One of the basic features of a computer is its ability to exchange data with other devices. This communication capability enables a human operator, for example, to use a keyboard and a display screen to process text and graphics. We make extensive use of computers to communicate with other computers over the Internet and access information around the globe. In other applications, computers are less visible but equally important. They are an integral part of home appliances, manufacturing equipment, transportation systems, banking and point-of-sale terminals. In such applications, input to a computer may come from a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be a sound signal to be sent to a speaker or a digitally coded command to change the speed of a motor, open a valve, or cause a robot to move in a specified manner. In short, a general-purpose computer should have the ability to exchange information with a wide range of devices in varying environments.

In this chapter, we will consider in detail various ways in which I/O operations are performed. First, we will consider the problem from the point of view of the programmer. Then, we will discuss some of the hardware details associated with buses and I/O interfaces and introduce some commonly used bus standards.

4.1 ACCESSING I/O DEVICES

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement, as shown in Figure 4.1. The bus enables all the devices connected to it to exchange information. Typically, it consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines. As mentioned in Section 2.7, when I/O devices and the memory share the same address space, the arrangement is called *memory-mapped I/O*.

With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address

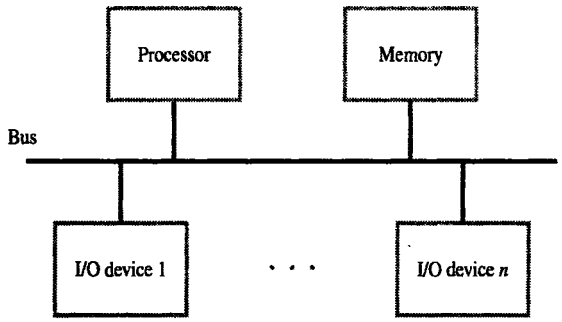


Figure 4.1 A single-bus structure.

of the input buffer associated with the keyboard, the instruction

```
Move    DATAIN,R0
```

reads the data from DATAIN and stores them into processor register R0. Similarly, the instruction

```
Move    R0,DATAOUT
```

sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer.

Most computer systems use memory-mapped I/O. Some processors have special In and Out instructions to perform I/O transfers. For example, processors in the Intel family described in Chapter 3 have special I/O instructions and a separate 16-bit address space for I/O devices. When building a computer system based on these processors, the designer has the option of connecting I/O devices to use the special I/O address space or simply incorporating them as part of the memory address space. The latter approach is by far the most common as it leads to simpler software. One advantage of a separate I/O address space is that I/O devices deal with fewer address lines. Note that a separate I/O address space does not necessarily mean that the I/O address lines are physically separate from the memory address lines. A special signal on the bus indicates that the requested read or write transfer is an I/O operation. When this signal is asserted, the memory unit ignores the requested transfer. The I/O devices examine the low-order bits of the address bus to determine whether they should respond.

Figure 4.2 illustrates the hardware required to connect an I/O device to the bus. The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and

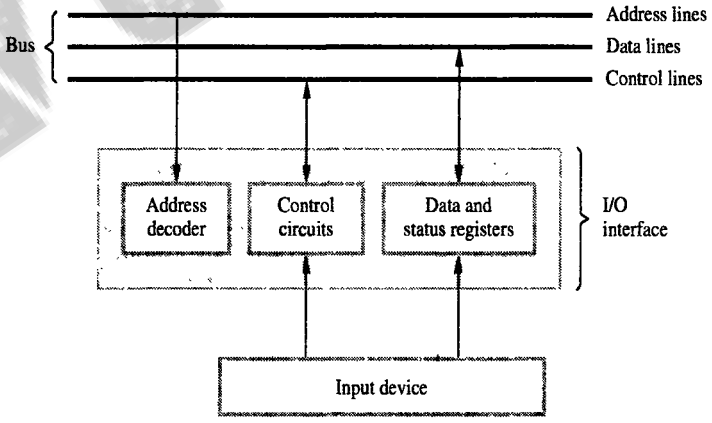


Figure 4.2 I/O interface for an input device.

assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's *interface circuit*.

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. Also, we must make sure that an input character is read only once.

The basic ideas used for performing input and output operations were introduced in Section 2.7. For an input device such as a keyboard, a status flag, SIN, is included in the interface circuit as part of the status register. This flag is set to 1 when a character is entered at the keyboard and cleared to 0 once this character is read by the processor. Hence, by checking the SIN flag, the software can ensure that it is always reading valid data. This is often accomplished in a program loop that repeatedly reads the status register and checks the state of SIN. When SIN becomes equal to 1, the program reads the input data register. A similar procedure can be used to control output operations using an output status flag, SOUT.

Example 4.1 To review the basic concepts, let us consider a simple example of I/O operations involving a keyboard and a display device in a computer system. The four registers shown in Figure 4.3 are used in the data transfer operations. Register STATUS contains two control flags, SIN and SOUT, which provide status information for the keyboard and the display unit, respectively. The two flags KIRQ and DIRQ in this register are used in conjunction with interrupts. They, and the KEN and DEN bits in register CONTROL, will be discussed in Section 4.2. Data from the keyboard are made available

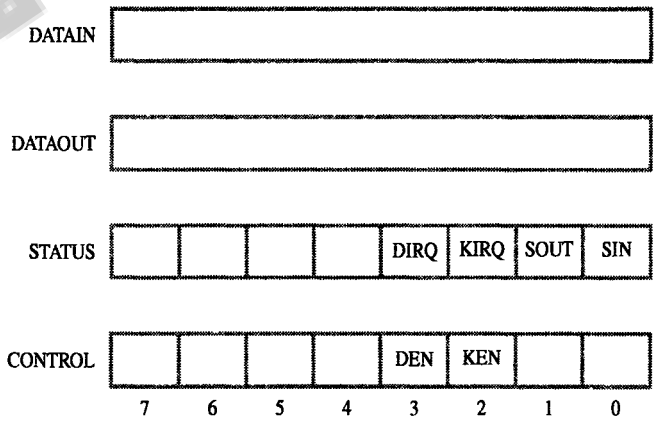


Figure 4.3 Registers in keyboard and display interfaces.

```
        Move      #LINE,R0          Initialize memory pointer.
WAITK   TestBit   #0,STATUS         Test SIN.
        Branch=0  WAITK             Wait for character to be entered.
        Move      DATAIN,R1         Read character.
WAITD   TestBit   #1,STATUS         Test SOUT.
        Branch=0  WAITD             Wait for display to become ready.
        Move      R1,DATAOUT        Send character to display.
        Move      R1,(R0)+          Store charater and advance pointer.
        Compare   #$0D,R1           Check if Carriage Return.
        Branch≠0  WAITK             If not, get another character.
        Move      #$0A,DATAOUT      Otherwise, send Line Feed.
        Call      PROCESS           Call a subroutine to process the
                                      the input line.
```

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

in the DATAIN register, and data sent to the display are stored in the DATAOUT register.

The program in Figure 4.4 is similar to that in Figure 2.20. This program reads a line of characters from the keyboard and stores it in a memory buffer starting at location LINE. Then, it calls a subroutine PROCESS to process the input line. As each character is read, it is *echoed back* to the display. Register R0 is used as a pointer to the memory buffer area. The contents of R0 are updated using the Autoincrement addressing mode so that successive characters are stored in successive memory locations.

Each character is checked to see if it is the Carriage Return (CR) character, which has the ASCII code 0D (hex). If it is, a Line Feed character (ASCII code 0A) is sent to move the cursor one line down on the display and subroutine PROCESS is called. Otherwise, the program loops back to wait for another character from the keyboard.

This example illustrates *program-controlled I/O*, in which the processor repeatedly checks a status flag to achieve the required synchronization between the processor and an input or output device. We say that the processor *polls* the device. There are two other commonly used mechanisms for implementing I/O operations: interrupts and direct memory access. In the case of interrupts, synchronization is achieved by having the I/O device send a special signal over the bus whenever it is ready for a data transfer operation. Direct memory access is a technique used for high-speed I/O devices. It involves having the device interface transfer data directly to or from the memory, without continuous involvement by the processor. We will discuss these mechanisms in the next three sections. Then, we will examine the hardware involved, which includes the processor bus and the I/O device interface.

4.2 INTERRUPTS

In the example of Figure 4.4, the program enters a wait loop in which it repeatedly tests the device status. During this period, the processor is not performing any useful computation. There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready. It can do so by sending a hardware signal called an *interrupt* to the processor. At least one of the bus control lines, called an *interrupt-request* line, is usually dedicated for this purpose. Since the processor is no longer required to continuously check the status of external devices, it can use the waiting period to perform other useful functions. Indeed, by using interrupts, such waiting periods can ideally be eliminated.

Example 4.2

Consider a task that requires some computations to be performed and the results to be printed on a line printer. This is followed by more computations and output, and so on. Let the program consist of two routines, COMPUTE and PRINT. Assume that COMPUTE produces a set of n lines of output, to be printed by the PRINT routine.

The required task may be performed by repeatedly executing first the COMPUTE routine and then the PRINT routine. The printer accepts only one line of text at a time. Hence, the PRINT routine must send one line of text, wait for it to be printed, then send the next line, and so on, until all the results have been printed. The disadvantage of this simple approach is that the processor spends a considerable amount of time waiting for the printer to become ready. If it is possible to overlap printing and computation, that is, to execute the COMPUTE routine while printing is in progress, a faster overall speed of execution will result. This may be achieved as follows. First, the COMPUTE routine is executed to produce the first n lines of output. Then, the PRINT routine is executed to send the first line of text to the printer. At this point, instead of waiting for the line to be printed, the PRINT routine may be temporarily suspended and execution of the COMPUTE routine continued. Whenever the printer becomes ready, it alerts the processor by sending an interrupt-request signal. In response, the processor interrupts execution of the COMPUTE routine and transfers control to the PRINT routine. The PRINT routine sends the second line to the printer and is again suspended. Then the interrupted COMPUTE routine resumes execution at the point of interruption. This process continues until all n lines have been printed and the PRINT routine ends.

The PRINT routine will be restarted whenever the next set of n lines is available for printing. If COMPUTE takes longer to generate n lines than the time required to print them, the processor will be performing useful computations all the time.

This example illustrates the concept of interrupts. The routine executed in response to an interrupt request is called the *interrupt-service routine,* which is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in Figure 4.5. The

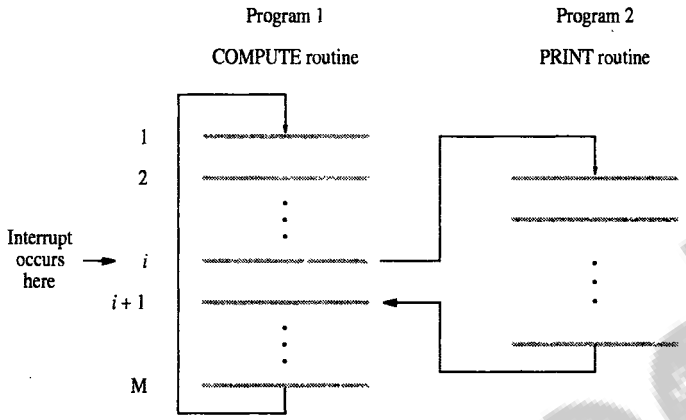


Figure 4.5 Transfer of control through the use of interrupts.

processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor has to come back to instruction $i + 1$. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction $i + 1$, must be put in temporary storage in a known location. A Return-from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction $i + 1$. In many processors, the return address is saved on the processor stack. Alternatively, it may be saved in a special location, such as a register provided for this purpose.

We should note that as part of handling interrupts, the processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. This may be accomplished by means of a special control signal on the bus. An *interrupt-acknowledge* signal, used in some of the interrupt schemes to be discussed later, serves this function. A common alternative is to have the transfer of data between the processor and the I/O device interface accomplish the same purpose. The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.

So far, treatment of an interrupt-service routine is very similar to that of a subroutine. An important departure from this similarity should be noted. A subroutine performs a function required by the program from which it is called. However, the interrupt-service routine may not have anything in common with the program being executed at the time the interrupt request is received. In fact, the two programs often belong to different users. Therefore, before starting execution of the interrupt-service routine, any information that may be altered during the execution of that routine must be saved. This information must be restored before execution of the interrupted program is resumed. In this way, the original program can continue execution without being affected in any

way by the interruption, except for the time delay. The information that needs to be saved and restored typically includes the condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine.

The task of saving and restoring information can be done automatically by the processor or by program instructions. Most modern processors save only the minimum amount of information needed to maintain the integrity of program execution. This is because the process of saving and restoring registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. This delay is called *interrupt latency*. In some applications, a long interrupt latency is unacceptable. For these reasons, the amount of information saved automatically by the processor when an interrupt request is accepted should be kept to a minimum. Typically, the processor saves only the contents of the program counter and the processor status register. Any additional information that needs to be saved must be saved by program instructions at the beginning of the interrupt-service routine and restored at the end of the routine.

In some earlier processors, particularly those with a small number of registers, all registers are saved automatically by the processor hardware at the time an interrupt request is accepted. The data saved are restored to their respective registers as part of the execution of the Return-from interrupt instruction. Some computers provide two types of interrupts. One saves all register contents, and the other does not. A particular I/O device may use either type, depending upon its response-time requirements. Another interesting approach is to provide duplicate sets of processor registers. In this case, a different set of registers can be used by the interrupt-service routine, thus eliminating the need to save and restore registers.

An interrupt is more than a simple mechanism for coordinating I/O transfers. In a general sense, interrupts enable transfer of control from one program to another to be initiated by an event external to the computer. Execution of the interrupted program resumes after the execution of the interrupt-service routine has been completed. The concept of interrupts is used in operating systems and in many control applications where processing of certain routines must be accurately timed relative to external events. The latter type of application is referred to as *real-time processing*.

4.2.1 INTERRUPT HARDWARE

We pointed out that an I/O device requests an interrupt by activating a bus line called interrupt-request. Most computers are likely to have several I/O devices that can request an interrupt. A single interrupt-request line may be used to serve n devices as depicted in Figure 4.6. All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt-request signals $INTR_1$ to $INTR_n$ are inactive, that is, if all switches are open, the voltage on the interrupt-request line will be equal to V_{dd}. This is the inactive state of the line. When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt-request signal, INTR, received by the processor to go to 1. Since the closing of one or more switches will cause the line voltage to drop to 0, the value

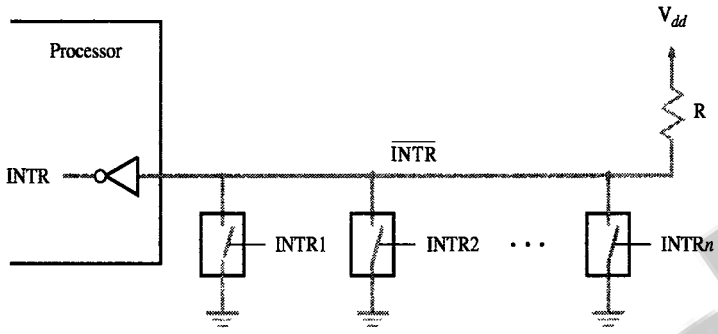


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

of INTR is the logical OR of the requests from individual devices, that is,

$$INTR = INTR_1 + \cdots + INTR_n$$

It is customary to use the complemented form, $\overline{INTR}$, to name the interrupt-request signal on the common line, because this signal is active when in the low-voltage state.

In the electronic implementation of the circuit in Figure 4.6, special gates known as *open-collector* (for bipolar circuits) or *open-drain* (for MOS circuits) are used to drive the $\overline{INTR}$ line. The output of an open-collector or an open-drain gate is equivalent to a switch to ground that is open when the gate's input is in the 0 state and closed when it is in the 1 state. The voltage level, hence the logic state, at the output of the gate is determined by the data applied to all the gates connected to the bus, according to the equation given above. Resistor R is called a *pull-up resistor* because it pulls the line voltage up to the high-voltage state when the switches are open.

4.2.2 Enabling and Disabling Interrupts

The facilities provided in a computer must give the programmer complete control over the events that take place during program execution. The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another. Because interrupts can arrive at any time, they may alter the sequence of events from that envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled. A fundamental facility found in all computers is the ability to enable and disable such interruptions as desired. We will now examine this and related facilities in some detail.

There are many situations in which the processor should ignore interrupt requests. For example, in the case of the Compute-Print program of Figure 4.5, an interrupt request from the printer should be accepted only if there are output lines to be printed. After printing the last line of a set of n lines, interrupts should be disabled until another set becomes available for printing. In another case, it may be necessary to guarantee that

a particular sequence of instructions is executed to the end without interruption because the interrupt-service routine may change some of the data used by the instructions in question. For these reasons, some means for enabling and disabling interrupts must be available to the programmer. A simple way is to provide machine instructions, such as Interrupt-enable and Interrupt-disable, that perform these functions.

Let us consider in detail the specific case of a single interrupt request from one device. When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request. This means that the interrupt-request signal will be active during execution of the interrupt-service routine, perhaps until an instruction is reached that accesses the device in question. It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop from which it cannot recover. Several mechanisms are available to solve this problem. We will describe three possibilities here; other schemes that can handle more than one interrupting device will be presented later.

The first possibility is to have the processor hardware ignore the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed. Then, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, the Interrupt-enable instruction will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction. The processor must guarantee that execution of the Return-from-interrupt instruction is completed before further interruption can occur.

The second option, which is suitable for a simple processor with only one interrupt-request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. After saving the contents of the PC and the processor status register (PS) on the stack, the processor performs the equivalent of executing an Interrupt-disable instruction. It is often the case that one bit in the PS register, called *Interrupt-enable,* indicates whether interrupts are enabled. An interrupt request received while this bit is equal to 1 will be accepted. After saving the contents of the PS on the stack, with the Interrupt-enable bit equal to 1, the processor clears the Interrupt-enable bit in its PS register, thus disabling further interrupts. When a Return-from-interrupt instruction is executed, the contents of the PS are restored from the stack, setting the Interrupt-enable bit back to 1. Hence, interrupts are again enabled.

In the third option, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be *edge-triggered.* In this case, the processor will receive only one request, regardless of how long the line is activated. Hence, there is no danger of multiple interruptions and no need to explicitly disable interrupt requests from this line.

Before proceeding to study more complex aspects of interrupts, let us summarize the sequence of events involved in handling an interrupt request from a single device. Assuming that interrupts are enabled, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.

3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge-triggered interrupts).
4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

4.2.3 HANDLING MULTIPLE DEVICES

Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions:

1. How can the processor recognize the device requesting an interrupt?
2. Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?

The means by which these problems are resolved vary from one computer to another, and the approach taken is an important consideration in determining the computer's suitability for a given application.

When a request is received over the common interrupt-request line in Figure 4.6, additional information is needed to identify the particular device that activated the line. Furthermore, if two devices have activated the line at the same time, it must be possible to break the tie and select one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced.

The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request, it sets to 1 one of the bits in its status register, which we will call the IRQ bit. For example, bits KIRQ and DIRQ in Figure 4.3 are the interrupt request bits for the keyboard and the display, respectively. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all the I/O devices connected to the bus. The first device encountered with its IRQ bit set is the device that should be serviced. An appropriate subroutine is called to provide the requested service.

The polling scheme is easy to implement. Its main disadvantage is the time spent interrogating the IRQ bits of all the devices that may not be requesting any service. An alternative approach is to use vectored interrupts, which we describe next.

Vectored Interrupts

To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. The term *vectored interrupts* refers to all interrupt-handling schemes based on this approach.

A device requesting an interrupt can identify itself by sending a special code to the processor over the bus. This enables the processor to identify individual devices even if they share a single interrupt-request line. The code supplied by the device may represent the starting address of the interrupt-service routine for that device. The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine. In many computers, this is done automatically by the interrupt-handling mechanism. The location pointed to by the interrupting device is used to store the starting address of the interrupt-service routine. The processor reads this address, called the *interrupt vector,* and loads it into the PC. The interrupt vector may also include a new value for the processor status register.

In most computers, I/O devices send the interrupt-vector code over the data bus, using the bus control signals to ensure that devices do not interfere with each other. When a device sends an interrupt request, the processor may not be ready to receive the interrupt-vector code immediately. For example, it must first complete the execution of the current instruction, which may require the use of the bus. There may be further delays if interrupts happen to be disabled at the time the request is raised. The interrupting device must wait to put data on the bus only when the processor is ready to receive it. When the processor is ready to receive the interrupt-vector code, it activates the interrupt-acknowledge line, INTA. The I/O device responds by sending its interrupt-vector code and turning off the INTR signal.

Interrupt Nesting

We suggested in Section 4.2.1 that interrupts should be disabled during the execution of an interrupt-service routine, to ensure that a request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved, in which case execution of a given interrupt-service routine, once started, always continues to completion before the processor accepts an interrupt request from a second device. Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices.

For some devices, however, a long delay in responding to an interrupt request may lead to erroneous operation. Consider, for example, a computer that keeps track of the time of day using a real-time clock. This is a device that sends interrupt requests to the processor at regular intervals. For each of these requests, the processor executes a short interrupt-service routine to increment a set of counters in the memory that keep track of time in seconds, minutes, and so on. Proper operation requires that the delay in responding to an interrupt request from the real-time clock be small in comparison

with the interval between two successive requests. To ensure that this requirement is satisfied in the presence of other interrupting devices, it may be necessary to accept an interrupt request from the clock during the execution of an interrupt-service routine for another device.

This example suggests that I/O devices should be organized in a priority structure. An interrupt request from a high-priority device should be accepted while the processor is servicing another request from a lower-priority device.

A multiple-level priority organization means that during execution of an interrupt-service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities higher than its own. At the time the execution of an interrupt-service routine for some device is started, the priority of the processor is raised to that of the device. This action disables interrupts from devices at the same level of priority or lower. However, interrupt requests from higher-priority devices will continue to be accepted.

The processor's priority is usually encoded in a few bits of the processor status word. It can be changed by program instructions that write into the PS. These are *privileged* instructions, which can be executed only while the processor is running in the supervisor mode. The processor is in the supervisor mode only when executing operating system routines. It switches to the user mode before beginning to execute application programs. Thus, a user program cannot accidentally, or intentionally, change the priority of the processor and disrupt the system's operation. An attempt to execute a privileged instruction while in the user mode leads to a special type of interrupt called a *privilege exception*, which we describe in Section 4.2.5.

A multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-acknowledge lines for each device, as shown in Figure 4.7. Each of the interrupt-request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

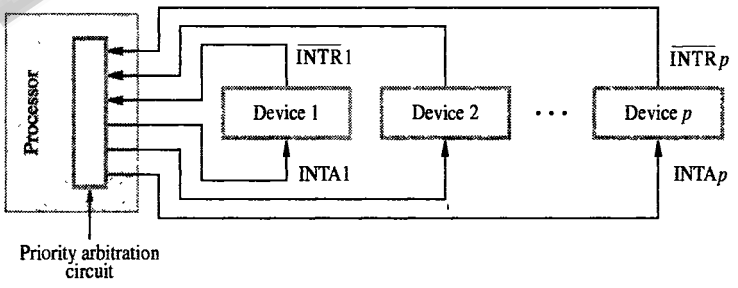
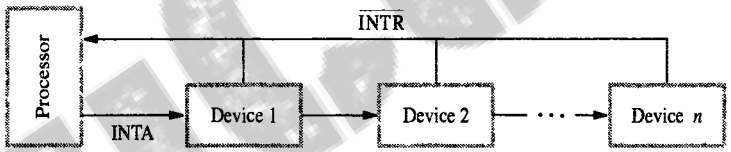


Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

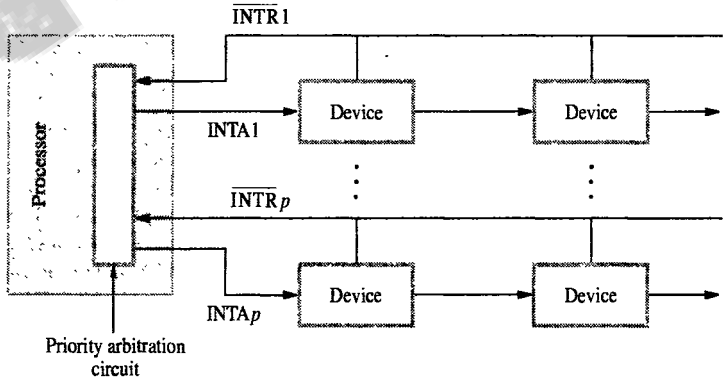
Simultaneous Requests

Let us now consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which request to service first. Using a priority scheme such as that of Figure 4.7, the solution is straightforward. The processor simply accepts the request having the highest priority. If several devices share one interrupt-request line, as in Figure 4.6, some other mechanism is needed.

Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. A widely used scheme is to connect the devices to form a *daisy chain*, as shown in Figure 4.8*a*. The interrupt-request line $\overline{\text{INTR}}$ is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. When several devices raise an interrupt request and the $\overline{\text{INTR}}$ line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1. Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for



(a) Daisy chain



(b) Arrangement of priority groups

Figure 4.8 Interrupt priority schemes.

interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority. The second device along the chain has second highest priority, and so on.

The scheme in Figure 4.8*a* requires considerably fewer wires than the individual connections in Figure 4.7. The main advantage of the scheme in Figure 4.7 is that it allows the processor to accept interrupt requests from some devices but not from others, depending upon their priorities. The two schemes may be combined to produce the more general structure in Figure 4.8*b*. Devices are organized in groups, and each group is connected at a different priority level. Within a group, devices are connected in a daisy chain. This organization is used in many computer systems.

4.2.4 CONTROLLING DEVICE REQUESTS

Until now, we have assumed that an I/O device interface generates an interrupt request whenever it is ready for an I/O transfer, for example whenever the SIN flag in Figure 4.3 is equal to 1. It is important to ensure that interrupt requests are generated only by those I/O devices that are being used by a given program. Idle devices must not be allowed to generate interrupt requests, even though they may be ready to participate in I/O transfer operations. Hence, we need a mechanism in the interface circuits of individual devices to control whether a device is allowed to generate an interrupt request.

The control needed is usually provided in the form of an interrupt-enable bit in the device's interface circuit. The keyboard interrupt-enable, KEN, and display interrupt-enable, DEN, flags in register CONTROL in Figure 4.3 perform this function. If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status flag in register STATUS is set. At the same time, the interface circuit sets bit KIRQ or DIRQ to indicate that the keyboard or display unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag.

To summarize, there are two independent mechanisms for controlling interrupt requests. At the device end, an interrupt-enable bit in a control register determines whether the device is allowed to generate an interrupt request. At the processor end, either an interrupt enable bit in the PS register or a priority structure determines whether a given interrupt request will be accepted.

Example 4.3

Consider a processor that uses the vectored interrupt scheme, where the starting address of the interrupt-service routine is stored at memory location INTVEC. Interrupts are enabled by setting to 1 an interrupt-enable bit, IE, in the processor status word, which we assume is bit 9. A keyboard and a display unit connected to this processor have the status, control, and data registers shown in Figure 4.3.

Assume that at some point in a program called Main we wish to read an input line from the keyboard and store the characters in successive byte locations in the

Main program

```
        MOV     EOL,0
        MOV     BL,4
        OR      CONTROL,BL      Set KEN to enable keyboard interrupts.
        STI                     Set interrupt flag in processor register.
        .
        .
        .
```

Interrupt-service routine

```
READ    PUSH    EAX             Save register EAX on stack.
        PUSH    EBX             Save register EBX on stack.
        MOV     EAX,PNTR        Load address pointer.
        MOV     BL,DATAIN       Get input character.
        MOV     [EAX],BL        Store character.
        INC     DWORD PTR [EAX] Increment PNTR.
        CMP     BL,0DH          Check if character is CR.
        JNE     RTRN
        MOV     BL,4
        XOR     CONTROL,BL      Clear bit KEN.
        MOV     EOL,1           Set EOL flag.
RTRN    POP     EBX             Restore register EBX.
        POP     EAX             Restore register EAX.
        IRET
```

Figure 4.17 An interrupt-servicing routine to read one line from a keyboard using interrupts on IA-32 processors.

4.4 DIRECT MEMORY ACCESS

The discussion in the previous sections concentrates on data transfer between the processor and I/O devices. Data are transferred by executing instructions such as

Move DATAIN,R0

An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready. To do this, the processor either polls a status flag in the device interface or waits for the device to send an interrupt request. In either case, considerable overhead is incurred, because several program instructions must be executed for each data word transferred. In addition to polling the status register of the device, instructions are needed for incrementing the memory address and keeping track of the word count. When interrupts are used, there is the additional overhead associated with saving and restoring the program counter and other state information.

To transfer large blocks of data at high speed, an alternative approach is used. A special control unit may be provided to allow transfer of a block of data directly

between an external device and the main memory, without continuous intervention by the processor. This approach is called *direct memory access,* or DMA.

DMA transfers are performed by a control circuit that is part of the I/O device interface. We refer to this circuit as a *DMA controller*. The DMA controller performs the functions that would normally be carried out by the processor when accessing the main memory. For each word transferred, it provides the memory address and all the bus signals that control data transfer. Since it has to transfer blocks of data, the DMA controller must increment the memory address for successive words and keep track of the number of transfers.

Although a DMA controller can transfer data without intervention by the processor, its operation must be under the control of a program executed by the processor. To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block, and the direction of the transfer. On receiving this information, the DMA controller proceeds to perform the requested operation. When the entire block has been transferred, the controller informs the processor by raising an interrupt signal.

While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer.

I/O operations are always performed by the operating system of the computer in response to a request from an application program. The OS is also responsible for suspending the execution of one program and starting another. Thus, for an I/O operation involving DMA, the OS puts the program that requested the transfer in the Blocked state (see Section 4.2.6), initiates the DMA operation, and starts the execution of another program. When the transfer is completed, the DMA controller informs the processor by sending an interrupt request. In response, the OS puts the suspended program in the Runnable state so that it can be selected by the scheduler to continue execution.

Figure 4.18 shows an example of the DMA controller registers that are accessed by the processor to initiate transfer operations. Two registers are used for storing the

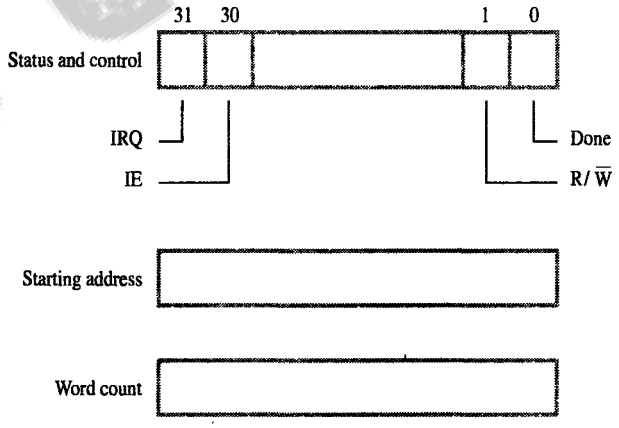


Figure 4.18 Registers in a DMA interface.

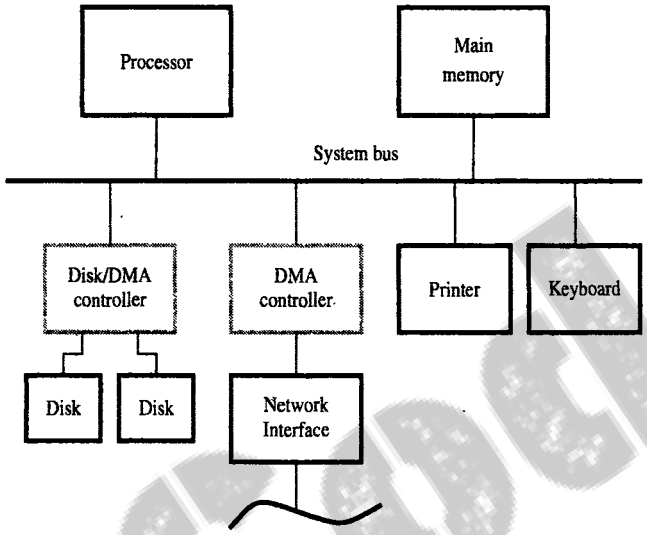


Figure 4.19 Use of DMA controllers in a computer system.

starting address and the word count. The third register contains status and control flags. The R/$\overline{W}$ bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a write operation. When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1. Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt.

An example of a computer system is given in Figure 4.19, showing how DMA controllers may be used. A DMA controller connects a high-speed network to the computer bus. The disk controller, which controls two disks, also has DMA capability and provides two DMA channels. It can perform two independent DMA operations, as if each disk had its own DMA controller. The registers needed to store the memory address, the word count, and so on are duplicated, so that one set can be used with each device.

To start a DMA transfer of a block of data from the main memory to one of the disks, a program writes the address and word count information into the registers of the corresponding channel of the disk controller. It also provides the disk controller with information to identify the data for future retrieval. The DMA controller proceeds independently to implement the specified operation. When the DMA transfer is completed, this fact is recorded in the status and control register of the DMA channel by setting the Done bit. At the same time, if the IE bit is set, the controller sends an interrupt request to the processor and sets the IRQ bit. The status register can also be used to record other information, such as whether the transfer took place correctly or errors occurred.

Memory accesses by the processor and the DMA controllers are interwoven. Requests by DMA devices for using the bus are always given higher priority than processor requests. Among different DMA devices, top priority is given to high-speed peripherals such as a disk, a high-speed network interface, or a graphics display device. Since the processor originates most memory access cycles, the DMA controller can be said to "steal" memory cycles from the processor. Hence, this interweaving technique is usually called *cycle stealing*. Alternatively, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as *block* or *burst* mode.

Most DMA controllers incorporate a data storage buffer. In the case of the network interface in Figure 4.19, for example, the DMA controller reads a block of data from the main memory and stores it into its input buffer. This transfer takes place using burst mode at a speed appropriate to the memory and the computer bus. Then, the data in the buffer are transmitted over the network at the speed of the network.

A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main memory. To resolve these conflicts, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.

4.4.1 BUS ARBITRATION

The device that is allowed to initiate data transfers on the bus at any given time is called the *bus master*. When the current master relinquishes control of the bus, another device can acquire this status. Bus arbitration is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. The selection of the bus master must take into account the needs of various devices by establishing a priority system for gaining access to the bus.

There are two approaches to bus arbitration: centralized and distributed. In centralized arbitration, a single *bus arbiter* performs the required arbitration. In distributed arbitration, all devices participate in the selection of the next bus master.

Centralized Arbitration

The bus arbiter may be the processor or a separate unit connected to the bus. Figure 4.20 illustrates a basic arrangement in which the processor contains the bus arbitration circuitry. In this case, the processor is normally the bus master unless it grants bus mastership to one of the DMA controllers. A DMA controller indicates that it needs to become the bus master by activating the Bus-Request line, $\overline{BR}$. This is an open-drain line for the same reasons that the Interrupt-Request line in Figure 4.6 is an open-drain line. The signal on the Bus-Request line is the logical OR of the bus requests from all the devices connected to it. When Bus-Request is activated, the processor activates the Bus-Grant signal, BG1, indicating to the DMA controllers that they may use the bus when it becomes free. This signal is connected to all DMA controllers using a daisy-chain arrangement. Thus, if DMA controller 1 is requesting the bus, it blocks the propagation of the grant signal to other devices. Otherwise, it passes the grant downstream by asserting BG2. The current bus master indicates to all

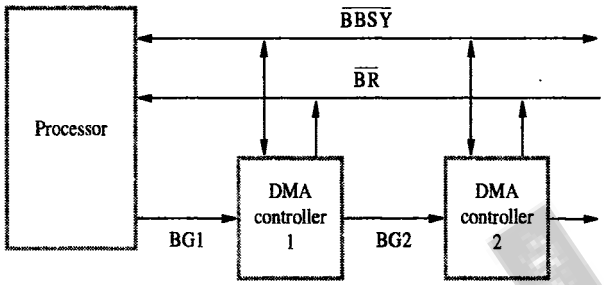


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.

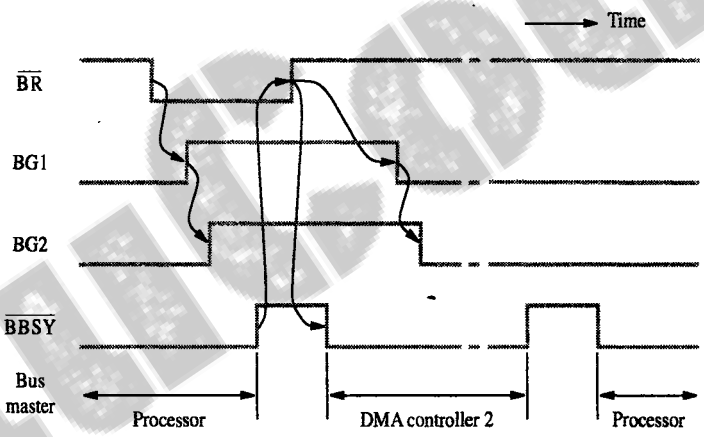


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

devices that it is using the bus by activating another open-collector line called Bus-Busy, BBSY. Hence, after receiving the Bus-Grant signal, a DMA controller waits for Bus-Busy to become inactive, then assumes mastership of the bus. At this time, it activates Bus-Busy to prevent other devices from using the bus at the same time.

The timing diagram in Figure 4.21 shows the sequence of events for the devices in Figure 4.20 as DMA controller 2 requests and acquires bus mastership and later releases the bus. During its tenure as the bus master, it may perform one or more data transfer operations, depending on whether it is operating in the cycle stealing or block mode. After it releases the bus, the processor resumes bus mastership. This figure shows the causal relationships among the signals involved in the arbitration process. Details of timing, which vary significantly from one computer bus to another, are not shown.

Figure 4.20 shows one bus-request line and one bus-grant line forming a daisy chain. Several such pairs may be provided, in an arrangement similar to that used

for multiple interrupt requests in Figure 4.8*b*. This arrangement leads to considerable flexibility in determining the order in which requests from different devices are serviced. The arbiter circuit ensures that only one request is granted at any given time, according to a predefined priority scheme. For example, if there are four bus request lines, BR1 through BR4, a fixed priority scheme may be used in which BR1 is given top priority and BR4 is given lowest priority. Alternatively, a rotating priority scheme may be used to give all devices an equal chance of being serviced. Rotating priority means that after a request on line BR1 is granted, the priority order becomes 2, 3, 4, 1.

Distributed Arbitration

Distributed arbitration means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process, without using a central arbiter. A simple method for distributed arbitration is illustrated in Figure 4.22. Each device on the bus is assigned a 4-bit identification number. When one or more devices request the bus, they assert the Start-Arbitration signal and place their 4-bit ID numbers on four open-collector lines, ARB0 through ARB3. A winner is selected as a result of the interaction among the signals transmitted over these lines by all contenders. The net outcome is that the code on the four lines represents the request that has the highest ID number.

The drivers are of the open-collector type. Hence, if the input to one driver is equal to one and the input to another driver connected to the same bus line is equal to 0 the

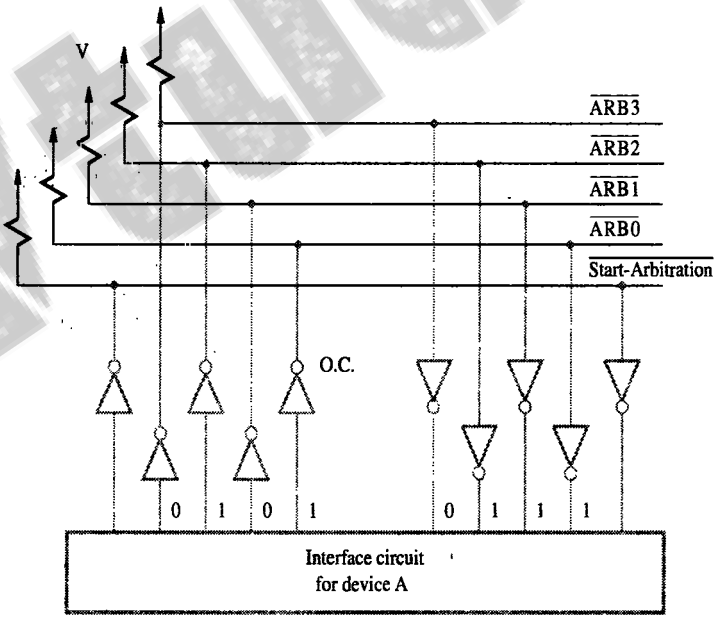


Figure 4.22 A distributed arbitration scheme.

bus will be in the low-voltage state. In other words, the connection performs an OR function in which logic 1 wins.

Assume that two devices, A and B, having ID numbers 5 and 6, respectively, are requesting the use of the bus. Device A transmits the pattern 0101, and device B transmits the pattern 0110. The code seen by both devices is 0111. Each device compares the pattern on the arbitration lines to its own ID, starting from the most significant bit. If it detects a difference at any bit position, it disables its drivers at that bit position and for all lower-order bits. It does so by placing a 0 at the input of these drivers. In the case of our example, device A detects a difference on line $\overline{ARB1}$. Hence, it disables its drivers on lines $\overline{ARB1}$ and $\overline{ARB0}$. This causes the pattern on the arbitration lines to change to 0110, which means that B has won the contention. Note that, since the code on the priority lines is 0111 for a short period, device B may temporarily disable its driver on line $\overline{ARB0}$. However, it will enable this driver again once it sees a 0 on line $\overline{ARB1}$ resulting from the action by device A.

Decentralized arbitration has the advantage of offering higher reliability, because operation of the bus is not dependent on any single device. Many schemes have been proposed and used in practice to implement distributed arbitration. The SCSI bus described in Section 4.7.2 provides another example.

4.5 BUSES

The processor, main memory, and I/O devices can be interconnected by means of a common bus whose primary function is to provide a communications path for the transfer of data. The bus includes the lines needed to support interrupts and arbitration. In this section, we discuss the main features of the bus protocols used for transferring data. A bus protocol is the set of rules that govern the behavior of various devices connected to the bus as to when to place information on the bus, assert control signals, and so on. After describing bus protocols, we will present examples of interface circuits that use these protocols.

The bus lines used for transferring data may be grouped into three types: data, address, and control lines. The control signals specify whether a read or a write operation is to be performed. Usually, a single a R/$\overline{W}$ line is used. It specifies Read when set to 1 and Write when set to 0. When several operand sizes are possible, such as byte, word, or long word, the required size of data is indicated.

The bus control signals also carry timing information. They specify the times at which the processor and the I/O devices may place data on the bus or receive data from the bus. A variety of schemes have been devised for the timing of data transfers over a bus. These can be broadly classified as either synchronous or asynchronous schemes.

Recall from Section 4.4.1 that in any data transfer operation, one device plays the role of a *master*. This is the device that initiates data transfers by issuing read or write commands on the bus; hence, it may be called an *initiator*. Normally, the processor acts as the master, but other devices with DMA capability may also become bus masters. The device addressed by the master is referred to as a *slave* or *target*.

disadvantage is that the flash memory will deteriorate after it has been written a number of times. Fortunately, this number is high, typically at least one million times.

5.4 SPEED, SIZE, AND COST

We have already stated that an ideal memory would be fast, large, and inexpensive. From the discussion in Section 5.2, it is clear that a very fast memory can be implemented if SRAM chips are used. But these chips are expensive because their basic cells have six transistors, which precludes packing a very large number of cells onto a single chip. Thus, for cost reasons, it is impractical to build a large memory using SRAM chips. The alternative is to use Dynamic RAM chips, which have much simpler basic cells and thus are much less expensive. But such memories are significantly slower.

Although dynamic memory units in the range of hundreds of megabytes can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data. A solution is provided by using secondary storage, mainly magnetic disks, to implement large memory spaces. Very large disks are available at a reasonable price, and they are used extensively in computer systems. However, they are much slower than the semiconductor memory units. So we conclude the following: A huge amount of cost-effective storage can be provided by magnetic disks. A large, yet affordable, main memory can be built with dynamic RAM technology. This leaves SRAMs to be used in smaller units where speed is of the essence, such as in cache memories.

All of these different types of memory units are employed effectively in a computer. The entire computer memory can be viewed as the hierarchy depicted in Figure 5.13. The fastest access is to data held in processor registers. Therefore, if we consider the registers to be part of the memory hierarchy, then the processor registers are at the top in terms of the speed of access. Of course, the registers provide only a minuscule portion of the required memory.

At the next level of the hierarchy is a relatively small amount of memory that can be implemented directly on the processor chip. This memory, called a *processor cache*, holds copies of instructions and data stored in a much larger memory that is provided externally. The cache memory concept was introduced in Figure 1.6 and is examined in detail in Section 5.5. There are often two levels of caches. A primary cache is always located on the processor chip. This cache is small because it competes for space on the processor chip, which must implement many other functions. The primary cache is referred to as *level 1* (L1) cache. A larger, secondary cache is placed between the primary cache and the rest of the memory. It is referred to as *level 2* (L2) cache. It is usually implemented using SRAM chips.

Including a primary cache on the processor chip and using a larger, off-chip, secondary cache is currently the most common way of designing computers. However, other arrangements can be found in practice. It is possible not to have a cache on the processor chip at all. Also, it is possible to have both L1 and L2 caches on the processor chip.

The next level in the hierarchy is called the *main memory*. This rather large memory is implemented using dynamic memory components, typically in the form of SIMMs,

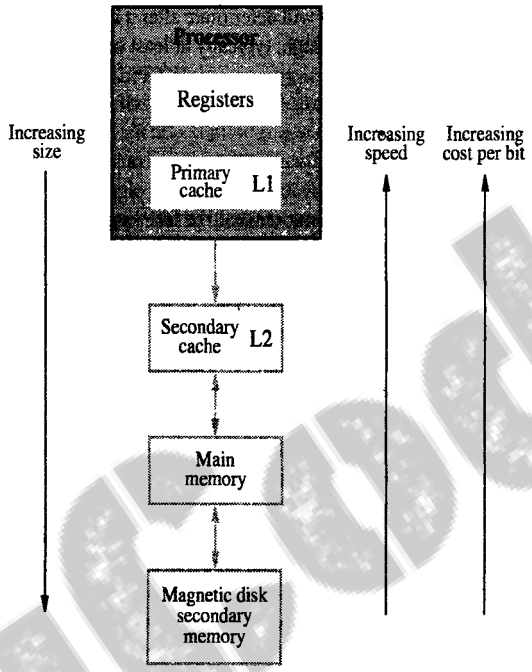


Figure 5.13 Memory hierarchy.

DIMMs, or RIMMs. The main memory is much larger but significantly slower than the cache memory. In a typical computer, the access time for the main memory is about ten times longer than the access time for the L1 cache.

Disk devices provide a huge amount of inexpensive storage. They are very slow compared to the semiconductor devices used to implement the main memory. We will discuss disk technology in Section 5.9.

During program execution, the speed of memory access is of utmost importance. The key to managing the operation of the hierarchical memory system in Figure 5.13 is to bring the instructions and data that will be used in the near future as close to the processor as possible. This can be done by using the mechanisms presented in the sections that follow. We begin with a detailed discussion of cache memories.

5.5 CACHE MEMORIES

The speed of the main memory is very low in comparison with the speed of modern processors. For good performance, the processor cannot spend much of its time waiting to access instructions and data in main memory. Hence, it is important to devise a scheme

blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function. When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the *replacement algorithm.*

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a *read* or *write hit* is said to have occurred. In a Read operation, the main memory is not involved. For a Write operation, the system can proceed in two ways. In the first technique, called the *write-through* protocol, the cache location and the main memory location are updated simultaneously. The second technique is to update only the cache location and to mark it as updated with an associated flag bit, often called the *dirty* or *modified* bit. The main memory location of the word is updated later, when the block containing this marked word is to be removed from the cache to make room for a new block. This technique is known as the *write-back,* or *copy-back,* protocol. The write-through protocol is simpler, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. Note that the write-back protocol may also result in unnecessary Write operations because when a cache block is written back to the memory all words of the block are written back, even if only a single word has been changed while the block was in the cache.

When the addressed word in a Read operation is not in the cache, a *read miss* occurs. The block of words that contains the requested word is copied from the main memory into the cache. After the entire block is loaded into the cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from the main memory. The latter approach, which is called *load-through,* or *early restart,* reduces the processor's waiting period somewhat, but at the expense of more complex circuitry.

During a Write operation, if the addressed word is not in the cache, a *write miss* occurs. Then, if the write-through protocol is used, the information is written directly into the main memory. In the case of the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.

5.5.1 MAPPING FUNCTIONS

To discuss possible methods for specifying where memory blocks are placed in the cache, we use a specific small example. Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K blocks of 16 words each. For simplicity, we will assume that consecutive addresses refer to consecutive words.

Direct Mapping

The simplest way to determine cache locations in which to store memory blocks is the *direct-mapping* technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure 5.15. Thus, whenever one of the main memory blocks 0, 128, 256, ... is loaded in the cache, it is stored in cache block 0. Blocks 1, 129, 257, ... are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a

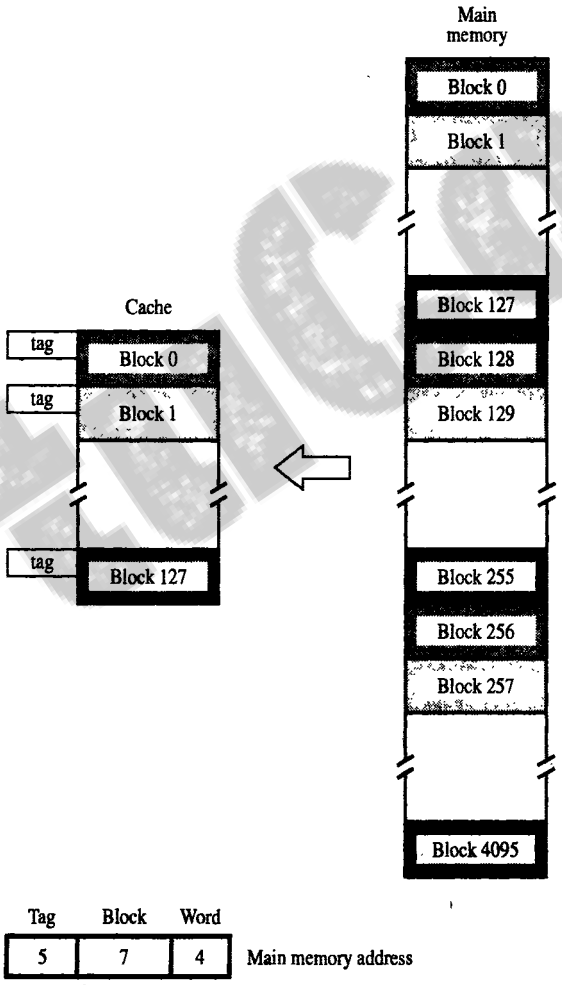


Figure 5.15 Direct-mapped cache.

program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block. In this case, the replacement algorithm is trivial.

Placement of a block in the cache is determined from the memory address. The memory address can be divided into three fields, as shown in Figure 5.15. The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the block are stored in 5 *tag* bits associated with its location in the cache. They identify which of the 32 blocks that are mapped into this cache position are currently resident in the cache. As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache. The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.

Associative Mapping

Figure 5.16 shows a much more flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique. It gives complete freedom in choosing the cache location in which to place the memory block. Thus, the space in the cache can be used more efficiently. A new block that has to be brought into the cache has to replace (eject) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. Many replacement algorithms are possible, as we discuss in Section 5.5.2. The cost of an associative cache is higher than the cost of a direct-mapped cache because of the need to search all 128 tag patterns to determine whether a given block is in the cache. A search of this kind is called an *associative search*. For performance reasons, the tags must be searched in parallel.

Set-Associative Mapping

A combination of the direct- and associative-mapping techniques can be used. Blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this *set-associative-mapping* technique is shown in Figure 5.17 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, ..., 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

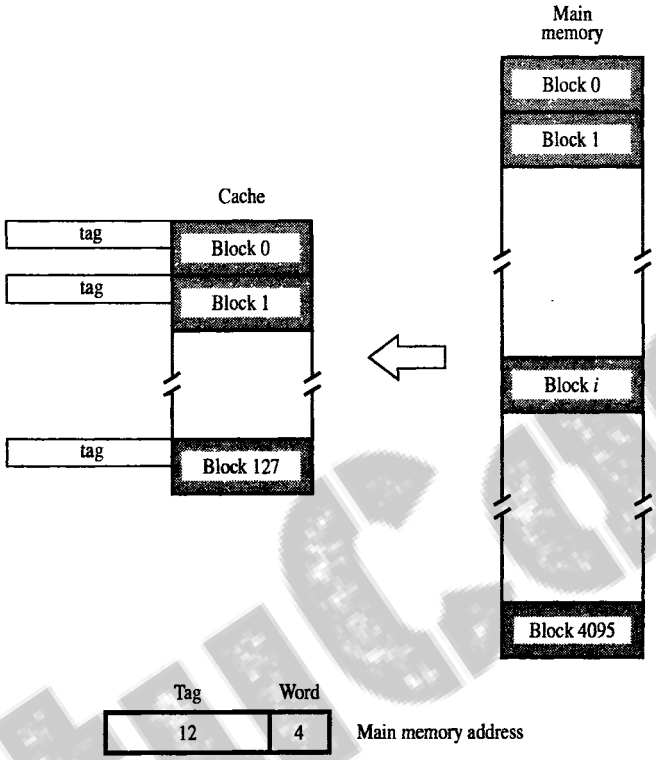


Figure 5.16 Associative-mapped cache.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 5.17, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. A cache that has k blocks per set is referred to as a k-way set-associative cache.

One more control bit, called the *valid bit,* must be provided for each block. This bit indicates whether the block contains valid data. It should not be confused with the modified, or dirty, bit mentioned earlier. The dirty bit, which indicates whether the block has been modified during its cache residency, is needed only in systems that do not use the write-through method. The valid bits are all set to 0 when power is initially applied to the system or when the main memory is loaded with new programs and data from the disk. Transfers from the disk to the main memory are carried out by a DMA mechanism. Normally, they bypass the cache for both cost and performance reasons. The valid bit of a particular cache block is set to 1 the first time this block is loaded

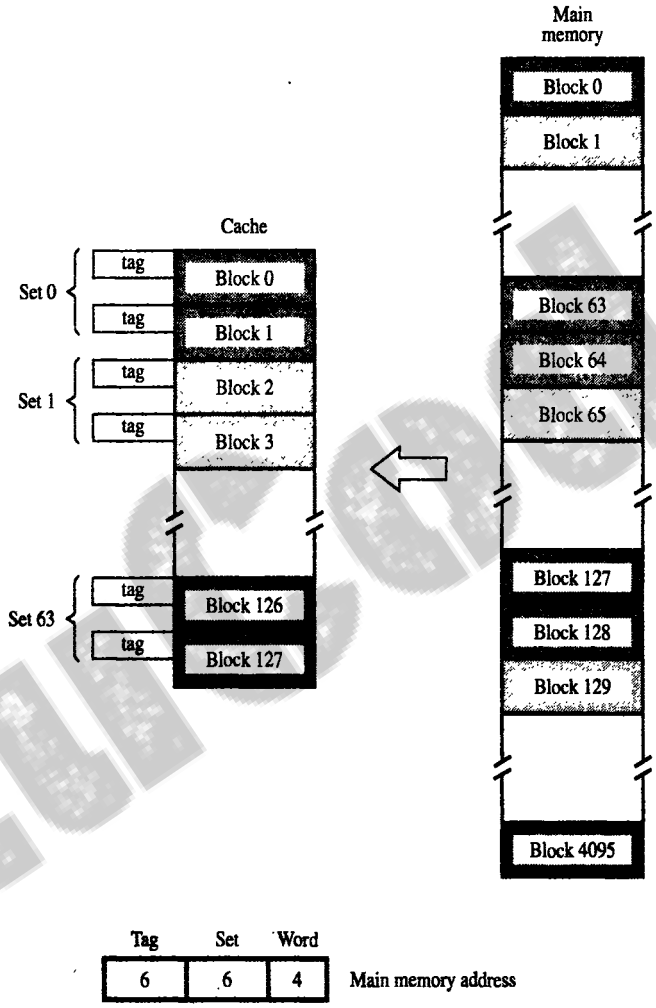


Figure 5.17 Set-associative-mapped cache with two blocks per set.

from the main memory. Whenever a main memory block is updated by a source that bypasses the cache, a check is made to determine whether the block being loaded is currently in the cache. If it is, its valid bit is cleared to 0. This ensures that *stale* data will not exist in the cache.

A similar difficulty arises when a DMA transfer is made from the main memory to the disk, and the cache uses the write-back protocol. In this case, the data in the memory might not reflect the changes that may have been made in the cached copy.

One solution to this problem is to *flush* the cache by forcing the dirty data to be written back to the memory before the DMA transfer takes place. The operating system can do this easily, and it does not affect performance greatly, because such disk transfers do not occur often. This need to ensure that two different entities (the processor and DMA subsystems in this case) use the same copies of data is referred to as a *cache-coherence* problem.

5.5.2 REPLACEMENT ALGORITHMS

In a direct-mapped cache, the position of each block is predetermined; hence, no replacement strategy exists. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is an important issue because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. However, it is not easy to determine which blocks are about to be referenced. The property of locality of reference in programs gives a clue to a reasonable strategy. Because programs usually stay in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique is called the *LRU replacement algorithm*.

To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds. Suppose it is required to track the LRU block of a four-block set in a set-associative cache. A 2-bit counter can be used for each block. When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a *miss* occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one. It can be easily verified that the counter values of occupied blocks are always distinct.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache (see Section 5.5.3 and Problem 5.12). Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

Several other replacement algorithms are also used in practice. An intuitively reasonable rule would be to remove the "oldest" block from a full set when a new block must be brought in. However, because this algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU