CHAPTER 15

The 80286, 80386, and 80486 Microprocessors

For most of the examples up to this point in the book, we have used the 8086/8088 microprocessor, because it is the simplest member of this family of Intel processors and is therefore a good starting point. Now it is time to look at the evolutionary offspring of the 8086. To give you an overview, here are a few brief notes about the members of this family.

The 80186 processor is basically an 8086 with an on-chip priority-interrupt controller, programmable timer, DMA controller, and address decoding circuitry. This processor has been used mostly in industrial control applications.

The 80286, another 16-bit enhancement of the 8086, was introduced at the same time as the 80186. Instead of the integrated peripherals of the 80186, it has virtual memory-management circuitry, protection circuitry, and a 16-Mbyte addressing capability. The 80286 was the first family member designed specifically for use as the CPU in a multiuser microcomputer.

The 80386, the next evolutionary step in the family, is a 32-bit processor with a 32-bit address bus. The 32-bit ALU allows the 80386 to process data faster, and the 32-bit address bus allows the 80386 to address up to 4 Gbytes of memory. Another enhancement of the 80386 is that segments can be as large as 4 Gbytes instead of only 64 Kbytes. The memory-management circuitry and protection circuitry in the 80386 are improved over that in the 80286, so the 80386 is much more versatile as the CPU in a multiuser system.

The latest current member of this family, the 80486, has the same CPU as the 80386, so it has the same addressing capability, memory-management, and protection features as the 80386. The main new features included in the 80486 are a built-in 8-Kbyte code/data cache and a 32-bit floating-point-unit, similar to the 8087 we discussed in Chapter 11.

As perhaps you can see from the preceding brief discussions, the 80286, 80386, and 80486 were designed for use as the CPU in a multitasking microcomputer system. To help you better understand the operation and design rationale of these processors, we start the chapter with a discussion of the problems that must be solved in writing a multitasking/multiuser operating system. We then discuss the 80286, 80386, and 80486 microprocessors in detail and explain how the features designed in these processors help solve the problems involved in implementing a multitasking operating system. After that we discuss how you develop real mode and protected mode programs for systems using these devices.

Finally in the chapter we discuss some of the directions in which microcomputer evolution seems to be heading. Included in this section are discussions of RISC processors, parallel processors, artificial intelligence, "fuzzy" logic, and neural networks.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Describe the difference between time-slice scheduling and preemptive priority-based scheduling.
2. Define the terms blocked, task queue, deadlock, deadly embrace, critical region, semaphore, kernel, memory-management unit, and virtual memory.
3. Describe how "expanded" memory is used to increase the amount of memory available in a microcomputer.
4. Describe how virtual memory gives a computer much more "logical" address space than the physical memory actually present in the system.
5. Describe the types of protection that should be implemented in a multitasking operating system.
6. Describe two methods that can be used to protect a critical region of code.
7. List the major hardware and software features that the 80286 microprocessor has beyond those in the 8086.
8. Show how the 80286 constructs physical addresses in its real address mode and in its protected virtual address mode.
9. List the evolutionary advances that the 80386 has over the 80286.
10. Describe how the 80386 produces a physical address when it is operating in paged mode.
11. Describe how segment-based protection is implemented in an 80386 system operating in protected mode.

12. Describe how an 80386 call gate is used to allow application programs to access operating systems procedures.
13. Describe how an 80386 performs a task switch.
14. Explain the term virtual 8086 mode for an 80386.
15. List the major advances that the 80486 has over the 80386.
16. Describe how system programs are developed for an 80386 or 80486 protected-mode system.
17. Describe how application programs are developed for 80386 or 80486 systems.
18. Describe the operation of the Microsoft Windows multitasking environment.
19. Define the terms RISC, CISC, artificial intelligence, expert system, neural network, and fuzzy logic.

MULTIUSER/MULTITASKING OPERATING SYSTEM CONCEPTS

Introduction

The basic principle of a timeshare system is that the CPU runs one user's program for a few milliseconds, then runs the next user's program for a few milliseconds, and so on until all of the users have had a turn. It cycles through the users over and over, fast enough that each user seems to have the complete attention of the CPU. An operating system which coordinates the actions of a timeshare system such as this is referred to as a *multiuser* operating system. The program or section of a program for each user is referred to as a *task* or *process*, so a multiuser operating system is also commonly referred to as *multitasking*. Multitasking operating systems are also used to control the operation of machines in industrial manufacturing environments. The factory controller program in Figure 10-35 is an example of a very simple multitasking operating system.

In this section we discuss some of the major problems encountered in building a multitasking operating system; then in later sections of the chapter we show you how the features of the 80286, 80386, and 80486 help solve these problems.

Scheduling

TSR PROGRAMS AND DOS

MS DOS is designed as a single-user, single-task operating system. This means that DOS can usually execute only one program at a time. The only exception to this in the basic DOS is the print program, print.com. You may have noticed that when you execute the print command, DOS returns a prompt and allows you to enter another command before the printing is completed. The print program starts printing the specified file and then returns execution to DOS. However, the print program continues to monitor DOS execution. When DOS is sitting in a loop waiting for a user command or some other event, the print program borrows the CPU for a short time and sends more data to the printer. It then returns execution to the interrupted DOS loop.

The DOS print command then is a limited form of multitasking. Products such as Borland's Sidekick use this same technique in DOS systems to provide pop-up menus of useful functions such as a calculator, an appointment book, and a notepad. The first time you run a program such as Sidekick, it is loaded into memory as other programs are. However, unlike other programs, Sidekick is designed so that when you terminate the program, it stays "resident" in memory. You can execute the program and pop up the menu again by simply pressing some hot key combination such as Ctrl-Alt. Programs which work in this way are called *terminate-and-stay-resident* or TSR programs. Because TSRs are so common in the PC world, we thought you might find it interesting to see how they work before we get into discussions of the scheduling techniques used in full-fledged multitasking operating systems.

When you boot up DOS, the basic 640 Kbytes of RAM are set up as shown in Figure 15-1*a*. Starting from absolute address 00000, the first section of RAM is reserved for interrupt vectors. The main part of the DOS program is loaded into the next-higher section of RAM. After this come device drivers such as ANSI.SYS, MOUSE.SYS, etc. The DOS command processor program, command.com, gets loaded into RAM at boot time. This program, which processes user commands and executes programs, has two parts. The resident part of the command processor is loaded in memory just above the device drivers and the transient part is loaded in at the very top of RAM. When you tell DOS to execute a

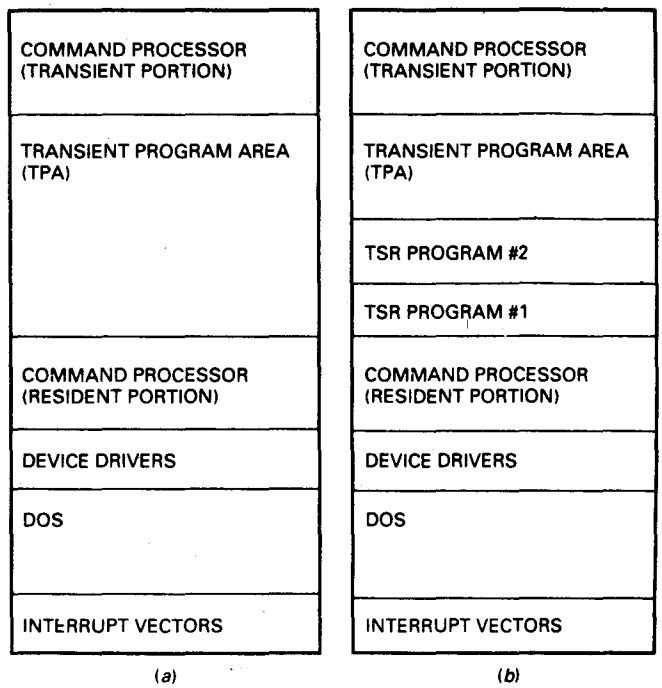


FIGURE 15-1 (*a*) DOS memory map without TSRs. (*b*) DOS memory map with TSRs.

.exe program, the program will be loaded into the transient program area of RAM and, if necessary, into the RAM where the transient part of the command processor was loaded. (The transient part of the command processor will be reloaded when the program terminates.)

Normally, when a program terminates all the transient program area is deallocated, so that another program can be loaded in it to be run. TSR programs, however, are terminated in a special way so that they are left resident in memory, as shown in Figure 15-1*b*. The transient program area is simply reduced by the size of the TSR program(s). When another program is loaded to be run, it is put in RAM above the TSRs.

One question that might occur to you at this point is, How do I make a program resident? To make the program stay resident when it terminates, you use the 31H subfunction of the DOS INT 21H function call. Specifically, you load AH with 31H, load AL with 00H, load DX with the length of the TSR program, and execute the INT 21H instruction. When the program is run from the command line or the AUTOEXEC.BAT file, it will be loaded into RAM, terminated, and left resident.

The next question that might occur to you then is, How does the TSR program get executed after it is resident? The answer to this question is that TSRs are executed as part of interrupt procedures. The exact mechanism depends on whether the TSR is active or passive.

An example of a passive TSR is the switch.com program which I use on my computer. The purpose of this program is to switch the functions of the Caps Lock key and the Ctrl key so that I don't have to retrain my finger to the key positions on my new keyboard. When DOS finds the statement "switch" in my AUTOEXEC.BAT file, it executes the switch.com program. The switch.com program terminates and remains resident. To accomplish the desired switch action, the program "intercepts" the BIOS keyboard interrupt, 09H, as shown in Figure 15-2*a*. You may remember that we showed you how to intercept interrupts at the start of the SDKCOM1 program in Figure 14-27. The result of this interception is that whenever a key is pressed, execution goes first to the switch TSR program. The switch program then calls the BIOS INT 09H procedure to read in codes from the keyboard. If the key code read from the keyboard represents a Caps Lock, it is replaced with the code for a Ctrl, and if the key code represents a Ctrl, it is replaced with the code for a Caps Lock. Other key codes are simply passed on as received. To DOS, then, the switch TSR is simply an interrupt procedure which is executed automatically when a key on the keyboard is pressed.

An example of an active TSR is Borland's Sidekick program, which pops up a menu of command options when you press the Ctrl key and the Alt key. As we mentioned before, Sidekick allows you to temporarily pause during some other program and write a note in a notebook file, perform a calculation on a screen-based calculator, check your appointment schedule, or any one of several other functions. To terminate Sidekick and return to the previously executing program, you press the Esc key. As with the passive switch.com TSR,

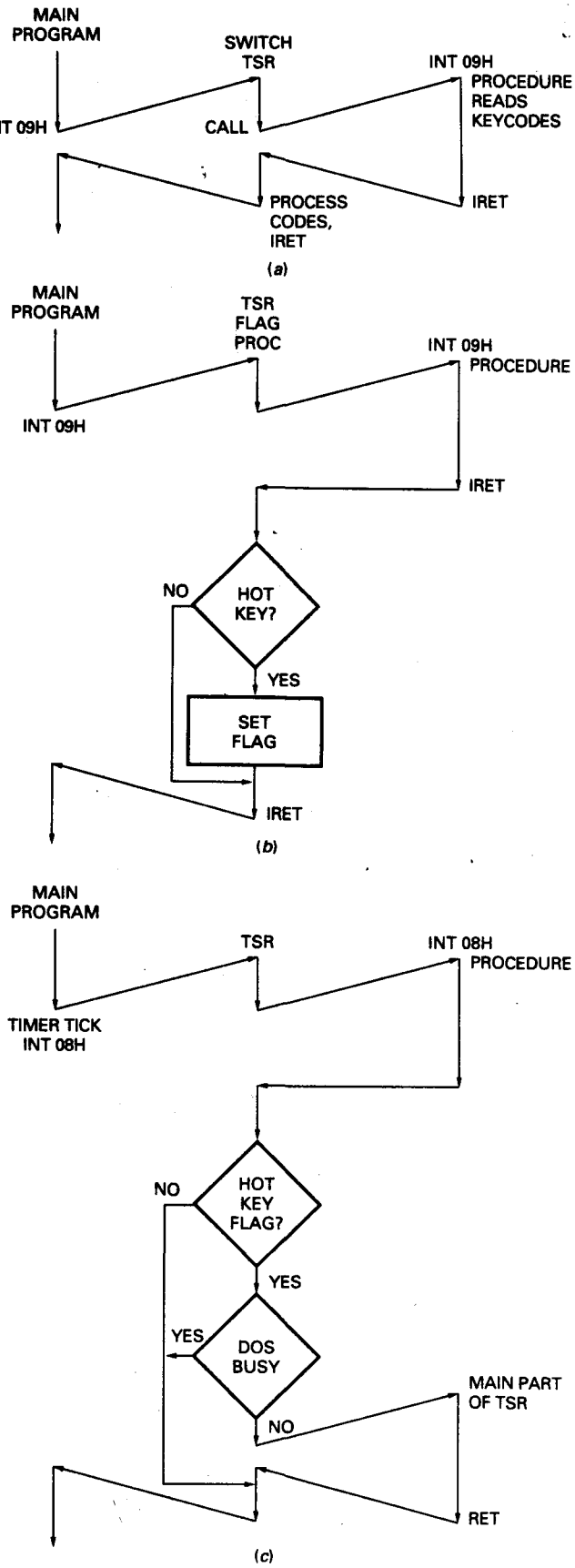


FIGURE 15-2 (*a*) Program flow for switch.com passive TSR. (*b*) Program flow for flag set part of active TSR. (*c*) Program flow for main part of TSR.

you make Sidekick resident by running it from the command line or as part of the AUTOEXEC.BAT file. Figure 15-2*b* and *c* show how an active TSR such as Sidekick is commonly executed when a hot key combination is pressed.

As shown in Figure 15-2*b*, the first part of the TSR intercepts the keyboard interrupt and immediately calls the BIOS keyboard procedure to read in the scan code from the keyboard. When execution returns from the BIOS INT 09H procedure, the TSR checks the returned key code to determine if a hot key was pressed. If a hot key was not pressed, execution simply returns to the interrupted program. If a hot key was pressed, the TSR procedure sets a global flag in memory before returning to the interrupted program. Another section of the TSR will check this flag at periodic intervals to determine if the main part of the TSR should be executed.

The part of the TSR which checks the hot key flag is often connected with the clock tick interrupt procedure, as shown in Figure 15-2*c*. The normal clock tick interrupt vector is replaced with the starting address of this section of the TSR. When a clock tick interrupt occurs (about every 18 ms for a PC- or PS/2-type computer), execution will then go to this section of the TSR. The TSR resets the hot key flag and immediately calls the normal BIOS clock procedure. This call is necessary because the clock procedure updates the system clock and controls the timing of many other system operations. When execution returns to the TSR from the BIOS clock procedure, a check is made to see if a hot key was pressed. If not, execution is simply returned to the program that was interrupted by the clock tick.

If a hot key was pressed, this section of the TSR usually has to determine if DOS or BIOS is executing any procedures before transferring execution to the main part of the TSR. The problem here is that for the most part, DOS and BIOS procedures are not reentrant. This means that the system would probably "lock up" if the TSR happened to call a DOS or BIOS procedure that was executing when the clock tick interrupt occurred. We don't have space to show you the details here, but the DOS INT 28H function can be used to determine if it is safe to run a TSR which uses DOS function calls to access disk files, etc.

If a DOS or BIOS function was in process when the clock tick occurred, execution is simply returned to the interrupted program. When the next clock tick occurs, this middle section of the TSR will again check if DOS is available. If no DOS or BIOS functions were executing when the interrupt occurred, execution will go to the main part of the TSR. When the main part of the TSR finishes, execution is returned to the interrupted main program. Note that the hot key flag was previously reset, so that if another clock tick interrupt occurs while the main part of the TSR is executing, the BIOS clock procedure will be executed, but the main part of the TSR will not be called again.

If you want to experiment with TSRs, the Bibliography lists a couple of references which have detailed examples of how to write TSRs.

As you can perhaps see from the preceding discussion, the TSR scheme allows a microcomputer to do limited multitasking, but it is not useful for controlling a multiuser system. In the next section we discuss the scheduling method commonly used in multiuser operating systems.

TIME-SLICE SCHEDULING

In a full-fledged multitasking or multiuser operating system, the part of the operating system which determines when it is time to switch from one task to another is called the *scheduler*, *dispatcher*, or *supervisor*. The most common method of scheduling task switches is the *time-slice* method which we discussed previously. In a simple round-robin implementation of this approach, the CPU executes one task for perhaps 20 ms and then switches to the next task. After all tasks have had their turn, execution returns to the first. In the program in Figure 10-35 we showed you how a programmable timer, priority-interrupt controller, and interrupt-service procedure can be used to implement this type of scheduling. The UNIX operating system and the OS/2 operating system use a more complex time-slice scheduling approach to implement multitasking. The advantage of the time-slice approach in a multiuser system is that all users are serviced at approximately equal time intervals. As more users are added, however, each user gets serviced less often, so each user's program takes longer to execute. This is referred to as *system degradation*. For industrial control operating systems, this variable scheduling is often not appropriate, so a different scheduling method is used.

PREEMPTIVE PRIORITY-BASED SCHEDULING

In a system which uses *preemptive priority-based scheduling*, an executing low-priority task can be interrupted by a higher-priority task. When the high-priority task finishes executing, execution returns to the low-priority task. This approach is well suited to some control applications because it allows the most important tasks to be done first. Priority-interrupt controllers such as the 8259A are often used to set up and manage the task service requests. The Intel RMX 86 operating system uses priority-based scheduling.

Preserving the Environment

The registers, data, pointers, etc., used by an executing task are referred to as its *environment*, *state*, or *context*. When a task switch occurs, the environment of the interrupted task must be saved so that the task can be restarted properly when it receives another time slice. The usual way of preserving the environment is to keep it in a special memory segment or on a stack. Some operating systems keep a separate stack for each task. In either case, when a task switch occurs the operating system saves the environment of the interrupted task and a pointer to the saved environment. When it is time to switch back to that task, the operating system uses the pointer to access the environment it saved. This process is commonly called "context switching."

A less obvious point in a multitasking system is that any global procedures have to be reentrant. This is

necessary so that if one task is executing a procedure and its time slice ends, other tasks can use the procedure, and the procedure will still complete correctly when execution returns to the first task. Refer to Figure 5-20 if you need a refresher on reentrancy.

Accessing Resources

Another problem encountered in a multitasking system is assuring that tasks have orderly access to resources such as printers, disk drives, etc. As one example of this, suppose that a user at a terminal needs to read a file from a hard disk and print it on the system printer. Obviously the file cannot be read in from the disk and printed in one of the 20-ms time slices allotted to that user, so several provisions must be made to gain access to the resources and hang on to them long enough to get the job done properly. A flag or *semaphore* in memory is used to indicate whether the disk drive is in use by another task or not. Likewise, another semaphore is used to indicate whether the printer is in use. If a task cannot access a resource because it is busy, the task is said to be *blocked*. Now, rather than making the user type in a print command over and over until the disk drive or the printer is available, most operating systems of this type set up queues of tasks waiting for each resource. When one task finishes with a resource, it resets the semaphore for that resource. The next task in the queue can set the semaphore to indicate the resource is busy and then use the resource.

The Need for Protection

An interesting problem can occur in a multitasking operating system when two or more users attempt to read and change the contents of a memory location at the same time. As an example, suppose that an airline ticket-reservation system is operating on a time-slice basis. Now, further suppose that just before the end of his or her time slice, one user examines the memory location which represents a seat on a plane and finds the seat empty. Another user on the system can then, in his or her time slice, examine the same memory location, find it empty, mark it full, and print out a reservation confirmation on the CRT. When execution returns to the first user, his or her program has already checked the seat during its previous time slice, so it marks the seat full, and prints out a reservation confirmation on the CRT. The two people assigned to the same seat may make nasty remarks about computers unless this problem is solved.

The section of a program where the value of a variable is being examined and changed must be protected from access by other tasks until the operation is complete. The section of code which must be protected is called a *critical region* or *critical section*. A technique called *mutual exclusion* is used to prevent two tasks from accessing a critical region at the same time. In the CHK_N_DISPLAY procedure in Figure 14-27 we showed how a critical region can be protected from an interrupt procedure by simply masking the interrupt. In a time-slice system, however, a semaphore is used to provide mutual exclusion.

Figure 15-3 shows how this can be done with 8086 assembly language instructions. The instruction sequence is the same for each task. If task 1 needs to enter a critical section of code, it first loads the semaphore value for critical-region-busy into AL. The single instruction XCHG AL, SEMAPHORE then swaps the byte in AL with the byte in the memory location named SEMAPHORE. It is important to do this in one instruction so that the time-slice mechanism cannot switch to another task halfway through the exchange and cause our airline problem.

After the semaphore is read in Figure 15-3, it is compared with the busy value. If the critical region is busy, execution will remain in a wait loop for as many time slices as are required for the critical region to become free. If the semaphore value is a 0, indicating not busy, then execution enters the critical region. The XCHG instruction has already set the semaphore to indicate the critical region is busy. After execution of

```
;Instructions for accessing critical region of code protected by semaphone - USER 1
      MOV  AL, 01         ; Load semaphore value for region busy
HOLD: XCHG AL, SEMAPHORE  ; Swap and set semaphore
      CMP  AL, 01         ; Check if region is busy
      JE   HOLD           ; Yes, loop until not busy. No enter critical region of code.
;     Instructions which access critical region are inserted here
      MOV  SEMAPHORE, 00  ; Reset semaphore to make critial region available to others.


;Instructions for accessing critical region of code protected by semaphone - USER 2
      MOV  AL, 01         ; Load semaphore value for region busy
HOLD: XCHG AL, SEMAPHORE  ; Swap and set semaphore
      CMP  AL, 01         ; Check if region is busy
      JE   HOLD           ; Yes, loop until not busy. No enter critical region of code.
;     Instructions which access critical region are inserted here
      MOV  SEMAPHORE, 00  ; Reset semaphore to make critial region available to others.
```

FIGURE 15-3 8086 assembly language sequences showing how a flag or semaphore can be used to provide mutual exclusion for a critical region of code.

the critical region finishes, the MOV SEMAPHORE, 00 instruction resets the semaphore to indicate that the critical region is no longer busy. Task 2 can then swap the semaphore and access the critical region when needed. The semaphore functions in the same way as the "occupied" sign on a restroom of a plane or train. If you mentally try interrupting each sequence of instructions at different points, you should see that there is no condition where both tasks can get into the critical region at the same time.

Another region that requires protection is the operating system code. Most single-user operating systems such as DOS do little to prevent user programs from corrupting the operating system code and data areas. The usual results of this and Murphy's law are that an incorrect address in a user program may cause it to write over critical sections of the operating system. The system then "locks up" and the only way to get control again is to reboot the system. In a multitasking system this is intolerable, so several methods are used to protect the operating system.

The major method is to construct the operating system in two or more *layers*. Figure 15-4 shows an "onionskin" diagram for a two-layer operating system. The basic principle here is that the inner circle represents the code and data areas used by the operating system. The outer layer represents the code and data areas of user programs or tasks that are being run under control of the operating system. The inner layer is protected because user programs can only access operating system resources through very specific mechanisms rather than a simple, accidental call or jump. Devices in the Motorola MC68000 family of microprocessors, for example, are designed to accommodate a two-level structure such as this. The MC68000 family devices have two modes of operation, user and supervisory. Certain privileged instructions which affect the operating system can only be executed when the processor is in supervisory mode. As we discuss in great detail later, the Intel 80286, 80386, and 80486 microprocessors have hardware features which allow up to four levels of protection to be built into a system. The 80286, 80386, and 80486 microprocessors also provide a hardware mechanism which can be used to protect tasks from each other.

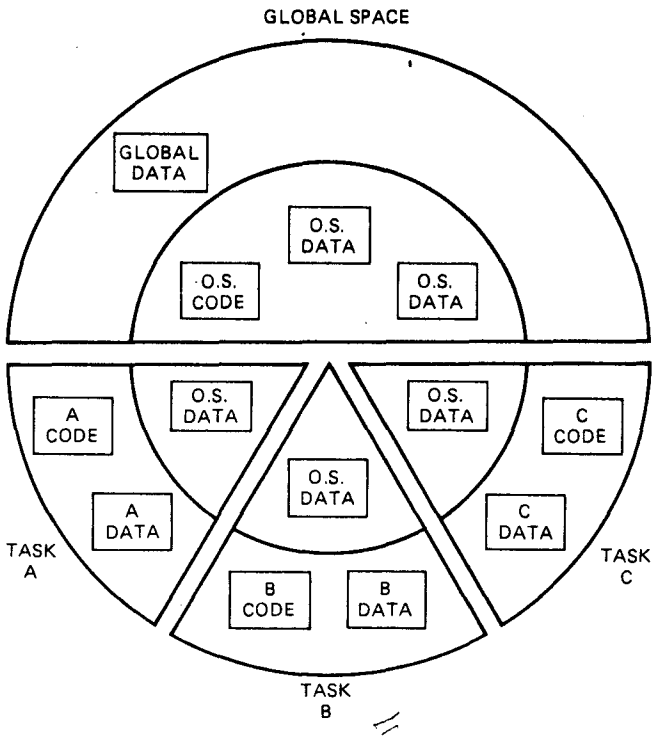


FIGURE 15-4 "Onionskin" diagram showing two-level-protection scheme for multitasking operating system. (*Intel Corporation*)

Memory Management

INTRODUCTION

There are two major reasons why memory must be specifically managed in a multitasking operating system. The first reason is that the physical memory is usually not large enough to hold the operating system and all of the application programs that are being executed by the different users. The second reason is to make sure that executing tasks do not access protected areas of memory. Some memory management can be done by the operating system software, but complete memory management and protection require the aid of hardware called a *memory-management unit* or MMU. Before we get into the operation of an MMU, we want to give you a little background on other methods used to solve the limited memory problem.

OVERLAYS

A common problem, even in older, single-user systems, is that the physical memory is not large enough to hold, for example, an assembler and the program being assembled. The traditional solution to this problem is to write the assembler in modules and use an *overlay* scheme. When the assembler is invoked, the executive module of the assembler is loaded into memory, and an additional block of memory space called the *overlay area* is reserved for the assembler. The first module of the assembler is loaded into this overlay area. When the assembler reaches a point where it needs the next module, it reads that module, referred to as an *overlay*, from disk into the overlay area reserved in memory. When the assembler reaches a point where it needs another overlay, it reads that overlay from disk and loads it into the same overlay area in memory. The overlay approach is commonly used with assemblers, compilers, word processors, and spreadsheet programs. Incidentally, the Borland Turbo C++ tools we introduced you to in Chapter 12 can be used to develop an overlay type program.

BANK SWITCHING, EXPANDED MEMORY, AND EXTENDED MEMORY

Another approach traditionally used to expand the available memory in a microcomputer is *bank switching*. Early microprocessors such as the Intel 8085 have only 16 address lines, so they can directly address only 64 Kbytes of memory. Figure 15-5 shows how the amount

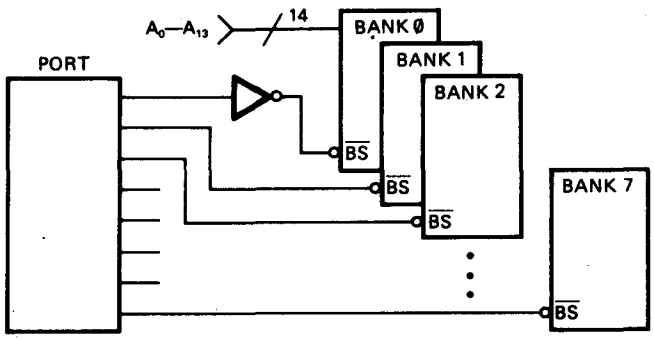


FIGURE 15-5 Block diagram showing how microcomputer memory can be expanded with bank switching.

of memory accessible in a system such as this can be expanded beyond the address limit. The hardware is configured so that when the power is first turned on, the 16-Kbyte bank labeled bank 0 is enabled. Let's assume that this bank occupies system address space 4000H–7FFFH and that system address lines A0–A13 are used to address the bytes in this bank.

To switch to bank 1, a byte which turns off bank 0 and turns on bank 1 is output to the selection port. The bank 1 devices now occupy the address space 4000H–7FFFH and system address lines A0–A13 are used to address the bytes in this bank. Any of the other banks can be switched into the 4000H–7FFFH memory window by simply sending the appropriate word to the control port. As you can see, this bank-switching scheme allows the processor to access 8 banks of 16 Kbytes each or a total of 128 Kbytes through a 16-Kbyte window in the processor address space. Let's see how this scheme is used in IBM PC- and PS/2-type microcomputers.

The 8086 or 8088 processor used in PC-type microcomputers can address up to 1 Mbyte of memory. At the time the IBM PC was developed, it seemed inconceivable that anyone would ever need more than 640 Kbytes of memory for application programs, so all the address space above 640 Kbytes was reserved for the system BIOS, the video frame buffer, and system uses, as shown in Figure 15-6. Also, since the processor could address only 1 Mbyte of memory, DOS was designed to directly address only 1 Mbyte.

As the memory needs of application programs such as spreadsheets and databases banged into the 640-Kbyte limit, designers again looked to bank switching as a means to overcome this limit. The result was the Lotus-Intel-Microsoft Expanded Memory Standard, LIM/EMS 3.2. This combination hardware-software standard has been widely implemented.

The hardware for this expanded memory is often implemented as a plug-in board which contains up to 8 Mbytes of 16-Kbyte pages (banks) and bank-switch registers. The bank-switch registers are used to control which pages from the expanded memory are selected. As shown along the left side of Figure 15-6, in a LIM/EMS 3.2 system the four 16-Kbyte pages selected from the expanded memory at a particular time are mapped into the system address space between C800H and D7FFH. This address space was chosen for the expanded

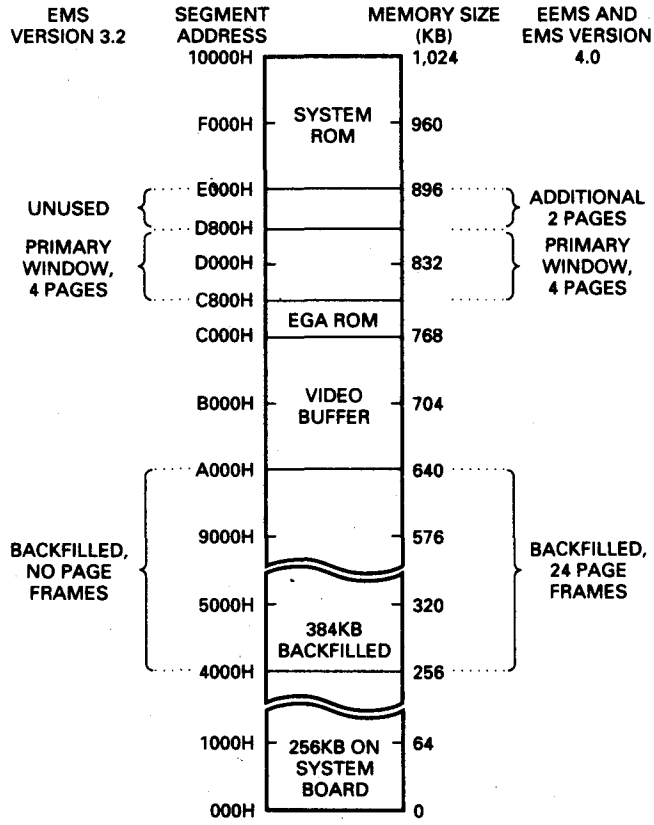


FIGURE 15-6 Memory maps for LIM/EMS 3.2 and LIM/EMS 4.0 expanded memory standards.

memory window because it is not usually used for system functions. The newer LIM/EMS 4.0 standard allows 16-Kbyte pages to be mapped into any system address space that is not populated with ROM or RAM, and it allows the expanded memory to contain up to 32 Mbytes. As shown along the right side of Figure 15-6, LIM/EMS 4.0 allows pages above the 640-Kbyte boundary and additional pages in the 384-Kbyte region below the boundary.

The software part of either EMS standard includes a driver program called EMM.SYS. This driver program is installed in memory by including the statement device=emm.sys in the CONFIG.SYS program which runs when you boot your system. The EMM.SYS driver contains the functions which allow application programs to allocate and access expanded memory. The expanded memory functions are called with a software INT 67H. The value in AH determines the specific function that is called. The complete list of expanded memory functions is extensive, but to give you an idea of some of what is available, Figure 15-7 shows a few of the functions. Basically, an application program must use these functions to allocate enough expanded memory for its code and data, switch in pages as needed, and deallocate the expanded memory when it terminates so that the memory is available for the next program. Incidentally, MS DOS versions 4.0 and later support LIM/EMS 4.0.

As we discuss in detail later, the 80286, 80386, and 80486 microprocessors have more address lines than

EXPANDED MEMORY FUNCTION	CALL WITH	RETURNS
GET STATUS	AH = 40H	AH = STATUS
GET PAGE FRAME ADDRESS	AH = 41H	AH = STATUS BX = PAGE FRAME SEGMENT
GET NUMBER OF EXPANDED MEMORY PAGES	AH = 42H	AH = STATUS BX = AVAILABLE PAGES DX = TOTAL PAGES
ALLOCATE EXPANDED MEMORY PAGES	AH = 43H BX = NO. OF PAGES	AH = STATUS DX = EMM HANDLE
MAP EXPANDED MEMORY PAGE	AH = 44H AL = PHYSICAL PAGE BX = LOGICAL PAGE DX = EMM HANDLE	AH = STATUS
RELEASE EXPANDED MEMORY PAGES	AH = 45H DX = EMM HANDLE	AH = STATUS
GET EMM VERSION	AH = 46H	AH = STATUS AL = VERSION

FIGURE 15-7 Examples of EMM functions called through INT 67H.

an 8086 and can directly address considerably more memory. Memory located in the address space above 1 Mbyte is commonly referred to as *extended memory* or *XMS memory*. If a system using one of these processors is running under a version of DOS before 5.0, however, it still has the 1-Mbyte memory limit imposed by DOS. In other words, the extended memory in a system is invisible to DOS and will not be used for programs. There are three common cures for this problem.

One solution is to use a memory-management-device driver program which allows the extended memory to function as expanded memory. Another solution is to use a "DOS extender" program such as Phar Lap Software's 386/DOS extender or A.I. Architect's OS/x86. These programs operate under DOS, so they use the familar DOS commands, but they allow programs to take advantage of the advanced features of the 80286, 80386, and 80486 processors. The third solution to the DOS memory limit is to switch to an operating system such as Microsoft's OS/2, which is designed to take advantage of the addressing range and other features of the newer processors.

The expanded memory scheme we described in the preceding section makes more memory available to a program, but it has several disadvantages. One disadvantage is that the system must contain enough expanded memory for the largest program to be run. With today's large programs this could be a major expense. A second disadvantage of expanded memory is that application programs must manage the switching of pages in and out of the expanded memory window. This adds overhead to the execution time, and if a program is modified, the switching points may have to be changed. Still another disadvantage is that operating system and user-task protection are not easily implemented. The virtual memory scheme we discuss next helps solve these problems.

VIRTUAL MEMORY AND MMUs

Virtual memory is basically an extension of the memory caching scheme we discussed in Chapter 11. To refresh your memory of a cache system, take another look at Figure 11-11. The virtual memory scheme simply adds a hard-disk drive to the memory hierarchy. The hard-disk drive becomes the main program and data memory, the DRAM functions as an intermediate cache, and the SRAM cache functions as a high-speed cache for the DRAM. In a virtual memory system the code and data segments currently being used for program execution are loaded from the disk into DRAM and accessed by the cache controller as needed. If an executing program needs a segment that is not currently in DRAM, the required segment is read in from the disk to the DRAM main memory. If the DRAM is full, one of the segments in the DRAM is swapped out to the disk to make room, and the required segment is swapped into DRAM.

There are three different ways of setting up the code and data blocks to be swapped in and out of DRAM. One scheme is to swap segments. The advantage of segment swapping is that segments correspond to the code and data structures in the program. The disadvantage of the segment scheme is that with processors such as the 80386 and 80486, segments can be very large. The time required to swap in a large segment would appreciably slow down the execution of a program. Also, it is often hard to fit variable-sized segments in memory. A second swapping scheme uses fixed-length pages of typically 4 Kbytes each. These small pages can be quickly swapped in and out of memory, but they don't correspond to the logical structure of the program. A third approach, implemented in the 80386 and 80486 microprocessors, allows a programmer to write a program using logical segments and divide the segments into 4-Kbyte pages for swapping in and out of physical memory.

The term virtual here refers to memory space that appears to be present from a programmer's viewpoint but is not physically present in the DRAM main memory. In other words, if you are writing a program for a system with virtual memory, you can create segments as if you had, for example, a gigabyte of memory space, even though the system has only perhaps 4 Mbytes of physical memory. The virtual memory space can be much larger than the physical memory, because all of the logical segments are not present in physical memory at any one time. As with the SRAM cache scheme, a virtual memory system works because most programs only need small sections of code and data at a particular time.

Virtual memory can be managed totally by the operating system, but most microcomputer systems use a hardware device called a *memory-management unit* or MMU to assist in the process. The Intel 80286, 80386, and 80486 and the Motorola MC68030 and MC68040 have a complete MMU integrated on the chip with the CPU. Separate MMUs are available for use with other processors. In either case the MMU is functionally positioned between the processor and the actual memory. Figure 15-8, page 542, shows an overview of how the MMUs in the 286, 386, and 486 processors manage segment-based virtual memory. The first step in explaining this is to clarify the terms logical address and physical address.

When you write an assembly language program, you usually refer to addresses by name. The addresses you work with in a program are called *logical addresses*, because they indicate the logical positions of code and data. An example of this is the 8086 instruction JNZ

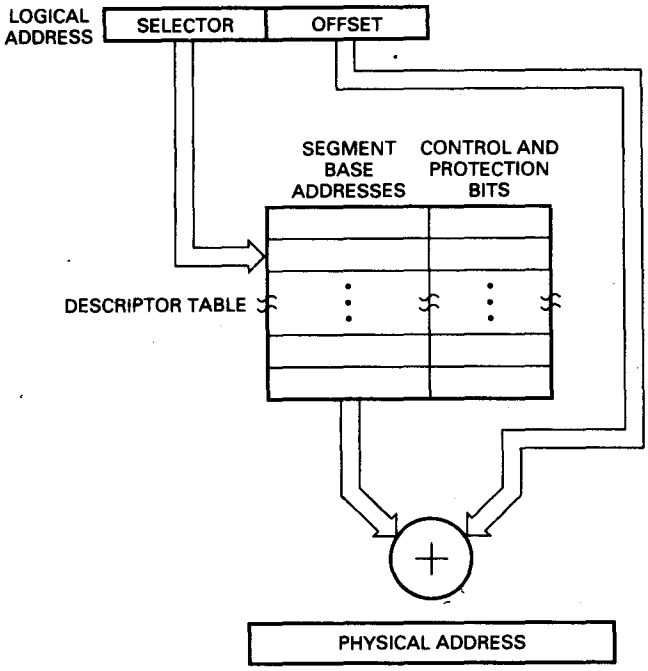


FIGURE 15-8 Block diagram showing how segment-based virtual memory is implemented in 80286, 80386, and 80486 processors.

NEXT. The label NEXT represents a logical address that execution will go to if the zero flag is not set. When an 8086 program is assembled, each logical address is represented with a 16-bit offset and a 16-bit segment base. The 8086 BIU then produces the actual physical memory address by simply adding these two parts together, as explained many times previously.

When a program is assembled or compiled to run on a system with an MMU, each logical or virtual address is also represented by two components, but the components function differently. In a segment-oriented system such as an 80286, the upper 16-bit component is referred to as a *segment selector*, and the lower component is referred to as the *offset*. As shown in Figure 15-8, the MMU uses the segment selector to access a *descriptor* for the desired segment in a table of descriptors in memory. A descriptor is a series of memory locations that contain the physical base address for a segment, the privilege level of the segment, and some control bits.

The selectors for the 80286, 80386, and 80486 have 14 address bits and 2 privilege-level bits. The 14 address bits in the selector can select any one of 16,384 descriptors in the descriptor table. Since each descriptor represents a segment, this means that a program can access up to 16,384 segments. For an 80286 the offset part of the virtual address is 16 bits, so each segment can contain up to 64 Kbytes. The logical or virtual address space accessible by an 80286 then is 16,384 segments × 65,536 bytes/segment, or about 1 Gbyte. What this means is that the operating system and other programs can function as if a gigabyte of memory were available.

The physical memory is the amount of RAM and ROM actually present in the system. For this example let's assume that the MMU has 24 address lines so it can address 16 Mbytes of physical memory. Remember from our previous discussion that the physical memory, whatever its actual size, is simply a holding place for the segments currently being used by the operating system and user programs.

When the MMU receives a logical address from the CPU, it checks to see if that segment is currently in the physical memory. If the segment is present in physical memory, the MMU adds the offset component of the address to the segment base component of the address from the segment descriptor to form the physical address. It then outputs the physical address to memory on the memory address bus. The addressed code or data word is returned to the CPU on the data bus.

If the MMU finds that the segment specified by the selector part of the logical address is not in memory, it sends an interrupt signal to the CPU. In response to the interrupt, the operating system executes an interrupt procedure which reads the desired code or data segment from disk and loads it into the physical memory. The MMU then computes and outputs the physical address as described before. The operation is semiautomatic, so other than a slight delay, the user is not aware that the segment had to be loaded. In a well-structured system with a reasonably large amount of physical memory, the *hit rate* may be 90 to 95 percent.

When the CPU or smart MMU wants to load a segment from disk into physical memory, it must first make space for it in the physical memory. Depending on the system, it may do this by compacting the segments already present and changing the descriptors to point to the new physical locations or by swapping the segment being brought in with one currently in physical memory. To help in deciding which segment to swap back to memory, many systems use an *accessed* bit in the descriptor to keep track of how many times the segment has been used. A low-use segment is the most likely candidate to swap back to disk. Some virtual memory systems also have a *dirty* bit in each descriptor. This bit will be set if the contents of a segment have been changed. If the dirty bit is set, a segment must be written back to disk when its space is needed. If the dirty bit is not set, then the segment has not been altered, and the copy of the segment on disk is current. In this case the segment can just be overwritten by the new segment. This check saves the time that would be required to write the segment to disk.

The use of a descriptor table to translate logical addresses to physical addresses has another major advantage besides making virtual memory possible. The selector component of each logical address contains 2 bits which represent the privilege level of the program section requesting access to a segment. The descriptor for each segment contains 2 bits which represent the privilege level of that segment. When an executing program attempts to access a segment, the MMU compares the privilege level in the selector with the privilege level in the descriptor. If the segment selector has the same or a greater privilege level, then the MMU allows

the segment to be accessed. If the selector privilege level is lower than the privilege level in the descriptor, the MMU refuses the access and sends an interrupt signal to the CPU indicating a privilege-level violation. As you can see, privilege bits and this indirect method of producing physical addresses provides a mechanism for protecting segments such as those containing the operating system kernel from application programs.

To summarize then, an MMU is used to manage virtual memory. The MMU uses a descriptor table to translate logical or virtual program addresses to physical addresses. This indirect approach makes possible a virtual address space much larger than the physical address space. The indirect approach also makes it possible to protect a memory segment from access by a program section with a lower privilege level. You will meet all these concepts again in the following sections, where we discuss the 80286, 80386, and 80486 microprocessors which have integrated MMUs.

THE INTEL 80286 MICROPROCESSOR

Introduction

The needs of a multitasking/multiuser operating system include environment preservation during task switches, operating system and user protection, and virtual memory management. The Intel 80286 was the first 8086 family processor designed to make implementation of these features relatively easy. The 80286 was used as the CPU in the IBM PC/AT and its clones, in the IBM PS/2 Model 50, and in the IBM PS/1. Although the 80286 has to a large extent been superseded by the 80386, the 80386SX, and the 80486, there are still many 80286-based systems in use and more '80286 systems being sold. Therefore, we will use a little space to tell you about the basic operation of an 80286.

80286 Architecture, Signals, and System Connections

As you can see in the block diagram in Figure 15-9, an 80286 contains four separate processing units.

The *bus unit* (BU) in the device performs all memory and I/O reads and writes, prefetches instruction bytes, and controls transfer of data to and from processor extension devices such as the 80287 math coprocessor.

The *instruction unit* (IU) fully decodes up to three prefetched instructions and holds them in a queue, where the execution unit can access them. This is a further example of how modern processors keep several instructions "in the pipeline" instead of waiting to finish one instruction before fetching the next.

The *execution unit* (EU) uses its 16-bit ALU to execute instructions it receives from the instruction unit. When operating in its real address mode, the 80286 register set is the same as that of an 8086 except for the addition of a 16-bit machine status word (MSW) register, which we will discuss later.

The *address unit* (AU) computes the physical addresses that will be sent out to memory or I/O by the BU. The 80286 can operate in one of two memory address modes, *real address mode* or *protected virtual address mode*. If the 80286 is operating in the real address mode, the address unit computes addresses using a segment base and an offset just as the 8086 does. The familiar CS, DS, SS, and ES registers are used to hold the base addresses for the segments currently in use. The maximum physical address space in this mode is 1 Mbyte, just as it is for the 8086.

If an 80286 is operating in its *protected virtual address mode* (protected mode), the address unit functions as a complete MMU. In this address mode the 80286 uses all 24 address lines to access up to 16 Mbytes of physical memory. In protected mode it also provides up to a gigabyte of virtual memory using the descriptor table scheme shown in Figure 15-8.

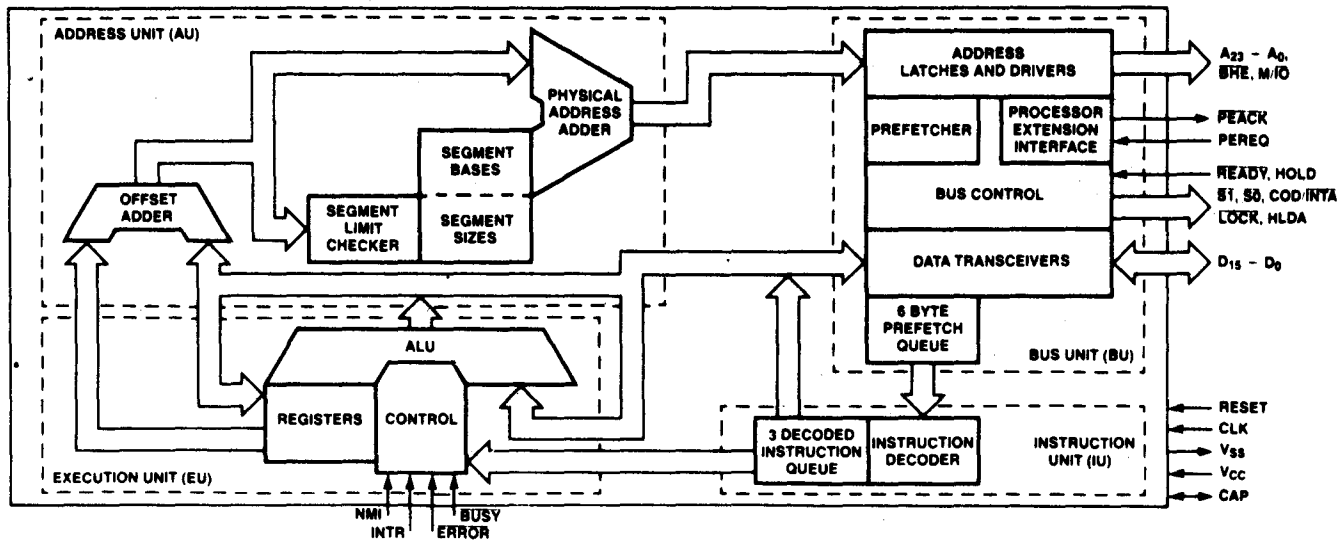


FIGURE 15-9 80286 internal block diagram. (*Intel Corporation*)

Figure 15-10 shows the 68-pin package that is usually used for an 80286, and Figure 15-11 shows how an 8086 is connected with some other components to form a simple system. Keep thse figures handy as we work our way around the major pins of the 80286. Many of the signals of the 80286 should be familiar to you from our discussion of the 8086 signals in Chapter 7.

The 80286 has a 16-bit data bus and a 24-bit nonmultiplexed address bus. The 24-bit address bus allows the processor to access 16 Mbytes of physical memory when operating in protected mode. Memory hardware for the 80286 is set up as an odd bank and an even bank, just as it is for the 8086. The even bank will be enabled when A0 is low, and the odd bank will be enabled when $\overline{BHE}$ is low. To access an aligned word, both A0 and $\overline{BHE}$ will be low. External buffers are used on both the address and the data bus.

From a control standpoint, the 80286 functions similarly to an 8086 operating in maximum mode. Status signals $\overline{S0}$, $\overline{S1}$, and M/$\overline{IO}$ are decoded by an external 82288 bus controller to produce the control bus, read, write, and interrupt-acknowledge signals.

The HOLD, HLDA, INTR, $\overline{INTA}$, (NMI), $\overline{READY}$, and $\overline{LOCK}$ and RESET pins function basically the same as they do on an 8086. An external 82284 clock generator is used to produce a clock signal for the 80286 and to synchronize RESET and $\overline{READY}$ signals.

The final four signal pins we need to discuss here are used to interface with processor extensions (coprocessors) such as the 80287 math coprocessor. The *processor extension request* (PEREQ) input pin will be asserted by a coprocessor to tell the 80286 to perform a data transfer to or from memory for it. When the 80286 gets around to do the transfer, it asserts the *processor extension acknowledge* ($\overline{PEACK}$) signal to the coprocessor to let it know the data transfer has started. Data transfers are done through the 80286 in this way so that the coprocessor uses the protection and virtual memory capability of the MMU in the 80286. The $\overline{BUSY}$ signal input on the 80286 functions the same as the $\overline{TEST}$ input does on the 8086. When the 80286 executes a WAIT instruction, it will remain in a WAIT loop until it finds the $\overline{BUSY}$ signal from the coprocessor high. If a coprocessor finds some error during processing, it will assert the $\overline{ERROR}$ input of the 80286. This will cause the 80286 to automatically do a type 16H interrupt call. An interrupt-service procedure can be written to make the desired response to the error condition.

The machine cycle waveforms for the 80286 are very similar to those of the 8086 that we showed and discussed in earlier chapters. You should be able to work your way through them in the Intel 80286 data sheets if you need that type of information.

As we mentioned before, the 80286 is used as the CPU in the IBM PC/AT and its clones. These AT-type machines use the AT/ISA bus shown in Figure 11-7*b* to interface with a CRT controller card, disk controller cards, and other peripheral cards.

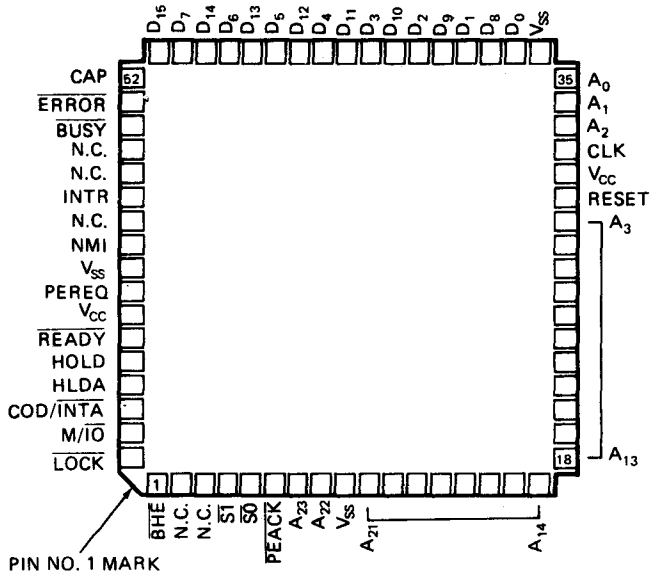


FIGURE 15-10 Pin diagram for 80286 microprocessor. (*Intel Corporation*)

80286 Real Address Mode Operation

After the 80286 is reset, it starts executing in its real address mode. This mode is referred to as real because physical memory addresses are produced by directly adding an offset to a segment base, just as they are in an 8086. In this mode the 80286 can address up to 1 Mbyte of physical memory and functions essentially as a "souped-up" 8086. Due to the extensive pipelining and other hardware improvements, the 80286 will execute most programs several times faster than an 8086 with the same-frequency clock signal.

When operating in real address mode, the interrupt-vector table of the 80286 is located in the first 1 Kbyte of memory, just as it is for an 8086, and the response to an interrupt is the same as that of an 8086. As shown in Figure 15-12 the 80286 has several additional built-in interrupt types. Some of these types will not make much sense until we dig a little deeper into the operations of the 80286 and the 80386, but while we are here we will introduce you to a few new terms used in Figure 15-12.

The 80186 and later processors separate interrupts into two categories, interrupts and exceptions. Asynchronous external events which affect the processor through the INTR or NMI input are referred to as interrupts. An exception-type interrupt is generated by some error condition that occurred during the execution of an instruction. Dividing by zero is an example of an operation that will cause an exception. Software interrupts produced by the INT n instruction are classified as exceptions, because they are synchronous with the processor.

Exceptions are further divided into faults and traps. Faults are exceptions that are detected and signaled before the faulting instruction is executed. The segment-not-present exception is an example of a fault. Traps are exceptions which are reported after the instruction which caused the exception executes. The divide-by-zero exception and the INT n interrupts are examples of traps.

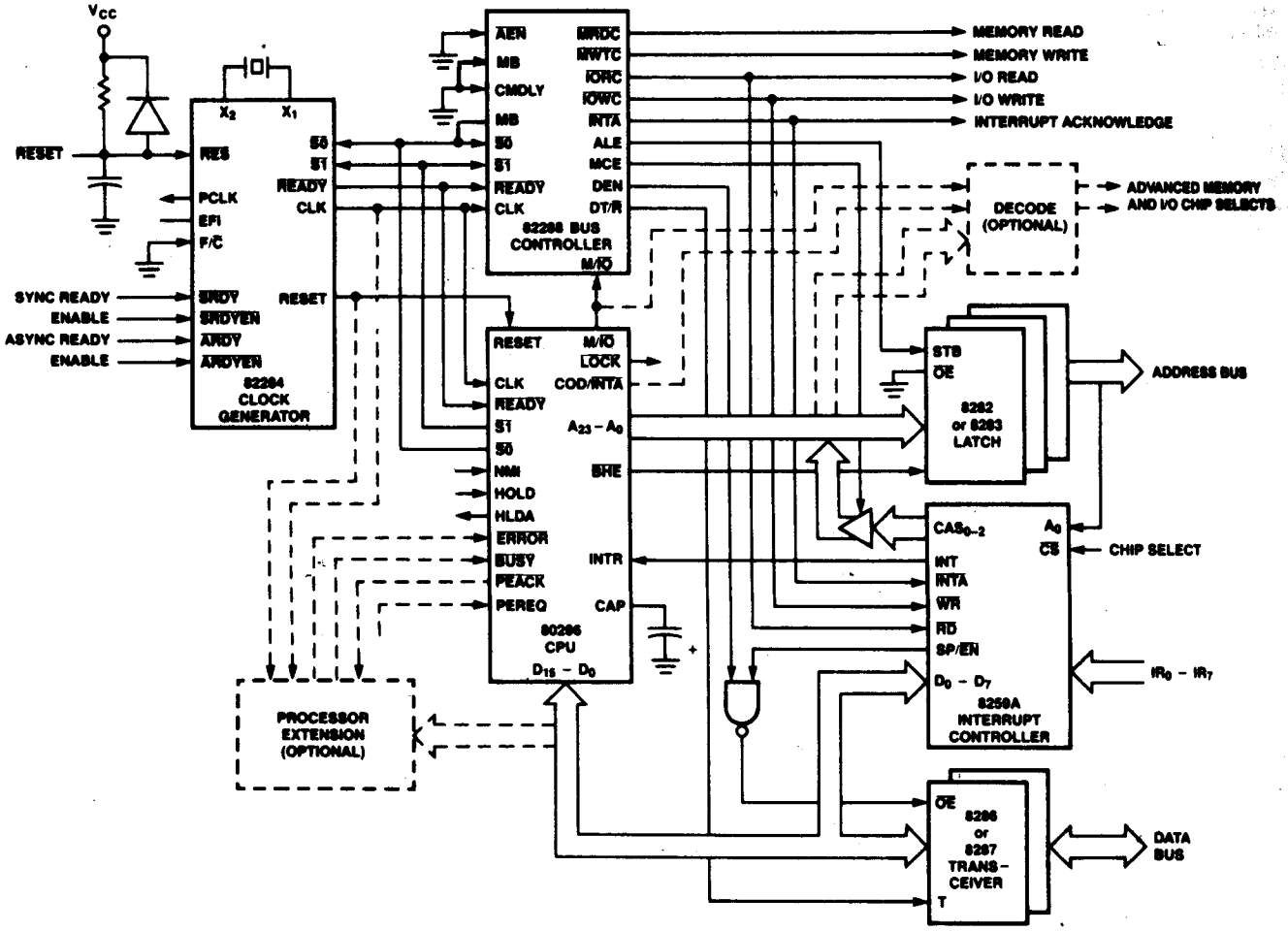


FIGURE 15-11 Circuit connections for simple 80286 system. (*Intel Corporation*)

FUNCTION	INTERRUPT NUMBER
DIVIDE ERROR EXCEPTION	0
SINGLE STEP INTERRUPT	1
NMI INTERRUPT	2
BREAKPOINT INTERRUPT	3
INTO DETECTED OVERFLOW EXCEPTION	4
BOUND RANGE EXCEEDED EXCEPTION	5
INVALID OPCODE EXCEPTION	6
PROCESSOR EXTENSION NOT AVAILABLE EXCEPTION	7
INTERRUPT TABLE LIMIT TOO SMALL	8
PROCESSOR EXTENSION SEGMENT OVERRUN INTERRUPT	9
INVALID TASK STATE SEGMENT	10
SEGMENT NOT PRESENT	11
STACK SEGMENT OVERRUN OR NOT PRESENT	12
SEGMENT OVERRUN EXCEPTION	13
RESERVED	14,15
PROCESSOR EXTENSION ERROR INTERRUPT	16
RESERVED	17-31
USER DEFINED	32-255

FIGURE 15-12 80286 interrupt types. (*Intel Corporation*)

80286 Protected-Mode Operation

As we said before, after a reset the 80286 operates in real address mode. On an 80286-based system running under MS DOS or a similar operating system, the 80286 is left in real address mode because current versions of DOS are not designed to take advantage of the protected-mode features of the 80286. If an 80286-based system is running an operating system such as Microsoft's OS/2, which uses the protected mode, the real mode will be used to initialize perhipheral devices, load the main part of the operating system from disk into memory, load some registers, enable interrupts, set up descriptor tables, and switch the processor to protected mode. The first step in switching an 80286 to protected mode is to set the protection enable bit in the *machine status word* (MSW) register in the 80286. Figure 15-13*a*, page 546, shows the format for the MSW. Bits 1, 2, and 3 of the MSW are for the most part used to indicate whether a processor extension (coprocessor) is present in the system or not. Bit 0 of the MSW is used to switch the 80286 into protected mode. To change bits in the MSW you load the desired word in a register or memory location and execute the *load machine status word* (LMSW) instruction. The final step to get the 80286 operating

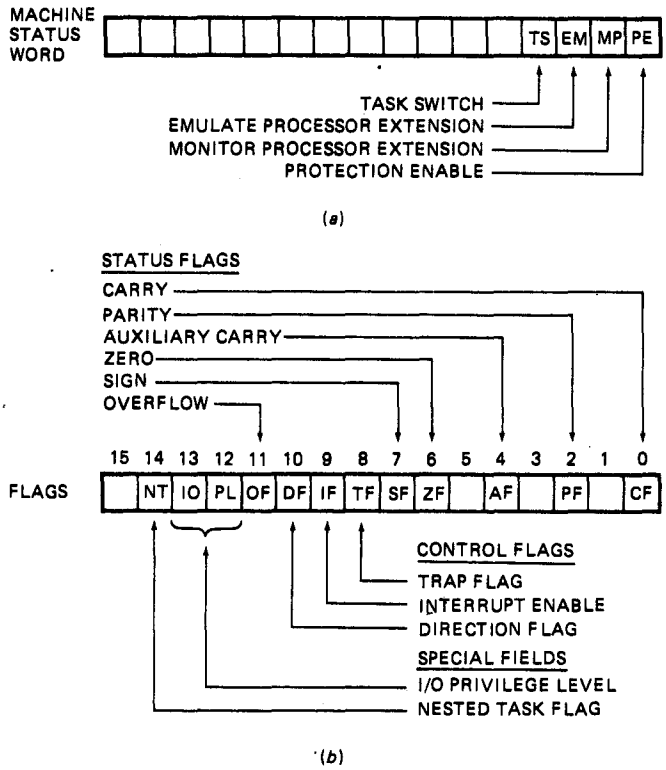


FIGURE 15-13 (*a*) 80286 machine status word bits. (*b*) 80286 flag register bits. (*Intel Corporation*)

in protected mode is to execute an intersegment jump to the start of the main system program. This jump is necessary to flush the instruction byte queue because in protected mode the queue functions differently from the way it does in real mode.

Switching an 80286 to protected mode enables the integrated MMU to provide virtual memory and protection. As we described in an earlier section on virtual memory, a 286 virtual address consists of a 16-bit selector and a 16-bit offset. The MMU uses 14 bits of the selector to access a descriptor for the desired segment in a table of descriptors. The descriptor contains the 24-bit physical base address, the privilege level, and some control bits for the segment. If the privilege level contained in the selector is as high as or higher than the privilege level contained in the descriptor, then access to the segment will be allowed. If not, an exception will be generated. The MMU also checks the "P" bit in the descriptor to determine if the segment is present in physical memory. If not, the MMU will generate a segment-not-present exception. The service procedure for this exception will load the segment in memory and return to the interrupted program. If the memory access meets the privilege level test and the segment is present in physical memory, the MMU will add the 16-bit offset from the logical address to the 24-bit base address from the descriptor to produce the 24-bit physical address for the desired byte or word in the segment. Remember that in protected mode an 80286 uses all 24 address lines, so it can address 16 Mbytes of memory instead of just the 1 Mbyte addressable in real mode.

Once an 80286 is switched into protected mode by executing the LMSW instruction, the only way to get an 80286 back to its real address mode is by resetting the system. The 80286 was designed this way so that a "clever" programmer could not switch the system back into real address mode to defeat the protection schemes in protected mode. Unfortunately, this design also prevents an operating system running in protected mode on an 80286 from easily switching back to real mode to run a section of an 8086 real-mode program during a time slice. In other words, an 80286 operating in protected mode cannot easily multitask a mixture of programs with 8086 segment-offset-type addressing and 80286 selector-offset-type addressing. For this and other reasons, relatively little software has been written to take advantage of the memory-management and protection features available in the 80286 protected mode. The designs of the 80386 and 80486 processors solved the 80286 problems and added other features which make multitasking easier to implement. Much of the new software written during the lifetime of this book will utilize the advanced features of the 386 and 486. Therefore, we decided that the limited space we have available is better used to discuss the details of how the 386 and 486 manage virtual memory and provide protection. The protected mode operation of the 386 is very similar to that of the 80286, so if you have to work on a protected-mode 80286 system, you should have little difficulty "going back."

80286 New and Enhanced Instructions

From a software standpoint the 80286 was designed to be upward-compatible from the 8086 so that the huge amount of software developed for the 8086/8088 could be easily transported to the 80286. The instruction set of the 80286 and later processors are "supersets" of the 8086 instructions. Here's a brief description of the new and enhanced instructions available on the 80286.

Real- or protected-mode instructions

INS—Input string.

OUTS—Output string.

PUSHA—Push eight general-purpose registers on stack.

POPA—Pop eight general-purpose registers from stack.

PUSH immediate—Push immediate number on stack.

SHIFT/ROTATE destination, immediate—Shift or rotate destination register or memory location specified number of bit positions.

IMUL destination, immediate—Signed multiply destination by immediate number.

IMUL destination, multiplicand, immediate multiplier—Signed multiply, result in specified destination.

ENTER—Set up stack frame in procedure. Saves BP, points BP to TOS, and allocates stack space for local variables.

LEAVE—Undo ENTER actions before RET in procedure.

BOUND—Causes a type 5 execution if value in specified register is not within the specified range for an array.

LMSW—Load machine status word (LMSW) is used to switch the 80286 from real mode to protected mode.

Protected-mode instructions

> NOTE: We postponed much of the discussion of protected mode to a later section on the 386 processor, so many of these instructions will be much more understandable to you after you read that section.

CTS—Clear task-switched flag in machine status word.

LGDT—Load global descriptor table register from memory.

SGDT—Store global descriptor table register contents in memory.

LIDT—Load interrupt descriptor table register from memory.

LLDT—Load selector and associated descriptor into LDTR.

SLDT—Store selector from LDTR in specified register or memory.

LTR—Load task register with selector and descriptor for TSS.

STR—Store selector from task register in register or memory.

LMSW—Load machine status register from register or memory.

SMSW—Store machine status word in register or memory.

LAR—Load access rights byte of descriptor into register or memory.

LSL—Load segment limit from descriptor into register or memory.

ARPL—Adjust requested privilege level of selector (down only).

VERR—Determine if segment pointed to by selector is readable.

VERW—Determine if segment pointed to by selector is writeable.

THE INTEL 80386 32-BIT MICROPROCESSOR

Introduction

Some of the limitations of the 80286 microprocessor are that it has only a 16-bit ALU, its maximum segment size is 64 Kbytes, and it cannot easily be switched back and forth between real and protected modes. The Intel 80386 microprocessor was designed to overcome these limits, while maintaining software compatibility with the 80286 and earlier processors. The 80386 has a 32-bit ALU, so it can operate directly on 32-bit data words. 80386 segments can be as large as 4 Gbytes and a program can have as many as 16,384 segments. The virtual address space then is 16,384 segments × 4 Gbytes, or about 64 Tbytes (terabytes). A 32-bit address bus allows an 80386 to address up to 4 Gbytes of physical memory. The 80386 has a "virtual 8086" mode, which allows it to easily switch back and forth between 80386 protected-mode tasks and 8086 real-mode tasks. Later we will discuss 80386 memory addressing, protection, and operating modes, but for now we want to discuss the hardware operation and system connections.

80386 Architecture, Pins, and Signals

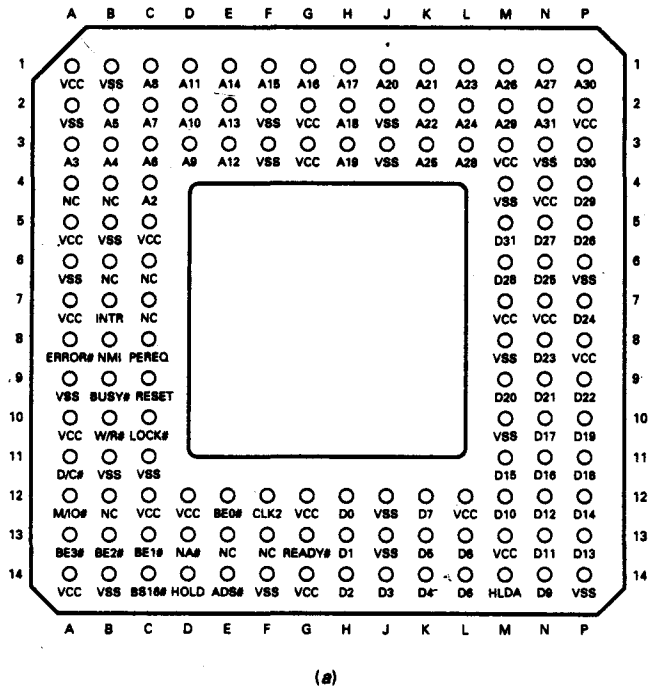
The 80386 processor is available in two different versions, the 386DX and the 386SX. The 386DX has a 32-bit address bus and a 32-bit data bus. It is packaged in the 132-pin ceramic pin grid array package shown in Figure 15-14*a*, page 548. The 386SX, which is packaged in the 100-pin flatpack shown in Figure 15-14*b*, has the same internal architecture as the 386DX, but it has only a 24-bit address bus and a 16-bit data bus. The lower cost package and the ease of interfacing to 8-bit and 16-bit memory and peripherals make the 386SX suitable for use in lower cost systems. The trade-off here, of course, is that the 386SX address range and memory transfer rate are lower than those of the 386DX. Any reference to the 386 in the rest of this chapter will mean the 386DX unless specifically indicated otherwise.

Figure 15-15, page 548, shows the major signal groups for a 386DX. Most of these signals should be familiar to you from the discussions of earlier processors. Let's work our way around the device to pick up the new ones.

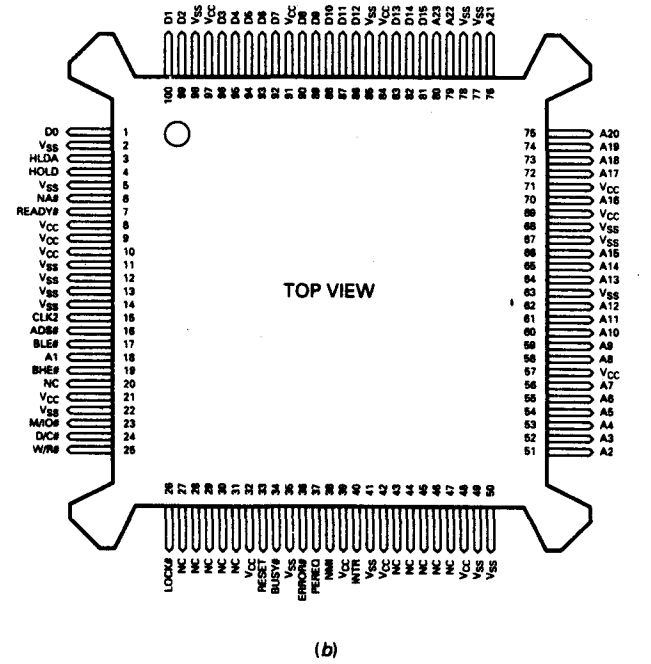
The clock signal applied to the 386 CLK2 input is internally divided by 2 to produce the clock signal which actually drives processor operations. For 33-MHz operation then, a 66-MHz signal is applied to the CLK2 input by an external clock generator such as the 82384.

The 386 address bus consists of the A2–A31 address lines and the byte enable lines BE0#–BE3#. The BE0#–BE3# lines are decoded from internal address signals A0 and A1 and function very similarly to the way A0 and BHE function in an 8086 or 80286 system. The 386 has a 32-bit data bus, so memory can be set up as four byte-wide banks. The BE0#–BE3# signals function as enables for the four banks. These individual enables allow the 386 to transfer bytes, words, or double words to and from memory. Incidentally, the # symbol after the BE signal names indicates that these signals are active low.

The bus cycle definition signals identify the type of operation that is occurring during a bus cycle. The WR/R# signal indicates whether a read or write operation is taking place and the D/C# indicates whether the bus operation is a data read/write or a control-word transfer such as an op-code fetch. M/IO# indicates whether the operation is a memory or a direct input/output operation. Incidentally, the 386 direct I/O port structure



(a)



(b)

FIGURE 15-14 (a) Pin diagram for 386DX processor view from pin side. (b) Top view pin diagram for 386SX processor. (*Intel Corporation*)

is simply an extension of the 8086 and 80286 port structure to include 32-bit ports. Simple 32-bit I/O ports can be constructed by connecting 8-bit I/O port devices such as the 8255A in parallel. A 386 can use an IN or OUT instruction followed by an 8-bit port address to address up to 256 8-bit ports, 128 16-bit ports or 64 32-bit ports. Using the DX register to hold a 16-bit port address, a 386 can access up to 64K 8-bit ports, 32K 16-bit ports, or 8K 32-bit ports.

The PEREQ signal is output by a coprocessor such as an 80387 floating point processor to tell the 386 to fetch the first part of a data word for the coprocessor. The coprocessor will then take over the buses and read the rest of the data word, as we described for the 8087 in Chapter 11. As we also described in Chapter 11, the BUSY# signal is used by the coprocessor to prevent the 386 from going on with its next instruction before the coprocessor is finished with the current instruction. If the ERROR# signal is asserted by a coprocessor, the 386 will perform a type 16 exception.

Regarding the V_{cc} and ground connections, note in Figure 15-15 that the 386 has a large number of V_{cc} pins. It also has a large number of ground connections labeled V_{ss}. These pins are all connected to the appropriate power plane in the PC board.

The RESET, NMI, INTR, HOLD, and HLDA inputs function similarly to the way they do in earlier processors. In a later section we will describe how the 386 handles interrupts while operating in protected mode.

The final group of 386 signals to discuss is the bus control group. The READY# signal is used to insert wait states in bus cycles as needed to interface with slow memory and IO devices.

The BS16# input allows the 386 to work with a 16-bit and/or a 32-bit data bus. If BS16# is asserted, the 386 will transfer data only on the lower half of the 32-bit data bus. If BS16# is asserted and a 32-bit operand is being read from a 16-bit-wide memory, the 386 will automatically generate a second bus cycle to read the second word. For misaligned transfers the 386 will also generate the required number of bus cycles if BS16# is asserted.

The ADS# signal will be asserted when valid addresses, BE signals, and bus cycle definition signals are present on the buses. The 386 address bus is not multiplexed, so an 8086-type ALE signal is not needed. However, in some 386 systems the ADS# signal is used to transfer the address to the outputs of external latches for a

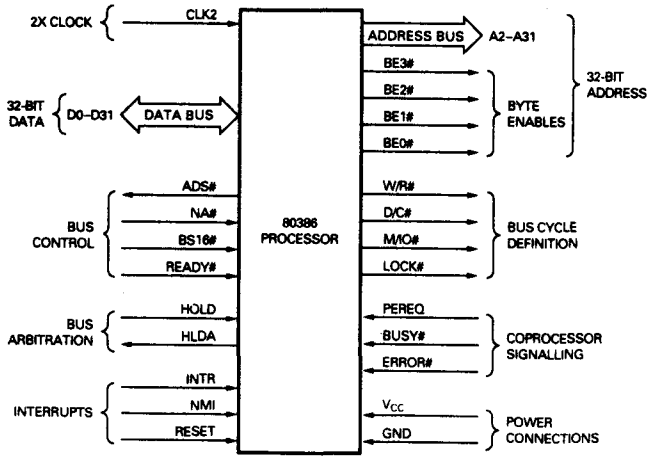


FIGURE 15-15 Signal groups of 386DX. (*Intel Corporation*)

scheme called *address pipelining*. The principle of address pipelining is that if an address is held on the outputs of external latches, the 386 can remove the old address from its address pins and output the address for the next operation earlier in the bus cycle. External control circuitry asserts the next address signal, NA#, to tell the 386 when to output the address for the next operation. Pipelined addressing is not usually necessary in a system with an SRAM cache, because the SRAM cache is fast enough that no wait states are needed.

To help you understand the relationship of some 386 signals, Figure 15-16 shows some 386 nonpipelined read cycles. As you can see, each read operation requires two states, T1 and T2. Note that READY# is made low during T2 so that no wait states are inserted. If the device being read is not fast enough to output data during T2 as required, READY# would be held high longer by external circuitry and a wait state would be inserted in the read cycle after T2.

Incidentally, the 386 contains a large amount of built-in self-test (BIST) circuitry. If the 386 BUSY# input is held low while RESET is held low, the processor will automatically test about 60 percent of its internal circuitry. The self-test requires about 2^{20} CLK2 cycles. If the 386 passes all tests, a "signature" of all 0's will be left in the EAX register.

Now that you have had a short trip around the 386 pins, the next step is to discuss how a 386 can be connected in a system.

386 System Connections and Interface Buses

THE URDA SDK-386 BOARD

A relatively low cost 386 system useful for prototyping 386-based instruments is the SDK-386 shown in Figure 15-17. This board is similar to the SDK-86 board we discussed in Chapter 7. Both boards are available from University Research and Development, Inc. in Pittsburgh, PA. The SDK-386 board contains a 12-MHz 386, 16 Kbytes of EPROM, 32 Kbytes of static RAM, a keyboard, and a 40-character LCD display. The board also has a serial port and software which allows programs to be developed on a PC-type computer and downloaded to the board for testing and debugging.

This board is useful as a simple, protected-mode learning tool, because the monitor program in ROM on the board runs the 386 in protected mode. The monitor runs as one task and user programs run as another task. A simple keypress allows the user to switch from the user task to the monitor. This feature is very useful for debugging programs and hardware. The documentation for the board shows how the descriptor tables, etc. are set up, and how user programs can call monitor procedures to interface with the keyboard and the display.

386 FULL SYSTEMS

Examples of more complex 386 systems are the IBM PS/2 Model 80, the Compaq SYSTEMPRO 386/33, and

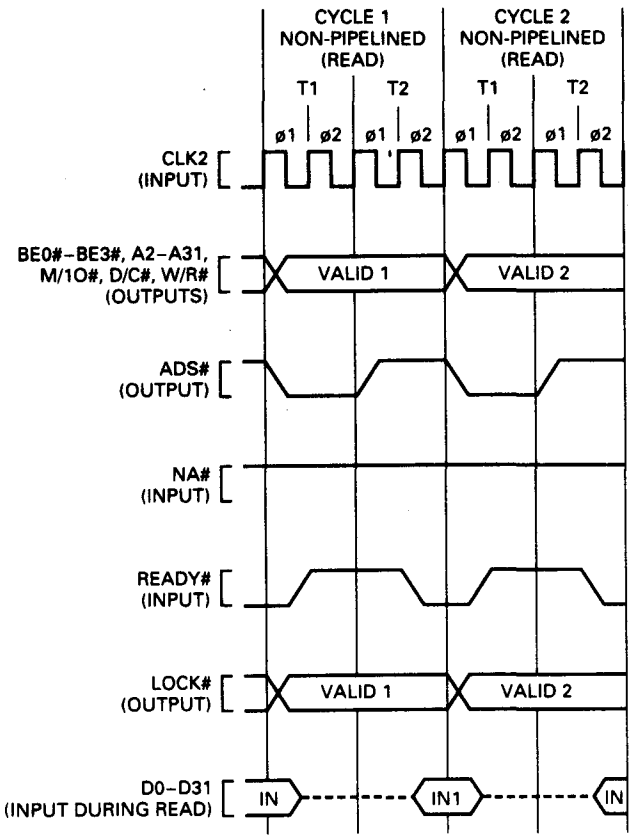


FIGURE 15-16 386 nonpipelined read cycles without wait states. (*Intel Corporation*)

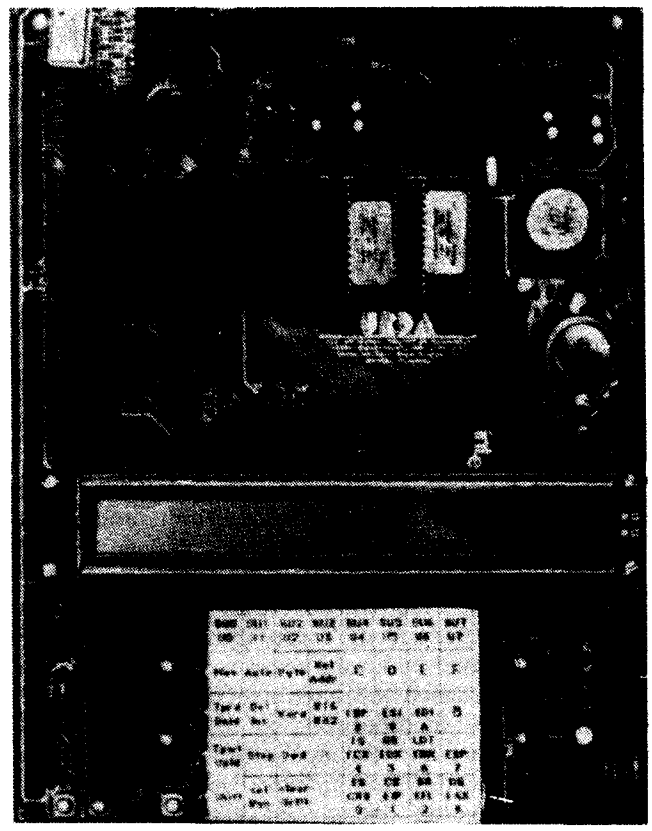


FIGURE 15-17 The SDK-386 prototyping board from University Research and Development Associates.

many similar machines. These systems typically have a megabyte or more of RAM, a 100-Mbyte hard-disk drive, a couple of floppy-disk drives, a VGA CRT controller, parallel ports, and serial ports. In most systems such as this, a 32-bit local data bus is used to interface with the SRAM cache and the DRAM main memory. The 32-bit data bus allows maximum transfer rate between memory and the 386. To interface with the on-board peripheral devices such as timers, priority-interrupt controllers, CRT controllers, serial ports, and parallel ports, the system uses a 16-bit local data bus. A separate I/O expansion bus is used to interface with peripheral boards such as disk controller cards, a network interface card, and a high-resolution graphics card.

Initially we wanted to show you some circuit diagrams for one of these 386 systems, but for several reasons we decided this was not practical. First of all, each of the 40 or 50 different 386 machines currently available uses a different circuit configuration. Second, the motherboards of the newest machines contain mostly large ASICs, which combine many functions in single devices. The Intel 82830, for example, contains an 8-channel DMA controller, a 20-input priority interrupt controller, four programmable timers, wait-state-generating circuitry, a complete DRAM refresh controller, and more. VLSI disk controllers and video controllers are now often included directly on the motherboard of 386 systems. The circuit diagram for a system built with these large ASICs look more like a block diagram than a circuit diagram and shows you little more than you already know about microcomputer structure. Also, most of the manufacturers consider their circuitry proprietary and are unwilling to release diagrams. What we decided would be useful here is to discuss the three bus standards commonly used to interface peripheral boards with the motherboards in these systems.

THE ISA BUS REVIEWED

In Figure 11-7 we showed you the bus used in the original IBM PC and the bus used in the IBM PC/AT. The PC/AT bus is one of the buses used in 386 systems and is now commonly called the *industry standard architecture* or ISA (pronounced e-sah) bus. The other bus schemes commonly used in 386 systems are the *extended industry standard architecture* or EISA (pronounced eye-sah) bus and IBM's *Micro Channel Architecture* or MCA bus.

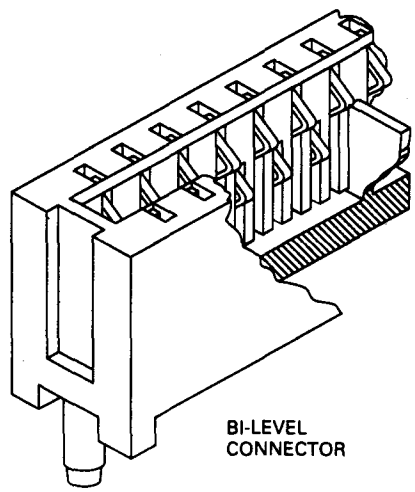
The ISA bus standard has the advantage that many peripheral boards have been developed for it, and competition has kept the price of these boards low. The ISA bus, however, has only 16 data lines and 24 address lines, so it cannot take full advantage of the 32-bit data bus and the 32-bit address bus of the 386. This reduces the speed at which data can be transferred on the bus. Most of the ISA-based 386 machines have partially solved this problem by incorporating up to 16 Mbytes of 32-bit-wide memory and perhaps a cache directly on the motherboard. Since this memory can be accessed directly without going through the peripheral bus, it can operate at the full speed of the 386. In these systems the ISA bus is used only to communicate with peripherals such as disk controller boards, CRT controller boards, network interface boards, etc. Since most of these boards transfer data only 8 bits or 16 bits at a time, the ISA bus limitations do not have an appreciable effect on the performance of a single-user system.

THE EISA BUS STANDARD

For high-performance applications such as network file servers, communication servers, and other multitasking/multiuser systems, the ISA bus does not allow fast-enough data transfer. Also, the ISA bus has no mechanism to arbitrate requests for bus use by "smart" peripheral boards. Systems designed for these applications usually use an EISA or an MCA bus.

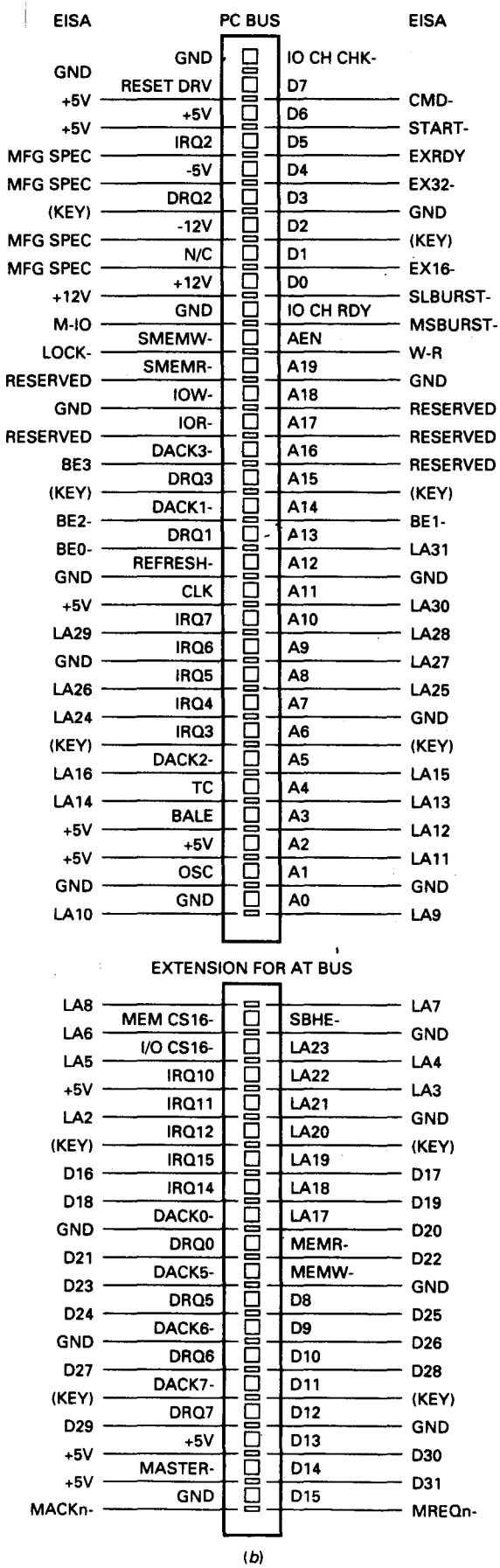
As the name indicates, the EISA bus is an extension of the ISA bus to accommodate the needs of the 386 and 486 processors and multimaster systems. It was developed by nine companies as an alternative to IBM's MCA bus standard. The EISA bus uses edge connectors of the same physical size as the ISA bus so that either ISA or EISA boards can be inserted in a slot. However, the EISA connectors have two levels of contacts, as shown in Figure 15-18*a*. If an ISA-based board is inserted, it will go in the connectors only far enough to reach the top level of contacts which contain the ISA bus signals. A notch cut in EISA boards allows them to go into the connectors far enough to also contact the lower level of contacts. These lower contacts contain the additional EISA signals. Figure 15-18*b* shows how these added signal lines are interspersed between the ISA signals. Note the additional address (labeled LA), data, BE#, power, ground, and control pins. The pin positions labeled KEY represent thin slots cut in the PC board so that it aligns properly when inserted. The pins labeled MFG SPEC are used to output a code which identifies the type of board to aid in system configuration during bootup.

In a multiprocessor microcomputer system a processor that can take over the bus to transfer data is called a *bus master*. A board which can take over the bus for a



(*a*)

FIGURE 15-18 (*a*) Two layers of contacts in EISA bus connectors. (*See also next page.*)



(b)

FIGURE 15-18 (*Continued*) (*b*) Pin assignments, EISA bus.

DMA operation is called a *DMA slave*. The EISA bus supports up to six bus masters and 8 DMA slaves. The MACKn and MREQn lines on the EISA bus are used to arbitrate bus requests by multiple masters. These signals are not bused. An individual trace runs from each of these pins to the arbitration logic on the motherboard. The n in these signal names represents the slot number in the system. When a master wants to use the buses, it asserts its MREQ line. If the buses are free and that master is the highest-priority master requesting use of the buses, the arbitration circuitry will assert the MACK signal connected to that master. The master will use the bus for its data transfer. DMA slaves issue requests through the DREQ lines on the bus and receive control from the arbitration circuitry through the DACK lines.

Another feature of the EISA bus is that its interrupts can be individually programmed as edge-triggered for compatibility with ISA boards or level-triggered so that they are less susceptible to noise spikes and they can be shared by several sources. EISA boards use level-triggered interrupts, which can be pulled low by any one of several sources. When the CPU detects an interrupt, it polls each board or device to determine the source of the interrupt.

To help implement an EISA bus in a system, Intel makes the 82358 Bus Controller, the 82357 Integrated System Peripheral, and the 82355 Bus Master Interface Controller. Consult the data sheets for these devices to get more detailed information about the operation of an EISA bus.

THE MICROCHANNEL ARCHITECTURE BUS

IBM's MicroChannel Architecture Bus contains the same types of signals and accomplishes the same functions as EISA, but the two are completely incompatible. MCA boards are smaller and use different edge connectors. Figure 15-19, page 522, shows the MCA bus connector types used in the IBM PS/2 Model 80.

The MCA bus is designed to work with peripheral boards that transfer data in 8-bit, 16-bit, or 32-bit words. For 8-bit peripheral boards, just a 46-pin edge connector is used. For 16-bit peripheral boards, an additional 12-pin connector is used. One of the 16-bit slots also has a 20-pin video extension connector. This slot can be used for an 8514/A high-resolution graphics card. For 32-bit boards an additional 44-pin connector is used in place of the 12-pin connector used on 16-bit slots. Each of the 32-bit slots also has an 8-pin "matched-memory" extension connector.

The Model 80 has five 16-bit MCA slots and three 32-bit slots. One of the 16-bit slots is used for an ESDI hard-disk controller. The 32-bit slots can be used for 32-bit memory boards or other 32-bit peripherals. The signals on the matched-memory extension of the 32-bit slots allow the processor to interrogate memory boards to see if they are designed to transfer data faster than the basic bus rate. If they are, other signals on the connector manage transfers.

The MCA bus uses a distributed scheme to arbitrate bus requests by up to 16 bus masters and DMA request sources. To request the bus, one or more masters assert

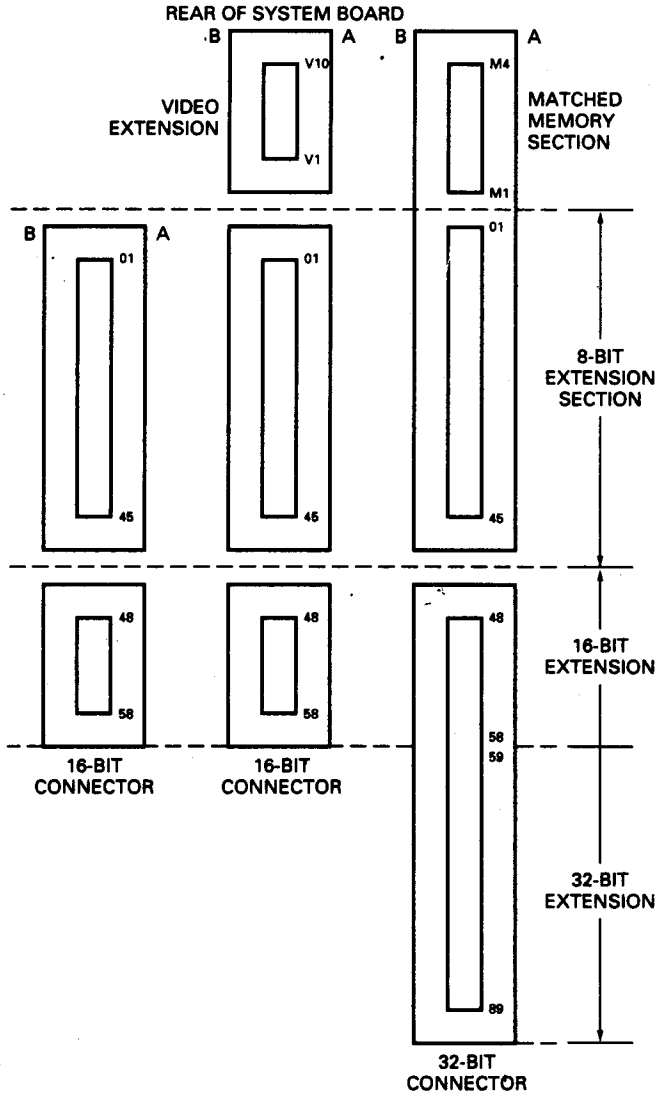


FIGURE 15-19 IBM's MicroChannel Architecture bus connector types.

the PREEMPT line to the central control circuitry low. At the appropriate time the control circuitry drives the ARB/GNT line high. The arbiter on each master then asserts its arbitration code on the ARB0–ARB3 lines. If an arbiter sees a code that is lower than its code, it removes its arbitration signals. This means that the master with the lower arbitration code assumes control of the bus. To signal the arbitration is complete, the central control point asserts the ARB/GNT signal low. Incidentally, the interrupt lines on the MCA bus are level-triggered.

Now that you have had a brief introduction to the system connections and buses used in 386 systems, let's take a look at the internal architecture of the device and talk about the different 386 operating modes.

Real Operating Mode

A 386 can operate in real mode, protected mode, or a variation of protected mode called *virtual 8086* mode. After a reset the 386 operates in real address mode. In this mode it functions basically as a fast 8086 or real-mode 80286. The register set for the 386 in real mode in a superset of the 8086 and 80286 real-mode register sets. As shown in Figure 15-20, the 32-bit general-purpose registers are referred to as extended AX or EAX, EBX, ECX, EDX, etc. Instructions can, for example, refer to AL, AH, AX, or EAX. The assembler automatically codes the instruction for the register size referred to in an instruction.

The 386 in real mode computes memory addresses using the same segment base and offset mechanism used by the 8086. For this mode only the selectors or visible parts of the segment registers are used. Note that the 386 has two additional data segment registers, FS and GS, so programs can have up to four data segments. The length of segments in 386 real mode is fixed at 64 Kbytes, and any attempt to access a location outside a segment will cause a type 13 exception.

The address range of 386 real mode is limited to 1 Mbyte, so address lines A20–A31 are normally all low. The only exception to this is that during a reset these address lines are all made high to access the boot ROM at the highest locations in the 32-bit address space of the 386. As soon as the boot-ROM code does a far jump or call, the A20–A31 lines will go low and stay low as long as the 386 is in real mode. A 386 in real mode uses the address space 00000–003FFH for the interrupt-vector table and services interrupts in the same way as an 8086 does.

One new feature of the 386 is the debug registers shown in Figure 15-20. A software debugger can load breakpoint addresses in these registers to aid in debugging. A 386 can be instructed to "break" when the address unit in the processor computes a linear address which matches one of the addresses in the debug registers. The older method of setting a breakpoint involved replacing an instruction with a breakpoint instruction such as INT 3. This method, of course, cannot be used to debug code in ROM, but the breakpoint register method can because it does not depend on modifying code bytes.

The 32-bit EFLAGS register in the 386 is an extension of the 16-bit registers in the 8086 and 80286. For future reference the upper right corner of Figure 15-20 shows the names of the bits in the EFLAGS register, but in real mode only the lower 16 bits have meaning.

The final real-mode registers to note in Figure 15-20 are the control registers CR0–CR3. The lower 16 bits of CR0 correspond to the machine status word (MSW) of the 80286. As with an 80286, a 386 is switched to protected-mode operation by setting the LSB of this register to a 1. Register CR1 is reserved by Intel, and registers CR2 and CR3 are used for paged mode functions, which we discuss later.

386 Protected-Mode Operation

INTRODUCTION

The real power of a 386 lies in its protected-mode and virtual 8086-mode features. These features are designed in a very versatile way, so that almost any conceivable

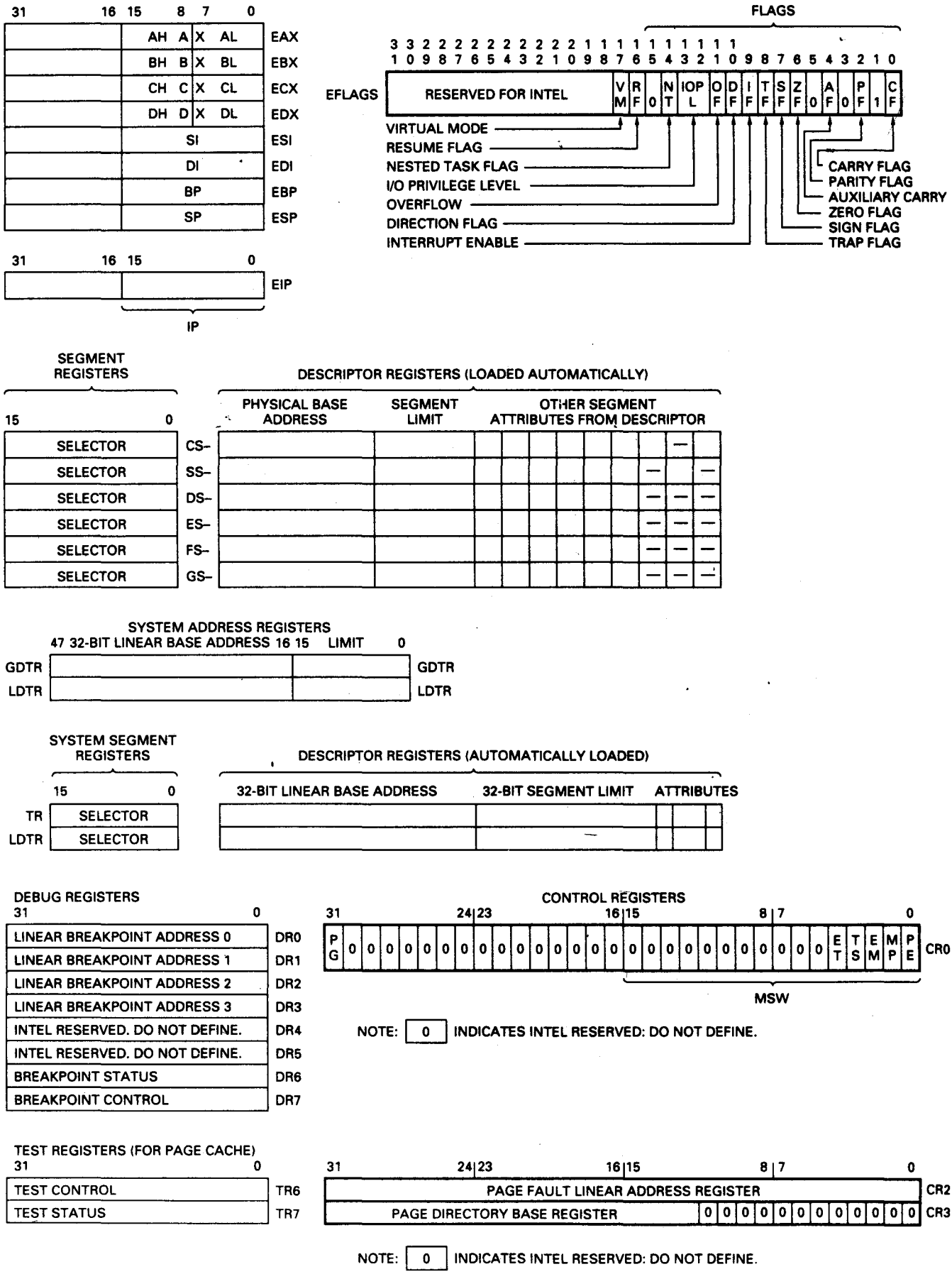


FIGURE 15-20 Intel 386 microprocessor register set. (*Intel Corporation*)

operating system or program can be implemented on a 386. The problem with this versatility is that it leads to an almost unbelievable amount of detail in a complete description of how the 386 operates in these modes. In reality, unless you are writing a 386-based operating system, you can probably live a very happy life without knowing all these details. In the following sections we have tried to give just enough details so that you can understand the basic protected-mode operation of a 386, how its features fit the needs of a multiuser/multitasking operating system, and how programs are written for a 386. If you need to know all the minute details, consult the Intel 80386 Programmer's Reference Manual and the Intel 80386 System Software Writer's Guide.

As you read through the following sections, the key concepts you should try to fix in your mind are: how a 386 computes physical addresses in segments-only mode and in paged mode, how a 386 provides protection for operating system code and protection for user tasks, the basic operation of a gate, the protected-mode interrupt response, the task switch process, and the operation of the "flat" system model.

SEGMENTATION AND VIRTUAL MEMORY

As we said in the preceding section, a 386 is switched from real mode to protected mode by setting the LSB of the CR0 register. The virtual memory addressing scheme of a 386 in protected mode is very similar to that of the 80286 we described earlier, except that 386 segments can be much larger and an optional paging mechanism allows segments to be divided into 4-Kbyte pages for faster swapping in and out of physical memory.

In protected mode each 386 address consists of a 16-bit segment selector and a 32-bit offset. As we described earlier in a section on virtual memory, the selector points to a descriptor for the segment in a table of descriptors and the offset specifies the location of the desired code or data in the segment. Using a 32-bit offset value means that segments can be anywhere from 1 byte in length to 2^{32} or about 4 Gbytes in length.

Figure 15-21 shows the format for 386 segment selectors and how these selectors are used to access a descriptor in a descriptor table. The 13-bit index part of this selector is multiplied by 8 and used as a pointer to the desired descriptor in a descriptor table. The index value is multiplied by 8 because each descriptor requires 8 bytes in the descriptor table. Among other things the descriptor contains the physical base address for the segment. The MMU adds the base address from the descriptor to the effective address or offset part of the logical address from the instruction to produce the physical memory address.

There are two major categories of descriptor table in

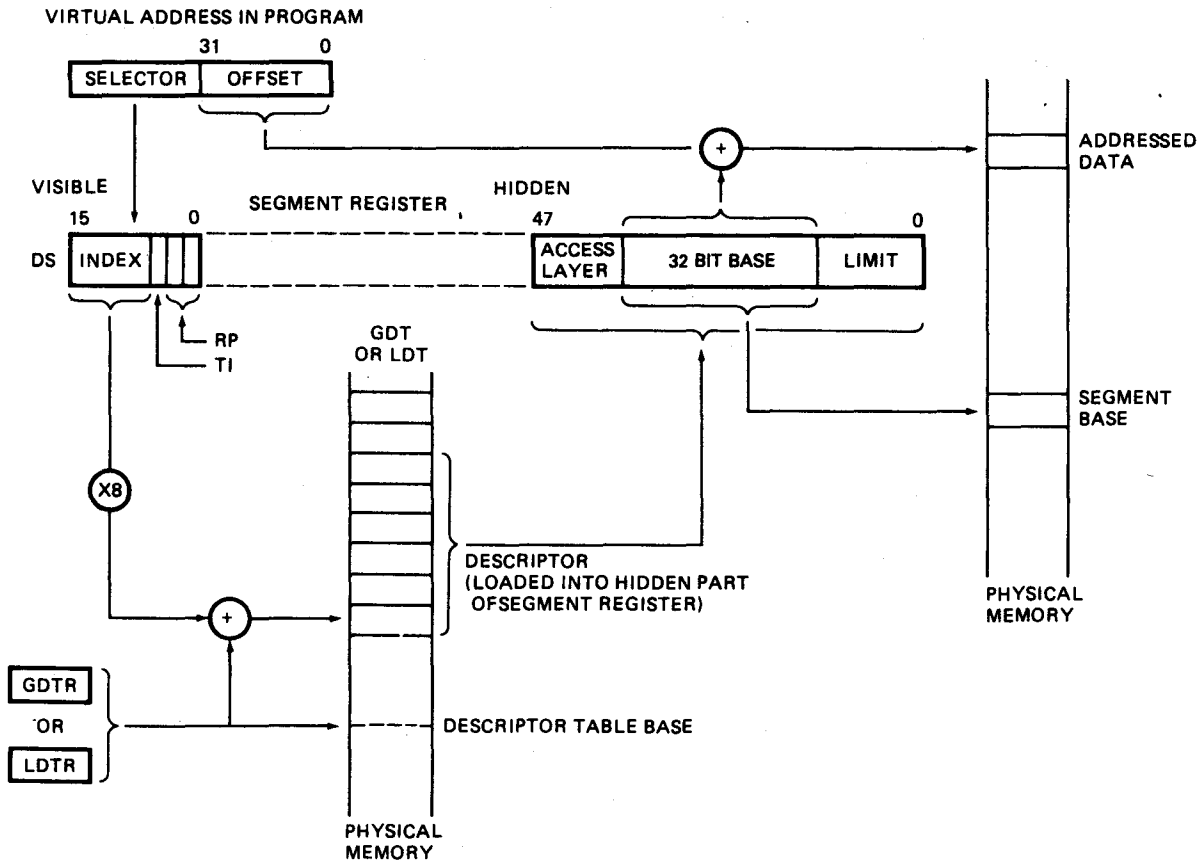


FIGURE 15-21 Diagram showing how the 386 uses a selector to access a descriptor in a descriptor table and how it computes the physical (linear) address. (*Intel Corporation*)

a 386 system, global and local. A system has only one *global descriptor table* or GDT. The GDT contains, among other things, the segment descriptors for the operating system segments and the descriptors for segments which need to be accessed by all user tasks. A *local descriptor* table or LDT is set up in the system for each task or closely related group of tasks. Figure 15-22 shows, in diagram form, how this works. Tasks share a global descriptor table and the memory area defined by the descriptors in it. Each task can have its own local descriptor table and memory area defined by the descriptors in it. Setting up individual LDTs protects tasks from each other because one task cannot access the LDT of another task.

If the *table indicator* bit (bit 2) of a segment selector is a 0, then the upper 13 bits will index a segment descriptor in the global descriptor table. If the TI bit of the selector is a 1, then the upper 13 bits of the selector will index a segment descriptor in a local descriptor table.

The least significant 2 bits of a segment selector, the *requested privilege level* or RPL bits, are part of the 386's built-in protection features, which we discuss later. For now, let's take a closer look at segment descriptors.

Figure 15-23*a* and Figure 15-23*b* show the formats for the 386 segment descriptors and the access rights byte of the descriptors. First notice in the descriptor that 32 bits are set aside for the segment's physical base address and 20 bits are set aside for the size or limit of the segment. If you remember that we said 386 segments can be up to 2^{32} bytes long, you may wonder why only 20 bits are set aside here for the size of the segment. The answer to this is that if the granularity or G bit in the descriptor is a 0, the 20-bit limit value represents the length of the segment in bytes. With a 0 value in the

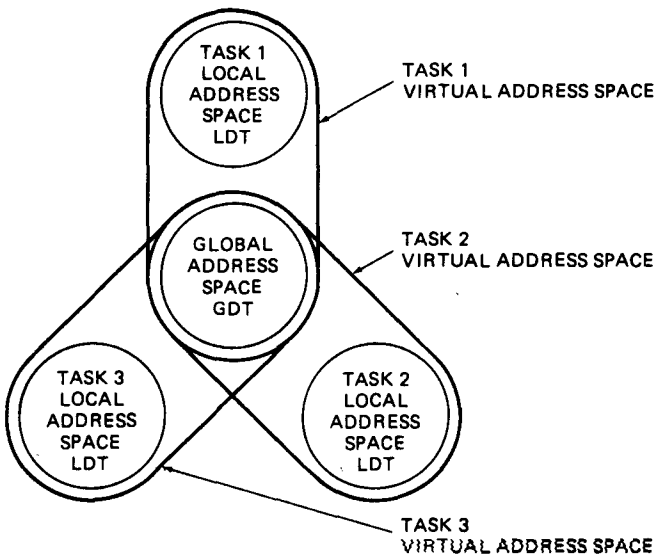
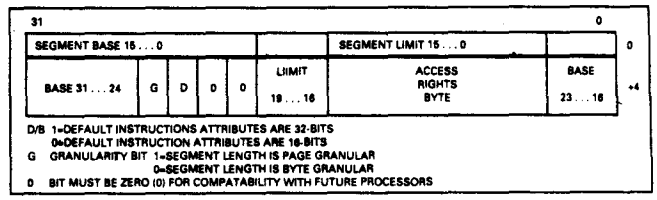


FIGURE 15-22 Diagram showing how tasks can be isolated from each other by having separate local descriptor tables but can share a common global descriptor table. (*Intel Corporation*)



(*a*)

	BIT POSITION	NAME	FUNCTION	
	7	PRESENT (P)	P=1 SEGMENT IS MAPPED INTO PHYSICAL MEMORY. P=0 NO MAPPING TO PHYSICAL MEMORY EXITS. BASE AND LIMIT ARE NOT USED.	
	6 - 5	DESCRIPTOR PRIVILEGE LEVEL (DPL)	SEGMENT PRIVILEGE ATTRIBUTE USED IN PRIVILEGE TESTS.	
	4	SEGMENT DESCRIPTOR (S)	S=1 CODE OR DATA (INCLUDES STACKS) SEGMENT DESCRIPTOR S=0 SYSTEM SEGMENT DESCRIPTOR OR GATE DESCRIPTOR	
TYPE FIELD DEFINITION	3	EXECUTABLE (E)	E=0 DATA SEGMENT DESCRIPTOR TYPE IS:	IF DATA SEGMENT (S = 1, E = 0)
	2	EXPANSION DIRECTION (ED)	ED=0 EXPAND UP SEGMENT, OFFSETS MUST BE ≤ LIMIT. ED=1 EXPAND DOWN SEGMENT, OFFSETS MUST BE > LIMIT.	
	1	WRITEABLE (W)	W=0 DATA SEGMENT MAY NOT BE WRITTEN INTO. W=1 DATA SEGMENT MAY BE WRITTEN INTO.	
	3	EXECUTABLE (E)	E=1 CODE SEGMENT DESCRIPTOR TYPE IS:	IF CODE SEGMENT (S = 1, E = 1)
	2	CONFORMING (C)	C=1 CODE SEGMENT MAY ONLY BE EXECUTED WHEN CPL ≥ DPL AND CPL REMAINS UNCHANGED.	
	1	READABLE (R)	R=0 CODE SEGMENT MAY NOT BE READ. R=1 CODE SEGMENT MAY BE READ.	
	0	ACCESSED (A)	A=0 SEGMENT HAS NOT BEEN ACCESSED. A=1 SEGMENT SELECTOR HAS BEEN LOADED INTO SEGMENT REGISTER OR USED BY SELECTOR TEST INSTRUCTIONS.	

(*b*)

FIGURE 15-23 (*a*) 386 descriptor format. (*b*) Access rights byte format for code and data segment descriptors. (*Intel Corporation*)

G bit, then, a segment can be up to 1 Mbyte in length. If the G bit in the descriptor is a 1, the 20-bit limit value represents the length of the segment in 4-Kbyte blocks. The maximum limit value of 1,048,576 blocks × 4 Kbytes/block then gives a maximum segment length of 4 Gbytes. If an attempt is made to access a location outside the specified limit for a segment, a type 5 exception will be produced. This mechanism prevents a program from accessing memory outside its defined segments.

Byte 5 of a descriptor, the access byte, contains information about the privilege level, access, and type of the segment. To give you an idea of the kind of information contained in the access byte of a descriptor, Figure 15-23*b* summarizes the meanings of the bits in the access bytes of code segment and data segment descriptors. Skim through the descriptions to get an overview. Note the P bit, which is used to indicate whether the segment is present in physical memory, the privilege-level bits, which specify the privilege level that a program must have to access the segment, and the A bit, which is set if the segment has been accessed. The operating system can periodically read and reset the A bit to determine how often the segment has been accessed. A segment which has not been recently used can be swapped out to disk when space for a new segment is needed.

When a program attempts to access a segment, the selector for the segment is loaded into the visible part of the segment register. To access a data segment, for

example, the selector might be loaded into the visible part of the DS, ES, FS, or GS register. When the selector is loaded into the visible part of the segment register, the descriptor for the segment is automatically loaded into the hidden part of the segment register or segment descriptor cache, as it is commonly called. If the privilege level of the selector and the privilege level of the current code segment is not as high (or is higher than) the privilege level of the descriptor, an exception will be produced and the access will not be allowed. If the privilege level is high enough, the P bit in the descriptor will be checked to see if the segment is present in physical memory. If the segment is not present, a segment-not-present (type 11) exception will be generated and the exception handler will read the segment in from disk to physical memory. Once the segment is in physical memory, the address unit computes the physical addresses as needed to access the data words in the segment. As shown in Figure 15-21, the offset from the original address is added to the segment base address from a descriptor to form a linear address. For a 386 operating in segments-only mode, this linear address is the physical address that will be output on the address and BE lines to memory.

To complete the general picture of how a 386 manages virtual segments, we simply need to show you how it keeps track of where the descriptor tables are in memory. The 386 keeps the base addresses and limits for GDT and LDT descriptor tables currently being used in internal registers. The *global descriptor table register* (GDTR) shown in the middle of Figure 15-20 is used to hold the 32-bit base address and limit for the global descriptor table. This register is initialized with a load global descriptor table register (LGDT) instruction when the system is booted. The *local descriptor table register* (LDTR) shown in Figure 15-20 is used to hold the base address and limit of the local descriptor table for the task currently being executed. The LLDT instruction is used to load this register. The LLDT instruction can be executed only by programs executing at the highest privilege level. Therefore, unless a task is operating at the highest privilege level, it cannot intentionally or maliciously access the local descriptor table of another task. Task switching is usually handled by the operating system kernel, which operates at the highest-priority level.

386 SEGMENT PRIVILEGE LEVELS AND PROTECTION

When an attempt is made to access a segment by loading a segment selector into the visible part of a segment register, the 386 automatically makes several checks. First of all, it checks to see if the descriptor table indexed by the selector contains a valid descriptor for that selector. If the selector attempts to access a location outside the limit of the descriptor table or the location indexed by the selector in the descriptor table does not contain a valid descriptor, then an exception is produced.

The 386 also checks to see if the segment descriptor is of the right type to be loaded into the specified segment register cache. The descriptor for a read-only data segment, for example, cannot be loaded into the SS register, because a stack must be able to be written to. A selector for a code segment which has been designated "execute only" cannot be loaded into the DS register to allow reading the contents of the segment.

If all these protection conditions are met, the limit, base, and access rights byte of the segment descriptor are copied into the hidden part of the segment register. The 386 then checks the P bit of the access byte to see if the segment for that descriptor is present in physical memory. If it is not present, a type 11 exception is produced. The exception-handler procedure for this exception will swap the segment into physical memory, set the P bit in the descriptor, and restart the interrupted instruction.

After a segment selector and descriptor are loaded into a segment register, further checks are made each time a location in the actual segment is accessed. An attempt to write to a code segment or a read-only data segment, for example, will cause an exception. Also, the limit value contained in the segment descriptor is used to check that an address produced by program instructions does not fall outside the limit defined for the segment.

User tasks can be protected from each other in a 386 system by giving each task its own local descriptor table. The LDT register, which points to a user's local descriptor table, can only be changed with the LDTR instruction or by a task switch. The LDTR instruction can be executed only at the highest privilege level, which is usually reserved for the operating system. Likewise, a switch from one user task to another is done by the operating system at the highest privilege level, so user tasks operating at lower privilege levels cannot cause switches to other user tasks. Also, because of limit checking, a task cannot accidentally or intentionally access descriptors in another task's local descriptor table.

System software, such as the operating system kernel, is protected from corruption in several ways. One way we have already mentioned is that code segments can be made "execute only" so that they cannot be written to. The second and most important way that the operating system can be protected is with privilege levels. Figure 15-24 illustrates how a 386 protected-mode system can be set up with four privilege levels. As we mentioned before, the operating system kernel is assigned the highest privilege level, which is privilege level 0. System services such as BIOS procedures might be run at privilege level 1, and custom device drivers, etc. might operate at privilege level 2. Application programs and user tasks are usually operated at the privilege level 3, the lowest level.

The privilege level for a segment is represented by bits 5 and 6 of the access byte in the segment descriptor. (See Figure 15-23*b* for access byte format.) These 2 bits are referred to as the *descriptor privilege level* or DPL. This privilege level is established when the program is built.

The privilege level of an executing task is represented by the DPL bits in the access byte of the descriptor currently in the CS descriptor cache. This privilege level is referred to as the *current privilege level* or CPL.

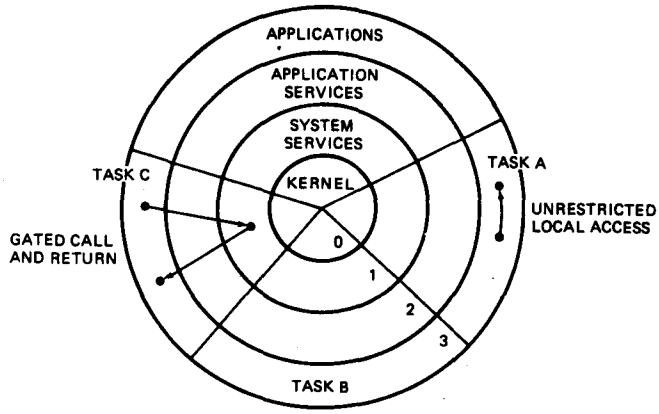


FIGURE 15-24 Diagram showing how a 386 system can be set up with four privilege levels. (*Intel Corporation*)

When a program needs to access a data segment, it does so by loading a segment selector into, for example, the visible part of the DS register. The privilege level encoded in the least significant bits of this selector is referred to as the *requesting privilege level* or RPL.

To successfully access a segment, both the RPL and the CPL must be a number less than or equal to the DPL of the segment. In other words, the privilege level of the currently executing task and the privilege level of the requesting selector must both be greater than or the same as the privilege level of the desired segment in order for access to be granted. If these conditions are not met, then an exception will be generated. The point here is that normally a task cannot directly access a segment which has a higher DPL.

CALL GATES

The question that might come to mind at this point is, If a task cannot access a segment with a more privileged DPL, how can user programs access the operating system kernel, BIOS, or utility procedures in segments which have more privileged DPLs? The answer to this is that a procedure located in a segment which has a higher privilege level can be called indirectly through a special structure called a *gate*. There are four types of gates: call, trap, interrupt, and task. For now, we will just describe how a call gate operates.

A gate is simply a special type of descriptor. Gate descriptors are put in the GDT or in an LDT, just as segment and other descriptors are. When a program does a call to a procedure in another segment, the selector for that segment's call gate is loaded into the CS register, and the call gate descriptor is loaded into the hidden part of the CS register. The call gate descriptor contains a selector which points to the descriptor for the segment where the procedure is actually located. The call gate descriptor also contains the offset of the called procedure in its segment.

If the call is determined to be valid, then the selector from the call gate and the corresponding segment descriptor will be loaded into the CS register. The processor then uses the base address from the segment descriptor and the offset value from the call gate descriptor to compute the physical address of the called procedure. Therefore, the call is done indirectly, through the call gate descriptor, rather than directly through a segment descriptor.

This indirect access has two major advantages. First, this approach permits another level of privilege checking before access to the procedure in the higher-privileged segment is allowed. The privilege level of the calling program is compared with the privilege level specified in the call gate. If the privilege level of the calling program is lower than the privilege level specified in the call gate, the access will not be allowed. If, for example, the DPL in the call gate descriptor is 2, a level 2 program can use the call gate to call a privilege level 1 procedure, but a level 3 program cannot.

Another advantage of the indirect call gate approach is that user programs cannot accidentally enter higher-privileged segments at just any old point. If they are going to enter at all, they must enter at the specific offsets contained in the call gate descriptors. This is similar to the type of protection provided by using software interrupts to call BIOS and DOS functions instead of calling them directly.

I/O PRIVILEGE LEVELS

When a 386 is operating in protected mode, the 386 has two mechanisms for protecting I/O ports. The first mechanism involves the I/O privilege-level bits in the 386 EFLAGS register shown in Figure 15-20. Only the operating system or a procedure operating at a privilege level 0 can set these IOPL bits. In order to execute the IN, INS, OUT, OUTS, CLI, and STI instructions, the CPL of a procedure or task must be the same or a lower number than IOPL represented by these bits. If a procedure does not meet the IOPL test, a privilege-level exception will be generated.

The second mechanism for protecting ports from unauthorized access is an optional I/O permission bit map which allows ports to be associated only with specific tasks. If this feature is used, a map is set up for each task. Each bit in the map represents a byte-wide port address, so 16-bit ports use 2 bits each and 32-bit ports use 4 bits each. A 0 in a map bit means the port is available to the task.

When a task attempts to access a port, the 386 first compares the CPL of the task with the IOPL. If the access passes the IOPL test and an I/O bit map is in force, the 386 will then check the map bits corresponding to the addressed port. If the map has a 0 in the bit(s) for that port, access will be granted. If not, an exception will be generated. Incidentally, when a 386 is operating in real address mode, none of the port protection mechanisms are in effect.

INTERRUPT AND EXCEPTION HANDLING

For operation in protected mode, gate descriptors for the interrupt and exception procedures are kept in a special descriptor table called the interrupt descriptor table or IDT. This table can be located anywhere in memory. During initialization the base address and

limit for the interrupt descriptor table are loaded into the *interrupt descriptor table register* (IDT) shown in Figure 15-20 with an LIDT instruction.

When an interrupt or exception occurs, its type is multiplied by 8 and added to the IDT base address in the IDT register. The result is a pointer to a gate descriptor in the interrupt descriptor table. The gate here can be an interrupt gate, a trap gate, or a task gate.

An interrupt gate, for example, contains a selector for the segment where the interrupt procedure is located, not the base address of the segment. The reason for this is so that the privilege level can be checked before access to the interrupt procedure is granted. If the CPL is high enough, then the selector from the gate will be loaded into the CS register and used to access the descriptor for the segment containing the interrupt procedure. The segment descriptor can be in the LDT or GDT. The 32-bit offset from the gate will be added to the base address from the descriptor to produce the linear address for the actual interrupt procedure. This is basically the same mechanism we described previously for the operation of a call gate, except that at the end of the procedure an IRET instruction is used instead of an RET. Incidentally, an interrupt procedure that needs to be accessible from any privilege level is put in a code segment that is made "conforming" by setting bit 2 in the access byte of its descriptor.

TASK SWITCHING

In a multiuser operating system each user's program can be set up as a separate task. When a user's time slice is up, the operating system switches execution from the current user's task to the next user's task. A similar process takes place in a single-user system which is operating in a multitasking mode. As we pointed out earlier, one of the main concerns in a multitasking system is saving the state or context of a task so that it will continue execution properly when it gets another time slice.

Each task in a 386 protected-mode system is assigned a *task state segment* or TSS. Figure 15-25 shows the format for a 386 TSS. As you can see, the TSS holds copies of all registers and flags, the selector for the task's LDT, and a link to the task state segment of the previously executing task. Descriptors for the task state segments are kept in the global descriptor table, where they can be accessed by the operating system during a task switch. The *task register* (TR) in the 386 holds the selector and the descriptor for the task state segment of the currently executing task. The *load task register* (LTR) instruction can be used to load the task register with the selector and segment descriptor for a specific task, but during a task switch the task register is automatically loaded with the selector and descriptor for the new task.

A task switch may be done in any one of four ways:

1. A long jump or call instruction contains a selector which points at a task state segment descriptor. The call instruction is used if a return to the previously executing task is desired. A jump instruction is used if a return to the previously executing task is not desired. This is the simplest method and can be easily implemented by the operating system kernel at the end of a time slice.
2. The selector in a long jump or call instruction points to a task gate. In this case the selector for the destination TSS is in the task gate. The indirect mechanism here is similar to that we described above for call gates and has the same advantages regarding privilege levels and protection.
3. An interrupt occurs, and the interrupt selector points to a task gate in the interrupt descriptor table. The task gate contains the selector for the new task state segment. If the access passes all the privilege level tests, the selector and descriptor for the interrupt task will be loaded into the task register. The nested task (NT) bit in the EFLAGS register will be set.
4. An IRET instruction is executed with the NT bit in the EFLAGS register set. Complex interrupt procedures are often written and managed as separate tasks. The IRET instruction uses the back link selector in the task state segment to return execution to the interrupted task. This is similar to the way the IRET instruction works in real-mode operation.

We don't have space or inclination to explain the details of all the possible task switch scenarios, but we will make a few comments about the CALL/JMP method.

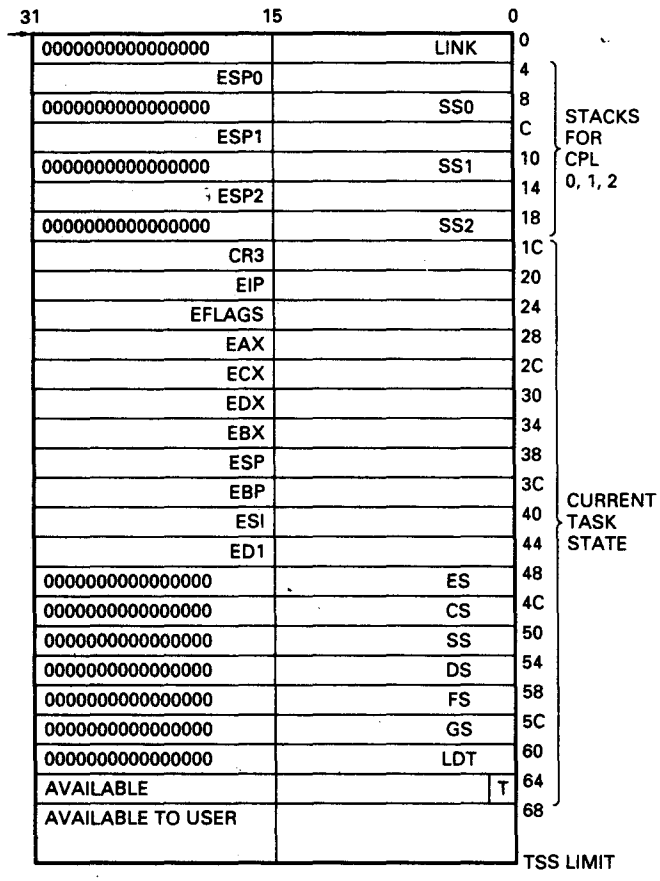


FIGURE 15-25 386 task state segment format. (*Courtesy Intel Corporation*)

When a far CALL or JMP is executed to switch tasks, the privilege levels are first checked. As with any far call or far jump instruction, the RPL of the CALL selector and the CPL of the executing program must both be less than or equal to the DPL of the desired segment, or an exception will be produced.

Assuming proper privilege levels, the 386 will check if the task state segment for the new task is present in physical memory and generate a not-present exception if it is not. If necessary, the exception handler will load the TSS for the new task.

The 386 then copies all the register values for the current task to its task state segment. The value copied for the EIP is the offset of the next instruction after the one that caused the task switch.

At this point the old TSS is no longer needed, so the 386 loads the task register with the selector and the descriptor for the TSS of the new task. The 386 then automatically copies all the values for the new task from its TSS to the 386 registers. Execution then continues using the segment and offset values copied from the TSS.

In a multiuser/multitasking system, the operating system might use a JMP instruction to switch from the operating system task to a user task. A clock tick will interrupt the processor at the end of the time slice. If the interrupt descriptor table contains a task gate which points to the operating system task, then the state of the current task will be saved in its TSS, and execution will switch to the operating system task. The operating system can use another JMP instruction to switch to the next user's task.

PAGING MODE

The protected-mode segmentation and virtual memory scheme we described for the 386 in the preceding section is essentially the same as that for the 80286. The main difference is that 386 segments can be as large as 4 Gbytes, instead of only 64 Kbytes. The designers of the 386 realized that the time required to swap very large segments in and out of physical memory would be too long, so they added an optional paging mechanism to the design of the 386. The paging mechanism allows segments to be divided into 4-Kbyte pages for faster swapping.

The 386 is switched into paging mode by setting the MSB of the CR0 register with a simple MOV CR0, EAX-type instruction. In this mode the paging unit in the 386 uses the linear address, computed by the segmentation unit as described above, to produce the physical address. Figure 15-26 shows how this paging scheme works. To help fix it in your mind, we will first explain the paging scheme from the bottom up and then from the top down.

The Intel data sheets refer to each 4-Kbyte page in physical memory as a *page frame*. The least significant 12 bits of the linear address from the segmentation unit represent the offset of the desired data word within a 4-Kbyte page frame. The 32-bit base addresses and some other information for up to 1024 page frames are kept in a *page table* in memory. For future reference Figure 15-27*a* shows the details of the 4-byte entry placed in a

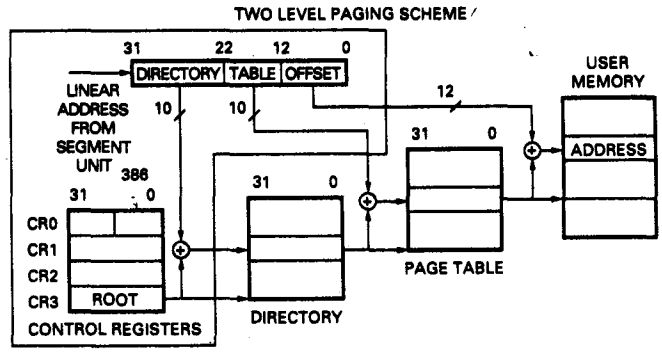
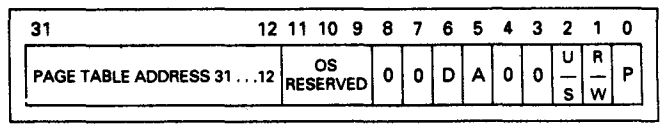


FIGURE 15-26 Diagram showing how a 386 computes physical addresses when paging mode is enabled.

page table for each page frame. The 10 address bits, A12–A21, from the linear address are used to select the desired entry in one of the page tables.

A system can contain up to 1024 page tables. The 32-bit base addresses and some other information for the page tables are kept in another table called the *page directory*. The format for the 4-byte entries in the page directory is the same as the format shown for a page table entry in Figure 15-27*a*. The 10 address bits, A22–A31, are used to select the desired entry in the page directory. The 32-bit base address for the page directory is kept in control register 3 (CR3) in the 386.

Looking at this from the top down then, CR3 points to the base of the page directory and linear address bits A22–A31 point to one of 1024 possible entries in the page directory. The selected entry in the page directory points to the base address of one of up to 1024 page tables, and linear address bits A12–A21 point to one of the entries in the selected page table. The selected entry in the page table contains the 32-bit base address of the desired 4-Kbyte page frame. Linear address bits A0–A11 are used to access the desired code or data word in the selected page frame. These bits are added to the base address from the page table entry to produce the physical



(*a*)

U/S	R/W	PERMITTED LEVEL 3	PERMITTED ACCESS LEVELS 0, 1, OR 2
0	0	NONE	READ/WRITE
0	1	NONE	READ/WRITE
1	0	READ-ONLY	READ/WRITE
1	1	READ/WRITE	READ/WRITE

(*b*)

FIGURE 15-27 (*a*) Format for 386-page directory and page table entries. (*b*) Access rights produced by combinations of R/W and U/S bits in 386-page table entries.

address that will be output to memory. The maximum amount of memory represented by this structure is 1024 page tables × 1024 pages/page table × 4096 bytes/page, or about 4 Gbytes, the full 32-bit address space of the 386. A system can be set up with just one page directory, but a more common practice is to give each task its own page directory and, thereby, its own set of page tables. Later we show you how the 386 task switch mechanism makes provisions for easily switching page directories.

As we said before, the page directory is located in memory and the page tables are located in memory. To avoid having to read page directory entries and page table entries from memory tables during each memory access, the 386 maintains a special cache called a *translation lookaside buffer* or TLB. The TLB is a four-way set-associative cache which holds the page table entries for the 32 most recently used pages. (Refer to the discussion of caches in Chapter 11 if the term set-associative is a little rusty in your mind.) When the 386 generates a linear address, the upper 20 bits of that address are compared with the tags for the 32 entries in the TLB. If there is a match, the page table entry for the desired page is in the TLB. The base address from this entry is used to compute the physical address. If there is no match, the 386 reads the page table entry from memory and puts it in the TLB. If the P bit in the page table entry is a 1, indicating that the page is present in physical memory, then the physical address will be computed and the desired word in the page accessed. If the P bit in the page table entry is a 0, indicating that the page is not present in physical memory, the processor will generate a page fault exception (type 14). After the page fault exception handler swaps the page into physical memory, the paging unit will compute and output the physical address for the desired word.

When the 386 paging mode is enabled, the U/S and R/W bits in the page directory entries and the page table entries can be used in place of or in addition to the segmentation protection mechanisms. The U/S bit in a directory or page table entry is used to specify one of two privilege levels, user or supervisor. A 0 in the U/S bit specifies the user privilege level, which corresponds to segment privilege level 3, the lowest level. A 1 in the U/S bit specifies supervisor privilege level, which corresponds to segment privilege levels 0, 1, and 2.

The R/W bit in a page directory or page table entry can be used to establish read-write access rights for pages or page tables. Figure 15-27*b* shows the access rights produced by various combinations of U/S and R/W. Note that for these bits 11 represents the most privilege and 00 the least privilege. If the access rights specified in a page directory entry are different from the access rights specified in a page table entry, the least privileged of these determines the access rights.

SUMMARY OF MEMORY MODELS

The memory in a 386 or 486 system can be set up using the segments-only model, the segmented-paged model, the simple flat model, or the paged flat model.

We thoroughly described the segments-only model in a previous section. This is the only protected-mode memory model available on an 80286. Versions 1.1 and 1.2 of Microsoft's OS/2 protected-mode operating system were designed to run on an 80286 system, so they use this model.

As we explained before, 386 segments can be too large to be conveniently swapped in and out of memory, so the 386 allows a paging mechanism to be switched in after the segmentation unit. The paging unit divides segments into 4-Kbyte pages for swapping in and out of physical memory. This segmented-paged model allows a programmer to think in terms of logical segments and the virtual memory hardware to think in terms of easily moved pages. However, one problem with this combined approach is that the amount of time required to manage all the descriptor tables, segments, page tables, and pages in a complex system becomes too large. A second problem is that developing the software to manage all this is a complex task. Also, the amount of memory used by all the tables can become excessively large. For these and other reasons, Microsoft's OS/2 for the 386, Novell's Netware 386, and many other programs for 386 and 486 systems use the flat memory model, which effectively removes segmentation.

The 386 does not have a way to turn off segmentation, but you can effectively eliminate segmentation by initializing all the segment registers with the same base address and initializing the segment limits for 4 Gbytes. Each segment then corresponds to the 4-Gbyte physical address space of the 386. The 32-bit offset or effective address part of each memory address is large enough to access any location in this 4-Gbyte space. The different parts of programs are simply located at different offsets in the address space.

This memory mode is referred to as the *simple flat system model* and is useful for dedicated control applications that need the fastest possible task switching and don't need all the segment-based protection features. The SDK-386 board we discussed earlier uses the simple flat model, and as we show later, this makes program development for it quite easy. Also, the flat system model makes it easy to transport software written for nonsegmented devices such as those in the Motorola 68000 family devices to a 386.

Paged flat model systems enable the 386 paging mechanism to provide virtual memory-management and protection features. The present (P) bit in a page directory entry indicates whether the requested page table is present in memory and the P bit in the page table entry indicates whether the requested page is present in memory. The accessed (A) bit in a page table entry indicates whether the page has been accessed. The operating system can periodically check and reset this bit to determine how often the page is being used. If the page has not been used lately, it can be replaced when the operating system needs space for a new page. The dirty (D) bit in a page table entry will be set if data has been written to the page. In this case the operating system must write the modified page out to disk before swapping a new page into its space. As we discussed earlier, the user/supervisor (U/S) bit in the page directory

entries and the page table entries provide two privilege levels. The read/write (R/W) bit in a page table entry allows a page to be marked as read only or read/write. The I/O permission bit map which we mentioned earlier can provide protection for I/O ports.

The point of all this is that the paged flat memory model provides fast virtual memory capability and a degree of protection adequate for most applications.

386 Virtual 8086-Mode Operation

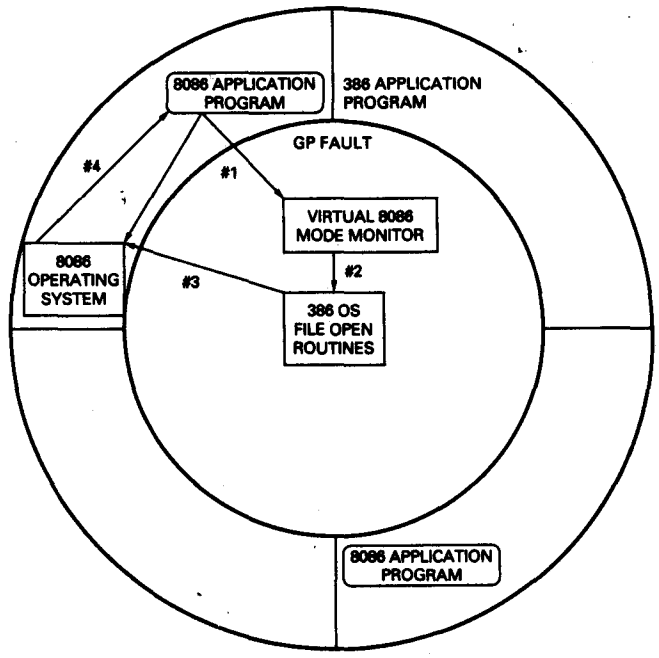
As we pointed out in an earlier discussion, it is difficult to switch a 286 processor back and forth between real and protected mode. This limitation makes a 286 hard to use for a multitasking system, which must run a mixture of tasks that use segment-offset addressing and protected-mode tasks that use descriptors. The 386 virtual 8086 mode solves this problem. A 386 operating in protected mode can easily switch to virtual 8086 mode to execute a time slice of an 8086-type program and then easily switch back to protected mode to execute a time slice of a protected-mode task. This means that some users in a multiuser system can be running programs under protected mode UNIX V and other users can be running real-mode DOS programs.

When a 386 operating in protected mode does a task switch, it examines the VM bit in the EFLAGS register. If this bit is set, the 386 will enter virtual 8086 mode to execute the new task. If the VM bit is not set, the 386 will execute the new task as a normal protected mode task.

In virtual 8086 mode the 386 computes physical addresses using the segment-offset mechanism used by an 8086. Therefore, the address range of a virtual 8086 mode task is 1 Mbyte. For a single virtual 8086 task this address range is in the lowest 1 Mbyte in the processor address space. If a system needs to run several different 8086 type tasks, then the 386 is operated in paging mode so that each 8086 task can be given a different page table and a different set of pages in physical memory. A side benefit of using the paging mode is that the U/S and R/W bits in the page directory entries and the page table entries provide protection that is normally not available in real mode.

In order to run virtual 8086 mode tasks, the operating system must have a section of privilege level 0 code called a *virtual machine monitor*. The main purpose of this monitor is to intercept interrupts, exceptions, and INT n instructions which occur during the execution of the 8086 task. Figure 15-28 shows how this works for an INT n instruction.

As you well know from previous chapters, most 8086 system programs use INT n software interrupts to access BIOS and DOS I/O procedures. In virtual 8086 mode the INT n instruction can be executed only at privilege level 0, the highest privilege level. Since an 8086 task always operates at level 3, the lowest privilege level, the 386 will generate an exception whenever the 8086 program executes an INT n instruction. The handler for this exception is in the virtual machine monitor, so the monitor effectively takes over execution at this point.



8086 APPLICATION MAKES "OPEN FILE CALL" → CAUSES GENERAL PROTECTION FAULT (ARROW #1)
VIRTUAL 8086 MONITOR INTERCEPTS CALL. CALLS 386 OS (ARROW #2)
386 OS OPENS FILE RETURNS CONTROL TO 8086 OS (ARROW #3)
8086 OS RETURNS CONTROL TO APPLICATION. (ARROW #4)
TRANSPARENT TO APPLICATION

FIGURE 15-28 Operation of virtual machine monitor when 8086 virtual mode application program makes DOS call to open a file. (*Courtesy Intel Corporation*)

During the task switch to the monitor, the state of the 8086 task is saved in its TSS. Also the VM bit in the EFLAGS register is reset, so the monitor can operate in normal protected mode.

If the call was to a function such as the DOS "open file" command, the monitor will call the equivalent procedure in the 386 protected-mode operating system to open the file. This mechanism maintains all the protection built into the main 386 operating system. When the file has been opened, execution is returned to the DOS operating system. The IRET instruction used to return to the virtual 8086 DOS program restores the 8086 task state. As part of this, the VM bit in the EFLAGS register is restored to a 1 so that the 8086 program restarts in the virtual 8086 mode. For other DOS function calls which do not involve I/O, the monitor may return execution to DOS to perform the function. After the function is completed, DOS returns execution to the 8086 program.

In virtual 8086 mode interrupts are also intercepted by the monitor. In most cases the monitor will transfer execution to the 386 protected mode operating system to service the interrupt. To service interrupts the 386 operating system uses the interrupt descriptor table and gate scheme we described previously, so protection is maintained. If protection is not an issue, the monitor may return execution to DOS or to the 8086 program to service the interrupt.

When a clock tick interrupt occurs to signal the end of a time slice, execution will switch from the 8086 task to the monitor task. The monitor task through an IDT gate will switch to the 386 operating system scheduler. The scheduler will then switch to the next user task. The VM bit in the EFLAGS register image of the TSS for the new task will determine whether the task is executed in virtual 8086 mode or 386 protected mode. The point here is that the 386 provides a relatively simple mechanism to alternate between 8086-type programs and 386 protected-mode programs.

Now, before we dig into 386 instruction set enhancements and programming, let's summarize what we have found out about the 386 so far.

Summary of 386 Hardware and Operating Modes

The 386 is a 32-bit processor which is upward compatible from the 8086, 80186, and 80286. In real address mode the 386 functions as a fast 8086 and uses the segment-offset address mechanism to address 1 Mbyte of memory.

In its protected mode a 386 can address 4 Gbytes of physical memory and 64 Terabytes of virtual memory. Each protected-mode address consists of a 16-bit selector and a 32-bit offset or effective address. The 32-bit offset component means that segments can be as large as 4 Gbytes. An optional paging mechanism allows segments to be broken into 4 Kbytes pages for faster swapping in and out of memory. The 386 uses the 16-bit selector to access the descriptor for the segment in the global descriptor table or in a local descriptor table. The segment base address from the descriptor is added to the 32-bit offset to produce the linear address. In segments-only mode, the linear address is the physical address. If paging is enabled, the paging unit uses the linear address, a page directory, and a page table to produce the physical address.

The 386 contains several mechanisms to protect OS code from user tasks and user tasks from each other. One of these mechanisms is privilege levels. The operating system code is given a privilege level of 0, the highest privilege level, and user code is given a lower privilege level. Any direct attempt by a program to access a code or a data segment with a higher privilege level will generate an exception. Programs can, however, access procedures at a higher privilege level through an indirect method called a gate. The gate allows a second check on the privilege level of the access and makes sure the access is to the correct location in the procedure. A second protection mechanism is bounds checking. Any attempt to access a location outside the limit specified for a segment in its descriptor will generate an exception.

For a 386 operating in protected mode, interrupts are vectored through gates in the interrupt descriptor table. This indirect approach allows interrupt procedures to be protected.

In a 386 system using the flat memory model, the entire physical memory is treated as a single large segment. All segments are given the same base address and limit, so they share this segment. The 32-bit offset contained in every memory address is large enough to access any location in the 4-Gbyte physical address space of the 386. In a larger system using the flat memory model, paging is enabled so that virtual memory and protection can be implemented.

When the 386 does a protected-mode task switch, it automatically copies the state of the current task to a task state segment created for that task and loads the state of the new task from its TSS. If the 386 finds the VM bit of the EFLAGS register set when it does a task switch, the 386 goes to virtual 8086 mode. In this mode the 386 can directly execute 8086 type programs which use segment-offset addressing. The interrupt at the end of a time slice will cause the 386 to switch back to full protected mode so the operating system can switch to the next task using protected-mode features.

386 Instruction Set Additions and Enhancements

A SECOND LOOK AT THE 386 REGISTER SET

In Figure 15-20 we showed you that the 386 register set is a superset of the 8086 and 80286 register sets. The 386 register-type instructions allow you to specify 8-bit registers and 16-bit registers as you do in 8086 instructions or to specify 32-bit registers. In 386 instructions you can specify, for example, AH, AL, AX or EAX as an operand. The instruction MOV EAX,EBX, for example, will copy the 32-bit number in the extended BX register to the extended AX register. You cannot copy an 8-bit part of a register to a 32-bit register with an instruction such as MOV EBX,AL. Also, you cannot directly access just the upper 16 bits of a 32-bit register. If you need to copy just the upper 16 bits of, for example, the EAX register into the BX register, you can first rotate the upper 16 bits of EAX into the lower 16 bits with the ROR EAX,16 instruction and then use MOV BX,AX. If you need to put EAX back in its initial condition, you just do another ROR EAX,16 instruction. The 386 contains a "barrelshifter," which can shift an operand any number of bits in one clock pulse, so these rotates do not appreciably slow the overall operation. Incidentally, even a 386 real-mode program which uses 16-bit segments can use the 32-bit extended registers for data operations.

For real-mode programs only the lower 16 bits of the extended instruction pointer (EIP) are used, because only 16 bits are needed to access any location in a 64-Kbyte real-mode segment. In a protected-mode program a code segment can be specified as 16-bit or 32-bit. To specify a code segment as 16-bit, you simply write the term USE16 after SEGMENT in the segment declaration line. The line CODE SEGMENT USE16, for example, declares a 16-bit segment named CODE. A segment is specified as 32-bit by putting the term USE32 after SEGMENT in the segment declaration. If the segment is specified as a USE16 segment, then the maximum segment limit is 64 Kbytes, and only the lower 16 bits of EIP will be used to access instruction bytes. If the segment is specified as USE32, then the maximum segment limit is 4 Gbytes, and all 32 bits of EIP are used to hold the offset of an instruction byte.

The 386 contains two new segment registers, FS and GS, which can be used as additional data segments. None of the 386 instructions use these segments as their default segment, so you usually have to use a segment override prefix on an instruction which accesses a data item in one of these segments. We will show you how to do this in a later program example.

NEW ADDRESSING MODES AND SCALING

When an 8086 executes the instruction MOV AX,PATIENT_RECORD [BX] [DI], it computes the effective address of the memory operand by adding a displacement represented by the name PATIENT_RECORD, an offset contained in the BX register, and an index value contained in the DI register. For an 8086 only BX and BP can be used as base registers in this way, and only SI and DI can be used as index registers. A 386 can use any of the eight 32-bit, general-purpose registers as a base register, and it can use any of the 32-bit, general-purpose registers except ESP as an index register. When a 386 executes the instruction MOV BX, [EAX+EDX], for example, it will compute the effective address of the memory operand by adding the 32-bit number in EAX to the 32-bit number in EDX. Note that these new addressing modes work only with the 32-bit extended registers. You can't, for example, use just AX as a base pointer.

The 386 also has another powerful addressing feature called *index scaling*, which is useful for accessing successive elements in an array of words, double words, or quad words. Index scaling allows the value contained in an index register to be automatically multiplied by a specified scale factor of 2, 4, or 8 when an instruction executes. If a 386 executes the instruction MOV EAX, [EBX+EDI*4], for example, it will multiply the index value in EDI by a scale factor of 4 and add the result to the value from the EBX register to produce the effective address. The double word pointed to in DS by this effective address will be copied into the EAX register. If this instruction is part of a loop which processes an array of double words, then all that you have to do to get ready for the next trip around the loop is to increment the index value in EDI. When the MOV instruction is executed again, the new index value will automatically be multiplied by the scale factor in computing the effective address.

Even though real-mode data segments can be only 64 Kbytes long, you can still use index scaling and these new 32-bit addressing as long as the effective address produced does not exceed 16 bits. In a later section we show you an example program which demonstrates how to do this.

NEW INSTRUCTIONS

The 386 instruction set includes all the 8086/80186/80286 instructions and extends these instructions to work with 32-bit data words and 32-bit offsets. The 386 also includes several new instructions. In this section we briefly explain the functions of these new instructions and give you an example of each. Then we make a few comments about 8086 instructions that have been enhanced in the 386.

Bit Scan and Test Instructions

BSF—Bit scan to the left until nonzero bit is found.

EXAMPLE:

```
BSF CX, DX    ; Scan DX to left until nonzero bit,
              ; leave bit number in CX
              ; Zero flag set if all DX=0
```

BSR—Bit scan to the right until nonzero bit is found.

EXAMPLE:

```
BSR CX, DX    ; Scan DX to right until nonzero bit,
              ; leave bit number in CX
              ; Zero flag set if all DX=0
```

BT—Bit test and put specified bit in carry flag.

EXAMPLE:

```
BT EBX, 4     ; Copy bit 4 of EBX to carry flag
```

BTC—Bit test and complement.

EXAMPLE:

```
BT EBX, 7     ; Copy complement of bit 7 to CF
```

BTR—Bit test and reset.

EXAMPLE:

```
BTR WORD PTR [BX], 3 ; Bit 3 of [BX] to CF
                     ; Reset bit 3 of [BX]
```

BTS—Bit test and set.

EXAMPLE:

```
MOV CL, 4
BTS EAX, CL   ; Copy bit 4 of EAX to CF
              ; Set bit 4 of EAX
```

Data-type conversions

CDQ—Convert signed double word in EAX to quadword in EDX:EAX.

CWDE—Converts signed word in AX to double word in extended EAX.

Segment load instructions

These instructions are similar to LDS and LES instructions described in Chapter 6.

LFS—Load FS segment register and specified base register with values from specified memory locations.

EXAMPLE:

```
LFS BX, DWORD PTR [DI]
                ; Load FS and BX with
                ; DWORD from memory at [BX]
```

EXAMPLE:

```
LFS EBX, FWORD PTR [DI]
                ; Load FS with 16 bit
                ; selector and EBX with 32-bit
                ; offset for memory at [DI]
```

LGS—Load GS segment register and specified register from specified memory locations.

LSS—Load SS segment register and specified register from specified memory locations.

Move and expand instructions

MOVSX—Move and sign extend to fill destination register.

EXAMPLE:

```
MOVSX CX, BL    ; Copy BL to CL, extend sign bit of
                ; BL through all of CH
```

MOVZX—Move and zero extend to fill destination register.

EXAMPLE:

```
MOVZX CX, BL    ; Copy BL to CL, fill CH with zeros
```

Set memory flag word instruction

SETxx—Set all bits in specified byte if condition xx is met. xx here can be any condition from conditional jump mnemonics.

EXAMPLE:

```
SETC TooBig     ; Set all bits in flag TooBig if
                ; Carry flag set
```

Shifts between words

SHLD—Shift specified number of bits left from one operand into another.

EXAMPLE:

```
SHLD EAX, EBX, 8  ; Shift upper 8 bits from EBX
                  ; into lower 8 bits of EAX
                  ; EBX unchanged
```

SHRD—Shift specified number of bits right from one operand into another.

EXAMPLE:

```
SHRD EAX, EBX, 8  ; Shift lower 8 bits from EBX
                  ; into upper 8 bits of EAX.
                  ; EBX unchanged
```

INSTRUCTION ENHANCEMENTS

Several of the 386 instructions have significant improvements over the 8086/80186/80286 versions. Here are a few notes about these improvements.

1. The 386 string instructions work with double-word operands as well as with word and byte operands. A "B" at the end of an instruction mnemonic specifies byte operands, a W specifies word operands, and a D specifies double-word operands. Examples are CMPSB, CMPSW, and CMPSD.
2. The destination for a 386 conditional jump can be anywhere in the segment containing the jump instruction. Conditional far jumps must still be done by changing the jump condition and using an unconditional far jump as we showed you for the 8086 in Chapter 4.
3. The LOOP instructions can use the CX register or the ECX register as a counter. If you want CX to be used, write the instructions as LOOPW, LOOPWE, and LOOPWNE. If you want the ECX register to be used as the counter, write the instructions as LOOPD, LOOPDE, and LOOPDNE.
4. PUSHFD pushes the 32-bit EFLAGS register and POPFD restores it.
5. PUSHAD pushes the 8 general-purpose 32-bit registers on the stack, and POPAD restores these registers except for the value of ESP which is ignored.
6. IRETD pops the double word EIP, a double word for CS, and the EFLAGS register off the stack. The high word of the value popped for CS is discarded.
7. The IMUL instruction can now perform signed multiplication on any general-purpose register and a memory location or another general-purpose register.
8. In addition to the protected-mode instructions inherited from the 80286, the 386 instructions used to move data to/from the control registers (CR0–CR3), the debug registers (DR0–DR7), and the test registers (TR0–TR7) can be executed only in protected mode at privilege level 0. These instructions are simple MOV register, register instructions.

386 Programming

INTRODUCTION

The tools and techniques used to write a program for a 386 or a 486 depend very much on whether it is a system program or an application program and whether it utilizes protected mode or not. The tools and techniques are also determined by whether the program is going to

execute in a graphical user-interface environment such as Microsoft's Windows 3.0 or OS/2. In this section we give you an introduction to writing 386 programs for five different programming environments.

386 PROGRAMS FOR MS DOS–BASED SYSTEMS

Current versions of DOS are designed to run on 8086/8088, real-mode 80286, or real-mode 386/486 systems. If you want a program to be able to run under DOS on any one of these systems, then you have to write it for the "weakest link" in the group, the 8086. The 8086 and C programming examples throughout this book were in fact compiled and run on a 386-based system. If you are writing a program that you are sure will only be run under DOS on a 386- or 486-based machine, you can write the program to take advantage of the 32-bit processing capability, addressing modes, and enhanced instructions of the 386. Assuming that you are using the MASM or TASM assembler, you tell the assembler to accept 386 instructions by putting the .386 directive at the top of your source program, as shown in Figure 15-29*a*.

If you are using the simplified segment directives, make sure to put the .386 directive after the .MODEL directive, as shown in Figure 15-29*b*. This order tells the assembler to create 16-bit segments which are compatible with real-mode operation. If you put the .386 directive before the .MODEL directive, the assembler will create 32-bit code and data segments which can be used only in protected mode.

Even though 386 real-mode programs are limited to 64-Kbyte segments and unless bank switching is implemented, to a 640-Kbyte address space, you can still use the 32-bit extended registers, addressing modes, and new instructions of the 386. The simple example program in Figure 15-29*a* shows some of the possibilities. The main points we included in this example are: how to declare segments; how to access data in the new segments, FS and GS; and how to use the 32-bit addressing modes and scaling.

First note that the code and data segments are specified as 16-bit with USE16 directives. The USE16 on the data segment directive specifies a maximum segment length of 64 Kbytes as required for real-mode operation. The USE16 on the CODE SEGMENT line tells the assembler to compute all memory addresses using the segment base and 16-bit offset method.

Next in the example note that we assume and initialize the FS register, just as we did the DS, ES, and SS registers in earlier program examples. The first action of the program then shows you how to read a double word from this segment to the 32-bit EAX register. A segment override prefix must always be used for references to the FS or GS segment, because there is no default as there is with DS. The ROR EAX,16 instruction rotates the 32-bit EAX register around 16 bits to the right. As we said earlier, if execution is limited to a 386 system, the 32-bit registers can be used for data operations, even in real mode.

The next section of the example in Figure 15-29*a* shows how the 32-bit addressing modes can be used to help process an array of words. The instruction MOV

```
  Some 386 addressing modes and instructions

 .386
DATA   SEGMENT USE16
       BIGVAL  DD 12345678H
       TABLE   DW 4235H, 7590H, 4968H, 3817H
DATA   ENDS

CODE   SEGMENT USE16
       ASSUME CS:CODE; FS:DATA

START: MOV  AX, DATA            ; Initialize FS
       MOV  FS, AX              ;  register
;Read 32-bit operand from memory
       MOV  EAX,FS:BIGVAL       ; Get double word
       ROR  EAX, 16             ; Swap word order
       MOV  FS: BIGVAL, EAX     ; Put back result
;Process table
       MOV  CX, 04
       MOV  EDI, 0
NEXT:  MOV  DX,FS:[TABLE+EDI*2] ; Word from array to DX
                                ; Scan from left for
       BSF  AX, DX              ; first zero bit
       JNZ  MORE                ; Skip if all zeros
       MOV  FS:[TABLE+EDI*2],AX ; Store number of
                                ;  first zero bit
MORE:  INC  EDI                 ; Increment index
       LOOP NEXT
       MOV  AX, 4C00H           ; Return to DOS
       INT  21H
CODE   ENDS
       END START
```

(a)

```
;Simplified segment directives example
; for 16-bit segments

DOSSEG
 .MODEL large
 .386
 .DATA

 TABLE DW 4235H, 7590H, 4968H, 3817H

 .CODE
 START:  MOV AX, DATA        ; Initialize FS register
         MOV FS, AX
```

(b)

FIGURE 15-29 (*a*) 386 real-mode program using traditional segment directives. (*b*) Simplified segment directives for generating 16-bit 386 segments.

DX,FS:[TABLE + EDI*2] copies a word from the array to DX. The effective address for this instruction is computed by multiplying the contents of EDI by 2 and adding the result to the displacement represented by the name TABLE. The first time through the loop EDI contains 0, so the effective address is just TABLE, the offset of the first word in the array. Before the loop executes again EDI is incremented to 1. During the

next execution of the MOV DX,FS: [TABLE+EDI*2] instruction the effective address will be TABLE+2, the offset of the second word in the array. An important point here is that in real-mode operation an exception will be generated if the effective address produced by an instruction is greater than 64 Kbytes.

Within the loop we use one of the new 386-bit instructions to process the word read in from memory. The BSF AX, DX instruction will scan the DX register starting from the left until it finds a nonzero bit. The number of the first nonzero bit will be loaded into AX. If all bits in DX are zeros, the zero flag will be cleared. In this case we just leave the word of all zeros in the array and process the next word. If the word is not all zeros, we write the number of the first nonzero bit in the memory word and then process the next word.

Finally, in the example program we use a familiar DOS function call to return execution to DOS. This is not a particularly significant program, but it does show you a little of what you can do with the added features of the 386 if you are willing to sacrifice 8086 downward capability.

386 PROGRAMS FOR THE SDK-386

As we told you earlier, you can download the binary programs from a PC- or PS/2-type computer to an URDA SDK-386 board through an RS-232C link. The SDK-386 board operates in protected mode using the simple flat memory model, so you can use it to experiment with a simple, dedicated 386 system such as those in Chapter 10.

During initialization the monitor program sets up a global descriptor table, a local descriptor table, and an interrupt descriptor table. The GDT, LDT, and IDT registers in the 386 are loaded with the base addresses and limits for these tables. The monitor also sets up a system task state segment and a user task state segment. As part of the initialization the selector for the user task state segment is loaded into the backlink field of the system TSS. When a user presses the RUN or the STEP key, an IRET instruction causes a task switch to the user task. This executes the user program. If the user presses the BREAK key, an NMI interrupt will be generated. This causes a task switch to the system task so that registers, memory locations, etc. can be examined. The BRPT key can be used to load up to four breakpoint addresses in the 386 debug registers. As we explained earlier, the 386 checks each memory address and will break if it finds any of the specified breakpoint addresses.

You have considerable flexibility in how you write a program for the SDK-386 board. The simplest approach is to use a format slightly modified from that in Figure 15-29*a*. The flat model memory mode used by this board means that all segments start from absolute address 00000000H, and all the segment registers contain the same base address. System and user programs use 32-bit offsets to access code and data words in this shared segment. The contents of the segment registers then do not have to be changed during a switch from the system task to the user task. This has important implications for how you write a program to run on the board.

The D bit in the code segment descriptor determines whether the 386 uses 16-bit effective addresses or 32-bit effective addresses. If the D bit is a 0, then 16-bit effective addresses are produced and if D = 1, 32-bit addresses are produced. The monitor program in the board sets the D bit of the code segment descriptor to a 1, so this means that 32-bit addressing is assumed. To make your program compatible, you make the code segment a USE32 type so that the assembler will produce 32-bit offsets. Your data segments should also be made USE32 type to be compatible with the segments set up by the monitor. Incidentally, you don't have to initialize data segments as part of your program, because the monitor loads the selectors and descriptors for these, and they are not changed when execution switches to the user task.

The user area of RAM on the board begins at 300H, so you should include an ORG 300H directive before the code segment in your program. The program will then be assembled to run in this address space in RAM. After the program is assembled and linked, it can be downloaded to the board and run.

For a more complex program the board allows you to use the segment-based protection features of the 386. The global descriptor table contains four user-definable descriptors and the local descriptor table contains six user-definable descriptors. The interrupt descriptor table also contains a user-definable descriptor for the USER INTERRUPT key on the board and another user-definable descriptor. We don't have a space here to show you how, but you can use these descriptors to define custom segments for your programs.

WRITING A 386 PROTECTED-MODE OPERATING SYSTEM

In the unlikely case that you should have to write a 386 protected-mode operating system or monitor program, you should be aware of Intel's 386 Relocation, Linkage, and Library (RLL) tools which run on IBM PC/AT or newer microcomputers. This tool set contains a *binder*, which is a high-powered linker that can combine object modules compiled from different languages into tasks, combine segments, resolve PUBLIC/EXTERNAL references, assign virtual addresses, and generate a file which can be loaded into RAM for debugging. The tool set also contains utilities for working with library functions. Another important part of the tool set is the *builder*, which allows a programmer to assign physical addresses to segments; set segment access rights and limits; create gates; create global, local, and interrupt descriptor tables; create task state segments; and set up the boot process.

Incidentally, the Intel 80386 System Software Writer's Guide shows a simple example of a flat system and a simple example of a segmented system.

MICROSOFT'S OS/2 2.0 OPERATING SYSTEM

As we told you earlier, MS DOS is for the most part a single-user, single-task operating system and does not take advantage of the virtual memory and multitasking capabilities of the 286/386/486 processors. The probable successor to DOS is Microsoft's OS/2, which is a single-user, multitasking operating system. Microsoft's OS/2

version 1.0 was an early attempt at a multitasking operating system for the 80286 processor. OS/2 1.0 and the later versions of OS/2 for the 80286 can multitask several protected mode tasks and one real mode task. The real-mode task is run in a "DOS compatibility box." The reason that only one real mode task can run is the difficulty in switching an 80286 from protected mode to real mode and back. The user interface for OS/2 1.0 was a typed command line similar to DOS.

The next version of OS/2, OS/2 1.1 introduced a new graphical user interface (GUI) called the *Presentation Manager* or PM. PM is similar to the screen-based interface you may have seen on Apple Macintosh computers. In PM you execute commands by using a mouse to move a cursor to a desired command in a menu of commands or to an *icon* which represents the command. You then execute the command by clicking a key on the mouse. PM also allows you to have multiple "windows" open on the screen. You can "cut" something from one window and "paste" it into another window. The file manager in PM allows you to display a directory tree on the screen, select a file from the tree, and perform some action on the file by just moving cursor around on the screen and clicking the mouse key at the appropriate points.

OS/2 1.2 kept Presentation Manager and added the High Performance File System (HPFS). Instead of the FAT used by the DOS file system, the HPFS uses a different system which allows much faster file access. Another obvious improvement in HPFS is that filenames can be longer than 8 characters. HPFS also sets up an "extended attribute" block for each file. The operating system or an application program can use this block to describe and control use of the file.

OS/2 version 2.0, designed to run only on 386 and 486 systems, uses the virtual 8086 mode of these processors to implement Multiple Virtual DOS Machines (MVDM) capability. In addition to the features of earlier versions, OS/2 2.0 can multitask any mixture of DOS programs, applications written for earlier versions of OS/2, and applications written specifically for version 2.0. OS/2 2.0 uses the flat paged memory model that we described earlier for the 386.

In the preceding chapters we showed you how to use DOS function calls to open files, read files, etc. in your programs. In OS/2 2.0 there are three different *application program interfaces* (APIs) or sets of functions which can be called to perform these functions. One is the real-mode DOS compatible API which is used by the programs operating in a DOS compatibility box. The functions in this API are called with software interrupts such as INT 21H. The second API contains the 16-bit functions that are compatible with OS/2 1.x application programs. To use one of these functions the required parameters are first pushed on the stack, and then the procedure is called by name. The third API contains the 32-bit functions used for OS/2 2.0 applications. The functions in this API are also called by name after pushing the required parameters on the stack. Unlike the 16-bit API calls, the parameters for these calls are pushed on the stack in the same order as parameters for C function calls are pushed. In fact, the 32-bit API is in some ways like an extension of the C Run Time Library you met in Chapter 12. One major difference between the 32-bit API and the C RTL is in the way the functions are connected to the .exe program.

When a C program is linked, the object code files for library functions are linked with the object code for the compiled C program modules to make the .exe program. The OS/2 APIs are dynamic link libraries (DLLs). When a program containing an API call is linked, a reference to the function is put in the .exe file instead of the object code for the called function. When the program is loaded into memory to be run, the library containing the function is loaded into memory where the program can access it. This approach may seem strange at first, but it has several advantages. First, the .exe programs are much smaller, because they do not contain the large library functions. This means that the .exe files take less space on a hard disk. Also, the API functions are reentrant, so a library can be shared by multiple tasks on a multitasking system. This saves memory, because each task does not have to have a copy of the library in memory. Still another advantage of the DLL approach is that by updating the library you can update all programs that use the library, without having to relink each program.

You can write DOS type programs for an OS/2-based system using the programming tools we described in earlier chapters. For simple protected-mode programs you can use Microsoft's C 5.2 32-bit C compiler and the latest version of Microsoft's Macroassembler (MASM). To develop a 32-bit OS/2 application which utilizes PM, you use Microsoft's OS/2 2.0 Software Developer's Kit which contains all the needed tools.

MICROSOFT WINDOWS 3.0

Microsoft's Windows 3.0 is a relatively inexpensive bridge between the DOS world and the high-powered OS/2 2.0 operating system we discussed briefly in the preceding section. As the name implies, this program uses a graphical user interface (GUI) very similar to Presentation Manager. Windows 3.0 is essentially a very flexible DOS extender which can take advantage of the protected-mode features of the 80286, 386, and 486 processors. It can be operated in any one of three different modes, depending on the processor and memory available in the system. On an 8086/8088-based system, Windows 3.0 must be operated in its real mode. On an 80286-based system with at least 1 Mbyte of extended memory, Windows 3.0 can be run in standard mode, which takes advantage of the protected mode features of the 80286. In real mode and standard mode, only one DOS type task can be run at a time.

On a 386- or 486-based system with at least 2 Mbytes of extended memory, Windows 3.0 can be run in its 386 enhanced mode. In this mode, which is the one we are interested in here, it uses the 386's virtual 8086 mode to run multiple 8086 tasks, and it implements paged virtual memory so it can run programs that require more memory than is physically present in the system.

When you run Windows 3.0 the program manager window shown in Figure 15-30, page 568, appears on the screen. The icons along the bottom of this window

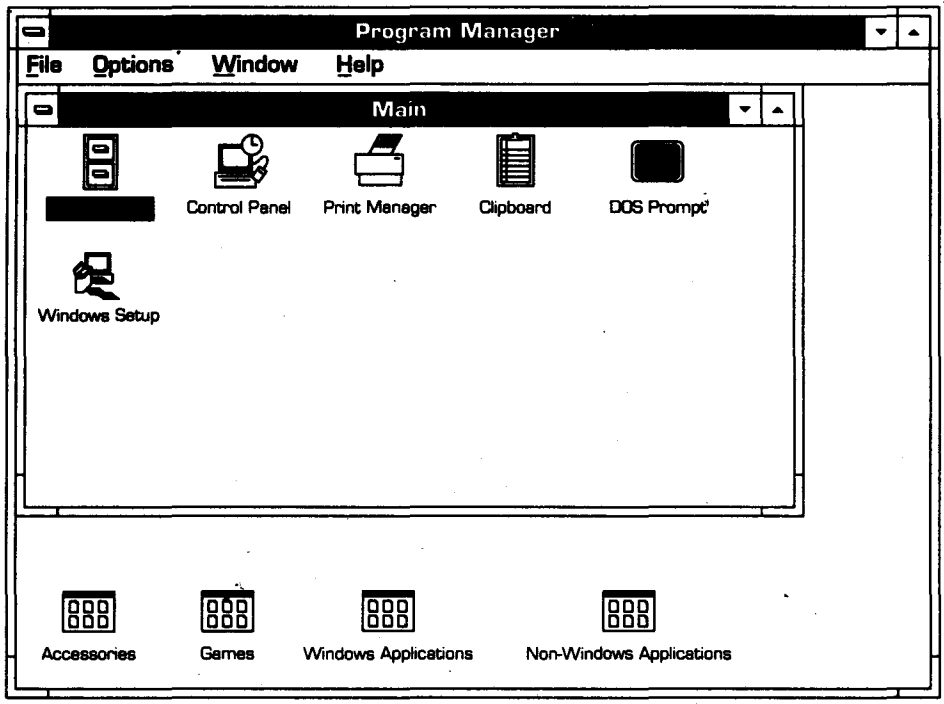


FIGURE 15-30 Microsoft Windows 3.0 Program Manager window display.

represent groups of programs you can execute. If you move the cursor to one of these icons and double-click the left mouse key, another window containing a menu of the programs in that group will appear. You can execute a program in the group by just double-clicking on the name of the program in the menu. If you want to start another program running at the same time, you can "minimize" the window for the running program to put that program in the background and then start another program running. The background program continues running after you start the new program in the foreground. Windows even allows you to specify the percentages of time you want the processor to spend on the background task and on the foreground task. Windows 3.0 keeps a task list of the currently running tasks. You can switch a task from background to foreground by bringing up the task list and clicking on the desired task.

One of the program icons in the program manager window is the file manager. When you double-click on this icon, it opens a file window and shows you a tree of the files and subdirectories in your current directory. In this window you can use the mouse to perform the usual file operations such as copy, delete, rename, etc. The point of all this is that instead of typing in commands, you can perform almost any operation by just moving the cursor to the appropriate location on the screen and clicking the mouse key. Windows 3.0 also has a very versatile on-screen help system which you can pop up as needed.

Windows 3.0 separates programs into two categories, windows applications and nonwindows applications. During setup Windows 3.0 scans your disk drive(s) and puts the programs it finds into the correct category. Programs not specifically written for 3.0 will be classified as nonwindows applications. In the real and standard mode, nonwindows applications are run in full-screen mode similar to the way they would run in a pure DOS environment. Windows applications take advantage of the GUI. Among the windows application-type programs that come with Windows 3.0 are a word processor, notepad, paintbrush, calendar, clock, print spooler, and a card file.

To write a simple Windows 3.0 application program you can use the Asymetrix Corp *Toolbook* which comes with Windows 3.0. Toolbook includes some impressive demonstrations. To develop more complex Windows 3.0 applications, you need tools such as version 2.0 of Borland's C++ or the Microsoft Windows 3.0 Software Development Kit. The programming guide that comes with this tool set contains a sequence of templates that you can use to develop a custom application. We had hoped to rewrite the SDKCOM1 program from Chapter 14 as an example windows application program, but we ran out of space and time. Maybe we can include this in the next book.

THE INTEL 80486 MICROPROCESSOR

The 32-bit 486 is the next evolutionary step up from the 386. The basic processor unit used in the 486 is the same as that used in the 386, so all of the preceding discussion of the 386 applies to the 486. All we have to discuss here are the additions and enhancements that designers were able to add by increasing the number of transistors on the die from 300,000 to about 1,200,000.

As you can see in Figure 15-31, one of the most obvious features included in a 486 is a built-in math

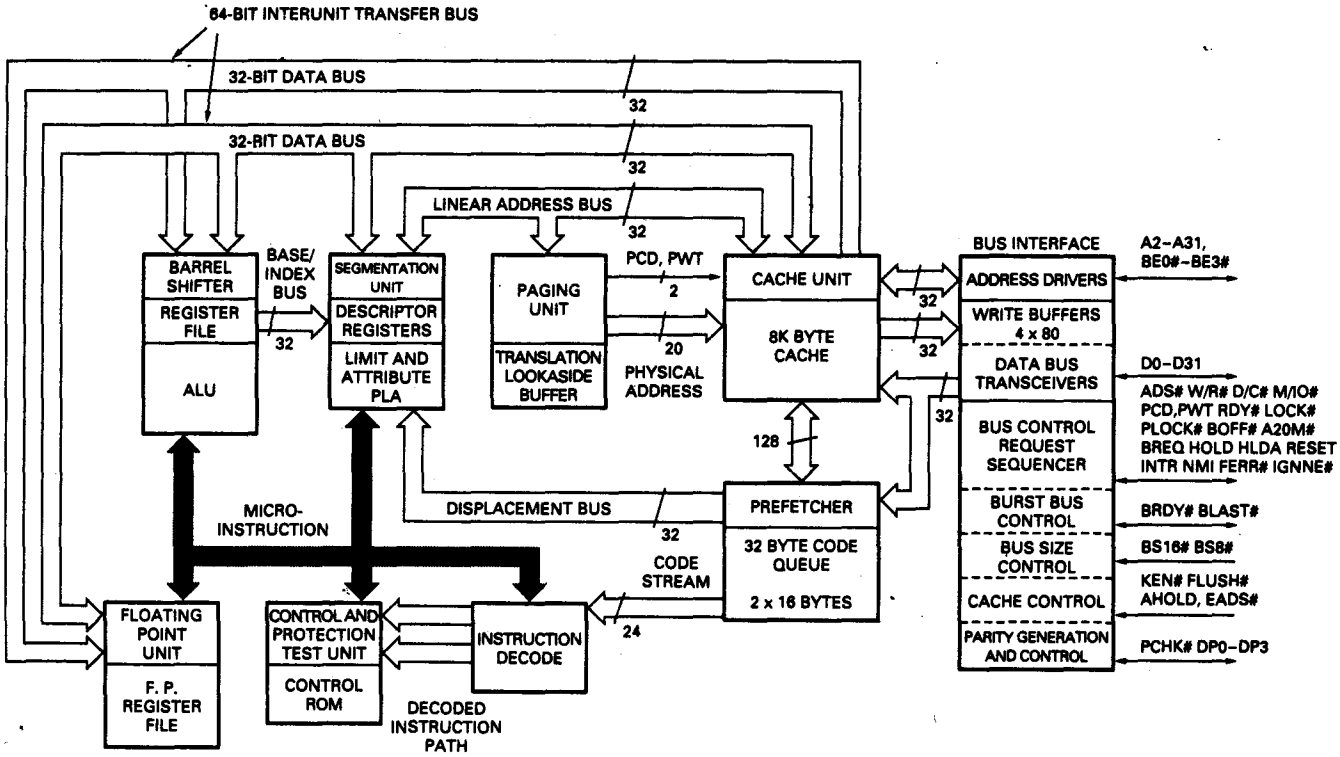


FIGURE 15-31 Intel 486 internal block diagram. (*Courtesy Intel Corporation*)

coprocessor. This coprocessor is essentially the same as the 387 processor used with a 386, but being integrated on the chip allows it to execute math instructions about three times as fast as a 386/387 combination.

Another fairly obvious feature included in a 486 is an 8-Kbyte code and data cache. This four-way set-associative cache works in basically the same manner as the external caches we described in Chapter 11. One difference is that a "line" for this cache is 16 bytes instead of 4 bytes.

A less obvious 486 improvement is the five-stage instruction pipeline scheme that allows it to execute instructions much faster than a 386. This scheme, commonly used in RISC processors, allows several instructions to be "in the pipeline" at a time. The 486 will fetch several instructions ahead of time, and while it is executing one instruction, it will decode and as soon as possible, start the execution of the next instruction. The 486 may actually be executing parts of several instructions at the same time. For example, suppose the 486 is given the meaningless sequence of instructions:

```
MOV AX, MEMORY_LOCATION
ADD CX, BX
SHR AX, 1
MOV MEMORY_LOCATION, CX
```

A few clock cycles before it gets to the first of these instructions, the 486 will have prefetched all these simple instructions and started decoding them. As the MOV AX, MEMORY_LOCATION instruction executes, decoding of the ADD instruction will be completed. Since the ADD instruction does not uses the buses or the data read in from memory by the MOV instruction, it can be executed before the MOV AX instruction is complete. Likewise, decoding of the SHR instruction will be completed while the ADD CX, BX instruction is executing, and on the next clock cycle the SHR instruction will be executed. During the clock cycle that the SHR instruction executes, the decoding of the MOV MEMORY_LOCATION, CX instruction will be completed, and the address of the memory location will be output on the address bus. On the next clock cycle the word in AX will be output on the data bus. This extensive overlapping of operations makes it possible for the 486 to execute many of its commonly used instructions in, effectively, a single clock cycle. The fetching, decoding, and executing of each of these instructions actually takes several clock cycles, but since these operations are overlapped with the decoding and execution of other instructions, the net time for each of the instructions is only one clock cycle. As an example of this, a 16-bit memory-write operation that takes 22 clock cycles to execute on an 8088 and 4 clock cyles to execute on a 386 takes only 1 clock cycle to execute on a 486. The conditional jump instructions have also benefited greatly from the pipelining in the 486. When the 486 decodes a conditional jump instruction, it automatically prefetches one or more instructions from the jump destination address just in case the jump is taken. If the branch is taken, then the 486 does not have to wait through a bus cycle for the first instruction at the branch address. A conditional jump instruction which takes 16/4 clock cycles on an 8088 and 8/3 clock cycles on a 386 takes only 3/1 clock cycles on a 486.

Most of the other improvements included in the 486

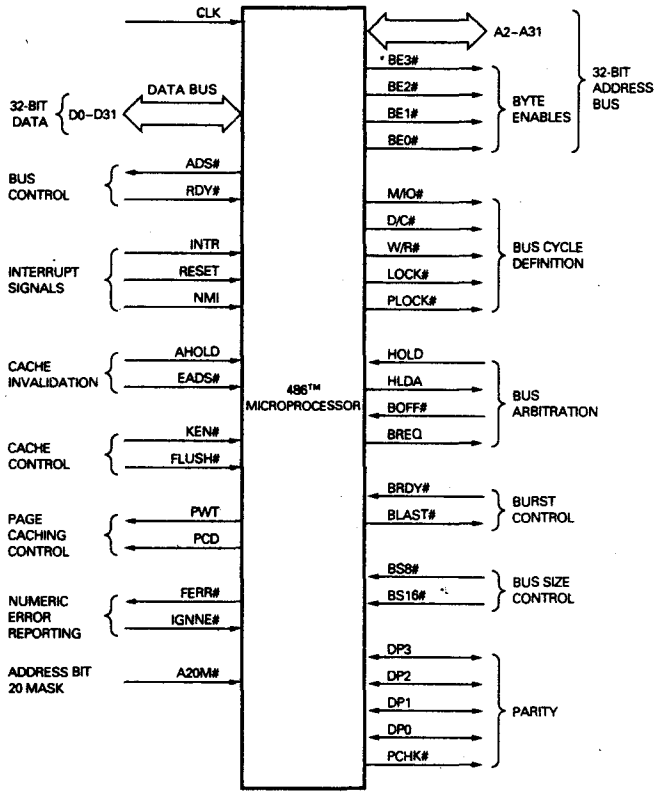


FIGURE 15-32 486 functional signal groups. (*Courtesy Intel Corporation*)

involve hardware signals and interfacing. To make room for the additional signals, the 486 is packaged in a 168-pin pin grid array package instead of the 132-pin PGA used for the 386. Figure 15-32 shows the 486 signals in functional groups. We will briefly work our way through some of these to give you an overview of the major new features.

The 486 data bus, address bus, byte enable, ADS#, RDY#, INTR, RESET, NMI, M/IO#, D/C#, W/R#, LOCK#, HOLD, HLDA, and BS16# signals function as we described for the 386, so these hold no surprises. The 486 requires a 1× clock instead of 2× clock required by the 386.

A new signal group on the 486 is the parity group, DP0–DP3, and PCHK#. These signals allow the 486 to implement parity detection/generation for memory reads and writes. During a memory write operation, the 486 generates an even parity bit for each byte and outputs these bits on the DP0–DP3 lines. As we described for the IBM PC in Chapter 11, these bits will be stored in a separate parity memory bank. During a read operation the stored parity bits will be read from the parity memory and applied to the DP0–DP3 pins. The 486 checks the parities of the data bytes read and compares them with the DP0–DP3 signals. If a parity error is found, the 486 asserts the PCHK# signal.

Another new signal group consists of the burst ready signal, BRDY#, and the burst last signal, BLAST#. These signals are used to control burst-mode memory reads and writes. Here's how this works. A normal 486 memory-read operation to, for example, read a line into the cache requires 2 clock cycles. However, if a series of reads is being done from successive memory locations, the reads can be done in burst mode with only 1 clock cycle per read. To start the process the 486 sends out the first address and asserts the BLAST# signal high. When the external DRAM controller has the first data word ready on the data bus, it asserts the BRDY# signal. The 486 reads the data word and outputs the next address. Since the data words are at successive addresses, only the lower address bits need to be changed. If the DRAM controller is operating in the page or the static column modes we described in Chapter 11, then it will only have to output a new column address to the DRAM. In this mode the DRAM will be able to output the new data word within 1 clock cycle. (If the DRAM is not fast enough for a high-speed 486, then two DRAM banks can be interleaved to gain the required speed.) When the processor has read the required number of data words, it asserts the BLAST# signal low to terminate the burst mode.

The final signals we want to discuss here are the bus request output signal, BREQ; the back-off input signal, BOFF#; the HOLD signal; and the hold-acknowledge signal, HLDA. These signals are used to control sharing the local 486 bus by multiple processors (bus masters). When a master on the bus needs to use the bus, it asserts its BREQ signal. An external priority circuit will evaluate requests to use the bus and grant bus use to the highest-priority master. To ask the 486 to release the bus, the bus controller asserts the 486 HOLD input or BOFF# input. If the HOLD input is asserted, the 486 will finish the current bus cycle, float its buses, and assert the HLDA signal. To prevent another master from taking over the bus during a critical operation, the 486 can assert its LOCK# or PLOCK# signal.

Because of all the possible variations, we don't have space here to discuss the operation of the 486 cache control signals. Consult the 486 data sheet if you need to know about these and the few other signals we didn't get around to.

The 486 has six additional instructions beyond those of the 386. INVD and WBINVD invalidate the cache, and INVLPG invalidates a TLB entry. The BSWAP instruction swaps the order of the bytes in a 32-bit register. This is useful in interfacing with, for example, an IBM mainframe which stores the least significant byte in the upper bits of a data word. The XADD and CMPXCHG instructions are used to work with semaphores such as those we showed you in Figure 15-3.

NEW DIRECTIONS

Microprocessor and microcomputer evolution has been proceeding very rapidly in the last few years, and the rate of evolution seems to be increasing. As we have shown in the preceding chapters, the overall direction of this evolution is toward microcomputers with greater screen resolution, more memory capability, larger data words, higher processing speeds, and network communication. David House of Intel recently revealed Intel's current plans for evolution beyond the 486. The 586,

expected in 1992, will contain about 2 million transistors, and the 686, expected in 1996, will contain about 5 million transistors. The added transistors, of course, will allow larger caches and many new functions to be implemented on the chips. The 786, to be available some time in the late 1990s, is projected to contain a 2-Mbyte cache, six separate integer and floating-point processors, and a complete digital video interactive or similar user interface. The 786 will maintain compatibility with the 386/486 instruction set and operate with a 250-MHz clock.

Throughout this book we have discussed the operation and evolution of primarily one processor family, the Intel 8086 family. We did this so that we could develop some depth rather than just an overview of all the different processors. To finish the book, however, we want to briefly discuss some other types of microcomputer systems that you should be aware of.

RISC Machines

High-performance engineering workstations often use a *reduced instruction set computer* or RISC-type processor. The term RISC is not precisely defined, but some of the main characteristics often associated with a RISC processor are the following:

1. The instruction set is limited to simple arithmetic-, logic-, load-, and store-type instructions. Fewer instructions and limited addressing modes mean a simpler and faster instruction decoder.
2. Extensive pipelining is used to achieve one-clock-cycle instruction execution. The 486 is a CISC processor, but as we described in a previous section, it uses a four-stage pipeline to achieve one-clock-cycle execution for many instructions. The assembler/compilers used for developing RISC programs are designed to put instructions in an order which will keep the pipeline full as much of the time as possible. To further overlap operations, some RISC machines use a *Harvard architecture*, which has separate data buses for code fetches and for data read/write operations.
3. Execution of conditional jump instructions is delayed to allow time to load the pipeline with instructions from the jump destination, or instructions from the jump destination are fetched ahead of time, as we described for the 486 in a previous section.
4. The CPU contains a large number of on-chip registers to give improved access to data operands.

One example of a current RISC implementation standard is the Scalable Processor Architecture RISC computer (SPARC) developed by Sun Microsystems and implemented in their SPARC stations. Fujitsu and Cypress Semiconductor have produced chip sets for this standard. Other common RISC chip sets are the Motorola 88000, the MIPS R3000, and the AMD 29000. A single-chip RISC processor now available is the Intel i860™.

The 1.2 million–transistor i860 contains a 64-bit RISC-based core with Harvard architecture, a floating-point coprocessor, and a graphics coprocessor with 3-D graphics capability. The processors in the i860™ operate relatively independently of each other, so they can all be working in parallel. With a 40-MHz clock an i860™, can perform at peak rate of 80 million floating-point operations per second (MFLOPS), or 85,000 drystones. (The drystone rating represents the relative performance of a computer executing a standard "benchmark" program.) Incidentally, the i860 does not use segmentation, but it does allow 386-type virtual memory paging. The data sheets for this device are a good source of information about RISC implementation.

Parallel Processing

Some computer applications such as analyzing weather data, simulating aircraft designs, or creating the graphics for high-tech science fiction movies require massive amounts of computing. The microcomputers we have discussed in this book so far do not operate nearly fast enough to be practical for many of these applications, so *supercomputers* are used. The peak execution speeds of the fastest current supercomputers are in the range of a few gigaFLOPS.

Most supercomputers are built by connecting several processing elements in parallel. One connection scheme, commonly called a *farm*, allows multiple processors to access a single large memory with a common bus. The difficulty with a simple bus structure such as this is that processors compete for shared resources. If one processor is using the bus, others must wait. This slows down the overall processing speed.

One of the more efficient multiprocessor architectures is the hypercube topology developed by Seitz and Fox at Caltech. A diagram of this topology is shown in Figure 15-33. Each node in the system consists of a complete processing unit which has the ability to communicate with other units. Each processor unit is typically connected to communicate with its nearest neighbors as

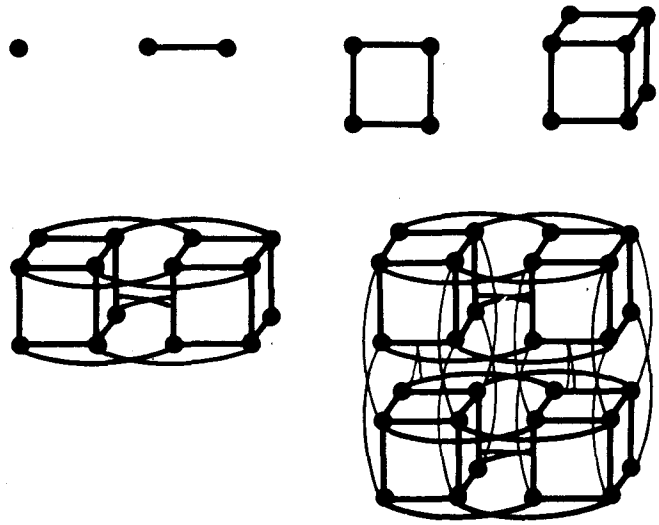


FIGURE 15-33 Hypercube connections for 1 to 32 processor nodes.

shown. The number of nodes can be expanded to give the power and speed needed to handle the problem the computer is being used to solve.

Intel Supercomputing Systems Division (ISSD) has produced the iPSC family of commercial products based on the hypercube topology. One of these, the iPSC/i860™ can be configured with up to 128 nodes, each containing a complete i860™-based microcomputer with high-speed network-type communication capability. The advantage of this structure is that each processor has enough memory to operate relatively independently, and communication between processors can take any one of several routes, instead of being limited to a single bus. Peak execution rates for the iPSC/i860™ range from 480 MFLOPS to 7.5 gigaFLOPS, depending on the number of processing units. These rates compare favorably with Cray Research Y-MP supercomputer's maximum execution rate of about 8 gigaFLOPS. However, because the iPSC/i860™ uses a larger number of common LSI components instead of a single or small number of very high speed processors made with gallium arsenide technology, the cost is less.

To program parallel computers such as these, new programming languages have had to be developed. A couple of these that you may be hearing more about are Scientific Computing Associates' C-Linda language and AIL Ltd.'s STRAND 88 language.

Expert Systems, Neural Networks, and Fuzzy Logic

INTRODUCTION

Artificial Intelligence or AI is the general term used to describe computers or computer programs which solve problems with "intuitive" or "best-guess" methods often used by humans instead of the strictly quantitative methods usually used by computers. Expert systems, neural networks, and fuzzy logic are the most common types of AI currently in use. These three are in relatively early stages of development and implementation, but they all have extensive implications for our lives.

EXPERT SYSTEMS

Probably the most developed area of AI at present is the area of *expert systems*. An expert-system program consists of a large data base and a set of rules for searching the data base to find the best solution for a particular type of problem. The data base and rules are developed by questioning "experts" in that particular problem area. The data base for a medical diagnosis expert system, for example, is built up by extensive questioning of experts in each medical specialty.

Unlike most computer programs, which require complete information to make a decision, expert system programs are designed to make a best guess, based on the available data, just as a human expert would do. A medical diagnosis expert system, for example, will indicate the illness that most likely corresponds to a given set of symptoms and test data. To enable it to make a better guess, the system may suggest additional tests to perform.

One advantage of a system such as this is that it can make the knowledge of many experts readily available to a physician anywhere in the world via a modem connection. Another advantage is that the data base and set of rules can be easily updated as new research results and drugs become available. Other examples of expert system programs are those used to lay out PC boards and those used to lay out ICs.

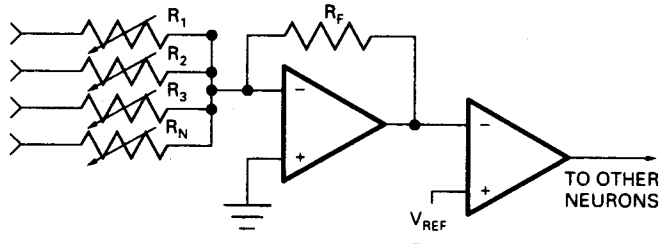
NEURAL NETWORKS

Programs for some problems such as image recognition, speech recognition, weather forecasting, and three-dimensional modeling are not easily or accurately implemented on fixed-instruction-set computers such as 386/i486-based systems. For applications such as these, a new computer architecture, modeled after the human brain, shows considerable promise.

As you may remember from a general science class, the brain is composed of billions of neurons. The output of each neuron is connected to the inputs of several thousand other neurons by synapses. If the sum of the signals on the inputs of a neuron is greater than a certain threshold value, the neuron "fires" and sends a signal to other neurons. The simple op-amp circuit in Figure 15-34*a* may help you see how a neuron works. Let's assume the output of the comparator is initially low. If the sum of the input signals to the adder produces an output voltage more negative than the comparator threshold voltage, the output of the comparator will go high. This is analogous to the neuron firing. The weight or relative influence of an input is determined by the value of the resistor on that input. Figure 15-34*b* shows a symbol commonly used to represent a neuron in neural network literature and Figure 15-34*c* shows a simple mathematical model of a neuron.

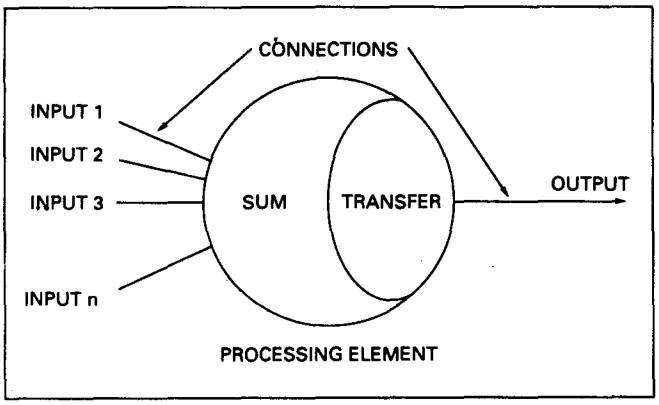
As with the neurons in the human brain, the neurons in a neural network are connected to many other neurons. Figure 15-34*d* shows a simple three-layer neural network. This network configuration is referred to as "feedforward," because none of the output signals are connected back to the inputs. In a "feedback" or "resonance" configured network, some intermediate or final output signals are connected back to network inputs. Researchers are currently experimenting with many different network configurations to determine the one that works best for each type of application.

Neural network based computing can be implemented in several ways. One way is to use a dedicated processor

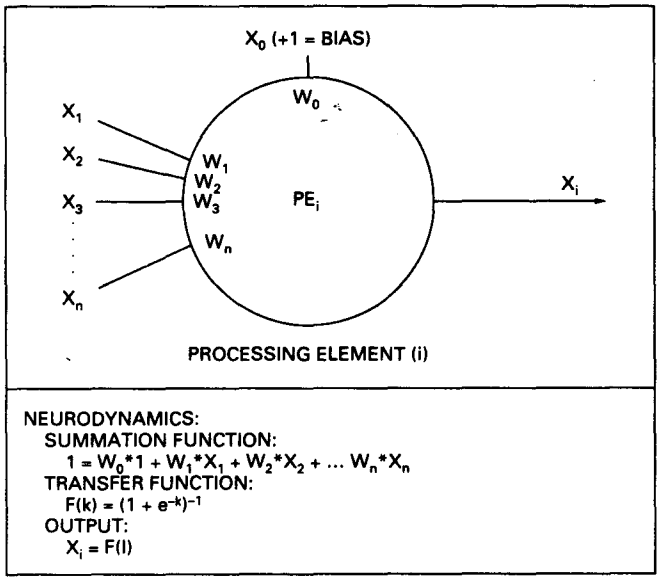


(*a*)

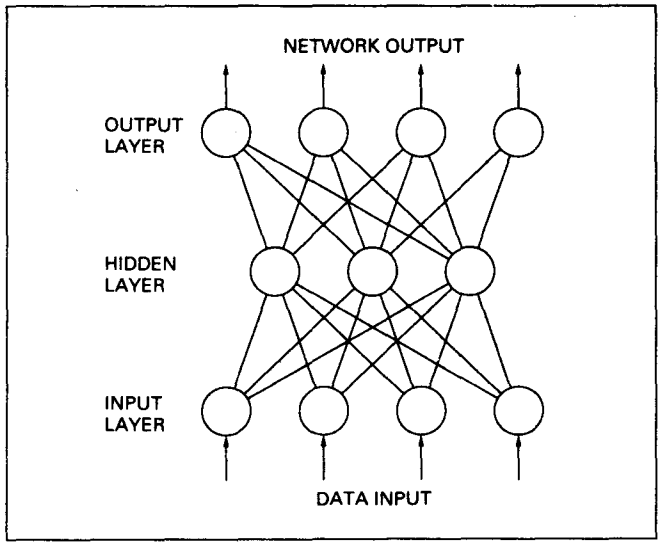
FIGURE 15-34 (*a*) Op-amp model of a neuron. (*See also next page.*)



(b)



(c)



(d)

FIGURE 15-34 (*Continued*) (*b*) Neural network processing element. (*c*) Mathematical representation of processing element. (*d*) Simple three-layer neural network. (*Courtesy NeuralWare, Inc.*)

for each neuron. The large number of neurons usually makes this impractical, and most applications don't need the speed capability. An alternative approach is to use a single processor and simulate neurons with lookup tables. The lookup table for each neuron contains the connections, input weight values, and output equation constants. Hecht-Nielsen Neurocomputers markets a PC/AT-compatible coprocessor board which uses this approach.

A neural network can also be implemented totally in software. NeuralWare, Inc. markets neural net simulation programs for both PC and Macintosh type computers. These packages can be used to learn about neural nets or develop actual applications which do not have to operate in real time. Another interesting neural network program is BrainMaker from California Scientific Software.

Neural network computers are not programmed in the way that digital computers are, they are trained. Instead of being programmed with a set of rules the way a classic expert system is, a neural network computer learns the desired behavior. The learning process may be supervised, unsupervised, or self-supervised.

In the supervised method a set of input conditions and the expected output conditions are applied to the network. The network learns by adjusting or "adapting" the weights of the interconnections until the output is correct. Another input-output set is then applied, and the network is allowed to learn this set. After a few sets the network will have learned or generalized its response so that it can give the correct response to most applied input data.

The scheme used to adapt the network is called the learning rule. As an example, one of the simplest learning rules that can be used is the Hebbean Learning Law. This law decrees that each time the input of a neuron contributes to the firing, its weight should be increased, and each time an input does not contribute, its weight should be decreased. This is somewhat analogous to a positive-negative reinforcement scheme often used in human behavior modification. In the case of the network the result is that these successive "nudges" adapt the network output to the desired result.

The major advantages of neural networks are these:

1. They do not need to be programmed; they can simply be taught the desired response. This eliminates most of the cost of programming.
2. They can improve their response by learning. A neural network designed to evaluate loan applications, for example, can automatically adapt its criteria based on loan-failure feedback data.
3. Input data does not have to be precise, because the network works with the sum of the inputs. A neural network image-recognition system, for example, can recognize a person even though he or she has a somewhat different hairstyle than when the "learning" image was taken. Likewise, a neural network-based speech-recognition system can recognize words spoken by different people. Traditional digital techniques have a very hard time with these tasks.

4. Information is not stored in a specific memory location the way it is in a normal digital computer; it is stored *associatively* as a network of interconnections and weightings. The result of this is that the "death" of a few neurons will usually not seriously degrade the operation of the system. This characteristic is also fortunate for us humans!

Software-based neural networks can be used for nonrealtime applications such as forecasting the weather or the stock market. For realtime applications such as image recognition and speech recognition, the software methods are obviously not fast enough. University researchers and companies such as TRW and Texas Instruments are working on ICs which implement neural networks in hardware. In the not-too-distant future these ICs should allow you to talk to your computer instead of using a mouse, allow your computer to read typed messages to you, and allow your car to drive itself down the freeway.

FUZZY LOGIC

Consumer products such as video camcorders, cameras, refrigerators, washing machines, and automobiles are increasingly using fuzzy logic control circuits. Linking the term fuzzy, which here means "not precisely defined," with the term logic may seem to create an oxymoron like "work party," but the concept is very real. The original work on fuzzy logic was done by Professor Lofti A. Zadeh at U.C. Berkeley in the mid-1960s, but Japanese companies have been the main ones to patent the technology and implement it in products.

A fuzzy logic controller is programmed with rules as is an expert system, but the rules are very flexible. Figure 15-35 shows the graphic method Professor Bart Kosko of the University of Southern California uses to illustrate the difference between traditional fixed value logic and fuzzy logic. Each corner of the cube represents one of the eight possibilities for a three-variable digital logic function. For this example, let's assume that the function is true for the 010, 001, and 100 combinations shown. In a traditional digital logic system the variables can only have values of 0 or 1, so the only values that will produce a true output are these three. In a fuzzy logic system the variables can have values other than 1 or 0, so the set of all the possible values that will produce a true output is represented by the triangular plane formed by the three points. One way to look at this is that traditional digital logic is just a special case of fuzzy logic.

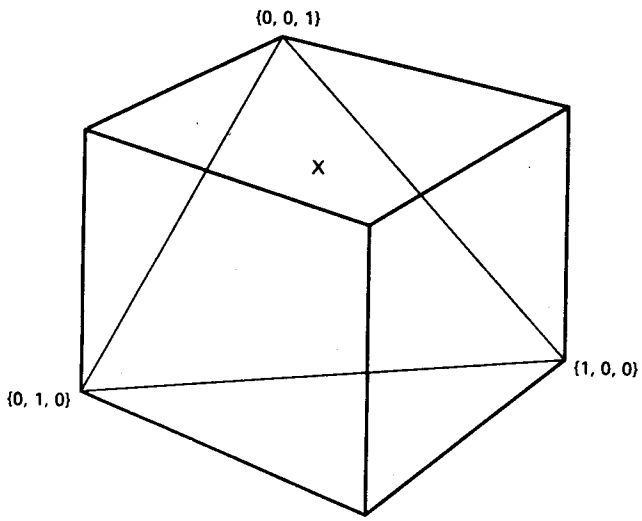


FIGURE 15-35 Comparison of binary logic values and fuzzy logic values for a 3-input function.

One advantage of fuzzy logic systems is that they can work with imprecise terms such as cold, warm, hot, or near boiling that humans commonly use. In hardware terms this means that a fuzzy logic system often doesn't need precise A/D converters. The Sanyo Fisher Corp. Model FVC-880 camcorder, for example, uses fuzzy logic to directly process the outputs from six sensors and set the camera lens for best possible focusing and exposure.

Fuzzy logic can provide very smooth control of mechanical systems. The fuzzy logic-controlled subway in Sendai, Japan, is reportedly so smooth in operation that standing riders do not use the hand straps during starts and stops.

In the United States, Togai InfraLogic, Inc. in Irvine, California, has developed a Digital Fuzzy Processor chip. They have also developed a Fuzzy-C compiler which can be used to write a program containing the rules and knowledge base for the processor.

SUMMARY

The three AI approaches we have discussed in this section will obviously not replace standard digital computers for most applications, especially those that involve numerical processing, but they do give some new choices for difficult applications. The most likely scenario for the future is that a combination of these techniques will be used to design a system which best fits the particular application. The results should be very exciting.

EPILOGUE

This book has been able to show you only a small view of current microcomputers and the directions in which they seem to be evolving. Hopefully we have given you enough of a start that you can continue learning on your own and play a part in the evolution. Whenever you feel overwhelmed by the amount of new material there is to learn, remember the 5-minute rule and the old saying "Grapevines and people bear the best fruit on new growth."

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms in the following list, use the index to help you find them in the chapter for review.

Multiuser, multitasking

TSR program

Time-slice and preemptive priority-based scheduling

Semaphore

Deadlock

Critical region

Overlay

Bank switching

Expanded memory

Extended memory

Descriptor table

Virtual memory

Memory-Management Unit

80286
- Real address mode
- Protected virtual address mode
- Interrupt, exception, fault, trap

80386
- Architecture, pins, and signals
- System connections and interface buses
  - ISA bus standard
  - EISA bus standard
  - MicroChannel Architecture Bus
- Real-mode operation
- Protected-mode operation
  - Segmentation and virtual memory
  - Segment privilege levels and protection
  - Call gate
  - I/O privilege levels
  - Interrupt and exception handling
  - Task switching and task state segment
  - Paging mode
  - Flat system memory model
- Virtual 8086 mode operation
- Virtual machine monitor
- Index scaling
- OS/2
- Microsoft windows

80486
- Pipelining
- Cache
- Floating-point processor

Parallel processing
- Hypercube topology

RISC machine

Artifical intelligence

Expert system

Neural network

Fuzzy logic

5-minute rule

Grapevines and new growth

REVIEW QUESTIONS AND PROBLEMS

1. *a.* Describe the basic operation of a TSR program and draw a memory map to show how a TSR program is loaded in a DOS based system.
   b. Use a diagram to help explain how a passive TSR gets executed after it is installed.
2. Briefly describe the two types of scheduling commonly used in multiuser/multitasking operating systems.
3. Suppose that two users in a time-share computer system each want to print out a file. How can the system be prevented from printing lines from one file between lines of the other file?
4. Define the term deadlock and describe one way it can be prevented.
5. Define the term critical region and show with 8086 assembly language instructions how a semaphore can be used to protect a critical region.
6. Describe how an overlay scheme is used to run programs such as compilers which are too large to be loaded into physical memory all at once.
7. *a.* Describe how bank switching is implemented in a microcomputer system.
   b. Describe how LIM 4.0-type expanded memory works in a DOS-based system.
   c. How is extended memory different from expanded memory?
8. *a.* Define the term virtual memory and use Figure 15-8 to help you briefly describe how a logical address is converted to a physical address by a memory management unit.
   b. What action will the MMU take if it finds that a requested segment is not present in physical memory?
   c. What is another major advantage of the indirect addressing provided by descriptor tables, besides the ability to address a large amount of virtual memory?
9. List the four major processing units in an 80286 microprocessor and briefly describe the function of each.
10. Describe how the real-mode operation of an 80286 is different from protected-mode operation.
11. Define the terms interrupt, exception, fault, and trap.

12. Explain how an 80286 is switched from real address mode to protected virtual address mode and how it is switched back to real address mode operation.

13. *a.* Show the computations which tell how much virtual memory an 80286 can address.
    b. What factors determine how much physical memory an 80286 can address?

14. *a.* List three major advances that the 80386 microprocessor has over the 80286.
    b. What is the main difference between the 386DX processor and the 386SX processor?
    c. What is the purpose of the 386DX BE0-BE3 signals?

15. *a.* How is the EISA bus different from the ISA bus?
    b. If you found an interface board lying on the bench, what is one way you could tell whether it came from an EISA-based system or from a MicroChannel Architecture system?
    c. Briefly compare the EISA and MCA methods of arbitrating bus requests from multiple masters or DMA slaves.

16. *a.* Show the computations which tell how much virtual memory a 386 can address.
    b. How much physical memory can a 386 address in real mode and in protected mode?

17. *a.* Give the names of the two parts of a 386 protected-mode address.
    b. Using Figure 15-21 to help, describe how a 32-bit virtual address for a data segment location in a task's local memory is translated to the actual 32-bit physical address for a 386 operating in segments only protected mode.
    c. How would the discussion in part b differ if the desired memory location were in the global memory area?
    d. How does a 386 keep track of where the global descriptor table and the currently used local descriptor table are located in memory?
    e. Why is the length of the segment included in the descriptor for a segment?

18. How are tasks in a 386 system protected from each other?

19. How can operating system kernel procedures and data be protected from access by application programs in a 386 system?

20. In a 386 system a task operating at a level 2 privilege can in a special way call a procedure at a higher privilege level. Describe briefly the mechanism that is used to make this access.

21. *a.* A 386 maintains a task state segment for each active task in a system. How are these task state segments accessed?
    b. Briefly describe how a 386 does a task switch using a FAR JMP or a FAR CALL instruction.

22. *a.* Use Figure 15-26 to help you explain how a 386 computes a physical address when its paging mode is enabled.
    b. What is the advantage of paged-based virtual memory over segments only based virtual memory?
    c. Define the term simple flat memory model and the term paged flat memory model for a 386.

23. *a.* How is a 386 switched into virtual 8086 mode during a task switch?
    b. Briefly describe the response of the virtual machine monitor when a real-mode 8086 program executes an INT 21H instruction.

24. Using the program in Figure 15-29*a* as a model, write a program which uses some of the new 386 instruction features to treat the four words in table as a 64-bit word, and rotate it 8 bits to the left.

25. Describe three major additions or improvements that the 486 processor has over the 386 processor.

26. List three major features characteristic of a RISC-based computer and describe how each of these features helps produce faster execution.

27. What are the major advantages of using parallel processors with, for example, a hypercube connection architecture over using a single fast processor?

28. *a.* Describe the basic operation of a neuron in a neural computer.
    b. How is the "programming" of a neural network computer different from the programming of a standard, fixed-instruction-set computer?
    c. List some advantages of a neural network–type computer.
    d. For what types of applications are neural network–type computers best suited?

29. *a.* How is a fuzzy logic control system different from a traditional digital logic control system?
    b. What are some advantages of fuzzy logic?