor is driving the address bus, or if HITM# is active, or in the clock after ADS# or EADS# is asserted.

4. Event Monitor Counters

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-25, the last sentence of the FLOPs event description should be changed as follows: The integer multiply instructions and other instructions which use the FP arithmetic circuitry will be counted.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-25, it states that the performance monitor counters can be programmed to count the number of breakpoint matches on DR0, DR1, DR2 or DR3 registers and the count will be incremented in the event of a breakpoint match whether or not breakpoints are enabled. It should be noted that when breakpoints are not enabled, for code breakpoints the counter will only be incremented if a code breakpoint match was on an instruction that was executed in the u pipe. Code breakpoint matches are not checked for instructions executed in the v pipe when breakpoints are not enabled and thus will not cause the counter to increment.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-24, the "Instructions Executed" event counter (event 010110) is used to count the total number of instructions executed. It should be noted that for any Repeat prefixed string instruction (ex., REP MOVS, REPNE SCAS) the "Instructions Executed" counter will only increment once despite the fact that the repeat loop executes the same instruction multiple times until the loop criteria is satisfied. This applies to all the Repeat string instruction prefixes (i.e., REP, REPE, REPZ, REPNE, and REPNZ). It should also be noted that the "Instructions Executed" counter will only increment once per each HALT instruction executed regardless of how many cycles the processor remains in the HALT state. In addition, it should be noted that there are additional events which can also trigger the "Instructions Executed" (event 010110) event counter. All hardware and software interrupts and exceptions will also cause the count to be incremented.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-24, there are additional events which trigger the "Instructions Executed in the v-pipe" (event 010111) event counter which should be noted. All hardware and software interrupts and exceptions will also cause the count to be incremented.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-24, it should be noted that the "Pipeline Flushes" (event 010101) event counter will not be incremented for serializing instructions (serializing instructions cause the prefetch queue to be flushed but will not trigger the Pipeline Flushed event counter). Software interrupts will also not cause this event counter to be incremented since they do not flush the pipeline.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-22, the "Data Read Miss" (event 000011) event counter counts the number of memory data reads that miss the internal data cache whether or not the access is cacheable or non-cacheable. It should be noted that additional reads to the same cache line after the first BRDY# of the burst linefill is returned but before the final (fourth) BRDY# has been returned, **will not** cause this event counter to be incremented additional times.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-23, the description for the "Code Read" (event 001100), "Code TLB Miss" (event 001101) and "Code Cache Miss" (event 001110) event counters should say that individual eight byte non-cacheable instruction reads **are** counted.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-23, it should be noted that the VERR an VERW instructions are treated as branches and will cause the "Branches" (event 010010) event counter to be incremented.

In Volume 1 of the *Pentium® Processor Family Developer's Manual*, Section 33.4.5, page 33-23, it should be noted that the "Taken branch or BTB hit" (event 010100) event counter also is incremented for jumps, calls, returns, software interrupts, interrupt returns, serializing instructions, some segment descriptor loads, hardware interrupts, programmatic exceptions that invoke a trap or fault handler and the VERR and VERW instructions.

5. KEN# Sets Cacheability For Restarted Cycles

The *Pentium® Processor Family Developer's Manual*, Volume 1, Section 5.1.36, the fourth paragraph under "Signal Description" should read as follows:

Once KEN# is sampled active for a cycle, the cacheability cannot be changed. If a cycle is **restarted for any reason** after the cacheability of the cycle has been determined, the same cacheability attribute on KEN# must be returned to the processor when the cycle is redriven."

6. NMI Signal Description

The *Pentium® Processor Family Developer's Manual*, Volume 1, Section 5.1.40, the second paragraph under "Signal Description" should read as follows:

If NMI is asserted during the execution of the NMI service routine it will remain pending and will be recognized after the IRET is executed by the NMI service routine. At most, one assertion of NMI will be held pending. **If NMI is reasserted prior to the NMI service routine entry it will be ignored.**

7. BTB Behavior When Entering SMM

Section 3-2, page 3-6 of Volume 1 of the *Pentium® Processor Family Developer's Manual*, states that the Pentium processor speculatively runs code fetches corresponding to instructions executed sometime in the past. Such code fetch cycles are run based on past execution history, regardless of whether the instructions retrieved are relevant to the currently executing instruction sequence.

One effect of the branch prediction mechanism is that the Pentium processor may run code fetch bus cycles to retrieve instructions which are never executed. Although the opcodes retrieved are discarded, the system must complete the code fetch bus cycle by returning BRDY#. It is particularly important that the system return BRDY# for all code fetch cycles regardless of the address.

The following text should be added to Section 3-2 of Volume 1 of the *Pentium® Processor Family Developer's Manual*, and also to Section 14.2.1 of Volume 1 as item number 5 on page 14-5.

It should also be noted that upon entering SMM, the branch target buffer (BTB) is not flushed and thus is possible to get a speculative prefetch to an address outside of SMRAM address space due to branch predictions based on code executed prior to entering SMM. If this occurs, the system must still return BRDY# for each code fetch cycle.

8. SMI# Activation May Cause a Nested NMI Handling

In the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 20.1.4.4, the following note should be added just before the last paragraph.

During NMI interrupt handling NMI interrupts are disabled. NMI interrupts are serviced and completed with IRET one at a time. When the processor enters SMM from the NMI interrupt handler, the processor saves the SMRAM State Save Map (e.g. contents of status registers) but does not save the attribute to keep NMI interrupts disabled. Potentially a NMI could be latched (while in SMM or upon exit) and serviced upon exit of SMM even though the previous NMI handler has still not completed. One or more NMI's could be nested in the first NMI handler. The interrupt handler should take this into consideration.

9. Exit from Shutdown

A note in the *Pentium® Processor Family Developer's Manual*, Volume 1, Section 6.3.8 and Volume 3, Section 14.9.8 should be added as follows:

Upon entering shutdown, the state of the CPU is unpredictable and may or may not be recoverable. RESET or INIT should be asserted to return the system to a known state. Although some system operations (i.e.

FLUSH#, SMI# and R/S#) are generally recognized during shutdown, these operations may not complete successfully in some cases once shutdown is entered.

Furthermore, upon exit from shutdown with NMI (to the NMI handler), the SS, ESP and EIP of the task that was executing when shutdown occurred can no longer be relied upon to be valid. Therefore, using NMI to exit shutdown should be used only for debugging purposes and not to resume execution from where shutdown occurred.

If invoking NMI to exit shutdown, use a task gate rather than an interrupt or trap gate in slot 2 of the IDT. One of the conditions that may lead to shutdown is an attempt to use an invalid stack segment selector (SS). In this case, if the NMI successfully exits shutdown, it will immediately re-enter shutdown because it has no valid stack on which to push the return address. It is more robust to vector NMI through a task gate rather than an interrupt gate in the IDT, since the task descriptor allocates a new stack for the NMI handler context.

10. Code Breakpoints Set on Meaningless Prefixes Not Guaranteed to be Recognized

The following should be added to the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 17.3.1.1 (Instruction-Breakpoint Fault).

Code breakpoints set on meaningless instruction prefixes (a prefix which has no logical meaning for that instruction, e.g. a segment override prefix on an instruction that does not access memory) are not guaranteed to be recognized.

Code breakpoints should be set on the instruction opcode, not on a meaningless prefix.

In the *Pentium® Processor Family Developer's Manual*, Volume 3, Sections 3-4 (Instruction Format) and 25.2 (Instruction Format), after "For each instruction one prefix may be used from each group. The effect of redundant prefixes (more than on prefix from a group) is undefined and may vary from processor to processor." The following should be added:

Some prefixes when attached to specific instructions have no logical meaning (e.g. a segment override prefix on an instruction that does not access memory). The effect of attaching meaningless prefixes to instructions is undefined and may vary from processor to processor.

11. Resume Flag Should Be Set by Software

The lead-in sentences and first bullet of section 14.3.3 in the *Pentium® Processor Family Developer's Manual*, Volume 3 should be replaced with the following:

The RF (Resume Flag) in the EFLAGS register should be used during debugging to avoid servicing an instruction breakpoint fault multiple times. RF works as follows:

- The debug handler (interrupt #1) should set the RF bit in the EFLAGS image on the stack whenever it is servicing an instruction breakpoint fault (rather than a data breakpoint trap), and the breakpoint is being left in place. If this is not done, the CPU will return to execute the instruction, fault on the breakpoint again to interrupt #1, and so on.

The following should be added as fifth and sixth bullets:

- If a fault type breakpoint coincides with another fault (the instruction accesses a not present page, violates a general protection rule, etc.) one spurious repetition of the breakpoint will occur after the second fault is handled, even though the debug handler sets RF. As an optional debugging convenience, to avoid this occasional confusion, all interrupt handlers that could interact during debugging in this way can be modified by having them also set the RF bit in the EFLAGS image on their stack.

- The CPU, in branching to fault handlers under some circumstances, will set the RF bit in the EFLAGS image on the stack by hardware action. Exactly when the CPU does this is implementation specific and should not be relied upon by software. No problem is caused by setting this bit again if it is already set.

12. Data Breakpoints on INS Delayed One Iteration

The *Pentium® Processor Family Developer's Manual,* Volume 3, last paragraph and sentence of section 17.3.1.2 states, "Repeated INS and OUTS instructions generate a memory breakpoint debug exception trap after the iteration in which the memory address breakpoint location is accessed."

The sentence should read, "Repeated OUTS instructions generate a memory breakpoint debug exception trap after the iteration in which the memory address breakpoint location is accessed. Repeated INS instructions generate the memory breakpoint debug exception trap one iteration later".

13. CPUID Feature Flags

The *Pentium® Processor Family Developer's Manual* , Volume 3, page 25-74 defines the feature flags returned in EDX after a CPUID instruction is executed. Some additional bits are defined as follows:

EDX[0:0]	FPU on chip
EDX[1:1]	Virtual Mode Extension
EDX[2:2]	Debugging Extension
EDX[3:3]	Page Size Extension
EDX[4:4]	Time Stamp Counter
EDX[5:5]	Model Specific Registers
EDX[6:6]	Physical Address Extension
EDX[7:7]	Machine Check Exception
EDX[8:8]	CMPXCHG8 Instruction Supported
EDX[9:9]	On-Chip APIC Hardware Enabled
EDX[10:11]	Reserved
EDX[12:12]	Memory Type Range Register
EDX[13:13]	Page Global Enable
EDX[14:14]	Machine Check Architecture
EDX[15:15]	CMOV Instruction Supported
EDX[16:22]	Reserved
EDX[23:23]	MMX™ Technology Supported
EDX[24:31]	Reserved

Additional information on the function of these bits can be found in Application Note AP-485, Intel Processor Identification With the CPUID Instruction order number 241618.

14. When L1 Cache Disabled, Inquire Cycles Are Blocked

The last line in Table 18-2 in the *Pentium® Processor Family Developer's Manual,* Volume 3 presently reads "Invalidation is inhibited". This is part of the description of L1 cache behavior when it is "disabled" by setting CR0 bits CD = NW = 1. This line will be clarified to read "Inquire cycles (triggered by EADS# active) and resulting invalidation and any APCHK# assertions are inhibited."

15. Serializing Operation Required When One CPU Modifies Another CPU's Code

A new subsection, 19.2.1, will be added to the *Pentium® Processor Family Developer's Manual,* Volume 3, titled *Processor Modifying Another Processor's Code* **,** and it will be referenced in the current subsection 18.2.3 on self modifying code.

A particular problem in memory access ordering occurs in a multiprocessing system if one processor (CPU1) modifies the code of another (CPU2). This obviously requires a semaphore check by CPU2 before executing in the area being modified, to assure that CPU1 is finished with the changes before CPU2 begins executing the changed code. In addition, it is necessary for CPU2 to execute a serializing operation after the semaphore allows access but before the modified code is executed. This is needed because the external snoops into CPU2 caused by the code modification by CPU1 will invalidate any matching lines in CPU2's code cache, but not in its prefetch buffers or execution pipeline. Note that this is different from the situation described in section 18.2.3 on self-modifying code. When the CPU modifies its own code, the prefetch buffers and pipeline as well as the code cache are checked and invalidated if necessary.

16. Cache Test Registers Are Modified During FLUSH#

The following Note will be added to the *Pentium® Processor Family Developer's Manual,* Volume 1, Section 33.2.1.1.1 on **Direct Cache Access**:

Note: When the FLUSH# pin is asserted, it is treated as an interrupt, and when serviced at the next instruction boundary, it causes writeback of the data cache and then invalidation of the internal caches. The cache test registers TR2, TR3, TR4 and TR5 are used in this process, and thus their values after FLUSH# has been serviced are unpredictable. Therefore FLUSH# should not be asserted while code is being executed which uses these test registers.

17. Extra Code Break Can Occur on I/O or HLT Instruction if SMI Coincides

If a code breakpoint is set on an I/O instruction, as usual the breakpoint will be taken before the I/O instruction is executed. If the I/O instruction is also used as part of an I/O restart protocol, I/O restart is enabled, and executing the instruction triggers SMI, RSM from the SMI handler will return to the start of the I/O instruction, and the code breakpoint will be taken again before the I/O instruction is executed a second time.

Similarly, if a code breakpoint is set on an HLT instruction, the breakpoint will be taken before the processor enters the HLT state. If SMI occurs during this state, and the SMI handler chooses to RSM to the HLT instruction (the usual choice, for SMI to be transparent), the code breakpoint will be taken again before the HLT state is re-entered. In this case, other problems can occur, because an internal HLT flag remains set incorrectly. These problems are documented in Erratum # 55, case 2.

This information will be added to the end of Section 17.3.1.1, on "Instruction-Breakpoint Faults", in the *Pentium® Processor Family Developer's Manual,* Volume 3.

18. 4-Mbyte Page Extensions

The pentium processor allows 4-Mbyte page extensions which are enabled by setting CR4.PSE bit (bit 4) to 1. For more information on 4-Mbyte pages, please refer to the Volume 3, Section 3.6 of the *Pentium® Pro Family Developer's Manual* (order number 242692).

19. Recognizing INTR After Its INTA Cycle

The *Pentium® Processor Family Developer's Manual,* Volume 1, Section 5.1.32 and 21.1.21, states "To guarantee recognition if INTR is asserted asynchronously it must have been deasserted for a minimum of 2 clocks before being returned active to the Pentium processor". However, to avoid an inadvertent second recognition of INTR, a clarification is being added to specify when INTR should be deasserted.

If INTR remains asserted beyond completion of its second INTA cycle and the IF flag is set, INTR can be recognized again. In real mode for example, if the handler enables interrupts prior to INTR being deasserted, a second INTR will be recognized. Interrupts may be re-enabled quickly if the STI instruction is executed early in the handler or the handler is very short, such that IF is set soon after the completion of the second INTA cycle. If trap gates are used in protected mode, INTR may be recognized even earlier because the IF flag remains set (and does not get cleared).

To clarify this, the *Pentium® Processor Family Developer's Manual,* will add a NOTE, "To ensure INTR is not recognized inadvertently a second time, deassert INTR no later than the BRDY# of the second INTA cycle and no earlier than the BRDY# of the first INTA cycle.

20. Multiple Interrupt Delaying Instructions May Not Delay More Than One Instruction

In the next publication of the *Intel Architecture Developer's Manual,* Volume 2: The Instruction Set, the following footnote will be added to the descriptions of the STI and the MOV SS,.. instructions as below. This footnote will be referenced from the part of the description of the instruction where the feature that delays interrupts for one more instruction is explained.

Footnote:
In a sequence of instructions which individually delay interrupts past the following instruction, only the first in the sequence is guaranteed to have this effect. Thus in the sequence STI/ MOV SS/ MOV ESP/ interrupts may be recognized before MOV ESP executes.

21. FYL2XP1 Does Not Generate Exceptions for X Out of Range

The FYL2XP1 instruction is intended to be used only for taking the log of numbers very close to one, to provide improved accuracy. For X values outside of the FYL2XP1 instruction's valid range, the FYL2X instruction should be used instead. The present documentation of what happens when X is outside of the FYL2XP1 instruction's valid range is inconsistent. For FYL2XP1, out of range behavior will be replaced by "If the ST operand is outside of its acceptable range, the result is undefined, and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST is out of range, but this behavior is implementation specific and not guaranteed." The information on pages 7-15 and 25-161 of the *Pentium® Processor Family Developer's Manual,* Volume *3* will be clarified.

22. Enabling NMI Inside SMM

Page 20-11 of the *Pentium® Processor Family Developer's Manual* Volume. 3 states "Although NMI requests are blocked when the CPU enters SMM, they may be enabled through software by invoking a dummy interrupt and vectoring to an Interrupt Service Routine." This will be changed to: "Although NMI requests are blocked when the CPU enters SMM, they may be enabled by first enabling interrupts through INTR by setting the IF flag, and then by triggering INTR. Also, for the Pentium processor, exceptions that invoke a trap or fault handler will enable NMI inside of SMM. This behavior of exceptions enabling NMI within SMM is not part of the Intel Architecture, and is implementation specific".

DOCUMENTATION CHANGES

The Documentation Changes listed in this section apply to the *Pentium® Processor at iComp index 510\60 MHz and the Pentium Processor at iComp index 567\66 MHz* datasheet (Order number 241595) .

1. Flatness Specification, Volume 1, Table 9-2

The flatness specification F of 0.127 millimeters is incorrectly located in the Min column in Table 9-2 of Volume 1 of the *Pentium® Processor Family Developer's Manual* . It should be moved from the Min column to the Max column.

2. JMP Cannot Do a Nested Task Switch, Volume 3, Page 13-12

In the *Pentium® Processor Family Developer's Manual* , Volume 3, Section 13.6, the sentence "When an interrupt, exception, jump, or call causes a task switch..." incorrectly includes the **jump** in the list of actions that can cause a **nested** task switch. The word "jump" will be removed from the sentence. The Table 13-2 correctly shows the effects of task switches via jumps vs. Task switches via CALL's or interrupts, on the NT flag and the Link field of the TSS.

3. Interrupt Sampling Window, Volume 3, Page 23-39

In the *Pentium® Processor Family Developer's Manual* , Volume 3, Section 23.3.7 the first sentence of the second paragraph "The Pentium processor ... asserts the FERR# pin." Should be replaced with the following:

The Pentium processor and the Intel486 processor implement the "No-Wait" Floating-Point instructions (See Section 6.3.7) in the DOS-Compatibility mode (CR0.NE = 0) in the following manner:

In the event of a pending unmasked numeric exception, the "No-Wait" class of instructions asserts the FERR# pin.

4. Errors in the Detailed Descriptions of FSUB, FSUBR, FDIV, FDIVR and Related Instructions

The FSUB/FSUBP/FISUB instructions (page 25-145 in the 1995 *Pentium® Processor Family Developer's Manual* , Volume 3) and the FSUBR/FSUBRP/FISUBR instructions (page 25-147 in Volume 3 of the Manual) each have three erroneous entries in their tables of eight specific forms (the fourth, fifth and sixth forms are wrong in both cases). Also there are errors under the **Operation** and **Description** headings. Further, in the individual descriptions of the specific forms, three different styles are used. In the next revision, all descriptions will be changed to match the style used here for the corrections.

FSUB/FSUBP/FISUB	**Instruction**	**Description**
	FSUB ST(i), ST	ST(i) ← (ST subtracted from ST(i)).
	FSUBP ST(i), ST	ST(i) <-- (ST subtracted from ST(i)); POP ST.
	FSUBP	ST(1) <-- (ST subtracted from ST(1)); POP ST.

Operation DEST <-- DEST - Other Operand <<2nd line is O.K.>>

Description The subtraction instructions subtract the other operand from the destination (which is the leftmost operand in the forms with two explicit operands; otherwise it is always the stack top, ST) and return the difference to the destination.

FSUBR/FSUBRP/FISUBR	Instruction	Description
	FSUBR ST(i), ST	ST(i) <-- (ST(i) subtracted from ST).
	FSUBRP ST(i), ST	ST(i) <-- (ST(i) subtracted from ST); POP ST.
	FSUBRP	ST(1) <-- (ST(1) subtracted from ST); POP ST.

Operation DEST <-- Other Operand - DEST <<2nd line is O.K.>>

Description The reverse subtraction instructions subtract the destination (which is the leftmost operand in the forms with two explicit operands; otherwise it is always the stack top, ST) from the other operand and return the difference to the destination.

The FDIV/FDIVP/FIDIV instructions (page 25-98 in the 1995 *Pentium® Processor Family Developer's Manual*, Volume 3) and the FDIVR/FDIVRP/FIDIVR instructions (page 25-100 in Volume 3 of the Manual) have errors under the **Operation** and **Description** headings. Also, in the table of eight specific forms for FDIVR/FDIVRP/FIDIVR the fourth, fifth and sixth forms have a meaningless (but probably not misleading) word "Divide" in front of their otherwise equation style descriptions; it should be ignored. Further, among the total of 16 individual descriptions of the specific forms, three different styles are used. In the next revision, all descriptions will be changed to match the style given here: *DEST* <-- (*OP1* divided by *OP2*)

FDIV/FDIVP/FIDIV

Operation <<The abbreviation for Source, which should be SRC, is misspelled twice as SCR>>

Description The division instructions divide the destination (which is the leftmost operand in the forms with two explicit operands; otherwise it is always the stack top, ST) by the other operand and return the quotient to the destination.

FDIVR/FDIVRP/FIDIVR

Operation <<O.K. as is>>

Description The reverse division instructions divide the other operand by the destination (which is the leftmost operand in the forms with two explicit operands; otherwise it is always the stack top, ST) and return the quotient to the destination.

5. PUSHA, PUSHF, POPA & POPF Can Cause Alignment Faults

The instructions **POPA/POPAD** (*Pentium® Processor Family Developer's Manual*, Volume 3, page 25-250), **POPF/POPFD** (page 25-252), **PUSHA/PUSHAD** (page 25-256) and **PUSHF/PUSHFD** (page 25-258) can cause the alignment check fault (#AC; interrupt 17) when it is enabled, just as PUSH and POP can, but that specification has been omitted from the previous instructions' descriptions. For each of these four instruction descriptions, the following should be added:

To the sections called **Protected Mode Exceptions,** add at the end: #AC for unaligned memory reference if the current privilege level is 3 (the word operand form is unaligned unless the address is divisible by 2; the dword operand form (used by the mnemonic ending in D) is unaligned unless the address is divisible by 4).

To the sections called **Virtual 8086 Mode Exceptions,** add at the end: #AC for unaligned memory reference if the current privilege level is 3 (alignment is the same as for protected mode).

6. Corrections to BT, BTC, BTR and BTS Instruction Descriptions

The specific descriptions of the BTU, BTU, BTU and BCE instructions, on pages 25-42, 25-44, 25-46 and 25-48 of the *Pentium® Processor Family Developer's Manual* , Volume 3, are incomplete. The **Flags Affected** section for all of these bit test instruction descriptions should include the statement that the flags OF, SF, ZF, AF and PF are all left undefined after execution of one of these instructions. These flag effects are already correctly specified on page B1 of the *Pentium® Processor Family Developer's Manual* , Volume 3.

Also, on p. 25-44 for the BTC instruction, the current sentence in the Flags Affected section should be changed from "The CF flag contains the complement of the selected bit" to "The CF flag contains the value of the selected bit", meaning the value before the BTC execution.

7. Corrections for Description of a Stack Overflow On Interrupt to Inner Privilege Case

In the *Pentium® Processor Family Developer's Manual,* Volume 3 starting on p. 25-173, the Pentium processor operation is described in pseudo-code for the INT/INTO instructions, and also all the external interrupts and internal exceptions which go through the interrupt vector/descriptor table. On p. 25-175 an IF/FI clause inside the "INT-TO-INNER-PRIV PROC" reads as follows:

IF 32-bit gate
THEN new stack must have room for 20 bytes else #SS(0)
ELSE new stack must have room for 10 bytes else #SS(0)
FI

The error code of zero given for stack overflows is incorrect. The correct error code is the SS selector, so the two entries "#SS(0)" will be changed to "#SS(SS selector)". A new clause will also be added immediately after the corrected clause above describing the checking done for the additional stack space required if the interrupt has an error code. The corrected text is as follows:

IF 32-bit gate
THEN new stack must have room for 20 bytes else #SS(SS selector)
ELSE new stack must have room for 10 bytes else #SS(SS selector)
IF interrupt was caused by exception with error code
THEN Stack limit must allow push of 4 more bytes if 32-bit gate, or 2 more bytes if 16-bit gate
ELSE #SS(SS selector);
FI;

These clauses apply in protected mode when an interrupt tries to transfer control through a trap or interrupt gate in the Interrupt Descriptor Table to an inner privilege level service routine. If the 20 or 24 bytes (or 10 or 12 bytes in 16-bit mode) that must be stored on the new stack would cause it to overflow, the protection system triggers #SS (interrupt 12) instead, which allows the O.S. to allocate more stack space.

8. FSETPM Is Like NOP, Not Like FNOP

In the *Pentium® Processor Family Developer's Manual,* Volume 3, page 23-37 in the Section 23.3.4 on Instructions, the 80287 instruction FSETPM is described as being equivalent to an FNOP when executed in the Intel387 math coprocessor and the Intel486 and Pentium processors. In fact, FSETPM is treated as a NOP in these processors, as is correctly explained (along with the difference between FNOP and NOP) on the next page in sec. 23.3.6. "FNOP" will be changed to "NOP" in the FSETPM description.

9. Corrections in the Pseudocode Descriptions of the Instructions CALL, IRET(D) & RET

In the *Pentium® Processor Family Developer's Manual*, Volume 3 the pseudocode descriptions of the detailed operation of the instructions CALL, IRET(D) and RET have the following errors:

For the CALL instruction, on p. 25-52, in both the section labeled "CONFORMING-CODE-SEGMENT" and the section labeled "NONCONFORMING-CODE-SEGMENT", just after the line "Stack must be big enough for return address ELSE #SS(0)" the following line should be added: " **Push return address onto stack**".

Also on p. 25-52, in the section labeled "CALL-GATE", just after the line "DPL of selected descriptor must be ≤ CPL ELSE #GP(code segment selector)" the following line should be added: " **Segment must be present ELSE #NP(code segment selector)**".

For the IRET(D) instruction, on p. 25-187, in the section labeled "RETURN-OUTER-LEVEL", the line "SS must be present, else #NP(SS Selector)" should be corrected to read " **Stack segment must be present, ELSE #SS(SS Selector)".** (A not present stack segment always causes #SS, while other not present segments cause #NP. The capitalization of ELSE and spelling out of SS are just minor style improvements.)

Also on p. 25-187, 5 lines below the correction just given, the line "Load EFLAGS with values aT (eSP+8);" should be followed by the line "**IncremeNT eSP by 12**". Similarly, 3 lines below that line, the line "Load EFLAGS with values aT (eSP+4);" should be followed by the line " **IncremeNT eSP by 6**". These steps are needed, even thouGH eSP is overwritten in the *next* step, because the new ss:eSP are read from the stack using the current ss:eSP values.

For the RET instruction, on p. 25-273, in the fifteenth line from the top of the page, "Segment must be present, else #NP(selector)" should be corrected to read " **Stack segment must be present, ELSE #SS(SS Selector)".**

10. Errors in 3 Tables of Special Descriptor Types

In the *Pentium® Processor Family Developer's Manual,* Volume 3 on page 25-199 and on page 25-222, in the descriptions of the **LAR** and **LSL** instructions respectively, tables are given of the special segment and gate descriptor types and names, with indication of which ones are valid with the given instruction. The same two pairs of descriptor types are interchanged in these two tables. Descriptor type 6 is the 16-bit interrupt gate, not trap gate, and type 7 is the 16-bit trap gate, not interrupt gate. Similarly, descriptor type 0Eh is the 32-bit interrupt gate, and 0Fh is the 32-bit trap gate. Table 12-1 gives a completely correct listing of the special descriptor types, but in the same chapter, Table 12-3 (page 12-22) incorrectly indicates that the 16-bit gates are not valid for the **LSL** instruction (this table *does* have the correct types for the interrupt and trap gates that it shows).

11. INTR Required to Enable NMI Inside SMM

Page 20-11 of the *Pentium® Processor Family Developer's Manual* Volume 3 states "Although NMI requests are blocked when the CPU enters SMM, they may be enabled through software by invoking a dummy interrupt and vectoring to an Interrupt Service Routine." This will be changed to: "Although NMI requests are blocked when the CPU enters SMM, they may be enabled by first enabling interrupts through INTR by setting the IF flag, and then by triggering INTR." NMI cannot be enabled inside of SMM by a software interrupt (INT n), or by executing the IRET instruction.