80386-B1
STEPPING INFORMATION

REVISION: DECEMBER 17, 1986

This document contains specification changes, errata, and design notes.

Specification changes listed are permanent; the 80386 data sheet will be modified to incorporate the changes.

The errata items described herein will be corrected on future steppings of the 80386.

NOTES:

80386-B1 component identifier readable in DH after reset: 03H
80386-B1 revision identifier readable in DL after reset: 03H

At this time, B1 stepping parts are identified with one of the marks shown below:

ii ii A80386-16 ii S40344 ii (FPO number) ii m c i '85 '86	ii ii A80386-20 ii S40362 ii (FPO number) ii m c i '85 '86
ii ii A80386 ES B1 ii ii ii m c i '85 '86	

Specification Changes

The specification changes numbered 1 through 4 for previous versions of the 80386 have now been incorporated in the latest version of the 80386 datasheet, version -002. The remaining specification changes, here, are now renumbered beginning with 1.

1. NT Bit and IOPL Bits in Real Mode

   The NT bit and IOPL bits of the FLAGS register can be set in Real Mode of the 80386. The exact behavior of these bits in 80386 Real Mode was not previously documented. Note that in 80286 Real Mode, these bits can not be set (they always remain 0 in 80286 Real Mode).

2. Coprocessor Data Pointer Stored by FSAVE/FSTENV Instructions is Undefined after Non-memory Instructions

   The contents of the operand address field resulting from a FSTENV or FSAVE are undefined if the preceeding coprocessor arithmetic instruction did not have a memory operand. The exact contents of the operand address field in this case was specified previously. This now confirms that the operand address field is undefined in that case.

3. Bit String Insert and Extract Instructions Removed

   Since the 80386 has unique and powerful 64-bit Double Shift instructions, and fast multi-bit shift and rotate instructions, the "Bit String Insert" and "Bit String Extract" instructions were removed. The insert/extract complex instructions did not provide an additional benefit that fully justified including them in 80386 silicon and all future compatible processors. A review concluded that the 80386 user obtains full performance in bit string manipulations using other powerful instructions such as 64-bit Double Shift, and other multi-bit shift/rotate instructions. These instructions support extremely fast manipulation of general unaligned bit strings of any length, by processing them in 32-bit chunks.

4. ERROR# Input Difference - Effect on PC/AT Compatible Coprocessor Connection

   On the 80386, latching the level of BUSY# when ERROR# becomes active will cause FST and FSTP instructions which get errors to hang the 80386. On the 80286, latching BUSY# when ERROR# becomes active (as performed in the PC/AT) did not cause any problems.

   <u>Implications</u>: The PC/AT uses a non-standard scheme to report 80287 errors to the 80286 (a scheme compatible with the non-standard scheme used to report 8087 errors to the 8088 in the original PC). The scheme used in the PC/AT works because a separate data channel is used by the 80286 to communicate with the 80287. However, the 80386 communicates with the math coprocessor using microcode loops. Therefore, PC/AT-compatible 80386 systems using an 80287 or 80387 numerics coprocessor must carefully follow the recommendation below when replicating the PC/AT's non-standard method of reporting coprocessor errors.

   <u>How to properly replicate the PC/AT coprocessor error-reporting scheme</u>: A workaround exists when replicating the PC/AT coprocessor interface in 80386-based systems. Note that this workaround needs to be incorporated for the non-standard PC/AT scheme; the standard recommended 80386/80387 connection functions properly and the 80386 implementation will not be altered. To understand the workaround, let us review the AT interface. In the PC/AT, the ERROR# input to the 80286 is tied inactive (high) permanently. The ERROR# output of the 80287 is tied to an interrupt port (IRQ13). This interrupt replaces error signalling via the 80286's ERROR# input. To guarantee (in the case of an 80287 error) that INTR 13 will be serviced prior to the execution of any further 80287 instructions, an

edge-triggered flip-flop latches BUSY# using ERROR# as a clock. The output of this latch is ORed with the BUSY# output of the 80287 and drives the BUSY# input of the 80286. This PC/AT scheme effectively delays BUSY# deactivation at the 80286 whenever an 80287 ERROR# is signalled. Since the 80286 BUSY# input remains active, the 80286 INTR 13 handler is guaranteed to execute before any other 80287 instructions may begin. The INTR 13 handler clears the BUSY# latch (via a write to a special I/O port) thus re-allowing execution of 80287 instructions. The INTR 13 handler then branches to the NMI handler, where the user-defined numerics error handler resides in PC-compatible systems.

The use of an interrupt guarantees that an error from a coprocessor instruction will be detected. Latching BUSY# guarantees that any coprocessor instruction (except FINIT, FSETPM, FCLEX) following the instruction that raised the error will not be executed before the NMI handler is executed. This approximates the way the 8087-8088 error-reporting interface works in the original PC.

The 80386 can use a PC/AT-compatible interface to communicate with an 80287/80387 provided that while BUSY# is latched active, the 80386 PEREQ input is also activated, and the 80287/80387 coprocessor is disabled. An 80287 can be disabled using either NPS1# or NPS2. An 80387 should be disabled using its STEN input (do not use the 80387 NPS1# or NPS2 inputs to disable the 80387 in this case). Note that while PEREQ is artificially activated as described above, the 80386 may issue I/O read cycles for the coprocessor. It is permissible for the 80386 data pins to float throughout such I/O read cycles.

5. Read Cycles Require Valid Data Bus Levels

The 80386 requires that all data bus pins be at a valid logic state (high or low) at the end of each read cycle, when READY# is asserted. The system MUST be designed to meet this requirement. Therefore, do NOT allow any data lines to be floating when the read cycle completes. NOTE: The I/O read cycles just mentioned in the previous item, item 4, are free from this requirement.

<u>Implications</u>: If the device being read is a 32-bit device, such as a 32-bit memory, the system should present 32-bits of data to the 80386 even if not all of the 80386 byte enables are asserted.

If the device being read is a 16-bit or an 8-bit device, however, pullup resistors can be used to guarantee valid logic levels on the upper data lines, which otherwise would be floating. Note that bus cycles to 16-bit and 8-bit devices typically include several wait states, but always calculate the effects of R-C time constants to ensure the pullups will drive proper logic levels onto the bus within the time required.

6. I/O Permission Bitmap Must Reside Within TSS Offset 0FFFFh

The 80386 requires that the entire I/O permission bitmap (including the terminating byte of "0FFh"), which is part of an 80386 TSS, begin at an offset no larger than 0DFFFh. This guarantees the entire bitmap (up to 8 kilobytes + 1 terminator byte of 0FFh) will reside at TSS offsets of 0FFFFh or less. Therefore, the pointer within a 386 TSS called Bit_Map_Offset(15:0) must contain a value of 0DFFFh or less under all conditions, even when you intend the Bit_Map_Offset to point beyond the limit of the TSS itself.

7. BS16# Must Not Be Asserted During Pipelined Bus Cycles

In datasheet figures 5-16, 5-17, 5-19, and 5-22, the bus size 16 (BS16#) input is shown as "don't care" during T2P and T21 in pipelined bus cycles. This is incorrect. In these figures, BS16# should be high during states T2P and T21. That is, once address pipelining has been requested by asserting next address (NA#), BS16# must be negated for the remainder of the current bus cycle.

Implications: Don't assert BS16# if NA# has already been sampled asserted in the current bus cycle.

Errata

1. Opcode Field Incorrect for FSAVE and FSTENV

   Problem: If an FSAVE or an FSTENV is executed in REAL mode or in VIRTUAL 8086 mode, the opcode field stored in memory is incorrect if it should have referred to a coprocessor instruction which transfers either two bytes or ten bytes from memory to the coprocessor. The instruction and operand linear address fields are correctly stored. Note that coprocessor error-handling routines are the only routines possibly affected. Also note that the problem does not occur in PROTECTED mode programs (since no opcode is saved by FSAVE or FSTENV in that case).

   Workaround: In REAL mode or in VIRTUAL 8086 mode, the instruction linear address field can be used to read the opcode from memory. Note that the two bytes fetched need to be swapped to yield the image that FSAVE and FSTENV normally stores. The following is a possible fixup sequence.

```
FSTENV  [BX]        ;save environment
MOV     CX,[BX+8]   ;get linear IP<19:16>
AND     CX,0F000h   ;treat it like a selector
MOV     SI,[BX+6]   ;get linear IP<15:0>
MOV     FS,CX       ;establish addressability
MOV     CX,FS:[SI]  ;get raw opcode value
XCHG    CH,CL       ;swap bytes and
AND     CX,7FFh     ; mask out top bits
;CX now has the opcode -- store back if needed
MOV     SI,[BX+8]   ;get opcode word
AND     SI,0F800h   ;mask out the bad
OR      SI,CX       ;mask in the good
MOV     [BX+8],SI   ;and store back
```

   The opcode saved within the FSAVE FSTENV operand is in the following format:

```
              10 9 8        7 6 5 4 3 2 1 0
              |______|      |_______________|
         lower three bits     mod r/m byte
          of ESC byte
```

2. FSAVE, FRESTOR, FSTENV and FLDENV Anomolies with Paging

   Problem: If either of the last two bytes of an FSAVE or an FSTENV operand are for any reason not writeable, or either of the last two bytes of an FRESTOR or FLDENV are for any reason not readable, the instruction is not restartable.

   Workaround: This does not not affect typical systems with reasonably-assigned page access rights. In an obscure situation where this problem arises, a workaround is to avoid having the operand of these instructions span a page boundary. This can be accomplished by aligning these operands on any 128-byte boundary.

3. Wraparound Coprocessor Operands

   Problem: This can affect only situations where a coprocessor operand straddles the limit of a segment of maximum size (i.e. 0FFFFh for a 16-bit segment or 0FFFFFFFFh for a 32-bit segment) or within 108 bytes of maximum size, thus wrapping around to offset 0 of the segment. Since a wraparound situation is very abnormal for a compiler or programmer to create, this does not affect a typical system.

   Formally, the 80386 architecture does not permit an operand (coprocessor operands included) to wrap around the end of a segment. If the user issues such an instruction nonetheless in a Protected Mode system, and the operand starts and ends in valid, present pages of a segment, BUT spans through an invalid or inaccessible page, the coprocessor may be put in an indeterminate state. In such cases, an FCLEX or FINIT instruction needs to be executed before any other coprocessor instruction is issued.

Workaround: In Real Mode, this is not a problem since protection is not enabled. In Protected Mode, this problem is avoided simply by not creating coprocessor operands which wrap around the end of the segment, or by aligning the base of all segments on page boundaries.

4. IRET to TSS with Limit too Small

Problem: If an IRET performs a task switch to a TSS of proper descriptor type but invalid (too small) limit, a Double Fault (exception 8) will result instead of a Invalid TSS Fault (exception 10) as should result. Furthermore, if the Double Fault entry in the IDT is a trap gate, a shutdown results. In a related topic, if the TSS Fault entry in the IDT is invalid for any reason (e.g. bad AR byte), then instead of a Double Fault (exception 8), a shutdown results.

Workaround: A working system, one that creates TSS segments of adequate size to hold the processor state (44 bytes for the TSS of a 16-bit task, 104 bytes for the TSS of a 32-bit task), will not encounter any problems here. A working system should also provide a valid gate (interrupt, trap, or task gate) in the IDT for exception 8.

5. Single-Stepping First Iteration of REP MOVS

Problem: If a REPeated MOVS instruction is executed when single-stepping is enabled (TF = 1 in EFLAGS register), a single-step trap (exception 1) is taken every two move steps, but should occur each move step. Also, if a data breakpoint is hit during a odd iteration number of REP MOVS, the data breakpoint trap is not taken until after the next even-numbered iteration. If the REP MOVS ends with an odd number of iterations, and single-stepping or data breakpoints are enabled, then a single-step trap or data breakpoint trap on the final iteration will properly occur after the final, odd-numbered iteration.

Workaround: When using the Trap Flag or data breakpoints with a debugger utility, this minor variation of REP MOVS must be accepted, unless an effort is made to have the debugger emulate the REP MOVS rather than actually execute it.

6. Task Switch to Virtual 8086 Mode Doesn't Update Prefetch Limit

Problem: When a task switch to Virtual 8086 Mode is performed, the prefetch limit is not updated to become 0FFFFh, but instead remains at its previous value.

Workaround: Use the IRET instruction to transfer to Virtual 8086 Mode. Using IRET is the preferred method for most instances, especially when the master OS dispatches a Virtual 8086 Mode program, because IRET can cause the transition without a task switch.

7. Wrong Register Size for String Instructions in Mixed 16/32-bit Addressing Systems

Problem: If certain string and loop instructions are followed by instructions that either:

1) use a different address size (that is, if either the string instruction or the following instruction uses an address size prefix), or

2) reference the stack (e.g. PUSH/POP/CALL/RET) and the "B" bit in the SS descriptor is different from the address size used by the string instructions,

then one or more of (E)CX, (E)SI, or (E)DI is not updated properly. The size of the register (16 vs. 32) is taken from the following instruction rather than from the string or loop instruction. This could result in updating only the lower 16 bits of a 32-bit register, or in updating all 32 bits of a register being used as 16 bits. The instructions and registers affected by this are listed below:

Instruction	Register(s)
MOVS	(E)DI
REP MOVS	(E)SI
STOS	(E)DI
INS	(E)DI
REP INS	(E)CX

Workaround: No workaround is necessary if all code is 16-bit or if all code is 32-bit. The problem only occurs if instructions with different address sizes are mixed together, or if a code segment of one size is used with a stack segment of the other size.

In a system which mixes address sizes, add a NOP after each of the above instructions and ensure that the NOP has the same address size as the string/loop (i.e., if the string/loop instruction includes an address prefix, place the same address prefix before the NOP; conversely, if the string/loop instruction does not have an address prefix, do not place a prefix before the NOP).

8. FAR Jump Located Near Page Boundary in Virtual 8086 Mode Paged Systems

Problem: In Virtual 8086 Mode, if a direct FAR jump (opcode EAh) instruction is located at the end of a page (or within 16 bytes of the end), and the next page is not cached in the TLB, the prefetcher limit is not set by the FAR jump instruction to the "end" on the new code segment, but rather is left at the "end" of the old code segment. This can allow execution beyond the end of the new segment without triggering a segment limit violation. Or it can result in a spurious GP fault if the old and new segments overlap, and a prefetch occurs beyond the limit of the old segment.

Note that the prefetch limit is checked on the <u>linear address</u>, not by comparing IP to 0FFFFh.

Workaround: All existing 8086 programs use only 16-bit addressing, and thus will not execute code at offsets greater than 0FFFFh from the code segment base. Thus the lack of detection of walking off the end of a code segment should not impact working 8086 programs.

A workaround to the spurious GP fault, if it occurs, is to simply IRET back to the faulting instruction, since the IRET will correctly set the prefetch limit. If the fault handler has control of the single-step function, a very simple workaround is to attempt to single-step the faulting instruction. If the single-step succeeded, the handler could clear the fault, turn off single-stepping, and IRET. If a GP fault occurred attempting to single-step the instruction, a "real" GP fault is the cause.

If the fault handler cannot access the single-stepping function, it still can check for "real" GP faults which must be emulated by the master OS, for example, I/O instructions that need to be emulated, CLI/STI instructions that must be emulated, etc. If none of these faults are recognized, the fault handler can assume this errata caused the GP fault and simply IRET back to the instruction.

9. Page Fault Error Code on Stack Not Reliable

Problem: When a Page Fault (exception 14) occurs, the 3 defined bits in the error code may be unreliable if a certain sequence of prefetch is happening at the same time.

Workaround: Although the page fault error code pushed onto the page fault handler's stack can be unreliable, as described, the page fault linear address stored in register CR2 is always correct. The page fault handler should refer to the page fault linear address in CR2 to access the corresponding page table entry and thereby determine whether the page fault was due to a page "not present" condition, or to a usage violation.

10. Certain I/O Addresses Incorrect when Paging is Enabled

Problem: When Paging is enabled, accessing I/O addresses in the range 00001000h-0000FFFFh (4K through 64K-1) or accessing coprocessor ports (I/O addresses 800000F8h-800000FFh) as a result of executing coprocessor opcodes, can generate incorrect I/O addresses if paging is enabled and the corresponding linear memory address is marked "present" and "dirty."

Furthermore, when paging has been enabled and is then turned off, paging translation continues to occur for memory or I/O cycles (I/O as described above) to linear addresses still stored in the TLB, but paging does not occur for linear addresses that result in a TLB miss.

Workaround: Unless paging is used, this item is not a problem. If paging is used but all I/O ports are below 00001000h (as in a PC-DOS system), then I/O is no problem.

If paging is used and I/O ports exist in the range 00001000h-0000FFFFh, then either have the memory pages at those linear addresses marked "not present" (to avoid having those pages table entries cached in the TLB), or if "present," have those pages mapped such that bits 12-15 of the physical address equal bits 12-15 of the linear address. Alternatively, re-assign any I/O ports in the range 00001000h-0000FFFFh to below 00001000h.

If paging is used and the coprocessor is also used, then have the memory page at linear address 80000xxxh either marked "not present" (to avoid having that page table entry cached in the TLB), or if "present," have the page mapped such that bit 31 (the most significant bit) of that page's physical address is a 1.

To completely disable 80386 paging when paging was previously enabled, the 80386 TLB should be flushed immediately after resetting the PG bit in CR0. The TLB can be flushed, you recall, by writing a Page Table Directory base address to register CR3.

11. Wrong ECX Update by REP INS

Problem: The ECX register (or CX in case of 16-bit operations) is not updated properly in the case of a REP INS instruction (INPut string instruction with any REPeat prefix) that is followed by an early-start instruction (e.g. PUSH, POP or memory reference instructions). After any REP-prefixed instruction, ECX is supposed to be 0 (null). But in the case of a REP INS instruction, ECX is not updated correctly and is 0FFFFFFFFh (or CX is 0FFFFh in case of 16-bit operations). It should be noted that the REP INS executes the correct number of iterations and EDI (or DI) is updated properly.

Workaround: After a REP INS instruction, do not rely on ECX (or CX) being zero. Hence, a new count (if any) should be MOVed into ECX, rather than being ADDed into ECX.

12. NMI Doesn't Always Bring Chip Out of Shutdown in Obscure Condition with Paging Enabled

Problem: If paging is enabled, and if the IDT gate for the Double Fault handler (the gate for exception 8) points to the null descriptor slot, descriptor 0, in the GDT (this would be very a strange way to set up a system), and a TLB miss occurs when accessing the null descriptor slot, the chip enters shutdown as it should in this case. In this specific case however, an incoming NMI will not be able to bring the 386 out of shutdown. In this specific case, only reset will bring the 386 out of shutdown.

Workaround: Ensure that the IDT gate for the Double Fault Handler has a non-null selectors for CS, and that SS of the destination level is also non-null.

13. HOLD Input During Protected Mode Interlevel IRET when Paging is Enabled

Problem: Under specific situations involving paging and the page privilege bits, the HOLD input, and a RET or IRET instruction performing an inter-level return to level 3, a problem can develop. These situations can be avoided by the workarounds given.

The first situation, when the inner level stack (levels 0, 1, and 2) <u>is</u> <u>not</u> dword aligned (or not word aligned in the case of a 16-bit (I)RET), requires that several conditions occur simultaneously:

1) Paging must be enabled, and the page table and directory entries for the inner level stacks must be marked as supervisor access only.

2) The software must execute an inter-level RET or IRET to a Protected Mode program at privilege level 3. An inter-level IRET to Virtual 8086 Mode does not exhibit this problem. An inter-level RET or IRET to level 1 or 2 does not exhibit this problem.

3) The inner level stack must be <u>un</u>aligned to a dword boundary (word boundary for a 16-bit (I)RET).

When the first situation occurs, a page fault (exception 14) occurs spuriously, indicating a page level protection violation during a "user" level read of the inner level stack.

The second situation, whether or not the inner level stack is dword aligned (or word aligned in the case of a 16-bit (I)RET), also requires that several conditions occur simultaneously:

1) Paging must be enabled, and the page table and directory entries for the inner level stacks must be marked as supervisor access only.

2) The software must execute an inter-level RET or IRET to a Protected Mode program at privilege level 3. An inter-level IRET to Virtual 8086 Mode does not exhibit this problem. An inter-level RET or IRET to level 1 or 2 does not exhibit this problem.

3) The bus HOLD input must be asserted during the read cycle which pops ESP (or SP) off the inner stack as a result of a RET or IRET instruction.

When the second situation occurs, no exception is generated, but the processor will drive an incorrect physical address during the read cycle in which SS is popped from the inner level stack.

Workarounds: <u>A software workaround to both situations</u> is to mark all pages which contain the inner level stacks as user readable. This prevents either the first or second situation from occurring. The segmentation system can be used to prevent user access to the linear addresses containing the inner-level stacks.

<u>A workaround if not using the HOLD input</u> is merely to keep the inner-level stacks aligned.

<u>A Hardware workaround if using the HOLD input but not using the software workaround above</u> is the following: Since the problem occurs during the first cycle after a locked cycle to read the CS descriptor, a hardware workaround is to prevent a HOLD request from hitting the

processor during bus cycle following a LOCKed cycle. This can be accomplished with a latch that delays the LOCK# signal through a flip-flop clocked by READY# to gate a HOLD request going into the chip. This will prevent a hold request from getting to the 80386 until after the completion of the first cycle after a LOCKed cycle. For the hardware workaround to be sufficient, all stacks must be properly aligned, and BS16# must be tied inactive.

14. Protected Mode LSL Instruction Should not be Followed by PUSH/POP

Problem: This item pertains only to Protected Mode. If the Protected Mode LSL instruction (Load Segment Limit instruction, executable only in Protected Mode) is immediately followed by certain instructions that perform a stack operation, such as PUSH or POP (see exact list below), the value of the (E)SP register may be incorrect after the stack operation. Note that stack operations resulting from interrupts or exceptions following LSL do update (E)SP correctly.

Workaround: Do not immediately follow the Protected Mode LSL instruction with any of the following stack operation instructions: IRET (intra-task), POPA, POPF, POP (mem, reg, seg-reg), RET (intrasegment or intersegment), CALL (direct intrasegment, direct intersegment, indirect intrasegment via reg), ENTER, PUSHA, PUSHF, PUSH (mem, reg, seg-reg, immed). Other instructions that operate on the stack (e.g. CALL indirect via memory, and LEAVE) can be used safely after the Protected Mode LSL. Note that even if a forbidden instruction immediately follows LSL, (E)SP may still be updated correctly, since this problem is data-dependent and only occurs if the LSL operation succeeded (i.e. if LSL set the ZF flag).

15. LSL/LAR/VERR/VERW Instructions Malfunction with Null Selector

Problem: The Protected Mode instructions LSL, LAR, VERR or VERW executed with a null selector (i.e. bits 15 through 2 of the selector set to zero) as the operand will operate on the descriptor at entry 0 of the GDT instead of unconditionally clearing the ZF flag.

Workaround: The "null descriptor" (i.e. the descriptor at entry 0 of the GDT) should be initialized to all zeroes. If the "null descriptor" is initialized to all zeroes (i.e. an invalid value), the access made by these instructions to the "null descriptor" will fail (since these instructions only operate on valid descriptors). The failure will be reported with ZF cleared, which is the desired behavior when the operand is a null selector. Note that many systems already have the "null descriptor" in the GDT initialized to zeroes, as is desired for this workaround.

16. "Not Present" LDT in VM86 Task Raises Wrong Exception

Problem: A task switch to a VM86 task that has a "not present" LDT descriptor will cause a Segment Not Present fault (exception 11) rather than an Invalid TSS fault (exception 10).

Workaround: The simplest workaround is to use a NULL selector for the LDT in a VM86 task, since the LDT is not used when executing in Virtual 86 mode. However, if an interrupt or exception occurs, the processor will switch out of Virtual 86 mode, into protected mode to handle the interrupt, without switching tasks. Thus, the operating system should be structured so that all Interrupt and Trap gates active when executing a VM86 task reference segments in the GDT.

If an LDT must be supplied for a task that executes in Virtual 86 mode, there are several easy workarounds. One is to ensure that LDT segments are never marked "not present" in their segment descriptors. Paging is not affected by this errata. LDT segments can be paged out and marked "not present" in their page descriptors in systems which use paging.

If the operating system must mark the LDT segment descriptor "not present", the "not present" (exception 11) handler must be able to handle the case of a "not present" LDT during a task switch. The "not present" exception is reported with the LDT selector as the error code and with the VM bit set to 1 in the EFLAGS image of the caller. Since a VM86 task cannot normally raise a "not present" fault, the "not present" exception handler can detect this case by checking if the stored VM bit is set. If so, the fault can be redirected to the TSS Fault handler.

17. Coprocessor Instructions Crossing Page/Segment Boundaries

Problem: If the first byte of a coprocessor (ESC) instruction is located on the last byte of a page or segment, and the second byte is located on a page or segment which would create a fault, then the processor will hang when it tries to signal the fault. The processor remains stopped until an interrupt, NMI, or RESET occurs. This errata applies only to coprocessor instructions in systems which use virtual memory.

Workaround: In virtual memory systems, the time-slice or watchdog timer provides an easy workaround, since a timer interrupt will always cause the processor to begin interrupt processing. The timer routine should test the following conditions to determine if this errata was encountered.

1) The saved CS:EIP must point within 8 bytes of the end of a page.
2) The last byte within the page must contain an ESC opcode.
3) All bytes between the saved CS:EIP and the ESC opcode must contain valid prefix opcodes (segment override 26h, 2Eh, 36h, 3Eh, 64h, 65h, address size override 67h, operand size override 66h).
4) The next page is not present, or not accessable.

If all four conditions are true, then the timer routine can assume this errata was encountered, and signal a page fault, which will clear the condition. This workaround should be placed in the Operating System, so that applications programs are unaffected.

18. Double Page Faults Do Not Raise Double Fault Exception

Problem: If a second page fault occurs, while the processor is attempting to enter the service routine for the first, then the processor will invoke the page fault (exception 14) handler a second time, rather than the double fault (exception 8) handler. A subsequent fault, though, will lead to shutdown.

Workaround: No workaround is necessary in a working system.

Design Notes

1. Read Cycles Require Valid Data Bus Levels

Please refer to Specification Change 5 for important news on proper system design for 386 read cycles.

2. Use of ESP as a Base Register With CALL, PUSH, and POP Instructions

This clarifies how ESP behaves with instructions that implicitly reference the stack and explicitly reference another location in memory using ESP as a base register.

Instruction	Explicit Memory Reference uses the ESP value...	ESP value used as base
CALL-indirect-thru-memory	before decrementing	old ESP
PUSH-from-memory	before decrementing	old ESP
POP-to-memory	after incrementing	new ESP

This is consistent in that the CALL-indirect-thru-memory and the PUSH-from-memory both use the same ESP value.

Furthermore, the relation between PUSH-from-memory and POP-to-memory is such that it allows the instruction sequence:

```
PUSH [ESP+n]
POP  [ESP+n]
```

to have the desirable property of both instructions referencing the same memory location.

3. Use of Code Breaks to Debug 86/286 Operating Systems

The RF bit in the EFLAGS register is cleared by a 16-bit IRET, making it difficult to use the on-chip debug registers to set code breakpoints to debug 16-bit operating systems. Data breakpoints work fine in all cases, and code breakpoints work fine as long as all interrupt handlers are 32-bits and return with 32-bit IRETs or task switches. In 16-bit environments, software debuggers should use the CC (single byte INT 3 instruction) to place software breakpoints in code.

4. Use of ESP in 16-bit Code with 32-bit Interrupt Handlers

When a 32-bit IRET is used to return to another privilege level, and the old level uses a 4G stack (B=1), while the new level uses a 64k stack (B=0), then only the lower word of ESP is updated. The upper word remains unchanged. This is fine for pure 16-bit code, as well as pure 32-bit code. However, when 32-bit interrupt handlers are present, 16-bit code should avoid any dependence on the upper word of ESP. No changes are necessary in existing 16-bit code, since the only way to access ESP in USE16 segments is through the 32-bit address size prefix.