AP-578

APPLICATION NOTE

Software and Hardware Considerations for FPU Exception Handlers for Intel Architecture Processors

February 1997

Order Number: 243291-002

CONTENTS

1.0 INTRODUCTION AND READING GUIDE

The primary purpose of this application note is to provide information to help software engineers write the most robust Floating-Point Unit (FPU) exception handlers possible. This note also provides the basic hardware information needed to design the MS-DOS* compatible interface[1] for the most recent generations of Intel Architecture processors, starting with the Intel486™ processor. (Because of the small amount of new design activity, the hardware interfaces for the 8086 through the Intel386™ processors are treated only briefly.) The third purpose is to provide a compendium of the history of the development and variations of the Intel Architecture Floating-Point Units (FPUs) as relevant to their exception handling. Following is a list of Intel Architecture processors and math coprocessors in chronological order.

- 8086 processor
- 8087 math coprocessor
- 80286 processor
- 80287 math coprocessor
- Intel386™ processor
- Intel387 math coprocessor
- Intel486™ DX processor (with integrated FPU)
- Intel486 SX processor
- Intel487 math coprocessor
- Pentium® processor (with integrated FPU)
- Pentium Pro processor (with integrated FPU)

Much of this material is in various sections of the *Pentium® Processor Family Developer's Manual,* Volume 3. There is also some material in this application note that is not published elsewhere. On the other hand, there is much additional material on the FPU from the *Pentium® Processor Family Developer's Manual,* Volume 3 which has not been reproduced here, including the details on each of its specific exceptions. Much of this will be useful in writing FPU exception handlers, so Volume 3 should be used as an essential reference along with this appliction note.

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).

The materials are presented in a mostly chronological order, which supports the history preservation purpose, and also minimizes forward references. Thus the main body of this application note begins with Section 2 which covers the six presently available generations of Intel Architecture FPUs in chronological order starting with the 8087. The history of the FPU exception handling has been complicated both by Intel's successful efforts to improve the performance and flexibility of the FPU through the generations, and by the decision to support upward compatibility for a large customer base which was implementing FPU exception handling in a way compatible with the first 8088 Personal Computers (PCs) and major Operating Systems (OSs). This second complication has resulted in two different systems or modes for FPU exception handling starting with the 80286 and 80287.

Beginning with the 80286 and 80287, Intel provided a dedicated input pin (ERROR#) on the 80286, to be connected to the ERROR# output pin on the 80287, for the FPU exceptions. When asserted, the ERROR# input triggers interrupt 16. The use of this dedicated interrupt for the FPU exception handler is referred to as the "native mode", and is recommended by Intel. However, for reasons explained in Sections 2.1 and 2.2, the majority of the Intel Architecture (IA) customer base has not been using the native mode, but rather the "MS-DOS compatible mode" for FPU exception handling. Since the MS-DOS compatible mode has the largest customer base, is the more complicated mode, and has changed the most between generations, it is the main focus of Section 2. In addition to the history of the

Footnotes

[1] WINDOWS* 95 and WINDOWS 3.1 (and earlier versions) use almost the same interface as MS-DOS*, and the recommendations herein for an MS-DOS compatable system apply to all three operating systems.

architecture and interfaces for FPU exception handling, Section 2 provides the basic hardware information needed to design the MS-DOS compatible interface for the most recent generations of IA processors, and discusses in detail several important system implications.

Section 3 describes the recommended protocol for writing MS-DOS compatible FPU exception handlers, with various options, along with discussions of several problems and how to avoid them. Most of the material is also applicable to native mode handlers.

Although the native mode of FPU exception handling was available from the second generation of the six presently available generations of the Intel Architecture FPUs (and brief discussions of it are provided in Section 2), we give the main presentation of it last, in Section 4. This is more chronologically consistent than it would seem because it has not become widely used until recently.

A software engineer who needs to write an MS-DOS compatible FPU exception handler but does not want to review the FPU history (or read any more about hardware than necessary) may skip Section 2 and begin reading Section 3. Then some subsections of Section 2 should be read as needed when referenced in Section 3. Someone writing a native mode exception handler that wants to read only what's necessary should start with Section 4, but then should also read Section 3, as most of the recommended protocol for FPU exception handling is the same for MS-DOS compatible and native modes and is not repeated in Section 4. Studying Section 4 first will allow this reader more easily to skip references back into Section 2 which are not relevant to the native mode.

A note on TERMINOLOGY: There are many variations of the words which are used to label an (unmasked) FPU error condition, and also the code which handles it. "Error", "exception" and "fault" are used to refer to the condition. Such a condition results in an interrupt, if no mask or block is in effect along the interrupt pathway. The code which handles the interrupt can be referred to as an error or exception or fault handler, or an interrupt or exception service routine, etc. The phrase "**exception handler**" has been used consistently (as much as possible) in this application note, for several reasons: "Exception" is less general than interrupt (which includes external hardware interrupts and software interrupts, as well as the processor problem conditions called exceptions or faults), but correctly **more** general than error or fault (because e.g. a precision exception caused by the fact that the number 1/3 cannot be exactly represented in the 80 bit FPU format is not really due to any mistake or error!). However, the reader should be aware that a number of the variations given above can be found in the literature, and that when applied to the FPU, they all mean the same thing.

2.0 MS-DOS* COMPATIBLE HANDLERS AND THEIR ISSUES OVER GENERATIONS

2.1 Origin of MS-DOS* Mode: 8088 and 8087

The 8087 has an output pin, INT, which it asserts when an unmasked exception occurs. There is no dedicated pin or interrupt vector number in the 8088 or 8086 specific for an FPU error assertion. Intel recommended that the FPU INT be routed to the 8088 or 8086 INTR pin through an 8259A Programmable Interrupt Controller (PIC), and not to the NMI input. However, the original PC design attached INT to NMI anyway, because by the time the 8087 was available, the original PC had already assigned other functions to the 8 inputs of the single PIC used in that design.

2.2 Development of MS-DOS* Mode with 80286 and 80287; Intel386Ô Processor and Intel387 Math Coprocessor

The 80286 and 80287 and Intel386 processor and Intel387 math coprocessor pairs are each provided with ERROR# pins that are recommended to be connected between the processor and FPU. If this is done, when an unmasked FPU exception occurs, the FPU records the exception, and asserts its ERROR# pin. The processor recognizes this active condition of the ERROR# status line at the next WAIT or ESC instruction in its instruction stream, and branches to the FPU exception handler at interrupt vector 16. This is the native mode.

However, it was important to maintain maximum compatibility with the already significant 8088 and 8086 PC software base, where the NMI vector (#2) was used for FPU exceptions and vector 16 was used for the BIOS video software interrupt. So the original IBM PC-AT* design for the 80286 and 80287 maintained Vector #16 for the BIOS video, and vector 2 was shared between the FPU exception and the new parity checking feature. A parity error detected by external hardware directly triggered vector 2 through the NMI pin. The FPU exception was handled by tying the 80286 RROR# input permanently high, and the 80287 ERROR# output was tied to the IRQ13 interrupt input on the second (cascaded) PIC in the PC-AT design. The PIC was programmed to issue vector 75H when IRQ13 was triggered.[2] But to maintain compatibility with older PC software that expected to access its own FPU exception handler by changing vector 2, the BIOS routine activated by INT 75H branches to INT 2. The standard INT 2 routine tests to see if the signal is due to the NMI pin (in which case it branches to the Parity Error handler) or an FPU exception.

2.2.1 SPECIAL HARDWARE FOR THE 80287 INTERFACE

It is necessary to guarantee, in the case of an 80287 exception, that the exception will be handled through the external loop using IRQ13 in the cascaded PIC before other 80287 instructions are sent over from the 80286. This is done by asserting BUSY# to the 80286, which normally means that the 80287 is still busy with a previous instruction, and so blocks the 80286 from sending another until BUSY# is de-asserted. This additional use of BUSY# is implemented by an edge triggered flip-flop which latches BUSY# using ERROR# from the 80287 as a clock. The output of this latch is OR'ed with the BUSY# output of the 80287 and drives the BUSY# input of the 80286. This PC-AT scheme effectively delays

deactivation of BUSY# at the 80286 whenever an 80287 ERROR# is signaled.

Since the BUSY# signal to the 80286 remains active after an exception, the IRQ13 interrupt (exception) handler (accessed through interrupt vector 75H) is guaranteed to execute before any other 80287 instruction can begin (except for some special control instructions).The IRQ13 handler clears the BUSY# latch (by writing to a special I/O port defined at 0F0H), thus allowing execution of 80287 instructions to proceed. The handler then branches to the NMI handler (interrupt vector 2), where the user defined numeric exception handler resides in PC compatible systems. Thus the PC-AT scheme approximates the exception reporting scheme between the 8087 and 8088 in the original PC.

2.2.2 SPECIAL HARDWARE FOR THE INTEL387™ MATH COPROCESSOR INTERFACE

The Intel386 processor can use a PC-AT compatible interface to communicate with an Intel387 math coprocessor, that is similar to the one in the 80286 and 80287 system above. As with the 80286, the Intel386 processor ERROR# pin should be tied permanently inactive (high), and the Intel387 ERROR# output used both to drive IRQ13, and to latch BUSY# in a flip-flop. The IRQ13 handler (vector 75H) should clear the BUSY# latch and branch to the NMI handler, as in the 80286 case.

However, an additional hardware feature is needed to manage the PEREQ signal to the Intel386 processor. After the Intel387 math coprocessor asserts ERROR#, and then its BUSY# signal has gone inactive, external hardware must re-assert the PEREQ signal to the Intel386 processor. This is needed for store instructions (for example, FST *mem*) because the Intel387 math coprocessor drops PEREQ once it

Footnotes

[2] WINDOWS 95 and WINDOWS 3.1 (and earlier versions) use interrupt 5DH instead of 75H, but the recommendations herein apply to systems using these WINDOWS operating systems, as well as MS-DOS.

signals an exception. While the Intel386 processor has not yet recognized the occurrence of the exception, it still expects the data transfers to complete via PEREQ re-activation. It is permissible for the Intel386 processor to receive undefined data during such I/O read cycles. Disabling the Intel387 math coprocessor is not necessary, because the dummy data transfer cycles directed to the Intel387 math co processor when PEREQ is externally reactivated for the Intel386 processor will not disturb the operation of the Intel387math coprocessor. The IRQ13 interrupt handler should remove the extension of BUSY# and also the re-activation of PEREQ via a write to PC/AT compatible hardware at I/O port 0F0H.

An Intel387 math coprocessoroffers significant performance improvements over the 80287, but because the Intel386 processor was ready for production before the Intel387math coprocessor, the Intel386 processor was designed to work with either the 80287 or Intel387 math coprcoessors. The Intel386 processor automatically configures itself for the attached FPU on reset by testing the ERROR# pin, and setting or clearing bit 4 in CR0 (see Section 10.1.3 in the *Pentium® Processor Family Developer's Manual,* Volume 3). This bit is the ET (Extension Type) bit, and it will be set if ERROR# is low (meaning an Intel387 is attached) and cleared if ERROR# is high (meaning there is an 80287 or no FPU attached). The MS-DOS compatible hardware interface is similar to that for the Intel386 processor and Intel387 math coprocessor combination.

2.3 FERR# & IGNNE# with Intel486™ and Pentium® Processors with CR0.NE=0

In the Intel486 and Pentium® processors, more enhancements and speedup features have been added to the corresponding FPUs. Also, the FPU is built into the same chip as the processor, which allows further increases in speed. MS-DOS compatibility for exception handling has also been built in, with the NE bit in control register CR0 selecting the MS-DOS compatible mode if made zero. (NE=1 selects the native or internal mode, which generates Interrupt 16, which is the same as the native version of exception handling for the 80286 and 80287 and the Intel386 processors and Intel 387 math coprocessor.)

In MS-DOS compatible mode, the FERR# (Floating-point ERRor) output replaces the ERROR# signal from the previous generations, and is connected to a PIC. A new input signal, IGNNE# (**IGN**ore **N**umeric **E**rror), is provided to allow the FPU exception handler to execute FPU instructions, if desired, without first clearing the error condition and without triggering the interrupt a second time. This IGNNE# feature is needed to replicate the capability that was provided on MS-DOS compatible Intel 80286 and 80287 and the Intel386 processors and INtel 387 math coprocessor-based systems by turning off the BUSY# signal, when inside the FPU exception handler, before clearing the error condition.

Note that Intel, in order to provide Intel486 processors for market segments which had no need for an FPU, created the “SX” versions. These Intel486 SX processors did not contain the floating-point unit. Intel also produced Intel487 SX math coprocessors for end users who later decided to upgrade to a system with an FPU. These Intel487 SX math coprocessors are similar to standard Intel486 processors with a working FPU on board. Thus the external circuitry necessary to support the MS-DOS compatible mode for Intel487 SX math coprocessors is the same as for standard Intel486 DX processors.

Note that the special DP (Dual Processing) mode for Pentium processors, and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors, support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatible FPU mode for systems using more than one processor.

2.3.1 BASIC RULES: WHEN FERR# IS GENERATED

- Assume the following conditions: NE=0, the IGNNE# input is de-asserted, and then an FPU instruction causes an unmasked FPU exception. Then in most cases, deferred error reporting occurs. This means that the processor does not respond immediately, but rather freezes just before executing the *next* WAIT or FPU instruction (except for “No-Wait” instructions, which the FPU executes regardless of an error condition).
- At the same time that the processor freezes, it also asserts the FERR# output.

- The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# assertion.
- In MS-DOS compatible systems, FERR# is fed to the IRQ13 input in the cascaded PIC, which generates interrupt 75H, which then branches to interrupt 2, as described above for the 80286and 80287 and Intel386 processor and Intel387 processor-based systems.

These cases in which FERR# is not asserted at the time of the error, but rather at the next FPU or WAIT instruction, include all exceptions caused by the basic arithmetic instructions (including FADD, FSUB, FMUL, FDIV, FSQRT, FCOM and FUCOM), precision exceptions caused by all types of FPU instructions, and numeric underflow and overflow on all types of FPU instructions except stores to memory. We will refer to these cases as deferred (error reporting).

On the other hand, there are some exceptions, which when caused by some instructions, drive FERR# at the time that the exception occurs. These include FPU stack fault, invalid operation and denormal exceptions caused by all transcendental instructions, FSCALE, FXTRACT, FPREM and others, and all exceptions (except precision) when caused by FPU store instructions. These cases are called immediate (error reporting). (These cases will, like the deferred, cause the processor to freeze just before executing the next WAIT or FPU instruction if the error condition has not been cleared by that time.) Note that in general, whether an FPU exception case is deferred or immediate depends both on which exception occurred, and which instruction caused that exception. A complete specification of these cases, which applies also to the Intel486, is given in Section 5.1.21 in the *Pentium® Processor Family Developer's Manual,* Volume 1.

If NE=0 but the IGNNE# input is active while an unmasked FPU exception is in effect, the processor disregards the exception, does not assert FERR#, and continues. If IGNNE# is then de-asserted and the FPU exception has not been cleared, the processor will respond as described above. (That is, an immediate exception case will assert FERR# immediately. A deferred exception case will assert FERR# and freeze just before the next FPU or WAIT instruction.) The assertion of IGNNE# is intended for use only inside the FPU exception handler, where it is needed if one wants to execute non-control FPU instructions for diagnosis, before clearing the exception condition. When IGNNE# is asserted inside the exception handler, a preceding FPU exception has already caused FERR# to be asserted, and the external interrupt hardware has responded, but IGNNE# assertion still prevents the freeze at FPU instructions. Note that if IGNNE# is left active outside of the FPU exception handler, additional FPU instructions may be executed after a given instruction has caused an FPU exception. In this case, if the FPU exception handler ever did get invoked, it could not determine which instruction caused the exception.

To properly manage the interface between the processor's FERR# output, its IGNNE# input, and the IRQ13 input of the PIC, additional external hardware is needed. A recommended configuration is described below.

2.3.2 RECOMMENDED EXTERNAL HARDWARE TO SUPPORT MS-DOS* COMPATIBILITY

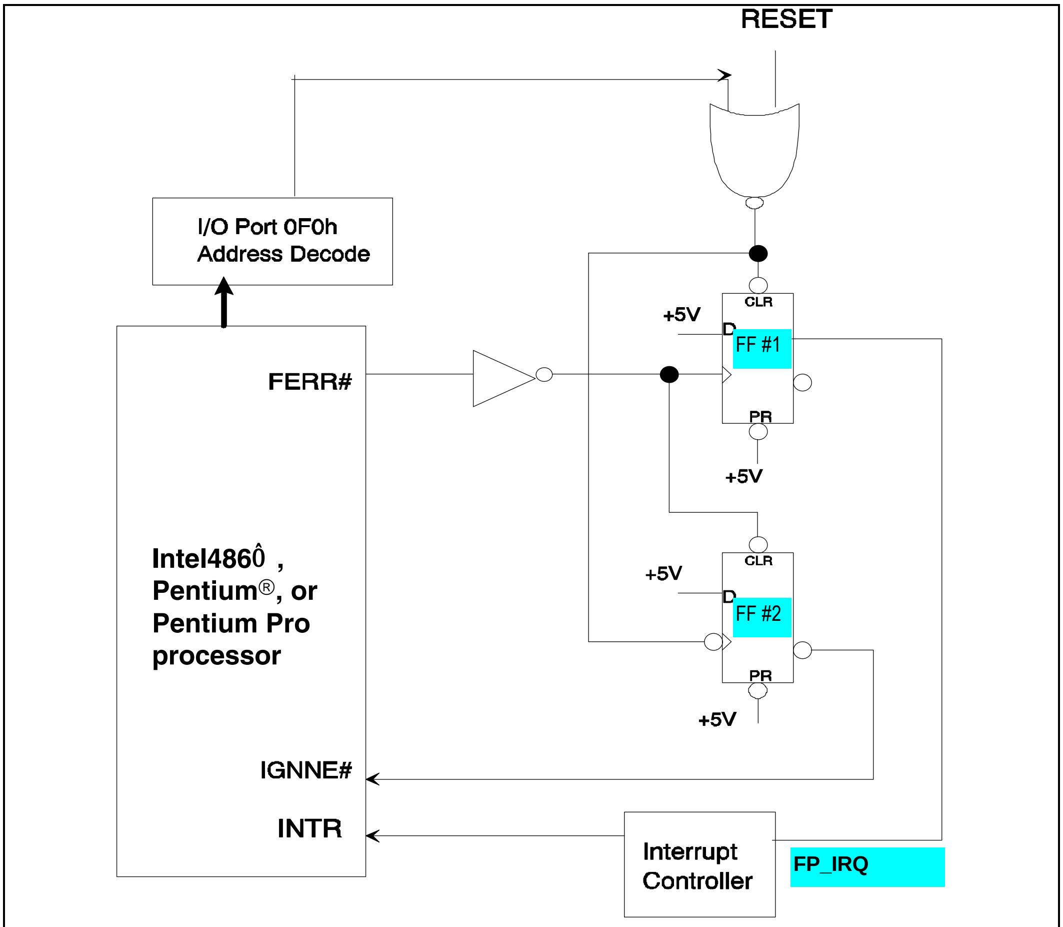
Figure 1 below provides an external circuit which will assure proper handling of FERR# and IGNNE# when an FPU exception occurs. In particular, it assures that IGNNE# will be active only inside the FPU exception handler without depending on the order of actions by the exception handler. Some hardware implementations have been less robust because they have depended on the exception handler to clear the FPU exception interrupt request to the PIC (FP_IRQ signal) *before* the handler causes FERR# to be de-asserted by clearing the exception from the FPU itself. Figure 2 below shows the details of how IGNNE# will behave when the circuit in Figure 1 is implemented. The temporal regions within the FPU exception handler activity are described as follows:

1. The FERR# signal is activated by an FPU exception and sends an interrupt request through the PIC to the processor's INTR pin.
2. During the FPU interrupt service routine (exception handler) the processor will need to clear the interrupt request latch (Flip Flop #1). It may also want to execute non-control FPU instructions before the exception is cleared

from the FPU. For this purpose the IGNNE# must be driven low. Typically in the PC environment an I/O access to Port 0F0H clears the external FPU exception interrupt request (FP_IRQ). In the recommended circuit, this access also is used to activate IGNNE#. With IGNNE# active the FPU exception handler may execute any FPU instruction without being blocked by an active FPU exception.

3. Clearing the exception within the FPU will cause the FERR# signal to be deactivated and then there is no further need for IGNNE# to be active. In the recommended circuit, the deactivation of FERR# is used to deactivate IGNNE#. If another circuit is used, the software and circuit together must assure that IGNNE# is deactivated no later than the exit from the FPU exception handler.

4. In the circuit in Figure 1 when the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. So the handler can activate IGNNE#, if needed, by doing this 0F0H access before clearing



LEGEND	
FF #n:	Flip Flop #n
CLR:	Clear or reset

Figure 1. Recommended Circuit for MS-DOS* Compatible FPU Exception Handling

the FPU exception condition (which de-asserts FERR#). However, the circuit does not depend on the order of actions by the FPU exception handler to guarantee the correct hardware state upon exit from the handler. The flip flop which drives IGNNE# to the processor has its CLEAR input attached to the inverted FERR#. This ensures that IGNNE# can never be active when FERR# is inactive. So if the handler clears the FPU exception condition *before* the 0F0H access, IGNNE# does not get activated and left on after exit from the handler.

2.3.3 "NO-WAIT" FPU INSTRUCTIONS CAN GET FPU INTERRUPT IN WINDOW

The Pentium and the Intel486 processors implement the "No-Wait" Floating-Point instructions (FNINIT, FNCLEX, FNSTENV, FNSAVE, FNSTSW, FNSTCW, FNENI, FNDISI or FNSETPM - See Section 6.3.7 in the *Pentium® Processor Family Developer's Manual*, Volume 3) in the MS-DOS Compatibility mode (CR0.NE = 0) in the following manner:

If an unmasked numeric exception is pending from a preceding FPU instruction, a member of the "No-Wait" class of instructions will, at the beginning of its execution, assert the FERR# pin in response to that exception just like other FPU instructions, but then, unlike the other FPU instructions, FERR# will be de-asserted. This de-assertion was implemented to allow the "No-Wait" class of instructions to proceed without an interrupt due to any pending numeric exception. However, the brief assertion of FERR# is sufficient to latch the FPU exception request into most hardware interface implementations (including Intel's recommended circuit).

All the FPU instructions are implemented such that during their execution, there is a window in which the processor will sample and accept external interrupts. If there is a pending interrupt, the processor services the interrupt first before resuming the execution of the instruction. Consequently, it is possible that the "No-Wait" Floating-Point instruction may accept the external interrupt caused by it's own assertion of the FERR# pin in the event of a pending unmasked numeric exception, which is not an explicitly documented behavior of a "No-Wait" instruction. This process is illustrated by Figure 3, which is followed by a detailed description of the several cases possible.

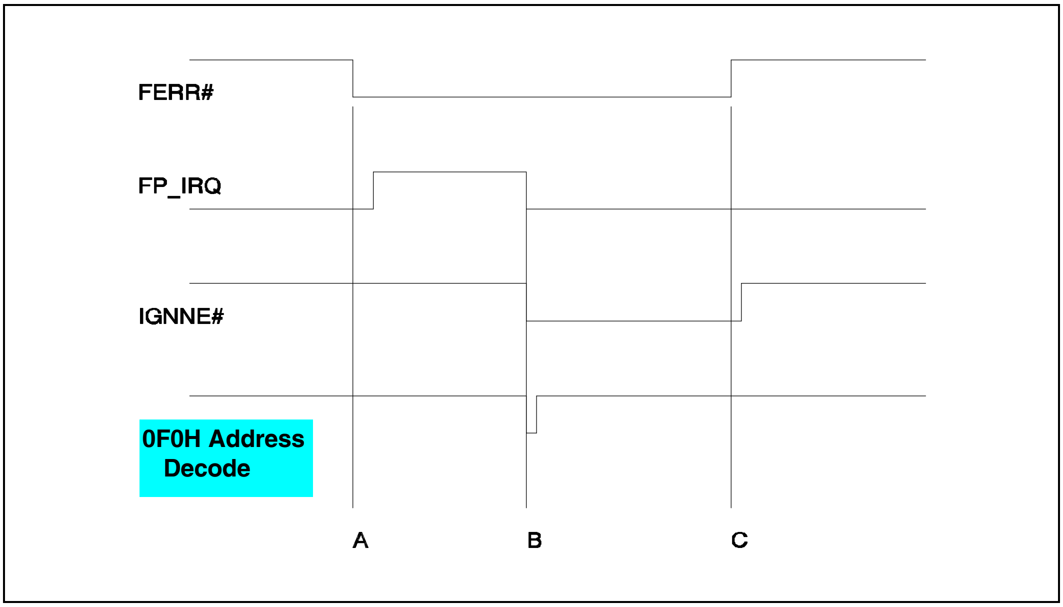


Figure 2. Behavior of Signals During FPU Exception Handling

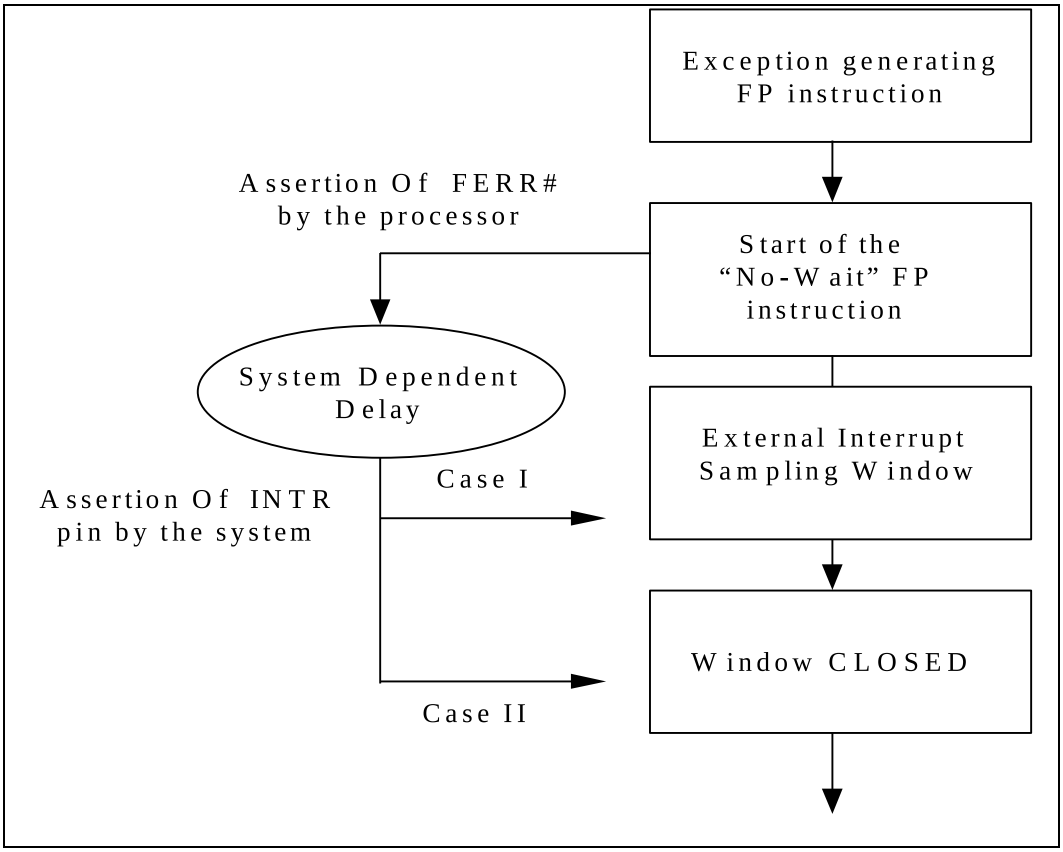

Exception generating FP instruction
Assertion Of FERR# by the processor
Start of the "No-Wait" FP instruction
System Dependent Delay
External Interrupt Sampling Window
Assertion Of INTR pin by the system
Case I
Window CLOSED
Case II

Figure 3: Timing of Receipt of External Interrupt

Figure 3 assumes that a floating-point instruction which generates a "deferred" error (as defined above in the Section 2.3.1), which asserts the FERR# pin only on encountering the next floating-point instruction, causes an unmasked numeric exception. Assume that the next floating-point instruction following this instruction is one of the "No-Wait" floating-point instructions. The FERR# pin is asserted by the processor to indicate the pending exception on encountering the "No-Wait" floating-point instruction. After the assertion of the FERR# pin the "No-Wait" floating-point instruction opens a window where the pending external interrupts are sampled.

Then there are two cases possible depending on the timing of the receipt of the interrupt via the INTR pin (asserted by the system in response to the FERR# pin) by the processor.

Case 1
If the system responds to the assertion of FERR# pin by the "No-Wait" floating-point instruction via the INTR pin during this window then the interrupt is serviced first, before resuming the execution of the "No-Wait" floating-point instruction.

Case 2
If the system responds via the INTR pin after the window has closed then the interrupt is recognized only at the next instruction boundary.

There are two other ways, in addition to Case I above, in which a "No-Wait" floating-point instruction can service a numeric exception inside its interrupt window. First, the first floating-point error condition could be of the "immediate" category (as defined in Section 2.3.1) that assert FERR#

immediately. If the system delay before asserting INTR is long enough, relative to the time elapsed before the "No-Wait" floating-point instruction, INTR can be asserted inside the interrupt window for the latter. Second, consider two "No-Wait" FPU instructions in close sequence, and assume that a previous FPU instruction has caused an unmasked numeric exception. Then if the INTR timing is too long for an FERR# signal triggered by the first "No-Wait" instruction to hit the first instruction's interrupt window, it could catch the interrupt window of the second.

The possible malfunction of a "No-Wait" FPU instruction explained above cannot happen if the instruction is being used in the manner for which Intel originally designed it. The "No-Wait instructions were intended to be used inside the FPU exception handler, to allow manipulation of the FPU before the error condition is cleared, without hanging the processor because of the FPU error condition, and without the need to assert IGNNE#. They will perform this function correctly, since before the error condition is cleared, the assertion of FERR# that caused the FPU error handler to be invoked is still active. Thus the logic that would assert FERR# briefly at a "No-Wait" instruction causes no change since FERR# is already asserted. The "No-Wait" instructions may also be used without problem in the handler after the error condition is cleared, since now they will not cause FERR# to be asserted at all.

If a "No-Wait" instruction is used outside of the FPU exception handler, it may malfunction as explained above, depending on the details of the hardware interface implementation and which particular processor is involved. The actual interrupt inside the window in the "No-Wait" instruction may be blocked by surrounding it with the instructions: PUSHFD, CLI, "No-Wait", then POPFD. (CLI blocks interrupts, and the push and pop of flags preserves and restores the original value of the interrupt flag.) However, if FERR# was triggered by the "No-Wait", its latched value and the PIC response will still be in effect. Further code can be used to check for and correct such a condition, if needed. Section 3.6 (Considerations When FPU Shared Between Tasks) discusses an important example of this type of problem and gives a solution.

2.4 Pentium® Pro Processor with CR0.NE=0

When bit NE=0 in CR0, the MS-DOS* compatible mode of the Pentium Pro processor provides FERR# and IGNNE# functionality that is almost identical to the Intel486 and Pentium processors. The same external hardware, as described in Section 2.3.2 above, is recommended for the Pentium Pro processor as well as the two previous generations. The only change to MS-DOS compatible FPU exception handling with the Pentium Pro processor is that all exceptions for all FPU instructions cause immediate error reporting. That is, FERR# is asserted as soon as the FPU detects an unmasked exception; there are no cases in which error reporting is deferred to the next FPU or WAIT instruction. (As is discussed in Section 2.3.1, most exception cases in the Intel486 and Pentium processors are of the deferred type.)

Although FERR# is asserted immediately upon detection of an unmasked FPU error, this certainly does not mean that therequested interrupt will always be serviced before the next instruction in the code sequence is executed. To begin with, the Pentium Pro processor executes several instructions simultaneously. There also will be a delay, which depends on the external hardware implementation, between the FERR# assertion from the processor and the responding INTR assertion to the processor. Further, the interrupt request to the PICs (IRQ13) may be temporarily blocked by the OS, or delayed by higher priority interrupts, and processor response to INTR itself is blocked if the OS has cleared the IF bit in EFLAGS.

However, just as with the Intel486 and Pentium processors, if the IGNNE# input is inactive, a floating-point exception which occurred in the previous FPU instruction and is unmasked causes the processor to freeze immediately when encountering the next WAIT or FPU instruction (except for "No-Wait" instructions). This means that if the FPU exception handler has not already been invoked due to the earlier exception (and therefore the handler has not cleared that exception state from the FPU), the processor is forced to wait for the handler to be invoked and handle the exception, before the processor can execute another WAIT or FPU instruction.

As explained in Section 2.3.3, if a “No-Wait” instruction is used outside of the FPU exception handler, in the Intel486 and Pentium processors, it may accept an unmasked exception from a previous FPU instruction which happens to fall within the external interrupt sampling window that is opened near the beginning of execution of all FPU instructions. This will not happen in the Pentium Pro processor, because this sampling window has been removed from the “No-Wait” group of FPU instructions.

3.0 RECOMMENDED PROTOCOL FOR MS-DOSÔ AND WINDOWS* 95 COMPATIBLE HANDLERS[3]

The activities of numeric programs can be split into two major areas: program control and arithmetic. The program control part performs activities such as deciding what functions to perform, calculating addresses of numeric operands, and loop control. The arithmetic part simply adds, subtracts, multiplies, and performs other operations on the numeric operands. The processor is designed to handle these two parts separately and efficiently. An FPU exception handler, if a system chooses to implement one, is often one of the most complicated parts of the program control code.

3.1 Numeric Exceptions and their Defaults

The FPU can recognize six classes of numeric exception conditions while executing numeric instructions:

1. #I — Invalid operation
   #IS—Stack fault
   #IA—IEEE standard invalid operation
2. #Z—Divide-by-zero
3. #D—Denormalized operand
4. #O—Numeric overflow
5. #U—Numeric underflow
6. #P—Inexact result (precision)

For complete details on these exceptions and their defaults, see the *Pentium*Ô *Processor Family Developer’s Manual,* Volume 3, Sections 7.1.7 through 7.1.13.

3.1.1 TWO OPTIONS FOR HANDLING NUMERIC EXCEPTIONS

Depending on options determined by the software system designer, the processor takes one of two possible courses of action when a numeric exception occurs:

1. The FPU can handle selected exceptions itself, producing a default fix-up that is reasonable in most situations. This allows the numeric program execution to continue undisturbed. Programs can mask individual exception types to indicate that the FPU should generate this safe, reasonable result whenever the exception occurs. The default exception fix-up activity is treated by the FPU as part of the instruction causing the exception; no external indication of the exception is given (except that the instruction takes longer to execute when it handles a masked exception.) When masked exceptions are detected, a flag is set in the numeric status register, but no information is preserved regarding where or when it was set.

Footnotes

[3] Although there are some differences in the way FPU exceptions are handled between MS-DOS, and WINDOWS 95 and WINDOWS 3.1 (and earlier versions), the WINDOWS operating systems operate the processor in the MS-DOS compatable mode, and the recommended protocol given here applies to all these systems. On the other hand, current versions of WINDOWS NT use the FPU native mode.

2. Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. The exception handler can then implement any sort of recovery procedures desired for any numeric exception detectable by the FPU.

3.1.2 AUTOMATIC EXCEPTION HANDLING: USING MASKED EXCEPTIONS

Each of the six exception conditions described above has a corresponding flag bit in the FPU status word and a mask bit in the FPU control word. If an exception is masked (the corresponding mask bit in the control word = 1), the processor takes an appropriate default action and continues with the computation. The processor has a default fix-up activity for every possible exception condition it may encounter. These masked-exception responses are designed to be safe and are generally acceptable for most numeric applications.

For example, if the Inexact result (Precision) exception is masked, the system can specify whether the FPU should handle a result that cannot be represented exactly by one of four modes of rounding: rounding it normally, chopping it toward zero, always rounding it up, or always down. If the Underflow exception is masked, the FPU will store a number that is too small to be represented in normalized form as a denormal (or zero if it's smaller than the smallest denormal). Note that when exceptions are masked, the FPU may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the FPU could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow.

As an example of how even severe exceptions can be handled safely and automatically using the default exception responses, consider a calculation of the parallel resistance of several values using only the standard formula (Figure 4). If R1 becomes zero, the circuit resistance becomes zero. With the divide-by-zero and precision exceptions masked, the processor will produce the correct result. FDIV of R1 into 1 gives infinity, and then FDIV of (infinity +R2 +R3) into 1 gives zero.

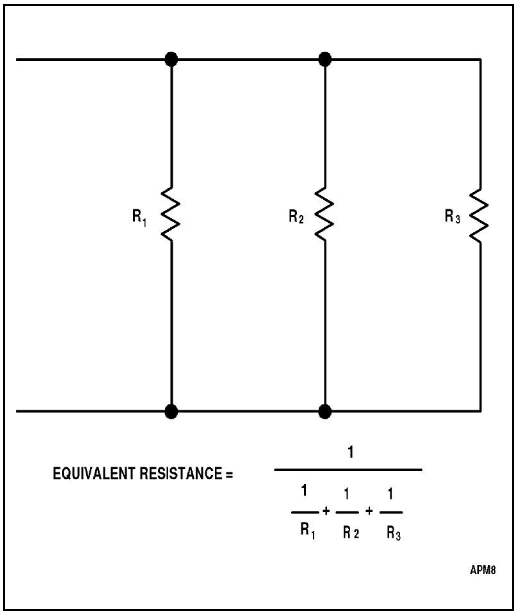

R_1
R_2
R_3
EQUIVALENT RESISTANCE = $\dfrac{1}{\dfrac{1}{R_1}+\dfrac{1}{R_2}+\dfrac{1}{R_3}}$
APM8

Figure 4. Arithmetic Example Using Infinity

By masking or unmasking specific numeric exceptions in the FPU control word, programmers can delegate responsibility for most exceptions to the processor, reserving the most severe exceptions for programmed exception handlers. Exception-handling software is often difficult to write, and the masked responses have been tailored to deliver the most reasonable result for each condition. For the majority of applications, masking all exceptions yields satisfactory results with the least programming effort. Certain exceptions can usefully be left unmasked during the debugging phase of software development, and then masked when the clean software is actually run. An invalid-operation exception for example, typically indicates a program error that must be corrected.

The exception flags in the FPU status word provide a cumulative record of exceptions that have occurred since these flags were last cleared. Once set, these flags can be cleared only by executing the FCLEX/FNCLEX (clear exceptions) instruction, by reinitializing the FPU with FINIT/FNINIT or FSAVE/FNSAVE, or by overwriting the flags with an FRSTOR or FLDENV instruction. This allows a programmer to mask all exceptions, run a calculation, and then inspect the status word to see if any exceptions were detected at any point in the calculation.

3.2 Software Exception Handling

If the FPU in or with an Intel family processor (80286 and onwards) encounters an unmasked exception condition, with the system operated in the MS-DOS compatible mode and with IGNNE# not asserted, a software exception handler is invoked through a PIC and the processor's INTR pin. The FERR# (or ERROR#) output from the FPU that begins the process of invoking the exception handler may occur when the error condition is first detected, or when the processor encounters the next WAIT or FPU instruction. Which of these two cases occurs depends on the processor generation and also on which exception and which FPU instruction triggered it, as discussed earlier in Section 2. The elapsed time between the initial error signal and the invocation of the FPU exception handler depends of course on the external hardware interface, and also on whether the external interrupt for FPU errors is enabled. But the architecture ensures that the handler will be invoked before execution of the next WAIT or floating-point instruction since an unmasked floating-point exception causes the processor to freeze just before executing such an instruction (unless the IGNNE# input is active, or it is a "No-Wait" FPU instruction).

The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# (or ERROR#) output of the processor (or coprocessor), usually through IRQ13 on the "slave" PIC, and then through INTR. Then the external interrupt invokes the exception handling routine. Note that if the external interrupt for FPU errors is disabled when the processor executes an FPU instruction, the processor will freeze until some other (enabled) interrupt occurs if an unmasked FPU exception condition is in effect. If NE = 0 but the IGNNE# input is active, the processor disregards the exception and continues. Error reporting via an external interrupt is supported for MS-DOS compatibility. Chapter 23 of the *Pentium® Processor Family Developer's Manual,* Volume 3 contains further discussion of compatibility issues.

The references above to the ERROR# output from the FPU apply to the Intel387 and 80287 math coprocessors (NPX chips). If one of these coprocessors encounters an unmasked exception condition, it signals the exception to the 80286 or Intel386 processor using the ERROR# status line between the processor and the coprocessor. See Section 2.2 above, and Chapter 23 of the *Pentium® Processor Family Developer's Manual,* Volume 3 for differences in FPU exception handling.

The exception-handling routine is normally a part of the systems software. The routine must clear (or disable) the active exception flags in the FPU status word before executing any FP instructions that cannot complete execution when there is a pending FP exception. Otherwise, the FP instruction will trigger the FPU interrupt again, and the system will be caught in an endless loop of nested FP exceptions, and hang. In any event, the routine must clear (or disable) the active exception flags in the FPU status word after handling them, and before IRET(D). Typical exception responses may include:

- Incrementing an exception counter for later display or printing
- Printing or displaying diagnostic information (e.g., the FPU environment and registers)
- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it

Applications programmers should consult their operating system's reference manuals for the appropriate system response to numerical exceptions. For systems programmers, some details on writing software exception handlers are provided in Chapter 14 of the *Pentium® Processor Family Developer's Manual,* Volume 3, as well as in this application note.

As discussed in Section 2.3.2, some early FERR# to INTR hardware interface implementations are less robust than the recommended circuit. This is because they depended on the exception handler to clear the FPU exception interrupt request to the PIC (by accessing port 0F0H) *before* the handler causes FERR# to be de-asserted by clearing the exception from the FPU itself. To eliminate the chance of a problem with this early hardware, Intel recommends that FPU exception handlers always access port 0F0H before clearing the error condition from the FPU.

3.3 Synchronization Required for Use of FPU Exception Handlers

Concurrency or synchronization management requires a check for exceptions before letting the processor change a value just used by the FPU. It is important to remember that almost any numeric instruction can, under the wrong circumstances, produce a numeric exception.

3.3.1 EXCEPTION SYNCHRONIZATION: WHAT, WHY AND WHEN

Exception synchronization means that the exception handler inspects and deals with the exception in the context in which it occurred. If concurrent execution is allowed, the state of the processor when it recognizes the exception is often *not* in the context in which it occurred. The processor may have changed many of its internal registers and be executing a totally different program by the time the exception occurs. If the exception handler cannot recapture the original context, it cannot reliably determine the cause of the exception or to recover successfully from the exception. To handle this situation, the FPU has special registers updated at the start of each numeric instruction to describe the state of the numeric program when the failed instruction was attempted. This provides tools to help the exception handler recapture the original context, but the application code must also be written with synchronization in mind. Overall, exception synchronization must ensure that the FPU and other relevant parts of the context are in a well defined state when the handler is invoked after an unmasked numeric exception occurs.

When the FPU signals an unmasked exception condition, it is requesting help. The fact that the exception was unmasked indicates that further numeric program execution under the arithmetic and programming rules of the FPU will probably yield invalid results. Thus the exception must be handled, and with proper synchronization, or the program will not operate reliably.

For programmers in higher-level languages, all required synchronization is automatically provided by the appropriate compiler. However, for assembly language programmers exception synchronization remains the responsibility of the programmer. It is not uncommon for a programmer to expect that their numeric program will not cause numeric exceptions after it has been tested and debugged, but in a different system or numeric environment, exceptions may occur regularly nonetheless. An obvious example would be use of the program with some numbers beyond the range for which it was designed and tested. The example in Section 3.3.2 shows a more subtle way in which unexpected exceptions can occur.

As described in Section 3.1.1, depending on options determined by the software system designer, the processor can perform one of two possible courses of action when a numeric exception occurs.

- The FPU can provide a default fix-up for selected numeric exceptions. If the FPU performs its default action for all exceptions, then the need for exception synchronization is not manifest. However, code is often ported to contexts and operating systems for which it was not originally designed. The example below illustrates that it is safest to always consider exception synchronization when designing code that uses the FPU.
- Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. When an FPU exception handler will be invoked, synchronization must always be considered to assure reliable performance.

The following examples illustrate the need to always consider exception synchronization when writing numeric code, even when the code is initially intended for execution with exceptions masked.

3.3.2 EXCEPTION SYNCHRONIZATION: EXAMPLES

In the following examples, three instructions are shown to load an integer, calculate its square root, then increment the integer. The synchronous execution of the FPU will allow both of these

Incorrect Error Synchronization:

```
FILD     COUNT          ; FPU instruction
INC      COUNT          ; integer instruction alters operand
FSQRT                   ; subsequent FPU instruction -- error
         ; from previous FPU instruction detected here
```

Proper Error Synchronization:

```
FILD     COUNT                ; FPU instruction
FSQRT                   ; subsequent FPU instruction -- error from
                  ; previous FPU instruction detected here
INC COUNT               ; integer instruction alters operand
```

programs to execute correctly, with INC COUNT being executed in parallel in the processor, as long as no exceptions occur on the FILD instruction. However, if the code is later moved to an environment where exceptions are unmasked, the code in the first example will not work correctly:

In some operating systems supporting the FPU, the numeric register stack is extended to memory. To extend the FPU stack to memory, the invalid exception is unmasked. A push to a full register or pop from an empty register sets SF (Stack Fault flag) and causes an invalid operation exception. The recovery routine for the exception must recognize this situation, fix up the stack, then perform the original operation. The recovery routine will not work correctly in the first example shown in the figure. The problem is that the value of COUNT is incremented before the exception handler is invoked, so that the recovery routine will load an incorrect value of COUNT, causing the program to fail or behave unreliably.

3.3.3 PROPER EXCEPTI ON SYNCHRONIZATION IN GENERAL

As explained before (see Section 3.2), if the FPU encounters an unmasked exception condition a software exception handler is invoked *before* execution of the *next* WAIT or floating-point instruction. This is because an unmasked floating-point exception causes the processor to freeze immediately before executing such an instruction (unless the IGNNE# input is active, or it is a “No-Wait” FPU instruction). Exactly when the exception handler will be invoked (in the interval between when the exception is detected and the next WAIT or FPU instruction) is dependent on the processor generation, the system, and which FPU instruction and exception is involved.

To be safe in exception synchronization, one should assume the handler will be invoked at the end of the interval. Thus the program should not change any value that might be needed by the handler (such as COUNT in the above example) until *after* the *next* FPU instruction following an FPU instruction that could cause an error. If the program needs to modify such a value before the next FPU instruction (or if the next FPU instruction could also cause an error), then a WAIT instruction should be inserted before the value is modified. This will force the handling of any exception before the value is modified. A WAIT instruction should also be placed after the last floating-point instruction in an application so that any unmasked exceptions will be serviced before the task completes.

3.4 FPU Exception Handling Examples

There are many approaches to writing exception handlers. One useful technique is to consider the exception handler procedure as consisting of "prologue," "body," and "epilogue" sections of code.

In the transfer of control to the exception handler due to an INTR, NMI, or SMI, external interrupts have been disabled by hardware. The prologue performs all functions that must be protected from possible interruption by higher-priority sources. Typically, this involves saving registers and transferring diagnostic information from the FPU to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler. The standard "prologue" not only saves the registers and transfers diagnostic information from the FPU to memory but also clears the FP exception flags in the status word. Alternatively, when it is not necessary for the handler to be re-entrant, another technique may also be used. In this technique, the exception flags are not cleared in the "prologue" and the body of the handler must not contain any FP instructions that cannot complete execution when there is a pending FP exception. (The “No-Wait” instructions are discussed in Section 6.3.7 of the *Pentium® Processor Family Developer’s Manual,* Volume 3). Note that the handler must still clear the exception flag(s) before executing the IRET. If the exception handler uses neither of these techniques the system will be caught in an endless loop of nested FP exceptions, and hang.

The body of the exception handler examines the diagnostic information and makes a response that is necessarily application-dependent. This response may range from halting execution, to displaying a message, to attempting to repair the problem and proceed with normal execution. The epilogue essentially reverses the actions of the prologue, restoring the processor so that normal execution can be resumed. The epilogue must not load an unmasked exception flag into the FPU or another exception will be requested immediately.

The following code examples show the ASM386 and ASM486 coding of three skeleton exception handlers, with the save spaces given as correct for 32 bit protected mode. They show how prologues and epilogues can be written for various situations, but the application dependent exception handling body is just indicated by comments showing where it should be placed.

The first two are very similar; their only substantial difference is their choice of instructions to save and restore the FPU. The tradeoff here is between the increased diagnostic information provided by FNSAVE and the faster execution of FNSTENV. (Also, after saving the original contents, FNSAVE re-initializes the FPU, while FNSTENV only masks all FPU exceptions.) For applications that are sensitive to interrupt latency or that do not need to examine register contents, FNSTENV reduces the duration of the "critical region," during which the processor does not recognize another interrupt request. (See the *Pentium® Processor Family Developer’s Manual*, Volume 3*,* Section 6.2.1.6 for a complete description of the FPU save image.)

After the exception handler body, the epilogues prepare the processor to resume execution from the point of interruption (i.e., the instruction following the one that generated the unmasked exception). Notice that the exception flags in the memory image that is loaded into the FPU are cleared to zero prior to reloading (in fact, in these examples, the entire status word image is cleared).

Example 1 and Example 2 assume that the exception handler itself will not cause an unmasked exception. Where this is a possibility, the general approach shown in 3 can be employed. The basic technique is to save the full FPU state and then to load a new control word in the prologue. Note that considerable care should be taken when designing an exception handler of this type to prevent the handler from being reentered endlessly.

Example 1. Full-State Exception Handler

```
SAVE_ALL    PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
      PUSH  EBP
      .
      .
      MOV   EBP, ESP
      SUB   ESP, 108 ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE FULL FPU STATE, RESTORE INTERRUPT ENABLE FLAG (IF)
      FNSAVE      [EBP-108]
      PUSH [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
      POPFD       ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED STATE IMAGE
      MOV         BYTE PTR [EBP-104], 0H
      FRSTOR      [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
      MOV         ESP, EBP
      .
      .
      POP         EBP
;
; RETURN TO INTERRUPTED CALCULATION
      IRETD
SAVE_ALL    ENDP
```

Example 2. Reduced-Latency Exception Handler

```
SAVE_ENVIRONMENT  PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU ENVIRONMENT
      PUSH        EBP
      .
      .
      MOV         EBP, ESP
      SUB         ESP, 28 ; ALLOCATES 28 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE ENVIRONMENT, RESTORE INTERRUPT ENABLE FLAG (IF)
      FNSTENV     [EBP-28]
      PUSH [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
      POPFD       ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED ENVIRONMENT IMAGE
      MOV         BYTE PTR [EBP-24], 0H
      FLDENV      [EBP-28]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
      MOV         ESP, EBP
      .
      .
      POP         EBP
;
; RETURN TO INTERRUPTED CALCULATION
      IRETD
SAVE_ENVIRONMENT  ENDP
```

Example 3. Reentrant Exception Handler

```
LOCAL_CONTROL DW ?      ; ASSUME INITIALIZED
      .
      .
REENTRANT   PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
```

```
      PUSH  EBP
      .
      .
      MOV          EBP, ESP
      SUB          ESP, 108 ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)

; SAVE STATE, LOAD NEW CONTROL WORD, RESTORE INTERRUPT ENABLE FLAG (IF)
      FNSAVE       [EBP-108]
      FLDCW        LOCAL_CONTROL
      PUSH [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
      POPFD        ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

      .
      .
;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE. AN UNMASKED EXCEPTION
; GENERATED HERE WILL CAUSE THE EXCEPTION HANDLER TO BE REENTERED.
; IF LOCAL STORAGE IS NEEDED, IT MUST BE ALLOCATED ON THE STACK.
;
      .
      .
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED STATE IMAGE
      MOV          BYTE PTR [EBP-104], 0H
      FRSTOR       [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
      MOV          ESP, EBP
      .
      .
      POP          EBP
;
; RETURN TO POINT OF INTERRUPTION
      IRETD
REENTRANT   ENDP
```

3.5 Need for Preserving the State of IGNNE# Circuit if Use FPU and SMM

In Section 2.3.2 the recommended circuit (Figure 2) for MS-DOS compatible FPU exception handling for Intel486 processors and beyond contains two flip flops. When the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. The assertion of IGNNE# may be used by the handler if needed to execute any FPU instruction while ignoring the pending FPU errors. The problem here is that the state of Flip Flop #2 is effectively an additional (but hidden) status bit that can affect processor behavior, and so ideally should be saved upon entering SMM, and restored before resuming to normal operation. If this is not done, and also the SMM code saves the FPU state, AND an FPU error handler is being used which relies on IGNNE# assertion, then (very rarely) the FPU handler will nest inside itself and malfunction. The following example shows how this can happen.

The problem will only occur if the processor enters SMM between the OUT and the FLDCW instructions. But if that happens, AND the SMM code saves the FPU state using FNSAVE, then the IGNNE# Flip Flop will be cleared (because FNSAVE clears the FPU errors and thus de-asserts FERR#). When the processor returns from SMM it will restore the FPU state with FRSTOR, which will re-assert FERR#, but the IGNNE# Flip Flop will not get set. Then when the FPU error handler executes the FLDCW instruction, the active error condition will cause the processor to re-enter the FPU error handler from the beginning. This may cause the handler to malfunction.

Note that NMI (or any interrupt through INTR that is enabled inside the FPU exception handler) will cause this same problem, if the interrupt routine saves and restores the FPU state, and it happens to occur between the OUT and the FLDCW instructions. SMI is the main focus here because it is much more likely to invoke FNSAVE/FRSTOR than other interrupts because of 0V suspend (see below). The problem can easily be eliminated from all interrupts besides SMI and NMI by not enabling INTR inside the FPU exception handler.

To avoid this problem, Intel recommends two measures:

1. Do not use the FPU for calculations inside SMM code (or code for NMI, or any other interrupts enabled inside the FPU exception handler). (The normal power management, and sometimes security, functions provided by SMM have no need for FPU calculations; if they are needed for some special case, use scaling or emulation instead.) This eliminates the need to do FNSAVE/FRSTOR inside SMM code, *except* when going into an 0V suspend state (in which, in order to save power, the processor is turned off completely, requiring its complete state to be saved).
2. The system should not call upon SMM code to put the processor into 0V suspend while the processor is running FPU calculations, or just after an interrupt has occurred. Normal power management protocol avoids this by going into power down states only after timed intervals in which no system activity occurs.

3.6 Considerations When FPU Shared Between Tasks

The Intel Architecture allows speculative deferral of floating-point state swaps on task switches. This feature allows postponing an FPU state swap until an FPU instruction is actually encountered in another task. Since kernel tasks rarely use floating-point, and some applications do not use floating-point or use it infrequently, the amount of time saved by avoiding unnecessary stores of the floating-point state is significant. Speculative deferral of FPU saves does, however, place an extra burden on the kernel in three key ways:

1. The kernel must keep track of which thread owns the FPU, which may be different from the currently executing thread.
2. The kernel must associate any floating-point exceptions with the generating task. This requires special handling since floating-point exceptions are delivered asynchronous with other system activity.
3. There are conditions under which spurious floating-point exception interrupts are generated, which the kernel must recognize and discard.

Suppose that the FPU exception handler includes the following sequence:

```
FNSTSW save_sw         ; save the FPU status word using a "No-Wait" FPU instruction

OUT 0F0H, AL           ; clears IRQ13 & activates IGNNE#

 . . . .

FLDCW new_cw  ; loads new CW ignoring FPU errors, since IGNNE# is assumed active; or any
                       ; other FPU instruction that is not a "No-Wait" type will cause the
same problem

  . . . .

FCLEX                  ; clear the FPU error conditions & thus turn off FERR# & reset the
IGNNE# FF
```

3.6.1 SPECULATIVELY DEFERRING FPU SAVES, GENERAL OVERVIEW

In order to support multi-tasking, each thread in the system needs a save area for the general purpose registers, and each task that is allowed to use floating-point needs an FPU save area large enough to hold the entire FPU stack and associated FPU state such as the control word and status word. (See the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 6.2.1.6 for a complete description of the FPU save image.)

On a task switch, the general purpose registers are swapped out to their save area for the suspending thread, and the registers of the resuming thread are loaded. The FPU state does not need to be saved at this point. If the resuming thread does not use the FPU before it is itself suspended, then both a save and a load of the FPU state has been avoided. It is often the case that several threads may be executed without any usage of the FPU.

The processor supports speculative deferral of FPU saves via interrupt 7 "Device Not Available" (DNA), used in conjunction with CR0 bit 3, the "Task Switched" bit (TS). (See the *Pentium® Processor Family Developer's Manual,* Volume 3, Sections 10.1.3 & 14.9.7) Every task switch via the hardware supported task switching mechanism (see Section 13.5 of the *Pentium® Processor Family Developer's Manual,* Volume 3) sets TS. Multi-threaded kernels that use software task switching[4] can set the TS bit by reading CR0, ORing a '1' into bit 3, and writing back CR0[5]. Any subsequent floating-point instructions (now being executed in a new thread context) will fault via interrupt 7 before execution.

Footnotes

[4]In a software task switch, the operating system uses a sequence of instructions to save the suspending thread's state and restore the resuming thread's state instead of the single long, noninterruptable task switch operation provided by the Intel Architecture.

[5] Although CR0, bit 2, the emulation flag (EM), also causes a DNA exception, *do not* use the EM bit as a surrogate for TS. EM means that no floating-point unit is available and that FP instructions must be emulated. Using EM to trap on task switches is not compatible with Intel Architecture MMX™ Technology. If the EM flag is set, MMX instructions raise the invalid opcode exception.

This allows the DNA handler to save the old floating-point context and reload the FPU state for the current thread. The handler should clear the TS bit before exit using the CLTS instruction. On return from the handler the faulting thread will proceed with its floating-point computation.

Some operating systems save the FPU context on every task switch, typically because they also change the linear address space between tasks. The problem and its solution discussed below apply to these operating systems also.

3.6.2 TRACKING FPU OWNERSHIP

Since the contents of the FPU may not belong to the currently executing thread, the thread identifier for the last FPU user needs to be tracked separately. This is not complicated -- the kernel should simply provide a variable to store the thread identifier of the FPU owner, separate from the variable that stores the identifier for the currently executing thread. This variable is updated in the DNA exception handler, and is used by the DNA exception handler to find the FPU save areas of the old and new threads. A simplified flow for a DNA exception handler is then:

1. Use the 'FPU Owner' variable to find the FPU save area of the last thread to use the FPU.
2. Save the FPU contents to the old thread's save area, typically using an FNSAVE instruction.
3. Set the 'FPU Owner' variable to the identify the currently executing thread.
4. Reload the FPU contents from the new thread's save area, typically using an FRSTOR instruction.
5. Clear TS using the CLTS instruction and exit the DNA exception handler.

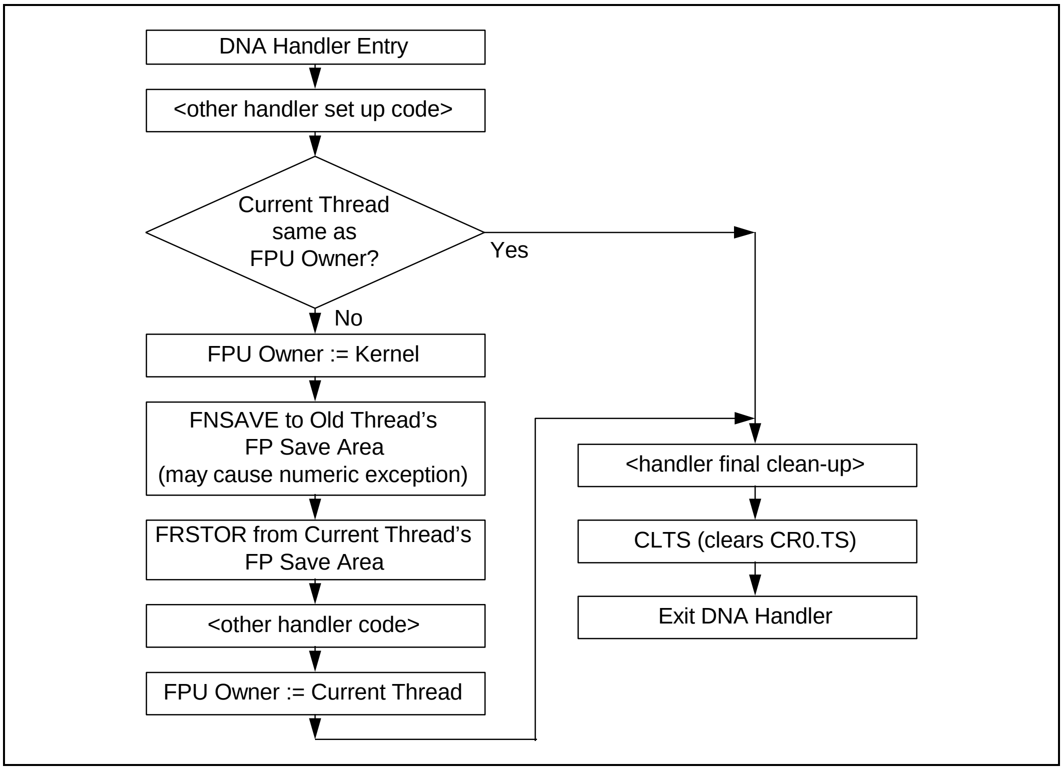
While this flow covers the basic requirements for speculatively deferred FPU state swaps, there are some additional subtleties that need to be handled in a robust implementation.

3.6.3 INTERACTION OF FPU STATE SAVES AND FP EXCEPTION ASSOCIATION

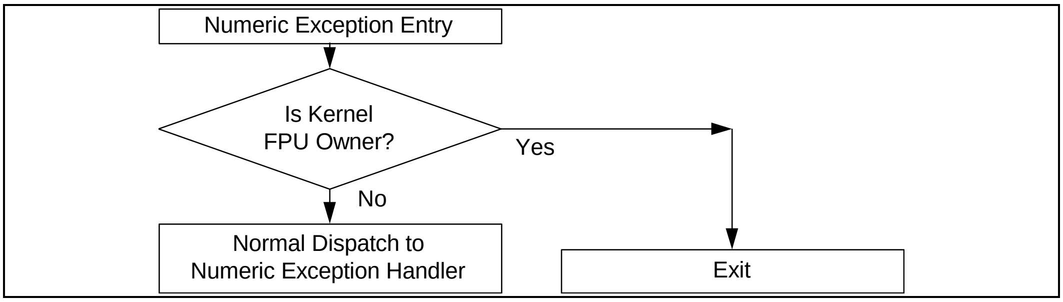
Recall these key points from earlier in this document: When considering FP exceptions across all implementations of the Intel Architecture, and across all FP instructions, an FP exception can be

initiated from any time during the excepting FP instruction, up to just before the next FP instruction. The 'next' FP instruction may be the FNSAVE used to save the FPU state for a task switch. In the case of "no-wait:" instructions such as FNSAVE, the interrupt from a previously excepting instruction (NE=0 case) may arrive just before the "no-wait" instruction, during, or shortly thereafter with a system dependent delay. Note that this implies that an FP exception might be registered during the state swap process itself, and the kernel and FP exception interrupt handler must be prepared for this case.

A simple way to handle the case of exceptions arriving during FPU state swaps is to allow the kernel to be one of the FPU owning threads. A reserved thread identifier is used to indicate kernel ownership of the FPU. During an FP state swap, the 'FPU owner' variable should be set to indicate the kernel as the current owner. At the completion of the state swap, the variable should be set to indicate the new owning thread. The numeric exception handler needs to check the FPU owner and *discard* any numeric exceptions that occur while the kernel is the FPU owner. A more general flow for a DNA exception handler that handles this case is shown next:


DNA Handler Entry
<other handler set up code>
Current Thread same as FPU Owner?
Yes
No
FPU Owner := Kernel
FNSAVE to Old Thread's FP Save Area (may cause numeric exception)
FRSTOR from Current Thread's FP Save Area
<other handler code>
FPU Owner := Current Thread
<handler final clean-up>
CLTS (clears CR0.TS)
Exit DNA Handler

Numeric exceptions received while the kernel owns the FPU for a state swap must be discarded in the kernel without being dispatched to a handler. A flow for a numeric exception dispatch routine is shown below:



It may at first glance seem that there is a possibility of FP exceptions being lost because of exceptions that are discarded during state swaps. This is not the case, as the exception will be re-issued when the FP state is reloaded. Walking through state swaps both with and without pending numeric exceptions will clarify the operation of these two handlers.

Case 1: FPU State Swap Without Numeric Exception

Assume two threads 'A' and 'B', both using the floating-point unit. Let A be the thread to have most recently executed a FP instruction, with no pending numeric exceptions. Let B be the currently executing thread. CR0.TS was set when thread A was suspended. When B starts to execute a FP instruction the instruction will fault with the DNA exception because TS is set.

At this point the handler is entered, and eventually it finds that the current FPU Owner is not the currently executing thread. To guard the FPU state swap from extraneous numeric exceptions, the FPU Owner is set to be the kernel. The old owner's FPU state is saved with FNSAVE, and the current thread's FPU state is restored with FRSTOR. Before exiting, the FPU owner is set to thread B, and the TS bit is cleared.

On exit, thread B resumes execution of the faulting FP instruction and continues.

Case 2: FPU State Swap with Discarded Numeric Exception

Again, assume two threads 'A' and 'B', both using the floating-point unit. Let A be the thread to have most recently executed a FP instruction, but this time let there be a pending numeric exception. Let B be the currently executing thread. When B starts to execute a FP instruction the instruction will fault with the DNA exception and enter the DNA handler. (If both numeric and DNA exceptions are pending, the DNA exception takes precedence, in order to support handling the numeric exception in its own context.)

When the FNSAVE starts, it will trigger an interrupt via FERR# because of the pending numeric exception. After some system dependent delay, the numeric exception handler is entered. It may be entered before the FNSAVE starts to execute, or it may be entered shortly after execution of the FNSAVE. Since the FPU Owner is the kernel, the numeric exception handler simply exits, discarding the exception. The DNA handler resumes execution, completing the FNSAVE of the old FP context of thread A and the FRSTOR of the FP context for thread B.

Thread A eventually gets an opportunity to handle the exception that was discarded during the task switch. After some time, thread B is suspended, and thread A resumes execution. When thread A starts to execute an FP instruction, once again the DNA exception handler is entered. B's FPU state is FNSAVE'ed, and A's FPU state is FRSTOR'ed. Note that in restoring the FPU state from A's save area, the pending numeric exception flags are reloaded in to the FP status word. Now when the DNA exception handler returns, thread A resumes execution of the faulting FP instruction just long enough to immediately generate a numeric exception, which now gets handled in the normal way. The net result is that the task switch and resulting FPU state swap via the DNA exception handler causes an 'extra' numeric exception which can be safely discarded.

3.6.4 INTERRUPT ROUTING FROM THE KERNEL

In MS-DOS, an application that wishes to handle numeric exceptions hooks interrupt 2 by placing its handler address in the interrupt vector table, and exiting via a jump to the previous interrupt 2 handler. Protected mode systems that run MS-DOS programs under a subsystem can emulate this exception delivery mechanism. For example, assume a protected mode O.S. that runs with CR.NE = 1, and that runs MS-DOS programs in a virtual machine subsystem. The MS-DOS program is set up in a virtual machine that provides a virtualized interrupt table. The MS-DOS application hooks interrupt 2 in the virtual machine in the normal way. A numeric exception will trap to the kernel via the real INT 16 residing in the kernel at ring 0. The INT 16 handler in the kernel then locates the correct MS-DOS virtual machine, and reflects the interrupt to the virtual machine monitor. The virtual machine monitor then emulates an interrupt by jumping through the address in the virtualized interrupt table, eventually reaching the application's numeric exception handler.

4.0 DIFFERENCES FOR HANDLERS USING NATIVE MODE

The 8087 has a pin INT which it asserts when an unmasked exception occurs. But there is no interrupt input pin in the 8086 or 8088 dedicated to its attachment, nor an interrupt vector number in the 8086 or 8088 specific for an FPU error assertion. But beginning with the Intel 80286 and 80287, hardware connections were dedicated to support the FPU exception, and interrupt vector 16 assigned to it.

4.1 Origin with 80286 and 80287; Intel386™ Processor and Intel387 Math Coprocessor

The 80286 and 80287 and Intel386 processor and Intel387 math coprocessor pairs are each provided with ERROR# pins that are recommended to be connected between the processor and FPU. If this is done, when an unmasked FPU exception occurs, the FPU records the exception, and asserts its ERROR# pin. The processor recognizes this active condition of the ERROR# status line immediately before execution of the next WAIT or FPU instruction (except for the “N0-Wait” type) in its instruction stream, and branches to the routine at interrupt vector 16. Thus an FPU exception will be handled before any other FPU instruction (after the one causing the error) is executed (except for “No-Wait” instructions, which will be executed without triggering the FPU exception interrupt, but it will remain pending).

Using the dedicated interrupt 16 for FPU exception handling is referred to as the native mode. It is the simplest approach, and the one recommended most highly by Intel.

4.2 Changes with Intel486Ô , PentiumÒ and Pentium Pro Processors with CR0.NE=1

With these latest three generations of the Intel Architecture, more enhancements and speedup features have been added to the corresponding FPUs. Also, the FPU is now built into the same chip as the processor, which allows further increases in the speed at which the FPU can operate as part of the integrated system. This also means that the native mode of FPU exception handling, selected by setting bit NE of register CR0 to 1, is now entirely internal.

If an unmasked exception occurs during an FPU instruction, the FPU records the exception internally, and triggers the exception handler through interrupt 16 immediately before execution of the next WAIT or FPU instruction (except for “No-Wait” instructions, which will be executed as described in Section 4.1 above).

An unmasked numerical exception causes the FERR# output to be activated even with NE=1, and at exactly the same point in the program flow as it would have been asserted if NE were zero. However, the system would not connect FERR# to a PIC to generate INTR when operating in the native, internal mode. (If the hardware of a system has FERR# connected to trigger IRQ13 in order to support MS-DOS, but an OS using the native mode is actually running the system, it is the OSs responsibility to make sure that IRQ13 is not enabled in the slave PIC.) With this configuration a system is immune to the problem discussed in Section 2.3.3, where for Intel486 and Pentium processors a “No-Wait” FPU instruction can get an FPU exception.

4.3 Considerations When FPU Shared Between Tasks Using Native Mode

The protocols recommended in Section 3.6 for MS-DOS compatible FPU exception handlers that are shared between tasks may be used without change with the native mode. However, the protocols for a handler written specifically for native mode can be simplified, because the problem of a spurious floating-point exception interrupt occurring while the kernel is executing cannot happen in native mode.

The problem as actually found in practical code in a MS-DOS compatible system happens when the DNA handler uses FNSAVE to switch FPU contexts. If an FPU exception is active, then FNSAVE triggers FERR# briefly, which usually will cause the FPU exception handler to be invoked inside the DNA handler. In native mode, neither FNSAVE nor any other “No-Wait” instructions can trigger interrupt 16. (As discussed above, FERR# gets asserted independent of the value of the NE bit, but when NE=1, the OS should not enable its path through the PIC.) Another possible (very rare) way a floating-point exception interrupt could occur while the kernel is executing is by an FPU immediate exception case having its interrupt delayed by the external hardware until execution has switched to the kernel. This also cannot happen in native mode because there is no delay through external hardware.

Thus the native mode FPU exception handler can omit the test to see if the kernel is the FPU owner, and the DNA handler for a native mode system can omit the step of setting the kernel as the FPU owner at the handler's beginning. Since however these simplifications are minor and save little code, it would be a reasonable and conservative habit (as long as the MS-DOS compatible mode is widely used) to include these steps in all systems.

Note that the special DP (Dual Processing) mode for Pentium processors, and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors, support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatible FPU mode for systems using more than one processor.