INITIALIZATION

CONTENTS

Introduction

- After a signal on the RESET pin, certain registers of the 80386 are set to predefined values.
- These values are adequate to enable execution of a bootstrap program, But additional initialization must be performed by software before all the features of the processor can be utilized.
- After reset initialization, the DH register holds a number which identifies the processor type.

PROCESSOR STATE AFTER RESET

- A self-test may be requested at power-up. The self-test is requested by asserting the signal on the BUSY # pin during the falling edge of the RESET# signal.
- Reset initialization takes 350 to 450 clock periods. If the self-test is selected, it takes about 220 clock periods. For a 16 MHz processor, this takes about 33 milliseconds.
- The EAX register is clear if the 386 DX microprocessor passed the test.
- A non-zero value in the EAX register after self-test indicates the processor is faulty.
- If the self-test is not requested, the contents of the EAX register after reset initialization are undefined (possibly non-zero). The DX register holds a component identifier and revision number after reset initialization, as shown in Figure 10-1.
- The DH register contains the value 3, which indicates a 386 DX microprocessor. The DL register contains a unique identifier of the revision level.

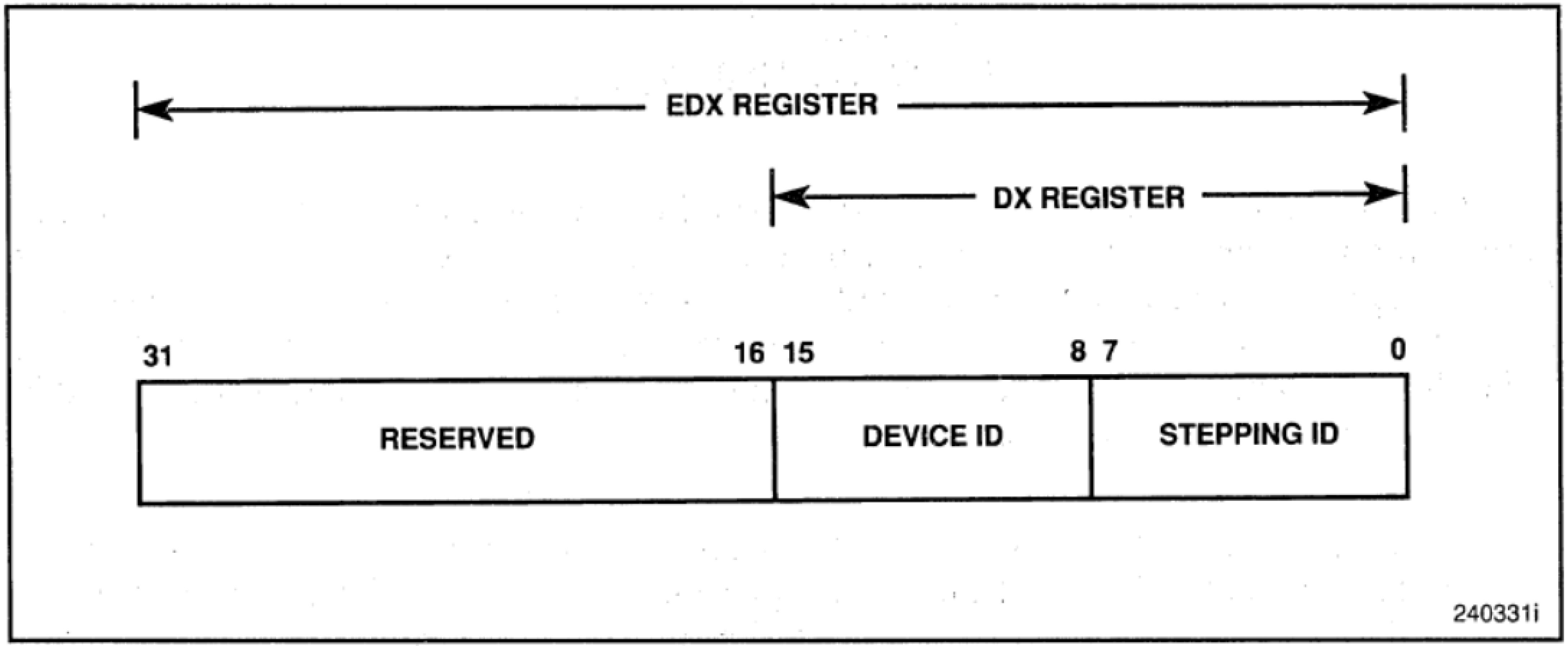


Figure 10-1. Contents of the EDX Register after Reset

- The state of the CRO register following power-up is shown in Figure 10-2. These states put the processor into real-address mode with paging disabled.

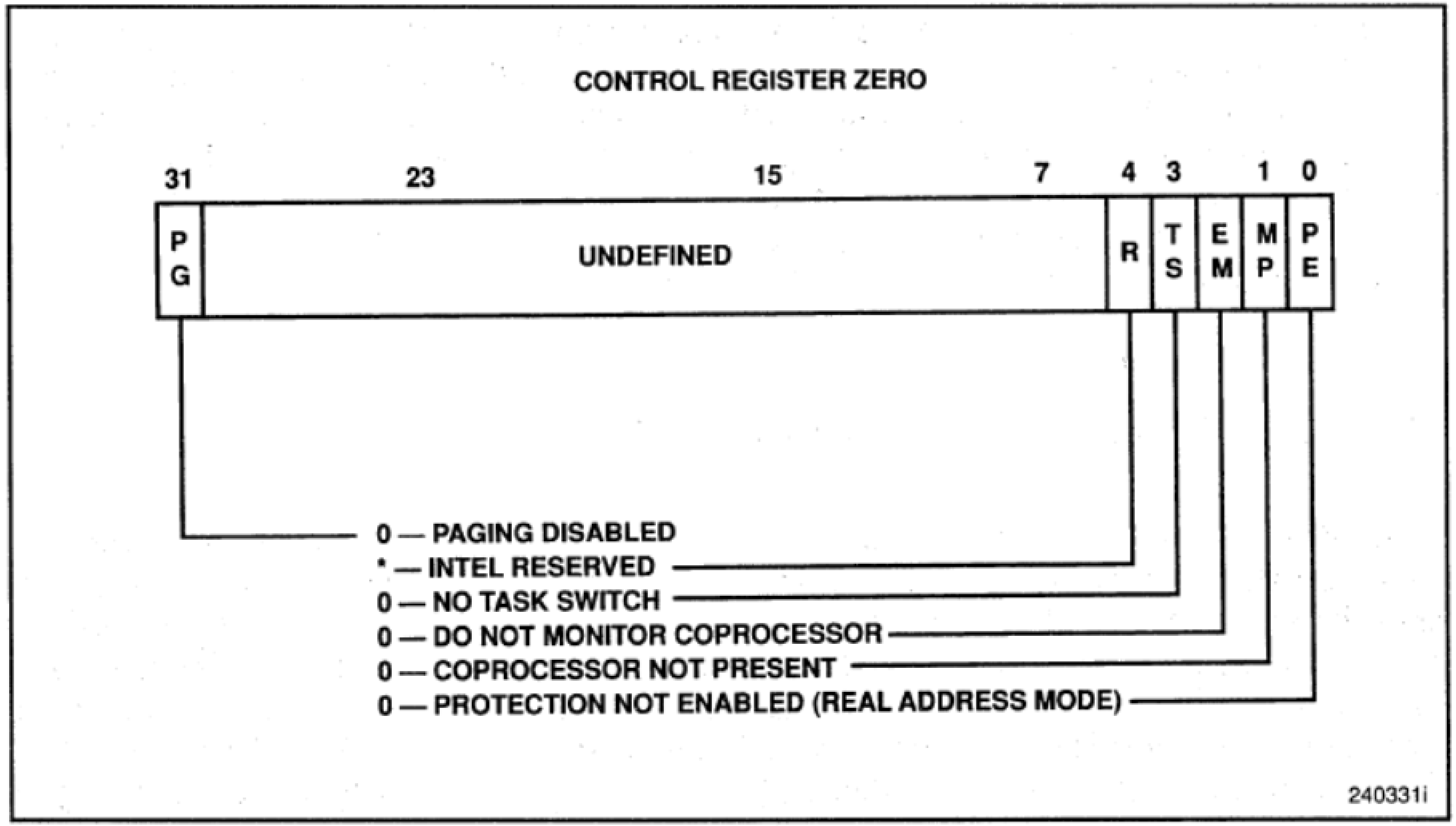
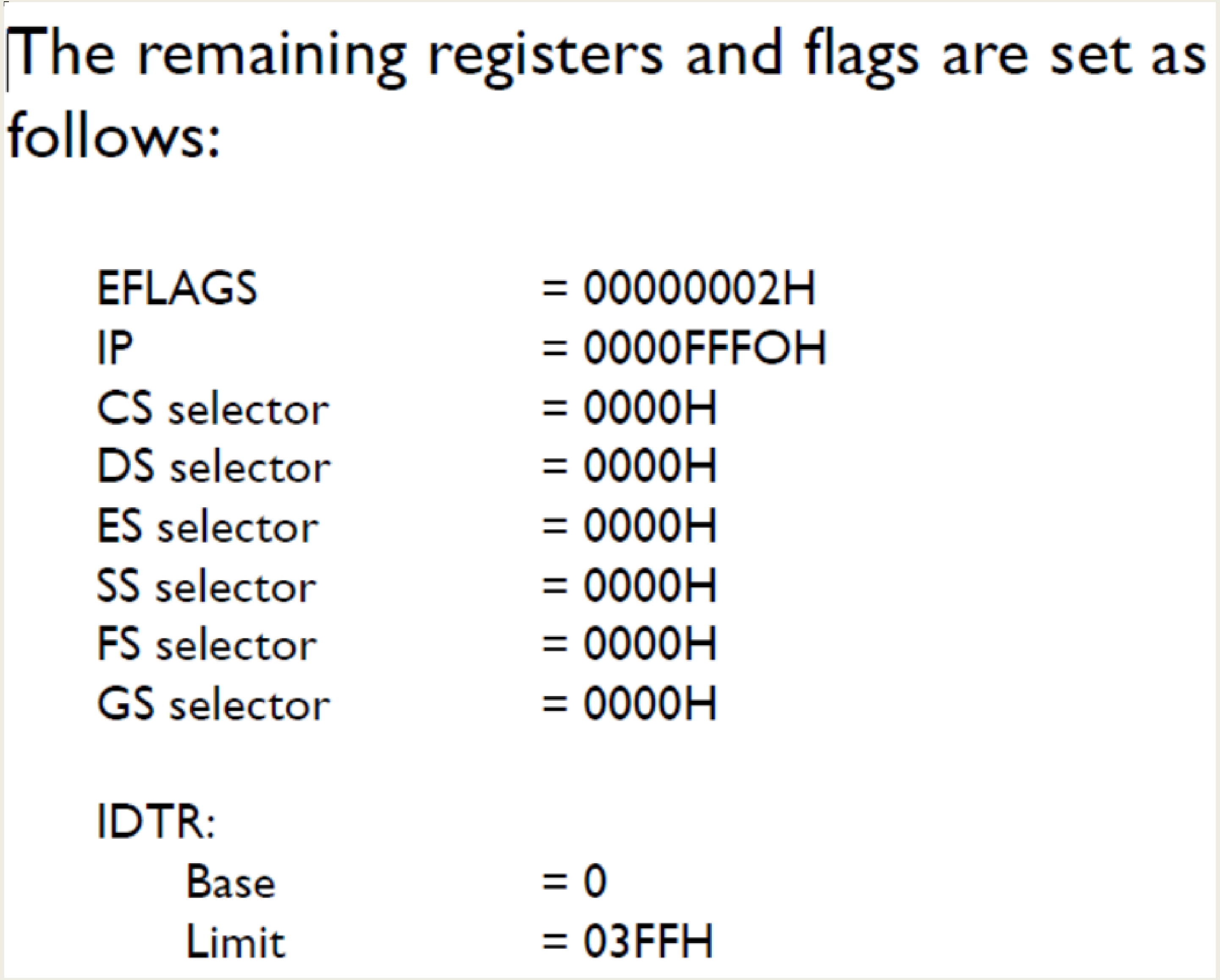


Figure 10-2. Contents of the CR0 Register after Reset

- The state of the EBX, ECX, ESI, EDI, EBP, ESP, GDTR, LDTR, TR, and debug registers is undefined following power-up.



The remaining registers and flags are set as follows:

EFLAGS	= 00000002H
IP	= 0000FFFOH
CS selector	= 0000H
DS selector	= 0000H
ES selector	= 0000H
SS selector	= 0000H
FS selector	= 0000H
GS selector	= 0000H
IDTR:	
Base	= 0
Limit	= 03FFH

SOFTWARE INITIALIZATION IN REAL-ADDRESS MODE

- After reset initialization, software sets up data structures needed for the processor to perform basic system functions, such as handling interrupts.
- If the processor remains in real-address mode, software sets up data structures in the form used by the 8086 processor.
- If the processor is going to operate in protected mode, software sets up data structures in the form used by the 80286 and 386 DX microprocessors, then switches modes.

System Tables

- In real-address mode, no descriptor tables are used.
- The interrupt vector table, which starts at address 0, needs to be loaded with pointers to exception and interrupt handlers before interrupts can be enabled.
- The NMI interrupt is always enabled.
- If the interrupt vector table and the NMI interrupt handler need to be loaded into RAM, there will be a period of time following reset initialization when an NMI interrupt cannot be handled.

NMI Interrupt

- Hardware must provide a mechanism to prevent an NMI interrupt from being generated while software is unable to handle it.
- For example, the interrupt vector table and NMI interrupt handler can be provided in ROM. This allows an NMI interrupt to be handled immediately after reset initialization.
- Another solution would be to provide a mechanism which passes the NMI signal through an AND gate controlled by a bit in an I/O port.
- Hardware can clear the bit when the processor is reset, and software can set the bit when it is ready to handle NMI interrupts.

First Instruction

- Execution begins with the instruction addressed by the initial contents of the CS and IP registers.
- To allow the initialization software to be placed in a ROM at the top of the address space, the high 12 bits of addresses issued for the code segment are set, until the first instruction which loads the CS register, such as a far jump or call.
- Because the size of the ROM is unknown, the first instruction is intended to be a jump to the beginning of the initialization software.

SWITCHING TO PROTECTED MODE

System Tables

- To allow protected mode software to access programs and data, at least one descriptor table, the GDT, and two descriptors must be created.
- Descriptors are needed for a code segment and a data segment.
- The stack can be be placed in a normal read/write data segment, so no descriptor for the stack is required. Before the GDT can be used, the base address and limit for the GDT must be loaded into the GDTR register using an LGDT instruction.

NMI Interrupt

- If hardware· allows NMI interrupts to be generated, the IDT and a gate for the NMI interrupt handler need to be created.
- Before the IDT can be used, the base address and limit for the IDT must be loaded into the IDTR register using an LIDT instruction.

PE Bit

- Protected mode is entered by setting the PE bit in the CRO register. Either an LMSW or MOV CRO instruction maybe. used to set this bit (the MSW register is part of the CRO register).

SOFTWARE INITIALIZATION IN PROTECTED MODE

- The data structures needed in protected mode are determined by the memory management features which are used.
- The processor supports segmentation models which range from a single, uniform address space (flat model) to a highly structured model with several independent, protected address spaces for each task (multi segmented model).
- Paging can be enabled for allowing access to large data structures which are partly in memory and partly on disk.
- Both of these forms of address translation require data structures which are setup by the operating system and used by the memory management hardware.

Segmentation

- A flat model without paging only requires a GDT with one code and one data segment descriptor.
- A flat model with paging requires code and data descriptors for supervisor mode and another set of code and data descriptors for user mode.
- A multisegmented model may require additional segments for the operating system, as well as segments and LDTs for each application program.
- LDTs require segment descriptors in the GDT.

Paging

- Paging is controlled by a mode bit. If the PG bit in the CRO register is clear (its state following reset initialization), the paging mechanism is completely absent from the processor architecture seen by programmers,
- If the PG bit is set, paging is enabled. The bit may be set using a MOV CRO instruction.
- Before setting the PG bit, the following conditions must be true:
  - *Software has created at least two page tables, the page directory and at least one second-level page table.*
  - *The PDBR register (same as the CR3 register) is loaded with the base address of the page directory.*
  - *The processor is in protected mode (paging is not available in real-address mode). If all other restrictions are met, the PO and PE bits can be set at the same time.*

Tasks

- If the multitasking mechanism is not used, it is unnecessary to initialize the TR register.
- If the multitasking mechanism is used, a TSS and a TSS descriptor for the initialization software must be created.
- TSS descriptors must not be marked as busy when they are created; TSS descriptors should be marked as busy only as a side-effect of performing a task switch.
- As with descriptors for LDTs, TSS descriptors reside in the GDT.
- The LTR instruction is used to load a selector for the TSS descriptor of the initialization software into the TR register. This instruction marks the TSS descriptor as busy, but does not perform a task switch.
- The selector must be loaded before performing the first task switch, because a task switch copies the current task state into the TSS.
- After the LTR instruction has been used, further operations on the TR register are performed by task switching.

DEBUGGING

CONTENTS

Introduction

- Debugging is the process of identifying and removing bug from software or program. It refers to identification of errors in the program logic, machine codes, and execution. It gives step by step information about the execution of code to identify the fault in the program.
- The debugging features of the 386 DX microprocessor give the system programmer valuable tools for looking at the dynamic state of the processor.
- The debugging support is accessed through the debug registers. They hold the addresses of memory locations, called breakpoints, which invoke debugging software.

DEBUGGING SUPPORT

The features of the 80386 architecture which support debugging are:

- **Reserved debug interrupt vector**
  - Specifies a procedure or task to be called when an event for the debugger occurs.
- **Debug address registers**
  - Specifies the addresses of up to four breakpoints.
- **Debug control register**
  - Specifies the forms of memory access for the breakpoints.
- **Debug status register**
  - Reports conditions which were in effect at the time of the exception.
- **Trap bit of TSS (T-bit)**
  - Generates a debug exception when an attempt is made to perform a task switch to a task with this bit set in its TSS.
- **Resume flag (RF)**
  - Suppresses multiple exceptions to the same instruction.

- **Trap flag (TF)**
  - Generates a debug exception after every execution of an instruction.
- **Breakpoint instruction**
  - Calls the debugger (generates a debug exception). This instruction is an alternative way to set code breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.
- **Reserved interrupt vector for breakpoint exception**
  - Calls a procedure or task when a breakpoint instruction is executed.

- The following conditions can be used to call the debugger:
  - Task switch to a specific task.
  - Execution of the breakpoint instruction.
  - Execution of any instruction.
  - Execution of an instruction at a specified address.
  - Read or write of a byte, word, or doubleword at a specified address.
  - Write to a byte, word, or doubleword at a specified address.
  - Attempt to change the contents of a debug register.

DEBUG REGISTERS

- Six registers are used to control debugging. These registers are accessed by forms of the MOV instruction.
- A debug register may be the source or destination operand for one of these instructions. The debug registers are privileged resources; the MOV instructions which access them may be executed only at privilege level 0.
- **Debug Address Registers (DR0-DR3)**
  - Each of these registers holds the linear address for one of the four breakpoints. If paging is enabled, these addresses are translated to physical addresses by the paging algorithm. Each breakpoint condition is specified further by the contents of the DR7 register.
- Debug Control Register (DR7)
  - *The debug control register shown in Figure 12-1 specifies the sort of memory access associated with each breakpoint.*
  - *Each address in registers DRO to DR3 corresponds to a field R/W0 to R/W3 in the DR7 register.*
  - *The processor interprets these bits as follows:*
    - 00 - Break on instruction execution only
    - 01 - Break on data writes only
    - 10-undefined
    - 11-Break on data reads or writes but not instruction fetches

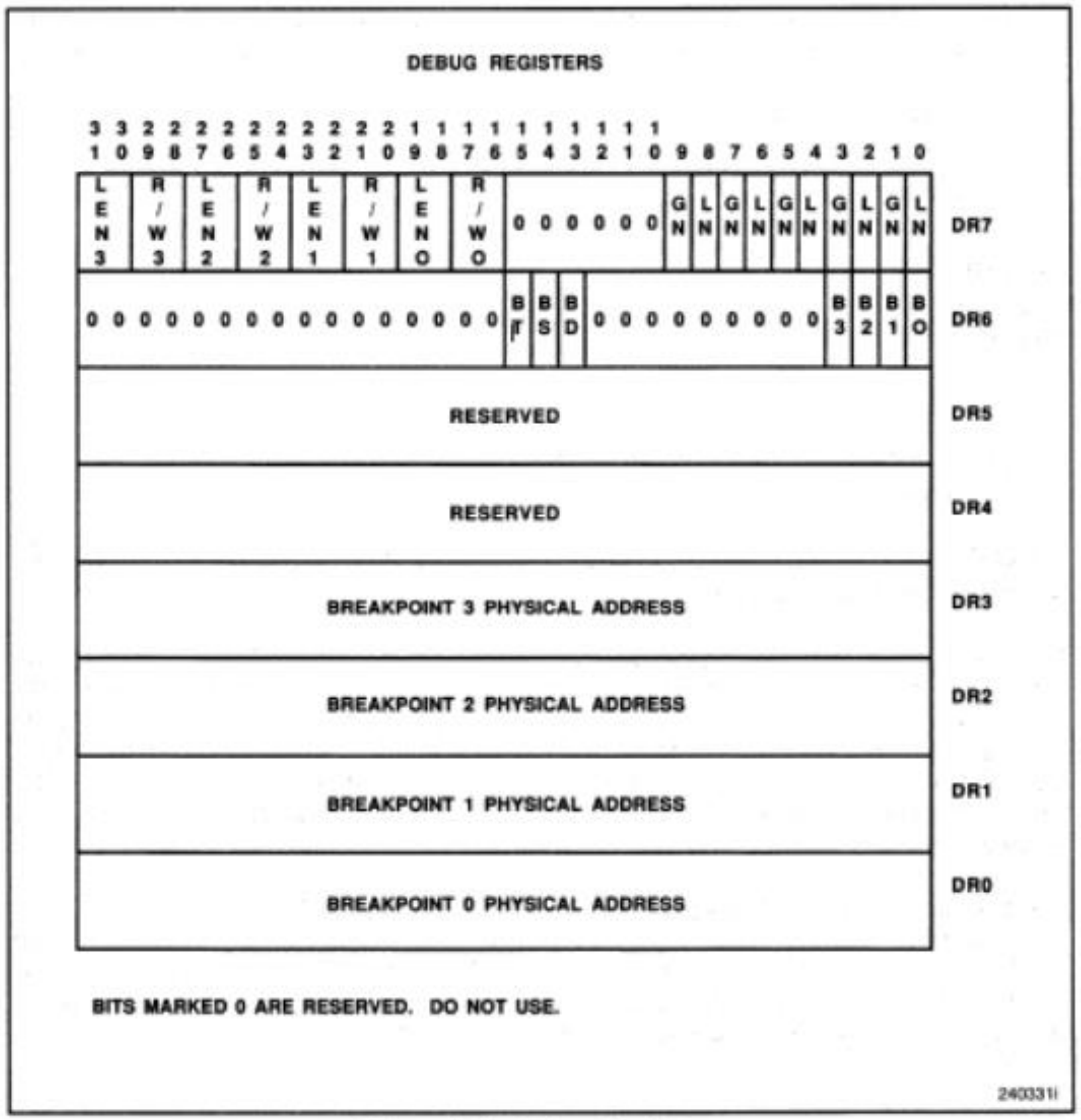


Figure 12-1. Debug Registers

- The LEN0 to LEN3 fields in the DR7 register specify the size of the breakpointed location in memory.
- A size of 1, 2, or 4 bytes may be specified.
- The length fields are interpreted as follows:
  - 00 - one-byte length
  - 01 - two-byte length
  - 10 - *undefined*
  - 11 - four-byte length
- If RWn is 00 (instruction execution), then LENn should also be 00. The effect of using any other length is undefined.
- The low eight bits of the DR7 register (fields LO to L3 and GO to G3) individually enable the four address breakpoint condition,s.
- There are two levels of enabling: the local (L0 through L3) and global (G0 through G3) levels.

Debug Status Register (DR6)

- *The debug status register shown in Figure 12-1 reports conditions sampled at the time the debug exception was generated. Among other information, it reports which breakpoint triggered the exception.*

- When an enabled breakpoint generates a debug exception, it loads the low four bits of this register (B0 through B3) before entering the debug exception handler.
- The B bit is set if the condition described by the DR, LEN, and R/W bits is true, even if the breakpoint is not enabled by the L and G bits.
- The BT bit is associated with the T bit (debug trap bit) of the TSS.
- The BS bit is associated with the TF flag. The BS bit is set if the debug exception was triggered by the single-step execution mode (TF flag set).
- The BD bit is set if the next instruction will read or write one of the eight debug registers while they are being used by in-circuit emulation.

DEBUG EXCEPTIONS

- Two of the interrupt vectors of the 386 DX microprocessor are reserved for debug exceptions. The **debug exception** is the usual way to invoke debuggers designed for the 386 DX microprocessor; the **breakpoint exception** is intended for putting breakpoints in debuggers.

Interrupt 1 - Debug Exceptions

- The handler for this exception usually is a debugger or part of a debugging system.
- The processor generates a debug exception for any of several conditions. The debugger can check flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions also might apply.
- Table 12-2 shows the states of these bits for each kind of breakpoint condition.

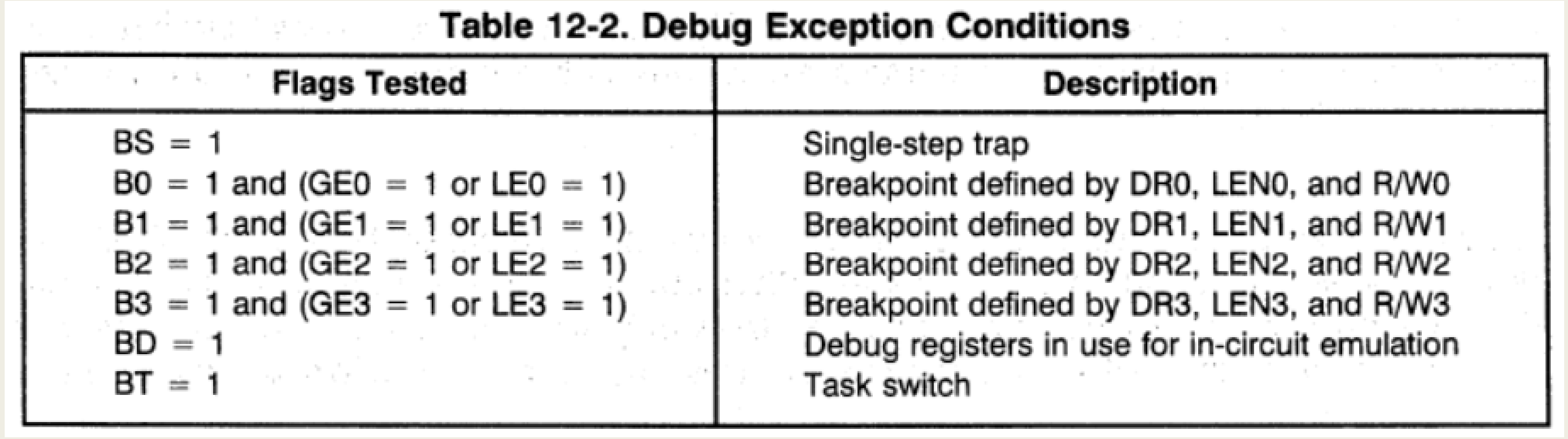


Table 12-2. Debug Exception Conditions

Flags Tested	Description
BS = 1	Single-step trap
B0 = 1 and (GE0 = 1 or LE0 = 1)	Breakpoint defined by DR0, LEN0, and R/W0
B1 = 1 and (GE1 = 1 or LE1 = 1)	Breakpoint defined by DR1, LEN1, and R/W1
B2 = 1 and (GE2 = 1 or LE2 = 1)	Breakpoint defined by DR2, LEN2, and R/W2
B3 = 1 and (GE3 = 1 or LE3 = 1)	Breakpoint defined by DR3, LEN3, and R/W3
BD = 1	Debug registers in use for in-circuit emulation
BT = 1	Task switch

Breakpoint Exception

Interrupt 3 - Breakpoint Instruction

- The breakpoint trap is caused by execution of the INT 3 instruction.
- Aa debugger prepares a breakpoint by replacing the first opcode byte of an instruction with the opcode for the breakpoint instruction.
- When execution of the INT 3 instruction calls the exception handler, the return address points to the first byte of the instruction following the INT 3 instruction.
- The breakpoint exception still is useful for breakpointing debuggers, because the breakpoint exception can call an exception handler other than itself.
- The breakpoint exception also can be useful when it is necessary to set a greater number of breakpoints than permitted by the debug registers, or when breakpoints are being placed in the source code of a program under development.

VIRTUAL 8086 MODE

CONTENTS

INTRODUCTION

- The 386™ DX microprocessor supports execution of one or more 8086, 8088, 80186, or 80188 programs in a 386 DX microprocessor protected-mode environment.
- An 8086 program runs in this environment as part of a virtual-8086 task.
- Virtual-8086 tasks take advantage of the hardware support of multitasking offered by the protected mode.
- The purpose of a virtual-8086 task is to form a "virtual machine" for running programs.

EXECUTING 8086 PROCESSOR CODE

- The processor runs in virtual-8086 mode when the VM (virtual machine) bit in the EFLAOS register is set.
- The processor tests this flag under two general conditions:

  1. When loading segment registers, to know whether to use 8086-style address translation.

  2. When decoding instructions, to determine which instructions are sensitive to IOPL.

- Registers and Instructions
  - *The register set available in virtual-8086 mode includes all the registers defined for the 8086 processor plus the new registers introduced by the 386 DX microprocessor: FS, OS, debug registers, control registers, and test registers.*
- Address Translation
  - *In virtual-8086 mode, the 386 DX microprocessor does not interpret 8086 selectors by referring to descriptors; instead, it forms linear addresses as an 8086 processor would. It shifts the selector left by four bits to form a 20-bit base address. The effective address is extended with four clear bits in the upper bit positions and added to the base address to create a linear address.*

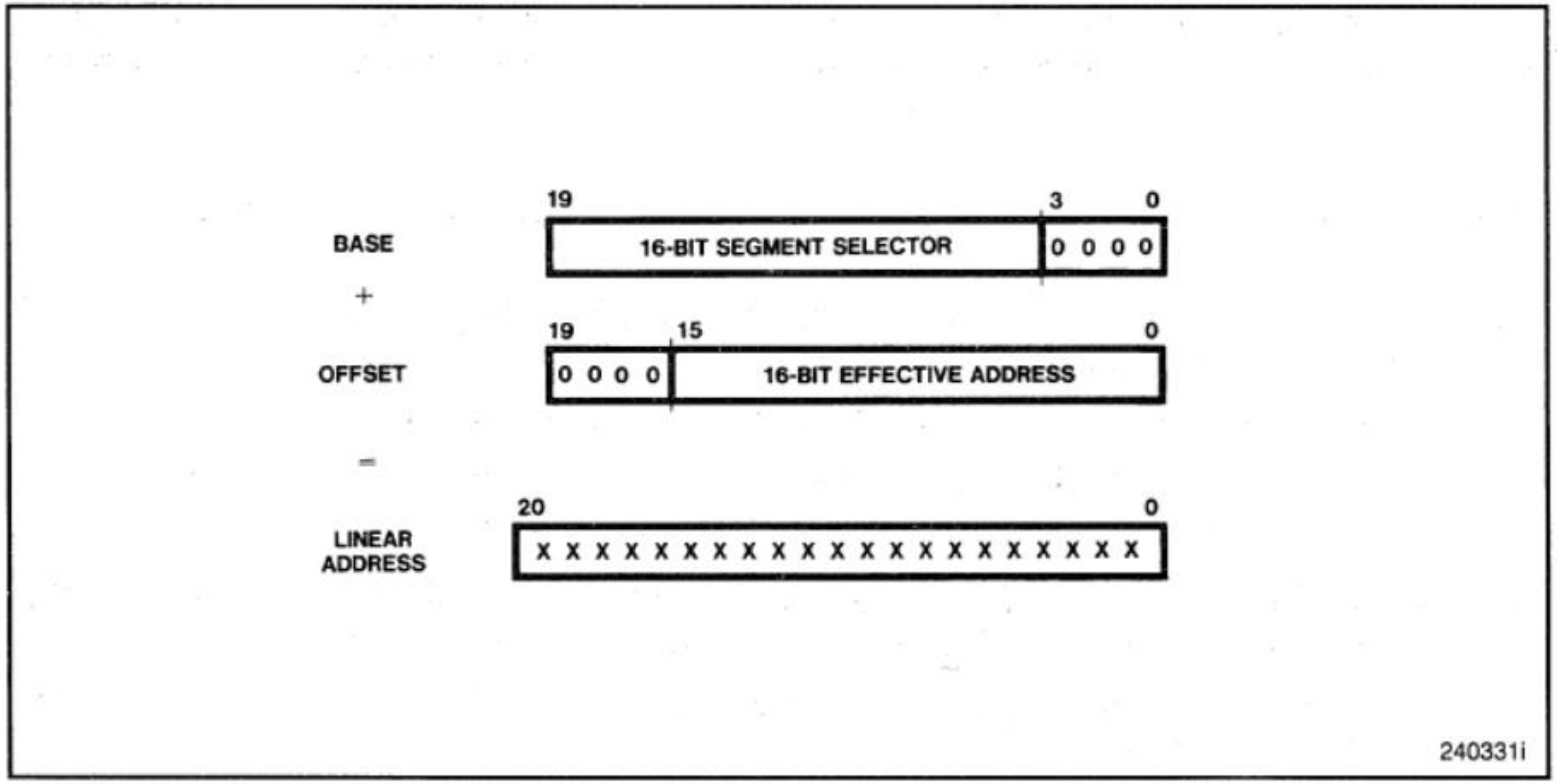


Figure 15-1. 8086 Address Translation

STRUCTURE OF A VIRTUAL-8086 TASK

- A virtual-8086 task consists of the 8086 program to be run and the 386 DX microprocessor "native mode" code which serves as the virtual-machine monitor. The task must be represented by a 386 DX microprocessor TSS (not an 80286 TSS).
- The processor enters virtual-8086 mode to run the 8086 program and returns to protected mode to run the monitor or other 386 DX CPU tasks.
- To run in virtual-8086 mode, an existing 8086 processor program needs the following:
  - *A virtual-8086 monitor.*
  - *Operating-system services.*
- The virtual-8086 monitor is 386 DX microprocessor protected-mode code which runs at privilege-level 0 (most privileged). The monitor mostly consists of initialization and exception-handling procedures.

- In general, there are two options for implementing the 8086 operating system:

  1. The 8086 operating system may run as part of the 8086 program. This approach is desirable for either of the following reasons:
     - *The 8086 application code modifies the operating system.*
     - *There is not sufficient development time to reimplement the 8086 operating system as a 386 DX microprocessor operating system.*

  2. The 8086 operating system may be implemented or emulated in the virtual-8086 monitor. This approach is desirable for any of the following reasons:
     - *Operating system functions can be more easily coordinated among several virtual-8086 tasks.*
     - *The functions of the 8086 operating system.*

- In general, there are two options for implementing the 8086 operating system:

Paging for Virtual-8086 Tasks

- Paging is not necessary for a single virtual-8086 task, but paging is useful or necessary for any of the following reasons:
  - *Creating multiple virtual-8086 tasks. Each task must map the lower megabyte of linear addresses to different physical locations.*
  - *Creating a virtual address space larger than the physical address space.*
  - *Sharing 8086 operating system or ROM code which is common to several 8086 programs running in multitasking.*
  - *Redirecting or trapping references to memory-mapped I/O devices*

Protection within a Virtual-SOS6 Task

- Protection is not enforced between the segments of an 8086 program.
- To protect the system software running in a virtual-8086 task from the 8086 application program, software designers may follow either of these approaches:
  - *Reserve the first megabyte (plus 64K bytes) of each task's linear address space for the 8086 processor program. An 8086 processor task cannot generate addresses outside this range.*
  - *Use the U/S bit of page-table entries to protect the virtual-machine monitor and other system software in each virtual-8086 task's space. When the processor is in virtual-8086 mode, the CPL is 3 (least privileged). Therefore, an 8086 processor program has only user privileges. If the pages of the virtual-machine monitor have supervisor privilege, they cannot be accessed by the 8086 program.*

ENTERING AND LEAVING VIRTUAL-8086 MODE

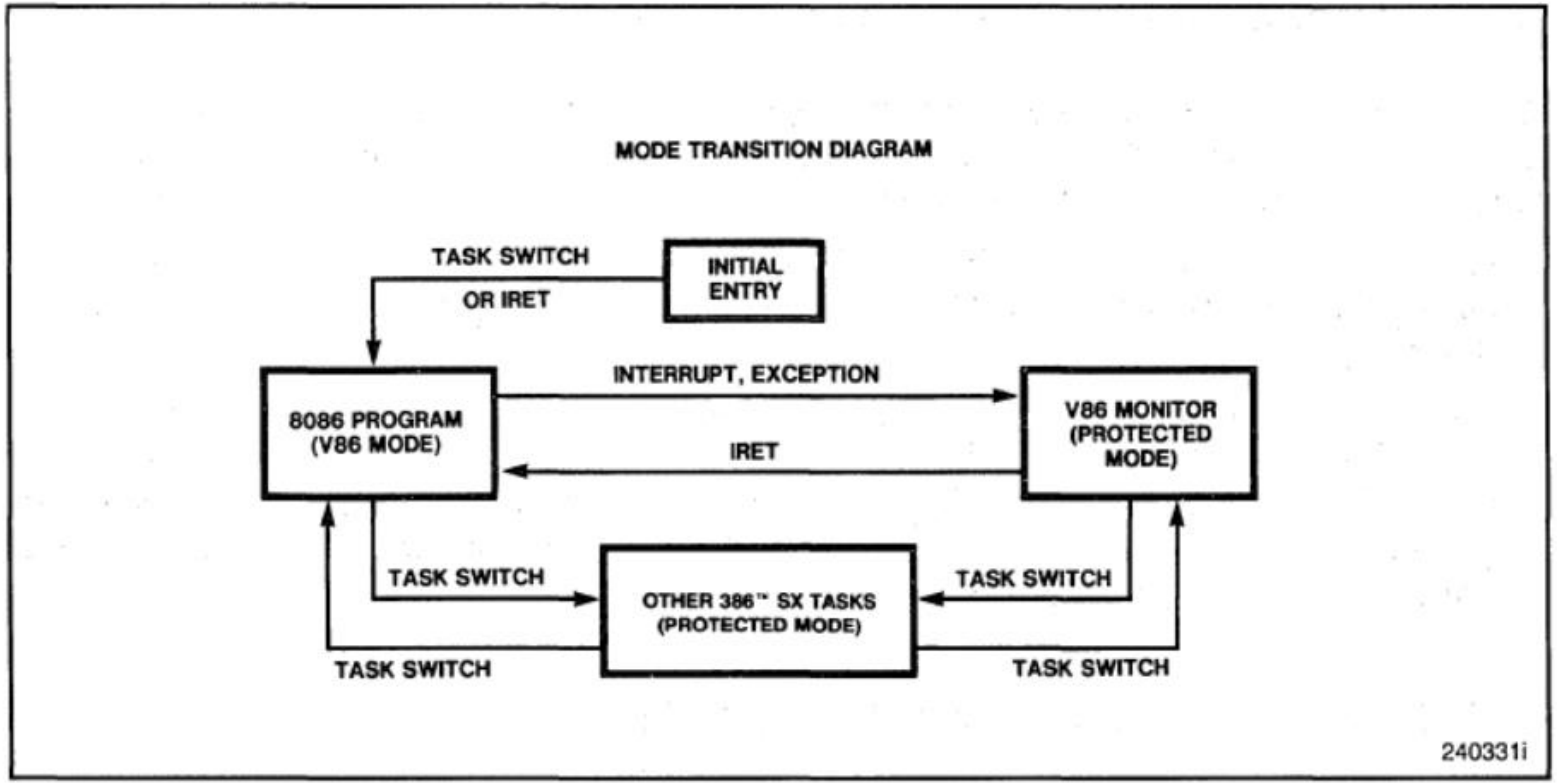


Figure 15-2. Entering and Leaving Virtual-8086 Mode

- Virtual-8086 mode is entered by setting the VM flag. There are two ways to do this:

  1. A task switch to a 386 DX microprocessor task loads the image of the EFLAGS register from the new TSS. A set VM flag in the new contents of the EFLAGS register indicates that the new task is executing 8086 instructions; therefore, while loading the segment registers from the TSS, the 386 DX microprocessor forms base addresses in the 8086 style.

  2. An IRET instruction from a procedure of a 386 DX task loads the EFLAGS register from the stack. A set VM flag indicates the procedure to which control is being returned to be an 8086 procedure.

80387

INTRODUCTION TO THE 80387

- The 80387 NPX is a high-performance numerics processing element that extends the 80386 architecture by adding significant numeric capabilities and direct support for floating-point, extended-integer, and BCD data types.
- The 80386 CPU with 80387 NPX easily supports powerful and accurate numeric applications.
- The 80387 has built-in facilities for commercial computing. It can process decimal numbers of up to 18 digits without round-off errors, performing exact arithmetic on integers as large as 2^{64} or $10^{18.}$
- Exact arithmetic is vital in accounting applications where rounding errors may introduce monetary losses that cannot be reconciled.
- The NPX contains a number of optional facilities that can be invoked by sophisticated users.
- These advanced features include directed rounding, gradual underflow, and programmed exception-handling facilities.

APPLICATIONS

Applications that exhibit any of the following characteristics can benefit by implementing numeric processing on the 80387:

- Numeric data vary over a wide range of values, or include nonintegral values.
- Algorithms produce very large or very small intermediate results.
- Computations must be very precise; i.e., a large number of significant digits must be maintained.
- Performance requirements exceed the capacity of traditional microprocessors.
- Consistently safe, reliable results must be delivered using a programming staff that is not expert in numerical techniques.
- The 80387 can reduce software development costs and improve the performance of systems that use not only real numbers, but operate on multiprecision binary or decimal integer values

- Business data processing
- Simulation
- Graphics transformation
- Process control
- Computer numerical control (CNC)
- Robotics
- Navigation
- Data acquisition

80387 NDP Features

- Control register bits for coprocessor support
- 80387 register stack
- Data types
- Load and store instructions
- Trigonometric and transcendental instructions
- Interfacing signals of 80386DX with 80387
- High performance 80-Bit Internal Architecture
- Implements ANSI/IEEE standard 754-1985 for Binary floating-point arithmetic
- Expands Intel386DX CPU data types to include 32, 64, 80-bit floating point, 32, 64-bit integers and 18-bit BCD operands
- Extends Intel386DX CPU instruction set to include Trigonometric, Logarithmic, Exponential and Arithmetic instructions for all data types
- Upward object code compatible

- Full-range transcendental operations for SINE, COSINE, TANGENT, ARCTANGENT and LOGARITHM
- Built-in Exception handling
- Operates independently in all modes of 80386
- Eight 80-bit Numeric registers
- Available in 68-pin PGA package
- One version supports 16MHz-33MHz

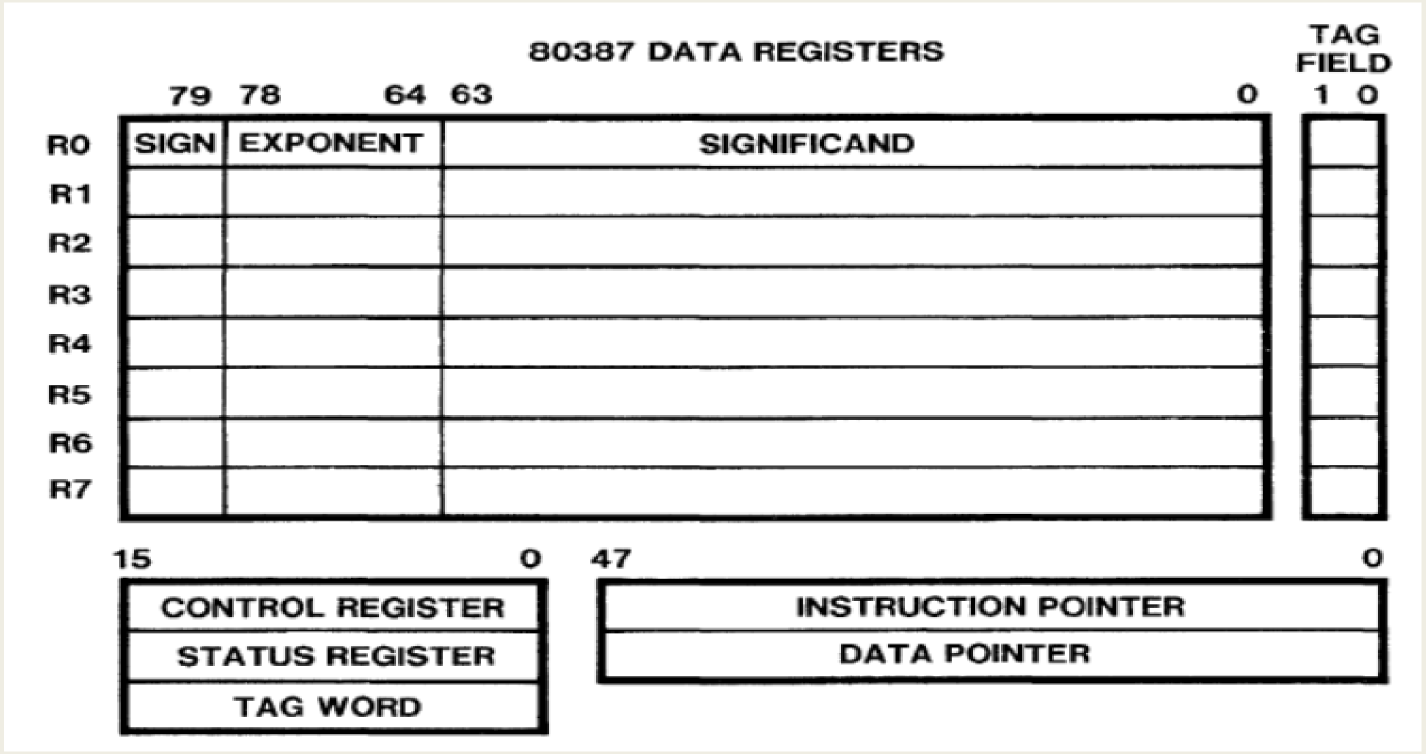
- Data registers: Eight 80-bit registers,
- Tag Word: the tag word marks the content of each numeric data register, two bits for each data register
- Status word: the 16-bit status word reflects the overall state of the MCP
- Instruction and Data pointers: two pointer registers allows identification of the failing numeric instruction which supply the address of failing numeric instruction and the address of its numeric memory operand.
- Control Word: several processing options are selected by loading a control word from memory into the control register

80387 REGISTERS

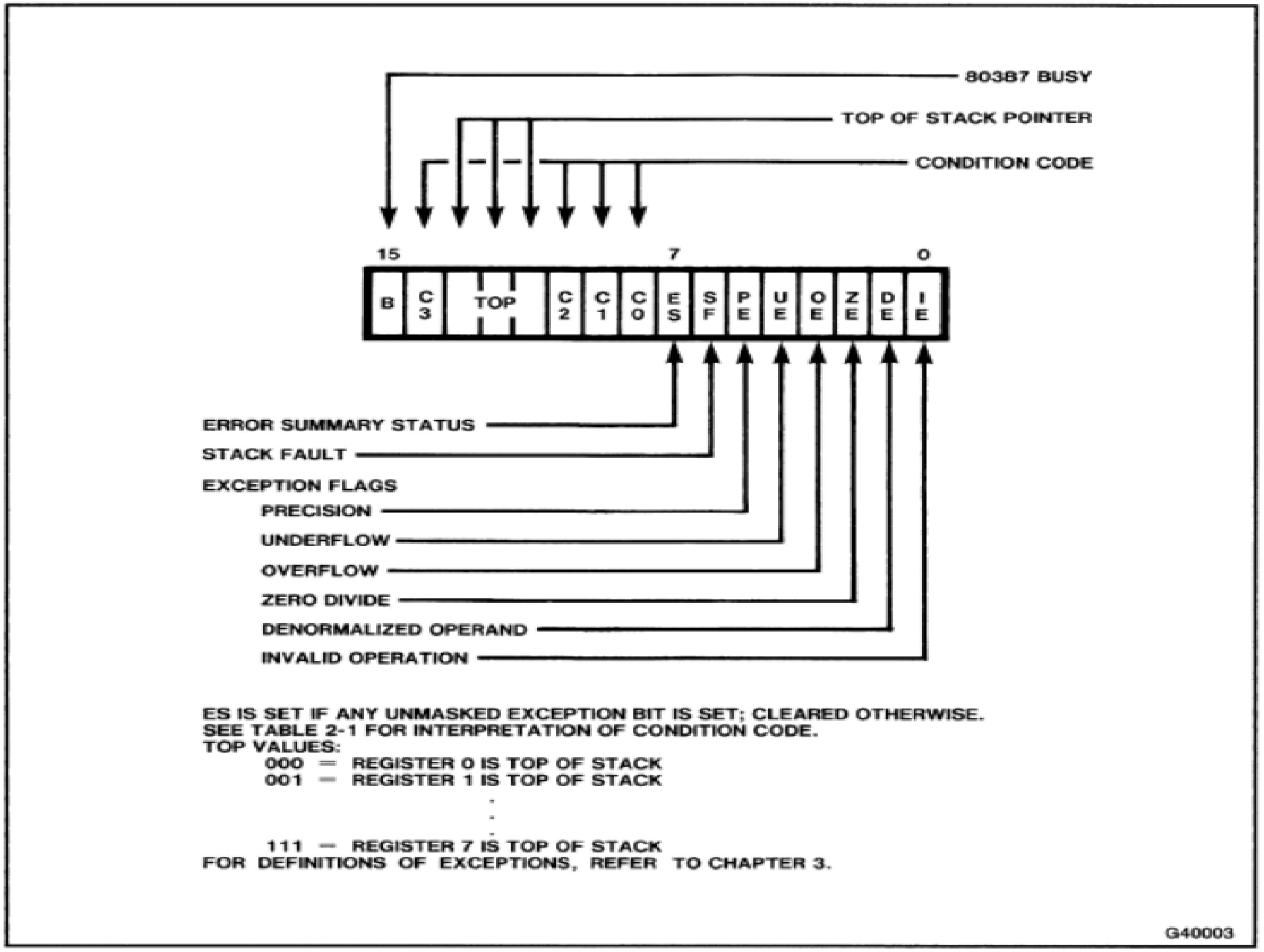
- The additional registers consist of
- Eight individually-addressable 80-bit numeric registers, organized as a register stack
- Three sixteen-bit registers containing:
  - *the NPX status word*
  - *the NPX control word*
  - *the tag word*
- Two 48-bit registers containing pointers to the current instruction and operand

The NPX Register Stack

- Each of the eight numeric registers in the 80387's register stack is 80 bits wide and is divided into fields corresponding to the NPX's extended real data type.
- Numeric instructions address the data registers relative to the register on the top of the stack. At any point in time, this top-of-stack register is indicated by the TOP (stack TOP) field in the NPX status word.
- Load or push operations decrement TOP by one and load a value into the new top register.
- A store-and-pop operation stores the value from the current TOP register and then increments TOP by one.



The NPX Status Word



ES IS SET IF ANY UNMASKED EXCEPTION BIT IS SET; CLEARED OTHERWISE.
SEE TABLE 2-1 FOR INTERPRETATION OF CONDITION CODE.
TOP VALUES:

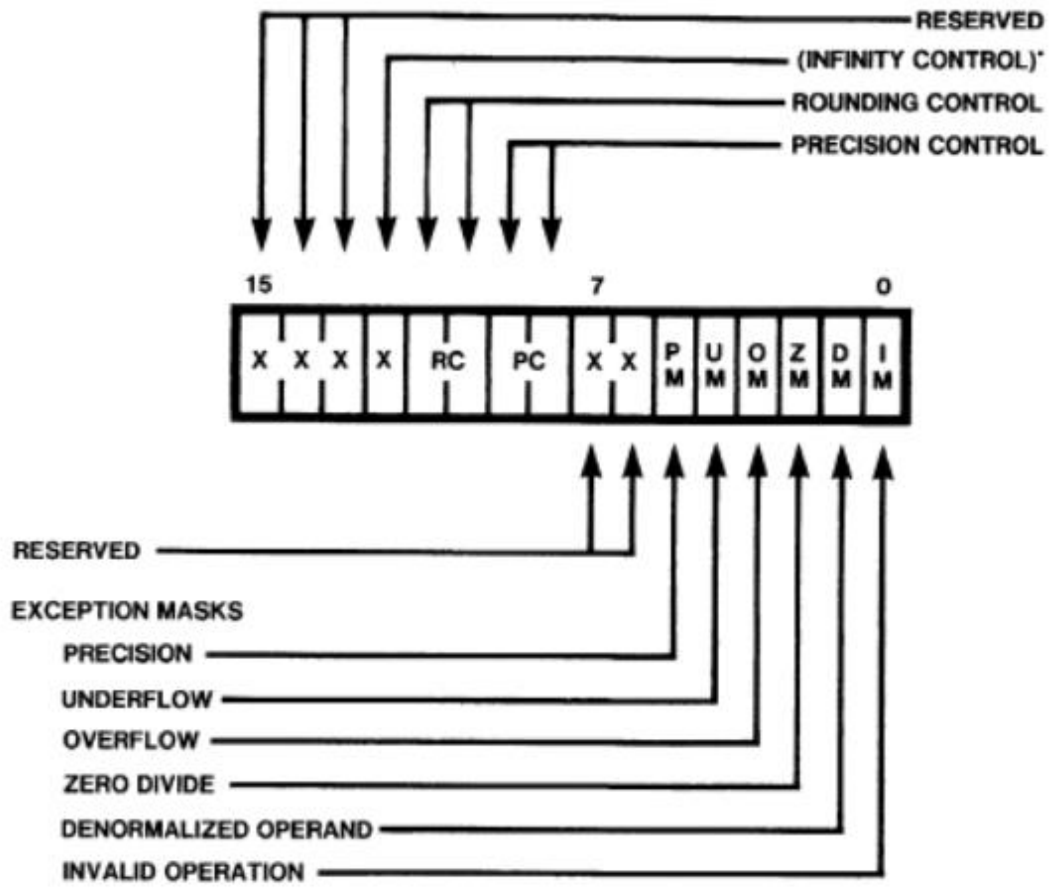
000	=	REGISTER 0 IS TOP OF STACK
001	=	REGISTER 1 IS TOP OF STACK
		.
		.
		.
111	=	REGISTER 7 IS TOP OF STACK

FOR DEFINITIONS OF EXCEPTIONS, REFER TO CHAPTER 3.

Figure 2-2. 80387 Status Word

Control Word

- The low-order byte of this control word configures the 80387 exception masking. Bits 0-5 of the control word contain individual masks for each of the six exception conditions recognized by the 80387.
- The high-order byte of the control word configures the 80387 processing options, including
- Precision control
- Rounding control



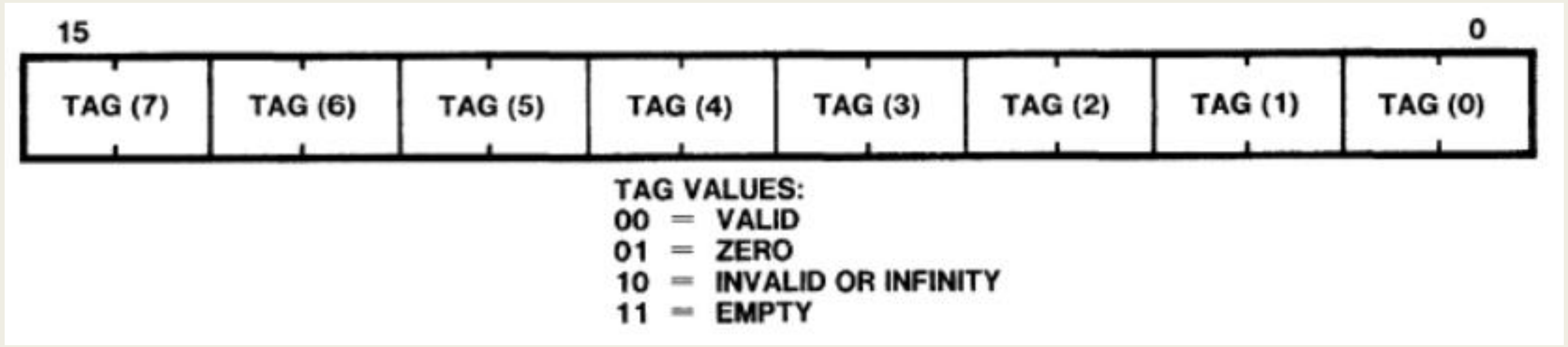
PRECISION CONTROL
- 00—24 BITS (SINGLE PRECISION)
- 01—(RESERVED)
- 10—53 BITS (DOUBLE PRECISION)
- 11—64 BITS (EXTENDED PRECISION)

ROUNDING CONTROL
- 00—ROUND TO NEAREST OR EVEN
- 01—ROUND DOWN (TOWARD $-\infty$)
- 10—ROUND UP (TOWARD $+\infty$)
- 11—CHOP (TRUNCATE TOWARD ZERO)

*This "infinity control" bit is not meaningful to the 80387. To maintain compatibility with the 80287, this bit can be programmed; however, regardless of its value, the 80387 treats infinity in the affine sense ($-\infty < +\infty$).

Tag Word

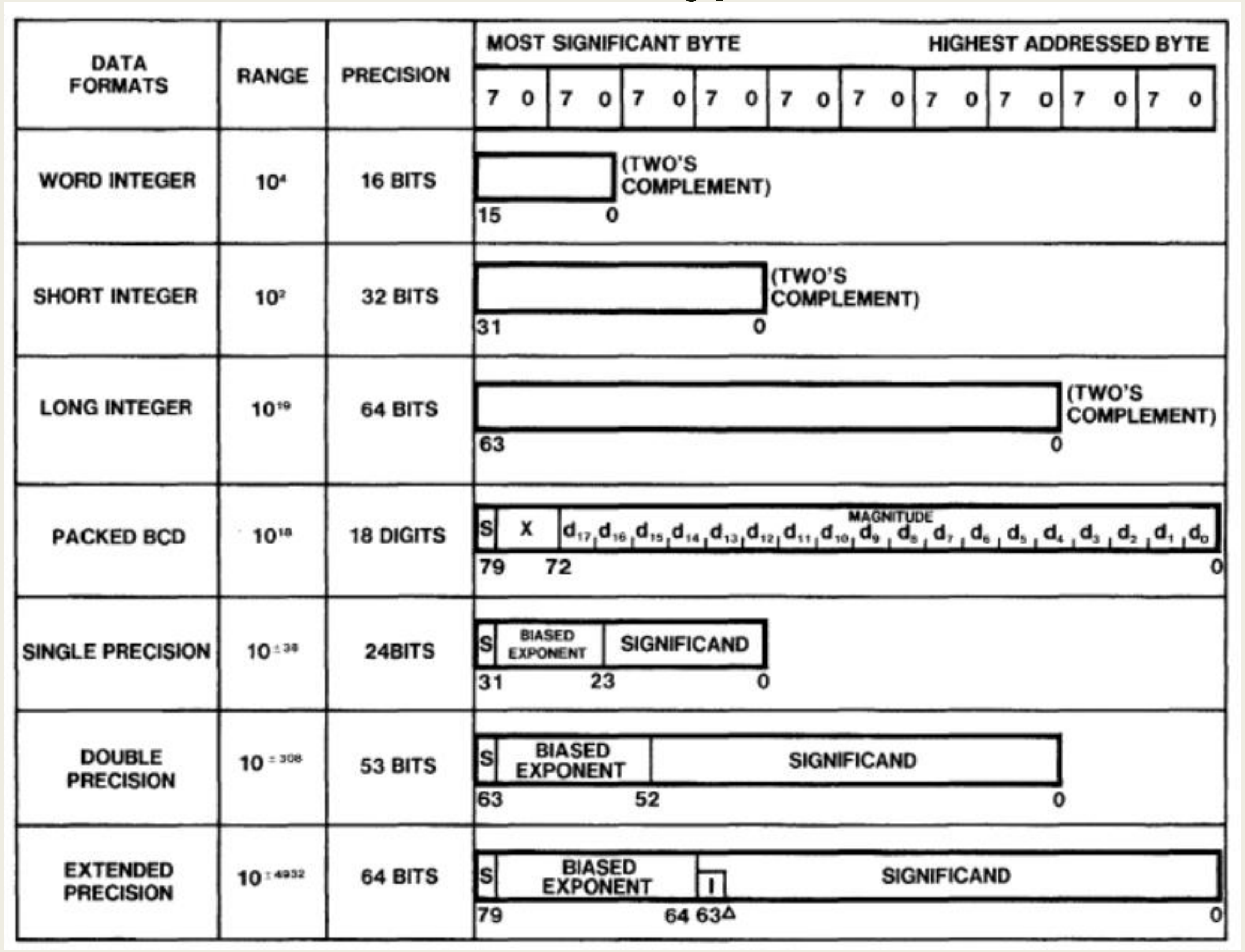
- The tag word indicates the contents of each register in the register stack.
- The tag word is used by the NPX itself to distinguish between empty and non empty register locations.
- Programmers of exception handlers may use this tag information to check the contents of a numeric register without performing complex decoding of the actual data in the register.
- The tag values from the tag word correspond to physical registers 0-7.
- Programmers must use the current top-of-stack (TOP) pointer stored in the NPX status word to associate these tag values with the relative stack registers ST(0) through ST(7).



TAG VALUES:

00	=	VALID
01	=	ZERO
10	=	INVALID OR INFINITY
11	=	EMPTY

Data Types



DATA FORMATS	RANGE	PRECISION	MOST SIGNIFICANT BYTE HIGHEST ADDRESSED BYTE 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0
WORD INTEGER	10^{4}	16 BITS	15 … 0 (TWO'S COMPLEMENT)
SHORT INTEGER	10^{9}	32 BITS	31 … 0 (TWO'S COMPLEMENT)
LONG INTEGER	10^{19}	64 BITS	63 … 0 (TWO'S COMPLEMENT)
PACKED BCD	10^{18}	18 DIGITS	S | X | MAGNITUDE d_{17} d_{16} d_{15} d_{14} d_{13} d_{12} d_{11} d_{10} d_{9} d_{8} d_{7} d_{6} d_{5} d_{4} d_{3} d_{2} d_{1} d_{0} 79 72 0
SINGLE PRECISION	$10^{\pm 38}$	24BITS	S | BIASED EXPONENT | SIGNIFICAND 31 23 0
DOUBLE PRECISION	$10^{\pm 308}$	53 BITS	S | BIASED EXPONENT | SIGNIFICAND 63 52 0
EXTENDED PRECISION	$10^{\pm 4932}$	64 BITS	S | BIASED EXPONENT | I | SIGNIFICAND 79 64 63^{Δ} 0

DATA TRANSFER INSTRUCTIONS

- These instructions move operands among elements of the register stack, and between the stack top and memory.
- Any of the seven data types can be converted to extended real and loaded (pushed) onto the stack in a single operation; they can be stored to memory in the same manner.

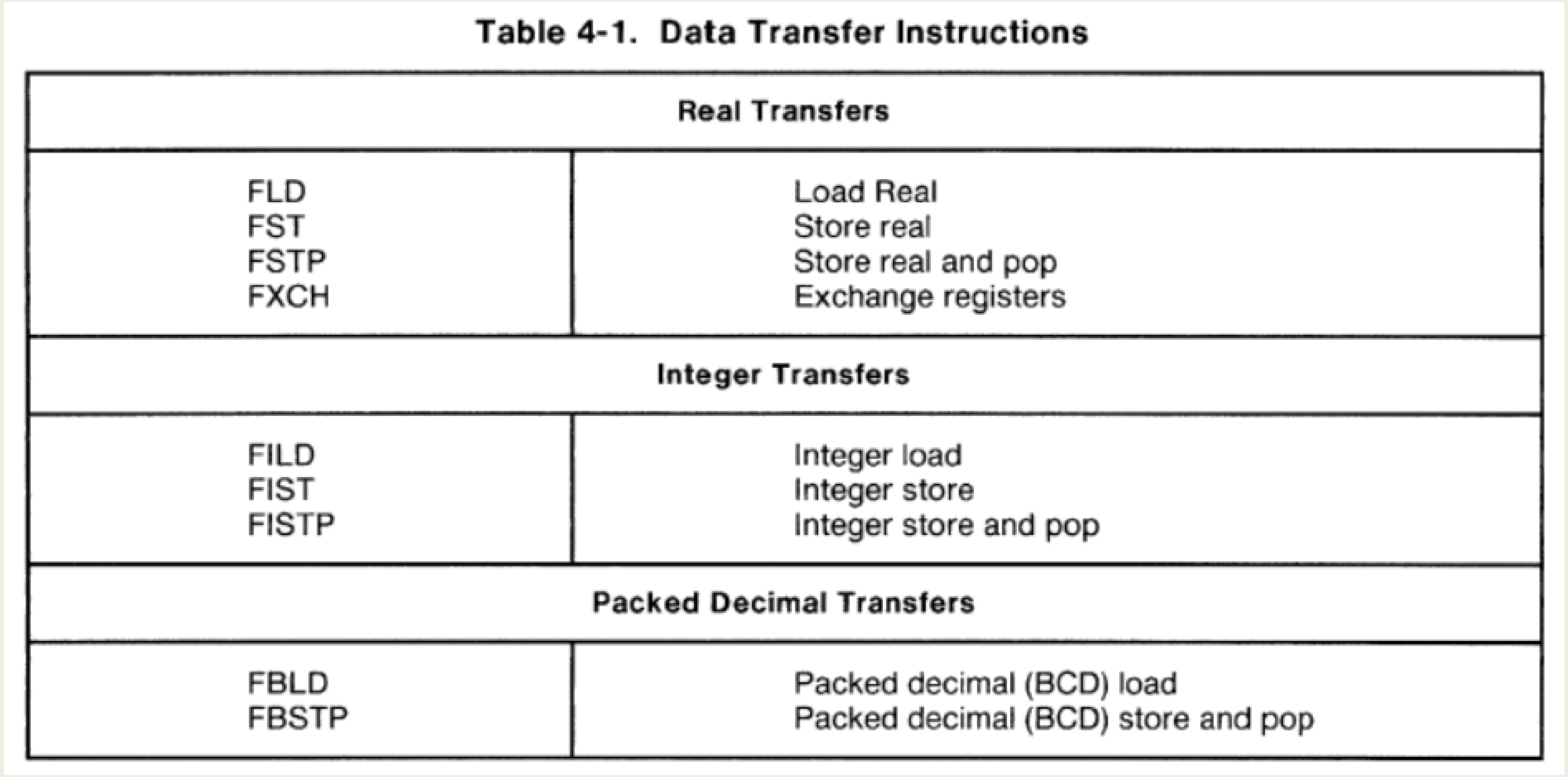


Table 4-1. Data Transfer Instructions

Real Transfers	
FLD	Load Real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

Real Transfers

- **FLD *source***

FLD (load real) loads (pushes) the source operand onto the top of the register stack. This is done by decrementing the stack pointer by one and then copying the content of the source to the new stack top.

- **FST *destination***

FST (store real) copies the NPX stack top to the destination, which may be another register on the stack or a single or double (but not extended-precision) memory operand.

- **FSTP *destination***

FSTP (store real and pop) operates identically to FST except that the NPX stack is popped following the transfer.

- **FXCH *destination***

FXCH (exchange registers) swaps the contents of the destination and the stack top registers. If the destination is not coded explicitly, ST(1) is used.

Integer Transfers

- **FILD *source***
  - FILD (integer load) converts the source memory operand from its binary integer format(word, short, or long) to extended real and pushes the result onto the NPX stack.

- **FIST *destination***
  - FIST (integer store) stores the content of the stack top to an integer according to the RC field (rounding control) of the control word and transfers the result to the destination, leaving the stack top unchanged.

- **FISTP *destination***
  - FISTP (integer and pop) operates like FIST except that it also pops the NPX stack following the transfer. The destination may be any of the binary integer data types.

Packed Decimal Transfers

- **FBLD *source***
  - FBLD (packed decimal (BCD) load) converts the content of the source operand from packed decimal to extended real and pushes the result onto the 80387 stack. ST(7) must be empty to avoid causing an exception.

- **FBSTP *destination***
  - FBSTP (packed decimal (BCD) store and pop) converts the content of the stack top to a packed decimal integer, stores the result at the destination in memory, and pops the stack.

TRANSCENDENTAL INSTRUCTIONS

- **FCOS**

  When complete, this function replaces the contents of ST with COS(ST). ST, expressed in radians, must lie in the range || < 263 (for most practical purposes unrestricted).

- **FSIN**

  When complete, this function replaces the contents of ST with SIN(ST). FSIN is equivalent to FCOS in the way it reduces the operand. ST is expressed in radians.

- **FSINCOS**

  When complete, this instruction replaces the contents of ST with SIN(ST), then pushes COS(ST) onto the stack. (ST(7) must be empty to avoid an invalid exception.) FSINCOS is equivalent to FCOS in the way it reduces the operand. ST is expressed in radians.

- FPTAN

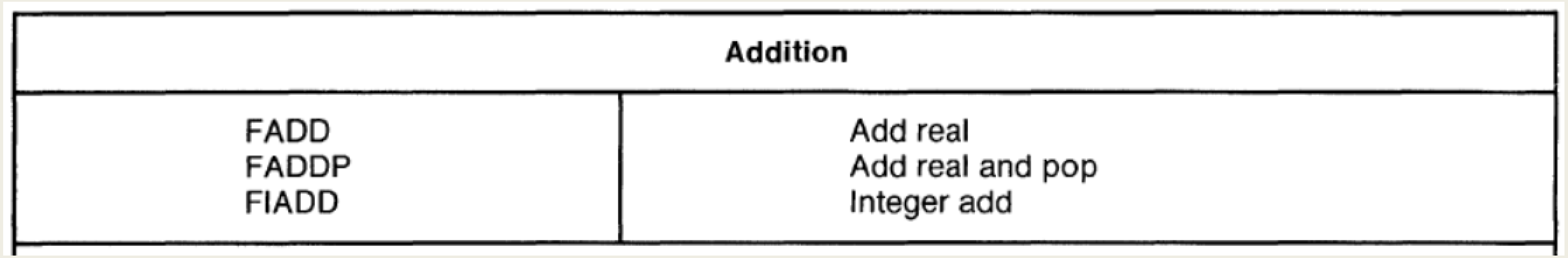
  When complete, FPTAN (partial tangent) computes the function Y = TAN (ST). ST is expressed in radians. Y replaces ST, then the value 1 is pushed, becoming the new stack top. (ST(7) must be empty to avoid an invalid exception.) When the function is complete ST(l) = TAN (arg) and ST = 1. FPTAN is equivalent to FCOS in the way it reduces the operand.

- FPATAN

  FPATAN (arctangent) computes the function 8 = ARCTAN (Y IX). X is taken from ST(O) and Y from ST(l). The instruction pops the NPX stack and returns 8 to the (new) stack top, overwriting the Y operand. The result is expressed in radians.

<u>Non-Transcendental Instructions</u>

Addition

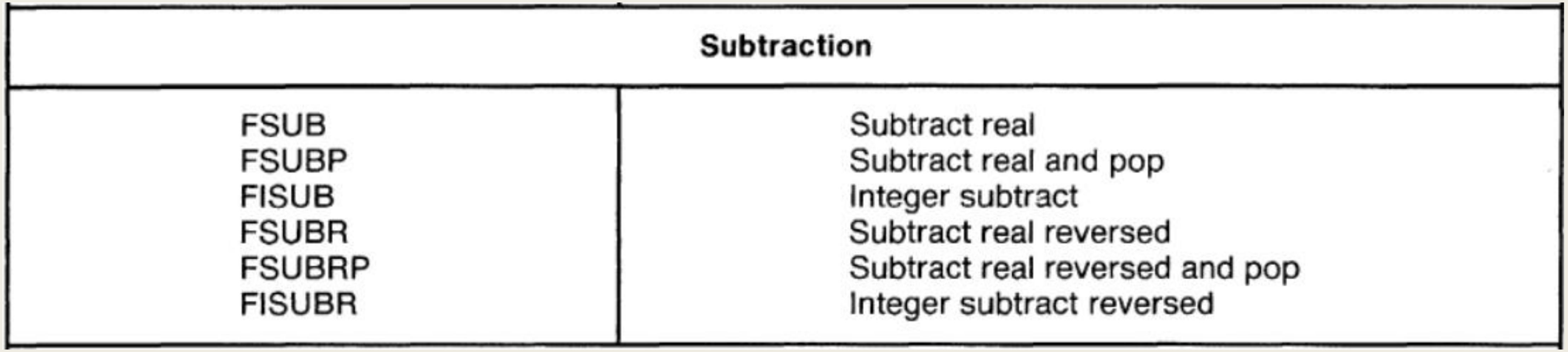


Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add

The addition instructions (add real, add real and pop, integer add) add the source and destination operands and return the sum to the destination. The operand at the stack top may be doubled by coding:

FADD ST, ST(O)

Subtraction

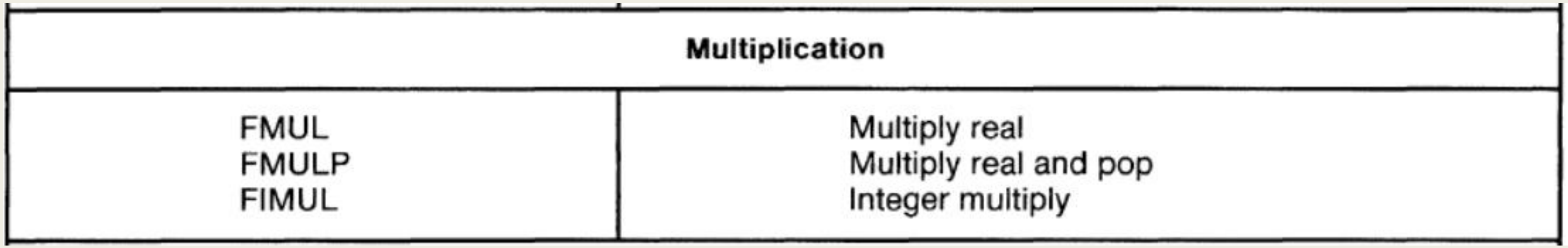


Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed

- The normal subtraction instructions (subtract real, subtract real and pop, integer subtract) subtract the source operand from the destination and return the difference to the destination.

- The reversed subtraction instructions (subtract real reversed, subtract real reversed and pop, integer subtract reversed) subtract the destination from the source and return the difference to the destination. For example, FSUBR ST, ST(1) means subtract ST from ST(1) and leave the result in ST.

Multiplication

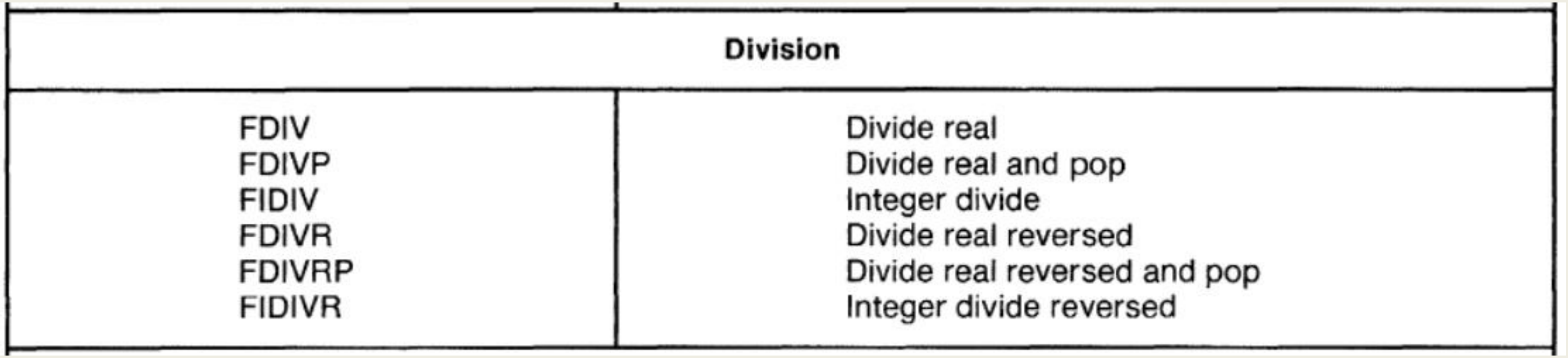


Multiplication	
FMUL FMULP FIMUL	Multiply real Multiply real and pop Integer multiply

- The multiplication instructions (multiply real, multiply real and pop, integer multiply) multiply the source and destination operands and return the product to the destination.

- Coding FMUL ST,ST(0) squares the content of the stack top.

Division

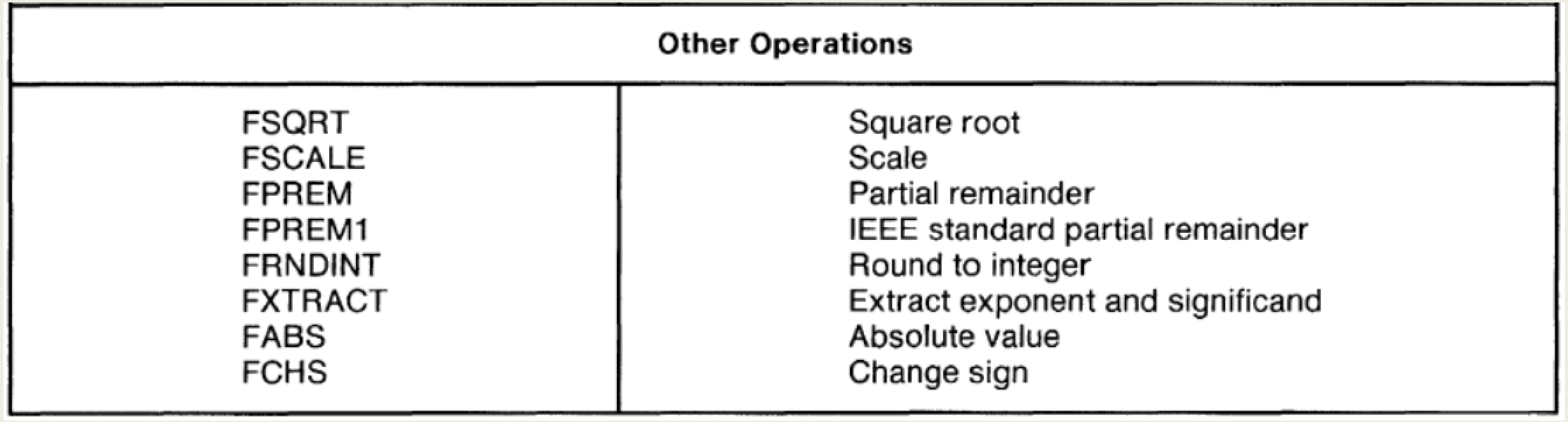


Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed

- The normal division instructions (divide real, divide real and pop, integer divide) divide the destination by the source and return the quotient to the destination.

- The reversed division instructions (divide real reversed, divide real reversed and pop, integer divide reversed) divide the source operand by the destination and return the quotient to the destination.

Other Operations



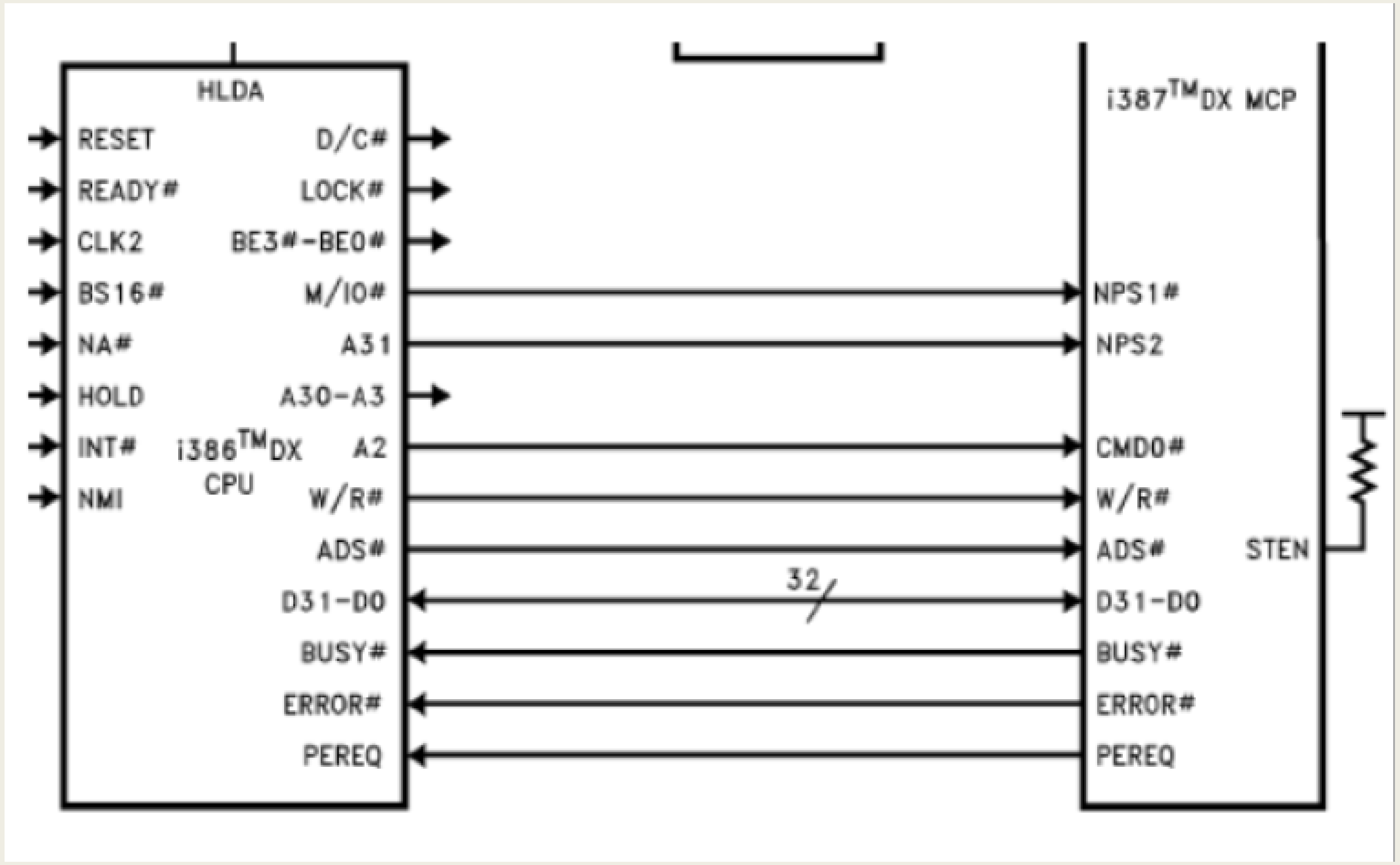
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FPREM1	IEEE standard partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

- FSQRT (square root) replaces the content of the top stack element with its square root.

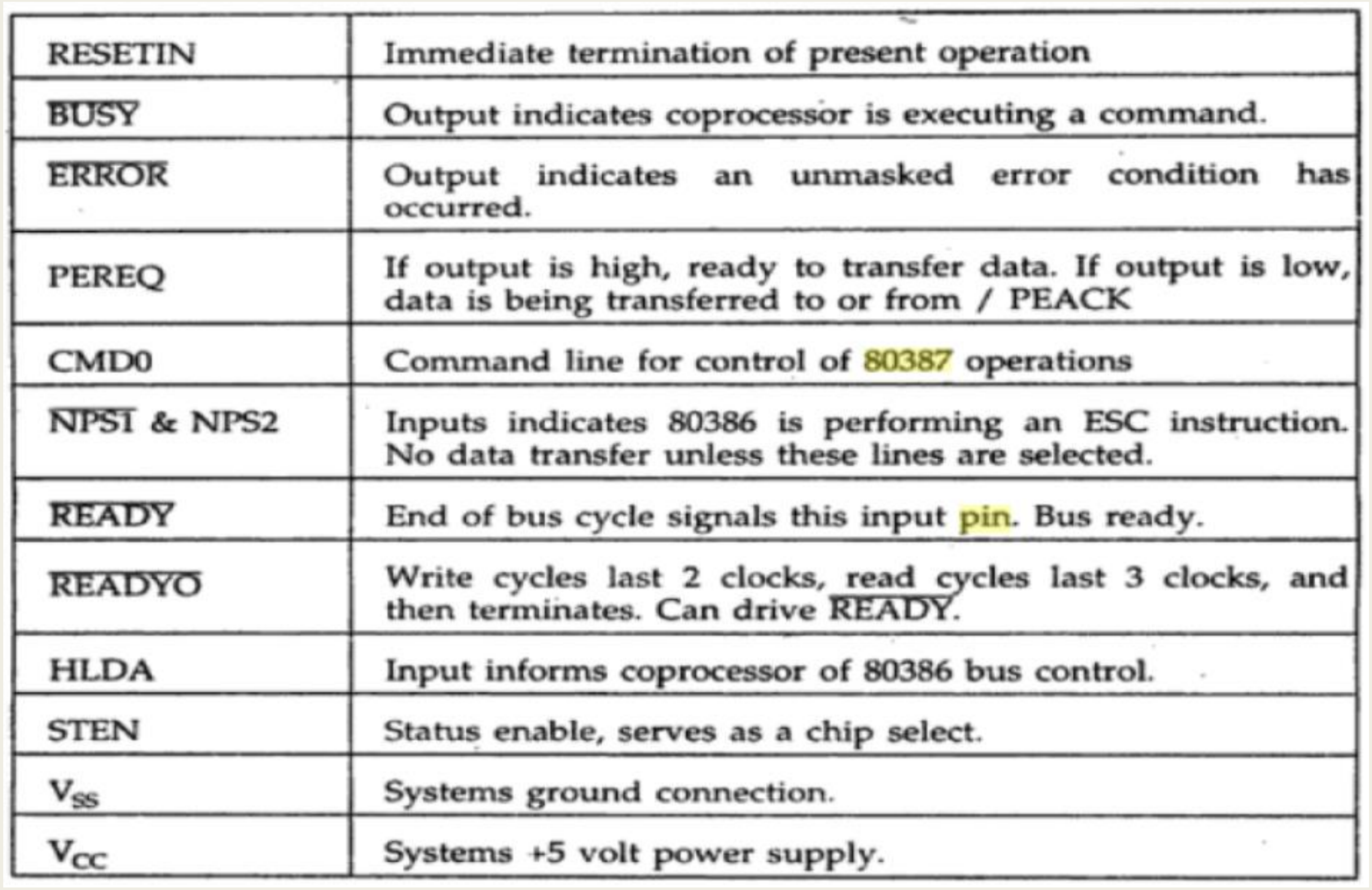
- FSCALE (scale) interprets the value contained in ST(1) as an integer and adds this value to the exponent of the number in ST. This is equivalent to $ST \leftarrow ST*2^{ST(1)}$

Interfacing Signals of 80386DX with 80387

80386 Pin	80387 Pin
M/IO#	NPS1#
A31	NPS2
A2	CMD0#
W/R#	W/R#
ADS#	ADS#
D31-D0	D31-D0
BUSY#	BUSY#
ERROR#	ERROR#
PEREQ	PEREQ



80387 Pin Description



RESETIN	Immediate termination of present operation
$\overline{\text{BUSY}}$	Output indicates coprocessor is executing a command.
$\overline{\text{ERROR}}$	Output indicates an unmasked error condition has occurred.
PEREQ	If output is high, ready to transfer data. If output is low, data is being transferred to or from / PEACK
CMD0	Command line for control of 80387 operations
$\overline{\text{NPS1}}$ & NPS2	Inputs indicates 80386 is performing an ESC instruction. No data transfer unless these lines are selected.
$\overline{\text{READY}}$	End of bus cycle signals this input pin. Bus ready.
$\overline{\text{READYO}}$	Write cycles last 2 clocks, read cycles last 3 clocks, and then terminates. Can drive $\overline{\text{READY}}$.
HLDA	Input informs coprocessor of 80386 bus control.
STEN	Status enable, serves as a chip select.
V_{SS}	Systems ground connection.
V_{CC}	Systems +5 volt power supply.