Microprocessors & Microcontrollers : Unit-3

Prof. Prajay P. Raikar
Visiting Faculty
Department of Computer Engineering
Goa College of Engineering, Farmagudi-Ponda, Goa
prajayraikar@gmail.com

Table of Contents

Initialization of 80386

---NOT PART OF SYLLABUS---

The initialization of the 80386 processor is a multi-step process that carefully transitions the CPU from a minimal, recoverable starting state to a fully configured environment ready for advanced functions like protected mode, multitasking, and memory protection. Here's a detailed look at each phase of the 80386's initialization:

---

1. Post-Reset State

When the 80386 is reset (either on power-up or a hardware reset), it enters a minimal but well-defined state:

- **Real Mode Operation:**
  The processor starts in real mode—a legacy mode carried forward from the 8086—where addressing is limited to a 20-bit (1 MB) space. This mode uses a simple segmented memory model without the advanced protections of later modes.
- **Reset Vector:**
  Immediately after reset, the CPU fetches its first instruction from the physical address **F000:FFF0**. This fixed starting point ensures that the system firmware, typically stored in ROM (commonly referred to as the BIOS), reliably begins execution from the same location every time.
- **Registers and Flags:**
  The EFLAGS register is initialized with only its reserved bit set (yielding a typical value of 0x00000002). Other critical flags—such as the Interrupt Flag (IF)—are cleared, meaning interrupts remain disabled until explicitly enabled by the boot code. General-purpose and most segment registers are in an undefined state (except for CS, which is set to F000h), ensuring that higher-level initialization code must explicitly configure them.

---

2. BIOS and Bootloader Intervention

After the CPU begins executing the BIOS code from the reset vector, the following responsibilities fall to the system firmware:

- **Hardware Self-Testing:**
  The BIOS conducts POST (Power-On Self-Test) routines to verify that the essential hardware components are functioning correctly.
- **Memory and Peripheral Initialization:**
  Initial setup of the Interrupt Vector Table (IVT) occurs in the low 1 MB memory, along with configuring basic hardware parameters such as the keyboard, disk controllers, and other interfaces.
- **Stack Setup:**
  An essential early step is establishing a reliable stack. The BIOS or bootloader sets up the stack segment (SS) and stack pointer (SP/ESP) so that subsequent calls and operations have a safe area in memory for temporary data storage.

This early environment, though limited by real mode addressing, sets the stage for more sophisticated initialization.

---

3. Preparing for Protected Mode: The Global Descriptor Table (GDT)

To take advantage of the advanced features of the 80386, such as 32-bit addressing, better memory protection, and multitasking, the system must transition from real mode to protected mode. A critical step in this transition is the setup of the Global Descriptor Table (GDT):

- **Global Descriptor Table (GDT):**
  The GDT is a memory-resident structure that defines the attributes of various memory segments (e.g., code, data, and system segments). Each descriptor in the GDT specifies the base address, segment limit, access permissions, and other flags that dictate how the segment behaves.
- **Loading the GDT:**
  The bootloader allocates space for the GDT, fills it with appropriate segment descriptors, and then uses the LGDT instruction to load the address and size of the GDT into the processor's **GDTR** register. This tells the CPU where to find the descriptors that will define a robust memory model in protected mode.

This step is crucial because, without a properly configured GDT, the processor cannot switch modes safely.

---

4. Transition to Protected Mode

Moving from real mode to protected mode involves a carefully orchestrated series of steps:

- **Interrupt Disabling:**
  Before initiating the transition, the system disables interrupts to prevent any interference that might corrupt the delicate state change.
- **Modifying Control Register CR0:**
  The process of entering protected mode is triggered by setting the **Protection Enable (PE)** bit in the **CR0** register. Initially, CR0 is set to a default that indicates real mode; setting the PE bit tells the processor to alter its behavior to use 32-bit addressing and enforce privilege levels and access rights.
- **Executing a Far Jump:**
  Simply setting the PE bit is not enough because the processor's internal pipelines may still be using preloaded instructions from real mode. A **far jump**—an instruction that changes both the instruction pointer (EIP) and the Code Segment (CS) register—is executed. This jump reloads CS from a descriptor in the newly loaded GDT, flushing the prefetch queue and ensuring that all subsequent instructions follow the rules of protected mode.

At this point, the processor fully commits to protected mode, unlocking the features that support modern operating system requirements.

---

5. Post-Transition Initialization and Enabling Advanced Features

Once operating in protected mode, additional initialization steps are typically undertaken by the operating system:

- **Local Descriptor Table (LDT):**
  The OS may set up one or more LDTs to define segments that are specific to individual tasks. This further refines the segmentation scheme established by the GDT.
- **Paging Setup (if applicable):**
  Many systems also enable paging to extend memory management capabilities. This involves configuring the **CR3** register with a pointer to a page directory and setting up page tables that map virtual addresses to physical memory. Paging allows for more efficient use of memory and isolation between processes.
- **Interrupt Descriptor Table (IDT):**
  The OS installs an IDT for handling interrupts and exceptions under the new protected mode rules, which further distinguishes it from the simpler, legacy real mode environment.
- **Re-enabling Interrupts:**
  With the new memory segmentation and protection in place, the OS then re-enables interrupts so that the system can respond to hardware events and task scheduling as needed.

---

6. Additional Hardware Structures

Besides the primary initialization steps, the 80386 provides internal structures that facilitate sophisticated operation:

- **TLB and Test Registers:**
  The Translation Lookaside Buffer (TLB) expedites the translation of linear addresses to physical addresses. The processor includes test registers (such as TR6 and TR7) aimed at verifying TLB functionality as part of system self-tests or advanced debugging routines.
- **Debug Registers:**
  Although not directly part of the basic initialization sequence, debug registers (DR0–DR7) are available should the system need hardware-assisted debugging. These registers remain inactive until explicitly configured by system software or a debugger.

In Summary

The initialization of the 80386 processor involves a carefully executed series of stages:

1. **Reset Phase:**
   The CPU boots into real mode—fetching its first instruction from F000:FFF0, with a minimal state (limited addressing, basic EFLAGS settings, undefined general-purpose registers).
2. **BIOS/Bootloader Role:**
   Early firmware performs self-tests, sets up the IVT, and establishes essential components like the system stack.

3. **GDT Setup:**
   The bootloader allocates and loads a Global Descriptor Table that defines the segments needed for a robust memory model in protected mode.

4. **Protected Mode Transition:**
   By setting the PE bit in CR0 and executing a far jump, the CPU switches from real mode to protected mode, thereby unlocking features like 32-bit addressing and improved security.

5. **Advanced Configuration:**
   Operating system code then takes over—setting up LDTs, enabling paging, installing the IDT, and re-enabling interrupts—thus preparing the processor for multitasking and secure operation.

This layered process ensures that the 80386 evolves from a minimal power-up state to a fully capable environment, paving the way for the complex operating systems that modern computing demands.

Processor State after Reset

- The contents of EAX depend upon the results of the power-up self-test. The self-test may be requested externally by assertion of BUSY pin at the end of RESET.
- The **EAX register holds zero if the 80386 passed the test.** A nonzero value in EAX after self-test indicates that the 80386 unit is faulty.
- If the self-test is not requested, the contents of EAX after RESET is undefined.
- **DX holds a component identifier and revision number** after RESET as Figure 10-1 illustrates.
- DH is set to a constant value of 3. This fixed value indicates that the processor is indeed an 80386 component. It serves as a hardware signature for the chip, making it possible for system software (or BIOS code) to quickly recognize the chip type.
- DL byte contains a unique identifier representing the chip's revision or stepping level. The value in DL is not arbitrary; it reflects the **specific revision of the 80386** being used. This information can be useful for low-level diagnostics, debugging, or when applying specific errata workarounds based on the particular stepping of the chip.
- **Control register zero (CR0)** contains the values shown in Figure 10-2.
- The ET (Extension Type) bit of CR0 is set if an 80387 (numeric coprocessor) is present in the configuration. If ET is reset, the configuration either contains an 80287 or does not contain a coprocessor.
- A software test is required to distinguish between these latter two possibilities.

```
The remaining registers and flags are set as follows:

   EFLAGS               =00000002H
   IP                   =0000FFF0H
   CS selector          =000H
   DS selector          =0000H
   ES selector          =0000H
   SS selector          =0000H
   FS selector          =0000H
   GS selector          =0000H
   IDTR:
              base      =0
              limit     =03FFH

All registers not mentioned above are undefined.
```

Figure 10-1. Contents of EDX after RESET

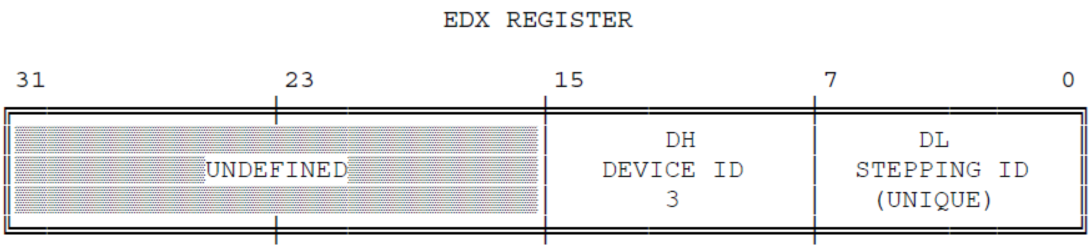
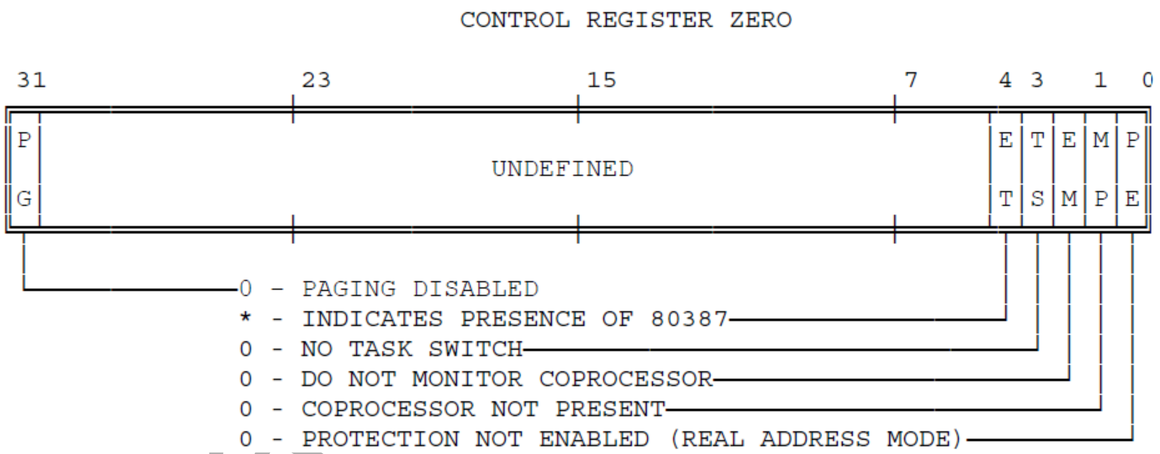


Figure 10-2. Initial Contents of CR0


CONTROL REGISTER ZERO
31
23
15
7
4
3
1
0
PG
UNDEFINED
ET
TS
EM
MP
PE
0 - PAGING DISABLED
* - INDICATES PRESENCE OF 80387
0 - NO TASK SWITCH
0 - DO NOT MONITOR COPROCESSOR
0 - COPROCESSOR NOT PRESENT
0 - PROTECTION NOT ENABLED (REAL ADDRESS MODE)

Software Initialization for Real Addressing Mode

Stack

No instructions that use the stack can be used until the stack-segment register (SS) has been loaded. SS must point to an area in RAM.

Interrupt Table

The initial state of the 80386 leaves interrupts disabled; however, the processor will still attempt to access the interrupt table if an exception or nonmaskable interrupt (NMI) occurs. Initialization software should take one of the following actions:

- Change the limit value in the IDTR to zero. This will cause a shutdown if an exception or non-maskable interrupt occurs. (Refer to the 80386 Hardware Reference Manual to see how shutdown is signalled externally.)
- Put pointers to valid interrupt handlers in all positions of the interrupt table that might be used by exceptions or interrupts.
- Change the IDTR to point to a valid interrupt table.

First Instructions

- After RESET, address lines A{31-20} are automatically asserted for instruction fetches. This fact, together with the initial values of CS:IP, causes instruction execution to begin at physical address FFFFFFF0H.
- Near (intrasegment) forms of control transfer instructions may be used to pass control to other addresses in the upper 64K bytes of the address space.
- The first far (intersegment) JMP or CALL instruction causes A{31-20} to drop low, and the 80386 continues executing instructions in the lower one megabyte of physical memory. This automatic assertion of address lines A{31-20} allows systems designers to use a ROM at the high end of the address space to initialize the system.

Switching to Protected Mode

- Setting the PE bit of the MSW in CR0 causes the 80386 to begin executing in protected mode. The current privilege level (CPL) starts at zero.
- The segment registers continue to point to the same linear addresses as in real address mode (in real address mode, linear addresses are the same physical addresses).

Software Initialization for Protected Mode

Interrupt Descriptor Table

- The IDTR may be loaded in either real-address or protected mode. However, the format of the interrupt table for protected mode is different than that for real-address mode.
- It is not possible to change to protected mode and change interrupt table formats at the same time; therefore, it is inevitable that, if IDTR selects an interrupt table, it will have the wrong format at some time.
- An interrupt or exception that occurs at this time will have unpredictable results**. To avoid this unpredictability, interrupts should remain disabled** until interrupt handlers are in place and a valid IDT has been created in protected mode.

Stack

The SS register may be loaded in either real-address mode or protected mode. If loaded in real-address mode, **SS continues to point to the same linear base-address** after the switch to protected mode.

Global Descriptor Table

- Before any segment register is changed in protected mode, the GDT register must point to a valid GDT.
- Initialization of the GDT and GDTR may be done in real-address mode.
- The GDT (as well as LDTs) should reside in RAM, because the processor modifies the accessed bit of descriptors.

Page Tables

- Page tables and the PDBR in **CR3 can be initialized in either real-address mode or in protected mode;** however, the **paging enabled (PG) bit of CR0 cannot be set until the processor is in protected mode.**
- PG may be set simultaneously with PE, or later.
- When PG is set, the PDBR in CR3 should already be initialized with a physical address that points to a valid page directory.

The initialization procedure should adopt one of the following strategies to ensure consistent addressing before and after paging is enabled:

- The page that is currently being executed should map to the same physical addresses both before and after PG is set.
- A JMP instruction should immediately follow the setting of PG.

First Task

The initialization procedure can run awhile in protected mode without initializing the task register; however, before the first task switch, the following conditions must prevail:

- There must be a valid task state segment (TSS) for the new task. The stack pointers in the TSS for privilege levels numerically less than or equal to the initial CPL must point to valid stack segments.
- The task register must point to an area in which to save the current task state. After the first task switch, the information dumped in this area is not needed, and the area can be used for other purposes.

Introduction to Debugging in 80386

Debugging on the 80386 was one of the early examples of **hardware-assisted debugging**, where the processor itself played an active role in helping developers diagnose and resolve issues in their code. Let's break down its major components and concepts:

Hardware Debug Registers

The 80386 introduced special registers dedicated entirely to debugging purposes. These include:

- **Debug Address Registers (DR0–DR3):**
  Each of these registers holds a linear memory address that you want to monitor. For instance, if you suspect that a particular variable at a certain memory location is being corrupted, you can set a breakpoint on that address. Because these registers store linear addresses, they work correctly whether or not paging (virtual memory) is enabled.
- **Debug Control Register (DR7):**
  This register is the control center for debugging. In DR7, you define how and when the breakpoints set in DR0–DR3 should trigger an exception. Here, you can specify:
  - The **type of access** that should cause a breakpoint. For example, you can choose to break only on instruction execution, only on data writes, or on both data reads and writes.
  - The **length of the memory area** to be monitored. Breakpoints can be set to trigger on accesses of 1, 2, or 4 bytes.
  - The **scope** of the breakpoint using local and global enable bits. Local enable bits are automatically cleared during a task switch to prevent breakpoints from affecting other tasks, while global enable bits persist across switches.

These registers make it possible to set hardware breakpoints rather than relying solely on software breakpoints (which might involve modifying the code being executed), thus providing a less intrusive and more efficient way of monitoring program behaviour.

Exception Handling and Breakpoint Triggers

When the processor detects that a specified condition has been met—say, the execution of an instruction at an address stored in DR0 or the data write to a location you're monitoring—it generates a debug exception. This exception temporarily halts the normal flow of execution and hands control over to a debugger or an exception handler.

- **Debug Exceptions:**
  These are precise events where the CPU signals that a monitored condition has occurred. A debug exception gives control to software (like a debugger) so that the state of the processor (registers, memory, etc.) can be inspected. This mechanism is crucial for stepping through code, inspecting variables, and understanding the exact circumstances under which an error takes place.
- **Privilege and Safety:**
  Only software running at the highest privilege (ring 0) is allowed to access these debug registers. This restriction helps protect the system from malicious or accidental misuse of the debugging hardware. Attempts to access these registers from a lower privilege level result in general protection faults, which safeguard the stability of the operating system.

How It Enhances Debugging

The 80386's approach to debugging provided several benefits:

1. **Precision:** By using hardware breakpoints defined with fine granularity (specifying exact addresses, access types, and data lengths), developers could zero in on elusive bugs in a way that was far more precise than software breakpoints.
2. **Performance:** Hardware breakpoints impose minimal performance overhead because they don't require altering the code or using software interrupts continuously.
3. **Versatility:** With the ability to monitor memory accesses (not just instruction fetches), the 80386 allowed for a broader range of debugging scenarios, such as catching unintended memory writes or ensuring that certain read/write operations occurred as expected.
4. **Task Isolation:** The distinction between local and global breakpoints allowed a debugger to control debugging behavior across different execution contexts, especially when the system was multitasking or running multiple processes concurrently.

Together, these features helped pave the way for later, more advanced debugging tools and techniques, laying the foundation for the sophisticated integrated debugging environments available in modern processors.

Beyond the Registers

While the core of debugging on the 80386 revolves around its debug registers, other features—like the use of the **trap flag** for single-stepping—also contributed to a richer debugging experience. The trap flag, part of the EFLAGS register, can be set to generate an exception after each instruction is executed. This is particularly useful for step-by-step debugging, allowing developers to inspect the state of the system meticulously after every instruction.

Furthermore, the 80386's integration of debugging facilities at the hardware level meant that the debugger's intervention was tightly coupled with the processor's operational flow, making it less likely that bugs would slip by unnoticed or that the overhead of debugging would disrupt normal program execution too severely.

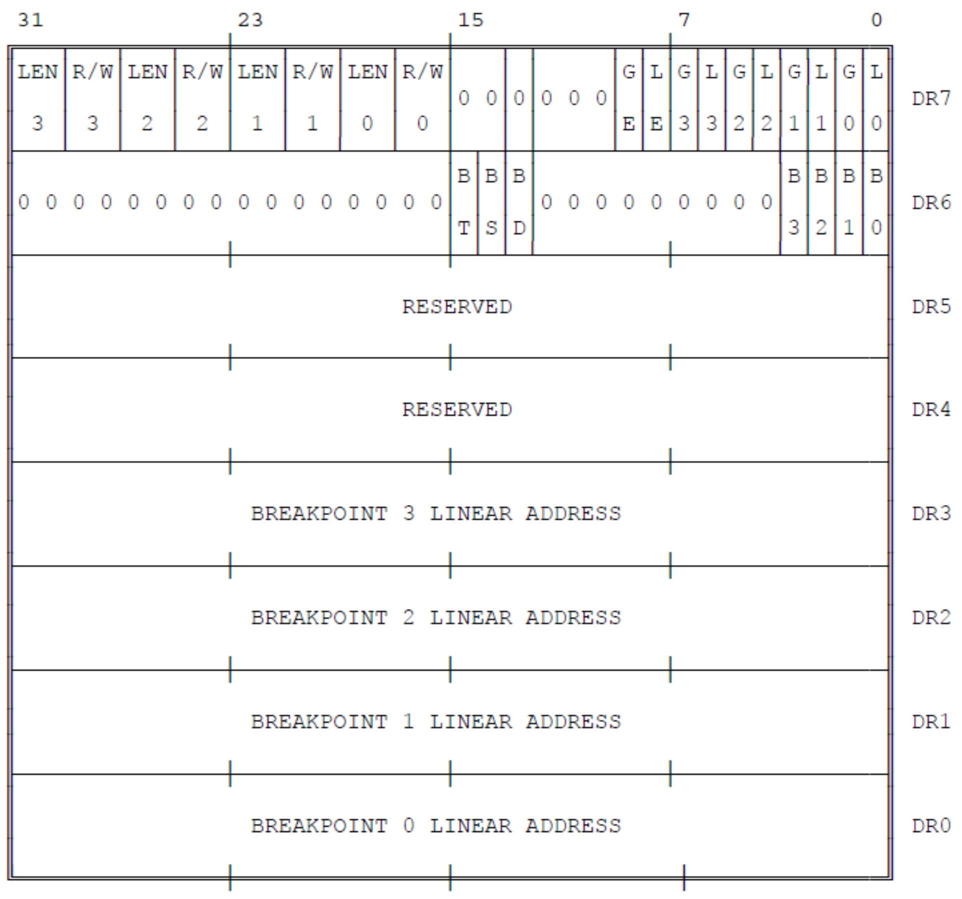
In Summary

Debugging in the 80386 is a prime example of early hardware-assisted diagnostic tools. The design of dedicated debug registers—especially DR0–DR3 for holding addresses, and DR7 for controlling the conditions that trigger interrupts—enabled developers to detect and investigate faults in a controlled, efficient, and precise manner. This system not only improved the reliability and debuggability of software but also set standards that have influenced x86 architectures ever since.

Debug Registers

- Six 80386 registers are used to control debug features. These registers are accessed by variants of the MOV instruction.
- A debug register may be either the source operand or destination operand. The debug registers are privileged resources; the MOV instructions that access them can only be executed at **privilege level zero**.
- An attempt to read or write the debug registers when executing at any other privilege level causes a general protection exception. Figure 12-1 shows the format of the debug registers

Figure 12-1. Debug Registers



NOTE
0 MEANS INTEL RESERVED. DO NOT DEFINE.

Debug Address Registers (DR0-DR3)

- Each of these registers contains the linear address associated with one of four breakpoint conditions. Each breakpoint condition is further defined by bits in DR7.
- The debug address registers are effective whether or not paging is enabled.
- The addresses in these registers are linear addresses. If paging is enabled, the linear addresses are translated into physical addresses by the processor's paging mechanism (as explained in Chapter 5). If paging is not enabled, these linear addresses are the same as physical addresses.
- Note that when paging is enabled, different tasks may have different linear-to-physical address mappings. When this is the case, an address in a debug address register may be relevant to one task but not to another. For this reason, **the 80386 has both global and local enable bits in DR7. These bits indicate whether a given debug address has a global (all tasks) or local (current task only) relevance.**

Debug Control Register (DR7)

The debug control register shown in Figure 12-1 both helps to define the debug conditions and selectively enables and disables those conditions.

For each address in registers DR0-DR3, the corresponding fields **R/W0 through R/W3** specify the type of action that should cause a breakpoint. 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

Fields LEN0 through LEN3 specify the length of data item to be monitored. A length of 1, 2, or 4 bytes may be specified. The values of 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. Any other length is undefined.

- The **low-order eight bits of DR7 (L0 through L3 and G0 through G3)** selectively enable the four address breakpoint conditions.
- There are two levels of enabling: the local (L0 through L3) and global (G0 through G3) levels.
- The local enable bits are automatically reset by the processor at every task switch to avoid unwanted breakpoint conditions in the new task.
- The global enable bits are not reset by a task switch; therefore, they can be used for conditions that are global to all tasks.
- The **LE and GE bits** control the "exact data breakpoint match" feature of the processor. If either LE or GE is set, the processor slows execution so that data breakpoints are reported on the instruction that causes them. It is recommended that one of these bits be set whenever data breakpoints are armed. The processor clears LE at a task switch but does not clear GE.

Debug Status Register (DR6)

The debug status register shown in Figure 12-1 permits the debugger to determine which debug conditions have occurred.

- When the processor detects an enabled debug exception, it sets the low-order bits of this register (**B0 thru B3**) before entering the debug exception handler.
- Bn is set if the condition described by DRn, LENn, and R/Wn occurs. (Note that the processor sets Bn regardless of whether Gn or Ln is set.
- If more than one breakpoint condition occurs at one time and if the breakpoint trap occurs due to an enabled condition other than n, Bn may be set, even though neither Gn nor Ln is set.)
- The **BT bit** is associated with the T-bit (debug trap bit) of the TSS (refer to 7 for the location of the T-bit).
- The processor sets the BT bit before entering the debug handler if a task switch has occurred and the T-bit of the new TSS is set.
- The **BS bit** is associated with the TF (trap flag) bit of the EFLAGS register.
- The BS bit is set if the debug handler is entered due to the occurrence of a single-step exception.
- The single-step trap is the highest-priority debug exception; therefore, when BS is set, any of the other debug status bits may also be set.
- The **BD bit** is set if the next instruction will read or write one of the eight debug registers

Virtual 8086 Mode (V86) Mode

The purpose of a V86 task is to form a "virtual machine" with which to execute an 8086 program. A complete virtual machine consists not only of 80386 hardware but also of systems software. Thus, the emulation of an 8086 is the result of cooperation between hardware and software:

- The hardware provides a virtual set of registers (via the TSS), a virtual memory space (the first megabyte of the linear address space of the task), and directly executes all instructions that deal with these registers and with this address space.
- The software controls the external interfaces of the virtual machine (I/O, interrupts, and exceptions) in a manner consistent with the larger environment in which it executes. In the case of I/O, software can choose either to emulate I/O instructions or to let the hardware execute them directly without software intervention.

Software that helps implement virtual 8086 machines is called a V86 monitor.

Executing 8086 Code

The processor executes in V86 mode when the **VM (Virtual Machine) bit** in the EFLAGS register is set. The processor tests this flag under two general conditions:

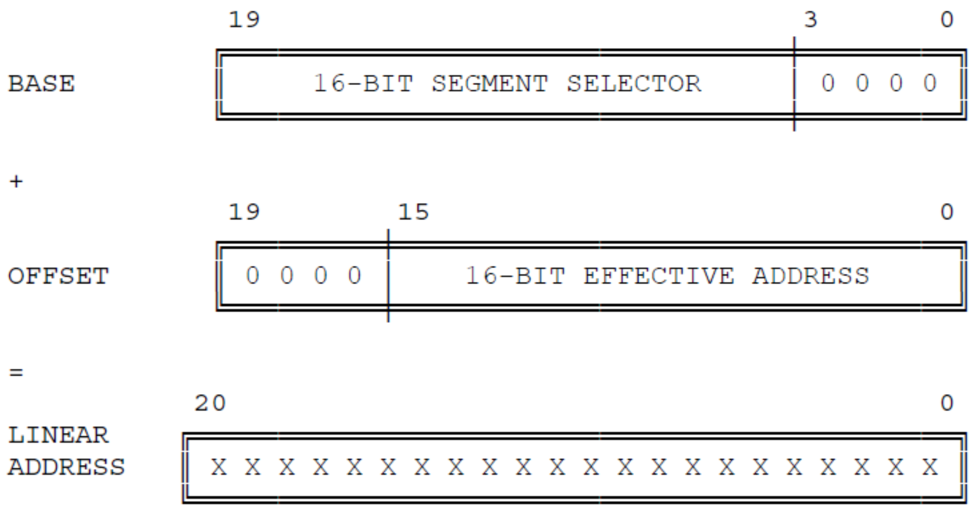
1. When loading segment registers to know whether to use 8086-style address formation.
2. When decoding instructions to determine which instructions are sensitive to IOPL.

Except for these two modifications to its normal operations, the 80386 in V86 mode operated much as in protected mode.

Address Formation in V86 Mode

- In V86 mode, the 80386 processor does not interpret 8086 selectors by referring to descriptors; instead, it forms linear addresses as an 8086 would.
- It shifts the selector left by four bits to form a 20-bit base address.
- The effective address is extended with four high-order zeros and added to the base address to create a linear address as Figure 15-1 illustrates.
- Even though the linear address is calculated using the real-mode formula, if paging is enabled, this linear address undergoes the standard page translation process. This means that behind the scenes, the operating system can remap memory, enforce protection, and provide isolation between tasks—even for code that believes it is operating in real mode.

Figure 15-1. V86 Mode Address Formation



Note: When you are asked to compare Real Mode (8086 mode), Protected Mode and Virtual 8086 Mode, first compare all 3 based on address formation techniques and then other points like structure of task.

Structure of V86 Tasks

- In its protected mode of operation, 80386DX provides a virtual 8086 operating environment to execute the 8086 programs.
- The real mode also can be used to execute the 8086 programs along with the capabilities of 80386, like protection and a few additional instructions.
- However, once the 80386 enters the protected mode from the real mode, it cannot return back to the real mode without a reset operation. Thus, the virtual 8086 mode of operation of 80386, offers an advantage of executing 8086 programs while in protected mode.
- The address forming mechanism in virtual 8086 mode is exactly identical with that of 8086 real mode.
- **In virtual mode, 8086 can address 1 Mbytes of physical memory that may be anywhere in the 4 Gbytes address space** of the protected mode of 80386.
- **Like 80386 real mode, the addresses in virtual 8086 mode lie within 1 Mbytes of memory.**
- In the virtual mode, the paging mechanism and protection capabilities are available at the service of the programmers (note that the 80386 supports multiprogramming, hence more than one programmer may use the CPU at a time).
- Paging unit may not be necessarily enabled in the virtual mode, but may be needed to run the 8086 programs which require more than 1 Mbyte of memory for memory
- management functions.
- In the virtual mode, **the paging unit allows only 256 pages**, each of 4 Kbytes size. Each of the pages may be located anywhere within the maximum 4 GBytes physical memory.
- The virtual mode allows the multiprogramming of 8086 applications. Figure 10.10 shows how the memory is managed in multitasking virtual 8086 environment.
- The **virtual 8086 mode executes all the programs at the privilege level 3**. Any of the other programmes may deny access to the virtual mode programs or data.
- However, **the real mode programs are executed at the highest privilege level, i.e. level 0**. Note that the instructions to prepare the processor for protected mode can only be executed at level 0.

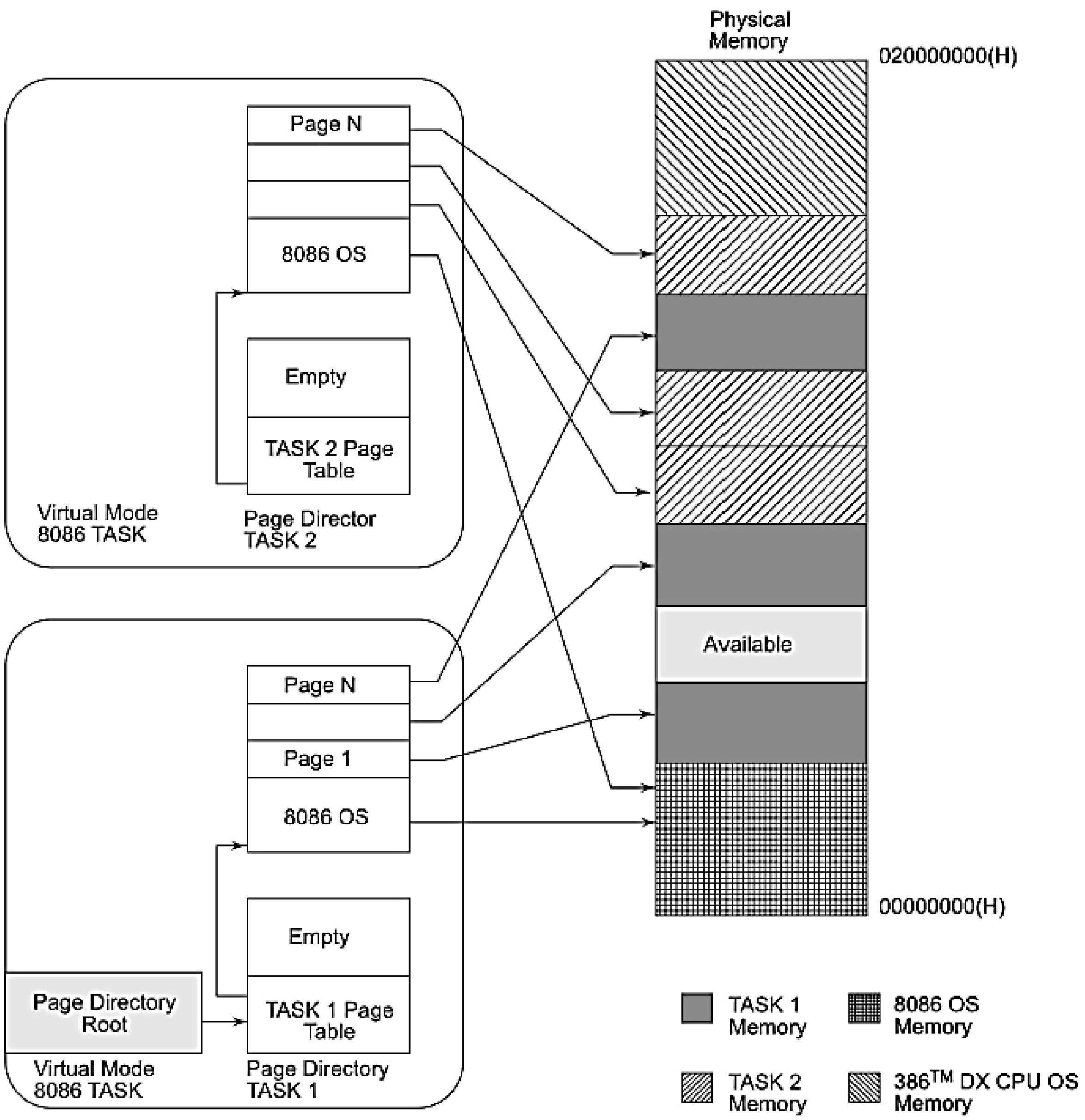


Fig. 10.10 *Memory Management in Virtual 8086 Mode (Multitasking) (Intel Corp.)*

Entering And Leaving V86 Mode

Figure 15-2 summarizes the ways that the processor can enter and leave an 8086 program.

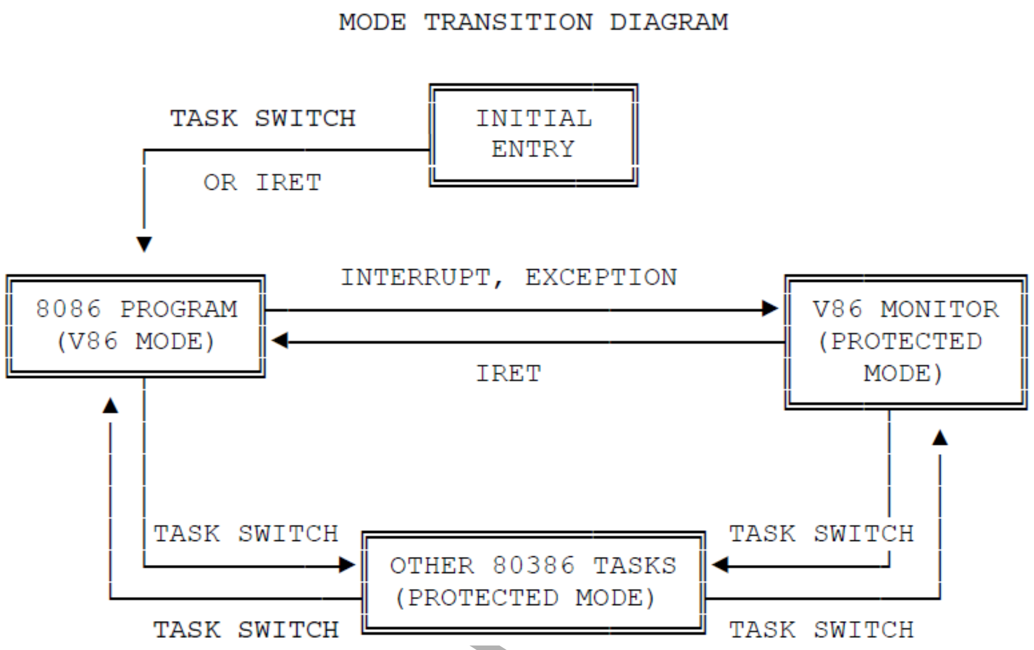
The processor can enter V86 by either of two means:

1. A task switch to an 80386 task loads the image of EFLAGS from the new TSS. A value of one in the VM bit of the new EFLAGS indicates that the new task is executing 8086 instructions.
2. An IRET from a procedure of an 80386 task loads the image of EFLAGS from the stack. A value of one in VM in this case indicates that the procedure to which control is being returned is an 8086 procedure. The CPL at the time the IRET is executed must be zero, else the processor does not change VM.

The processor leaves V86 mode when an interrupt or exception occurs. Specified by following two cases:

1. The interrupt or exception causes a task switch. If the new TSS is an 80386 TSS and the VM bit in the EFLAGS image is zero, then the processor clears the VM bit of EFLAGS, loads the segment registers from the new TSS using 80386-style address formation in protected mode.
2. The interrupt or exception vectors to a privilege-level zero procedure. The processor stores the current setting of EFLAGS on the stack, then clears the VM bit.

Figure 15-2. Entering and Leaving the 8086 Program



Numeric Coprocessor (80387) Support

Kindly note that 80387 is also referred as NDP – Numeric Data Processor.

Control Register (CR0) bits for Coprocessor Support

- The **ET bit** of CR0 indicates which type of coprocessor is present. ET=1 indicates presence of 80387 (32-bit). ET=0 indicates presence of 80287 (16-bit).
- The **MP (monitor coprocessor)** bit indicates whether a coprocessor is actually attached.
- The **EM bit** indicates whether coprocessor functions are to be emulated.

---NOT PART OF SYLLABUS---

The 80386 offers a built-in mechanism for handling coprocessor (floating point) instructions that is especially useful if an 80387 isn't present. When a program issues an 80387 instruction—typically identified by an opcode beginning with the bit pattern used for ESC (escape) instructions—the 80386 checks specific control flags in its CR0 register to decide how to handle it. Here's how it works in detail:

1. **Detection via Control Flags**
   The 80386's CR0 register contains flags such as EM (Emulation) and MP (Monitor Coprocessor). When the processor encounters an ESC instruction, it first examines the EM flag.
   - If the **EM flag is set** (indicating that hardware coprocessor functionality is being emulated), the processor raises exception 7. This exception informs the operating system or a dedicated emulator that an FPU instruction needs to be handled in software—effectively "intercepting" the instruction meant for an 80387.
   - In addition, the **TS (Task Switched) flag** is used to track whether there has been a recent context switch involving the floating point unit. This ensures that the software emulator has the correct state for the FPU registers before processing the instruction.
2. **Instruction Redirection**
   The ESC (escape) instructions are explicitly designed to be diverted to the coprocessor. In a system with an actual 80387 installed, the instruction would be sent directly to it. However, when the EM flag is set, the 80386 diverts these ESC instructions to an exception handler. The handler then:
   - Decodes the instruction internally.
   - Performs the equivalent floating point operation using software routines, mimicking the behavior of the hardware coprocessor.
   - Updates the floating point state accordingly, using the standard protocols defined by the 80387 (or the 80287 if the processor is set to that mode based on the ET bit).
3. **Software Emulation Strategy**
   The design of this emulation mode allows system software (such as the operating system or dedicated libraries) to provide floating point support even on machines that don't have a physical 80387 installed. This makes it possible to run floating point applications on a broader range of hardware, albeit with potential performance trade-offs since software emulation is generally slower than executing instructions on dedicated hardware.

Register set of 80387

- Intel's 80387 has eight 80-bit floating point data registers, which are used to store signed 80-bit data in the form of exponent and significand as shown in Fig. 10.13. Each of these registers has a corresponding 2-bit tag field.
- The 80387 has a 16-bit control, status and tag word registers.
- The 80387 has two more 48-bit registers known as instruction and data pointers. The instruction and data pointer registers respectively point to the failing math coprocessor instruction and the corresponding numeric data, which is referred by the CPU.
- Two bits are allotted for each of the registers R0-R7 in the tag word. These are used to optimize the performance of the coprocessor by identifying between the empty and non-empty of the R0-R7 registers.
- Also, the tag bits can be used by the exception handlers to check the contents of a stack location without any manipulation.
- The status word represents the overall status of the coprocessor.

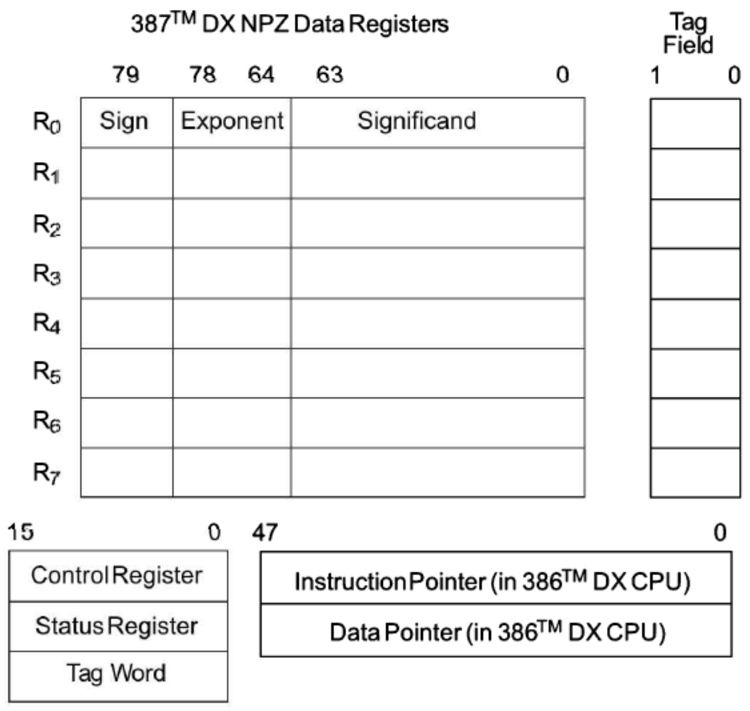


Fig. 10.13 ***Register Format of 80387***

Load & Store Instructions

These instructions are also known as data movement instructions.

Table 4-1. Data Transfer Instructions

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

FLD

Format- 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. ST(7) must be empty to avoid causing an invalid-operation exception.

The new stack top is tagged nonempty. The source may be a register on the stack (ST(i)) or any of the real data types in memory. If the source is a register, the register number used is that before TOP is decremented by the instruction. Coding FLD ST(0) duplicates the stack top. Single and double real source operands are converted to extended real automatically. Loading an extended real operand does not require conversion; therefore, the I and D exceptions do not occur in this case.

FST

Format- FST destination

FST (store real) copies the 80387 stack top to the destination, which may be another register on the stack or a single or double (but not extended-precision) memory operand. If the destination is single or double real, the copy of the significand is rounded to the width of the destination according to the RC field of the control word, and the copy of the exponent is converted to the width and bias of the destination format. The over/underflow condition is checked for as well.

FSTP

Format- FSTP destination

FSTP (store real and pop) operates identically to FST except that the 80387 stack is popped following the transfer. This is done by tagging the top stack element empty and then incrementing TOP. FSTP also permits storing to an extended-precision real memory variable, whereas FST does not. If the source operand is a register, the register number used is that before TOP is incremented by the instruction. Coding FSTP ST(0) is equivalent to popping the stack with no data transfer.

FXCH

Format- FXCHG destination

FXCH (exchange registers) swaps the contents of the destination and the stack top registers. If the destination is not coded explicitly, ST(l) is used. Many 80387 instructions operate only on the stack top; FXCH provides a simple means of effectively using these instructions on lower stack elements. For example, the following sequence takes the square root of the third register from the top (assuming that ST is nonempty):

```
FXCH ST(3)

FSQRT

FXCH ST(3)
```

FILD

Format- 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 80387 stack. ST(7) must be empty to avoid causing an exception. The (new) stack top is tagged nonempty. FILD is an exact operation; the source is loaded with no rounding error.

FIST

Format- 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. The destination may define a word or short integer variable. Negative zero is stored in the same encoding as positive zero: 0000 ... 00.

FISTP

Format- FISTP destination

FISTP (integer and pop) operates like FIST except that it also pops the 80387 stack following the transfer. The destination may be any of the binary integer data types.

FBLD

Format- 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. The sign of the source is preserved, including the case where the value is negative zero. FBLD is an exact operation; the source is loaded with no rounding error.

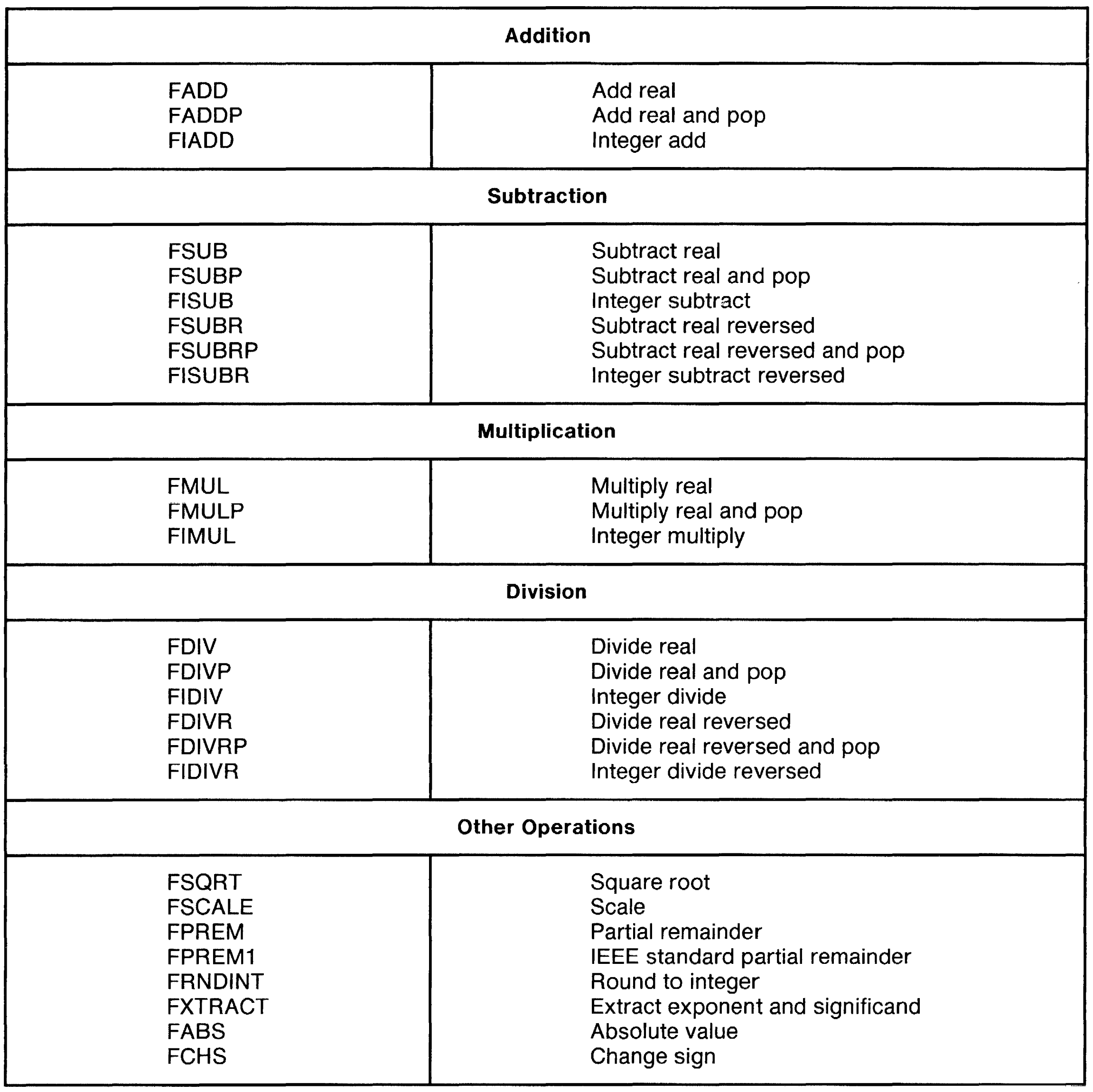
The packed decimal digits of the source are assumed to be in the range 0-9. The instruction does not check for invalid digits (A-FH), and the result of attempting to load an invalid encoding is undefined.

FBSTP

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. FBSTP rounds a non-integral value according to the RC (rounding control) field of the control word.

Non-Transcendental Instructions

Table 4-2. Nontranscendental Instructions



Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
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
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
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
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

Addition

FADD *source/destination, source*

FADDP *destination, source*

FIADD *source*

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(0)

Normal Subtraction

FSUB source/destination, source

FSUBP destination, source

FISUB source

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.

Reversed Subtraction

FSUBR *source/destina tion, source*

FSUBRP *destination, source*

FISUBR *source*

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

FMUL source/destination,source

FMULP destination, source

FIMUL source

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.

Normal Division

FDIV source/destination, source

FDIVP destination, source

FIDIV source

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.

Reversed Division

FDIVR Source/destination, source

FDIVRP destination, source

FIDIVR source

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.

FSQRT

FSQRT (square root) replaces the content of the top stack element with its square root. (Note: The square root of -0 is defined to be -0.)

FSCALE

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

$ST \leftarrow ST.\ 2^{ST(1)}$

Thus, FSCALE provides rapid multiplication or division by integral powers of 2. It is particularlyuseful for scaling the elements of a vector.

FPREM- Partial Remainder (80287/8087-Compatible)

FPREM computes the remainder of division of ST by ST(1) and leaves the result in ST. FPREM finds a remainder REM and a quotient Q such that

REM = ST - ST(1)*Q

FPREM1-Partial Remainder (IEEE Std. 754-Compatible)

FPREM 1 computes the remainder of division of ST by ST(1) and leaves the result in ST. FPREM1 finds a remainder REM1 and a quotient Q1 such that

REM1 = ST - ST(1)*Q1

FRNDINT

FRNDINT (round to integer) rounds the top stack element to an integer according to the RC bits of the control word. For example, assume that ST contains the 80387 real number encoding of the decimal value 155.625. FRNDINT will change the value to 155 if the RC field of the control word is set to down or chop, or to 156 if it is set to up or nearest.

FXTRACT

FXTRACT (extract exponent and significand) performs a superset of the IEEE recommended logb(x) function by "decomposing" the number in the stack top into two numbers that represent the actual value of the operand's exponent and significand fields.

The "exponent" replaces the original operand on the stack and the "significand" is pushed onto the stack.

FABS

FABS (absolute value) changes the top stack element to its absolute value by making its sign positive. Note that the invalid-operation exception is not signalled even if the operand is a signalling NaN or has a format that is not supported.

FCHS

FCHS (change sign) complements (reverses) the sign of the top stack element. Note that the invalid-operation exception is not signalled even if the operand is a signalling NaN or has a format that is not supported.

Interfacing signals of 80386 with 80387

- Figure 10.14 shows the interfacing of 80387 with 80386.
- The pins BUSY#, ERROR#, PEREQ, ADS#, W/R# and D0-D31 of 80387 can be directly connected with the corresponding pins of 80386.
- A separate clock generator may be added to the circuit, if 80387 is required to be run at a different frequency than the 80386.
- Otherwise, the 80387 may be driven by the same frequency generator that drives 80386.
- An optional wait state generator combines the ready signal from 80387 and ready signals given by the other peripherals to push the 80386 into wait state till 80387 execution is over.
- The NPS, and NPS2 (Numeric Processor Select) lines are directly connected with M/IO and A31 respectively to inform the 80387 that the CPU wants to communicate with it (NPS1) and it is using one of the reserved I/O addresses for 80387 (NPS2), i.e. 800000F8

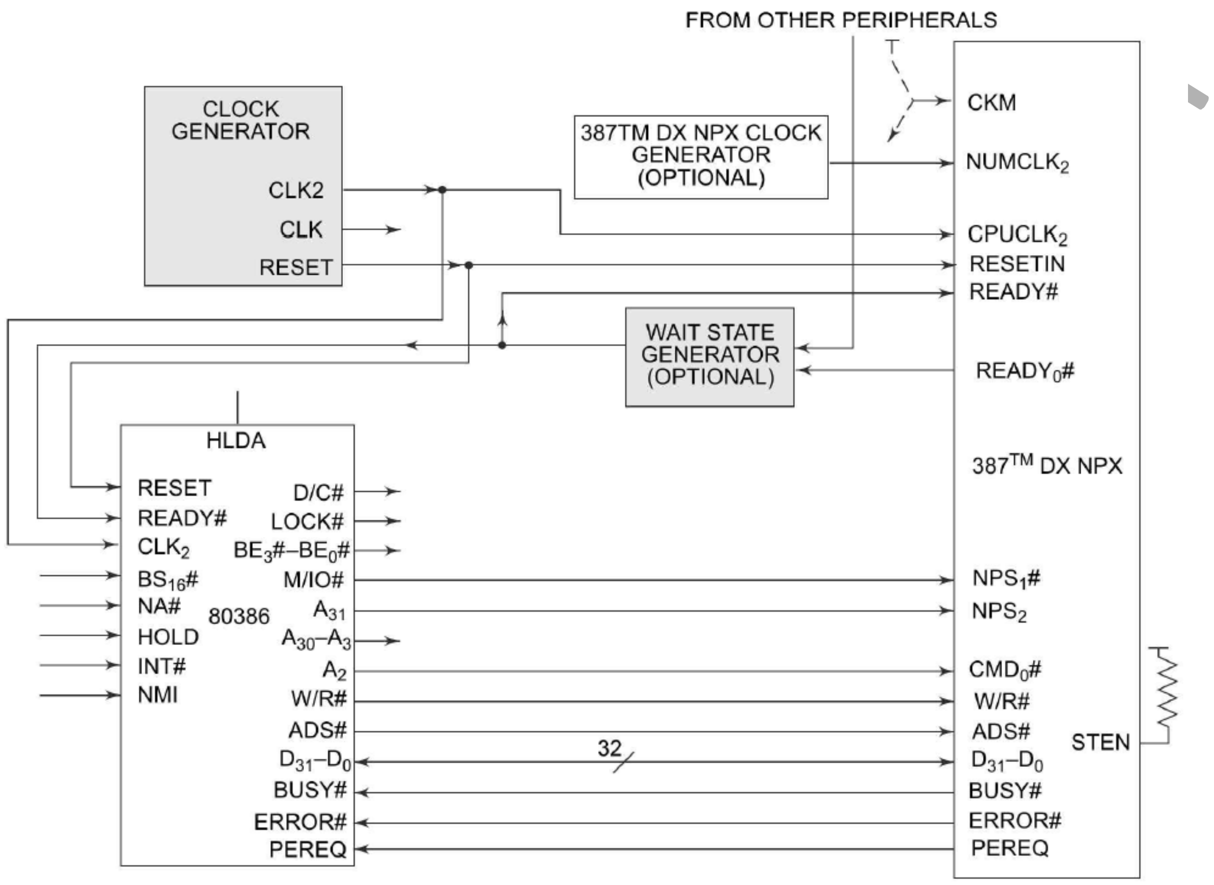


Fig. 10.14 *Interconnections of 80387 with 80386*