For example, suppose that four 32K-byte SRAMs are added to the middle memory area, beginning at location 80000H and ending at location 9FFFFH with no wait states. To program the middle memory selection lines for this area of memory, we place the leftmost seven address bits in register A6H, with bits 8–3 containing logic 0s, and the rightmost three bits containing the ready control bits. For this example, register A6H is loaded with 8004H. Register A8H is programmed with a 1F44H, assuming that *EX* = 0 and *MS* = 1 and no wait states and no READY are required for the peripherals.

Register A4H programs the peripheral chip selection pins (PCS6–PCS0) along with the EX and MS bits of register A8H. Register A4H holds the beginning or base address of the peripheral selection lines. The peripherals may be placed in memory or in the I/O map. If they are placed in the I/O map, A19–A16 of the port number must be 0000. Once the starting address is programmed on any 1K-byte I/O address boundary, the PCS pins are spaced at 128-byte intervals.

For example, if register A4H is programmed with a 0204H, with no waits and no READY synchronization, the memory address begins at 02000H or the I/O port begins at 2000H. In this case, the I/O ports are: PCS0 = 2000H, PCS1 = 2080H, PCS2 = 2100H, PCS3 = 2180H, PCS4 = 2200H, PCS5 = 2280H, and PCS6 = 2300H.

The MS bit of register A8H selects memory-mapping or I/O mapping for the peripheral select pins. If MS is a logic 0, then the PCS lines are decoded in the memory map; if it is a logic 1, then the PCS lines are in the I/O map.

The EX bit selects the function of the PCS5 and PCS6 pins. If *EX* = 1, these PCS pins select I/O devices; if *EX* = 0, these pins provide the system with latched address lines A1 and A2. The A1 and A2 pins are used by some I/O devices to select internal registers and are provided for this purpose.

Programming the Chip Selection Unit for EB and EC Versions. As mentioned earlier, the EB and EC versions have a different chip selection unit. These newer versions of the 80186/80188 contain an upper and lower memory chip selection pin as do earlier versions, but they do not contain middle selection and peripheral selection pins. In place of the middle and peripheral chip selection pins, the EB and EC versions contain eight general chip selection pins (GCS7–GCS0) that select either a memory device or an I/O device.

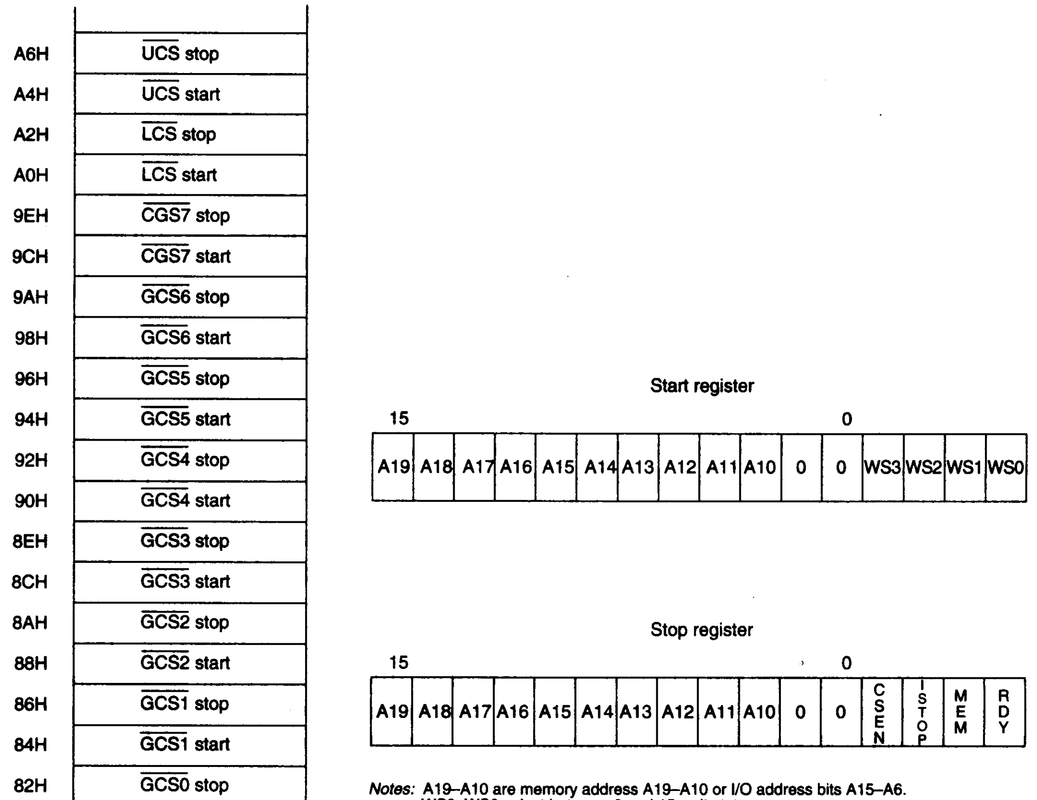
Programming is also different because each of the chip selection pins contains a starting address register and an ending address register. See Figure 14–23 for the offset address of each pin and the contents of the start and end registers.

Notice that programming for the EB and EC versions of the 80186/80188 are much easier than for the earlier XL and XA versions. For example, to program the UCS pin for an address that begins at location F0000H and ends at location FFFFFH (64K bytes), the starting address register (*offset* = A4H) is programmed with F002H for a starting address of F0000H with two wait states. The ending address register (*offset* = A6H) is programmed with 000EH for an ending address of FFFFFH for memory with no external ready synchronization. The other chip selection pins are programmed in a similar fashion.

14–3 80C188EB EXAMPLE INTERFACE

Because the 80186/80188 microprocessors are designed as embedded controllers, this section of the text provides an example of such an application. The example illustrates simple memory and I/O attached to the 80C188EB microprocessor. It also lists the software required to program the 80C188EB and its internal registers after a system reset. The software to control the system itself is not provided. Figure 14–24 illustrates the pin-out of the 80C188EB version of the 80188 microprocessor. Note the differences between this version and the XL version presented earlier in the text

The 80C188EB version contains some new features that were not present on earlier versions. These features include two I/O ports (P1 and P2) that are shared with other functions and two serial communications interfaces that are built into the processor. This version does not contain a DMA controller, as did the XL version.



Notes: A19–A10 are memory address A19–A10 or I/O address bits A15–A6.
WS3–WS0 select between 0 and 15 wait states.
CSEN enables the pin if CSEN = 1.
ISTOP = if ISTOP = 1 the memory address is OFFFFFH or the I/O address is OFFFFH.
MEM = MEM = 1 selects memory and MEM = 0 selects I/O.
RDY = enables external ready if RDY = 1 for more than 15 wait states.

FIGURE 14–23 The chip selection unit in the EB and EC versions of the 80186/80188.

The 80188 can be interfaced with a small system designed to be used as a microprocessor trainer. The trainer illustrated in this text uses a 27256 EPROM for program storage, three 62256 SRAMs for data storage, an 8279 programmable keyboard/display interface, and one of the built-in serial ports for serial communications. Figure 14–25 illustrates a small microprocessor trainer that is based on the 80C188EB microprocessor.

Memory is selected by the UCS pin for the 27256 EPROM and the LCS pin for one of the 62256 SRAMs; the GCS0 and GCS1 pins select the remaining SRAM devices. The 8270 keyboard/display peripheral is selected by GCS2. Note that five wait states are programmed for the EPROM, assuming a really slow 450 ns EPROM, two waits for the 250 ns SRAM, and two waits for the 8279 keyboard/display interface. Faster EPROM and SRAM reduce or eliminate the number of waits required for the memory.

The system places the EPROM at memory addresses F8000H–FFFFFH; the SRAM at 00000H–07FFFH, 80000H–87FFFH, and 88000H–8FFFFH; and the 8279 at I/O ports 1000H–107FH. In this system, as is normally the case, we do not modify the address of the peripheral control block, which resides at I/O ports FF00H–FFFFH.

Example 14–5 lists the software required to initialize the 80C188EB microprocessor. It does not list any of the software required to program the 8279, nor does it show the software required to operate the system as a microprocessor-based trainer.

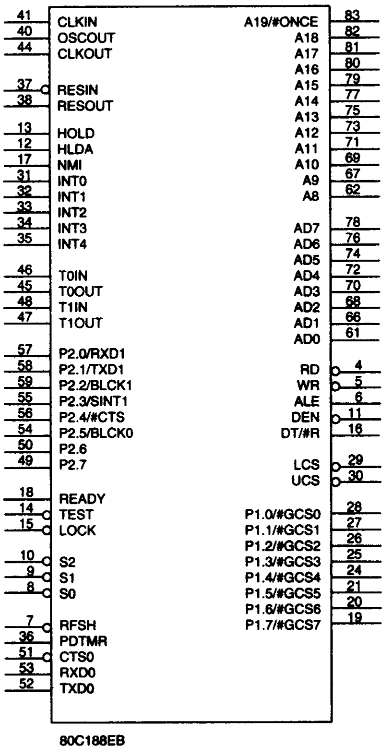


FIGURE 14–24 The pin-out of the 80C188EB version of the 80188 microprocessor.

EXAMPLE 14–5

```
                    .MODEL SMALL
                    .186
0000                .CODE
                    ;A program that initializes the 80C188EB.
                    ;
                             ORG    8000H          ;start of EPROM
                    ;
8000                MAIN:

8000  BA FFA6                MOV    DX,0FFA6H      ;address UCS stop
8003  B8 000E                MOV    AX,000EH       ;FFFFFH

8006  EE                     OUT    DX,AL          ;set stop address for UCS

8007  BA FFA0                MOV    DX,0FFA0H      ;address LCS start
800A  B8 0002                MOV    AX,0002H       ;00000H with 2 waits
800D  EE                     OUT    DX,AL          ;set start address for SRAM U4

800E  BA FFA2                MOV    DX,0FFA2H      ;address LCS stop
8011  B8 080A                MOV    AX,080AH       ;07FFFH
8014  EE                     OUT    DX,AL          ;set stop address for SRAM U4

8015  BA FF80                MOV    DX,0FF80H      ;address GCS0 start
8018  B8 0802                MOV    AX,0802H       ;08000H with 2 waits
```

```
801B  EE                        OUT    DX,AL          ;set start address for SRAM U5

801C  BA FF82                   MOV    DX,0FF82H      ;address GCS0 stop
801F  B8 100A                   MOV    AX,100AH       ;0FFFFH
8022  EE                        OUT    DX,AL          ;set stop address for SRAM U5

8023  BA FF84                   MOV    DX,0FF84H      ;address GCS1 start
8026  B8 1002                   MOV    AX,1002H       ;10000H with 2 waits
8029  EE                        OUT    DX,AL          ;set start address for SRAM U6

802A  BA FF86                   MOV    DX,0FF86H      ;address GCS1 stop
802D  B8 180A                   MOV    AX,180AH       ;17FFFH
8030  EE                        OUT    DX,AL          ;set stop address for SRAM U6

8031  BA FF88                   MOV    DX,0FF88H      ;address GCS2 start
8034  B8 1002                   MOV    AX,1002H       ;1000H with 2 waits
8037  EE                        OUT    DX,AL          ;set start address for 8279

8038  BA FF8A                   MOV    DX,0FF8AH      ;address GCS2 stop
803B  B8 1048                   MOV    AX,1048H       ;103FH (I/O)
803E  EE                        OUT    DX,AL          ;set stop address for 8279

803F  BA FF60                   MOV    DX,0FF60H      ;address serial baud rate
8042  B8 8067                   MOV    AX,8067H       ;generate a 9600 Baud rate
8045  EE                        OUT    DX,AL          ;set Baud rate

8046  BA FF62                   MOV    DX,0FF62H      ;address serial control register
8049  B8 0059                   MOV    AX,59H         ;7 data, even parity, 1 stop
804C  EE                        OUT    DX,AL          ;set serial port

804D  BA FF66                   MOV    DX,0FF66H      ;address serial status register
8050  ED                        IN     AX,DX          ;clear serial port

8051  BA FF62                   MOV    DX,0FF62H      ;address serial control register
8054  ED                        IN     AX,DX          ;read control register
8055  83 C8 20                  OR     AX,20H         ;set REN bit
8058  EE                        OUT    DX,AL          ;enable serial port
                       ;
                       ;
                       ;Remainder of system software is placed at this point.
                       ;
                       ;
                                ORG    0FFF0H         ;reset location
                       ;
FFF0  BA FFA4                   MOV    DX,0FFA4H      ;address UCS start
FFF3  B8 F805                   MOV    AX,0F805H      ;F8000H with 5 waits

FFF6  EE                        OUT    DX,AL          ;set start address for UCS
FFF7  E9 8006                   JMP    MAIN           ;jump to start of EPROM
                                END
```

14–4 SUMMARY

1. The 80186/80188 microprocessors contain the same basic instruction set as the 8086/8088 microprocessors, except that a few additional instructions are added. The 80186/80188 are thus enhanced versions of the 8086/8088 microprocessors. The new instructions include PUSHA, POPA, INS, OUTS, BOUND, ENTER, LEAVE, and immediate multiplication and shift/rotate counts.

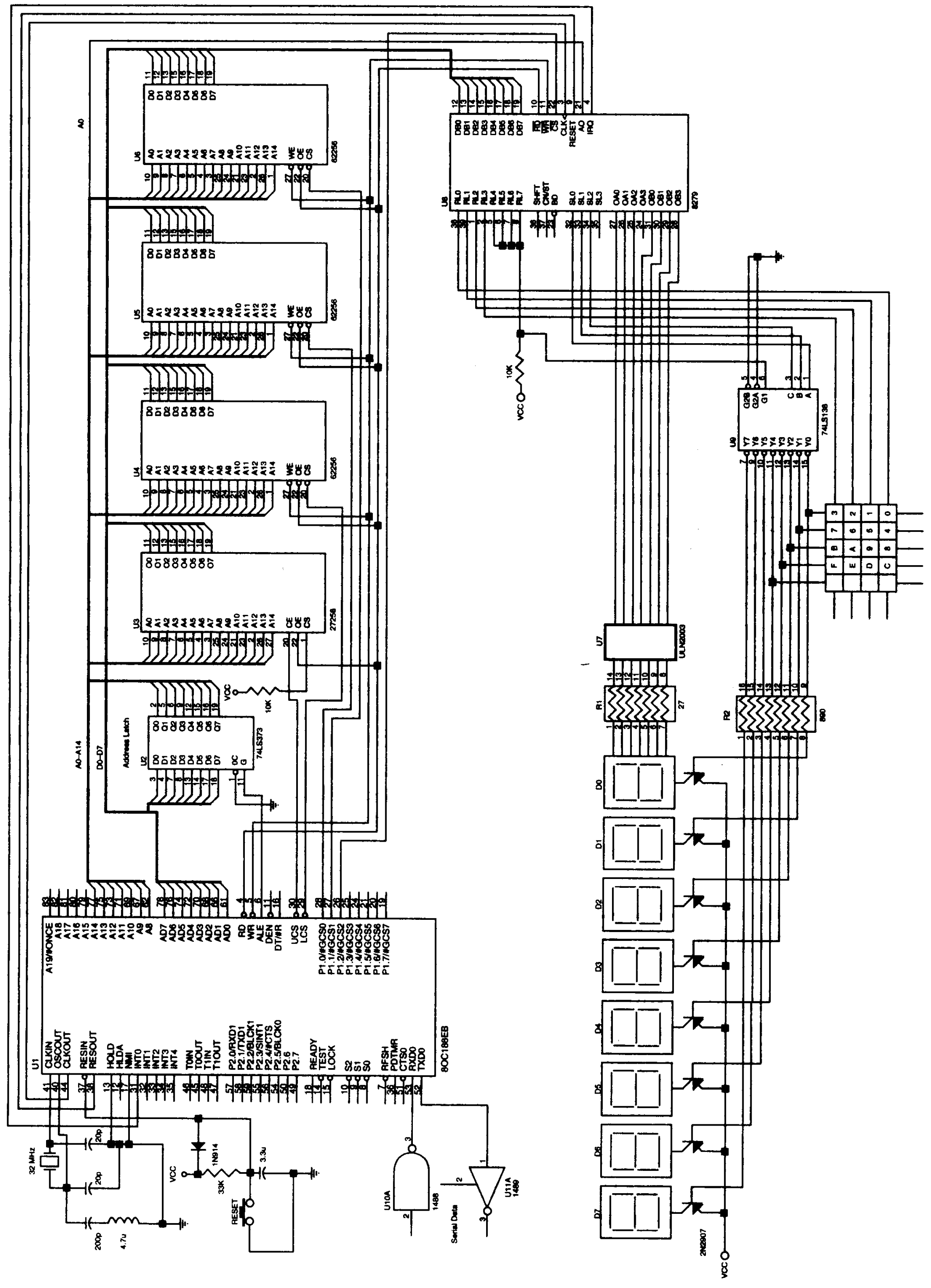


FIGURE 14–25 A small system using the 80C188EB embedded controller.

CHAPTER 15

The 80386 and 80486 Microprocessors

INTRODUCTION

The 80386 microprocessor is a full 32-bit version of the earlier 8086/80286 16-bit microprocessors, and represents a major advancement in the architecture—a switch from a 16-bit architecture to a 32-bit architecture. Along with this larger word size are many improvements and additional features. The 80386 microprocessor features multitasking, memory management, virtual memory (with or without paging), software protection, and a large memory system. All software written for the early 8086/8088 and the 80286 are upward-compatible to the 80386 microprocessor. The amount of memory addressable by the 80386 is increased from the 1M bytes found in the 8086/8088 and the 16M bytes found in the 80286, to 4G bytes in the 80386. The 80386 can switch between protected mode and real mode without resetting the microprocessor. Switching from protected mode to real mode was a problem on the 80286 microprocessor because it required a hardware reset.

The 80486 microprocessor is an enhanced version of the 80386 microprocessor that executes many of its instructions in one clocking period. The 80486 microprocessor also contains an 8K-byte cache memory and an improved 80387 numeric coprocessor. (Note that the 80486DX4 contains a 16K-byte cache.) When the 80486 is operated at the same clock frequency as an 80386, it performs with about a 50 percent speed improvement. In Chapter 18, we shall see that the Pentium and Pentium Pro, which both contain a 16K cache memory, perform at better than twice the speed of the 80486 microprocessor. The Pentium and Pentium Pro also contain improved numeric coprocessors that operate five times faster than the 80486 numeric coprocessor. Chapter 17 deals with additional improvements in the Pentium II–Pentium 4 microprocessors.

CHAPTER OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Contrast the 80386 and 80486 microprocessors with earlier Intel microprocessors.
2. Describe the operation of the 80386 and 80486 memory management unit and paging unit.
3. Switch between protected mode and real mode.
4. Define the operation of additional 80386/80486 instructions and addressing modes.
5. Explain the operation of a cache memory system.
6. Detail the interrupt structure and direct memory access structure of the 80386/80486.
7. Contrast the 80486 with the 80386 microprocessor.
8. Explain the operation of the 80486 cache memory.

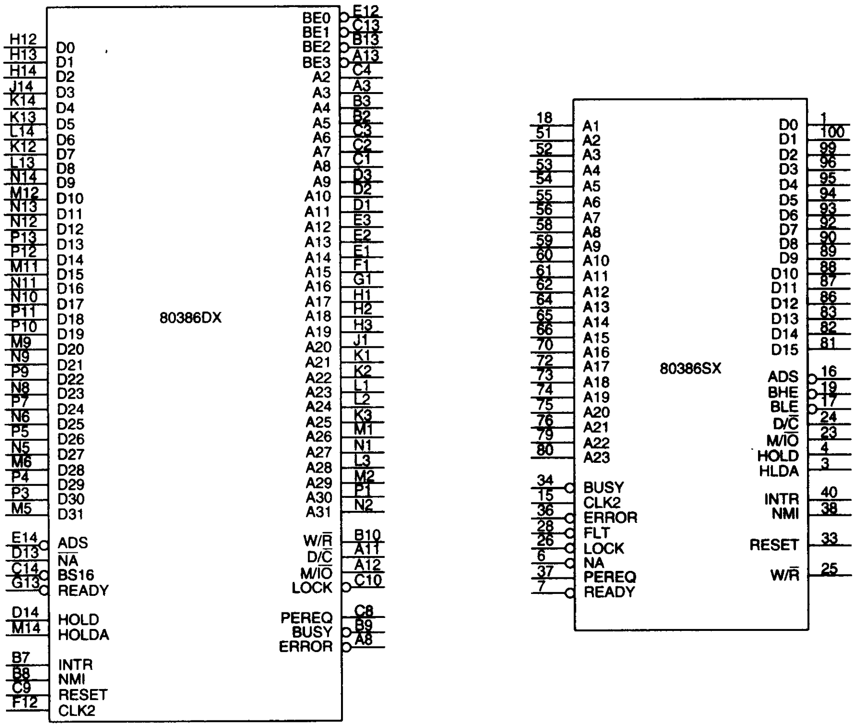


FIGURE 15–1 The pin-outs of the 80386DX and 80386SX microprocessors.

15–1 INTRODUCTION TO THE 80386 MICROPROCESSOR

Before the 80386 or any other microprocessor can be used in a system, the function of each pin must be understood. This section of the chapter details the operation of each pin, along with the external memory system and I/O structures of the 80386.

Figure 15–1 illustrates the pin-out of the 80386DX microprocessor. The 80386DX is packaged in a 132-pin PGA (pin grid array). Two versions of the 80386 are commonly available: the 80386DX, which is illustrated and described in this chapter; the other is the 80386SX, which is a reduced bus version of the 80386. A new version of the 80386—the 80386EX—incorporates the AT bus system, dynamic RAM controller, programmable chip selection logic, 26 address pins, 16 data pins, and 24 I/O pins. Figure 15–2 illustrates the 80386EX embedded PC.

The 80386DX addresses 4G bytes of memory through its 32-bit data bus and 32-bit address. The 80386SX, more like the 80286, addresses 16M bytes of memory with its 24-bit address bus via its 16-bit data bus. The 80386SX was developed after the 80386DX for applications that didn't require the full 32-bit bus version. The 80386SX is found in many personal computers that use the same basic motherboard design as the 80286. At the time that the 80386SX was popular, most applications, including Windows, required fewer than 16M bytes of memory, so the 80386SX is a popular and a less costly version of the 80386 microprocessor. Even though the 80486 has become a less expensive upgrade path for newer systems, the 80386 still can be used for many applications. For example, the 80386EX does not appear in computer systems, but it is becoming very popular in embedded applications.

As with earlier versions of the Intel family of microprocessors, the 80386 requires a single +5.0 V power supply for operation. The power supply current averages 550 mA for the 25 MHz version of the 80386, 500 mA for the 20 MHz version, and 450 mA for the 16 MHz version. Also available is a 33 MHz version that requires 600 mA of power supply current. The power supply current for the 80386EX is 320 mA when operated at 33 MHz. Note that during some modes of normal operation, power supply current can surge to over 1.0 A. This means that the power supply and power distribution network must be capable of supplying these current surges. This device contains multiple Vcc and Vss connections that must all be connected to +5.0 V and grounded for proper operation. Some of the pins are labeled N/C (no connection) and must not be connected. Additional versions of the 80386SX and 80386EX are available with a +3.3 V power supply. They are often found in portable notebook or laptop computers and are usually packaged in a surface mount device.

Each 80386 output pin is capable of providing 4.0 mA (address and data connections) or 5.0 mA (other connections). This represents an increase in drive current compared to the 2.0 mA available on earlier 8086, 8088, and 80286 output pins. The output current available on most 80386EX output pins is 8.0 mA. Each input pin represents a small load, requiring only ±10 μA of current. In some systems, except the smallest, these current levels require bus buffers.

The function of each 80386DX group of pins follows:

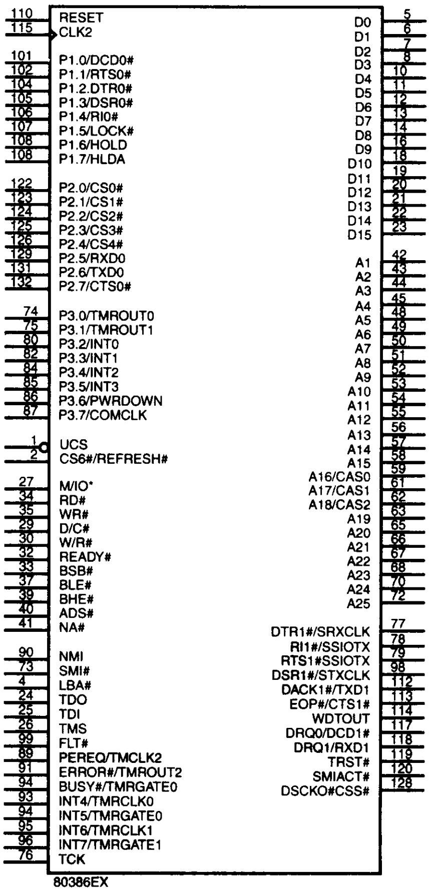


FIGURE 15–2 The 80386EX embedded PC.

A31–A2

Address bus connections address any of the 1G ∞ 32 memory locations found in the 80386 memory system. Note that A0 and A1 are encoded in the bus enable (BE3–BE0) to select any or all of the four bytes in a 32-bit wide memory location. Also note that because the 80386SX contains a 16-bit data bus in place of the 32-bit data bus found on the 80386DX, A1 is present on the 80386SX, and the bank selection signals are replaced with BHE and BLE. The BHE signal enables the upper data bus half; the BLE signal enables the lower data bus half.

D31–D0

Data bus connections transfer data between the microprocessor and its memory and I/O system. Note that the 80386SX contains D15–D0.

BE3–BE0

Bank enable signals select the access of a byte, word, or doubleword of data. These signals are generated internally by the microprocessor from address bits A1 and A0. On the 80386SX, these pins are replaced by BHE, BLE, and A1.

M/IO

Memory/IO selects a memory device when a logic 1 or an I/O device when a logic 0. During the I/O operation, the address bus contains a 16-bit I/O address on address connections A15–A2.

W/R

Write/read indicates that the current bus cycle is a write when a logic 1 or a read when a logic 0.

ADS

The **address data strobe** becomes active whenever the 80386 has issued a valid memory or I/O address. This signal is combined with the W/R signal to generate the separate read and write signals present in the earlier 8086–80286 microprocessor-based systems.

RESET **Reset** initializes the 80386, causing it to begin executing software at memory location FFFFFFF0H. The 80386 is reset to the real mode, and the leftmost 12 address connections remain logic 1s (FFFH) until a far jump or far call is executed. This allows compatibility with earlier microprocessors.

CLK2 **Clock times 2** is driven by a clock signal that is twice the operating frequency of the 80386. For example, to operate the 80386 at 16 MHz, we apply a 32 MHz clock to this pin.

READY **Ready** controls the number of wait states inserted into the timing to lengthen memory accesses.

LOCK **Lock** becomes a logic 0 whenever an instruction is prefixed with the LOCK: prefix. This is used most often during DMA accesses.

D/C **Data/control** indicates that the data bus contains data for or from memory or I/O when a logic 1. If D/C is a logic 0, the microprocessor is halted or executes an interrupt acknowledge.

BS16 **Bus size 16** selects either a 32-bit data bus (BS16 = 1) or a 16-bit data bus (BS16 = 0). In most cases, if an 80386DX is operated on a 16-bit data bus, we use the 80386SX that has a 16-bit data bus. On the 80386EX, the BS8 pin selects an 8-bit data bus.

NA **Next address** causes the 80386 to output the address of the next instruction or data in the current bus cycle.This pin is often used for pipelining the address.

HOLD **Hold** requests a DMA action.

HLDA **Hold acknowledge** indicates that the 80386 is currently in a hold condition.

PEREQ The **coprocessor request** asks the 80386 to relinquish control and is a direct connection to the 80387 arithmetic coprocessor.

BUSY **Busy** is an input used by the WAIT or FWAIT instruction that waits for the coprocessor to become not busy. This is also a direct connection to the 80387 from the 80386.

ERROR **Error** indicates to the microprocessor that an error is detected by the coprocessor.

INTR An **interrupt** request is used by external circuitry to request an interrupt.

NMI A **non-maskable interrupt** requests a non-maskable interrupt as it did on the earlier versions of the microprocessor.

The Memory System

The physical memory system of the 80386DX is 4G bytes in size and is addressed as such. If virtual addressing is used, 64T bytes are mapped into the 4G bytes of physical space by the memory management unit and descriptors. (Note that virtual addressing allows a program to be larger than 4G bytes if a method of swapping with a very large hard disk drive exists.) Figure 15–3 shows the organization of the 80386DX physical memory system.

The memory is divided into four 8-bit wide memory banks, each containing up to 1G bytes of memory. This 32-bit wide memory organization allows bytes, words, or doublewords of memory data to accessed directly. The 80386DX transfers up to a 32-bit wide number in a single memory cycle, whereas the early 8088 requires four cycles to accomplish the same transfer, and the 80286 and 80386SX require two cycles. Today, the data width is important, especially with single-precision floating-point numbers that are 32 bits wide. High-level software normally uses floating-point numbers for data storage, so 32-bit memory locations speed the execution of high-level software when it is written to take advantage of this wider memory.

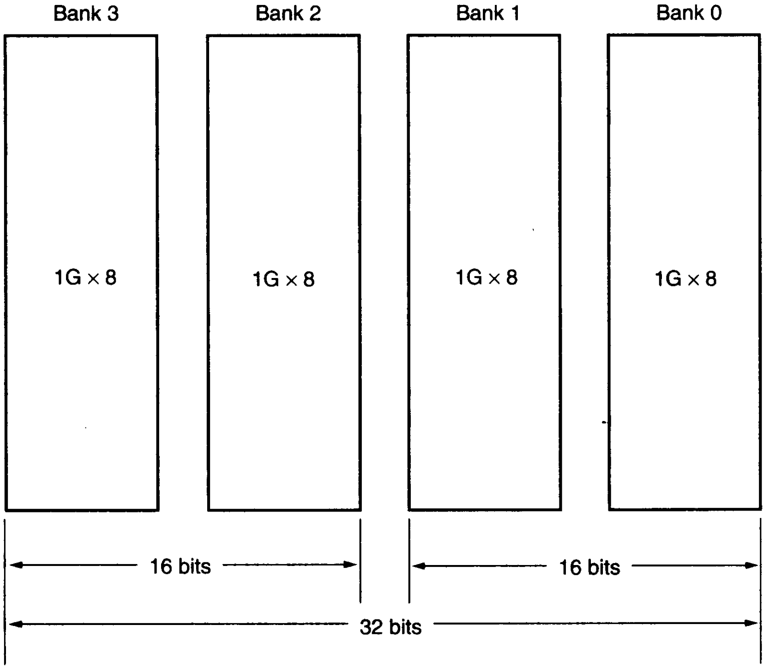


FIGURE 15–3 The memory system for the 80386 microprocessor. Notice that the memory is organized as four banks, each containing 1G byte. Memory is accessed as 8-, 16-, or 32-bit data.

Each memory byte is numbered in hexadecimal as they were in prior versions of the family. The difference is that the 80386DX uses a 32-bit wide memory address, with memory bytes numbered from location 00000000H–FFFFFFFFH.

The two memory banks in the 8086, 80286, and 80386SX system are accessed via BLE (A0 on the 8086 and 80286) and BHE. In the 80386DX, the memory banks are accessed via four bank enable signals BE3–BE0. This arrangement allows a single byte to be accessed when one bank enable signal is activated by the microprocessor. It also allows a word to be addressed when two bank enable signals are activated. In most cases, a word is addressed in bank 0 and 1, or in bank 2 and 3. Memory location 00000000H is in bank 0, location 00000001H is in bank 1, location 00000002H is in bank 2, and location 00000003H is in bank 3. The 80386DX does not contain address connections A0 and A1 because these have been encoded as the bank enable signals. Likewise, the 80386SX does not contain the A0 address pin because it is encoded in the BLE and BHE signals. The 80386EX addresses data either in two banks for a 16-bit wide memory system if BS8 = 1 or as an 8-bit system if BS8 = 0.

Buffered System. Figure 15–4 shows the 80386DX connected to buffers that increase fan-out from its address, data, and control connections. This microprocessor is operated at 25 MHz using a 50 MHz clock input signal that is generated by an integrated oscillator module. Oscillator modules are usually used to provide a clock in modern microprocessor-based equipment. The HLDA signal is used to enable all buffers in a system that uses direct memory access. Otherwise, the buffer enable pins are connected to ground in a non-DMA system.

Pipelines and Caches. The cache memory is a buffer that allows the 80386 to function more efficiently with lower DRAM speeds. A **pipeline** is a special way of handling memory accesses so the memory has additional time to access data. A 16 MHz 80386 allows memory devices with access times of 50 ns or less to operate at full speed. Obviously, there are few DRAMs currently available with these access times. In fact, the fastest DRAMs currently

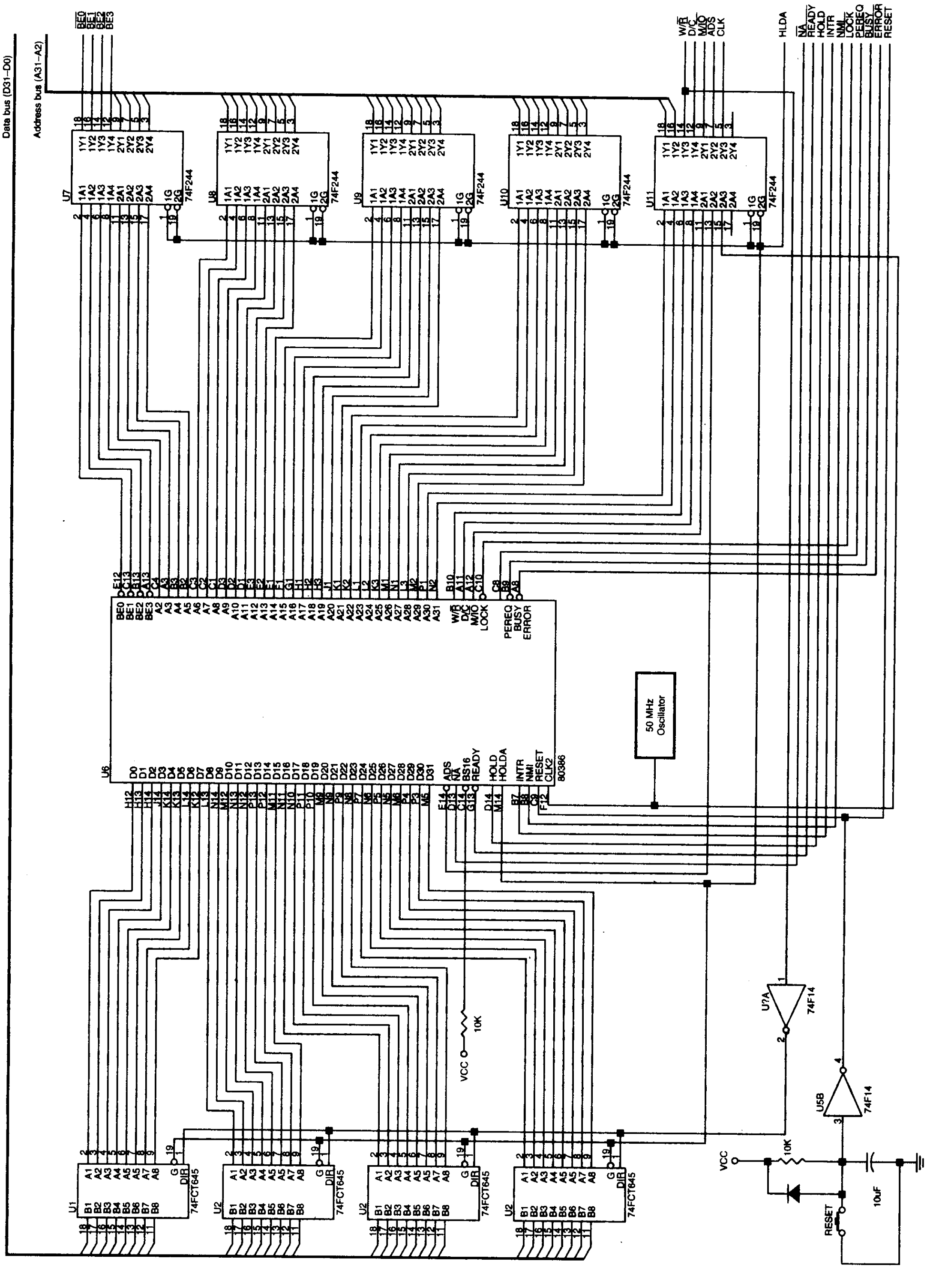


FIGURE 15–4 A fully buffered 25 MHz 80386DX.

in use have an access time of 60 ns or longer. This means that some technique must be found to interface these memory devices, which are slower than required by the microprocessor. Three techniques are available: interleaved memory, caching, and a pipeline.

The pipeline is the preferred means of interfacing memory because the 80386 microprocessor supports pipelined memory accesses. Pipelining allows memory an extra clocking period to access data. The extra clock extends the access time from 50 ns to 81 ns on an 80386 operating with a 16 MHz clock. The **pipe,** as it is often called, is set up by the microprocessor. When an instruction is fetched from memory, the microprocessor often has extra time before the next instruction is fetched. During this extra time, the address of the next instruction is sent out from the address bus ahead of time. This extra time (one clock period) is used to allow additional access time to slower memory components.

Not all memory references can take advantage of the pipe, which means that some memory cycles are not pipelined. These non-pipelined memory cycles request one wait state if the normal pipeline cycle requires no wait states. Overall, a pipe is a cost-saving feature that reduces the access time required by the memory system in low-speed systems.

Not all systems can take advantage of the pipe. Those systems typically operate at 20, 25, or 33 MHz. In these higher-speed systems, another technique must be used to increase the memory system speed. The **cache** memory system improves overall performance of the memory systems for data that are accessed more than once. Note that the 80486 contains an internal cache called a **level one cache** and the 80386 can only contain an external cache called a **level two cache.**

A **cache** is a high-speed memory system that is placed between the microprocessor and the DRAM memory system. Cache memory devices are usually static RAM memory components with access times of less than 25 ns. In many cases, we see level 2 cache memory systems with sizes between 32K and 1M byte. The size of the cache memory is determined more by the application than by the microprocessor. If a program is small and refers to little memory data, a small cache is beneficial. If a program is large and references large blocks of memory, the largest cache size possible is recommended. In many cases, a 64K-byte cache improves speed sufficiently, but the maximum benefit is often derived from a 256K-byte cache. It has been found that increasing the cache size much beyond 256K provides little benefit to the operating speed of the system that contains an 80386 microprocessor.

Interleaved Memory Systems. An **interleaved memory system** is another method of improving the speed of a system. Its only disadvantage is that it costs considerably more memory because of its structure. Interleaved memory systems are present in some systems, so memory access times can be lengthened without the need for wait states. In some systems, an interleaved memory may still require wait states, but may reduce their number. An interleaved memory system requires two or more complete sets of address buses and a controller that provides addresses for each bus. Systems that employ two complete buses are called a **two-way interleave;** systems that use four complete buses are called a **four-way interleave.**

An interleaved memory is divided into two or four parts. For example, if an interleaved memory system is developed for the 80386SX microprocessor, one part contains the 16-bit addresses 000000H–000001H, 000004H–000005H, etc.; the other part contains addresses 000002– 000003, 000006H–000007H, etc. While the microprocessor accesses locations 000000H–000001H, the interleave control logic generates the address strobe signal for locations 000002H–000003H. This selects and accesses the word at location 000002H–000003H, while the microprocessor processes the word at location 000000H–000001H. This process alternates memory sections, thus increasing the performance of the memory system.

Interleaving lengthens the amount of access time provided to the memory because the address is generated to select the memory before the microprocessor accesses it. This is because the microprocessor pipelines memory addresses, sending the next address out before the data are read from the last address.

The problem with interleaving, although not major, is that the memory addresses must be accessed so that each section is alternately addressed. This does not always happen as a program executes. Under normal program execution, the microprocessor alternately addresses memory approximately 93 percent of the time. For the remaining 7 percent, the microprocessor addresses data in the same memory section, which means that in these 7 percent of the memory accesses, the memory system must cause wait states because of the reduced access time.

The access time is reduced because the memory must wait until the previous data are transferred before it can obtain its address. This leaves the memory with less access time; therefore, a wait state is required for accesses in the same memory bank.

See Figure 15–5 for the timing diagram of the address as it appears at the microprocessor address pins. This timing diagram shows how the next address is output before the current data are accessed. It also shows how access time is increased by using interleaved memory addresses for each section of memory compared to a non-interleaved access, which requires a wait state.

Figure 15–6 pictures the interleave controller. Admittedly, this is a complex logic circuit, which needs some explanation. First, if the SEL input (used to select this section of the memory) is inactive (logic 0), then the WAIT signal is a logic 1. Also, both ALE0 and ALE1, used to strobe the address to the memory sections, are both logic 1s, causing the latches connected to them to become transparent.

As soon as the SEL input becomes a logic 1, this circuit begins to function. The A1 input is used to determine which latch (U2B or U5A) becomes a logic 0, selecting a section of the memory. Also the ALE pin that becomes a logic 0 is compared with the previous state of the ALE pins. If the same section of memory is accessed a second time, the WAIT signal becomes a logic 0, requesting a wait state.

Figure 15–7 illustrates an interleaved memory system that uses the circuit of Figure 15–6. Notice how the ALE0 and ALE1 signals are used to capture the address for either section of memory. The memory in each bank is 16-bits wide. If accesses to memory require 8-bit data, the system causes wait states, in most cases. As a program executes, the 80386SX fetches instruction 16-bits at a time from normally sequential memory locations. Program execution uses interleaving in most cases. If a system is going to access mostly 8-bit data, it is doubtful that memory interleaving will reduce the number of wait states.

The access time allowed by an interleaved system, such as the one shown in Figure 15–7, is increased to 112 ns from 69 ns by using a 16 MHz system clock. (If a wait state is inserted, access time with a 16 MHz clock is 136 ns, which means that an interleaved system performs at about the same rate as a system with one wait state.) If the clock is increased to 20 MHz, the interleaved memory requires 89.6 ns, where standard, non-interleaved memory interfaces allow 48 ns for memory access. At this higher clock rate, 80 ns DRAMs function properly, without wait states when the memory addresses are interleaved. If an access to the same section occurs, then a wait state is inserted.

The Input/Output System

The 80386 input/output system is the same as that found in any Intel 8086 family microprocessor-based system. There are 64K different bytes of I/O space available if isolated I/O is implemented. With isolated I/O, the IN and OUT instructions are used to transfer I/O data between the microprocessor and I/O devices. The I/O port address appears on address bus connections A15–A2, with BE3–BE0 used to select a byte, word, or doubleword of I/O data. If memory-mapped I/O is implemented, then the number of I/O locations can be any amount up to 4G bytes. With memory-mapped I/O, any instruction that transfers data between the microprocessor and memory system can be used for I/O transfers because the I/O device is treated as a memory device. Almost all 80386 systems use isolated I/O because of the I/O protection scheme provided by the 80386 in protected mode operation.

Figure 15–8 shows the I/O map for the 80386 microprocessor. Unlike the I/O map of earlier Intel microprocessors, which were 16-bits wide, the 80386 uses a full 32-bit wide I/O system divided into four banks. This is identical to the memory system, which is also divided into four banks. Most I/O transfers are 8-bits wide because we often use ASCII code (a 7-bit code) for transferring alphanumeric data between the microprocessor and printers and keyboards. This may change if Unicode, a 16-bit alphanumeric code, becomes common and replaces ASCII code. Recently, I/O devices that are 16- and even 32-bits wide have appeared for systems such as disk memory and video display interfaces. These wider I/O paths increase the data transfer rate between the microprocessor and the I/O device when compared to 8-bit transfers.

The I/O locations are numbered from 0000H–FFFFH. A portion of the I/O map is designated for the 80387 arithmetic coprocessor. Although the port numbers for the coprocessor are well above the normal I/O map, it is important that they be taken into account when decoding I/O space (overlaps). The coprocessor uses I/O location

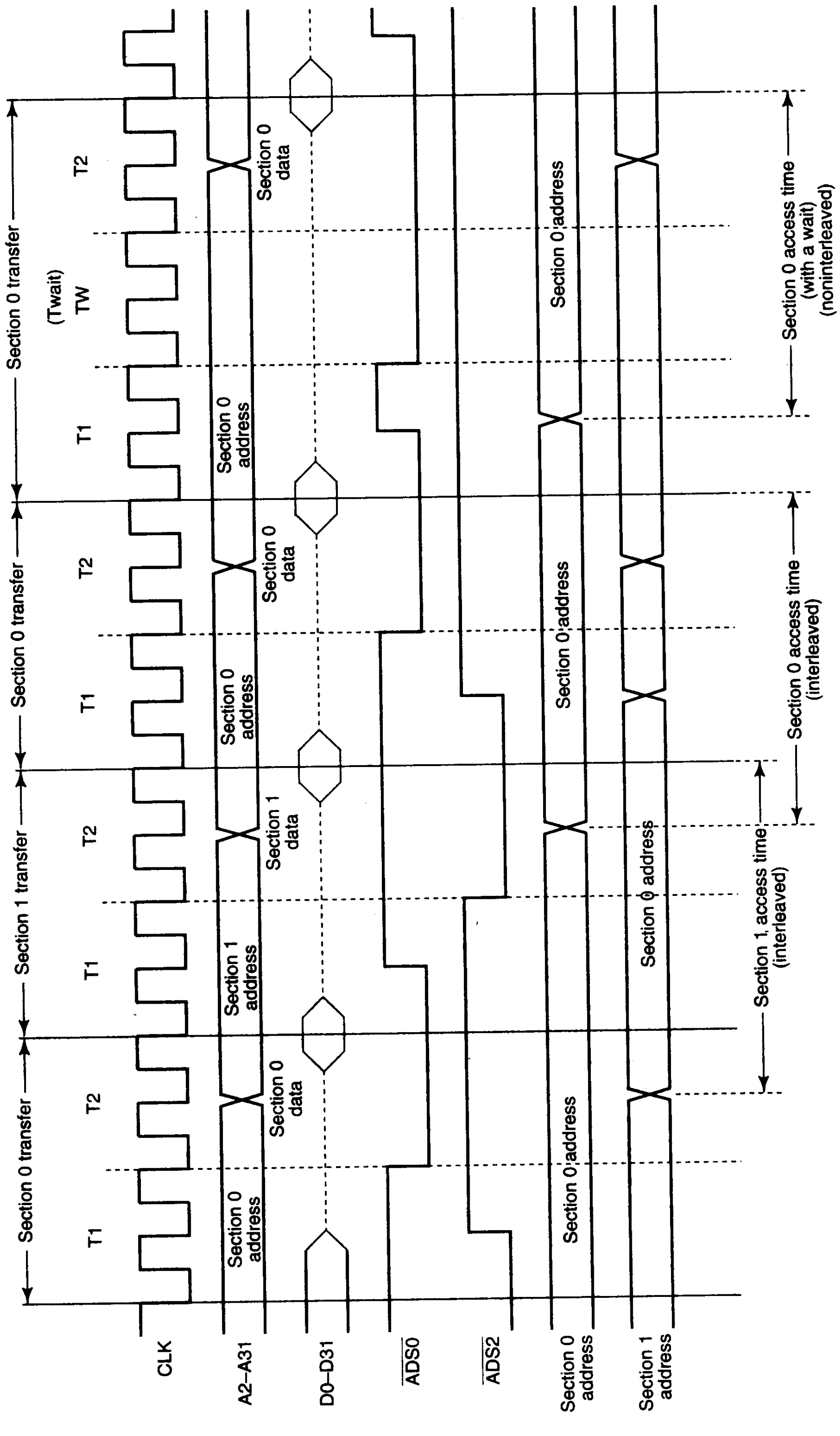


FIGURE 15–5 The timing diagram of an interleaved memory system showing the access times and address signals for both sections of memory.

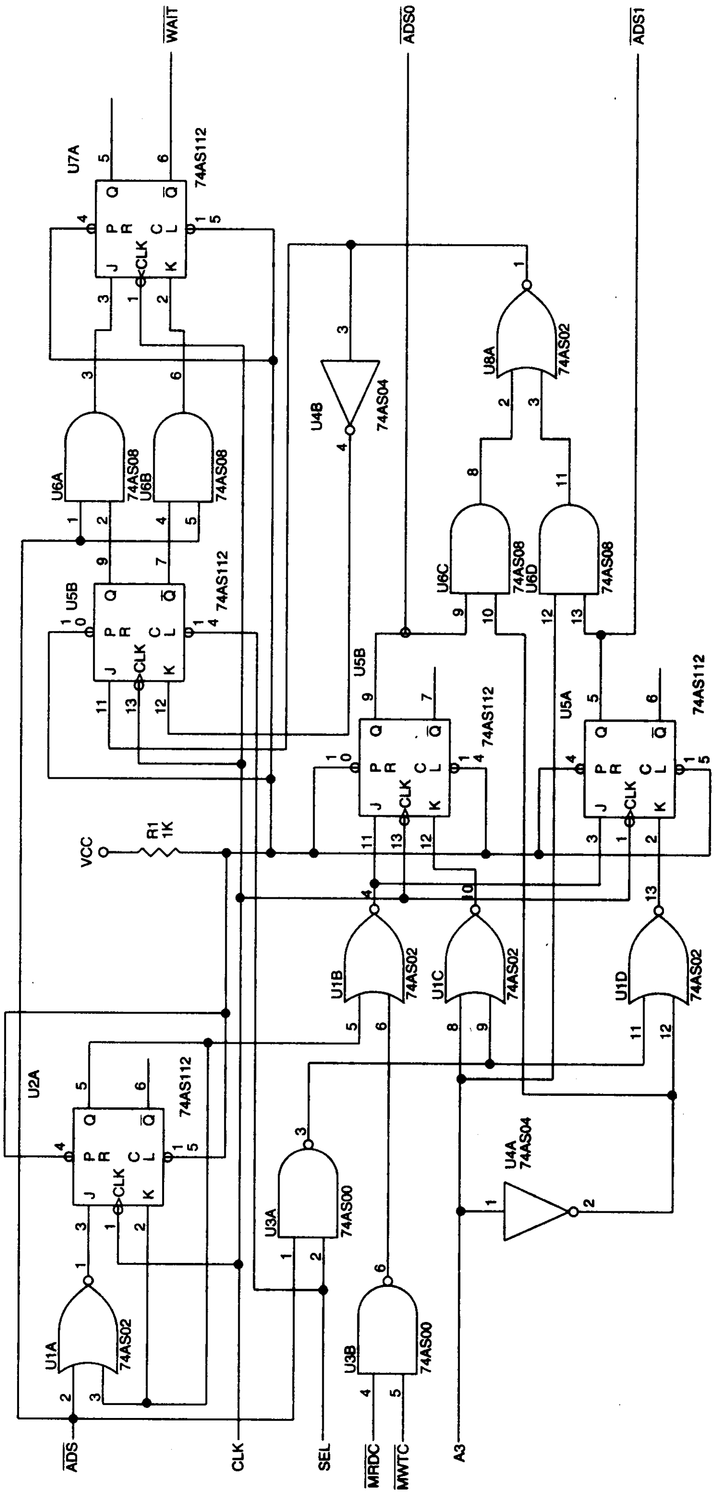


FIGURE 15–6 The interleaved control logic, which generates separate ADS signals and a WAIT signal used to control interleaved memory.

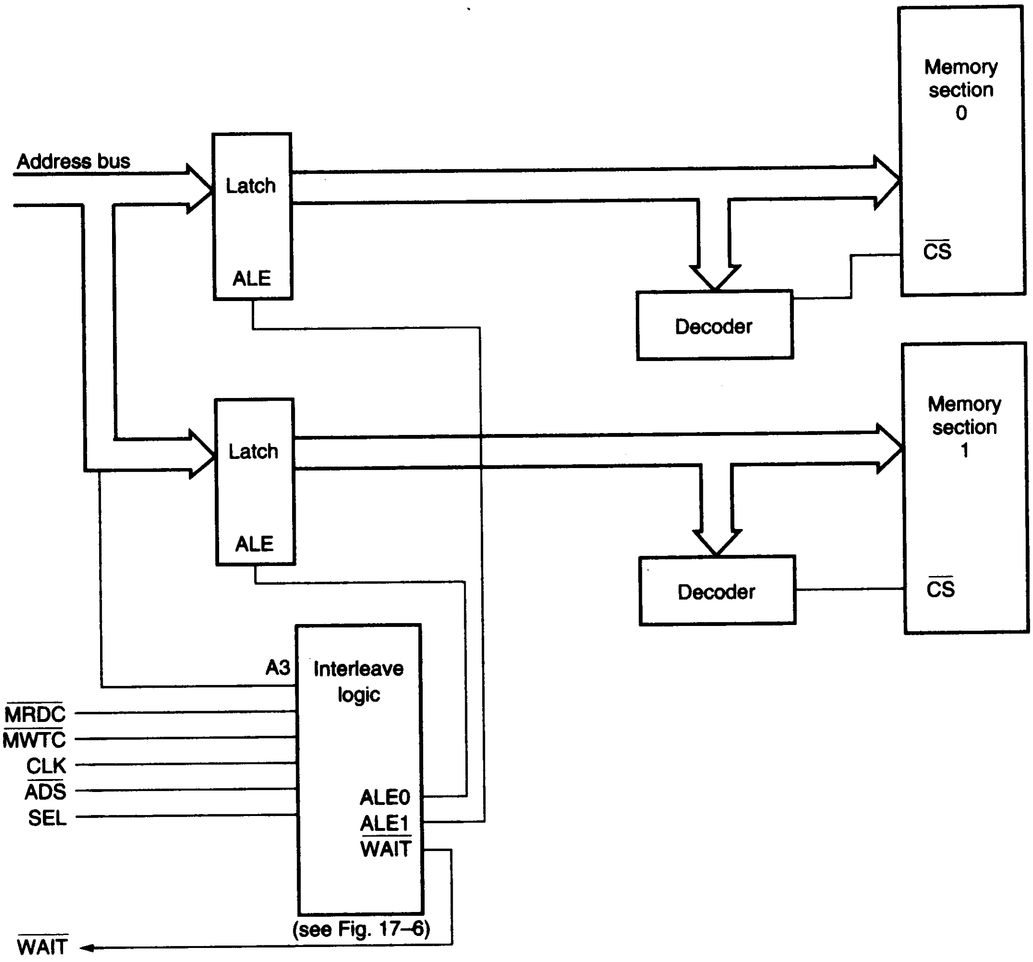


FIGURE 15–7 An interleaved memory system showing the address latches and the interleaved logic circuit.

800000F8H–800000FFH for communications between the 80387 and 80386. The 80287 numeric coprocessor designed to use with the 80286 uses the I/O addresses 00F8H–00FFH for coprocessor communications. Because we often decode only address connections A15–A2 to select an I/O device, be aware that the coprocessor will activate devices 00F8H–00FFH unless address line A31 is also decoded. This should present no problem because you really should not be using I/O ports 00F8H–00FFH for any purpose.

The only new feature that was added to the 80386 with respect to I/O is the I/O privilege information added to the tail end of the TSS when the 80386 is operated in protected mode. As described in the section on memory management, an I/O location can be blocked or inhibited in the protected mode. If the blocked I/O location is addressed, an interrupt (type 13, general fault) is generated. This scheme is added so that I/O access can be prohibited in a multiuser environment. Blocking is an extension of the protected mode operation, as are privilege levels.

Memory and I/O Control Signals

The memory and I/O are controlled with separate signals. The M/IO signal indicates whether the data transfer is between the microprocessor and the memory (*M/IO* = 1) or I/O (*M/IO* = 0). In addition to M/IO, the memory and

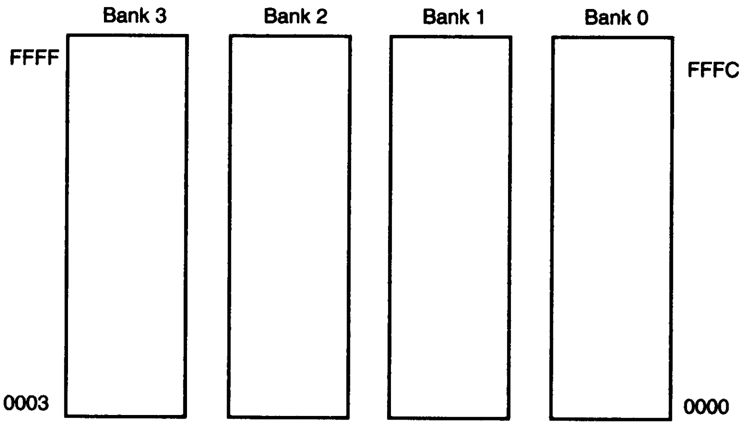


FIGURE 15–8 The isolated I/O map for the 80386 microprocessor. Here four banks of 8-bits each are used to address 64K different I/O locations. I/O is numbered from location 0000H to FFFFH.

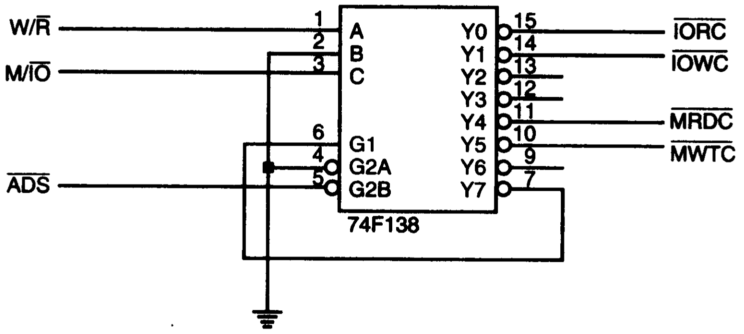


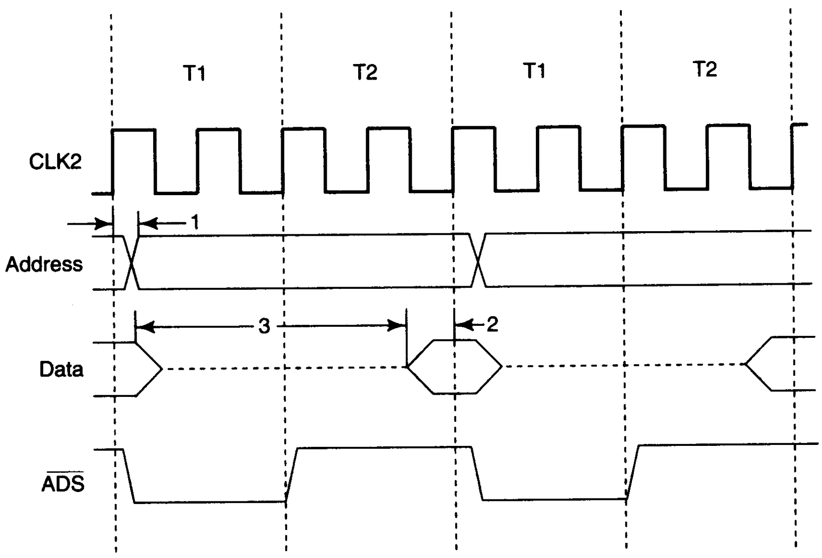
FIGURE 15–9 Generation of memory and I/O control signals for the 80386, 80486, and Pentium.

I/O systems must read or write data. The W/R signal is a logic 0 for a read operation, and a logic 1 for a write operation. The ADS signal is used to qualify the M/IO and W/R control signals. This is a slight deviation from earlier Intel microprocessors, which didn't use ADS for qualification.

See Figure 15–9 for a simple circuit that generates four control signals for the memory and I/O devices in the system. Notice that two control signals are developed for memory control (MRDC and MWTC) and two for I/O control (IORC and IOWC). These signals are consistent with the memory and I/O control signals generated for use in earlier versions of the Intel microprocessor.

Timing

Timing is important for understanding how to interface memory and I/O to the 80386 microprocessor. Figure 15–10 shows the timing diagram of a non-pipelined memory read cycle. Note that the timing is referenced to the CLK2 input signal and that a bus cycle consists of four clocking periods.



	33 MHz	25 MHz	20 MHz	16 MHz
Time 1:	4–15 ns	4–21 ns	4–30 ns	4–36 ns
Time 2:	5 ns	7 ns	11 ns	11 ns
Time 3:	46 ns	52 ns	59 ns	78 ns

FIGURE 15–10 The non-pipelined read timing for the 80386 microprocessor.

Each bus cycle contains two clocking states with each state (T1 and T2) containing two clocking periods. Note in Figure 15–10 that the access time is listed as time number 3. The 16 MHz version allows memory an access time of 78 ns before wait states are inserted in this non-pipelined mode of operation. To select the non-pipelined mode, we place a logic 1 on the NA pin.

Figure 15–11 illustrates the read timing when the 80386 is operated in the pipelined mode. Notice that additional time is allowed to the memory for accessing data because the address is sent out early. Pipelined mode is selected by placing a logic 0 on the NA pin and by using address latches to capture the pipelined address. The clock pulse that is applied to the address latches comes from the ADS signal. Address latches must be used with a pipelined system, as well as with interleaved memory banks. The minimum number of interleaved banks of two and four have been successfully used in some applications.

Notice that the pipelined address appears one complete clocking state before it normally appears with non-pipelined addressing. In the 16 MHz version of the 80386, this allows an additional 62.5 ns for memory access. In a non-pipelined system, a memory access time of 78 ns is allowed to the memory system; in a pipelined system, 140.5 ns is allowed. The advantages of the pipelined system are that no wait states are required (in many, but not all bus cycles) and much lower-speed memory devices may be connected to the microprocessor. The disadvantage is that we need to interleave memory to use a pipe, which requires additional circuitry and occasional wait states.

Wait States

Wait states are needed if memory access times are long compared with the time allowed by the 80386 for memory access. In a non-pipelined 33 MHz system, memory access time is only 46 ns. Currently, no DRAM memory exists that has an access time of 46 ns. This means that wait states must be introduced to access the DRAM (one wait for 60 ns DRAM) or an EPROM that has an access time of 100 ns (two waits). Note that this wait state is built into a motherboard and cannot be removed.

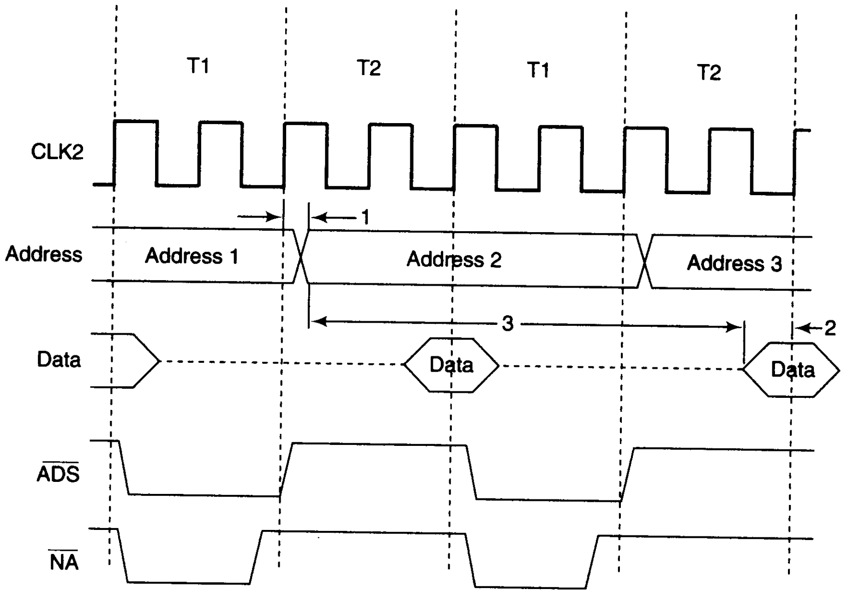


FIGURE 15–11 The pipelined read timing for the 80386 microprocessor.

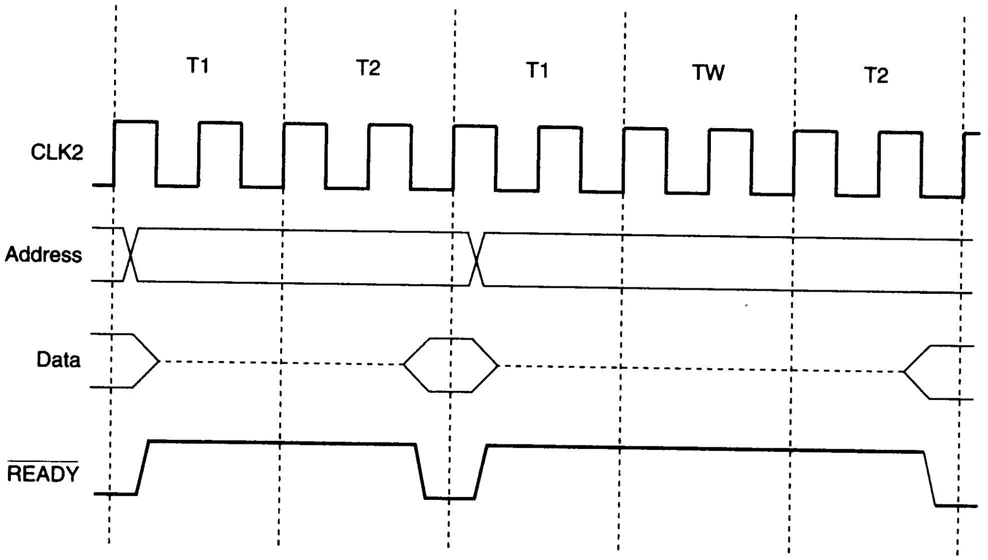
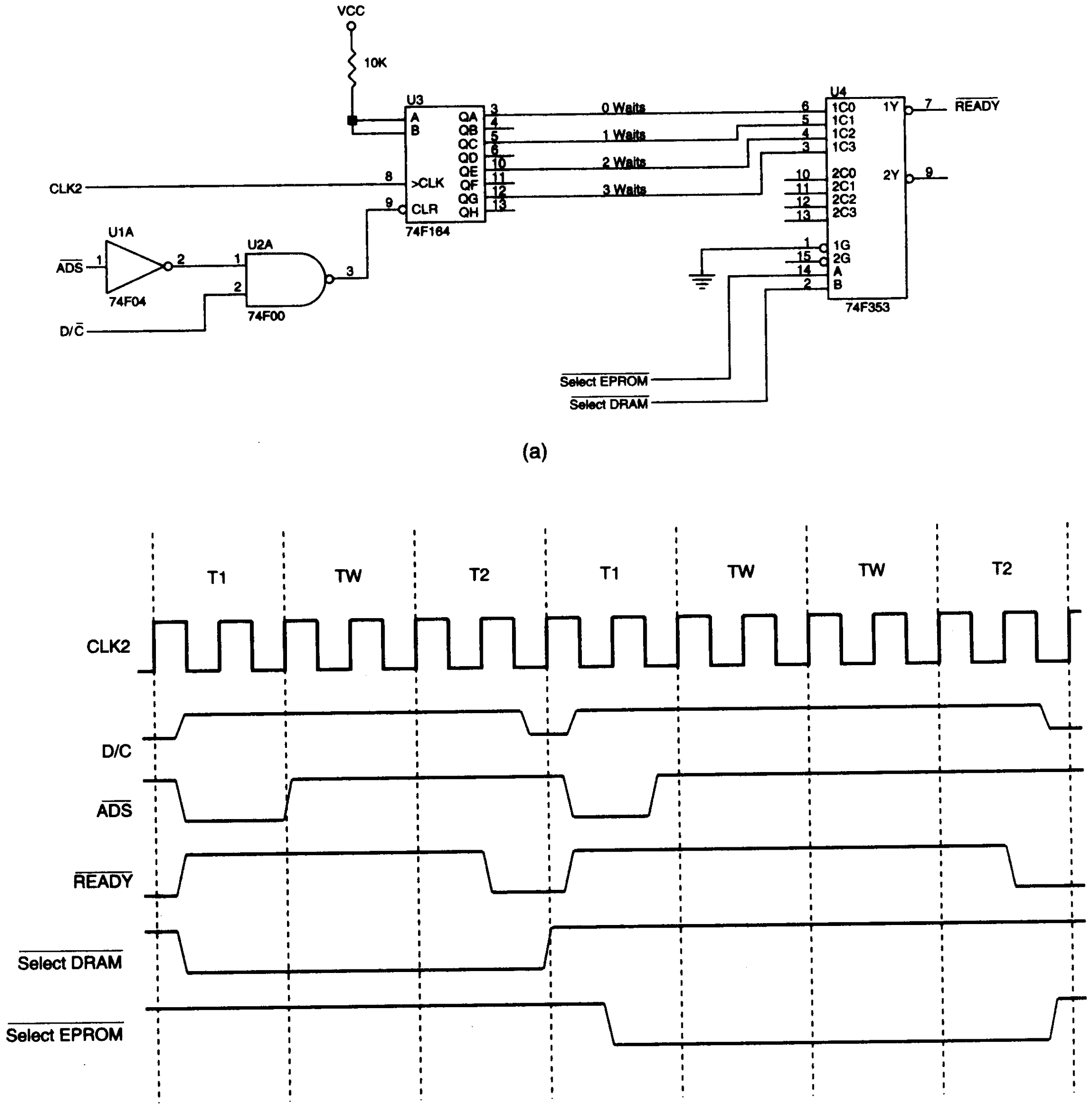


FIGURE 15–12 A non-pipelined 80386 with 0 and 1 wait states.

The READY input controls whether or not wait states are inserted into the timing. The READY input on the 80386 is a dynamic input that must be activated during each bus cycle. Figure 15–12 shows a few bus cycles with one normal (no wait) cycle and one that contains a single wait state. Notice how the READY is controlled to cause 0 or 1 wait.



(a)

(b)

FIGURE 15–13 (a) Circuit and (b) timing that selects 1 wait state for DRAM and 2 waits for EPROM.

The READY signal is sampled at the end of a bus cycle to determine whether the clock cycle is T2 or TW. If READY = 0 at this time, it is the end of the bus cycle or T2. If READY is 1 at the end of a clock cycle, the cycle is a TW and the microprocessor continues to test READY, searching for a logic 0 and the end of the bus cycle.

In the non-pipelined system, whenever ADS becomes a logic 0, a wait state is inserted if READY = 1. After ADS returns to a logic 1, the positive edges of the clock are counted to generate the READY signal. The READY signal becomes a logic 0 after the first clock to insert 0 wait states. If 1 wait state is inserted, the READY line must remain a logic 1 until at least two clocks have elapsed. If additional wait states are desired, then additional time must elapse before READY is cleared. This essentially allows any number of wait states to be inserted into the timing.

Figure 15–13 shows a circuit that inserts 0 through 3 wait states for various memory addresses. In the example, 1 wait state is produced for a DRAM access and 2 wait states for an EPROM access. The 74F164 clears whenever ADS is low and D/C is high. It begins to shift after ADS returns to a logic 1 level. As it shifts, the 00000000 in the shift register begins to fill with logic 1s from the QA connection toward the QH connection. The four different outputs are connected to an inverting multiplexer that generates the active low READY signal.

15–2 SPECIAL 80386 REGISTERS

A new series of registers, not found in earlier Intel microprocessors, appears in the 80386 as control, debug and test registers. Control registers CR0–CR3 control various features, DR0–DR7 facilitate debugging, and registers TR6 and TR7 are used to test paging and caching.

Control Registers

In addition to the EFLAGS and EIP as described earlier, there are other control registers found in the 80386. Control register 0 (CR0) is identical to the MSW (machine status word) found in the 80286 microprocessor, except that it is 32 bits wide instead of 16 bits wide. Additional control registers are CR1, CR2, and CR3.

Figure 15–14 illustrates the control register structure of the 80386. Control register CR1 is not used in the 80386, but is reserved for future products. Control register CR2 holds the linear page address of the last page accessed before a page fault interrupt. Finally, control register CR3 holds the base address of the page directory. The rightmost 12 bits of the 32-bit page table address contain zeros and combine with the remainder of the register to locate the start of the 4K-long page table.

Register CR0 contains a number of special control bits that are defined as follows in the 80386:

PG Selects page table translation of linear addresses into physical addresses when *PG* = 1. Page table translation allows any linear address to be assigned any physical memory location.

ET Selects the 80287 coprocessor when *ET* = 0 or the 80387 coprocessor when *ET* = 1. This bit was installed because there was no 80387 available when the 80386 first appeared. In most systems, ET is set to indicate that an 80387 is present in the system.

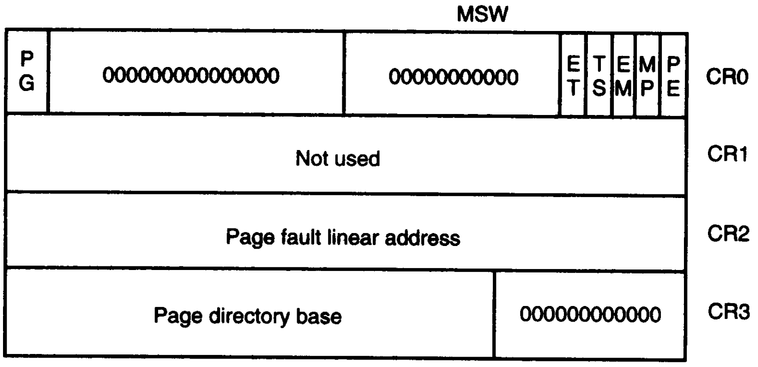


FIGURE 15–14 The control register structure of the 80386 microprocessor.

TS Indicates that the 80386 has switched tasks (in protected mode, changing the contents of TR places a 1 into TS). If *TS* = 1, a numeric coprocessor instruction causes a type 7 (coprocessor not available) interrupt.

EM Is set to cause a type 7 interrupt for each ESC instruction. (ESCape instructions are used to encode instructions for the 80387 coprocessor.) We often use this interrupt to emulate, with software, the function of the coprocessor. Emulation reduces the system cost, but it often requires at least 100 times longer to execute the emulated coprocessor instructions.

MP Is set to indicate that the arithmetic coprocessor is present in the system.

PE Is set to select the protected mode of operation for the 80386. It may also be cleared to re-enter the real mode. This bit can only be set in the 80286. The 80286 could not return to real mode without a hardware reset, which precludes its use in most systems that use protected mode.

Debug and Test Registers

Figure 15–15 shows the sets of debug and test registers. The first four debug registers contain 32-bit linear breakpoint addresses. (A **linear address** is a 32-bit address generated by a microprocessor instruction that may or may not be the same as the physical address.) The breakpoint addresses, which may locate an instruction or datum, are constantly compared with the addresses generated by the program. If a match occurs, the 80386 will cause a type 1 interrupt (TRAP or debug interrupt) to occur, if directed by debug registers DR6 and DR7. This feature is a much-expanded version of the basic trapping or tracing allowed with the earlier Intel mi-

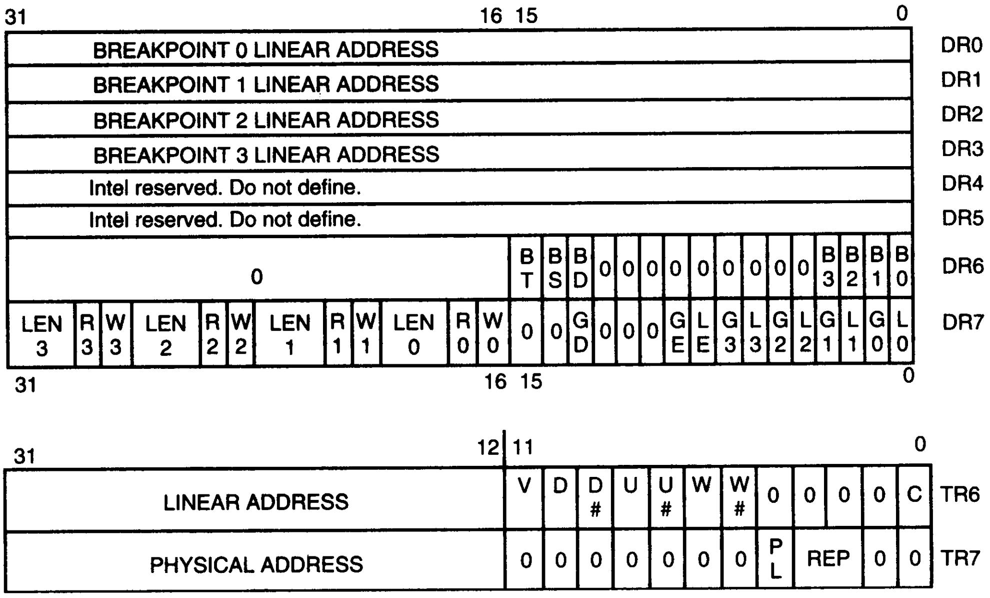

31 16 15 0
BREAKPOINT 0 LINEAR ADDRESS DR0
BREAKPOINT 1 LINEAR ADDRESS DR1
BREAKPOINT 2 LINEAR ADDRESS DR2
BREAKPOINT 3 LINEAR ADDRESS DR3
Intel reserved. Do not define. DR4
Intel reserved. Do not define. DR5
0 BT BS BD 0 0 0 0 0 0 0 0 0 B3 B2 B1 B0 DR6
LEN 3 R3 W3 LEN 2 R2 W2 LEN 1 R1 W1 LEN 0 R0 W0 0 0 GD 0 0 0 GE LE G3 L3 G2 L2 G1 L1 G0 L0 DR7
31 16 15 0
31 12 11 0
LINEAR ADDRESS V D D# U U# W W# 0 0 0 0 C TR6
PHYSICAL ADDRESS 0 0 0 0 0 0 0 PL REP 0 0 TR7

FIGURE 15–15 The debug and test registers of the 80386. (Courtesy of Intel Corporation.)

croprocessors through the type 1 interrupt. The breakpoint addresses are very useful in debugging faulty software. The control bits in DR6 and DR7 are defined as follows:

BT If set (1), the debug interrupt was caused by a task switch.

BS If set, the debug interrupt was caused by the TF bit in the flag register.

BD If set, the debug interrupt was caused by an attempt to read the debug register with the GD bit set. The GD bit protects access to the debug registers.

B3–B0 Indicate which of the four debug breakpoint addresses caused the debug interrupt.

LEN Each of the four length fields pertains to each of the four breakpoint addresses stored in DR0–DR3. These bits further define the size of access at the breakpoint address as 00 (byte), 01 (word), or 11 (doubleword).

RW Each of the four read/write fields pertains to each of the four breakpoint addresses stored in DR0–DR3. The RW field selects the cause of action that enabled a breakpoint address as 00 (instruction access), 01 (data write), and 11 (data read and write).

GD If set, GD prevents any read or write of a debug register by generating the debug interrupt. This bit is automatically cleared during the debug interrupt so that the debug registers can be read or changed, if needed.

GE If set, selects a global breakpoint address for any of the four breakpoint address registers.

LE If set, selects a local breakpoint address for any of the four breakpoint address registers.

The test registers, TR6 and TR7, are used to test the **translation look-aside buffer** (TLB). The TLB is used with the paging unit within the 80386. The TLB holds the most commonly used page table address translations. The TLB reduces the number of memory reads required for looking up page translation addresses in the page translation tables. The TLB holds the most common 32 entries from the page table, and it is tested with the TR6 and TR7 test registers.

Test register TR6 holds the tag field (linear address) of the TLB, and TR7 holds the physical address of the TLB. To write a TLB entry, perform the following steps:

1. Write TR7 for the desired physical address, PL, and REP values.
2. Write TR6 with the linear address, making sure that $C = 0$.

To read a TLB entry:

1. Write TR6 with the linear address, making sure that $C = 1$.
2. Read both TR6 and TR7. If the PL bit indicates a hit, then the desired values of TR6 and TR7 indicate the contents of the TLB.

The bits found in TR6 and TR7 indicate the following conditions:

V Shows that the entry in the TLB is valid.

D Indicates that the entry in the TLB is invalid or dirty.

U A bit for the TLB.

W Indicates that the area addressed by the TLB entry is writable.

C Selects a write (0) or immediate lookup (1) for the TLB.

PL Indicates a hit if a logic 1.

REP Selects which block of the TLB is written.

Refer to the section on memory management and the paging unit for more detail on the function of the TLB.

15–3 80386 MEMORY MANAGEMENT

The **memory-management unit** (MMU) within the 80386 is similar to the MMU inside the 80286, except that the 80386 contains a paging unit not found in the 80286. The MMU performs the task of converting linear addresses, as they appear as outputs from a program, into physical addresses that access a physical memory location located anywhere within the memory system. The 80386 uses the paging mechanism to allocate any physical address to any logical address. Therefore, even though the program is accessing memory location A0000H with an instruction, the actual physical address could be memory location 100000H, or any other location if paging is enabled. This feature allows virtually any software, written to operate at any memory location, to function in an 80386 because any linear location can become any physical location. Earlier Intel microprocessors did not have this flexibility. Paging is used with DOS to relocate 80386 and 80486 memory at addresses above FFFFFH and into spaces between ROMs at locations D0000–DFFFFH and other areas as they are available. The area between ROMs is often referred to as *upper memory;* the area above FFFFFH is referred to as *extended memory.*

Descriptors and Selectors

Before the memory paging unit is discussed, we examine the descriptor and selector for the 80386 microprocessor. The 80386 uses descriptors in much the same fashion as the 80286. In both microprocessors, a **descriptor** is a series of eight bytes that describe and locate a memory segment. A **selector** (segment register) is used to index a descriptor from a table of descriptors. The main difference between the 80286 and 80386 is that the latter has two additional selectors (FS and GS) and the most-significant two bytes of the descriptor are defined for the 80386. Another difference is that 80386 descriptors use a 32-bit base address and a 20-bit limit, instead of the 24-bit base address and a 16-bit limit found on the 80286.

The 80286 addresses a 16M-byte memory space with its 24-bit base address and has a segment length limit of 64K bytes, due to the 16-bit limit. The 80386 addresses a 4G-byte memory space with its 32-bit base address and has a segment length limit of 1M bytes or 4G bytes, due to a 20-bit limit that is used in two different ways. The 20-bit limit can access a segment with a length of 1M byte if the granularity bit (G) = 0. If G = 1, the 20-bit limit allows a segment length of 4G bytes.

The granularity bit is found in the 80386 descriptor. If G = 0, the number stored in the limit is interpreted directly as a limit, allowing it to contain any limit between 00000H and FFFFFH for a segment size up to 1M byte. If G = 1, the number stored in the limit is interpreted as 00000XXXH–FFFFFXXXH, where the XXX is any value between 000H and FFFH. This allows the limit of the segment to range between 0 bytes to 4G bytes in steps of 4K bytes. A limit of 00001H indicates that the limit is 4K bytes when G = 1 and 1 byte when G = 0. An example is a segment that begins at physical address 10000000H. If the limit is 00001H and G = 0, this segment begins at 10000000H and ends at 10000001H. If G = 1 with the same limit (00001H), the segment begins at location 10000000H and ends at location 10001FFFH.

Figure 15–16 shows how the 80386 addresses a memory segment in the protected mode using a selector and a descriptor. Note that this is identical to the way that a segment is addressed by the 80286. The difference is the size of the segment accessed by the 80386. The selector usesits leftmost 13 bits to select a descriptor from a descriptor table. The TI bit indicates either the local (TI = 1) or global (TI = 0) descriptor table. The rightmost two bits of the selector define the requested privilege level of the access.

Because the selector uses a 13-bit code to access a descriptor, there are at most 8192 descriptors in each table—local or global. Because each segment (in an 80386) can be 4G bytes in length, we can access 16,384 segments at a time with the two descriptor tables. This allows the 80386 to access a virtual memory size of 64T bytes. Of course, only 4G bytes of memory actually exist in the memory system (1T byte = 1024G bytes). If a program requires more than 4G bytes of memory at a time, it can be swapped between the memory system and a disk drive or other form of large volume storage.

The 80386 uses descriptor tables for both global (GDT) and local (LDT) descriptors. A third descriptor table appears for interrupt (IDT) descriptors or gates. The first six bytes of the descriptor are the same as in the 80286,

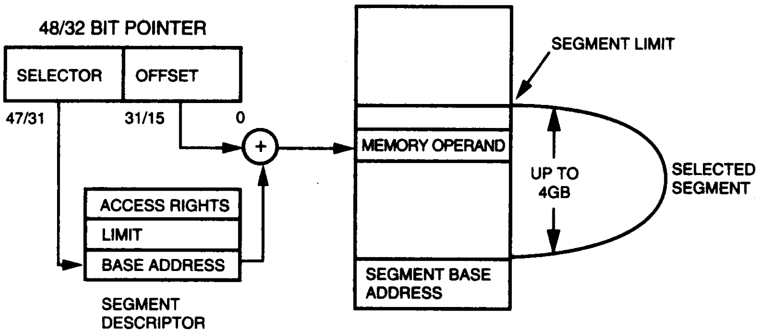


FIGURE 15–16 Protected mode addressing using a segment register as a selector. (Courtesy of Intel Corporation.)

80286 Descriptor

Reserved		6
Access rights	Base (B23–B16)	4
Base (B15–B0)		2
Limit (L15–L0)		0

80386 Descriptor

Base (B24–B31)	G	D	O	AVL	Limit (L16–L19)	6
Access rights	Base (B23–B16)					4
Base (B15–B0)						2
Limit (L15–L0)						0

FIGURE 15–17 The descriptors for the 80286 and 80386 microprocessors.

which allows 80286 software to be upward compatible with the 80386. (An 80286 descriptor used 00H for its most significant two bytes.) See Figure 15–17 for the 80286 and 80386 descriptor. The base address is 32 bits in the 80386, the limit is 20 bits, and a G bit selects the limit multiplier (1 or 4K times). The fields in the descriptor for the 80386 are defined as follows:

Base (B31–B0) Defines the starting 32-bit address of the segment within the 4G-byte physical address space of the 80386 microprocessor.

Limit (L19–L0) Defines the limit of the segment in units of bytes if the *G bit* = 0, or in units of 4K bytes if *G* = 1. This allows a segment to be of any length from 1 byte to 1M bytes if *G* = 0, and from 4K bytes to 4G bytes if *G* = 1. Recall that the limit indicates the last byte in a segment.

Access Rights Determines privilege level and other information about the segment. This byte varies with different types of descriptors and is elaborated with each descriptor type.

G The granularity bit selects a multiplier of 1 or 4K times for the limit field. If *G* = 0, the multiplier is 1; if *G* = 1, the multiplier is 4K.

D Selects the default register size. If *D* = 0, the registers are 16-bits wide, as in the 80286; if *D* = 1, they are 32-bits wide, as in the 80386. This bit determines whether prefixes are required for 32-bit data and index registers. If *D* = 0, then a prefix is required to access 32-bit registers and to use 32-bit pointers. If *D* = 1, then a prefix is required to access 16-bit registers and 16-bit pointers. The USE16 and USE32 directives appended to the

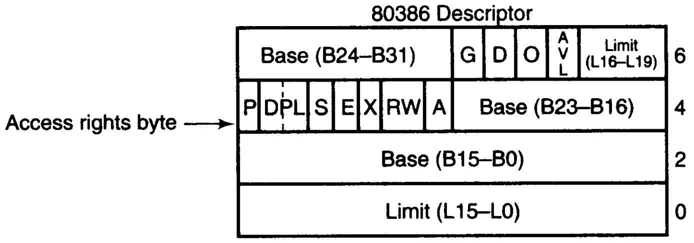


FIGURE 15–18 The format of the 80386 segment descriptor.

SEGMENT statement in assembly language control the setting of the D bit. In the real mode, it is always assumed that the registers are 16-bits wide, so any instruction that references a 32-bit register or pointer must be prefixed. The current version of DOS assumes $D = 0$.

AVL This bit is available to the operating system to use in any way that it sees fit. It often indicates that the segment described by the descriptor is available.

Descriptors appear in two forms in the 80386 microprocessor: the segment descriptor and the system descriptor. The segment descriptor defines data, stack, and code segments; the system descriptor defines information about the system's tables, tasks, and gates.

Segment Descriptors. Figure 15–18 shows the segment descriptor. This descriptor fits the general form, as dictated in Figure 15–17, but the access rights bits are defined to indicate how the data, stack, or code segment described by the descriptor functions. Bit position 4 of the access rights byte determines whether the descriptor is a data or code segment descriptor ($S = 1$) or a system segment descriptor ($S = 0$). Note that the labels used for these bits may vary in different versions of Intel literature, but they perform the same tasks.

Following is a description of the access rights bits and their function in the segment descriptor:

P **Present** is a logic 1 to indicate that the segment is present. If $P = 0$ and the segment is accessed through the descriptor, a type 11 interrupt occurs. This interrupt indicates that a segment was accessed that is not present in the system.

DPL **Descriptor privilege level** sets the privilege level of the descriptor, where 00 has the highest privilege and 11 has the lowest. This is used to protect access to segments. If a segment is accessed with a privilege level that is lower (higher in number) than the DPL, a privilege violation interrupt occurs. Privilege levels are used in multiuser systems to prevent access to an area of the system memory.

S **Segment** indicates a data or code segment descriptor ($S = 1$), or a system segment descriptor ($S = 0$).

E **Executable** selects a data (stack) segment ($E = 0$) or a code segment ($E = 1$). E also defines the function of the next two bits (X and RW).

X If $E = 0$, then X indicates the direction of expansion for the data segment. If $X = 0$, the segment expands upward, as in a data segment; if $X = 1$, the segment expands downward as in a stack segment. If $E = 1$, then X indicates whether the privilege level of the code segment is ignored ($X = 0$) or observed ($X = 1$).

RW If $E = 0$, then RW indicates that the data segment may be written ($RW = 1$) or not written ($RW = 0$). If $E = 1$, then RW indicates that the code segment may be read ($RW = 1$) or not read ($RW = 0$).

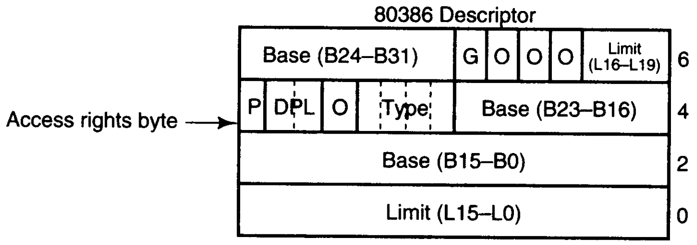


FIGURE 15–19 The general format of an 80386 system descriptor.

A **Accessed** is set each time that the microprocessor accesses the segment. It is sometimes used by the operating system to keep track of which segments have been accessed.

System Descriptor. The system descriptor is illustrated in Figure 15–19. There are 16 possible system descriptor types (see Table 15–1 for the different descriptor types), but not all are used in the 80386 microprocessor. Some of these types are defined for the 80286 so that the 80286 software is compatible with the 80386. Some of the types are new and unique to the 80386; some have yet to be defined and are reserved for future Intel products.

TABLE 15–1 80386 system descriptor types.

Type	*Purpose*
0000	Invalid
0001	Available 80286 TSS
0010	LDT
0011	Busy 80286 TSS
0100	80286 call gate
0101	Task gate (80286 or 80386)
0110	80286 interrupt gate
0111	80286 trap gate
1000	Invalid
1001	Available 80386 TSS
1010	Reserved for future Intel products
1011	Busy 80386 TSS
1100	80386 call gate
1101	Reserved for future Intel products
1110	80386 interrupt gate
1111	80836 trap gate

Descriptor Tables

The descriptor tables define all the segments used in the 80386 when it operates in the protected mode. There are three types of descriptor tables: the global descriptor table (GDT), the local descriptor table (LDT), and the interrupt descriptor table (IDT). The registers used by the 80386 to address these three tables are called the global descriptor table register (GDTR), the local descriptor table register (LDTR), and the interrupt descriptor table register (IDTR). These registers are loaded with the LGDT, LLDT, and LIDT instructions, respectively.

The **descriptor table** is a variable-length array of data, with each entry holding an 8-byte long descriptor. The local and global descriptor tables hold up to 8192 entries each, and the interrupt descriptor table holds up to 256 entries. A descriptor is indexed from either the local or global descriptor table by the selector that appears in a segment register. Figure 15–20 shows a segment register and the selector that it holds in the protected mode. The leftmost 13 bits index a descriptor, the TI bit selects either the local (*TI* = 1) or global (*TI* = 0) descriptor table, and the RPL bits indicate the requested privilege level.

Whenever a new selector is placed into one of the segment registers, the 80386 accesses one of the descriptor tables and automatically loads the descriptor into a program-invisible cache portion of the segment register. As long as the selector remains the same in the segment register, no additional accesses are required to the descriptor table.

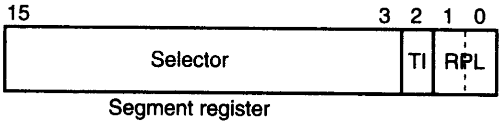


FIGURE 15–20 A segment register showing the selector, T1 bit, and requested privilege level (RPL) bits.

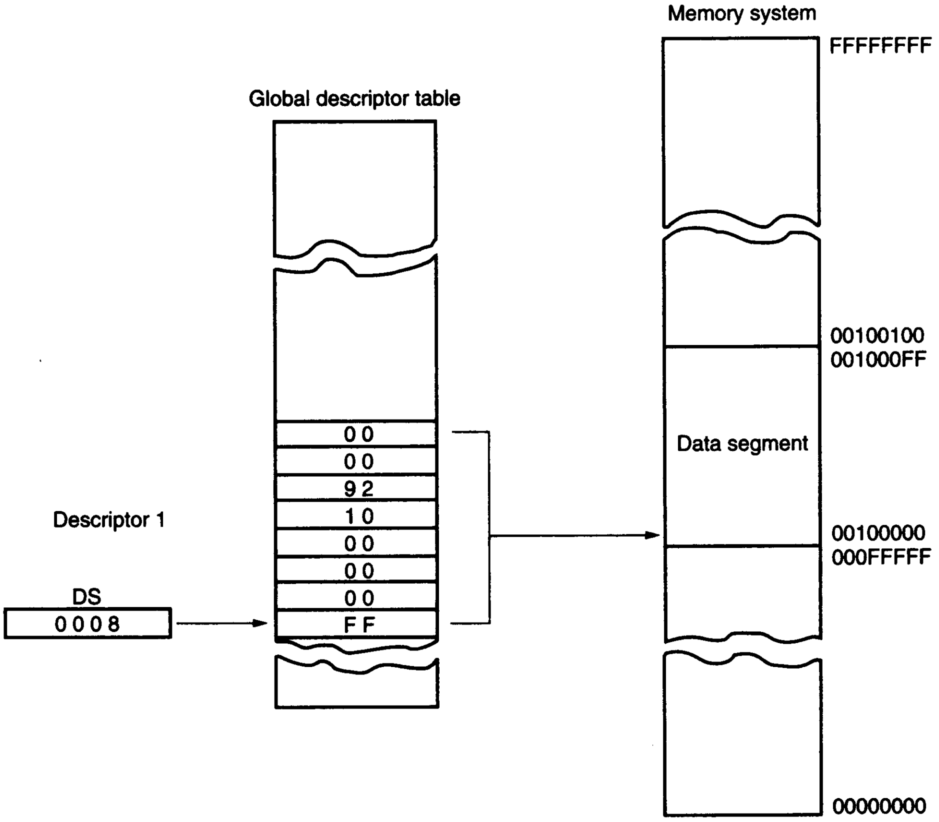


FIGURE 15–21 Using the DS register to select a descriptor from the global descriptor table. In this example, the DS register accesses memory locations 00100000H–001000FFH as a data segment.

The operation of fetching a new descriptor from the descriptor table is program-invisible because the microprocessor automatically accomplishes this each time that the segment register contents are changed in the protected mode.

Figure 15–21 shows how a sample global descriptor table (GDT), which is stored at memory address 00010000H, is accessed through the segment register and its selector. This table contains four entries. The first is a null (0) descriptor. Descriptor 0 must always be a null descriptor. The other entries address various segments in the 80386 protected mode memory system. In this illustration, the data segment register contains a 0008H. This means that the selector is indexing descriptor location 1 in the global descriptor table (*TI* = 0), with a requested privilege level of 00. Descriptor 1 is located eight bytes above the base descriptor table address, beginning at location 00010008H. The descriptor located in this memory location accesses a base address of 00200000H and a limit of 100H. This means that this descriptor addresses memory locations 00200000H–00200100H. Because this is the DS (data segment) register, the data segment is located at these locations in the memory system. If data are accessed outside of these boundaries, an interrupt occurs.

The local descriptor table (LDT) is accessed in the same manner as the global descriptor table (GDT). The only difference in access is that the TI bit is cleared for a global access and set for a local access. Another difference exists if the local and global descriptor table registers are examined. The global descriptor table register (GDTR) contains the base address of the global descriptor table and the limit. The local descriptor table register (LDTR) contains only a selector, and it is 16-bits wide. The contents of the LDTR addresses a type 0010 system

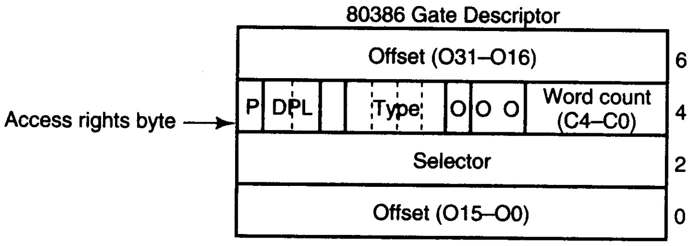


FIGURE 15–22 The gate descriptor for the 80386 microprocessor.

descriptor that contains the base address and limit of the LDT. This scheme allows one global table for all tasks; but allows many local tables, one or more for each task, if necessary. Global descriptors describe memory for the system, while local descriptors describe memory for applications or tasks.

Like the GDT, the interrupt descriptor table (IDT) is addressed by storing the base address and limit in the interrupt descriptor table register (IDTR). The main difference between the GDT and IDT is that the IDT contains only interrupt gates. The GDT and LDT contain segment and system descriptors, but never contain interrupt gates.

Figure 15–22 shows the gate descriptor, a special form of the system descriptor described earlier. (Refer to Table 15–1 for the different gate descriptor types.) Notice that the gate descriptor contains a 32-bit offset address, a word count, and a selector. The 32-bit offset address points to the location of the interrupt service procedure or other procedure. The word count indicates how many words are transferred from the caller's stack to the stack of the procedure accessed by a call gate. This feature of transferring data from the caller's stack is useful for implementing high-level languages such as C/C++. Note that the word count field is not used with an interrupt gate. The selector is used to indicate the location of the task state segment (TSS) in the GDT or LDT if it is a local procedure.

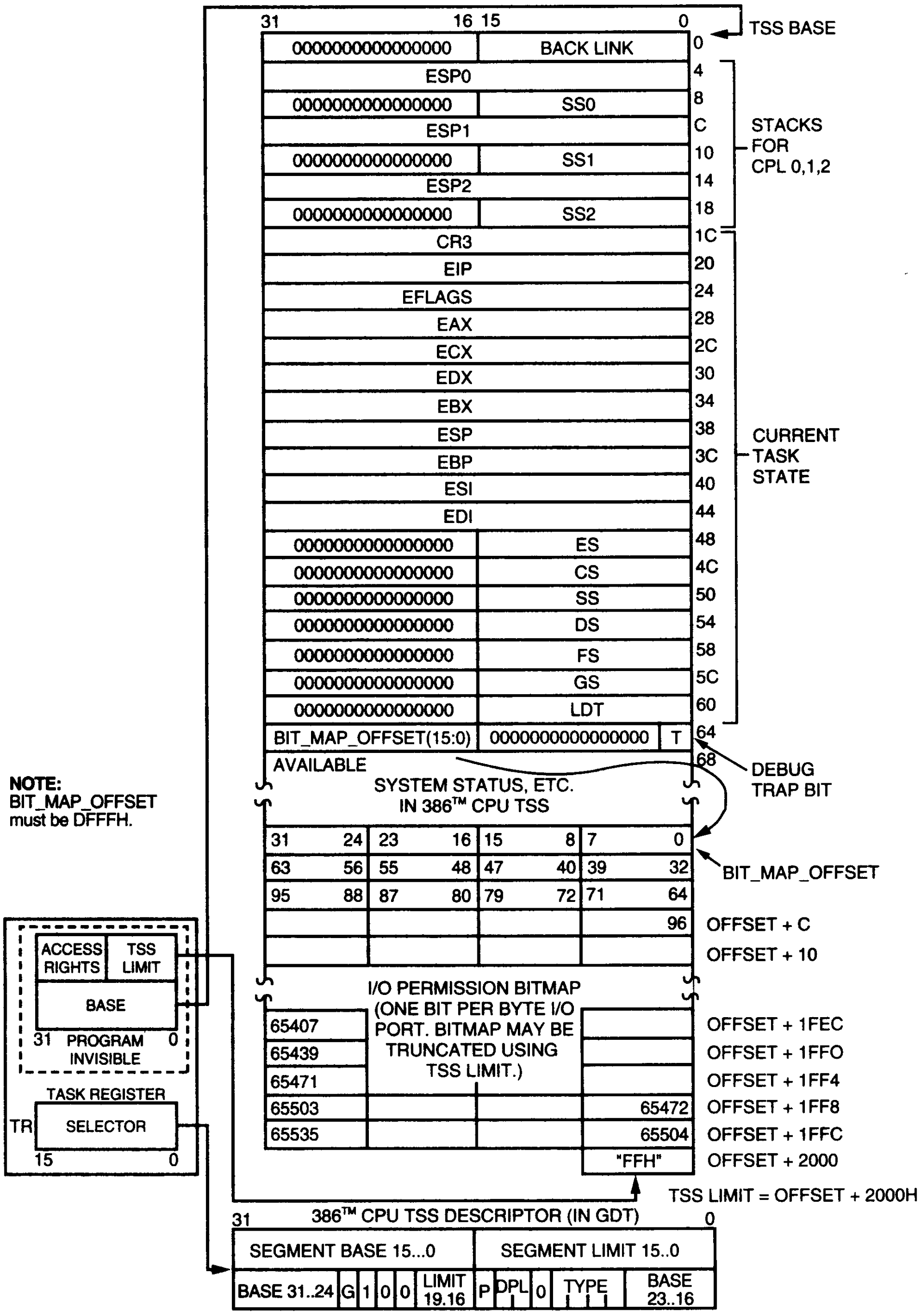
When a gate is accessed, the contents of the selector are loaded into the task register (TR), causing a task switch. The acceptance of the gate depends on the privilege and priority levels. A return instruction (RET) ends a call gate procedure and a return from interrupt instruction (IRET) ends an interrupt gate procedure. Tasks are usually accessed with a CALL or an INT instruction, where the call instruction addresses a call gate in the descriptor table and the interrupt addresses an interrupt descriptor.

The difference between real mode interrupts and protected mode interrupts is that the interrupt vector table is an IDT in the protected mode. The IDT still contains up to 256 interrupt levels, but each level is accessed through an interrupt gate instead of an interrupt vector. Thus, interrupt type number 2 is located at IDT descriptor number 2 at 16 locations above the base address of the IDT. This also means that the first 1K byte of memory no longer contains interrupt vectors, as it did in the real mode. The IDT can be located at any location in the memory system.

The Task State Segment (TSS)

The **task state segment** (TSS) descriptor contains information about the location, size, and privilege level of the task state segment, just as any other descriptor. The difference is that the TSS described by the TSS descriptor does not contains data or code. It contains the state of the task and linkage so tasks can be nested (one task can call a second, which can call a third, and so forth). The TSS descriptor is addressed by the **task register** (TR). The contents of the TR are changed by the LTR instruction. Whenever the protected mode program executes a far JMP or CALL instruction, the contents of TR are also changed. The LTR instruction is used to initially access a task during system initialization. After initialization, the CALL or JUMP instructions normally switch tasks. In most cases, we use the CALL instruction to initiate a new task.

The TSS is illustrated in Figure 15–23. As can be seen, the TSS is quite a formidable section of memory, containing many different types of information. The first word of the TSS is labeled **back-link.** This is the selector



NOTE: BIT_MAP_OFFSET must be DFFFH.

Type = 9: Available 386™ CPU TSS.
Type = B: Busy 386™ CPU TSS.

FIGURE 15–23 The task state segment (TSS) descriptor. (Courtesy of Intel Corporation.)

that is used, on a return (RET or IRET), to link back to the prior TSS by loading the back-link selector into the TR. The following word must contain a 0. The second through the seventh doublewords contain the ESP and ESS values for privilege levels 0–2. These are required in case the current task is interrupted so these privilege level (PL) stacks can be addressed. The eighth word (offset 1CH) contains the contents of CR3, which stores the base address of the prior state's page directory register. This must be restored if paging is in effect. The contents of the next 17 doublewords are loaded into the registers indicated. Whenever a task is accessed, the entire state of the machine (all of the registers) is stored in these memory locations and then reloaded from the same locations in the new TSS. The last word (offset 66H) contains the I/O permission bit map base address.

The I/O permission bit map allows the TSS to block I/O operations to inhibited I/O port addresses via an I/O permission denial interrupt. The permission denial interrupt is type number 13, the general protection fault interrupt. The I/O permission bit map base address is the offset address from the start of the TSS. This allows the same permission map to be used by many TSSs.

Each I/O permission bit map is 64K bits long (8K bytes), beginning at the offset address indicated by the I/O permission bit map base address. The first byte of the I/O permission bit map contains I/O permission for I/O ports 0000H–0007H. The rightmost bit contains the permission for port number 0000H. The leftmost bit contains the permission for port number 0007H. This sequence continues for the very last port address (FFFFH) stored in the leftmost bit of the last byte of the I/O permission bit map. A logic 0 placed in an I/O permission bit map bit enables the I/O port address, while a logic 1 inhibits or blocks the I/O port address. At present, only Windows NT uses the I/O permission scheme to disable I/O ports dependent on the application or the user.

In review of the operation of a task switch, which requires only 17 μs to execute, we list the following steps:

1. The gate contains the address of the procedure or location jumped to by the task switch. It also contains the selector number of the TSS descriptor and the number of words transferred from the caller to the user stack area for parameter passing.
2. The selector is loaded into TR from the gate. (This step is accomplished by a CALL or JMP that refers to a valid TSS descriptor.)
3. The TR selects the TSS.
4. The current state is saved in the current TSS and the new TSS is accessed with the state of the new task (all the registers) loaded into the microprocessor. The current state is saved at the TSS selector currently found in the TR. Once the current state is saved, a new value (by the JMP or CALL) for the TSS selector is loaded into TR and the new state is loaded from the new TSS.

The return from a task is accomplished by the following steps:

1. The current state of the microprocessor is saved in the current TSS.
2. The back-link selector is loaded to the TR to access the prior TSS so that the prior state of the machine can be returned to and be restored to the microprocessor. The return for a called TSS is accomplished by the IRET instruction.

15–4 MOVING TO PROTECTED MODE

In order to change the operation of the 80386 from the real mode to the protected mode, several steps must be followed. Real mode operation is accessed after a hardware reset or by changing the PE bit to a logic 0 in CR0. Protected mode is accessed by placing a logic 1 into the PE bit of CR0; before this is done, however, some other things must be initialized. The following steps accomplish the switch from the real mode to the protected mode:

1. Initialize the interrupt descriptor table so that it contains valid interrupt gates for at least the first 32 interrupt type numbers. The IDT may (and often does) contain up to 256 8-byte interrupt gates defining all 256 interrupt types.

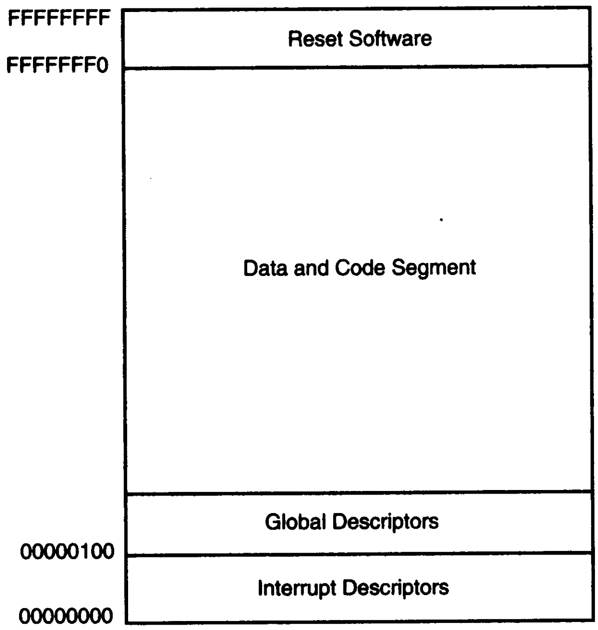


FIGURE 15–24 The memory map for Example 15–1.

2. Initialize the global descriptor table (GDT) so that it contains a null descriptor at descriptor 0, and valid descriptors for at least one code, one stack, and one data segment.
3. Switch to protected mode by setting the PE bit in CR0.
4. Perform an intrasegment (near) JMP to flush the internal instruction queue and load the TR with the base TSS descriptor.
5. Load all the data selectors (segment registers) with their initial selector values.
6. The 80386 is now operating in the protected mode, using the segment descriptors that are defined in GDT and IDT.

Figure 15–24 shows the protected system memory map set up by following steps 1–5. The software for this task is listed in Example 15–1. This system contains one data segment descriptor and one code segment descriptor with each segment set to 4G bytes in length. This is the simplest protected mode system possible: loading all the segment registers, except code, with the same data segment descriptor from the GDT. The privilege level is initialized to 00, the highest level. This system is most often used where one user has access to the microprocessor and requires the entire memory space. This program is designed for use in a system that does not use DOS or shell from Windows to DOS. Later in this section, we show how to go to protected mode in a DOS environment. (Please note that the software in Example 15–1 is designed for a standalone system such as the 80386EX, and not for use in the PC.)

EXAMPLE 15–1

```
                .MODEL SMALL
                .386P
0000            .DATA
0000  0040 [    IDT1   DD    64 DUP (?)     ;space for 32 interrupt vectors
           00000000
```

```
                      ]
                     ;
                     ;Global descriptor table
                     ;
0100                 DESC0  DQ      0                ;clear null descriptor
     0000000000000000
                     ;
                     ;code segment descriptor
                     ;
0108  FFFF           DESC1  DW      0FFFFH           ;limit = 4G
010A  0000                  DW      0                ;base address = 00000000H
010C  0000                  DW      0
010E  9E                    DB      9EH              ;code segment
010F  8F                    DB      8FH              ;G = 1
0110  00                    DB      0
                     ;
                     ;data segment descriptor
                     ;
0111  FFFF           DESC2  DW      0FFFFH           ;limit = 4G
0113  0000                  DW      0                ;base address = 00000000H
0115  0000                  DW      0
0117  92                    DB      92H              ;data segment
0118  8F                    DB      8FH              ;G = 1
0119  00                    DB      0
                     ;
                     ;IDT table data
                     ;
011A  00FF           IDT    DW      0FFH             ;set limit to FFH
011C  00000000       IDTA   DD      0
                     ;
                     ;GDT table data
                     ;
0120  0017           GDT    DW      17H              ;set limit to 17H
0122  00000000       GDTA   DD      0
0000                 .CODE
                     MAK32  MACRO   SEG,OFF          ;;make a seg+off a linear address
                            MOV     EAX,0
                            MOV     EBX,0
                            MOV     AX,SEG
                            MOV     BX,OFF
                            SHL     EAX,4
                            ADD     EAX,EBX
                            ENDM
                     .STARTUP
                            MAK32   DS,OFFSET IDT1
0028  66| A3 011C R         MOV     IDTA,EAX         ;save IDT address
                            MAK32   DS,OFFSET DESC0
0044  66| A3 0122 R         MOV     GDTA,EAX         ;save GDT address
0048  B9 0020               MOV     CX,32
004B  BF 0000 R             MOV     DI,OFFSET IDT1
004E  BE 0000               MOV     SI,0
0051  B8 0000               MOV     AX,0
0054  8E C0                 MOV     ES,AX
                     .REPEAT                         ;setup first 32 interrupts
                            MAK32   ES:[SI+2],ES:[SI]
0070  89 05                 MOV     [DI],AX
0072  66| C1 E8 10          SHR     EAX,16
0076  89 45 06              MOV     [DI+6],AX
0079  C7 45 02 0008         MOV     WORD PTR [DI+2],8
007E  C7 45 04 8F00         MOV     WORD PTR [DI+4],8F00H
0083  83 C7 08              ADD     DI,8
0086  83 C6 04              ADD     SI,4
                     .UNTILCXZ
```

```
008B  0F 01 1E 011A R    LIDT   FWORD PTR IDT  ;load IDT
0090  0F 01 16 0120 R    LGDT   FWORD PTR GDT  ;load GDT

0095  0F 20 C0           MOV    EAX,CR0        ;set PE
0098  66| 83 C8 01       OR     EAX,1
009C  0F 22 C0           MOV    CR0,EAX

009F  EB 00              JMP    START          ;near jump

00A1            START:
00A1  B8 0010            MOV    AX,10H         ;set selector 2
00A4  8E D8              MOV    DS,AX
00A6  8E C0              MOV    ES,AX
00A8  8E D0              MOV    SS,AX
00AA  8E E8              MOV    GS,AX
00AC  8E E0              MOV    FS,AX
00AE  66| BC FFFFF000    MOV    ESP,0FFFFF000H

                ;now in protected mode.

                         END
```

In more complex systems, the steps required to initialize the system in the protected mode are more involved. For complex systems that are often multiuser systems, the registers are loaded by using the task state segment (TSS). The steps required to place the 80386 into protected mode operation for a more complex system using a task switch follow:

1. Initialize the interrupt descriptor table so that it refers to valid interrupt descriptors with at least 32 descriptors in the IDT.
2. Initialize the global descriptor table so that it contains at least two task state segment (TSS) descriptors, and the initial code and data segments required for the initial task.
3. Initialize the task register (TR) so that it points to a valid TSS; when the initial task switch occurs and accesses the new TSS, the current registers are stored in the initial TSS.
4. Switch to protected mode by using an intrasegment (near) jump to flush the internal instruction queue. Load the TR with the current TSS selector.
5. Load the TR with a far jump instruction to access the new TSS and save the current state.
6. The 80386 is now operating in the protected mode under control of the first task.

Example 15–2 illustrates the software required to initialize the system and switch to protected mode by using a task switch. The initial system task operates at the highest level of protection (00) and controls the entire operating environment for the 80386. In many cases, it is used to boot (load) software that allows many users to access the system in a multiuser environment.

EXAMPLE 15–2

```
               .MODEL  SMALL
               .386P
               .STACK   800H
0000           .DATA
0008           DESC    STRUC                       ;define descriptor structure

0000  0000     LIM_L   DW    0
0002  0000     BAS_L   DW    0
0004  00       BAS_M   DB    0
0005  00       ACCESS  DB    0
0006  00       LIM_M   DB    0
0007  00       BAS_H   DB    0

               DESC    ENDS

0068           TSS     STRUC                       ;define TSS structure
```

```
0000  0000          BACK_L DW     0
0002  0000                 DW     0
0004  00000000      ESP0   DD     0
0008  0000          SS0    DW     0
000A  0000                 DW     0
000C  00000000      ESP1   DD     0
0010  0000          SS1    DW     0
0012  0000                 DW     0
0014  00000000      ESP2   DD     0
0018  0000          SS2    DW     0
001A  0000                 DW     0
001C  00000000      CCR3   DD     0
0020  00000000      EIP    DD     0
0024  00000000      TFALGS DD     0
0028  00000000      EEAX   DD     0
002C  00000000      EECX   DD     0
0030  00000000      EEDX   DD     0
0034  00000000      EEBX   DD     0
0038  00000000      EESP   DD     0
003C  00000000      EEBP   DD     0
0040  00000000      EESI   DD     0
0044  00000000      EEDI   DD     0
0048  0020          EES    DW     20H
004A  0000                 DW     0
004C  0018          ECS    DW     18H
004E  0000                 DW     0
0050  0020          ESS    DW     20H
0052  0000                 DW     0
0054  0020          EDS    DW     20H
0056  0000                 DW     0
0058  0020          EFS    DW     20H
005A  0000                 DW     0
005C  0020          EGS    DW     20H
005E  0000                 DW     0
0060  0000          ELDT   DW     0
0062  0000                 DW     0
0064  0000                 DW     0
0066  0000          BITM   DW     0

                    TSS    ENDS

0000  0000 0000     TSS1   TSS    <>                 ;task state 1
      00000000
      0000 0000 00000000
      0000 0000 00000000
      0000 0000 00000000
      00000000 00000000
      00000000 00000000
      00000000 00000000
      00000000 00000000
      00000000 00000000
      0020 0000 0018
      0000 0020 0000
      0020 0000 0020
      0000 0020 0000
      0000 0000 0000
      0000
0068  0000 0000     TSS2   TSS    <>                 ;task state 2
      00000000
      0000 0000 00000000
      0000 0000 00000000
      0000 0000 00000000
      00000000 00000000
```

EXAMPLE 15–3

```
                            ;A program that displays the contents of any area of memory
                            ;including extended memory.
                            ;***command line syntax***
                            ;EDUMP XXXX,YYYY  where XXXX is the start address and YYYY is
                            ;the end address.
                            ;Note:  this program must be executed from WINDOWS.
                            ;
                                  .MODEL SMALL
                                  .386
                                  .STACK 1024           ;stack area of 1,024 bytes
0000                              .DATA
0000  00000000              ENTRY DD   ?                ;DPMI entry point
0004  00000000              EXIT  DD   ?                ;DPMI exit point
0008  00000000              FIRST DD   ?                ;first address
000C  00000000              LAST1 DD   ?                ;last address
0010  0000                  MSIZE DW   ?                ;memory needed for DPMI
0012  0D 0A 0A 50 61 ERR1   DB   13,10,10,'Parameter error.$'
      72 61 6D 65 74
      65 72 20 65 72
      72 6F 72 2E 24
0026  0D 0A 0A 44 50 ERR2   DB   13,10,10,'DPMI not present.$'
      4D 49 20 6E 6F
      74 20 70 72 65
      73 65 6E 74 2E
      24
003B  0D 0A 0A 4E 6F ERR3   DB   13,10,10,'Not enough real memory.$'
      74 20 65 6E 6F
      75 67 68 20 72
      65 61 6C 20 6D
      65 6D 6F 72 79
      2E 24
0056  0D 0A 0A 43 6F ERR4   DB   13,10,10,'Could not move to protected mode.$'
      75 6C 64 20 6E
      6F 74 20 6D 6F
      76 65 20 74 6F
      20 70 72 6F 74
      65 63 74 65 64
      20 6D 6F 64 65
      2E 24
007B  0D 0A 0A 43 61 ERR5   DB   13,10,10,'Cannot allocate selector.$'
      6E 6E 6F 74 20
      61 6C 6C 6F 63
      61 74 65 20 73
      65 6C 65 63 74
      6F 72 2E 24
0098  0D 0A 0A 43 61 ERR6   DB   13,10,10,'Cannot use base address.$'
      6E 6E 6F 74 20
      75 73 65 20 62
      61 73 65 20 61
      64 64 72 65 73
      73 2E 24
00B4  0D 0A 0A 43 61 ERR7   DB   13,10,10,'Cannot allocate 64K to limit.$'
      6E 6E 6F 74 20
      61 6C 6C 6F 63
      61 74 65 20 36
      34 4B 20 74 6F
      20 6C 69 6D 69
      74 2E 24
00D5  0D 0A 24        CRLF  DB   13,10,'$'
00D8  50 72 65 73 73 MES1   DB   'Press any key...$'
      20 61 6E 79 20
      6B 65 79 2E 2E
      2E 24
                     ;
                     ;register array storage for DPMI function 0300H
                     ;
00E9  = 00E9         ARRAY EQU  THIS BYTE
00E9  00000000       REDI  DD   0                   ;EDI
00ED  00000000       RESI  DD   0                   ;ESI
00F1  00000000       REBP  DD   0                   ;EBP
00F5  00000000             DD   0                   ;reserved
00F9  00000000       REBX  DD   0                   ;EBX
00FD  00000000       REDX  DD   0                   ;EDX
0101  00000000       RECX  DD   0                   ;ECX
0105  00000000       REAX  DD   0                   ;EAX
0109  0000           RFLAG DW   0                   ;flags
010B  0000           RES   DW   0                   ;ES
010D  0000           RDS   DW   0                   ;DS
010F  0000           RFS   DW   0                   ;FS
0111  0000           RGS   DW   0                   ;GS
0113  0000           RIP   DW   0                   ;IP
0115  0000           RCS   DW   0                   ;CS
0117  0000           RSP   DW   0                   ;SP
0119  0000           RSS   DW   0                   ;SS
0000                       .CODE
                           .STARTUP
0010  8C C0                MOV  AX,ES
0012  8C DB                MOV  BX,DS               ;find size of program and data
0014  2B D8                SUB  BX,AX
0016  8B C4                MOV  AX,SP               ;find stack size
0018  C1 E8 04             SHR  AX,4
001B  40                   INC  AX
001C  03 D8                ADD  BX,AX               ;BX = length in paragraphs
001E  B4 4A                MOV  AH,4AH
0020  CD 21                INT  21H                 ;modify memory allocation
0022  E8 00D1              CALL GETDA               ;get command line information
0025  73 0A                JNC  MAIN1               ;if parameters are good
0027  B4 09                MOV  AH,9                ;parameter error
0029  BA 0012 R            MOV  DX,OFFSET ERR1
002C  CD 21                INT  21H
002E  E9 00AA              JMP  MAINE               ;exit to DOS
0031                 MAIN1:
0031  E8 00AB              CALL ISDPMI              ;is DPMI loaded?
0034  72 0A                JC   MAIN2               ;if DPMI present
0036  B4 09                MOV  AH,9
0038  BA 0026 R            MOV  DX,OFFSET ERR2
003B  CD 21                INT  21H                 ;display DPMI not present
003D  E9 009B              JMP  MAINE               ;exit to DOS
0040                 MAIN2:
0040  B8 0000              MOV  AX,0                ;indicate 0 memory needed
0043  83 3E 0010 R 00      CMP  MSIZE,0
0048  74 F6                JE   MAIN2               ;if DPMI needs no memory
004A  8B 1E 0010 R         MOV  BX,MSIZE            ;get amount
004E  B4 48                MOV  AH,48H
0050  CD 21                INT  21H                 ;allocate memory for DPMI
0052  73 09                JNC  MAIN3
0054  B4 09                MOV  AH,9                ;if not enough real memory
0056  BA 003B R            MOV  DX,OFFSET ERR3
0059  CD 21                INT  21H
005B  EB 7E                JMP  MAINE               ;exit to DOS
005D                 MAIN3:
005D  8E C0                MOV  ES,AX
005F  B8 0000              MOV  AX,0                ;16-bit application
0062  FF 1E 0000 R         CALL DS:ENTRY            ;switch to protected mode
0066  73 09                JNC  MAIN4
0068  B4 09                MOV  AH,9                ;if switch failed
```

```
006A  BA 0056 R            MOV  DX,OFFSET ERR4
006D  CD 21                INT  21H
006F  EB 6A                JMP  MAINE              ;exit to DOS
                    ;
                    ;PROTECTED MODE
                    ;
0071                 MAIN4:
0071  B8 0000              MOV  AX,0000H           ;get local selector
0074  B9 0001              MOV  CX,1               ;only one is needed
0077  CD 31                INT  31H
0079  72 48                JC   MAIN7              ;if error
007B  8B D8                MOV  BX,AX              ;save selector
007D  8E C0                MOV  ES,AX              ;load ES with selector
007F  B8 0007              MOV  AX,0007H           ;set base address
0082  8B 0E 000A R         MOV  CX,WORD PTR FIRST+2
0086  8B 16 0008 R         MOV  DX,WORD PTR FIRST
008A  CD 31                INT  31H
008C  72 3D                JC   MAIN8              ;if error
008E  B8 0008              MOV  AX,0008H
0091  B9 0000              MOV  CX,0
0094  BA FFFF              MOV  DX,0FFFFH          ;set limit to 64K
0097  CD 31                INT  31H
0099  72 38                JC   MAIN9              ;if error
009B  B9 0018              MOV  CX,24              ;load line count
009E  BE 0000              MOV  SI,0               ;load offset
00A1                 MAIN5:
00A1  E8 00F4              CALL DADDR              ;display address, if needed
00A4  E8 00CE              CALL DDATA              ;display data
00A7  46                   INC  SI                 ;point to next data
00A8  66| A1 0008 R        MOV  EAX,FIRST          ;test for end
00AC  66| 3B 06 000C R     CMP  EAX,LAST1
00B1  74 07                JE   MAIN6              ;if done
00B3  66| FF 06 0008 R     INC  FIRST
00B8  EB E7                JMP  MAIN5
00BA                 MAIN6:
00BA  B8 0001              MOV  AX,0001H           ;release descriptor
00BD  8C C3                MOV  BX,ES
00BF  CD 31                INT  31H
00C1  EB 18                JMP  MAINE              ;exit to DOS
00C3                 MAIN7:
00C3  BA 007B R            MOV  DX,OFFSET ERR5
00C6  E8 0096              CALL DISPS              ;display cannot allocate selector
00C9  EB 10                JMP  MAINE              ;exit to DOS
00CB                MAIN8:
00CB  BA 0098 R            MOV  DX,OFFSET ERR6
00CE  E8 008E              CALL DISPS              ;display cannot use base address
00D1  EB E7                JMP  MAIN6              ;release descriptor
00D3                MAIN9:
00D3  BA 00B4 R            MOV  DX,OFFSET ERR7
00D6  E8 0086              CALL DISPS              ;display cannot allocate 64K limit
00D9  EB DF                JMP  MAIN6              ;release descriptor
00DB                MAINE:
                           .EXIT
                    ;
                    ;The ISDPMI procedure tests for the presence of DPMI.
                    ;***exit parameters***
                    ;carry = 1; if DPMI is present
                    ;carry = 0; if DPMI is not present
                    ;
00DF                ISDPMI PROC NEAR

00DF  B8 1687              MOV  AX,1687H           ;get DPMI status
00E2  CD 2F                INT  2FH                ;DOS multiplex
00E4  0B C0                OR   AX,AX
```

```
00E6  75 0D                JNZ  ISDPMI1           ;if no DPMI
00E8  89 36 0010 R         MOV  MSIZE,SI          ;save amount of memory needed
00EC  89 3E 0000 R         MOV  WORD PTR ENTRY,DI
00F0  8C 06 0002 R         MOV  WORD PTR ENTRY+2,ES
00F4  F9                   STC
00F5               ISDPMI1:
00F5  C3                   RET

00F6               ISDPMI ENDP
                   ;
                   ;The GETDA procedure retrieves the command line parameters
                   ;for memory display in hexadecimal.
                   ;FIRST = the first address from the command line
                   ;LAST1 = the last address from the command line
                   ;***return parameters***
                   ;carry = 1; if error
                   ;carry = 0; for no error
                   ;
00F6               GETDA  PROC NEAR

00F6  1E                   PUSH DS
00F7  06                   PUSH ES
00F8  1F                   POP  DS
00F9  07                   POP  ES                ;exchange ES with DS
00FA  BE 0081              MOV  SI,81H            ;address command line
00FD               GETDA1:
00FD  AC                   LODSB                  ;skip spaces
00FE  3C 20                CMP  AL,' '
0100  74 FB                JE   GETDA1            ;if space
0102  3C 0D                CMP  AL,13
0104  74 1E                JE   GETDA3            ;if enter = error
0106  4E                   DEC  SI                ;adjust SI
0107               GETDA2:
0107  E8 0020              CALL GETNU             ;get first number
010A  3C 2C                CMP  AL,','
010C  75 16                JNE  GETDA3            ;if no comma = error
010E  66| 26: 89 16 0008 R MOV  ES:FIRST,EDX
0114  E8 0013              CALL GETNU             ;get second number
0117  3C 0D                CMP  AL,13
0119  75 09                JNE  GETDA3            ;if error
011B  66| 26: 89 16 000C R MOV  ES:LAST1,EDX
0121  F8                   CLC                    ;indicate no error
0122  EB 01                JMP  GETDA4            ;return no error
0124               GETDA3:
0124  F9                   STC                    ;indicate error
0125               GETDA4:
0125  1E                   PUSH DS                ;exchange ES with DS
0126  06                   PUSH ES
0127  1F                   POP  DS
0128  07                   POP  ES
0129  C3                   RET
012A               GETDA ENDP
                   ;
                   ;The GETNU procedure extracts a number from the command line
                   ;and returns with it in EDX and last command line character in
                   ;AL as a delimiter.
                   ;
012A               GETNU  PROC NEAR

012A  66| BA 00000000      MOV  EDX,0             ;clear result
0130               GETNU1:
0130  AC                   LODSB                  ;get digit from command line
                           .IF  AL >= 'a' && AL <= 'z'
```

```
0139  2C 20                        SUB  AL,20H         ;make uppercase
                          .ENDIF
013B  2C 30                SUB  AL,'0'               ;convert from ASCII
013D  72 12                JB   GETNU2               ;if not a number
                           .IF  AL > 9               ;convert A-F from ASCII
0143  2C 07                    SUB  AL,7
                          .ENDIF
0145  3C 0F                CMP  AL,0FH
0147  77 08                JA   GETNU2               ;if not 0-F
0149  66| C1 E2 04         SHL  EDX,4
014D  02 D0                ADD  DL,AL                ;add digit to EDX
014F  EB DF                JMP  GETNU1               ;get next digit
0151                GETNU2:
0151  8A 44 FF             MOV  AL,[SI-1]            ;get delimiter
0154  C3                   RET

0155                GETNU  ENDP
                    ;
                    ;The DISPC procedure displays the ASCII character found
                    ;in register AL.
                    ;***uses***
                    ;INT21H
                    ;
0155                DISPC  PROC NEAR

0155  52                   PUSH DX
0156  8A D0                MOV  DL,AL
0158  B4 06                MOV  AH,6
015A  E8 0084              CALL INT21H               ;do real INT 21H
015D  5A                   POP  DX
015E  C3                   RET

015F                DISPC ENDP
                    ;
                    ;The DISPS procedure displays a character string from
                    ;protected mode addressed by DS:EDX.
                    ;***uses***
                    ;DISPC
                    ;
015F                DISPS  PROC NEAR

015F  66| 81 E2 0000FFFF   AND  EDX,0FFFFH
0166  67& 8A 02            MOV  AL,[EDX]             ;get character
0169  3C 24                CMP  AL,'$'               ;test for end
016B  74 07                JE   DISP1                ;if end
016D  66| 42               INC  EDX                  ;address next character
016F  E8 FFE3              CALL DISPC                ;display character
0172  EB EB                JMP  DISPS                ;repeat until $
0174                DISP1:
0174  C3                   RET

0175                DISPS  ENDP
                    ;
                    ;The DDATA procedure displays a byte of data at the location
                    ;addressed by ES:SI. The byte is followed by one space.
                    ;***uses***
                    ;DIP and DISPC
                    ;
0175                DDATA PROC NEAR

0175  26: 8A 04            MOV  AL,ES:[SI]           ;get byte
0178  C0 E8 04             SHR  AL,4
017B  E8 000C              CALL DIP                  ;display first digit
```

```
017E  26: 8A 04            MOV  AL,ES:[SI]        ;get byte
0181  E8 0006              CALL DIP               ;display second digit
0184  B0 20                MOV  AL,' '            ;display space
0186  E8 FFCC              CALL DISPC
0189  C3                   RET

018A              DDATA ENDP
                  ;
                  ;The DIP procedure displays the right nibble found in AL as a
                  ;hexadecimal digit.
                  ;***uses***
                  ;DISPC
                  ;
018A              DIP   PROC NEAR

018A  24 0F                AND  AL,0FH            ;get right nibble
018C  04 30                ADD  AL,30H            ;convert to ASCII
                           .IF  AL > 39H          ;if A-F
0192  04 07                    ADD  AL,7
                           .ENDIF
0194  E8 FFBE              CALL DISPC             ;display digit
0197  C3                   RET

0198              DIP   ENDP
                  ;
                  ;The DADDR procedure displays the hexadecimal address found
                  ;in DS:FIRST if it is a paragraph boundary.
                  ;***uses***
                  ;DIP, DISPS, DISPC, and INT21H
                  ;
0198              DADDR PROC NEAR

0198  66| A1 0008 R        MOV  EAX,FIRST         ;get address
019C  A8 0F                TEST AL,0FH            ;test for XXXXXXX0
019E  75 40                JNZ  DADDR4            ;if not, don't display address
01A0  BA 00D5 R            MOV  DX,OFFSET CRLF
01A3  E8 FFB9              CALL DISPS             ;display CR and LF
01A6  49                   DEC  CX                ;decrement line count
01A7  75 18                JNZ  DADDR2            ;if not end of page
01A9  BA 00D8 R            MOV  DX,OFFSET MES1    ;if end of page
01AC  E8 FFB0              CALL DISPS             ;display press any key
01AF              DADDR1:
01AF  B4 06                MOV  AH,6              ;get any key, no echo
01B1  B2 FF                MOV  DL,0FFH
01B3  E8 002B              CALL INT21H            ;do real INT 21H
01B6  74 F7                JZ   DADDR1            ;if nothing typed
01B8  BA 00D5 R            MOV  DX,OFFSET CRLF
01BB  E8 FFA1              CALL DISPS             ;display CRLF
01BE  B9 0018              MOV  CX,24             ;reset line count
01C1              DADDR2:
01C1  51                   PUSH CX                ;save line count
01C2  B9 0008              MOV  CX,8              ;load digit count
01C5  66| 8B 16 0008 R     MOV  EDX,FIRST         ;get address
01CA              DADDR3:
01CA  66| C1 C2 04         ROL  EDX,4
01CE  8A C2                MOV  AL,DL
01D0  E8 FFB7              CALL DIP               ;display digit
01D3  E2 F5                LOOP DADDR3            ;repeat 8 times
01D5  59                   POP  CX                ;retrieve line count
01D6  B0 3A                MOV  AL,':'
01D8  E8 FF7A              CALL DISPC             ;display colon
01DB  B0 20                MOV  AL,' '
01DD  E8 FF75              CALL DISPC             ;display space
```

```
01E0                DADDR4:
01E0  C3                   RET

01E1                DADDR ENDP
                    ;
                    ;The INT21H procedure gains access to the real mode DOS
                    ;INT 21H instruction with the parameters intact.
                    ;
01E1                INT21H PROC NEAR

01E1  66| A3 0105 R        MOV  REAX,EAX          ;save registers
01E5  66| 89 1E 00F9 R     MOV  REBX,EBX
01EA  66| 89 0E 0101 R     MOV  RECX,ECX
01EF  66| 89 16 00FD R     MOV  REDX,EDX
01F4  66| 89 36 00ED R     MOV  RESI,ESI
01F9  66| 89 3E 00E9 R     MOV  REDI,EDI
01FE  66| 89 2E 00F1 R     MOV  REBP,EBP
0203  9C                   PUSHF
0204  58                   POP  AX
0205  A3 0109 R            MOV  RFLAG,AX
0208  06                   PUSH ES                ;do DOS interrupt
0209  B8 0300              MOV  AX,0300H
020C  BB 0021              MOV  BX,21H
020F  B9 0000              MOV  CX,0
0212  1E                   PUSH DS
0213  07                   POP  ES
0214  BF 00E9 R            MOV  DI,OFFSET ARRAY
0217  CD 31                INT  31H
0219  07                   POP  ES
021A  A1 0109 R            MOV  AX,RFLAG          ;restore registers
021D  50                   PUSH AX
021E  9D                   POPF
021F  66| 8B 3E 00E9 R     MOV  EDI,REDI
0224  66| 8B 36 00ED R     MOV  ESI,RESI
0229  66| 8B 2E 00F1 R     MOV  EBP,REBP
022E  66| A1 0105 R        MOV  EAX,REAX
0232  66| 8B 1E 00F9 R     MOV  EBX,REBX
0237  66| 8B 0E 0101 R     MOV  ECX,RECX
023C  66| 8B 16 00FD R     MOV  EDX,REDX
0241  C3                   RET

0242                INT21H ENDP
                           END
```

You might notice that the DOS INT 21H function call must be treated differently when operating in the protected mode. The procedure that calls a DOS INT 21H is at the end of Example 15–3. Because this is extremely long and time consuming, we have tended to move away from using the DOS interrupts from a Windows application. The best way to develop software for Windows is through the use of C/C++ with the inclusion of assembly language procedures for arduous tasks.

15–5 VIRTUAL 8086 MODE

One special mode of operation not discussed thus far is the virtual 8086 mode. This special mode is designed so that multiple 8086 real-mode software applications can execute at one time. The PC operates in this mode for DOS applications. Figure 15–25 illustrates two 8086 applications mapped into the 80386 using the virtual mode. If the operating system allows multiple applications to execute, it is usually done through a technique called **time-slicing.** The operating system allocates a set amount of time to each task. For example, if three tasks are executing, the

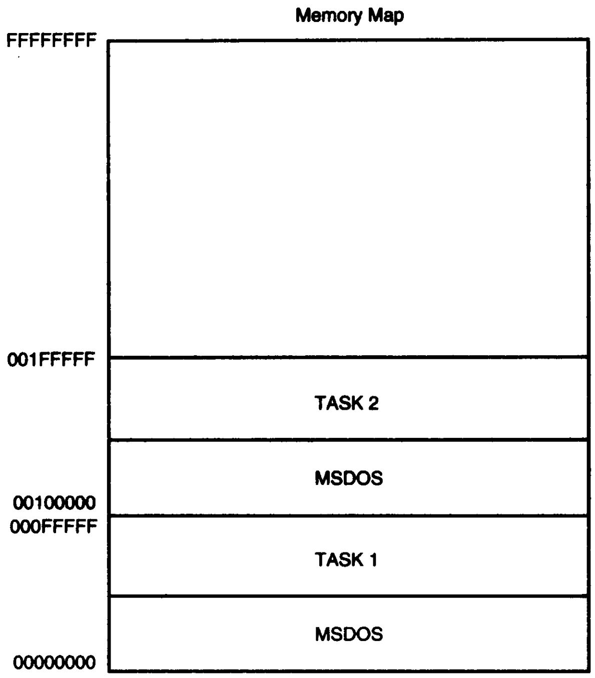


FIGURE 15–25 Two tasks resident to an 80386 operated in the virtual 8086 mode.

operating system can allocate 1 ms to each task. This means that after each millisecond, a task switch occurs to the next task. In this manner, all tasks receive a portion of the microprocessor's execution time, resulting in a system that appears to execute more than one task at a time. The task times can be adjusted to give any task any percentage of the microprocessor execution time.

A system that can use this technique is a print spooler. The print spooler can function in one DOS partition and be accessed 10 percent of the time. This allows the system to print using the print spooler, but it doesn't detract for the system because it uses only 10 percent of the system time.

The main difference between 80386 protected mode operation and the virtual 8086 mode is the way the segment registers are interpreted by the microprocessor. In the virtual 8086 mode, the segment registers are used as they are in the real mode: as a segment address and an offset address capable of accessing a 1M-byte memory space from location 00000H–FFFFFH. Access to many virtual 8086 mode systems is made possible by the paging unit that is explained in the next section. Through paging, the program still accesses memory below the 1M-byte boundary, yet the microprocessor can access a physical memory space at any location in the 4G-byte range of the memory system.

Virtual 8086 mode is entered by changing the VM bit in the EFLAG register to a logic 1. This mode is entered via an IRET instruction if the privilege level is 00. This bit cannot be set in any other manner. An attempt to access a memory address above the 1M-byte boundary will cause a type-13 interrupt to occur.

The virtual 8086 mode can be used to share one microprocessor with many users by partitioning the memory so that each user has its own DOS partition. User 1 can be allocated memory locations 00100000H–01FFFFFH, user 2 can be allocated locations 0020000H–02FFFFFH, and so forth. The system software located at memory locations 00000000H–000FFFFFH can then share the microprocessor between users by switching from one to another to execute software. In this manner, one microprocessor is shared by many users.

15–6 THE MEMORY PAGING MECHANISM

The paging mechanism allows any linear (logical) address, as it is generated by a program, to be placed into any physical memory page, as generated by the paging mechanism. A **linear memory page** is a page that is addressed with a selector and an offset in either the real or protected mode. A **physical memory page** is a page that exists at some actual physical memory location. For example, linear memory location 20000H could be mapped into physical memory location 30000H, or any other location, with the paging unit. This means that an instruction that accesses location 20000H actually accesses location 30000H.

Each 80386 memory page is 4K bytes long. Paging allows the system software to be placed at any physical address with the paging mechanism. Three components are used in page address translation: the page directory, the page table, and the actual physical memory page. Note that EEM386.EXE, the extended memory manager, uses the paging mechanism to simulate expanded memory in extended memory and to generate upper memory blocks between system ROMs.

The Page Directory

The page directory contains the location of up to 1024 page translation tables. Each page translation table translates a logic address into a physical address. The page directory is stored in the memory and accessed by the page descriptor address register (CR3) (see Figure 15–14). Control register CR3 holds the base address of the page directory, which starts at any 4K-byte boundary in the memory system. The MOV CR3,reg instruction is used to initialize CR3 for paging. In a virtual 8086 mode system, each 8086 DOS partition would have its own page directory.

The page directory contains up to 1024 entries, which are each four bytes long. The page directory itself occupies one 4K-byte memory page. Each entry in the page directory (see Figure 15–26) translates the leftmost 10 bits of the memory address. This 10-bit portion of the linear address is used to locate different page tables for different page table entries. The page table address (A32–A12), stored in a page directory entry, accesses a 4K-byte long page translation table. To completely translate any linear address into any physical address requires 1024 page tables that are each 4K bytes long, plus the page table directory, which is also 4K bytes long. This translation scheme requires up to 4M plus 4K bytes of memory for a full address translation. Only the largest operating systems support this size address translation. Many commonly found operating systems translate only the first 16M bytes of the memory system if paging is enabled. This includes programs such as Windows. This translation requires four entries in the page directory (16 bytes) and four complete page tables (16K bytes).

The page table directory entry control bits, as illustrated in Figure 15–26, each perform the following functions:

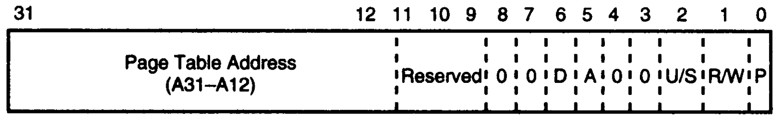


FIGURE 15–26 The page table directory entry.

D — **Dirty** is undefined for page table directory entries by the 80386 microprocessor and is provided for use by the operating system.

A — **Accessed** is set to a logic 1 whenever the microprocessor accesses the page directory entry.

R/W and U/S — **Read/write** and **user/supervisor** are both used in the protection scheme, as listed in Table 15–2. Both bits combine to develop paging priority level pro- tection for level 3, the lowest user level.

P — **Present,** if a logic 1, indicates that the entry can be used in address translation. If $P = 0$, the entry cannot be used for translation. A not present entry can be used for other purposes, such as indicating that the page is currently stored on the disk. If $P = 0$, the remaining bits of the entry can be used to indicate the location of the page on the disk memory system.

TABLE 15–2 Protection for level 3 using U/S and R/W.

U/S	*R/W*	*Access Level 3*
0	0	None
0	1	None
1	0	Read-only
1	1	Read/write

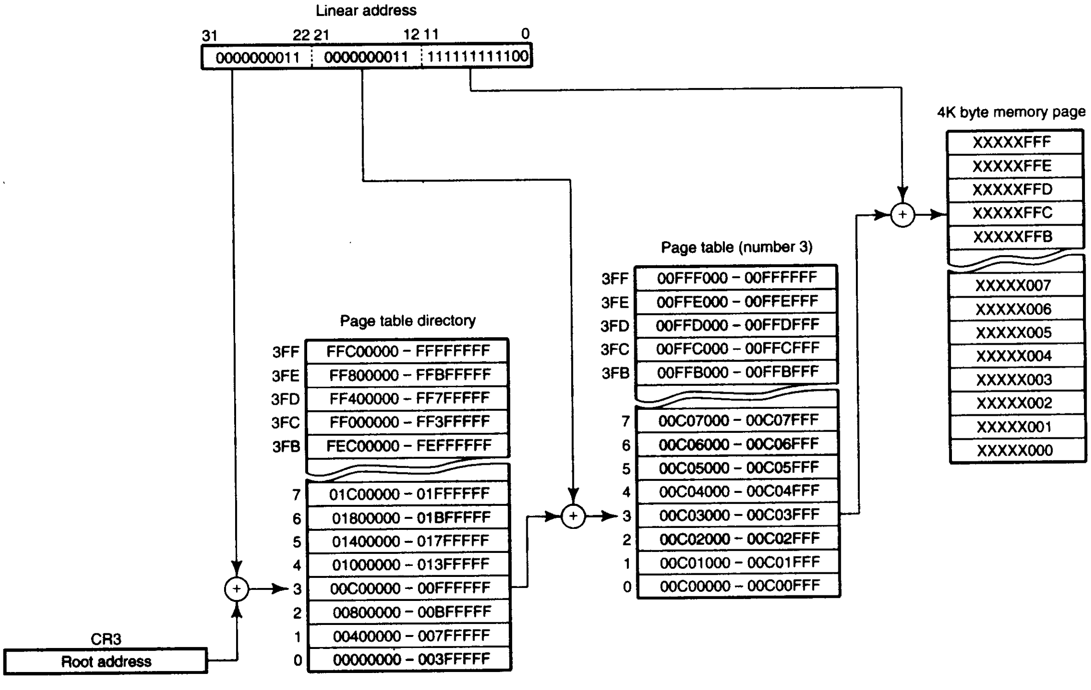
The Page Table

The page table contains 1024 physical page addresses, accessed to translate a linear address into a physical address. Each page table translates a 4M section of the linear memory into 4M of physical memory. The format for the page table entry is the same as for the page directory entry (refer to Figure 15–26). The main difference is that the page directory entry contains the physical address of a page table, while the page table entry contains the physical address of a 4K-byte physical page of memory. The other difference is the D (dirty bit), which has no function in the page directory entry, but indicates that a page has been written to in a page table entry.

Figure 15–27 illustrates the paging mechanism in the 80386 microprocessor. Here, the linear address 00C03FFCH, as generated by a program, is converted to physical address XXXXXFFCH, as translated by the paging mechanism. (Note: XXXXX is any 4K-byte physical page address.) The paging mechanism functions in the following manner:

1. The 4K-byte long page directory is stored as the physical address located by CR3. This address is often called the root address. One page directory exists in a system at a time. In the 8086 virtual mode, each task has its own page directory, allowing different areas of physical memory to be assigned to different 8086 virtual tasks.
2. The upper 10 bits of the linear address (bits 31–22), as determined by the descriptors described earlier in this chapter or by a real address, are applied to the paging mechanism to select an entry in the page directory. This maps the page directory entry to the leftmost 10 bits of the linear address.
3. The page table is addressed by the entry stored in the page directory. This allows up to 4K page tables in a fully-populated and translated system.
4. An entry in the page table is addressed by the next 10 bits of the linear address (bits 21–12).
5. The page table entry contains the actual physical address of the 4K-byte memory page.
6. The rightmost 12 bits of the linear address (bits 11–0) select a location in the memory page.

The paging mechanism allows the physical memory to be assigned to any linear address through the paging mechanism. For example, suppose that linear address 20000000H is selected by a program, but this memory location does not exist in the physical memory system. The 4K-byte linear page is referenced as locations 20000000H–20000FFFH by the program. Because this section of physical memory does not exist, the operating system might assign an existing physical memory page such as 12000000H–12000FFFH to this linear address range.



Note: 1. The address ranges illustrated in the page directory and page table represent the linear address ranges selected and not the contents of these tables.

2. The addresses (XXXXX) listed in the memory page are selected by the page table entry.

FIGURE 15–27 The translation of linear address 00C03FFCH to physical memory address XXXXXFFCH. The value of XXXXX is determined by the page table entry (not shown here).

In the address translation process, the leftmost 10 bits of the linear address select page directory entry 200H located at offset address 800H in the page directory. This page directory entry contains the address of the page table for linear addresses 20000000H–203FFFFFH. Linear address bits (21–12) select an entry in this page table that corresponds to a 4K-byte memory page. For linear addresses 2000000H–20000FFFH, the first entry (entry 0) in the page table is selected. This first entry contains the physical address of the actual memory page, or 12000000H–12000FFFH in this example.

Take, for example, a typical DOS-based computer system. The memory map for the system appears in Figure 15–28. Note from the map that there are unused areas of memory, which can be paged to a different location, giving a DOS real mode application program more memory. The normal DOS memory system begins at location 00000H and extends to location 9FFFFH, which is 640K bytes of memory. Above location 9FFFFH, we find sections devoted to video cards, disk cards, and the system BIOS ROM. In this example, an area of memory just above 9FFFFH is unused (A0000–AFFFFH). This section of the memory could be used by DOS, so that the total application-memory area is 704K instead of 640K. Be careful when using A0000H– AFFFFH for additional RAM because the video card uses this area for bit-mapped graphics in mode 12H and 13H.

This section of memory can be used by mapping it into extended memory at locations 102000H–11FFFFH. Software to accomplish this translation and initialize the page table directory, and page tables required to set up memory are illustrated in Example 15–4. Note that this procedure initializes the page table directory, a page table, and loads CR3. It does not switch to protected mode and it does enable paging. Note that paging functions in real mode memory operation.