CHAPTER

4

INPUT/OUTPUT ORGANIZATION

CHAPTER OBJECTIVES

In this chapter you will learn about:

- How program-controlled I/O is performed using polling
- The idea of interrupts and the hardware and software needed to support them
- Direct memory access as an I/O mechanism for high-speed devices
- Data transfer over synchronous and asynchronous buses
- The design of I/O interface circuits
- Commercial bus standards, in particular the PCI, SCSI, and USB buses

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

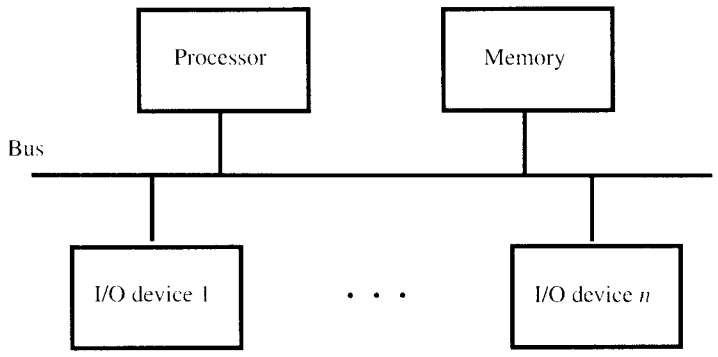


Figure 4.1 A single-bus structure.

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

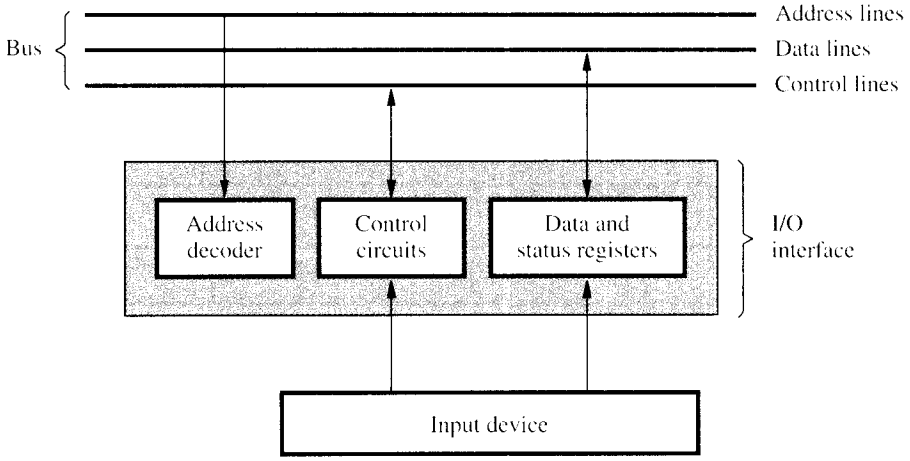


Figure 4.2 I/O interface for an input device.

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

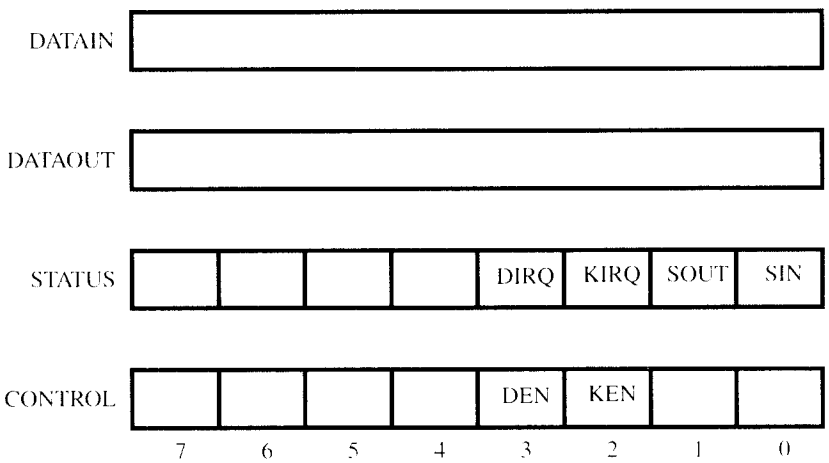


Figure 4.3 Registers in keyboard and display interfaces.

```
        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

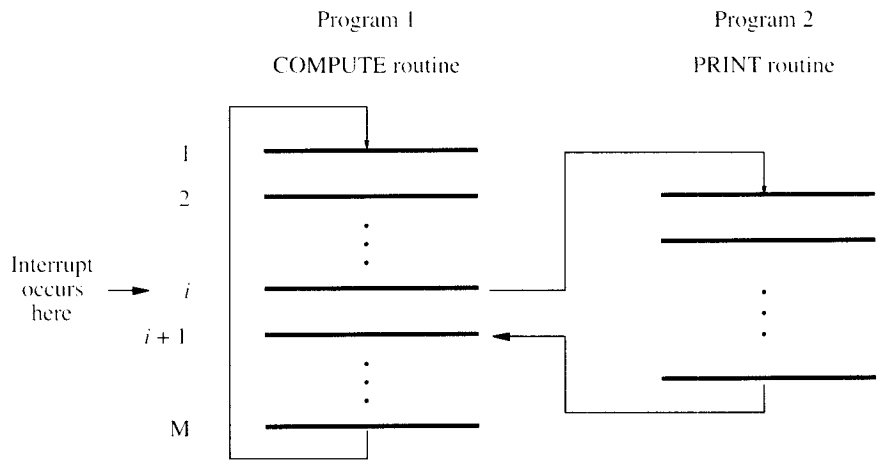


Figure 4.5 Transfer of control through the use of interrupts.

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

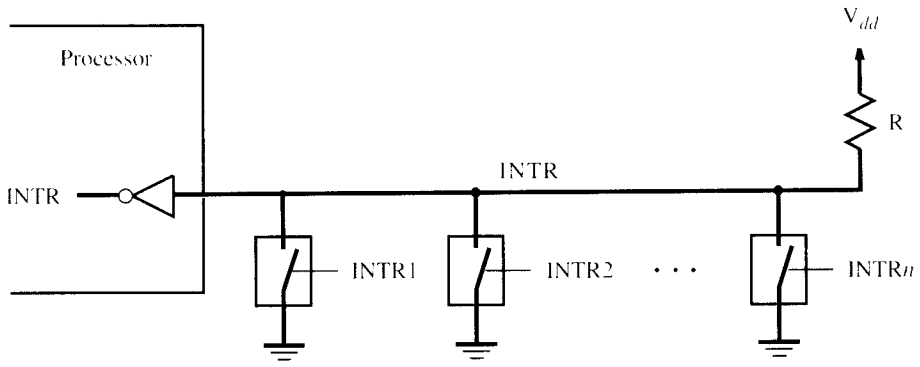


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

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.

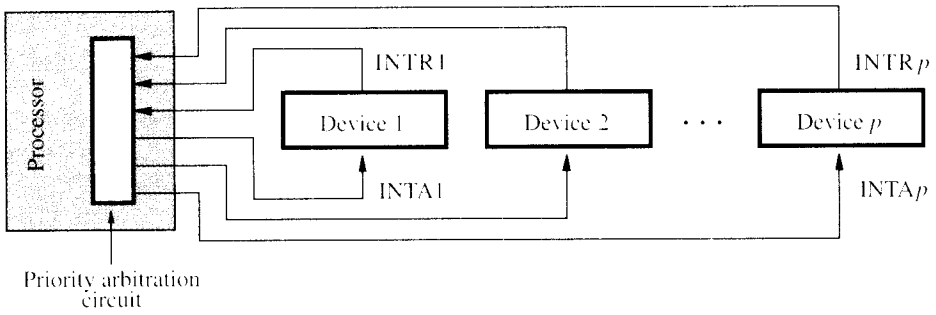
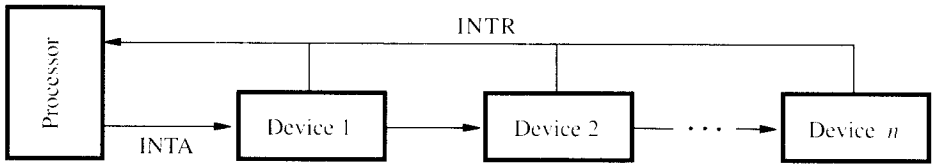


Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

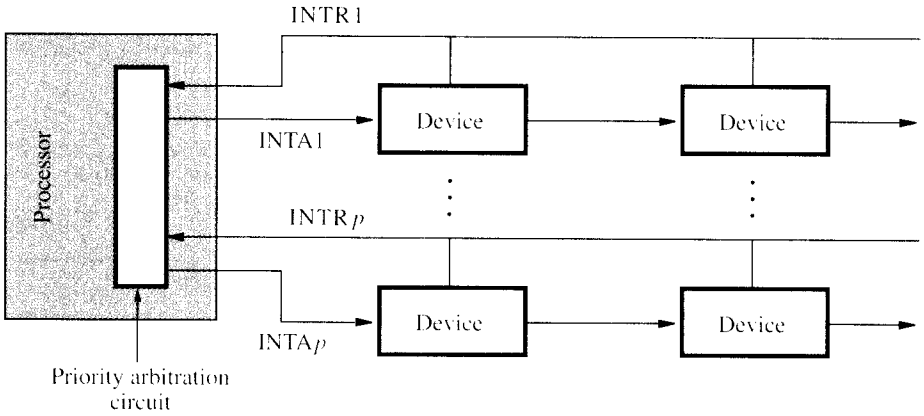
Simultaneous Requests

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

memory, starting at location LINE. To perform this operation using interrupts, we need to initialize the interrupt process. This may be accomplished as follows:

1. Load the starting address of the interrupt-service routine in location INTVEC.
2. Load the address LINE in a memory location PNTR. The interrupt-service routine will use this location as a pointer to store the input characters in the memory.
3. Enable keyboard interrupts by setting bit 2 in register CONTROL to 1.
4. Enable interrupts in the processor by setting to 1 the IE bit in the processor status register, PS.

Once this initialization is completed, typing a character on the keyboard will cause an interrupt request to be generated by the keyboard interface. The program being executed at that time will be interrupted and the interrupt-service routine will be executed. This routine has to perform the following tasks:

1. Read the input character from the keyboard input data register. This will cause the interface circuit to remove its interrupt request.
2. Store the character in the memory location pointed to by PNTR, and increment PNTR.
3. When the end of the line is reached, disable keyboard interrupts and inform program Main.
4. Return from interrupt.

The instructions needed to perform these tasks are shown in Figure 4.9. When the end of the input line is detected, the interrupt-service routine clears the KEN bit in register CONTROL as no further input is expected. It also sets to 1 the variable EOL (End Of Line). This variable is initially set to 0. We assume that it is checked periodically by program Main to determine when the input line is ready for processing.

Input/output operations in a computer system are usually much more involved than this simple example suggests. As we will describe in Section 4.2.5, the operating system of the computer performs these operations on behalf of user programs.

4.2.5 EXCEPTIONS

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin. So far, we have dealt only with interrupts caused by requests received during I/O data transfers. However, the interrupt mechanism is used in a number of other situations.

The term *exception* is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception. We now describe a few other kinds of exceptions.

Recovery from Errors

Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in

Main Program

```
        Move        #LINE.PNTR      Initialize buffer pointer.
        Clear       EOL             Clear end-of-line indicator.
        BitSet      #2.CONTROL      Enable keyboard interrupts.
        BitSet      #9.PS           Set interrupt-enable bit in the PS.
        .
        .
        .
```

Interrupt-service routine

```
READ    MoveMultiple  R0 R1,-(SP)   Save registers R0 and R1 on stack.
        Move          PNTR.R0       Load address pointer.
        MoveByte      DATAIN.R1     Get input character and
        MoveByte      R1,(R0)+        store it in memory.
        Move          R0.PNTR       Update pointer.
        CompareByte   #$0D.R1       Check if Carriage Return.
        Branch≠0      RTRN
        Move          #1.EOL        Indicate end of line.
        BitClear      #2.CONTROL    Disable keyboard interrupts.
RTRN    MoveMultiple  (SP)+,R0 R1   Restore registers R0 and R1.
        Return-from-interrupt
```

Figure 4.9 Using interrupts to read a line of characters from a keyboard via the registers in Figure 4.3.

the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt.

The processor may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero.

When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine. This routine takes appropriate action to recover from the error, if possible, or to inform the user about it. Recall that in the case of an I/O interrupt, the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error, execution of the interrupted instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging

Another important type of exception is used as an aid in debugging programs. System software usually includes a program called a *debugger*, which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities called trace and breakpoints.

When a processor is operating in the *trace* mode, an exception occurs after execution of every instruction, using the debugging program as the exception-service routine. The debugging program enables the user to examine the contents of registers, memory locations, and so on. On return from the debugging program, the next instruction in the program being debugged is executed, then the debugging program is activated again. The trace exception is disabled during the execution of the debugging program.

Breakpoints provide a similar facility, except that the program being debugged is interrupted only at specific points selected by the user. An instruction called Trap or Software-interrupt is usually provided for this purpose. Execution of this instruction results in exactly the same actions as when a hardware interrupt request is received. While debugging a program, the user may wish to interrupt program execution after instruction i. The debugging routine saves instruction $i + 1$ and replaces it with a software interrupt instruction. When the program is executed and reaches that point, it is interrupted and the debugging routine is activated. This gives the user a chance to examine memory and register contents. When the user is ready to continue executing the program being debugged, the debugging routine restores the saved instruction that was at location $i + 1$ and executes a Return-from-interrupt instruction.

Privilege Exception

To protect the operating system of a computer from being corrupted by user programs, certain instructions can be executed only while the processor is in the supervisor mode. These are called *privileged instructions*. For example, when the processor is running in the user mode, it will not execute an instruction that changes the priority level of the processor or that enables a user program to access areas in the computer memory that have been allocated to other users. An attempt to execute such an instruction will produce a privilege exception, causing the processor to switch to the supervisor mode and begin executing an appropriate routine in the operating system.

4.2.6 USE OF INTERRUPTS IN OPERATING SYSTEMS

The operating system (OS) is responsible for coordinating all activities within a computer. It makes extensive use of interrupts to perform I/O operations and communicate with and control the execution of user programs. The interrupt mechanism enables the operating system to assign priorities, switch from one user program to another, implement security and protection features, and coordinate I/O activities. We will discuss some of these aspects briefly. A discussion of operating systems is outside the scope of this book. Our objective here is to illustrate how interrupts are used.

The operating system incorporates the interrupt-service routines for all devices connected to a computer. Application programs do not perform I/O operations themselves. When an application program needs an input or an output operation, it points to the data to be transferred and asks the OS to perform the operation. The OS suspends the execution of that program temporarily and performs the requested I/O operation. When the operation is completed, the OS transfers control back to the application program. The OS and the application program pass control back and forth using software interrupts.

An operating system provides a variety of services to application programs. To facilitate the implementation of these services, most processors have several different software interrupt instructions, each with its own interrupt vector. They can be used to call different parts of the OS, depending on the service being requested. Alternatively, a processor may have only one software interrupt instruction, with an immediate operand that can be used to specify the desired service.

In a computer that has both a supervisor and a user mode, the processor switches its operation to supervisor mode at the time it accepts an interrupt request. It does so by setting a bit in the processor status register after saving the old contents of that register on the stack. Thus, when an application program calls the OS by a software interrupt instruction, the processor automatically switches to supervisor mode, giving the OS complete access to the computer's resources. When the OS executes a Return-from-interrupt instruction, the processor status word belonging to the application program is restored from the stack. As a result, the processor switches back to the user mode.

To illustrate the interaction between application programs and the operating system, let us consider an example that involves multitasking. *Multitasking* is a mode of operation in which a processor executes several user programs at the same time. A common OS technique that makes this possible is called *time slicing*. With this technique, each program runs for a short period called a time slice, τ, then another program runs for its time slice, and so on. The period τ is determined by a continuously running hardware clock, which generates an interrupt every τ seconds.

Figure 4.10 describes the routines needed to implement some of the essential functions in a multitasking environment. At the time the operating system is started, an initialization routine, called OSINIT in the figure, is executed. Among other things, this routine loads the appropriate values in the interrupt vector locations in the memory. These values are the starting addresses of the interrupt service routines for the corresponding interrupts. For example, OSINIT loads the starting address of a routine called SCHEDULER in the interrupt vector corresponding to the timer interrupt. Hence, at the end of each time slice, the timer interrupt causes this routine to be executed.

A program, together with any information that describes its current state of execution, is regarded by the OS as an entity called a *process*. A process can be in one of three states: Running, Runnable, or Blocked. The Running state means that the program is currently being executed. The process is Runnable if the program is ready for execution but is waiting to be selected by the scheduler. The third state, Blocked, means that the program is not ready to resume execution for some reason. For example, it may be waiting for completion of an I/O operation that it requested earlier.

Assume that program A is in the Running state during a given time slice. At the end of that time slice, the timer interrupts the execution of this program and starts the execution of SCHEDULER. This is an operating system routine whose function is to determine which user program should run in the next time slice. It starts by saving all the information that will be needed later when execution of program A is resumed. The information saved, which is called the *program state*, includes register contents, the program counter, and the processor status word. Registers must be saved because they may contain intermediate results for any computation in progress at the time of interruption. The program counter points to the location where execution is to resume

OSINIT	Set interrupt vectors: Time-slice clock — SCHEDULER Software interrupt — OSSERVICES Keyboard interrupts — IOData
	⋮
OSSERVICES	Examine stack to determine requested operation. Call appropriate routine.
SCHEDULER	Save program state. Select a runnable process. Restore saved context of new process. Push new values for PS and PC on stack. Return from interrupt.

(a) OS initialization, services, and scheduler

IOINIT	Set process status to Blocked. Initialize memory buffer address pointer and counter. Call device driver to initialize device and enable interrupts in the device interface. Return from subroutine.
IODATA	Poll devices to determine source of interrupt. Call appropriate driver. If END = 1, then set process status to Runnable. Return from interrupt.

(b) I/O routines

KBDINIT	Enable interrupts. Return from subroutine.
KBDDATA	Check device status. If ready, then transfer character. If character = CR, then {set END = 1; Disable interrupts} else set END = 0. Return from subroutine.

(c) Keyboard driver

Figure 4.10 A few operating system routines.

later. The processor status word is needed because it contains the condition code flags and other information such as priority level.

Then, SCHEDULER selects for execution some other program, B, that was suspended earlier and is in the Runnable state. It restores all information saved at the time program B was suspended, including the contents of PS and PC, and executes a Return-from-interrupt instruction. As a result, program B resumes execution for τ seconds, at the end of which the timer clock raises an interrupt again, and a *context switch* to another runnable process takes place.

Suppose that program A needs to read an input line from the keyboard. Instead of performing the operation itself, it requests I/O service from the operating system. It uses the stack or the processor registers to pass information to the OS describing the required operation, the I/O device, and the address of a buffer in the program data area where the line should be placed. Then it executes a software interrupt instruction. The interrupt vector for this instruction points to the OSSERVICES routine in Figure 4.10*a*. This routine examines the information on the stack and initiates the requested operation by calling an appropriate OS routine. In our example, it calls IOINIT in Figure 4.10*b*, which is a routine responsible for starting I/O operations.

While an I/O operation is in progress, the program that requested it cannot continue execution. Hence, the IOINIT routine sets the process associated with program A into the Blocked state, indicating to the scheduler that the program cannot resume execution at this time. The IOINIT routine carries out any preparations needed for the I/O operation, such as initializing address pointers and byte count, then calls a routine that performs the I/O transfers.

It is common practice in operating system design to encapsulate all software pertaining to a particular device into a self-contained module called the *device driver*. Such a module can be easily added to or deleted from the OS. We have assumed that the device driver for the keyboard consists of two routines, KBDINIT and KBDDATA, as shown in Figure 4.10*c*. The IOINIT routine calls KBDINIT, which performs any initialization operations needed by the device or its interface circuit. KBDINIT also enables interrupts in the interface circuit by setting the appropriate bit in its control register, and then it returns to IOINIT, which returns to OSSERVICES. The keyboard is now ready to participate in a data transfer operation. It will generate an interrupt request whenever a key is pressed.

Following the return to OSSERVICES, the SCHEDULER routine selects another user program to run. Of course, the scheduler will not select program A, because that program is now in the Blocked state. The Return-from-interrupt instruction that causes the selected user program to begin execution will also enable interrupts in the processor by loading new contents into the processor status register. Thus, an interrupt request generated by the keyboard's interface will be accepted. The interrupt vector for this interrupt points to an OS routine called IODATA. Because there could be several devices connected to the same interrupt request line, IODATA begins by polling these devices to determine the one requesting service. Then, it calls the appropriate device driver to service the request. In our example, the driver called will be KBDDATA, which will transfer one character of data. If the character is a Carriage Return, it will also set to 1 a flag called END, to inform IODATA that the requested I/O operation has been completed. At this point, the IODATA routine changes the state of process A from Blocked to Runnable, so that the scheduler may select it for execution in some future time slice.

4.3 PROCESSOR EXAMPLES

We have discussed the organization of interrupts in general in the previous section. Commercial processors provide many of the features and control mechanisms described, but not necessarily all of them. For example, vectored interrupts may be supported to enable the processor to branch quickly to the interrupt-service routine for a particular device. Alternatively, the task of identifying the device and determining the starting address of the appropriate interrupt-service routine may be left for implementation in software using polling. In the following sections we describe the interrupt-handling mechanisms of the three processors described in Chapter 3.

4.3.1 ARM INTERRUPT STRUCTURE

The ARM processor has a simple yet powerful exception-handling mechanism. There are five sources for exceptions, only two of which are external interrupt-request lines, IRQ and FIQ (Fast Interrupt Request). There is one software interrupt instruction, SWI, and two exceptions that may be caused by abnormal conditions encountered during program execution. These exceptions are an external abort following a bus error and an attempt to execute an undefined instruction. Exceptions are handled according to the following priority structure:

1. Reset (highest priority)
2. Data abort
3. FIQ
4. IRQ
5. Prefetch abort
6. Undefined instruction (lowest priority)

The Reset condition is included in this structure because it must override all other conditions to bring the processor to a known starting condition. Also note that there are two abort conditions. *Data Abort* arises from an error in reading or writing data, and *Prefetch Abort* arises from an error when prefetching instructions from the memory.

Figure 3.1 shows the status register of the ARM processor, CPSR (Current Program Status Register). The low-order byte of this register is shown in Figure 4.11. There are two interrupt mask bits, one each for IRQ and FIQ. When either of these bits is equal to 1, the corresponding interrupt is disabled. The register also contains five mode bits,

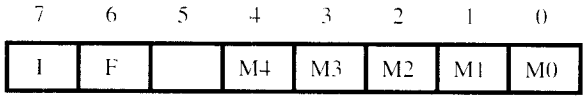


Figure 4.11 Low-order byte of the ARM processor status register.

M_{4-0}, which indicate the mode in which the processor is running. There are six modes — a User mode and five privileged modes, one for each of the five types of exception.

When the processor switches to a different mode, it also switches some of the registers accessible to the program. The register set that is accessible in each mode is shown in Figure 4.12. Registers R0 to R7, R15 (the PC), and CPSR are accessible in all modes. In all privileged modes, except FIQ, registers R8 to R12 are also accessible.

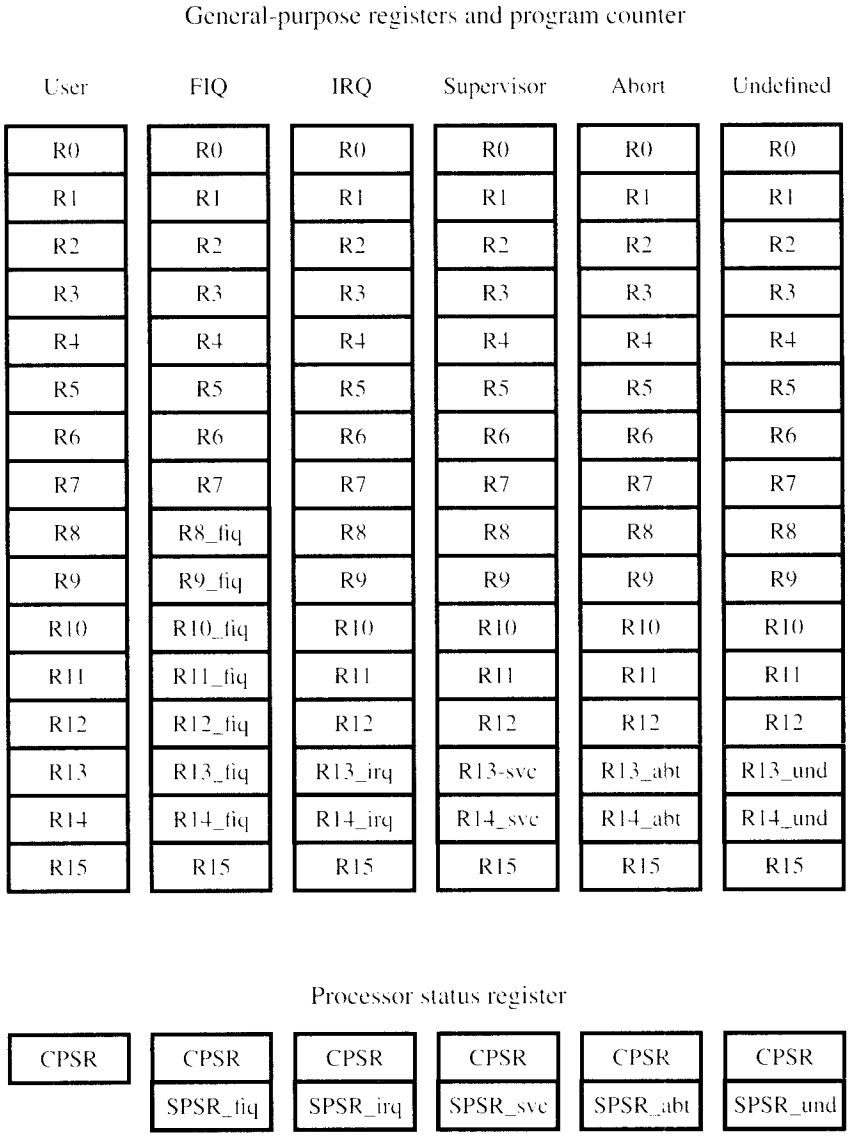


Figure 4.12 Accessible registers in different modes of the ARM processor.

Table 4.1 Interrupt vector addresses for ARM processor

Address (hex)	Exception	Mode entered
0	Reset	Supervisor
4	Undefined instruction	Undefined
8	Software interrupt	Supervisor
C	Abort during prefetch	Abort
10	Abort during data	Abort
14	Reserved	
18	IRQ	IRQ
1C	FIQ	FIQ

However, two new registers replace R13 and R14 in each of the modes: IRQ, Supervisor, Abort, and Undefined. In the case of FIQ, registers R8 to R14 are replaced by R8_fiq to R14_fiq. The registers that replace user mode registers are called *banked registers*. They can be used by interrupt-service routines without the need to save the contents of their User mode counterparts. For example, when an instruction refers to R13 while the processor is in the IRQ mode, the register accessed is R13_irq rather than User mode register R13. Also, each mode other than the User mode has a dedicated register called Saved Processor Status Register (SPSR_svc, SPRS_irq, etc.) for saving the contents of CPSR at the time the interruption occurs.

Exception-handling routines start at fixed locations in memory, as shown in Table 4.1. Following an interrupt, the processor enters the mode indicated and begins execution at the corresponding vector address. Since there is only space for one instruction at all but the last address (FIQ), these locations should contain branch instructions to the service routines. In the case of FIQ, the service routine does not need to use a branch instruction and may continue from the starting location shown.

When the processor accepts an interrupt, it takes the following actions:

1. It saves the return address of the interrupted program in register 14 of the corresponding mode. For example, in the case of FIQ, it saves the return address in R14_fiq. The exact value saved depends on the type of exception, as will be explained shortly.
2. It saves the contents of the processor status register, CPSR, in the corresponding SPSR.
3. It changes the mode bits in CPSR according to the type of interrupt, as shown in the last column of Table 4.1. For FIQ and IRQ, it also sets the corresponding mask bit in CPSR to 1, thus disabling further interrupts on the same line.
4. It branches to the interrupt-service routine starting at the appropriate vector address.

The ARM processor uses a pipelined structure. As we will explain in Chapter 8, this means that an instruction is fetched before the execution of the preceding instruction is completed. Suppose that the processor fetches instruction I_1 at some address A. The processor increments the contents of the PC to A+4 and begins executing instruction I_1. Before completing execution of that instruction, it fetches instruction I_2 at address A+4, then it increments the PC to A+8. Let us now assume that at the end of execution of instruction I_1 the processor detects that an IRQ interrupt has been received, and it begins to perform the actions described above. It copies the contents of the PC, which are now equal to A+8, into register R14_irq. Instruction I_2, which has been fetched but not yet executed, is discarded. This is the instruction to which the interrupt-service routine must return.

In the scenario described above, the address saved in R14_irq is A+8, but the return address of the interrupt-service routine must be A+4. This means that the interrupt-service routine must subtract 4 from R14_irq before using its contents as the return address. That is, the return instruction must load the value [R14_irq] − 4 into the PC. It must also copy the contents of SPSR_irq into CPSR. The latter action restores the processor to its operating mode before the interruption occurred, and it clears the interrupt mask so that interrupts are once again enabled. The required actions are carried out by the instruction:

```
SUBS    PC,R14_irq, #4
```

This instruction subtracts 4 from R14_irq and stores the result into R15. The suffix S normally means set condition codes. When the target register of the instruction is the PC, the S suffix causes the processor to copy the contents of SPSR_irq into CPSR, thus completing the actions required to return to the interrupted program.

The amount that needs to be subtracted from R14 to obtain the correct return address depends on the details of instruction execution in the processor pipeline. Hence, it differs from one type of exception to another. For example, in the case of a software interrupt triggered by the SWI instruction, the value saved in R14_svc is the correct return address. Hence, return from an SWI service routine could be accomplished using the instruction

```
MOVS    PC,R14_svc
```

Table 4.2 gives the correct value for the return address and the instruction that can be used to return to the interrupted program for each of the exceptions in Table 4.1. Note that for an abort interrupt, which may be caused by a bus error, the desired return address is shown as the address of the instruction that caused the error. It is assumed that the controlling software may wish to retry this instruction.

When running in a privileged mode, two special MOV instructions called MSR and MRS transfer data to or from either the current or the saved PSR. For example,

```
MRS     R0,CPSR
```

copies the contents of CPSR into R0. Similarly,

```
MSR     SPSR,R0
```

Table 4.2 Address correction during return from exception

Exception	Saved address*	Desired return address	Return instruction
Undefined instruction	PC+4	PC+4	MOVS PC,R14_und
Software interrupt	PC+4	PC+4	MOVS PC,R14_svc
Prefetch Abort	PC+4	PC	SUBS PC,R14_abt,#4
Data Abort	PC+8	PC	SUBS PC,R14_abt,#8
IRQ	PC+4	PC	SUBS PC,R14_irq,#4
FIQ	PC+4	PC	SUBS PC,R14_fiq,#4

*PC is the address of the instruction that caused the exception. For IRQ and FIQ, it is the address of the first instruction not executed because of the interrupt.

loads SPSR from register R0. These instructions are useful when the operating system needs to enable or disable interrupts, as we will see in Example 4.4.

Stacks and Nesting

The ARM interrupt mechanism stores the return address in a register and does not automatically implement a stacking mechanism to allow subroutine or interrupt nesting. The facilities provided have been carefully thought out to allow the programmer to implement such features when needed and to avoid unnecessary overhead when they are not needed. First, let us observe that nesting is possible when it is caused by different sources. For example, the IRQ interrupt-service routine, whose return address is in R14_irq, may be interrupted by the higher priority FIQ interrupt. The new return address will be stored in R14_fiq.

To allow nesting of interrupts from the same source, the contents of the corresponding R14 and SPSR must be saved on a stack. This can be readily done by program instructions using R13 as the stack pointer. For this reason, dedicated registers R13 and R14 are available in every mode. The interrupt-service routine can save R14 and SPSR on its private stack, then clear the interrupt mask in CPSR. It may also save other registers on the stack to create additional working space, as needed. For FIQ, which is intended as a fast interrupt, some dedicated register space is available, R8_fiq to R13_fiq, without the need to save registers on the stack.

We pointed out in Chapter 3 that the LDM and STM instructions, which transfer multiple words, are convenient for handling stack operations. For example, using R13 as a stack pointer, a subroutine or an interrupt-service routine may save some registers and the return address as follows:

```
STMFD   R13!,{R0,R1,R2,R14}
```

Similarly, an LDMFD instruction can be used to restore the saved values. In the case of

an SWI or instruction prefetch exception, the value restored to R14 is also the correct return address. Hence, it can be restored directly to R15, thus effecting a return to the interrupted program, as follows:

```
LDMFD   R13!,{R0,R1,R2,R15}^
```

The "^" symbol at the end of the instruction has the same effect as the S suffix in the case of the SUBS instruction used earlier. It causes the processor to copy SPSR into CSPR at the same time it loads R15. Note that LDM cannot be used to return from an IRQ or FIQ interrupt because the contents of R14 must be corrected first, as shown in Table 4.2.

Example 4.4

An example of the use of interrupts is given in Figure 4.13, which shows the program in Figure 4.9 rewritten for ARM. We have assumed that the keyboard is connected to interrupt line IRQ and that the corresponding interrupt vector location contains a branch instruction to READ. We have also assumed that at the time this code segment is entered the memory buffer address, LINE, has been loaded into location PNTR. Locations PNTR and EOL are assumed to be sufficiently close in the address space that they can be reached using the Relative addressing mode. The Main program enables interrupts in both the keyboard interface and the processor by setting the KEN flag in the keyboard's CONTROL register and clearing the I mask in the processor status register. The I mask, bit 7, is cleared by loading the value $50 into CPSR using the MSR instruction.

In the interrupt-service routine, we use the LDM and STM instructions to save and restore registers and the SUBS instruction to return to the interrupted program. The address of the keyboard's DATAIN register in Figure 4.3 is loaded in a processor register using the ADR instruction described in Section 3.4.1. We have assumed that the address of the control register, CONTROL, is equal to DATAIN+3.

4.3.2 68000 INTERRUPT STRUCTURE

The 68000 has eight interrupt priority levels. The priority at which the processor is running at any given time is encoded in three bits of the processor status word, as shown in Figure 4.14, with level 0 being the lowest priority. I/O devices are connected to the 68000 using an arrangement similar to that in Figure 4.8*b*, in which interrupt requests are assigned priorities in the range 1 through 7. A request is accepted only if its priority is higher than that of the processor, with one exception: An interrupt request at level 7 is always accepted. This is an edge-triggered *nonmaskable* interrupt. When the processor accepts an interrupt request, the priority level indicated in the PS register is automatically raised to that of the request before the interrupt-service routine is executed. Thus, requests of equal or lower priority are disabled, except for level-7 interrupts, which are always enabled.

The processor automatically saves the contents of the program counter and the processor status word at the time of interruption. The PC is pushed onto the processor

Main program

```
        MOV     R0,#0
        STR     R0,EOL          Clear EOL flag.
        ADR     R1,DATAIN       Load address of register DATAIN.
        LDRB    R0,[R1,#3]      Get contents of CONTROL register.
        ORR     R0,R0,#4        Set bit KEN in register CONTROL
        STRB    R0,[R1,#3]        to enable keyboard interrupts.
        MOV     R0,#&50         Enable IRQ interrupts in processor
        MSR     CPSR,R0           and switch to user mode.
        .
        .
        .
```

IRQ Interrupt-service routine

```
READ    STMFD   R13!,{R0-R2,R14_irq}  Save R0, R1, and R14_irq on the stack.
        ADR     R1,DATAIN             Load address of register DATAIN.
        LDRB    R0,[R1]               Get input character.
        LDR     R2,PNTR               Load pointer value.
        STRB    R0,[R2],#1            Store character and increment pointer.
        STR     R2,PNTR               Update pointer value in the memory.
        CMPB    R0,#&0D               Check if Carriage Return.
        LDMNEFD R13!,{R0-R2,R14_irq}  If not, restore registers
        SUBNES  PC,R14_irq,#4           and return.
        LDRB    R0,[R1,#3]            Otherwise, get CONTROL register.
        AND     R0,R0,#&FB            Clear bit KEN
        STRB    R0,[R1,#3]              to disable keyboard interrupts.
        MOV     R0,#1                 Set EOL flag.
        STR     R0,EOL
        LDMFD   R13!,{R0-R2,R14}      Restore registers
        SUBS    PC,R14_irq,#4           and return.
```

Figure 4.13 An ARM interrupt-service routine to read an input line from a keyboard, based on Figure 4.9.

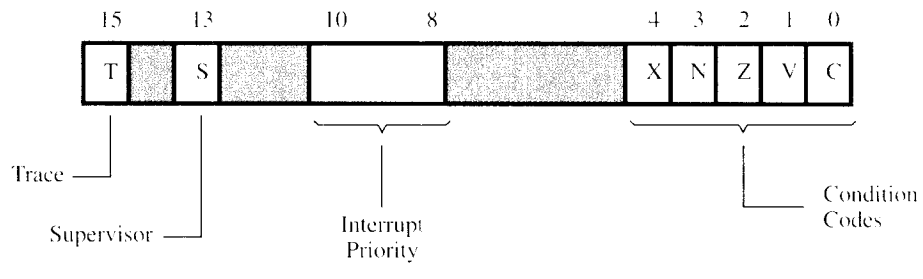


Figure 4.14 Processor status register in the 68000 processor.

stack followed by the PS, using register A7 as the stack pointer. A Return-from-interrupt instruction, called Return-from-exception (RTE) in the 68000 assembly language, pops the top element of the stack into the PS and pops the next element into the PC. As shown in Figure 4.14, the PS register contains a Supervisor bit, S, and a Trace bit, T. The S bit determines whether the processor is running in the Supervisor mode (S = 1) or User mode (S = 0). The T bit enables a special type of interrupt called a trace exception, as described in Section 4.2.4. This information is saved automatically at the time an interrupt is accepted and restored at the end of interrupt servicing. Any additional information to be saved, such as the contents of general-purpose registers, must be saved and restored explicitly inside the interrupt-service routine.

The 68000 processor uses vectored interrupts. When it accepts an interrupt request, it obtains the starting address of the interrupt-service routine from an interrupt vector stored in the main memory. There are 256 interrupt vectors, numbered 0 through 255. Each vector consists of 32 bits that constitute the required starting address. When a device requests an interrupt, it may point to the vector that should be used by sending an 8-bit vector number to the processor in response to the interrupt-acknowledge signal. As an alternative, the 68000 also provides an *autovector* facility. Instead of sending a vector number, the device can activate a special bus control line to indicate that it wishes to use the autovector facility. In this case, the processor chooses one of seven vectors provided for this purpose, based on the priority level of the interrupt request.

Example 4.5

An example of the use of interrupts in the 68000 is shown in Figure 4.15. This is the program given in Figure 4.9 rewritten for the 68000. We have assumed that the keyboard interface uses the autovector facility and generates interrupt requests at level 2. Hence, to enable interrupts, the processor priority must be set at a level less than 2. When the bit pattern $100 is loaded into register PS, it sets the processor's priority to 1.

4.3.3 PENTIUM INTERRUPT STRUCTURE

The IA-32 architecture, of which the Pentium processors are examples, uses two interrupt-request lines, a nonmaskable interrupt (NMI) and a maskable interrupt, also called user interrupt request, INTR. Interrupt requests on NMI are always accepted by the processor. Requests on INTR are accepted only if they have a higher privilege level than the program currently executing, as we will explain shortly. INTR interrupts can also be enabled or disabled by setting an interrupt-enable bit in the processor status register.

In addition to external interrupts, there are many events that arise during program execution that can cause an exception. These include invalid opcodes, division errors, overflow, and many others. They also include trace and breakpoint interrupts.

Main program

```
        MOVE.L    #LINE.PNTR      Initialize buffer pointer.
        CLR       EOL             Clear end-of-line indicator.
        ORI.B     #4.CONTROL      Set bit KEN.
        MOVE      #$100.SR        Set processor priority to 1.
        .
        .
        .
```

Interrupt-service routine

```
READ    MOVEM.L   A0/D0,-(A7)     Save registers A0, D0 on stack.
        MOVEA.L   PNTR.A0         Load address pointer.
        MOVE.B    DATAIN.D0       Get input character.
        MOVE.B    D0,(A0)+        Store it in memory buffer.
        MOVE.L    A0.PNTR         Update pointer.
        CMPI.B    #$0D.D0         Check if Carriage Return.
        BNE       RTRN
        MOVE      #1.EOL          Indicate end of line.
        ANDI.B    #$FB.CONTROL    Clear bit KEN.
RTRN    MOVEM.L   (A7)+.A0/D0     Restore registers D0, A0.
        RTE
```

Figure 4.15 A 68000 interrupt-service routine to read an input line from a keyboard, based on Figure 4.9.

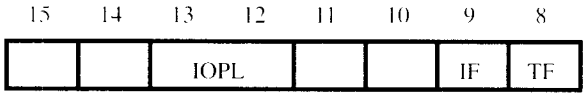


Figure 4.16 Part of the Pentium's processor status register.

The occurrence of any of these events causes the processor to branch to an interrupt-service routine. Each interrupt or exception is assigned a vector number. In the case of INTR, the vector number is sent by the I/O device over the bus when the interrupt request is acknowledged. For all other exceptions, the vector number is preassigned. Based on the vector number, the processor determines the starting address of the interrupt-service routine from a table called the Interrupt Descriptor Table.

A companion chip to the Pentium processor is called the Advanced Programmable Interrupt Controller (APIC). Various I/O devices are connected to the processor through this chip. The interrupt controller implements a priority structure among different devices and sends an appropriate vector number to the processor for each device.

The processor status register, which is called EFLAGS in Intel literature, is shown in Figure 3.37. Figure 4.16 shows bits 8 to 15 of this register, which contain the Interrupt Enable Flag (IF), the Trap flag (TF) and the I/O Privilege Level (IOPL). When IF = 1, INTR interrupts are accepted. The Trap flag enables trace interrupts after every instruction.

The Pentium processor has a sophisticated privilege structure, whereby different parts of the operating system execute at one of four levels of privilege. A different segment in the processor address space is used for each level. Switching from one level to another involves a number of checks implemented in a mechanism called a gate. This enables a highly secure OS to be constructed. It is also possible for the processor to run in a simple mode in which no privileges are implemented and all programs run in the same segment. We will only discuss this simple case here.

When an interrupt request is received or when an exception occurs, the processor takes the following actions:

1. It pushes the processor status register, the current segment register (CS), and the instruction pointer (EIP) onto the processor stack pointed to by the processor stack pointer, ESP.
2. In the case of an exception resulting from an abnormal execution condition, it pushes a code on the stack describing the cause of the exception.
3. It clears the corresponding interrupt-enable flag, if appropriate, so that further interrupts from the same source are disabled.
4. It fetches the starting address of the interrupt-service routine from the Interrupt Descriptor Table based on the vector number of the interrupt and loads this value into EIP, then resumes execution.

After servicing the interrupt request, for example, by transferring input or output data, the interrupt-service routine returns to the interrupted program using a return from interrupt instruction, IRET. This instruction pops EIP, CS, and the processor status register from the stack into the corresponding registers, thus restoring the processor state.

As in the case of subroutines, the interrupt-service routine may create temporary work space by saving registers or using the stack frame for local variables. It must restore any saved registers and ensure that the stack pointer ESP is pointing to the return address before executing the IRET instruction.

Example 4.6

The example in Figure 4.9 rewritten for the Pentium is shown in Figure 4.17. We have assumed that the keyboard sends an interrupt request with vector number 32 and that the corresponding entry in the Interrupt Descriptor Table has been loaded with the starting address READ of the interrupt-service routine. Interrupts are enabled in the processor using the STI instruction, which sets to 1 the IF flag in the processor status register.

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

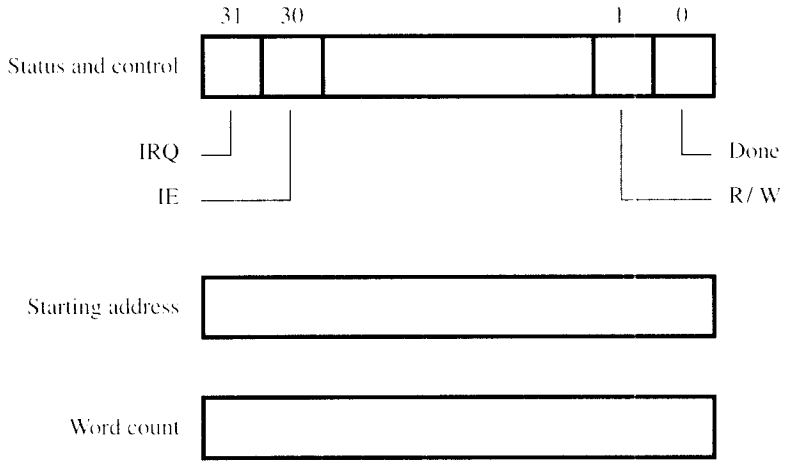


Figure 4.18 Registers in a DMA interface.

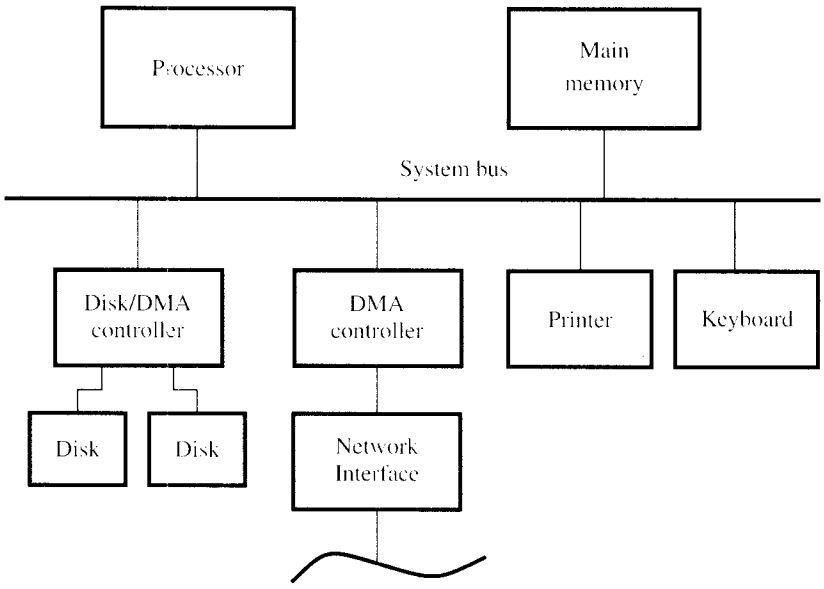


Figure 4.19 Use of DMA controllers in a computer system.

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

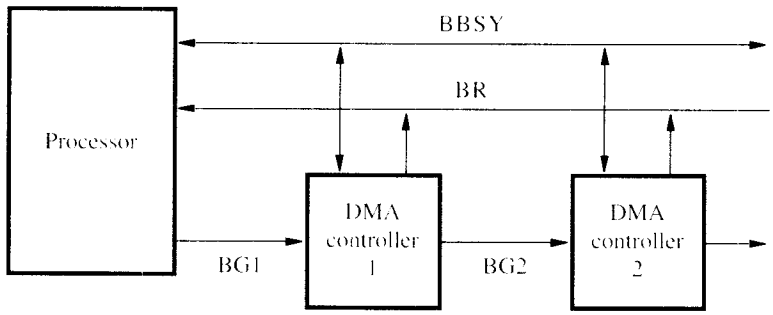


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.

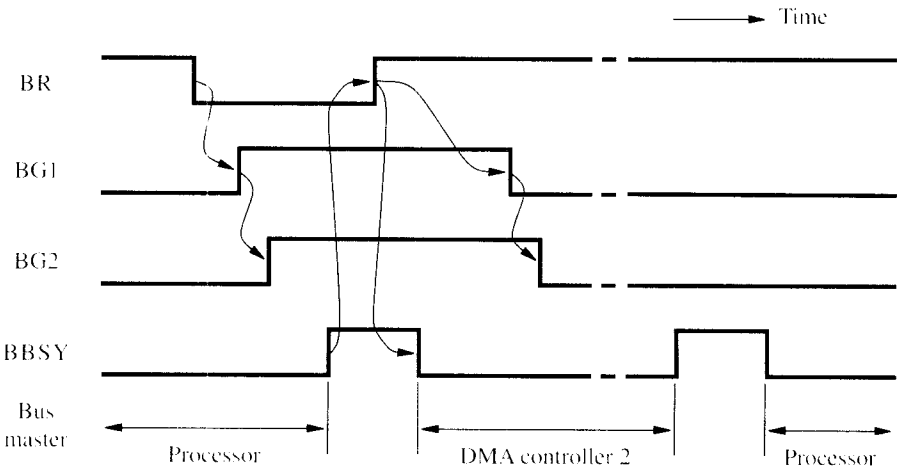


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

devices that it is using the bus by activating another open-collector line called Bus-Busy, $\overline{\text{BBSY}}$. 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

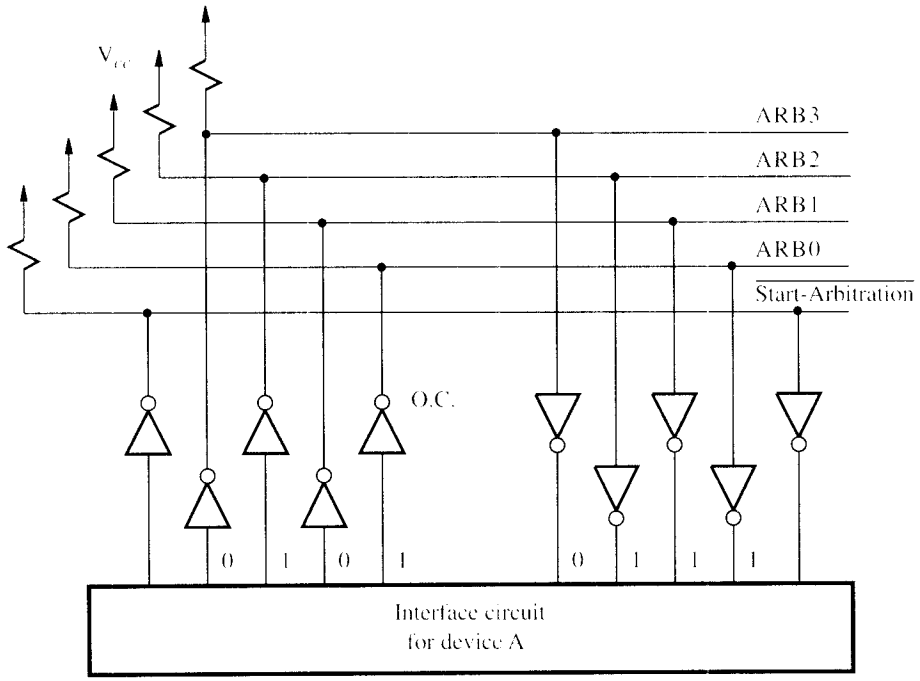


Figure 4.22 A distributed arbitration scheme.

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*.

4.5.1 SYNCHRONOUS BUS

In a *synchronous* bus, all devices derive timing information from a common clock line. Equally spaced pulses on this line define equal time intervals. In the simplest form of a synchronous bus, each of these intervals constitutes a *bus cycle* during which one data transfer can take place. Such a scheme is illustrated in Figure 4.23. The address and data lines in this and subsequent figures are shown as high and low at the same time. This is a common convention indicating that some lines are high and some low, depending on the particular address or data pattern being transmitted. The crossing points indicate the times at which these patterns change. A signal line in an indeterminate or high impedance state is represented by an intermediate level half-way between the low and high signal levels.

Let us consider the sequence of events during an input (read) operation. At time t_0, the master places the device address on the address lines and sends an appropriate command on the control lines. In this case, the command will indicate an input operation and specify the length of the operand to be read, if necessary. Information travels over the bus at a speed determined by its physical and electrical characteristics. The clock pulse width, $t_1 - t_0$, must be longer than the maximum propagation delay between two devices connected to the bus. It also has to be long enough to allow all devices to decode the address and control signals so that the addressed device (the slave) can respond at time t_1. It is important that slaves take no action or place any data on the bus before t_1. The information on the bus is unreliable during the period t_0 to t_1 because signals are changing state. The addressed slave places the requested input data on the data lines at time t_1.

At the end of the clock cycle, at time t_2, the master *strobes* the data on the data lines into its input buffer. In this context, "strobe" means to capture the values of the

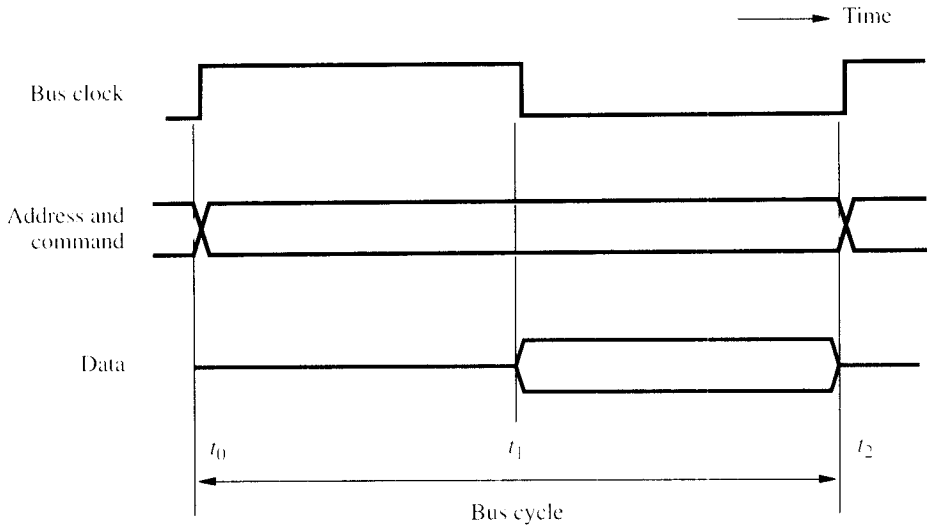


Figure 4.23 Timing of an input transfer on a synchronous bus.

data at a given instant and store them into a buffer. For data to be loaded correctly into any storage device, such as a register built with flip-flops, the data must be available at the input of that device for a period greater than the setup time of the device (see Appendix A). Hence, the period $t_2 - t_1$ must be greater than the maximum propagation time on the bus plus the setup time of the input buffer register of the master.

A similar procedure is followed for an output operation. The master places the output data on the data lines when it transmits the address and command information. At time t_2, the addressed device strobes the data lines and loads the data into its data buffer.

The timing diagram in Figure 4.23 is an idealized representation of the actions that take place on the bus lines. The exact times at which signals actually change state are somewhat different from those shown because of propagation delays on bus wires and in the circuits of the devices. Figure 4.24 gives a more realistic picture of what happens in practice. It shows two views of each signal, except the clock. Because signals take time to travel from one device to another, a given signal transition is seen by different devices at different times. One view shows the signal as seen by the master and the other as seen by the slave. We assume that the clock changes are seen at the same time by all devices on the bus. System designers spend considerable effort to ensure that the clock signal satisfies this condition.

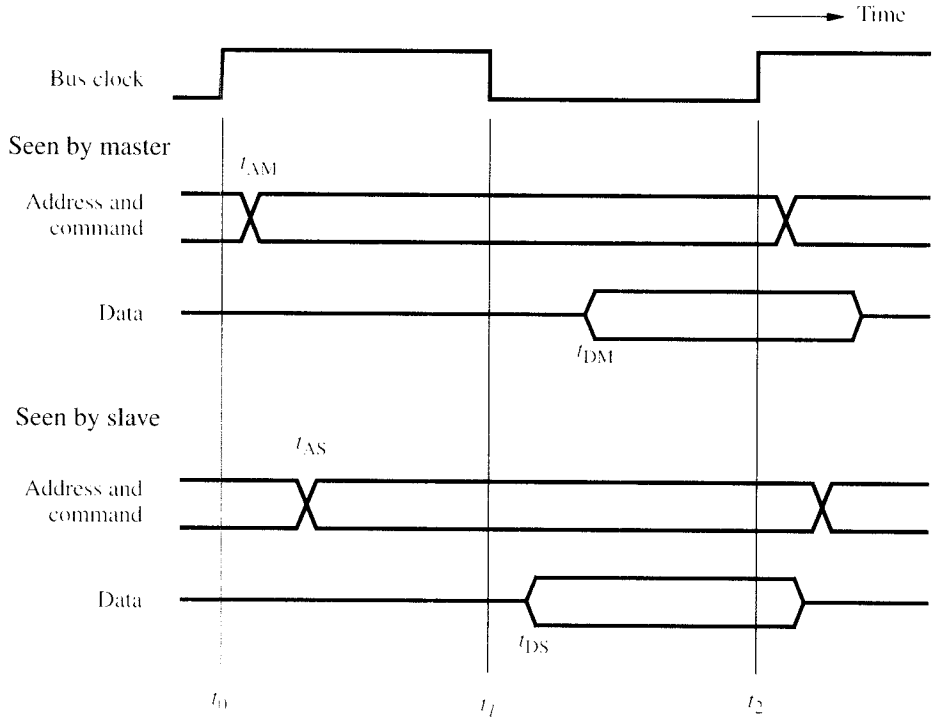


Figure 4.24 A detailed timing diagram for the input transfer of Figure 4.23.

The master sends the address and command signals on the rising edge at the beginning of clock period 1 (t_0). However, these signals do not actually appear on the bus until t_{AM}, largely due to the delay in the bus driver circuit. A while later, at t_{AS}, the signals reach the slave. The slave decodes the address and at t_1 sends the requested data. Here again, the data signals do not appear on the bus until t_{DS}. They travel toward the master and arrive at t_{DM}. At t_2, the master loads the data into its input buffer. Hence the period $t_2 - t_{DM}$ is the setup time for the master's input buffer. The data must continue to be valid after t_2 for a period equal to the hold time of that buffer.

Timing diagrams in the literature often give only the simplified picture in Figure 4.23, particularly when the intent is to give a conceptual overview of how data are transferred. But, actual signals will always involve delays as shown in Figure 4.24.

Multiple-Cycle Transfers

The scheme described above results in a simple design for the device interface. However, it has some limitations. Because a transfer has to be completed within one clock cycle, the clock period, $t_2 - t_0$, must be chosen to accommodate the longest delays on the bus and the slowest device interface. This forces all devices to operate at the speed of the slowest device.

Also, the processor has no way of determining whether the addressed device has actually responded. It simply assumes that, at t_2, the output data have been received by the I/O device or the input data are available on the data lines. If, because of a malfunction, the device does not respond, the error will not be detected.

To overcome these limitations, most buses incorporate control signals that represent a response from the device. These signals inform the master that the slave has recognized its address and that it is ready to participate in a data-transfer operation. They also make it possible to adjust the duration of the data-transfer period to suit the needs of the participating devices. To simplify this process, a high-frequency clock signal is used such that a complete data transfer cycle would span several clock cycles. Then, the number of clock cycles involved can vary from one device to another.

An example of this approach is shown in Figure 4.25. During clock cycle 1, the master sends address and command information on the bus, requesting a read operation. The slave receives this information and decodes it. On the following active edge of the clock, that is, at the beginning of clock cycle 2, it makes a decision to respond and begins to access the requested data. We have assumed that some delay is involved in getting the data, and hence the slave cannot respond immediately. The data become ready and are placed on the bus in clock cycle 3. At the same time, the slave asserts a control signal called Slave-ready. The master, which has been waiting for this signal, strobes the data into its input buffer at the end of clock cycle 3. The bus transfer operation is now complete, and the master may send a new address to start a new transfer in clock cycle 4.

The Slave-ready signal is an acknowledgment from the slave to the master, confirming that valid data have been sent. In the example in Figure 4.25, the slave responds in cycle 3. Another device may respond sooner or later. The Slave-ready signal allows the duration of a bus transfer to change from one device to another. If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, then aborts the operation. This could be the result of an incorrect address or a device malfunction.

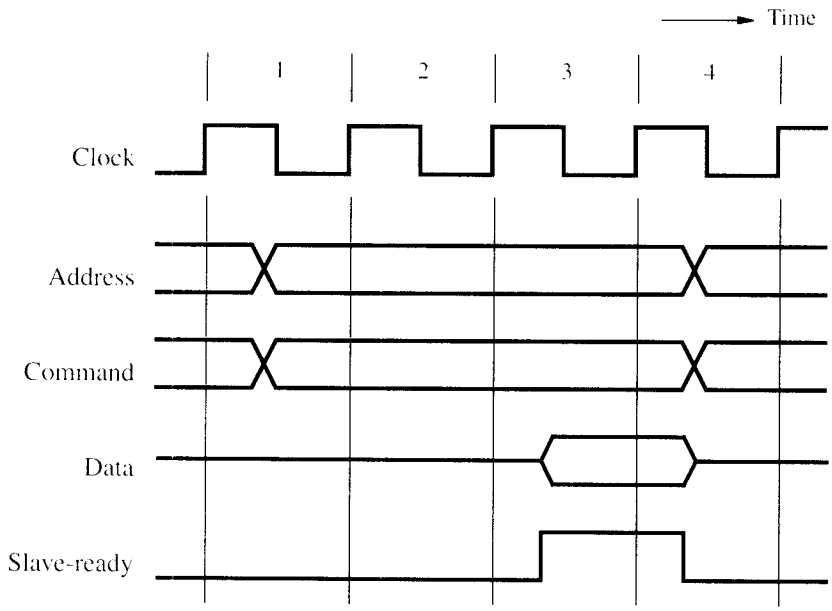


Figure 4.25 An input transfer using multiple clock cycles.

Note that the clock signal used on a computer bus is not necessarily the same as the processor clock. The latter is often much faster because it controls internal operation on the processor chip. The delays encountered by signals internal to a chip are much less than on a bus that interconnects chips on a printed circuit board, for example. Clock frequencies are highly technology dependent. In modern processor chips, clock frequencies above 500 MHz are typical. On memory and I/O buses, the clock frequency may be in the range 50 to 150 MHz.

Many computer buses, such as the processor buses of Pentium and ARM, use a scheme similar to that illustrated in Figure 4.25 to control the transfer of data. The PCI bus standard described in Section 4.7.1 is also very similar. We will now present a different approach that does not use a clock signal at all.

4.5.2 ASYNCHRONOUS BUS

An alternative scheme for controlling data transfers on the bus is based on the use of a *handshake* between the master and the slave. The concept of a handshake is a generalization of the idea of the Slave-ready signal in Figure 4.25. The common clock is replaced by two timing control lines, Master-ready and Slave-ready. The first is asserted by the master to indicate that it is ready for a transaction, and the second is a response from the slave.

In principle, a data transfer controlled by a handshake protocol proceeds as follows. The master places the address and command information on the bus. Then it indicates

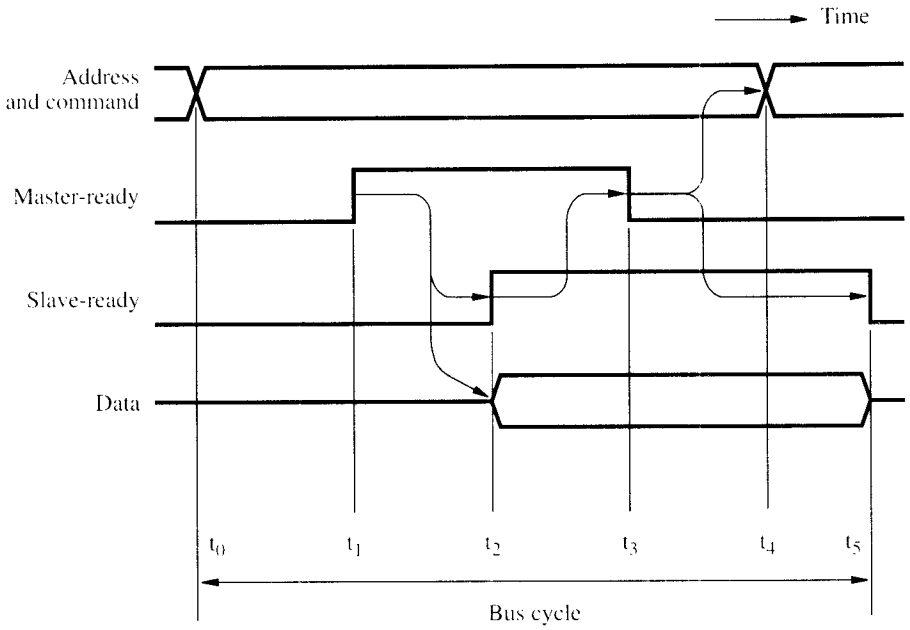


Figure 4.26 Handshake control of data transfer during an input operation.

to all devices that it has done so by activating the Master-ready line. This causes all devices on the bus to decode the address. The selected slave performs the required operation and informs the processor it has done so by activating the Slave-ready line. The master waits for Slave-ready to become asserted before it removes its signals from the bus. In the case of a read operation, it also strobes the data into its input buffer.

An example of the timing of an input data transfer using the handshake scheme is given in Figure 4.26, which depicts the following sequence of events:

t_0 — The master places the address and command information on the bus, and all devices on the bus begin to decode this information.

t_1 — The master sets the Master-ready line to 1 to inform the I/O devices that the address and command information is ready. The delay $t_1 - t_0$ is intended to allow for any *skew* that may occur on the bus. Skew occurs when two signals simultaneously transmitted from one source arrive at the destination at different times. This happens because different lines of the bus may have different propagation speeds. Thus, to guarantee that the Master-ready signal does not arrive at any device ahead of the address and command information, the delay $t_1 - t_0$ should be larger than the maximum possible bus skew. (Note that, in the synchronous case, bus skew is accounted for as a part of the maximum propagation delay.) When the address information arrives at any device, it is decoded by the interface circuitry. Sufficient time should be allowed for the interface circuitry to decode the address. The delay needed can be included in the period $t_1 - t_0$.

t_2 — The selected slave, having decoded the address and command information, performs the required input operation by placing the data from its data register on the data lines. At the same time, it sets the Slave-ready signal to 1. If extra delays are introduced by the interface circuitry before it places the data on the bus, the slave must delay the Slave-ready signal accordingly. The period $t_2 - t_1$ depends on the distance between the master and the slave and on the delays introduced by the slave's circuitry. It is this variability that gives the bus its asynchronous nature.

t_3 — The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. However, since it was assumed that the device interface transmits the Slave-ready signal at the same time that it places the data on the bus, the master should allow for bus skew. It must also allow for the setup time needed by its input buffer. After a delay equivalent to the maximum bus skew and the minimum setup time, the master strobes the data into its input buffer. At the same time, it drops the Master-ready signal, indicating that it has received the data.

t_4 — The master removes the address and command information from the bus. The delay between t_3 and t_4 is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.

t_5 — When the device interface receives the 1 to 0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.

The timing for an output operation, illustrated in Figure 4.27, is essentially the same as for an input operation. In this case, the master places the output data on the data lines

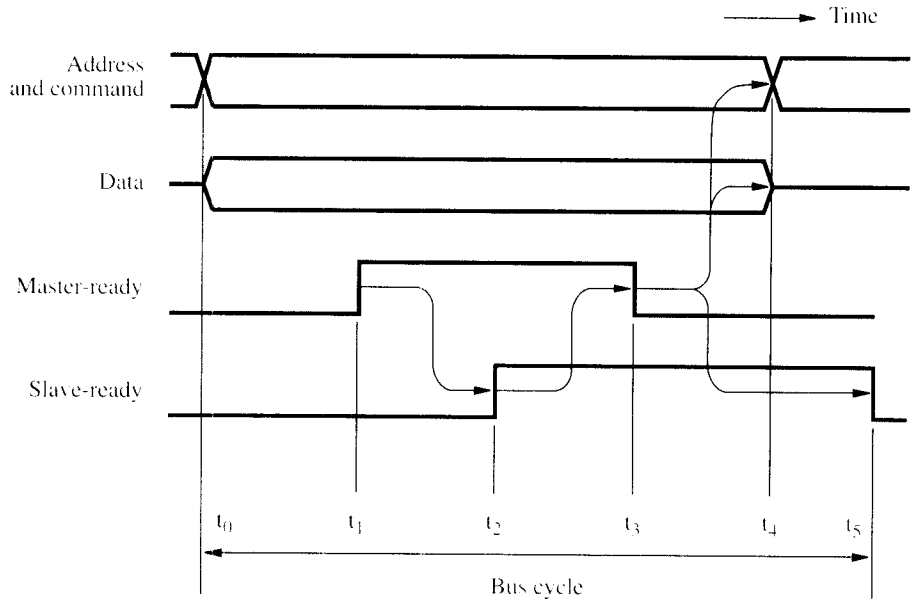


Figure 4.27 Handshake control of data transfer during an output operation.

at the same time that it transmits the address and command information. The selected slave strobes the data into its output buffer when it receives the Master-ready signal and indicates that it has done so by setting the Slave-ready signal to 1. The remainder of the cycle is identical to the input operation.

In the timing diagrams in Figures 4.26 and 4.27, it is assumed that the master compensates for bus skew and address decoding delay. It introduces the delays from t_0 to t_1 and from t_3 to t_4 for this purpose. If this delay provides sufficient time for the I/O device interface to decode the address, the interface circuit can use the Master-ready signal directly to gate other signals to or from the bus. This point will become clearer when we study the interface circuit examples in the next section.

The handshake signals in Figures 4.26 and 4.27 are fully interlocked. A change of state in one signal is followed by a change in the other signal. Hence this scheme is known as a *full handshake*. It provides the highest degree of flexibility and reliability.

4.5.3 DISCUSSION

Many variations of the bus techniques just described are found in commercial computers. For example, the bus in the 68000 family of processors has two modes of operation, one asynchronous and one synchronous. The choice of a particular design involves trade-offs among factors such as:

- Simplicity of the device interface
- Ability to accommodate device interfaces that introduce different amounts of delay
- Total time required for a bus transfer
- Ability to detect errors resulting from addressing a nonexistent device or from an interface malfunction

The main advantage of the asynchronous bus is that the handshake process eliminates the need for synchronization of the sender and receiver clocks, thus simplifying timing design. Delays, whether introduced by the interface circuits or by propagation over the bus wires, are readily accommodated. When these delays change, for example, due to a change in load when an interface circuit is added or removed, the timing of data transfer adjusts automatically based on the new conditions. For a synchronous bus, clock circuitry must be designed carefully to ensure proper synchronization, and delays must be kept within strict bounds.

The rate of data transfer on an asynchronous bus controlled by a full handshake is limited by the fact that each transfer involves two round-trip delays (four end-to-end delays). This can be readily seen in Figures 4.26 and 4.27 as each transition on Slave-ready must wait for the arrival of a transition on Master-ready, and vice versa. On synchronous buses, the clock period need only accommodate one end-to-end propagation delay. Hence, faster transfer rates can be achieved. To accommodate a slow device, additional clock cycles are used, as described above. Most of today's high-speed buses use this approach.

4.6 INTERFACE CIRCUITS

An I/O interface consists of the circuitry required to connect an I/O device to a computer bus. On one side of the interface we have the bus signals for address, data, and control. On the other side we have a data path with its associated controls to transfer data between the interface and the I/O device. This side is called a *port*, and it can be classified as either a parallel or a serial port. A parallel port transfers data in the form of a number of bits, typically 8 or 16, simultaneously to or from the device. A serial port transmits and receives data one bit at a time. Communication with the bus is the same for both formats; the conversion from the parallel to the serial format, and vice versa, takes place inside the interface circuit.

In the case of a parallel port, the connection between the device and the computer uses a multiple-pin connector and a cable with as many wires, typically arranged in a flat configuration. The circuits at either end are relatively simple, as there is no need to convert between parallel and serial formats. This arrangement is suitable for devices that are physically close to the computer. For longer distances, the problem of timing skew mentioned earlier limits the data rates that can be used. The serial format is much more convenient and cost-effective where longer cables are needed. Serial transmission formats will be discussed in Chapter 10.

Before discussing a specific interface circuit example, let us recall the functions of an I/O interface. According to the discussion in Section 4.1, an I/O interface does the following:

1. Provides a storage buffer for at least one word of data (or one byte, in the case of byte-oriented devices)
2. Contains status flags that can be accessed by the processor to determine whether the buffer is full (for input) or empty (for output)
3. Contains address-decoding circuitry to determine when it is being addressed by the processor
4. Generates the appropriate timing signals required by the bus control scheme
5. Performs any format conversion that may be necessary to transfer data between the bus and the I/O device, such as parallel-serial conversion in the case of a serial port

4.6.1 PARALLEL PORT

We now explain the key aspects of interface design with a practical example. First, we describe circuits for an 8-bit input port and an 8-bit output port. Then, we combine the two circuits to show how the interface for a general-purpose 8-bit parallel port can be designed. We assume that the interface circuit is connected to a 32-bit processor that uses memory-mapped I/O and the asynchronous bus protocol depicted in Figures 4.26 and 4.27. We will also show how the design can be modified to suit the bus protocol in Figure 4.25.

Figure 4.28 shows the hardware components needed for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally

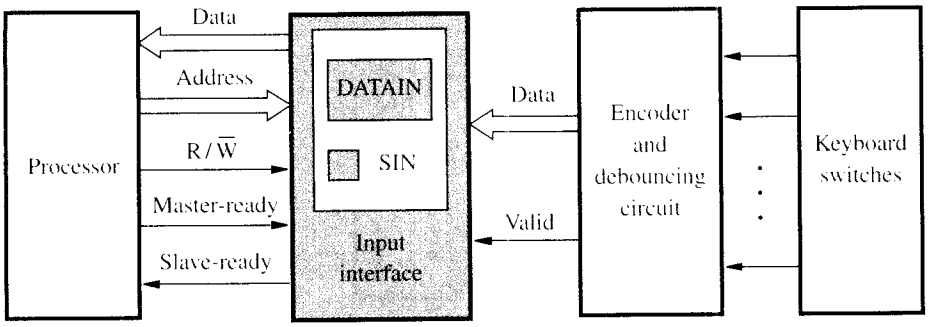


Figure 4.28 Keyboard to processor connection.

open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such push-button switches is that the contacts bounce when a key is pressed. Although bouncing may last only one or two milliseconds, this is long enough for the computer to observe a single pressing of a key as several distinct electrical events; this single pressing could be erroneously interpreted as the key being pressed and released rapidly several times. The effect of bouncing must be eliminated. We can do this in two ways: A simple debouncing circuit can be included, or a software approach can be used. When debouncing is implemented in software, the I/O routine that reads a character from the keyboard waits long enough to ensure that bouncing has subsided. Figure 4.28 illustrates the hardware approach; debouncing circuits are included as a part of the encoder block.

The output of the encoder consists of the bits that represent the encoded character and one control signal called Valid, which indicates that a key is being pressed. This information is sent to the interface circuit, which contains a data register, DATAIN, and a status flag, SIN. When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code to be loaded into DATAIN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register. The interface circuit is connected to an asynchronous bus on which transfers are controlled using the handshake signals Master-ready and Slave-ready, as indicated in Figure 4.26. The third control line, R/$\overline{W}$ distinguishes read and write transfers.

Figure 4.29 shows a suitable circuit for an input interface. The output lines of the DATAIN register are connected to the data lines of the bus by means of three-state drivers, which are turned on when the processor issues a read instruction with the address that selects this register. The SIN signal is generated by a status flag circuit. This signal is also sent to the bus through a three-state driver. It is connected to bit D0, which means it will appear as bit 0 of the status register. Other bits of this register do not contain valid information. An address decoder is used to select the input interface when the high-order 31 bits of an address correspond to any of the addresses assigned to this interface. Address bit A0 determines whether the status or the data registers is to be read when the Master-ready signal is active. The control handshake is accomplished by activating the Slave-ready signal when either Read-status or Read-data is equal to 1.

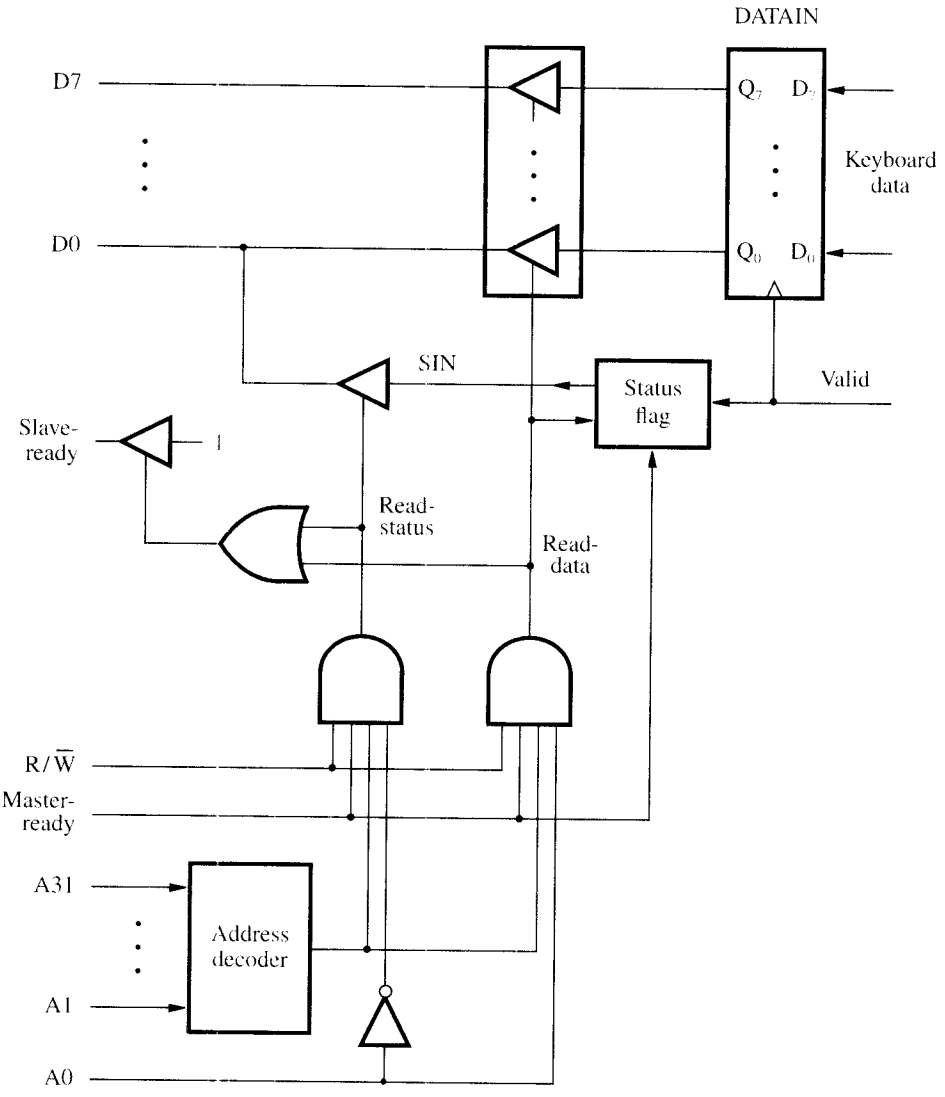


Figure 4.29 Input interface circuit.

A possible implementation of the status flag circuit is shown in Figure 4.30. An edge-triggered D flip-flop is set to 1 by a rising edge on the Valid signal line. This event changes the state of the NOR latch such that SIN is set to 1. The state of this latch must not change while SIN is being read by the processor. Hence, the circuit ensures that SIN can be set only while Master-ready is equal to 0. Both the flip-flop and the latch are reset to 0 when Read-data is set to 1 to read the DATAIN register.

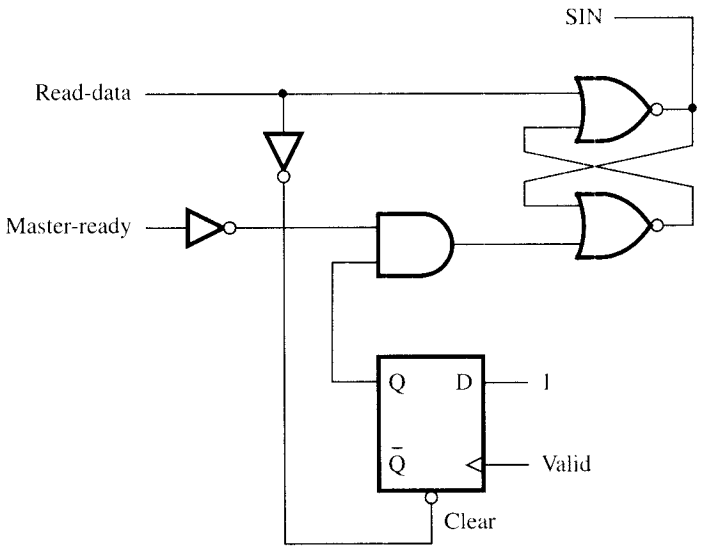


Figure 4.30 Circuit for the status flag block in Figure 4.29.

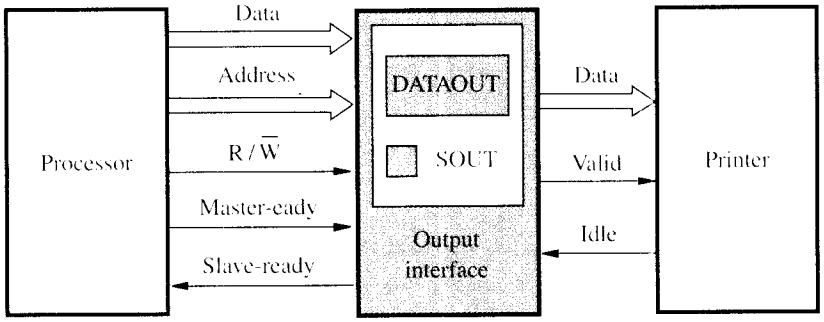


Figure 4.31 Printer to processor connection.

Let us now consider an output interface that can be used to connect an output device, such as a printer, to a processor, as shown in Figure 4.31. The printer operates under control of the handshake signals Valid and Idle in a manner similar to the handshake used on the bus with the Master-ready and Slave-ready signals. When it is ready to accept a character, the printer asserts its Idle signal. The interface circuit can then place a new character on the data lines and activate the Valid signal. In response, the printer starts printing the new character and negates the Idle signal, which in turn causes the interface to deactivate the Valid signal.

The interface contains a data register, DATAOUT, and a status flag, SOUT. The SOUT flag is set to 1 when the printer is ready to accept another character, and it is cleared to 0 when a new character is loaded into DATAOUT by the processor. Figure 4.32

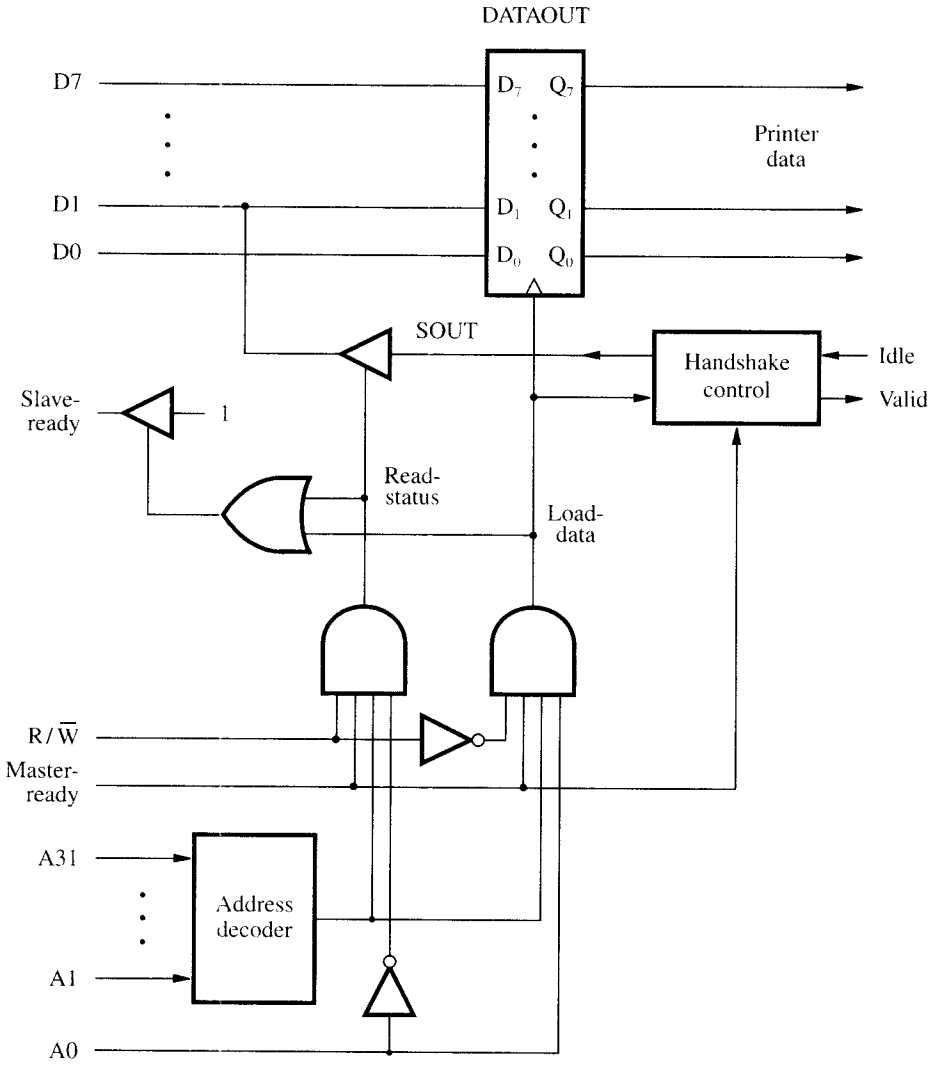


Figure 4.32 Output interface circuit.

shows an implementation of this interface. Its operation is similar to the input interface of Figure 4.29. The only significant difference is the handshake control circuit, the detailed design of which we leave as an exercise for the reader.

The input and output interfaces just described can be combined into a single interface, as shown in Figure 4.33. In this case, the overall interface is selected by the high-order 30 bits of the address. Address bits A1 and A0 select one of the three addressable locations in the interface, namely, the two data registers and the status register. The status register contains the flags SIN and SOUT in bits 0 and 1, respectively. Since

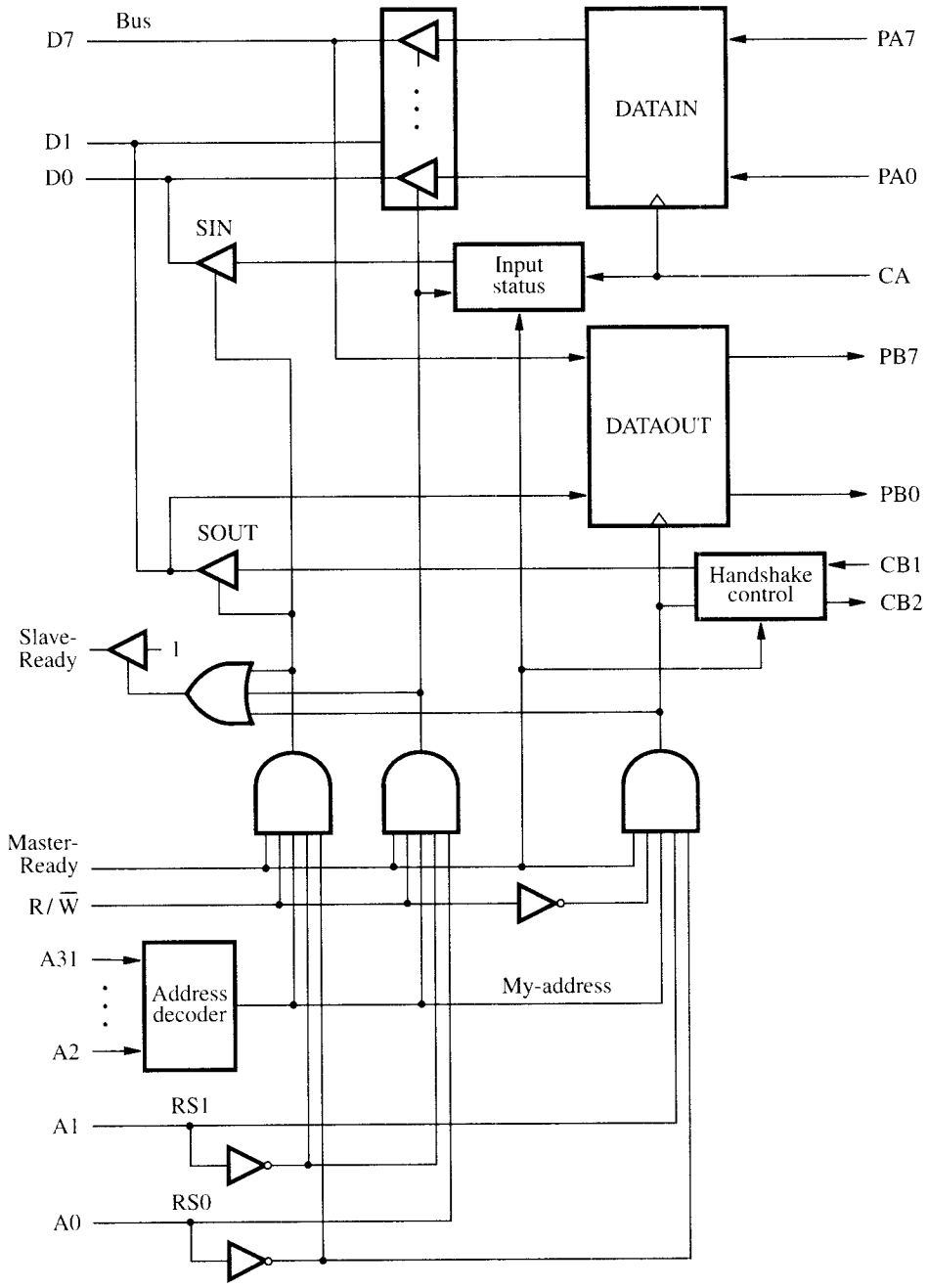


Figure 4.33 Combined input/output interface circuit.

such locations in I/O interfaces are often referred to as registers, we have used the labels RS1 and RS0 (for Register Select) to denote the two inputs that determine the register being selected.

The circuit in Figure 4.33 has separate input and output data lines for connection to an I/O device. A more flexible parallel port is created if the data lines to I/O devices are bidirectional. Figure 4.34 shows a general-purpose parallel interface circuit that can be configured in a variety of ways. Data lines P7 through P0 can be used for either input or output purposes. For increased flexibility, the circuit makes it possible for some lines to serve as inputs and some lines to serve as outputs, under program control. The

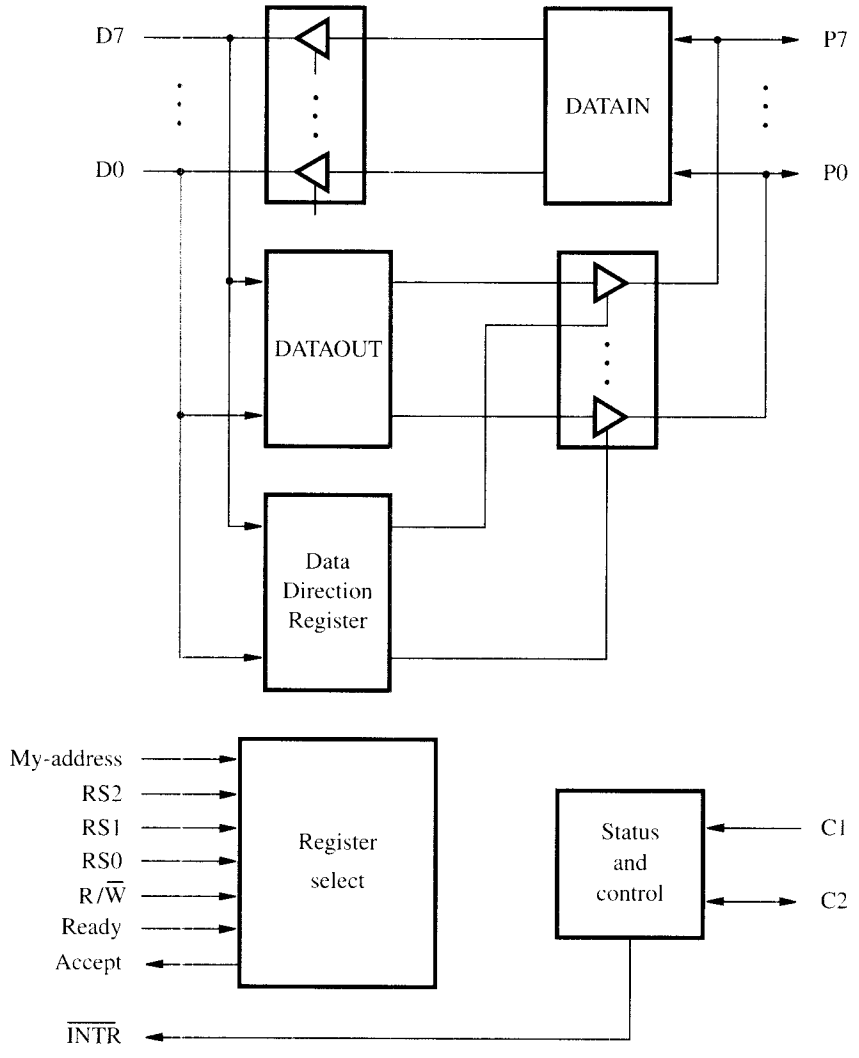


Figure 4.34 A general 8-bit parallel interface.

DATAOUT register is connected to these lines via three-state drivers that are controlled by a data direction register, DDR. The processor can write any 8-bit pattern into DDR. For a given bit, if the DDR value is 1, the corresponding data line acts as an output line; otherwise, it acts as an input line.

Two lines, C1 and C2, are provided to control the interaction between the interface circuit and the I/O device it serves. These lines are also programmable. Line C2 is bidirectional to provide several different modes of signaling, including the handshake. Not all the internal details are shown in the figure, but we can see how they may correspond to those in Figure 4.33. The Ready and Accept lines are the handshake control lines on the processor bus side, and hence would be connected to Master-ready and Slave-ready. The input signal My-address should be connected to the output of an address decoder that recognizes the address assigned to the interface. There are three register select lines, allowing up to eight registers in the interface, input and output data, data direction, and control and status registers for various modes of operation. An interrupt request output, $\overline{\text{INTR}}$, is also provided. It should be connected to the interrupt-request line on the computer bus.

Parallel interface circuits that have the features illustrated in Figure 4.34 are often encountered in practice. An example of their use in an embedded system is described in Chapter 9. Instead of having just one port for connecting an I/O device, two or more ports may be provided.

Let us now examine how the interface circuits in Figures 4.28 to 4.34 can be changed to work with the bus protocol of Figure 4.25. A modified circuit for the interface in Figure 4.32 is shown in Figure 4.35. We have introduced the Timing logic block to generate the Load-data and Read-status signals. The state diagram for this block is given at the bottom of the figure. Initially, the machine is in the Idle state. When the output of the address decoder, My-address, shows that this interface is being addressed, the machine changes state to Respond. As a result it asserts Go, which in turn asserts either Load-data or Read-status, depending on address bit A0 and the state of the R/$\overline{\text{W}}$ line.

A timing diagram for an output operation is shown in Figure 4.36. The processor sends the data at the same time as the address, in clock cycle 1. The Timing logic sets Go to 1 at the beginning of clock cycle 2, and the rising edge of that signal loads the output data into register DATAOUT. An input operation that reads the status register follows a similar timing pattern. The Timing logic block moves to the Respond state directly from the Idle state because the requested data are available in a register and can be transmitted immediately. As a result, the transfer is one clock cycle shorter than that shown in Figure 4.25. In a situation where some time is needed before the data becomes available, the state machine should enter a wait state first and move to Respond only when the data are ready.

In concluding the discussion of these interface circuit examples, we should point out that we have used simplified representations of some signals to help in readability and understanding of the ideas. In practice, the Slave-ready signal is likely to be an open-drain signal and would be called $\overline{\text{Slave-ready}}$, for the same reasons as for $\overline{\text{INTR}}$. This line must have a pull-up resistor connected to ensure that it is always in the negated (high-voltage) state except when it is asserted (pulled down) by some device.

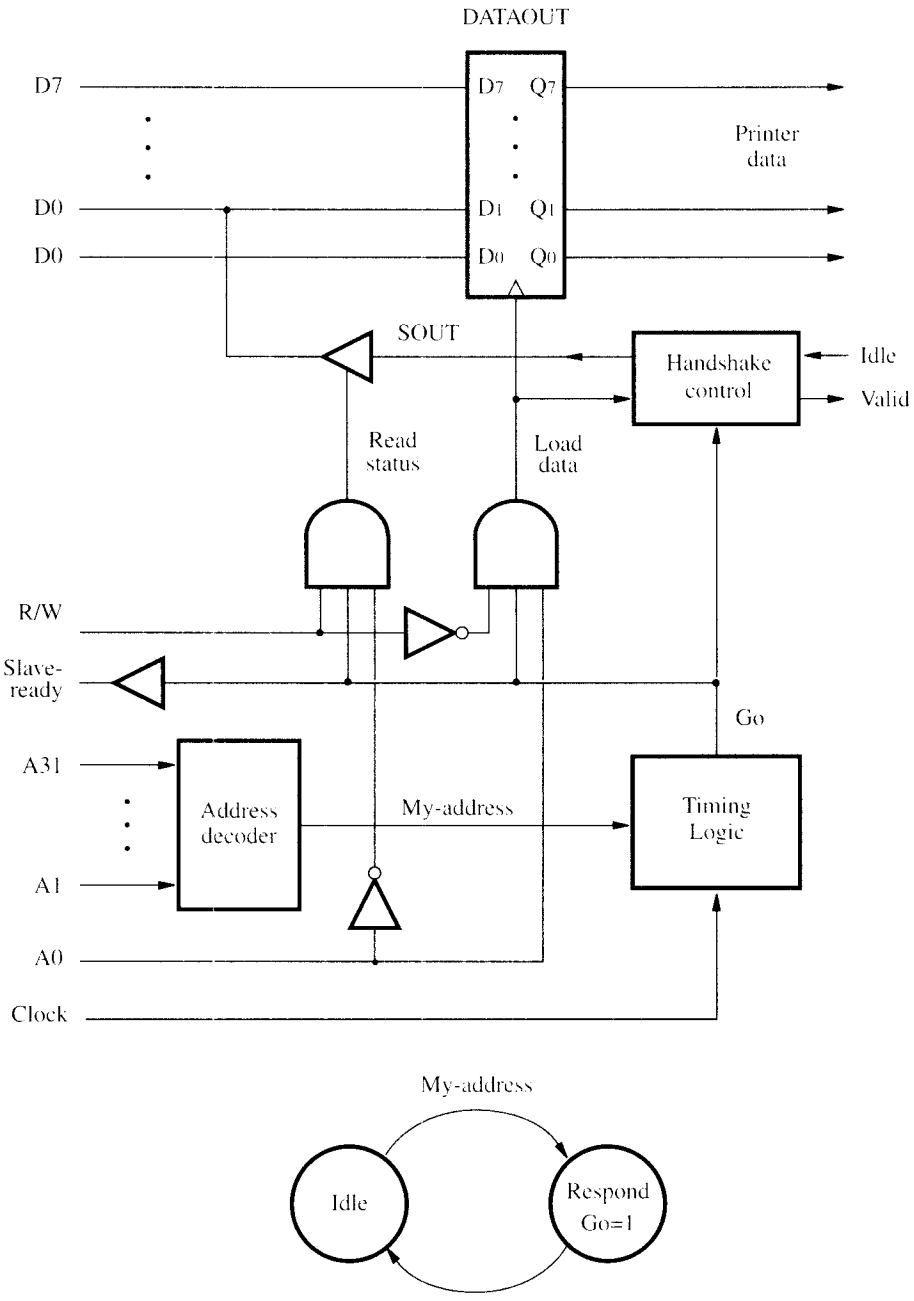


Figure 4.35 A parallel port interface for the bus of Figure 4.25, with a state-diagram for the timing logic.

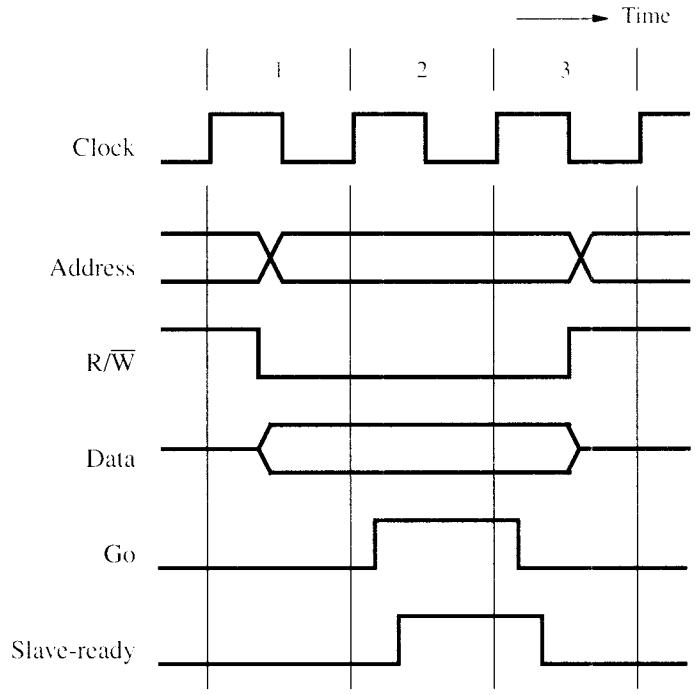


Figure 4.36 Timing for the output interface in Figure 4.35.

4.6.2 SERIAL PORT

A serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time. The key feature of an interface circuit for a serial port is that it is capable of communicating in a bit-serial fashion on the device side and in a bit-parallel fashion on the bus side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability. A block diagram of a typical serial interface is shown in Figure 4.37. It includes the familiar DATAIN and DATAOUT registers. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are loaded into the output shift register, from which the bits are shifted out and sent to the I/O device.

The part of the interface that deals with the bus is the same as in the parallel interface described earlier. The status flags SIN and SOUT serve similar functions. The SIN flag is set to 1 when new data are loaded in DATAIN; it is cleared to 0 when the processor reads the contents of DATAIN. As soon as the data are transferred from the input shift register into the DATAIN register, the shift register can start accepting the next 8-bit character from the I/O device. The SOUT flag indicates whether the output buffer is available. It is cleared to 0 when the processor writes new data into the DATAOUT

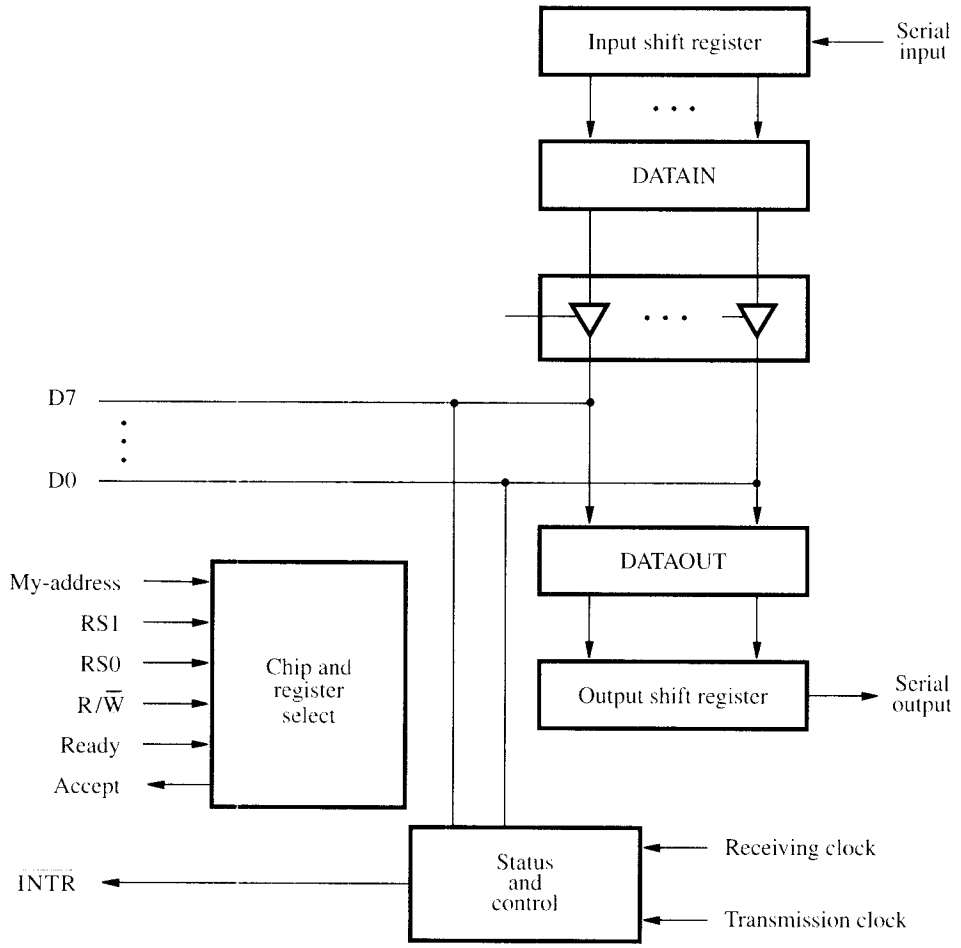


Figure 4.37 A serial interface.

register and set to 1 when data are transferred from DATAOUT into the output shift register.

The double buffering used in the input and output paths is important. A simpler interface could be implemented by turning DATAIN and DATAOUT into shift registers and eliminating the shift registers in Figure 4.37. However, this would impose awkward restrictions on the operation of the I/O device; after receiving one character from the serial line, the device cannot start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to allow the processor to read the input data. With the double buffer, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the DATAIN register. Thus, provided the processor reads the contents of DATAIN before the serial transfer of the second character is completed, the interface can receive