CHAPTER 5

80386 Microprocessors

Topics - Introduction to 80386 Microprocessor, The Memory System, Special 80386 Registers, 80386 Memory Management, Virtual 8086 Mode, Introduction to Protected Mode memory Addressing, Memory Paging Mechanism

Text Book Used – The INTEL Microprocessors; Architecture, Programming and Interfacing By Barry B Brey (8th Edition)

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 17–1 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 and 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 17–2 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 was found in many early personal computers that used the same basic motherboard design as the 80286. At the time that the 80386SX was popular, most applications, including Windows 3.11, 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.

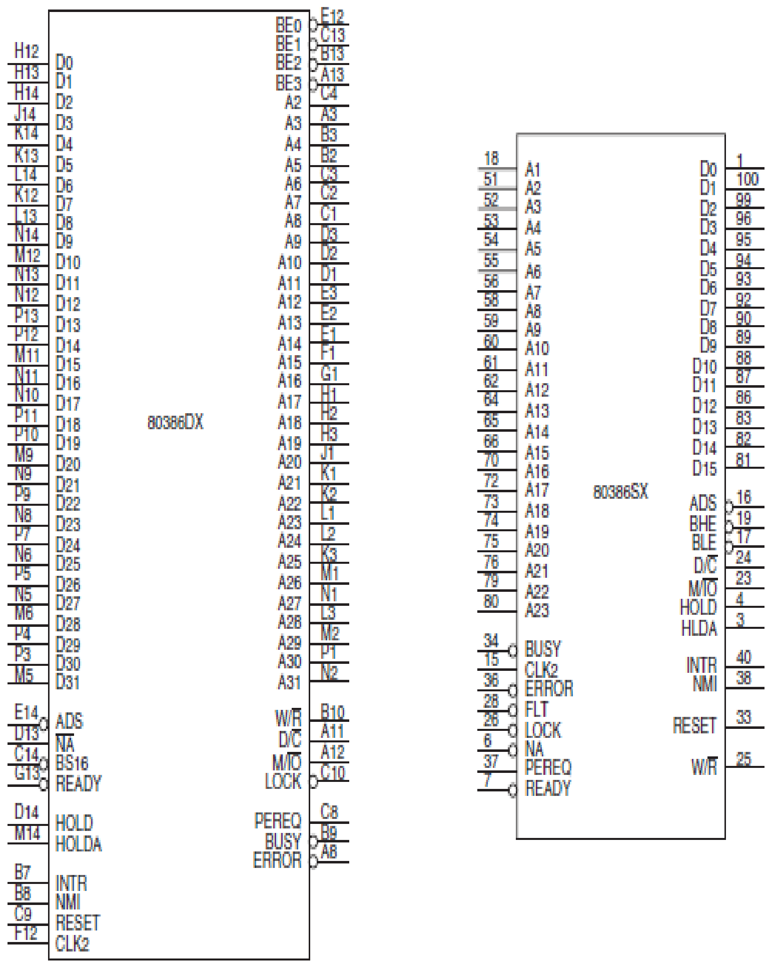


FIGURE 17–1 The pin-outs of the 80386DX and 80386SX microprocessors.

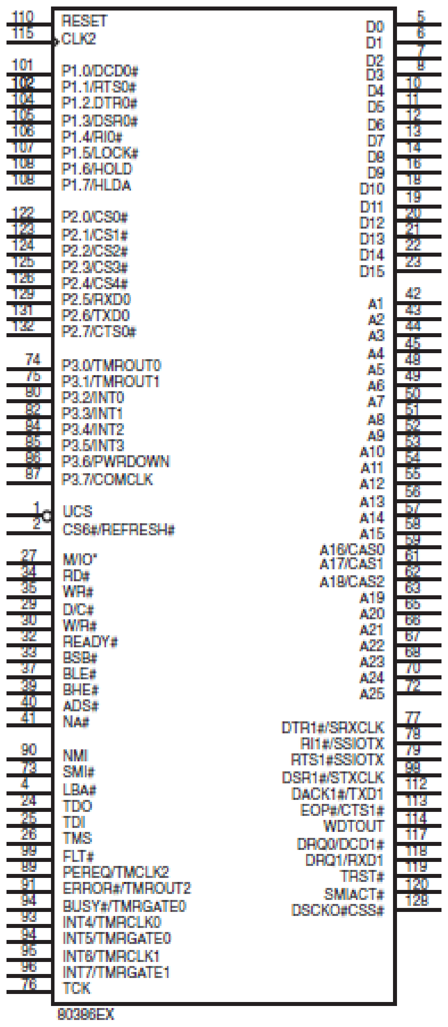


FIGURE 17–2 The 80386EX embedded PC.

The function of each 80386DX group of pins follows:

A_{31}–A_2 Address bus connections address any of the 1G × 32 (4G bytes) memory locations found in the 80386 memory system. Note that A0 and A1 are encoded in the bus enable ($\overline{\mathbf{BE3}}$ - $\overline{\mathbf{BE0}}$) 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 $\overline{\mathbf{BHE}}$ and $\overline{\mathbf{BLE}}$. The $\overline{\mathbf{BHE}}$ signal enables the upper data bus half; the $\overline{\mathbf{BLE}}$ 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.

$\overline{\mathbf{BE3}}$ - $\overline{\mathbf{BE0}}$ **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 $\overline{\mathbf{BHE}}$, $\overline{\mathbf{BLE}}$ and A_1.

M/ $\overline{\mathbf{IO}}$ 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/ $\overline{\mathbf{R}}$ WRITE/ $\overline{\mathbf{READ}}$ indicates that the current bus cycle is a write when a logic 1 or a read when a logic 0.

$\overline{\text{ADS}}$ 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/$\overline{\text{R}}$ 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.

CLK_2 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, apply a 32 MHz clock to this pin.

$\overline{\text{READY}}$ Ready controls the number of wait states inserted into the timing to lengthen memory accesses.

$\overline{\text{LOCK}}$ Lock becomes a logic 0 whenever an instruction is prefixed with the LOCK: prefix. This is used most often during DMA accesses.

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

$\overline{\text{BS16}}$ Bus size 16 selects either a 32-bit data bus ($\overline{\text{BS16}}$ = 1) or a 16-bit data bus ($\overline{\text{BS16}}$ = 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 pin $\overline{\text{BS8}}$ selects an 8-bit data bus.

$\overline{\text{NA}}$ 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.

$\overline{\text{PEREQ}}$ The **coprocessor request** asks the 80386 to relinquish control and is a direct connection to the 80387 arithmetic coprocessor.

$\overline{\text{BUSY}}$ 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.

$\overline{\text{ERROR}}$ 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 large hard disk drive exists.) Figure 17–3 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 be 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. 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 to FFFFFFFFH.

The two memory banks in the 8086, 80286, and 80386SX system are accessed via $\overline{\textbf{BLE}}$ (A0 on the 8086 and 80286) and $\overline{\textbf{BHE}}$. In the 80386DX, the memory banks are accessed via four bank enable signals, $\overline{\textbf{BE3}}$ - $\overline{\textbf{BE0}}$. 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 banks 0 and 1, or in banks 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 $\overline{\textbf{BLE}}$ and $\overline{\textbf{BHE}}$ signals. The 80386EX addresses data either in two banks for a 16-bit-wide memory system if $\overline{\textbf{BS8}}$ = 1 or as an 8-bit system if $\overline{\textbf{BS8}}$ = 0.

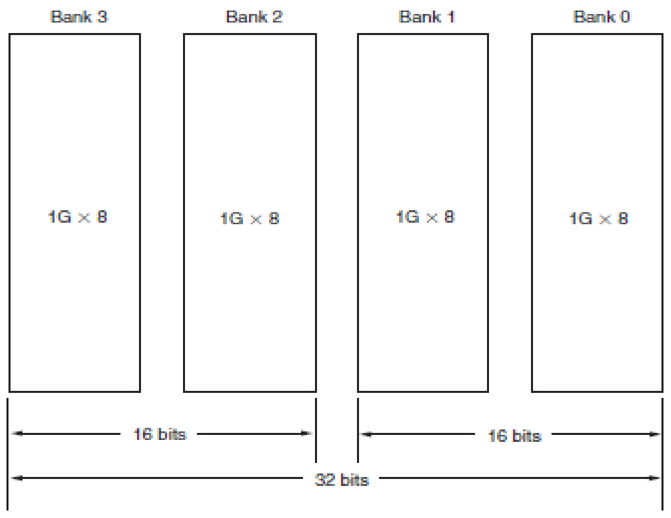


FIGURE 17–3 The memory system for the 80386 micro- processor. Notice that the memory is organized as four banks, each containing 1G byte. Memory is accessed as 8-, 16-, or 32-bit data.

Buffered System. Figure 17–4 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 in use have an access time of 40 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 nonpipelined 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 1 cache and the 80386 can only contain an external cache called a level 2 cache.

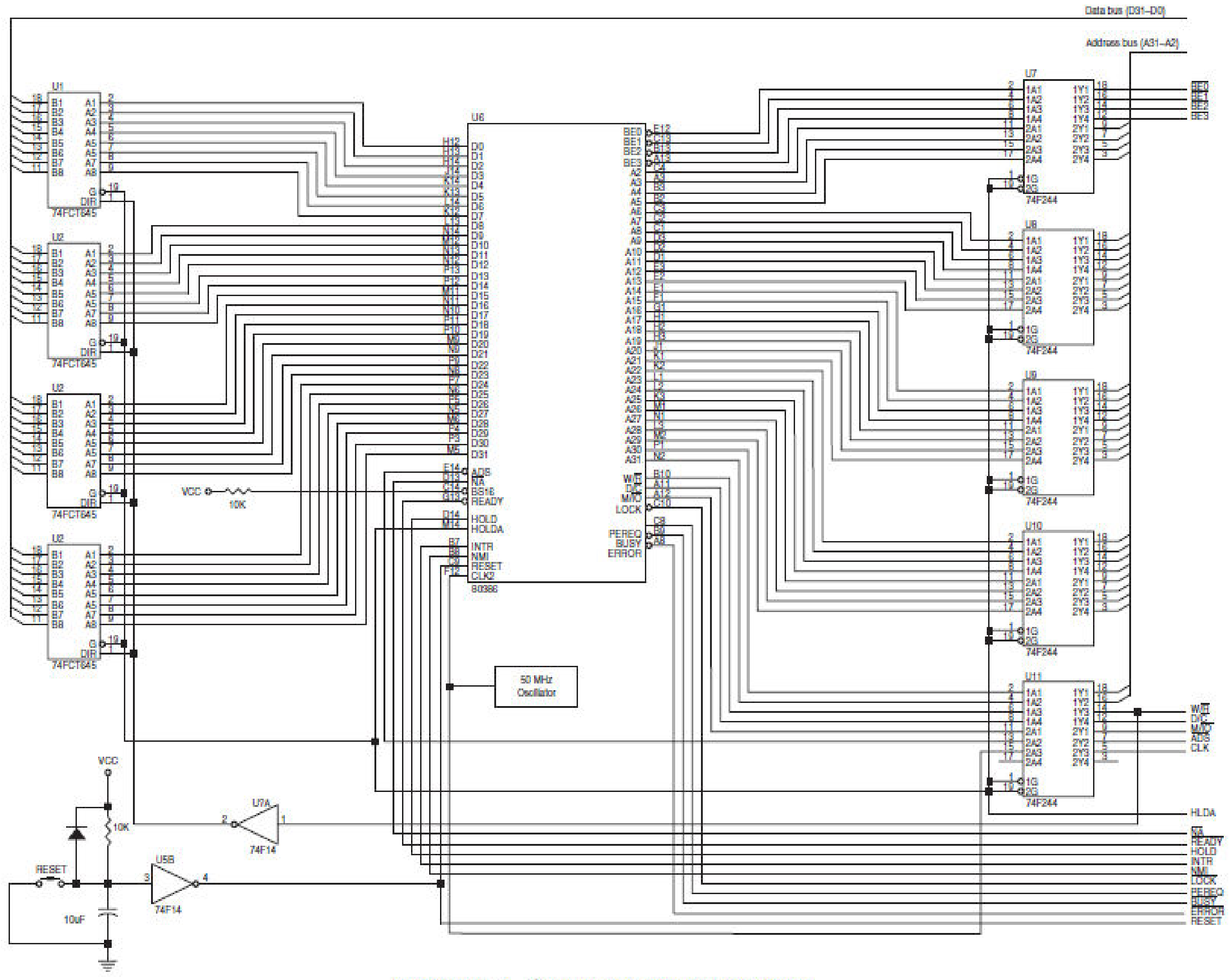


FIGURE 17–4 A fully buffered 25 MHz 80386DX.

A cache is a high-speed memory (SRAM) 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 10 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 64Kbyte 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 section contains the 16-bit addresses 000000H–000001H, 000004H–000005H, and so on; the other section contains addresses 000002–000003, 000006H–000007H, and so forth. 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 increases 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% of the time. For the remaining 7%, the microprocessor addresses data in the same memory section, which means that in these 7% 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 17–5 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 17–6 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 **$\overline{\text{WAIT}}$** 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 **$\overline{\text{WAIT}}$** signal becomes a logic 0, requesting a wait state.

Figure 17–7 illustrates an interleaved memory system that uses the circuit of Figure 17–6. 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 instructions 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.

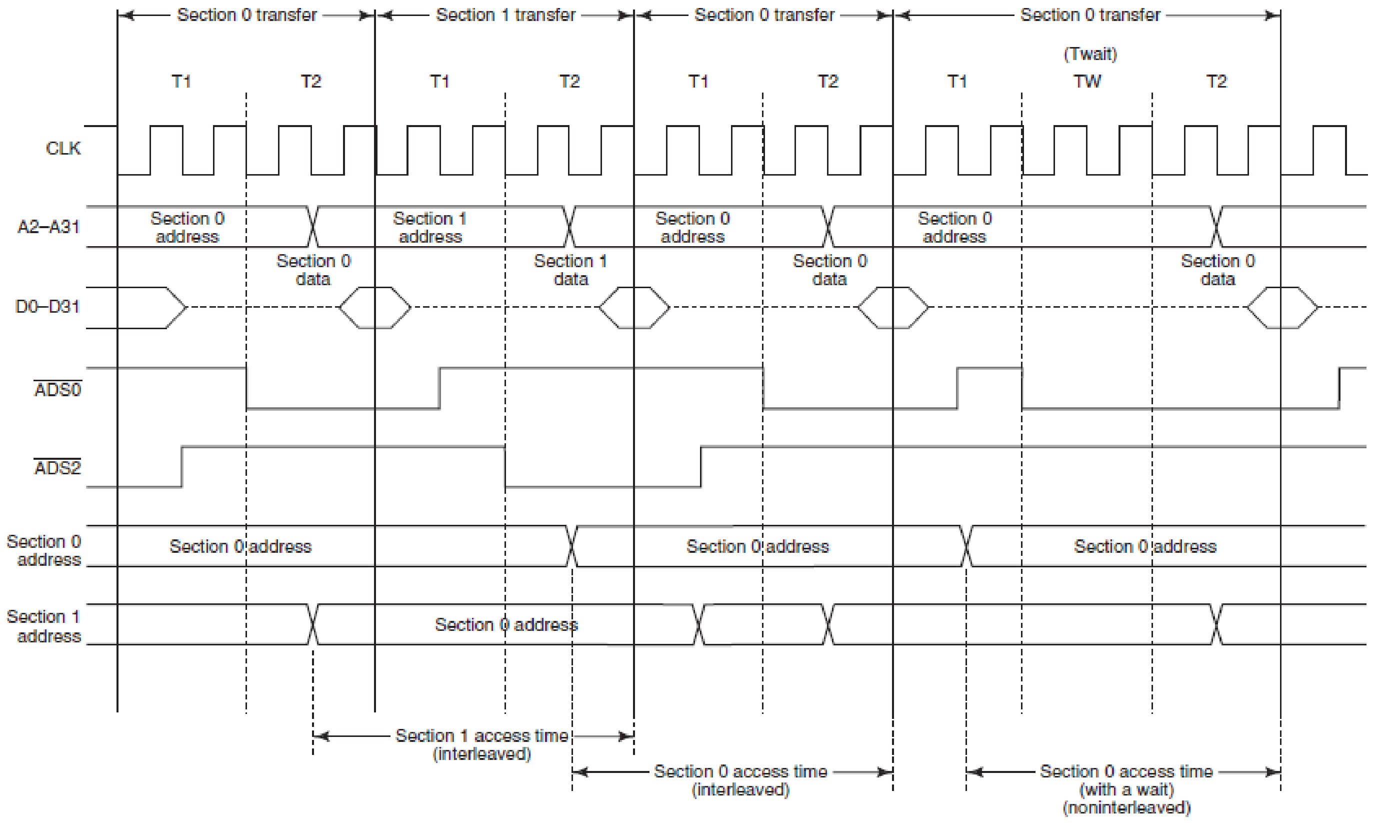


FIGURE 17–5 The timing diagram of an interleaved memory system showing the access times and address signals for both sections of memory.

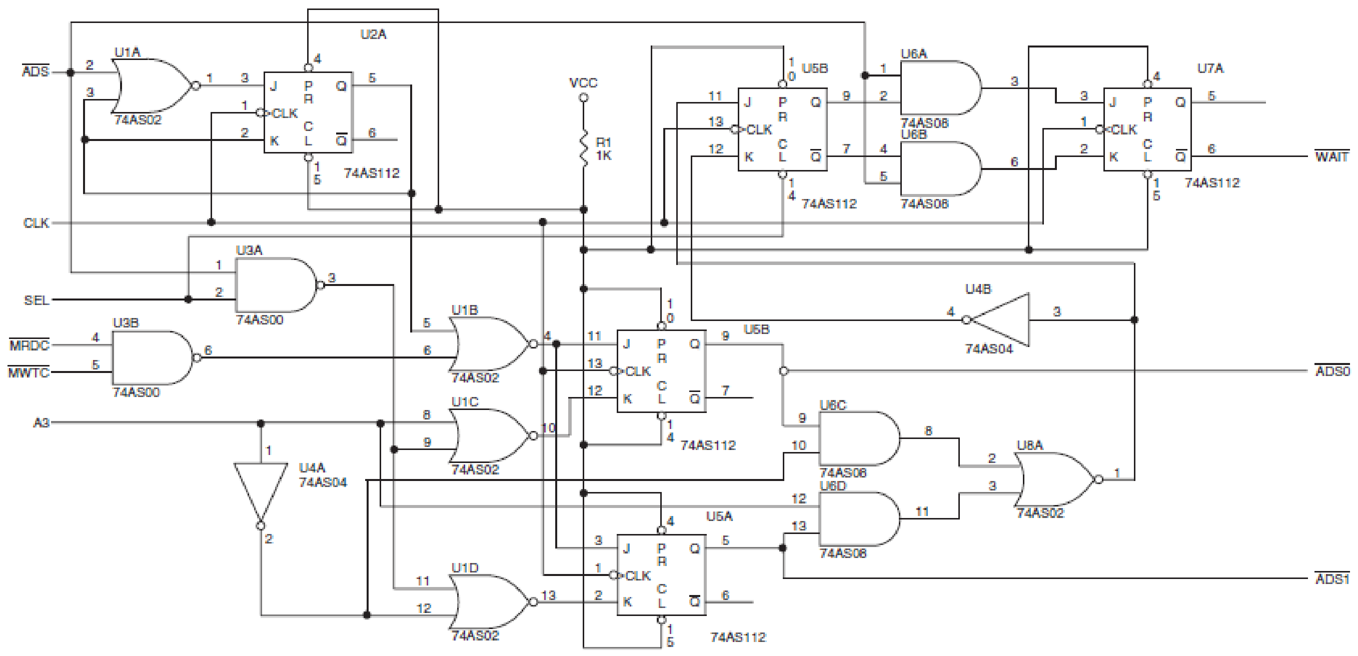


FIGURE 17–6 The interleaved control logic, which generates separate ADS signals and a WAIT signal used to control interleaved memory.

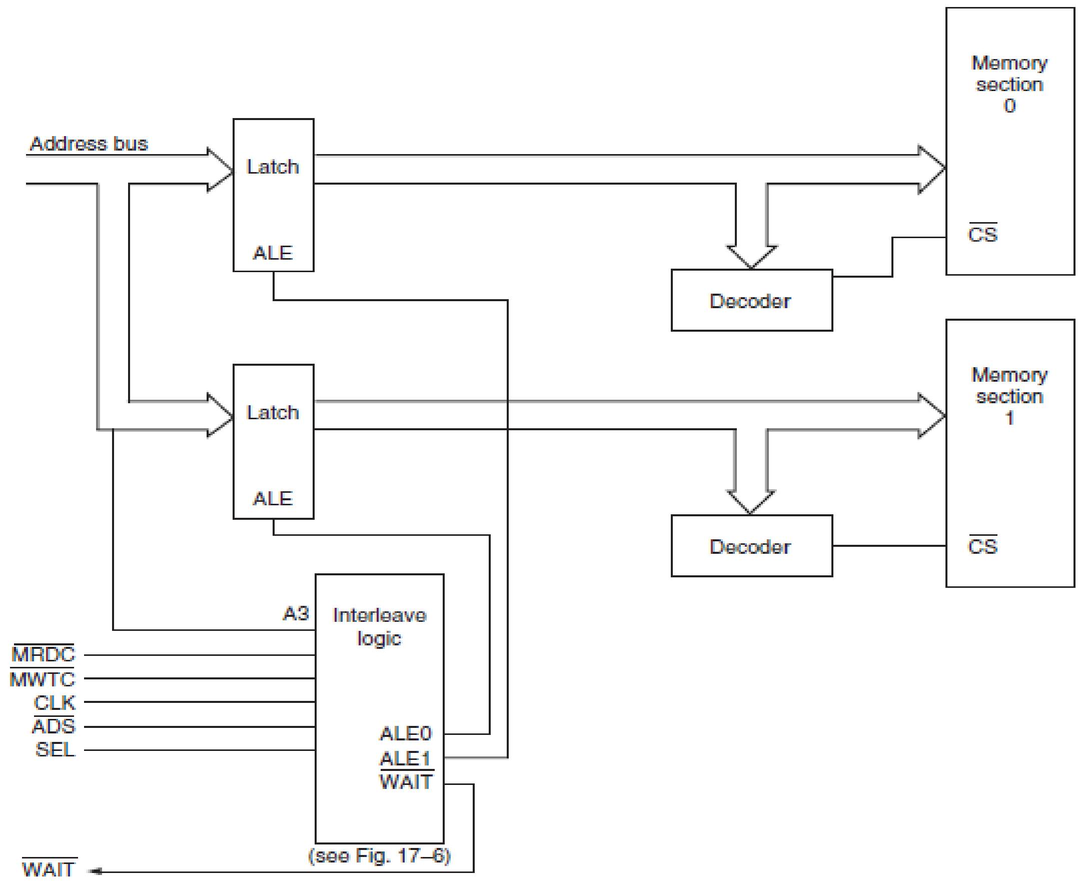


FIGURE 17–7 An interleaved memory system showing the address latches and the interleaved logic circuit.

The access time allowed by an interleaved system, such as the one shown in Figure 17–7, 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, noninterleaved 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, 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 $\overline{\textbf{BE3}}$ - $\overline{\textbf{BE0}}$ 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.

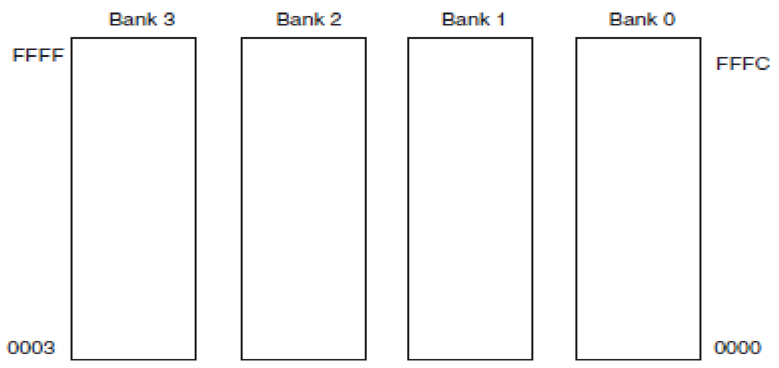


FIGURE 17–8 The isolated I/O map for the 80386 micro- processor. Here four banks of 8 bits each are used to address 64K different I/O locations. I/O is numbered from location 0000H to FFFFH.

Figure 17–8 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 to 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 locations 800000F8H–800000FFH for communications between the 80387 and 80386. The 80287 numeric coprocessor designed for 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/0 Control Signals

The memory and I/O are controlled with separate signals. The **M/$\overline{\text{IO}}$** signal indicates whether the data transfer is between the microprocessor and the memory (**M/$\overline{\text{IO}}$** =1) or I/O (**M/$\overline{\text{IO}}$** =0). In addition to **M/$\overline{\text{IO}}$** , the memory and I/O systems must read or write data. The **W/$\overline{\text{R}}$** signal is a logic 0 for a read operation and a logic 1 for a write operation. The **$\overline{\text{ADS}}$** signal is used to qualify the **M/$\overline{\text{IO}}$** and **W/$\overline{\text{R}}$** control signals. This is a slight deviation from earlier Intel microprocessors, which didn't use **$\overline{\text{ADS}}$** for qualification.
See Figure 17–9 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 (**$\overline{\text{MRDC}}$** and **$\overline{\text{MWTC}}$**) and

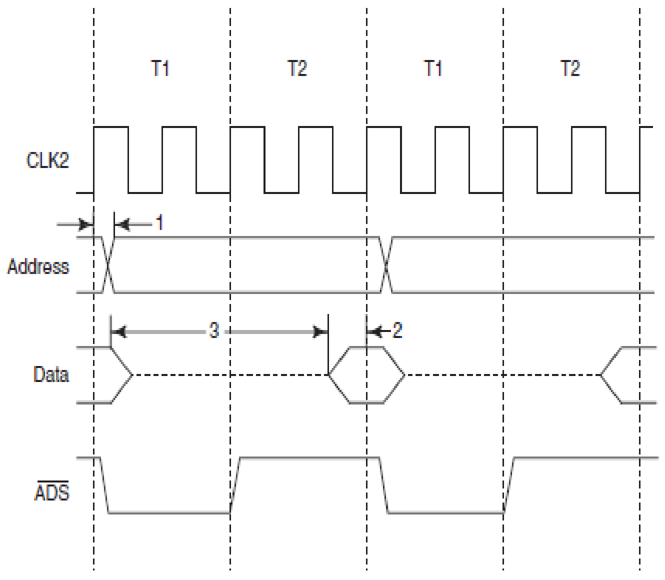
two for I/O control ($\overline{\mathbf{IORC}}$ and $\overline{\mathbf{IOWC}}$). These signals are consistent with the memory and I/O control signals generated for use in earlier versions of the Intel microprocessor.

Timing

Timing is important for understanding how to interface memory and I/O to the 80386 microprocessor.

Figure 17–10 shows the timing diagram of a nonpipelined memory read cycle. Note that the timing is referenced to the CLK2 input signal and that a bus cycle consists of four clocking periods. Each bus cycle contains two clocking states with each state (T1 and T2) containing two clocking periods. Note in Figure 17–10 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 nonpipelined mode of operation. To select the nonpipelined mode, we place a logic 1 on the $\overline{\mathbf{NA}}$ pin. Figure 17–11 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 $\overline{\mathbf{NA}}$ the pin and by using address latches to capture the pipelined address. The clock pulse that is applied to the address latches comes from the $\overline{\mathbf{ADS}}$ 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 nonpipelined addressing. In the 16 MHz version of the 80386, this allows an additional 62.5 ns for memory access. In a nonpipelined 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.



	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 17–10 The non- pipelined read timing for the 80386 microprocessor.

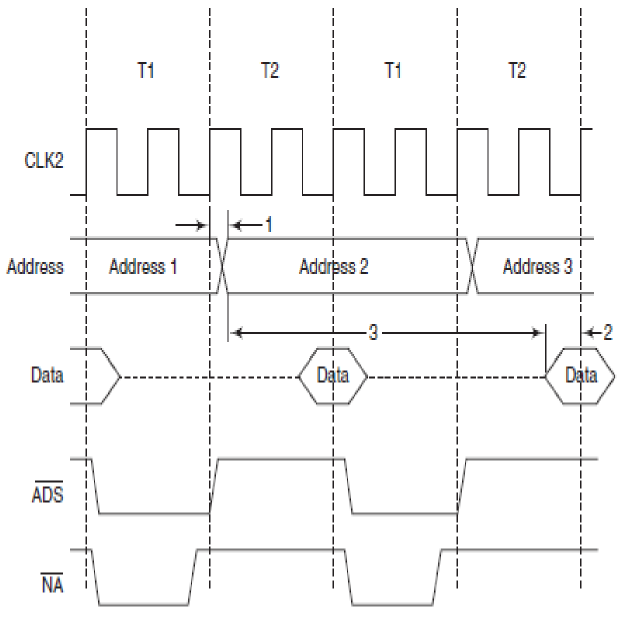


FIGURE 17–11 The pipelined read timing for the 80386 microprocessor.

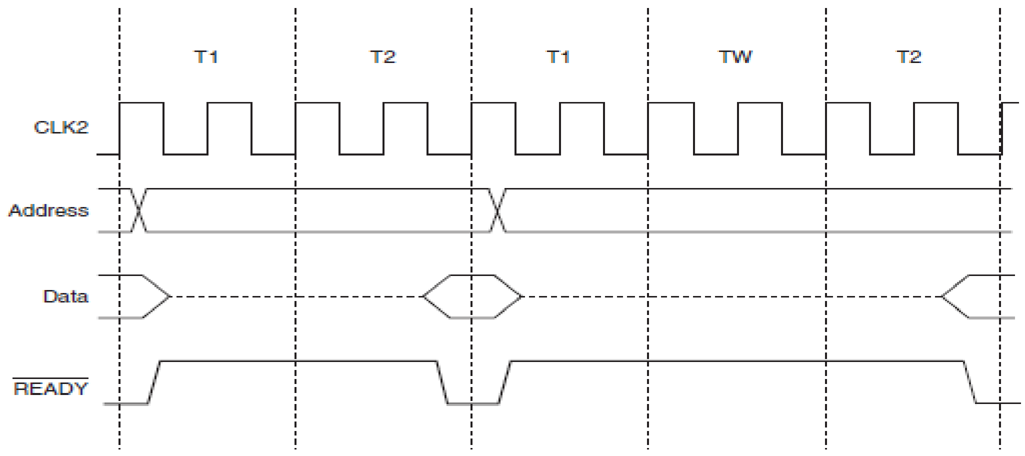


FIGURE 17–12 A nonpipelined 80386 with 0 and 1 wait states.

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 nonpipelined 33 MHz system, memory access time is only 46 ns.
Currently, only a few DRAM memories exist that have an access time of 46 ns. This means that often 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.
The $\overline{\textbf{READY}}$ input controls whether or not wait states are inserted into the timing. The $\overline{\textbf{READY}}$ input on the 80386 is a dynamic input that must be activated during each bus cycle.
Figure 17–12 on the previous page 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.
The $\overline{\textbf{READY}}$ signal is sampled at the end of a bus cycle to determine whether the clock cycle is T2 or TW. If $\overline{\textbf{READY}}$ = 0 at this time, it is the end of the bus cycle or T2. If $\overline{\textbf{READY}}$ is 1 at the end of a clock cycle, the cycle is a TW and the microprocessor continues to test $\overline{\textbf{READY}}$ searching for a logic 0 and the end of the bus cycle.
In the nonpipelined system, whenever $\overline{\textbf{ADS}}$ becomes a logic 0, a wait state is inserted if $\overline{\textbf{READY}}$ =1. After $\overline{\textbf{ADS}}$ returns to a logic 1, the positive edges of the clock are counted to generate the $\overline{\textbf{READY}}$ signal. The $\overline{\textbf{READY}}$ signal becomes a logic 0 after the first clock to insert 0 wait states. If one wait state is inserted, the $\overline{\textbf{READY}}$ 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 $\overline{\textbf{READY}}$ is cleared. This essentially allows any number of wait states to be inserted into the timing. Figure 17–13 on the previous page shows a circuit that inserts 0 through 3 wait states for various memory addresses. In the example, one wait state is produced for a DRAM access and two wait states for an EPROM access. The 74F164 clears whenever $\overline{\textbf{ADS}}$ is low and D/ $\overline{\textbf{C}}$ is high. It begins to shift after $\overline{\textbf{ADS}}$ 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 $\overline{\textbf{READY}}$ signal.

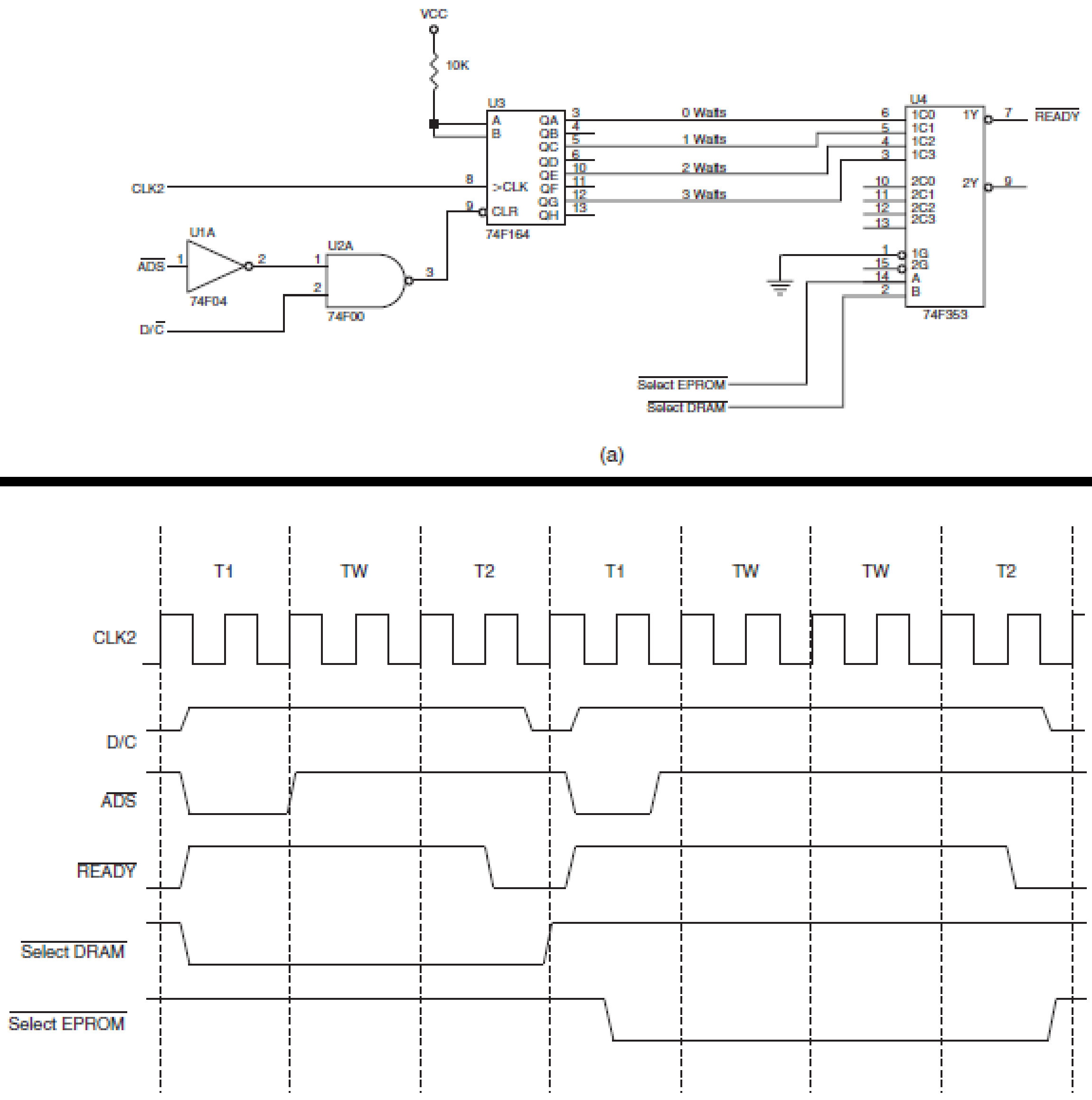


FIGURE 17–13 (a) Circuit and (b) timing that selects 1 wait state for DRAM and 2 waits for EPROM.

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 17–14 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 4Klong page table.

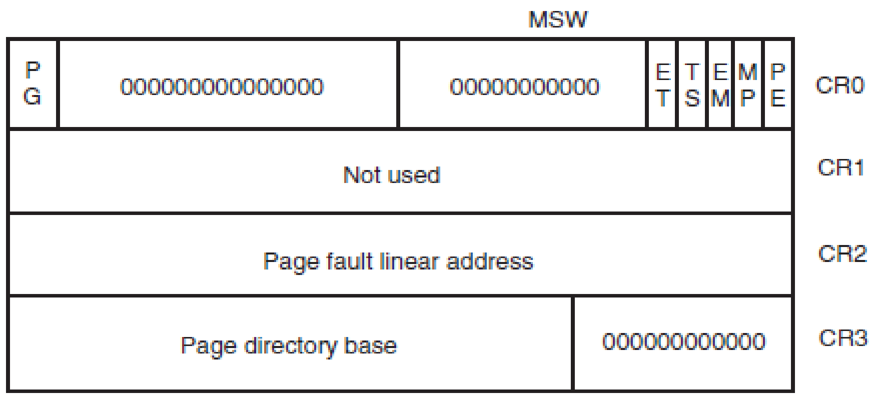


FIGURE 17–14 The control register structure of the 80386 microprocessor.

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 = l. 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.

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 The **emulate** bit is set to cause a type 7 interrupt for each ESC instruction.
(ESCape instructions are used to encode instructions for the 80387 coprocessor.)
Once this feature was used to emulate interrupts 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 reenter 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 17–15 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 microprocessors 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.

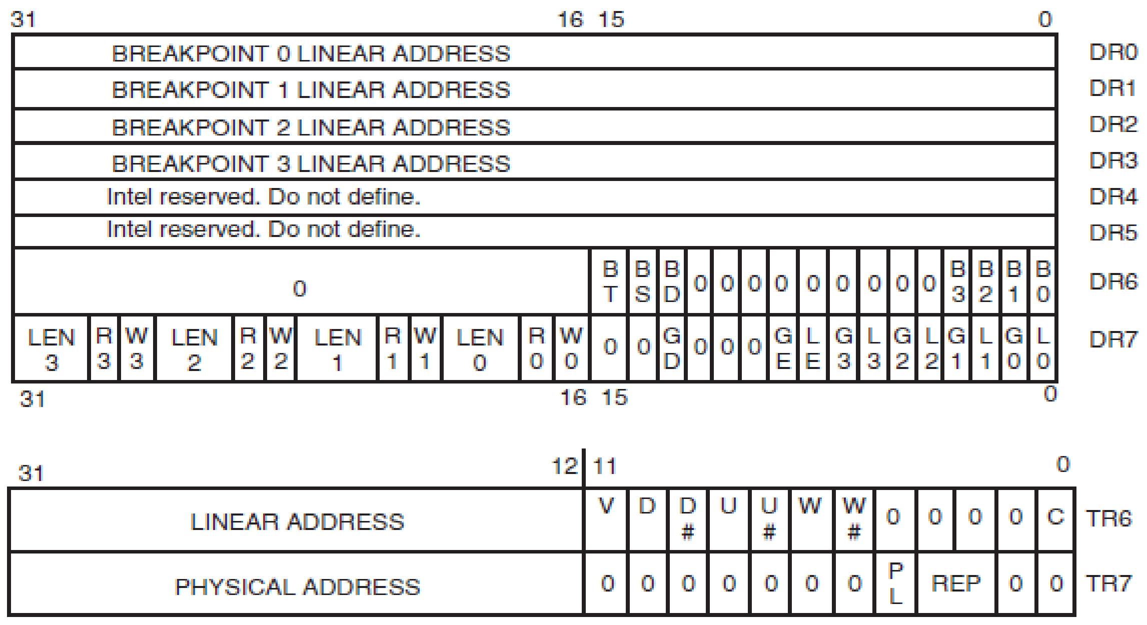
LEN Each of the four length fields pertains to each of the four breakpoint addresses stored in DR0–DR3. These bits further define the size of access at the breakpoint address as 00 (byte), 01 (word), or 11 (doubleword).

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.



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 17–15 The debug and test registers of the 80386. (Courtesy of Intel Corporation.)

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, which holds the most commonly used page table address translations, 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.

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 describes and locates 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 byte 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 17–16 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 uses its 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, 16,384 segments can be accessed 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 other form of large volume storage.

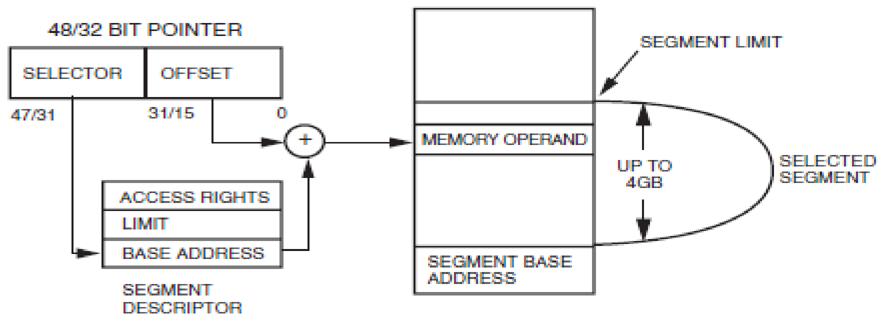


FIGURE 17–16 Protected mode addressing using a segment register as a selector. (Courtesy of Intel Corporation.)

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, which allows 80286 software to be upward compatible with the 80386. (An 80286 descriptor used 00H for its most significant two bytes.) See Figure 17–17 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:

Limit (L19–L0) Defines the starting 32-bit address of the segment within the 4G-byte physical address space of the 80386 microprocessor. 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 byte 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 instruction mode. If D = 0, the registers and memory pointers 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 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, and most Windows programs assume D = 1.

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.

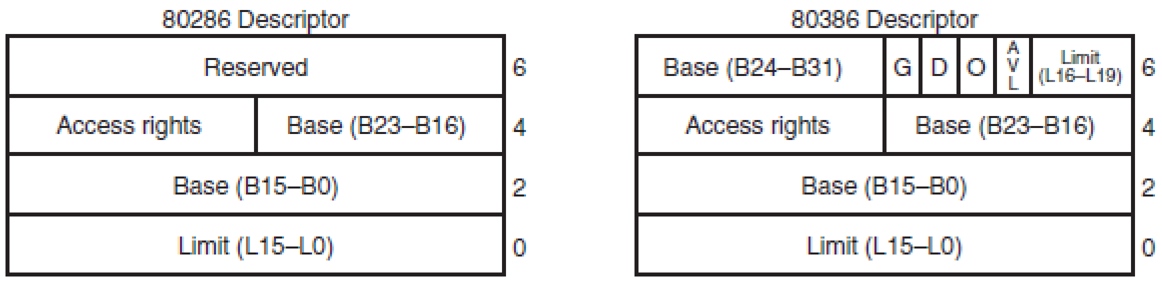


FIGURE 17–17 The descriptors for the 80286 and 80386 microprocessors.

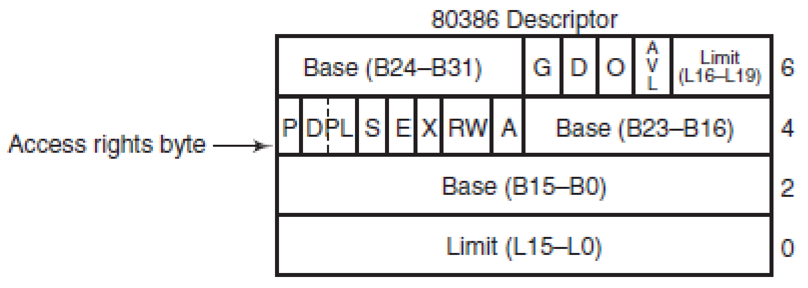


FIGURE 17–18 The format of the 80386 segment descriptor.

Segment Descriptors. Figure 17–18 shows the segment descriptor. This descriptor fits the general form, as dictated in Figure 17–17, 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; 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 the **read/write** bit (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).

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.

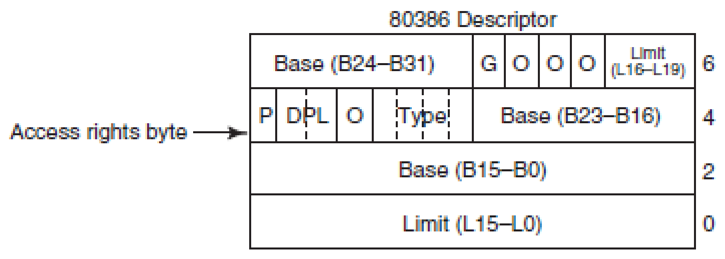


FIGURE 17–19 The general format of an 80386 system descriptor.

Type	*Purpose*
0000	Invalid
0001	Available 80286 TSS
0010	LDT
0011	Busy 80286 TSS
0100	80286 task gate
0101	Task gate (80386 and above)
0110	80286 interrupt gate
0111	80286 trap gate
1000	Invalid
1001	Available 80386 and above TSS
1010	Reserved
1011	Busy 80386 and above TSS
1100	80386 and above CALL gate
1101	Reserved
1110	80386 and above interrupt gate
1111	80386 and above trap gate

TABLE 17–1 System descriptor types.

System Descriptor. The system descriptor is illustrated in Figure 17–19. There are 16 possible system descriptor types (see Table 17–1 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.

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 17–20 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 = 1) 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. 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 17–21 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 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.

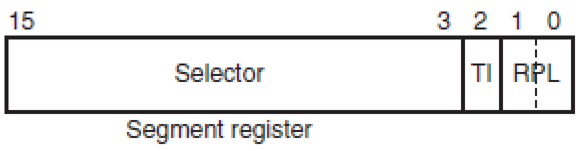


FIGURE 17–20 A segment register showing the selector, T_1 bit, and requested privilege level (RPL) bits.

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 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.

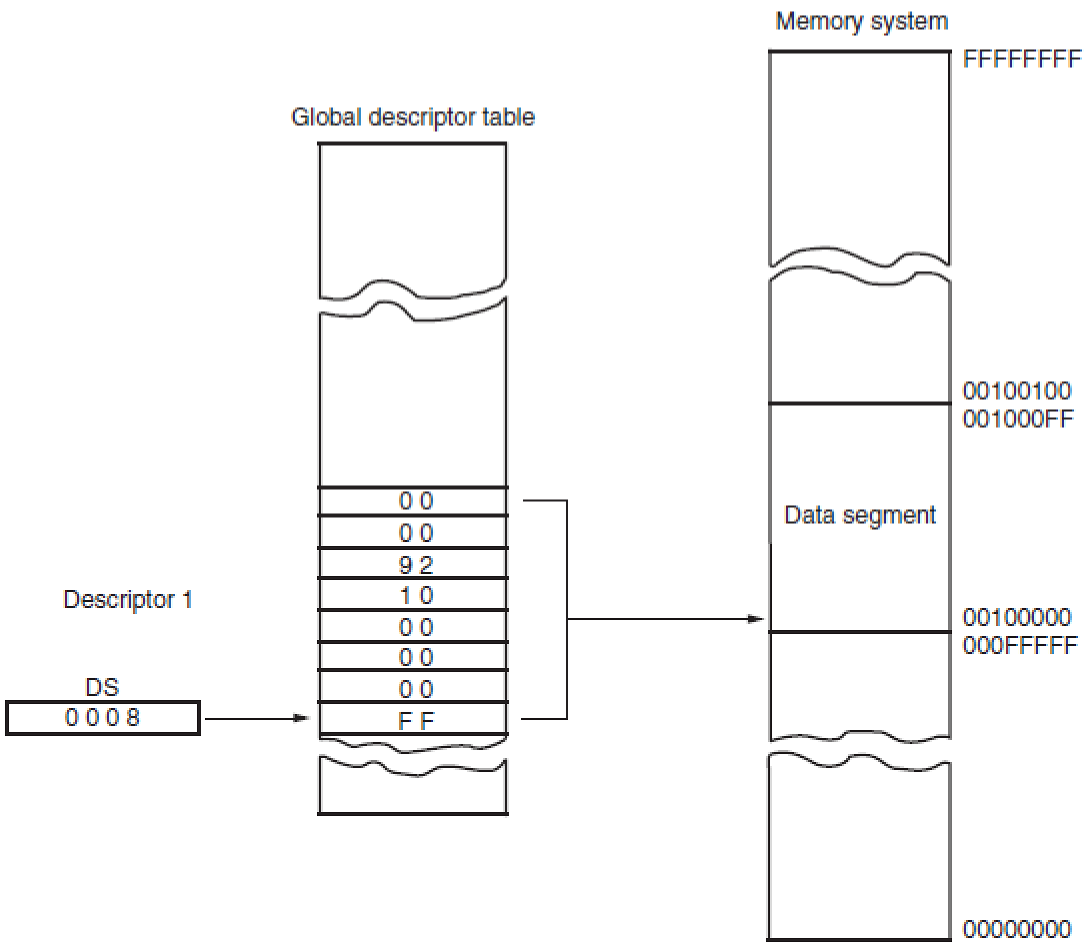


FIGURE 17–21 Using the DS register to select a descriptor from the global descriptor table. In this example, the DS register accesses memory locations 00100000H–001000FFH as a data segment.

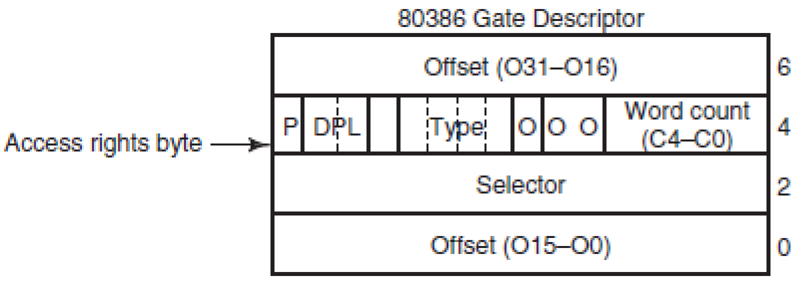


FIGURE 17–22 The gate descriptor for the 80386 microprocessor.

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 17–22 shows the gate descriptor, a special form of the system descriptor described earlier. (Refer to Table 17–1 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 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 (INT 2) is located at IDT descriptor number 2 at 16 locations above the base address of the IDT. This also means that the first IK 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 like any other descriptor. The difference is that the TSS described by the TSS descriptor does not contain 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 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 17–23. As can be seen, the TSS is quite a formidable data structure, containing many different types of information. The first word of the TSS is labeled **back-link.** This is the selector 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.

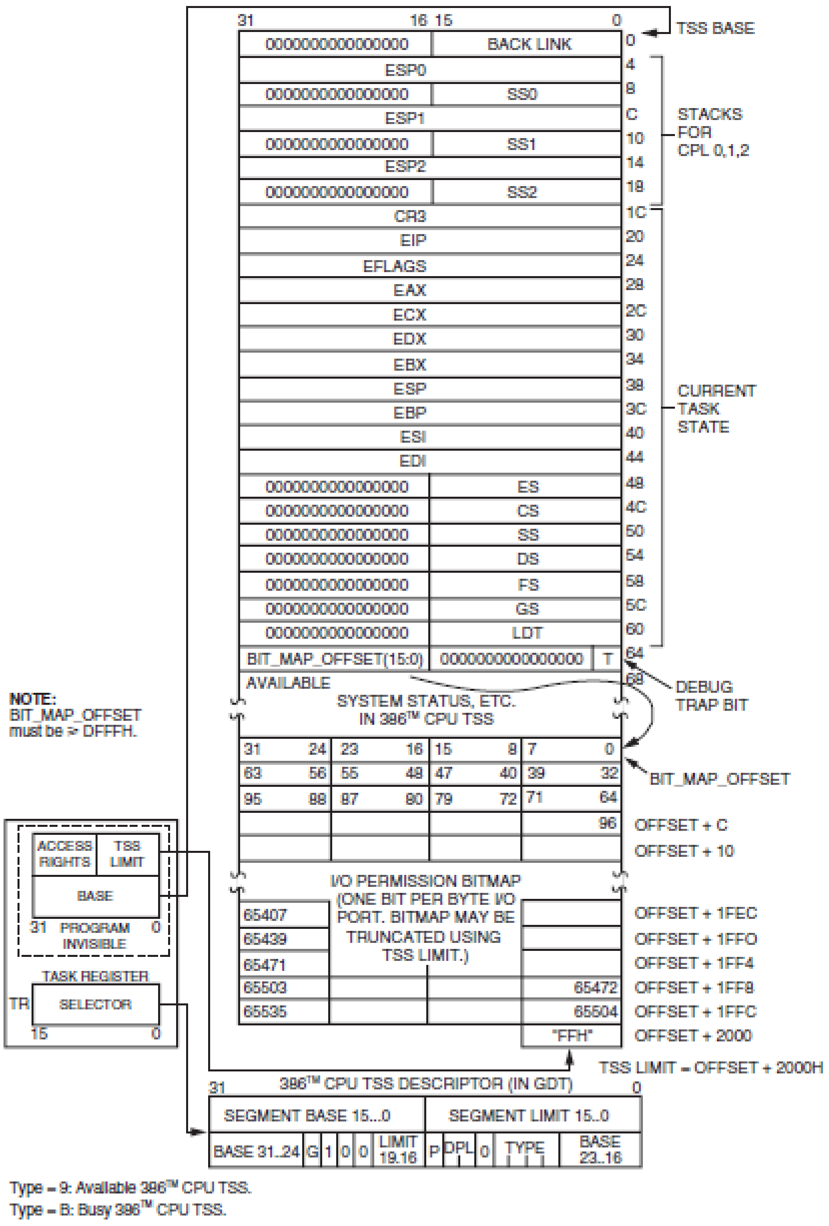


FIGURE 17–23 The task state segment (TSS) descriptor. (Courtesy of Intel Corporation.)

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, Windows 2000, and Windows XP uses the I/O permission scheme to disable I/O ports dependent on the application or the user.

To review the operation of a task switch, which requires only 17 μs to execute on an 80386 microprocessor, 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.

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 using the DOS emulator cmd.exe (the command prompt). Figure 17–25 illustrates two 8086 applications mapped into the 80386 using the virtual mode. The operating system allows multiple applications to execute, 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 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’s execution time.

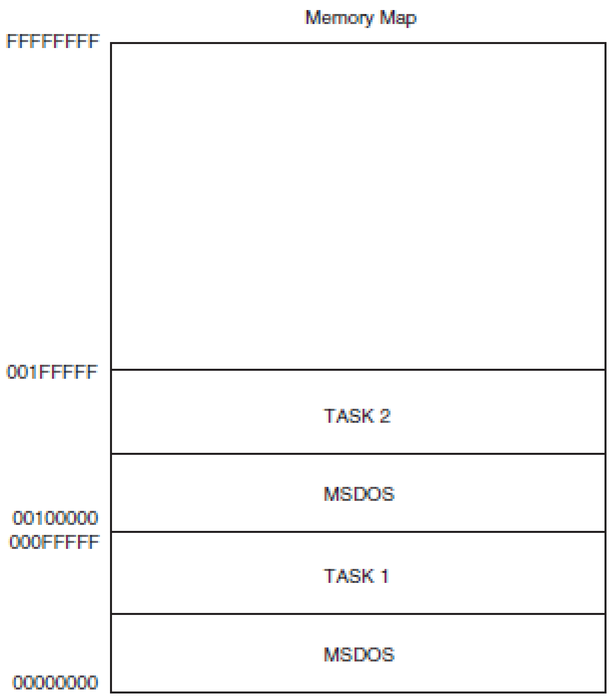


FIGURE 17–25 Two tasks resident in an 80386 operated in the virtual 8086 mode.

A system that can use this technique is a print spooler. The print spooler can function in one DOS partition and be accessed 10% of the time. This allows the system to print using the print spooler, but it doesn't detract from the system because it uses only 10% 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 locations 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–01FFFFFFH, 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.

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 eight-byte interrupt gates defining all 256 interrupt types.
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 intersegment (far) JMP to flush the internal instruction queue.
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 17–24 shows the protected system memory map set up by following steps 1–6. The software for this task is listed in Example 17–1. 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 (called the **flat model**): 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 does not 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 17–1 is designed for a stand-alone system such as the 80386EX embedded microprocessor, and not for use in the PC.)

Example 17–1 does not store any interrupt vectors into the interrupt descriptor table, because none are used in the example. If interrupt vectors are used, then software must be included to load the addresses of the interrupt service procedures into the IDT. The software must be generated as two separate parts that are then connected together and burned on a ROM.

The first part is written as shown in the real mode and the second part (see comment in listing) with the assembler set to generate protected mode code using the 32-bit flat model. This software will not function on a personal computer because it is written to function on an embedded system. The code must be converted to a binary file using EXE2BIN after it is assembled and before burning to a ROM.

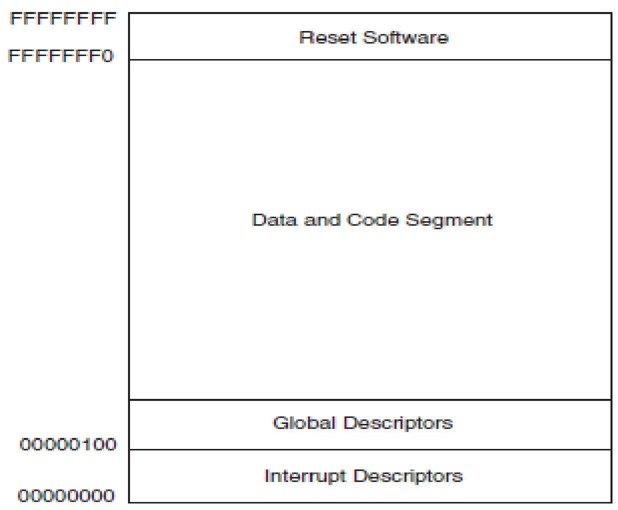


FIGURE 17–24 The memory map for Example 17–1.

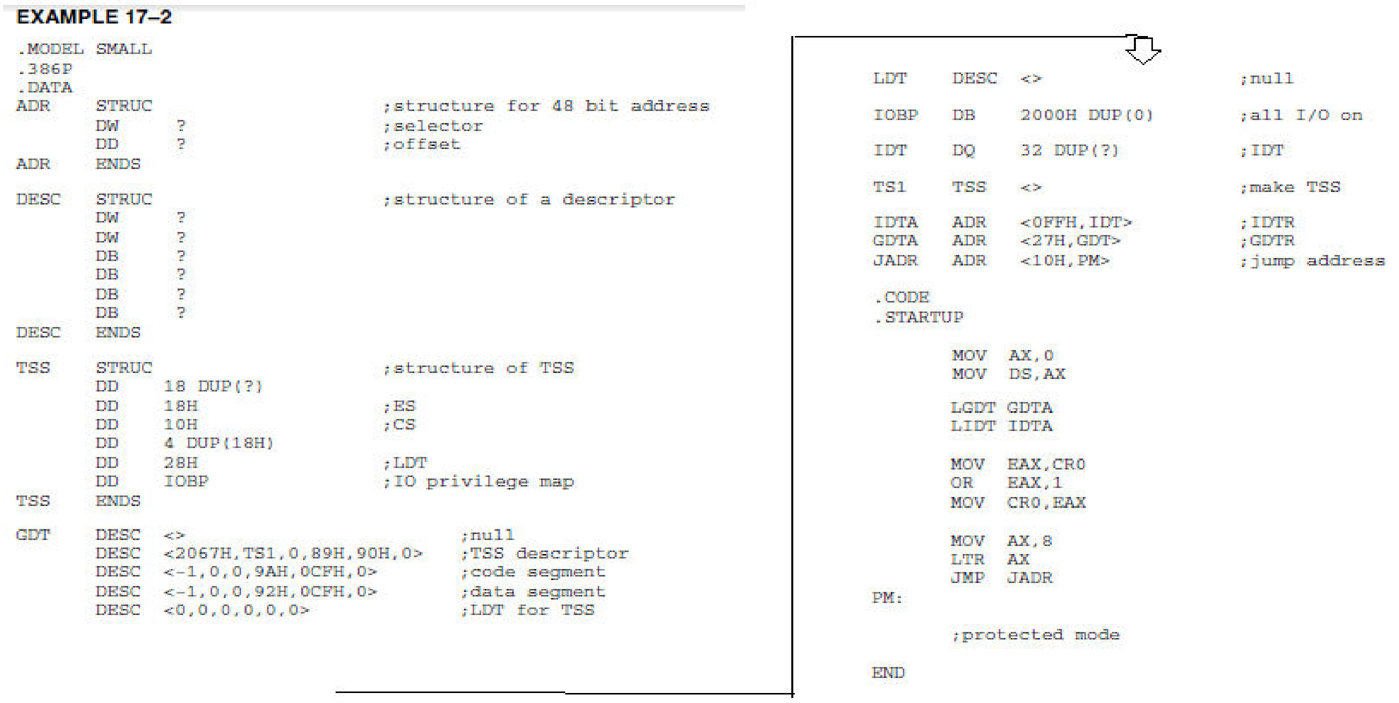
EXAMPLE 17–1

```
.MODEL SMALL
.386P
ADR       STRUC
          DW      ?              ;address structure
          DD      ?
ADR       ENDS

.DATA
IDT     DQ      32 DUP(?)        ;interrupt descriptor table
GDT     DQ      8                ;global descriptor table
DESC1   DW      0FFFFH           ;code segment descriptor
        DW      0
        DW      0
        DW      9EH
        DW      8FH
        DW      0
DESC2   DW      0FFFFH           ;data segment descriptor
        DW      0
        DW      0
        DW      92H
        DW      8FH
        DW      0
IDTR      ADR      <0FFH,IDT>    ;IDTR data
GDTR      ADR      <17H,GDT>     ;GDTR data
JADR      ADR      <8,PM>        ;far JMP data
.CODE
.STARTUP
          MOV   AX,0             ;initialize DS
          MOV   DS,AX

          LIDT  IDTR             ;initialize IDTR
          LGDT  GDTR             ;initialize GDTR

          MOV   EAX,CR0          ;set PE
          OR    EAX,1
          MOV   CR0,EAX

          JMP   JADR             ;far jump to PM
PM::                             ;force a far label

;the software that follows must be developed separately
;with the assembler set to generate 32-bit protected mode code
;ie:       .MODEL FLAT

          MOV   AX,10H           ;load segment registers
          MOV   DS,AX            ;now in protected mode
          MOV   ES,AX
          MOV   SS,AX
          MOV   FS,AX
          MOV   GS,AX
          MOV   SP,0FFFF000H

;other initialization appears here

end
```

In more complex systems (very unlikely to appear in embedded 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 a task state segment (TSS) descriptor, and the initial code and data segments required for the initial task.
3. Initialize the task register (TR) so that it points to a TSS.
4. Switch to protected mode by using an intersegment (far) jump to flush the internal instruction queue. This loads the TR with the current TSS selector and initial task.
5. The 80386 is now operating in the protected mode under control of the first task.

Example 17–2 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. As with Example 17–2 this software will not function on a personal computer and is designed to function only on an embedded system.



EXAMPLE 17–2

```
.MODEL SMALL
.386P
.DATA
ADR     STRUC                         ;structure for 48 bit address
        DW      ?                     ;selector
        DD      ?                     ;offset
ADR     ENDS

DESC    STRUC                         ;structure of a descriptor
        DW      ?
        DW      ?
        DB      ?
        DB      ?
        DB      ?
        DB      ?
DESC    ENDS

TSS     STRUC                         ;structure of TSS
        DD      18 DUP(?)
        DD      18H                   ;ES
        DD      10H                   ;CS
        DD      4 DUP(18H)
        DD      28H                   ;LDT
        DD      IOBP                  ;IO privilege map
TSS     ENDS

GDT     DESC    <>                              ;null
        DESC    <2067H,TS1,0,89H,90H,0>         ;TSS descriptor
        DESC    <-1,0,0,9AH,0CFH,0>             ;code segment
        DESC    <-1,0,0,92H,0CFH,0>             ;data segment
        DESC    <0,0,0,0,0,0>                   ;LDT for TSS

LDT     DESC    <>                    ;null

IOBP    DB      2000H DUP(0)          ;all I/O on

IDT     DQ      32 DUP(?)             ;IDT

TS1     TSS     <>                    ;make TSS

IDTA    ADR     <0FFH,IDT>            ;IDTR
GDTA    ADR     <27H,GDT>             ;GDTR
JADR    ADR     <10H,PM>              ;jump address

.CODE
.STARTUP
        MOV     AX,0
        MOV     DS,AX

        LGDT    GDTA
        LIDT    IDTA

        MOV     EAX,CR0
        OR      EAX,1
        MOV     CR0,EAX

        MOV     AX,8
        LTR     AX
        JMP     JADR
PM:

        ;protected mode

END
```

Neither Example 17–1 nor Example 17–2 is written to function in the personal computer environment. The personal computer environment requires the use of either the VCPI (virtual control program interface) driver provided by the HIMEM.SYS driver in DOS or the DPMI (**DOS protected mode interface**) driver provided by Windows when shelling to DOS.

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 17–14). 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 17–26) 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 17–26, each perform the following functions:

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 Read/write and **user/supervisor** are both used in the protection scheme, as listed in Table 17–2. Both bits combine to develop paging priority level protection 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.

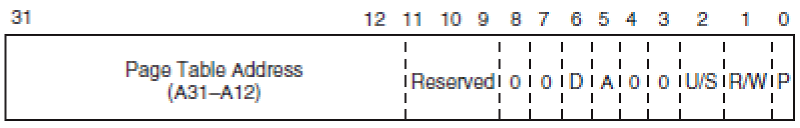


FIGURE 17–26 The page table directory entry.

TABLE 17–2 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	Write-only

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 17–26). 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 17–27 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 4Kbyte 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.

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 20000000H–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.

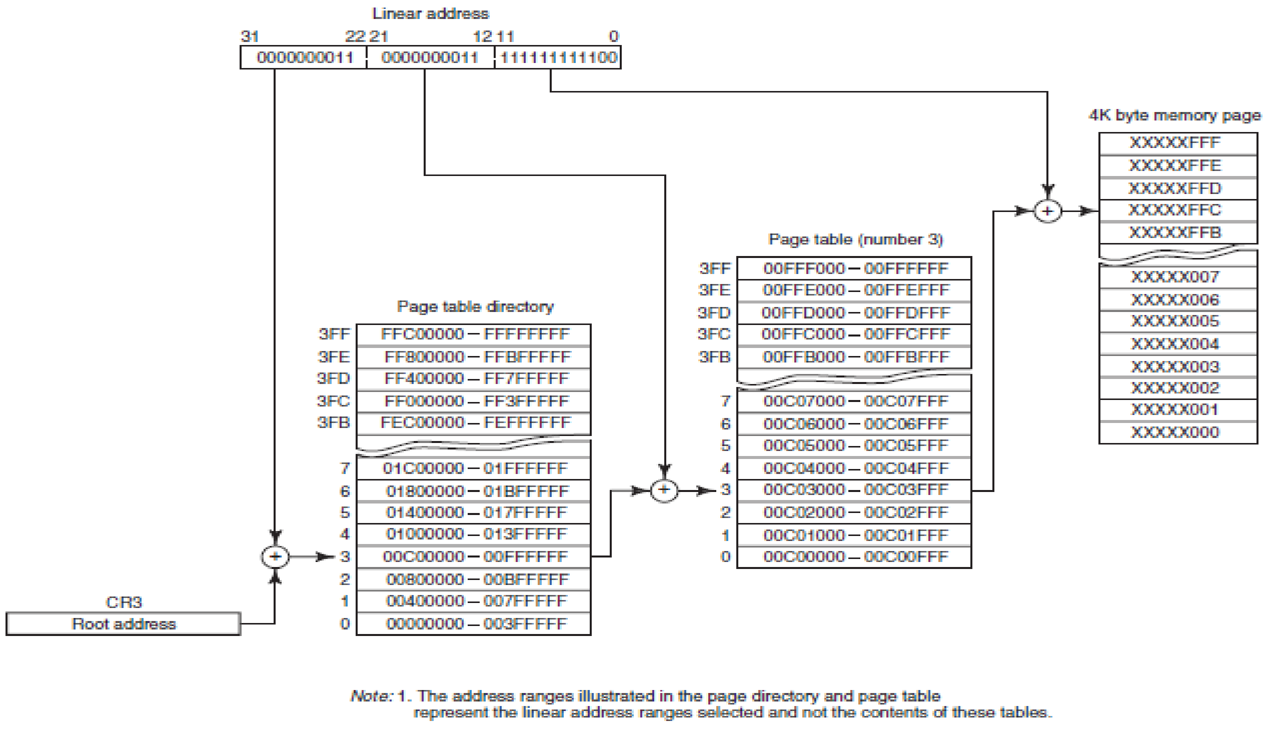


FIGURE 17–27 The translation of linear address 00C03FFCH to physical memory address XXXXXFFCH. The value of XXXXX is determined by the page table entry (not shown here).

Take, for example, a typical DOS-based computer system. The memory map for the system appears in Figure 17–28. 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 1002000H–11FFFFH. Software to accomplish this translation and initialize the page table directory, and page tables required to set up memory, are illustrated in Example 17–4. 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.

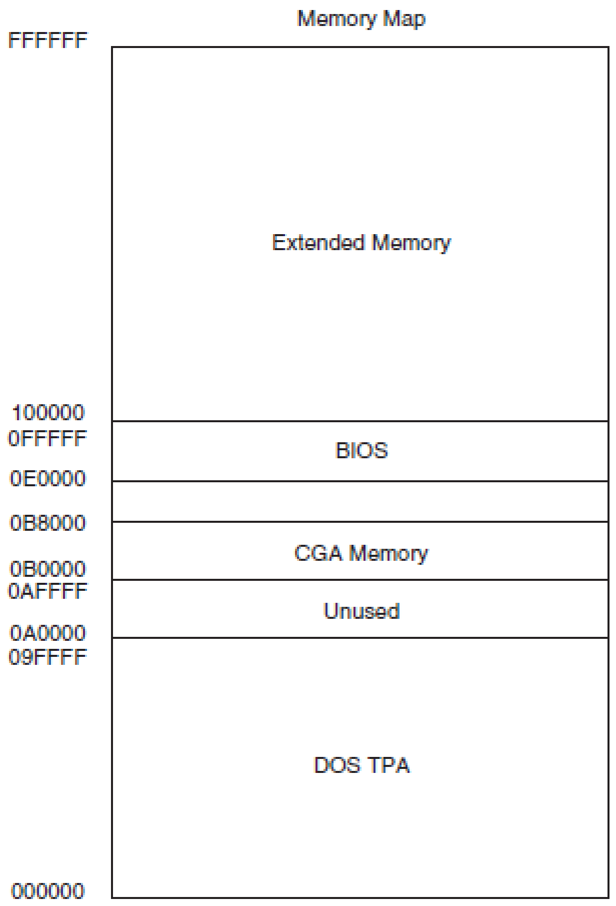


FIGURE 17–28 Memory map for an AT-style clone.

EXAMPLE 17–4

```
.MODEL SMALL
.386P
.DATA

;page directory

PDIR   DD     4

;page table 0

TAB0   DD     1024 dup(?)

.CODE
.STARTUP
        MOV  EAX,0
        MOV  AX,CS
        SHL  EAX,4
        ADD  EAX,OFFSET TAB0
        AND  EAX,0FFFFF000H
        ADD  EAX,7
        MOV  PDIR,EAX                ;address page directory
        MOV  ECX,256
        MOV  EDI,OFFSET TAB0
        MOV  AX,DS
        MOV  ES,AX
        MOV  EAX,7
        .REPEAT                      ;remap 00000H-9FFFFH
                STOSD                ;to 00000H-9FFFFH
                ADD  EAX,4096
        .UNTILCXZ
        MOV  EAX,102007H
        MOV  ECX,16
        .REPEAT                      ;remap A0000H-AFFFFH
                STOSD                ;to 102000H-11FFFFH
                ADD  EAX,4096

       .UNTILCXZ
       MOV  EAX,0
       MOV  AX,DS
       SHL  EAX,4
       ADD  EAX,OFFSET PDIR         ;load CR3
       MOV  CR3,EAX

;additional software to remap other areas of memory

       END
```