FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

COMPUTER ORGANIZATION
AND EMBEDDED SYSTEMS

SIXTH EDITION

Carl Hamacher
Queen's University

Zvonko Vranesic
University of Toronto

Safwat Zaky
University of Toronto

Naraig Manjikian
Queen's University

chapter

3

Basic Input/Output

Chapter Objectives

In this chapter you will learn about:

- Transferring data between a processor and input/output (I/O) devices
- The programmer's view of I/O transfers
- How program-controlled I/O is performed using polling
- How interrupts are used in I/O transfers

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 sent to a speaker, or a digitally coded command that changes the speed of a motor, opens a valve, or causes a robot to move in a specified manner. In short, computers should have the ability to exchange digital and analog information with a wide range of devices in many different environments.

In this chapter we will consider the input/output (I/O) capability of computers as seen from the programmer's point of view. We will present only basic I/O operations, which are provided in all computers. This knowledge will enable the reader to perform interesting and useful exercises on equipment found in a typical teaching laboratory environment. More complex I/O schemes, as well as the hardware needed to implement the I/O capability, are discussed in Chapter 7.

3.1 ACCESSING I/O DEVICES

The components of a computer system communicate with each other through an interconnection network, as shown in Figure 3.1. The interconnection network consists of circuits needed to transfer information between the processor, the memory unit, and a number of I/O devices.

In Chapter 2, we described the concept of an address space and how the processor may access individual memory locations within such an address space. Load and Store instructions use addressing modes to generate effective addresses that identify the desired locations. This idea of using addresses to access various locations in the memory can be

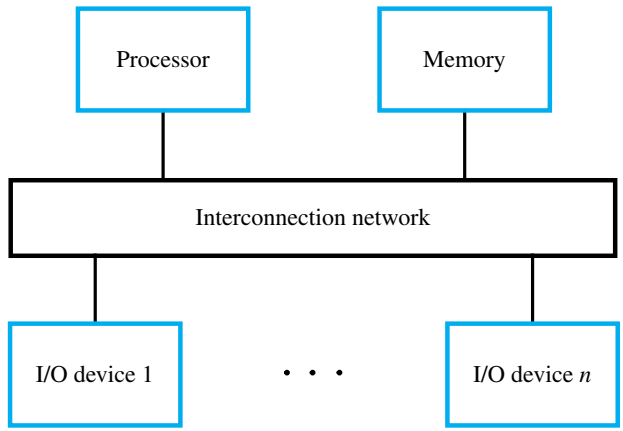


Figure 3.1 A computer system.

extended to deal with the I/O devices as well. For this purpose, each I/O device must appear to the processor as consisting of some addressable locations, just like the memory. Some addresses in the address space of the processor are assigned to these I/O locations, rather than to the main memory. These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of registers. It is customary to refer to them as *I/O registers*. Since the I/O devices and the memory share the same address space, this arrangement is called *memory-mapped I/O*. It is used in most computers.

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 of a register in an input device, the instruction

```
Load     R2, DATAIN
```

reads the data from the DATAIN register and loads them into processor register R2. Similarly, the instruction

```
Store    R2, DATAOUT
```

sends the contents of register R2 to location DATAOUT, which is a register in an output device.

3.1.1 I/O DEVICE INTERFACE

An I/O device is connected to the interconnection network by using a circuit, called the *device interface*, which provides the means for data transfer and for the exchange of status and control information needed to facilitate the data transfers and govern the operation of the device. The interface includes some registers that can be accessed by the processor. One register may serve as a buffer for data transfers, another may hold information about the current status of the device, and yet another may store the information that controls the operational behavior of the device. These *data*, *status*, and *control* registers are accessed by program instructions as if they were memory locations. Typical transfers of information are between I/O registers and the registers in the processor. Figure 3.2 illustrates how the keyboard and display devices are connected to the processor from the software point of view.

3.1.2 PROGRAM-CONTROLLED I/O

Let us begin the discussion of input/output issues by looking at two essential I/O devices for human-computer interaction—keyboard and display. Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen. A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action. This method is known as *program-controlled I/O*.

In addition to transferring each character from the keyboard into the memory, and then to the display, it is necessary to ensure that this happens at the right time. An input character must be read in response to a key being pressed. For output, a character must be sent to

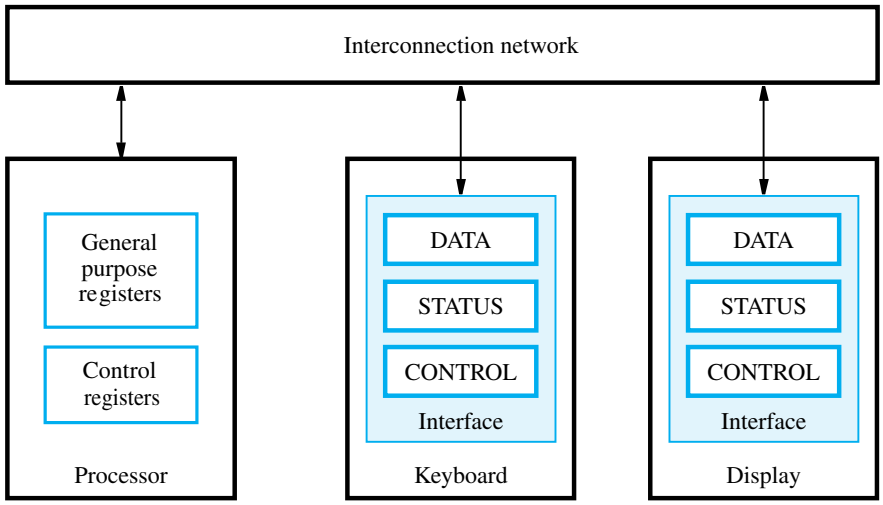


Figure 3.2 The connection for processor, keyboard, and display.

the display only when the display device is able to accept it. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted to and displayed on the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can execute billions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

One solution to this problem involves a signaling protocol. On output, the processor sends the first character and then waits for a signal from the display that the next character can be sent. It then sends the second character, and so on. An input character is obtained from the keyboard in a similar way. The processor waits for a signal indicating that a key has been pressed and that a binary code that represents the corresponding character is available in an I/O register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard includes a circuit that responds to a key being pressed by producing the code for the corresponding character that can be used by the computer. We will assume that ASCII code (presented in Table 1.1) is used, in which each character code occupies one byte. Let KBD_DATA be the address label of an 8-bit register that holds the generated character. Also, let a signal indicating that a key has been pressed be provided by setting to 1 a flip-flop called KIN, which is a part of an eight-bit status register, KBD_STATUS. The processor can read the *status flag* KIN to determine when a character code has been placed in KBD_DATA. When the processor reads the status flag to determine its state, we say that the processor *polls* the I/O device.

The display includes an 8-bit register, which we will call DISP_DATA, used to receive characters from the processor. It also must be able to indicate that it is ready to receive the

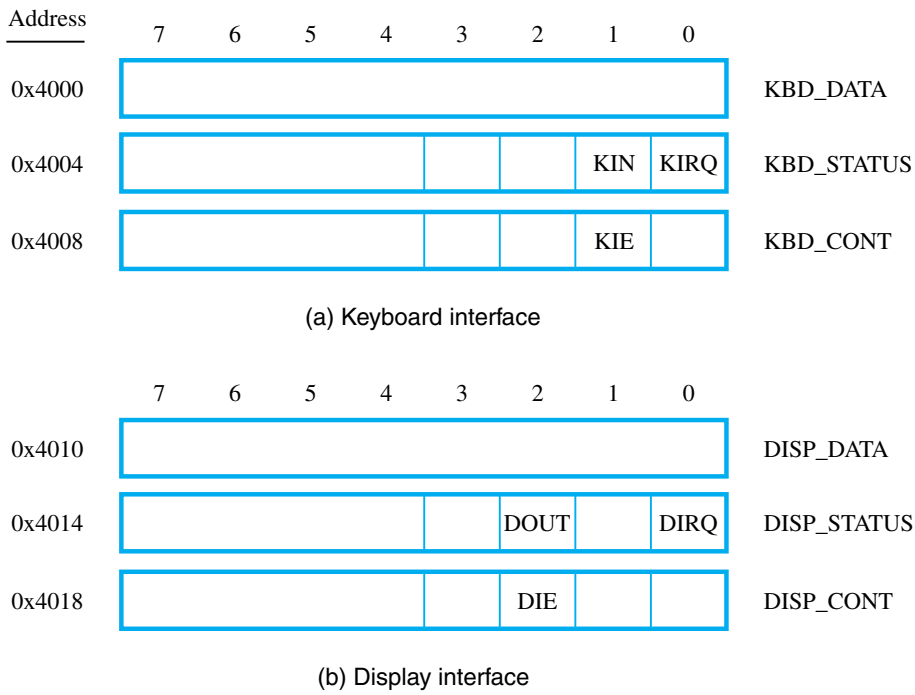


Figure 3.3 Registers in the keyboard and display interfaces.

next character; this can be done by using a status flag called DOUT, which is one bit in a status register, DISP_STATUS.

Figure 3.3 illustrates how these registers may be organized. The interface for each device also includes a control register, which we will discuss in Section 3.2. We have identified only a few bits in the registers, those that are pertinent to the discussion in this chapter. Other bits can be used for other purposes, or perhaps simply ignored.

If the registers in I/O interfaces are to be accessed as if they are memory locations, each register must be assigned a specific address that will be recognized by the interface circuit. In Figure 3.3, we assigned hexadecimal numbers 4000 and 4010 as base addresses for the keyboard and display, respectively. These are the addresses of the data registers. The addresses of the status registers are four bytes higher, and the control registers are eight bytes higher. This makes all addresses word-aligned in a 32-bit word computer, which is usually done in practice. Assigning the addresses to registers in this manner makes the I/O registers accessible in a program executed by the processor. This is the programmer's view of the device.

A program is needed to perform the task of reading the characters produced by the keyboard, storing these characters in the memory, and sending them to the display. To perform I/O transfers, the processor must execute machine instructions that check the state of the status flags and transfer data between the processor and the I/O devices.

Let us consider the details of the input process. When a key is pressed, the keyboard circuit places the ASCII-encoded character into the KBD_DATA register. At the same time, the circuit sets the KIN flag to 1. Meanwhile, the processor is executing the I/O program which continuously checks the state of the KIN flag. When it detects that KIN is set to 1, it transfers the contents of KBD_DATA into a processor register. Once the contents of KBD_DATA are read, KIN must be cleared to 0, which is usually done automatically by the interface circuit. If a second character is entered at the keyboard, KIN is again set to 1 and the process repeats. The desired action can be achieved by performing the operations:

```
READWAIT    Read the KIN flag
            Branch to READWAIT if KIN = 0
            Transfer data from KBD_DATA to R5
```

which reads the character into processor register R5.

An analogous process takes place when characters are transferred from the processor to the display. When DOUT is equal to 1, the display is ready to receive a character. Under program control, the processor monitors DOUT, and when DOUT is equal to 1, the processor transfers an ASCII-encoded character to DISP_DATA. The transfer of a character to DISP_DATA clears DOUT to 0. When the display device is ready to receive a second character, DOUT is again set to 1. This can be achieved by performing the operations:

```
WRITEWAIT   Read the DOUT flag
            Branch to WRITEWAIT if DOUT = 0
            Transfer data from R5 to DISP_DATA
```

The wait loop is executed repeatedly until the status flag DOUT is set to 1 by the display when it is free to receive a character. Then, the character from R5 is transferred to DISP_DATA to be displayed, which also clears DOUT to 0.

We assume that the initial state of KIN is 0 and the initial state of DOUT is 1. This initialization is normally performed by the device control circuits when power is turned on.

In computers that use memory-mapped I/O, in which some addresses are used to refer to registers in I/O interfaces, data can be transferred between these registers and the processor using instructions such as Load, Store, and Move. For example, the contents of the keyboard character buffer KBD_DATA can be transferred to register R5 in the processor by the instruction

```
LoadByte    R5, KBD_DATA
```

Similarly, the contents of register R5 can be transferred to DISP_DATA by the instruction

```
StoreByte   R5, DISP_DATA
```

The LoadByte and StoreByte operation codes signify that the operand size is a byte, to distinguish them from the Load and Store operation codes that we have used for word operands.

The Read operation described above may be implemented by the RISC-style instructions:

```
READWAIT:   LoadByte          R4, KBD_STATUS
            And               R4, R4, #2
            Branch_if_[R4]=0  READWAIT
            LoadByte          R5, KBD_DATA
```

The And instruction is used to test the KIN flag, which is bit b_1 of the status information in R4 that was read from the KBD_STATUS register. As long as $b_1 = 0$, the result of the AND operation leaves the value in R4 equal to zero, and the READWAIT loop continues to be executed.

Similarly, the Write operation may be implemented as:

```
WRITEWAIT:  LoadByte          R4, DISP_STATUS
            And               R4, R4, #4
            Branch_if_[R4]=0  WRITEWAIT
            StoreByte         R5, DISP_DATA
```

Observe that the And instruction in this case uses the immediate value 4 to test the display's status bit, b_2.

3.1.3 AN EXAMPLE OF A RISC-STYLE I/O PROGRAM

We can now put together a complete program for a typical I/O task, as shown in Figure 3.4. The program uses the program-controlled I/O approach described above to read, store, and display a line of characters typed at the keyboard. As the characters are read in, one by one, they are stored in the memory and then *echoed* back to the display. The program finishes when the carriage return character, CR, is encountered. The address of the first byte location of the memory where the line is to be stored is LOC. Register R2 is used to point to this part of the memory, and it is initially loaded with the address LOC by the first instruction in the program. R2 is incremented for each character read and displayed.

3.1.4 AN EXAMPLE OF A CISC-STYLE I/O PROGRAM

Let us now perform the same task using CISC-style instructions. In CISC instruction sets it is possible to perform some arithmetic and logic operations directly on operands in the memory. So, it is possible to have the instruction

```
TestBit     destination, #k
```

which tests bit b_k of the destination operand and sets the condition flag Z (Zero) to 1 if $b_k = 0$ and to 0 otherwise. Since the operand can be in a memory location, we can use the instruction

```
TestBit     KBD_STATUS, #1
```

```
        Move            R2, #LOC        Initialize pointer register R2 to point to the
                                          address of the first location in main memory
                                          where the characters are to be stored.
        MoveByte        R3, #CR         Load ASCII code for Carriage Return into R3.
READ:   LoadByte        R4, KBD_STATUS  Wait for a character to be entered.
        And             R4, R4, #2      Check the KIN flag.
        Branch_if_[R4]=0 READ
        LoadByte        R5, KBD_DATA    Read the character from KBD_DATA
                                          (this clears KIN to 0).
        StoreByte       R5, (R2)        Write the character into the main memory and
        Add             R2, R2, #1        increment the pointer to main memory.
ECHO:   LoadByte        R4, DISP_STATUS Wait for the display to become ready.
        And             R4, R4, #4      Check the DOUT flag.
        Branch_if_[R4]=0 ECHO
        StoreByte       R5, DISP_DATA   Move the character just read to the display
                                          buffer register (this clears DOUT to 0).
        Branch_if_[R5]≠[R3] READ        Check if the character just read is the
                                          Carriage Return. If it is not, then
                                          branch back and read another character.
```

Figure 3.4 A RISC-style program that reads a line of characters and displays it.

to test the state of the KIN flag in the keyboard interface. A Branch instruction that checks the state of the Z flag can then be used to cause a branch to the beginning of the wait loop.

Figure 3.5 gives a CISC-style program that reads and displays a line of characters. Observe that the first MoveByte instruction transfers each character directly from KBD_DATA to the memory location pointed to by R2. A Compare instruction

```
Compare    destination, source
```

performs the comparison by subtracting the contents of the source from the contents of the destination, and then sets the condition flags based on the result. It does not change the contents of either the source or the destination. Note that the CompareByte instruction in Figure 3.5 uses the autoincrement addressing mode, which automatically increments the value of the pointer R2 after the comparison has been made. In the RISC-style program in Figure 3.4 the pointer has to be incremented using a separate Add instruction.

We have discussed the memory-mapped I/O scheme, which is used in most computers. There is an alternative that can be found in some processors where there exist special In and Out instructions to perform I/O transfers. In this case, there exists a separate I/O address space used only by these instructions. When building a computer system that uses 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.

	Move	R2, #LOC	Initialize pointer register R2 to point to the address of the first location in main memory where the characters are to be stored.
READ:	TestBit	KBD_STATUS, #1	Wait for a character to be entered
	Branch=0	READ	in the keyboard buffer KBD_DATA.
	MoveByte	(R2), KBD_DATA	Transfer the character from KBD_DATA into the main memory (this clears KIN to 0).
ECHO:	TestBit	DISP_STATUS, #2	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	DISP_DATA, (R2)	Move the character just read to the display buffer register (this clears DOUT to 0).
	CompareByte	(R2)+, #CR	Check if the character just read is CR (carriage return). If it is not CR, then
	Branch≠0	READ	branch back and read another character. Also, increment the pointer to store the next character.

Figure 3.5 A CISC-style program that reads a line of characters and displays it.

Program-controlled I/O requires continuous involvement of the processor in the I/O activities. Almost all of the execution time for the programs in Figures 3.4 and 3.5 is spent in the two wait loops, while the processor waits for a key to be pressed or for the display to become available. Wasting the processor execution time in this manner can be avoided by using the concept of interrupts.

3.2 INTERRUPTS

In the examples in Figures 3.4 and 3.5, 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 request* to the processor. Since the processor is no longer required to continuously poll the status of I/O devices, it can use the waiting period to perform other useful tasks. Indeed, by using interrupts, such waiting periods can ideally be eliminated.

Example 3.1

Consider a task that requires continuous extensive computations to be performed and the results to be displayed on a display device. The displayed results must be updated every ten seconds. The ten-second intervals can be determined by a simple timer circuit, which

generates an appropriate signal. The processor treats the timer circuit as an input device that produces a signal that can be interrogated. If this is done by means of polling, the processor will waste considerable time checking the state of the signal. A better solution is to have the timer circuit raise an interrupt request once every ten seconds. In response, the processor displays the latest results.

The task can be implemented with a program that consists of two routines, COMPUTE and DISPLAY. The processor continuously executes the COMPUTE routine. When it receives an interrupt request from the timer, it suspends the execution of the COMPUTE routine and executes the DISPLAY routine which sends the latest results to the display device. Upon completion of the DISPLAY routine, the processor resumes the execution of the COMPUTE routine. Since the time needed to send the results to the display device is very small compared to the ten-second interval, the processor in effect spends almost all of its time executing the COMPUTE routine.

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 DISPLAY routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in Figure 3.6. The 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 returns 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

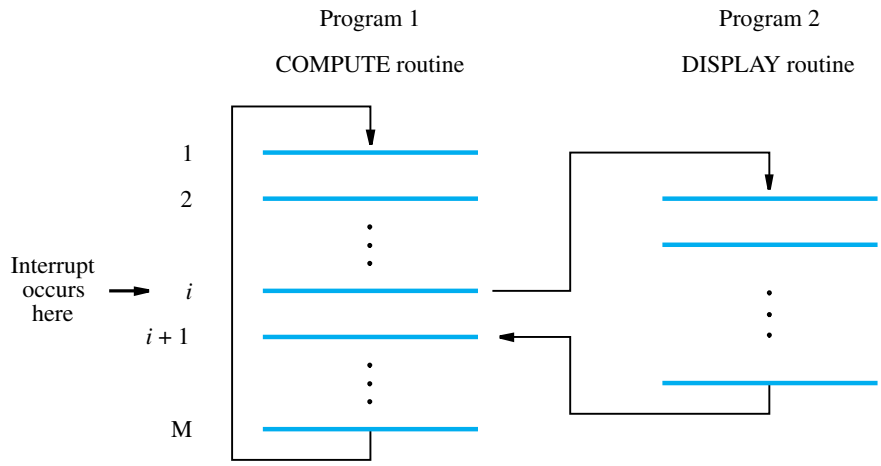


Figure 3.6 Transfer of control through the use of interrupts.

instruction $i + 1$. The *return address* must be saved either in a designated general-purpose register or on the processor stack.

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 can be accomplished by means of a special control signal, called *interrupt acknowledge*, which is sent to the device through the interconnection network. An 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 the 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. As such, potential changes to status information and contents of registers are anticipated. However, an interrupt-service routine may not have any relation to the portion of the program being executed at the time the interrupt request is received. Therefore, before starting execution of the interrupt-service routine, status information and contents of processor registers that may be altered in unanticipated ways during the execution of that routine must be saved. This saved 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 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 explicit 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. The duplicate registers are sometimes called the *shadow 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*.

3.2.1 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.

There are many situations in which the processor should ignore interrupt requests. For instance, the timer circuit in Example 3.1 should raise interrupt requests only when the COMPUTE routine is being executed. It should be prevented from doing so when some other task is being performed. 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.

It is convenient to be able to enable and disable interrupts at both the processor and I/O device ends. The processor can either accept or ignore interrupt requests. An I/O device can either be allowed to raise interrupt requests or prevented from doing so. A commonly used mechanism to achieve this is to use some control bits in registers that can be accessed by program instructions.

The processor has a *status register* (PS), which contains information about its current state of operation. Let one bit, IE, of this register be assigned for enabling/disabling interrupts. Then, the programmer can set or clear IE to cause the desired action. When IE = 1, interrupt requests from I/O devices are accepted and serviced by the processor. When IE = 0, the processor simply ignores all interrupt requests from I/O devices.

The interface of an I/O device includes a control register that contains the information that governs the mode of operation of the device. One bit in this register may be dedicated to interrupt control. The I/O device is allowed to raise interrupt requests only when this bit is set to 1. We will discuss this arrangement in Section 3.2.3.

Let us now consider 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.

A good choice is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. The processor saves the contents of the program counter and the processor status register. After saving the contents of the PS register, with the IE bit equal to 1, the processor clears the IE bit in the PS register, thus disabling further interrupts. Then, it begins execution of the interrupt-service routine. When a Return-from-interrupt instruction is executed, the saved contents of the PS register are restored, setting the IE bit back to 1. Hence, interrupts are again enabled.

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 in both the processor and the device, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed and saves the contents of the PC and PS registers.
3. Interrupts are disabled by clearing the IE bit in the PS to 0.
4. The action requested by the interrupt is performed by the interrupt-service routine, during which time the device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. Upon completion of the interrupt-service routine, the saved contents of the PC and PS registers are restored (enabling interrupts by setting the IE bit to 1), and execution of the interrupted program is resumed.

3.2.2 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 determine which device is 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 issues are handled 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 an interrupt request is received it is necessary to identify the particular device that raised the request. Furthermore, if two devices raise interrupt requests 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 the device raises an interrupt request, it sets to 1 a bit in its status register, which we will call the IRQ bit. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all I/O devices in the system. The first device encountered with its IRQ bit set to 1 is the device that should be serviced. An appropriate subroutine is then 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 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 interrupt-handling schemes based on this approach.

A device requesting an interrupt can identify itself if it has its own interrupt-request signal, or if it can send a special code to the processor through the interconnection network. The processor's circuits determine the memory address of the required interrupt-service routine. A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines. These addresses are usually referred to as *interrupt vectors*, and they are said to constitute the *interrupt-vector table*. For example, 128 bytes may be allocated to hold a table of 32 interrupt vectors. Typically, the interrupt-vector table is in the lowest-address range. The interrupt-service routines may be located anywhere in the memory. When an interrupt request arrives, the information provided by the requesting device is used as a pointer into the interrupt-vector table, and the address in the corresponding interrupt vector is automatically loaded into the program counter.

Interrupt Nesting

We suggested in Section 3.2.1 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, i.e., to nest interrupts.

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 a 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 that execution of an interrupt-service routine for some device is started, the priority of the processor is raised to that of the device either automatically or with special instructions. This action disables interrupts from devices that have the same or lower level of priority. However, interrupt requests from higher-priority devices will continue to be accepted. The processor's priority can be encoded in a few bits of the processor status register. While this scheme is used in some processors, we will use a simpler scheme in later examples.

Finally, we should point out that if nested interrupts are allowed, then each interrupt-service routine must save on the stack the saved contents of the program counter and the status register. This has to be done before the interrupt-service routine enables nesting by setting the IE bit in the staus register to 1.

Simultaneous Requests

We also need to 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. 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. This is done in hardware, by using arbitration circuits which we will discuss in Chapter 7.

3.2.3 CONTROLLING I/O DEVICE BEHAVIOR

It is important to ensure that interrupt requests are generated only by those I/O devices that the processor is currently willing to recognize. Hence, we need a mechanism in the interface circuits of individual devices to control whether a device is allowed to interrupt the processor. The control needed is usually provided in the form of an *interrupt-enable* bit in the device's interface circuit.

I/O devices vary in complexity from simple to quite complex. Simple devices, such as a keyboard, require little in the way of control. Complex devices may have a number of possible modes of operation, which must be controlled. A commonly used approach is to provide a control register in the device interface, which holds the information needed to control the behavior of the device. This register is accessed as an addressable location, just

like the data and status registers that we discussed before. One bit in the register serves as the interrupt-enable bit, IE. When it is set to 1 by an instruction that writes new information into the control register, the device is placed into a mode in which it is allowed to interrupt the processor whenever it is ready for an I/O transfer.

Figure 3.3 shows the registers that may be used in the interfaces of keyboard and display devices. Since these devices transfer character-based data, handling one character at a time, it is appropriate to use an eight-bit data register. We have assumed that the status and control registers are also eight bits long. Only one or two bits in these registers are needed in handling the I/O transfers. The remaining bits can be used to specify other aspects of the operation of the device, or ignored if they are not needed. The keyboard status register includes bits KIN and KIRQ. We have already discussed the use of the KIN bit in Section 3.1.2. The KIRQ bit is set to 1 if an interrupt request has been raised, but not yet serviced. The keyboard may raise interrupt requests only when the interrupt-enable bit, KIE, in its control register is set to 1. Thus, when both KIE and KIN bits are equal to 1, an interrupt request is raised and the KIRQ bit is set to 1. Similarly, the DIRQ bit in the status register of the display interface indicates whether an interrupt request has been raised. Bit DIE in the control register of this interface is used to enable interrupts. Observe that we have placed KIN and KIE in bit position 1, and DOUT and DIE in position 2. This is an arbitrary choice that makes the program examples that follow easier to understand.

3.2.4 PROCESSOR CONTROL REGISTERS

We have already discussed the need for a status register in the processor. To deal with interrupts it is useful to have some other control registers. Figure 3.7 depicts one possibility, where there are four processor control registers. The status register, PS, includes the interrupt-enable bit, IE, in addition to other status information. Recall that the processor will accept interrupts only when this bit is set to 1. The IPS register is used to automatically

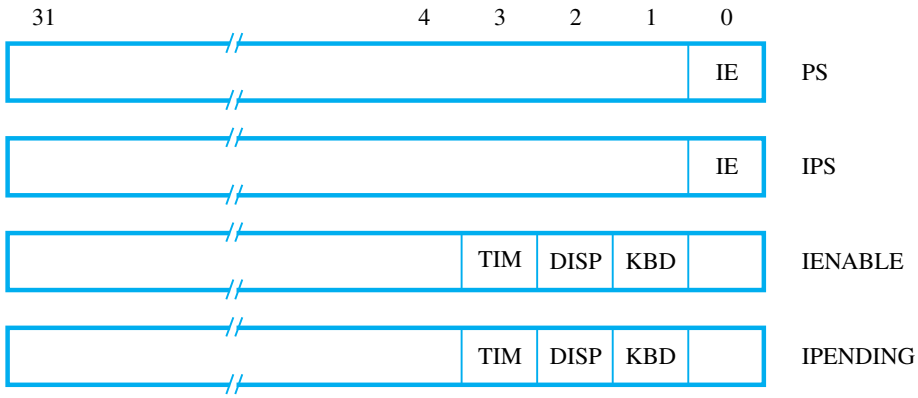


Figure 3.7 Control registers in the processor.

save the contents of PS when an interrupt request is received and accepted. At the end of the interrupt-service routine, the previous state of the processor is automatically restored by transferring the contents of IPS into PS. Since there is only one register available for storing the previous status information, it becomes necessary to save the contents of IPS on the stack if nested interrupts are allowed.

The IENABLE register allows the processor to selectively respond to individual I/O devices. A bit may be assigned for each device, as shown in the figure for the keyboard, display, and a timer circuit that we will use in a later example. When a bit is set to 1, the processor will accept interrupt requests from the corresponding device. The IPENDING register indicates the active interrupt requests. This is convenient when multiple devices may raise requests at the same time. Then, a program can decide which interrupt should be serviced first.

In a 32-bit processor, the control registers are 32 bits long. Using the structure in Figure 3.7, it is possible to accommodate 32 I/O devices in a straightforward manner.

Assembly-language instructions can refer to processor control registers by using names such as those in Figure 3.7. But, these registers cannot be accessed in the same way as the general-purpose registers. They cannot be accessed by arithmetic and logic instructions. They also cannot be accessed by Load and Store instructions that use the encoding format depicted in Figure 2.32*c*, because a five-bit field is used to specify a source or a destination register in these instructions, which makes it possible to specify only 32 general-purpose registers. Special instructions or special addressing modes may be provided to access the processor control registers. In a RISC-style processor, the special instructions may be of the type

```
MoveControl     R2, PS
```

which loads the contents of the program status register into register R2, and

```
MoveControl     IENABLE, R3
```

which places the contents of R3 into the IENABLE register. These instructions perform transfers between control and general-purpose registers.

3.2.5 EXAMPLES OF INTERRUPT PROGRAMS

Having presented the basic aspects of interrupts, we can now give some illustrative examples. We will use the keyboard and display devices with the register structure given in Figure 3.3.

Example 3.2

Let us consider again the task of reading a line of characters typed on a keyboard, storing the characters in the main memory, and displaying them on a display device. In Figures 3.4 and 3.5, we showed how this task may be performed by using the polling approach to detect when the I/O devices are ready for data transfer. Now, we will use interrupts with the keyboard, but polling with the display.

We assume for now that a specific memory location, ILOC, is dedicated for dealing with interrupts, and that it contains the first instruction of the interrupt-service routine. Whenever an interrupt request arrives at the processor, and processor interrupts are enabled, the processor will automatically:

- Save the contents of the program counter, either in a processor register that holds the return address or on the processor stack.
- Save the contents of the status register PS by transferring them into the IPS register, and clear the IE bit in the PS.
- Load the address ILOC into the program counter.

Assume that in the Main program we wish to read a line from the keyboard and store the characters in successive byte locations in the memory, starting at location LINE. Also, assume that the interrupt-service routine has been loaded in the memory, starting at location ILOC. The Main program has to initialize the interrupt process as follows:

1. Load the address LINE into a memory location PNTR. The interrupt-service routine will use this location as a pointer to store the input characters in the memory.
2. Enable interrupts in the keyboard interface by setting to 1 the KIE bit in the KBD_CONT register.
3. Enable the processor to accept interrupts from the keyboard by setting to 1 the KBD bit in its control register IENABLE.
4. Enable the processor to respond to interrupts in general 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 must 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. Display the character using the polling approach.
4. When the end of the line is reached, disable keyboard interrupts and inform the Main program.
5. Return from interrupt.

A RISC-style program that performs these tasks is shown in Figure 3.8. The comments in the program explain the relevant details. When the end of the input line is detected, the interrupt-service routine clears the KIE bit in register KBD_CONT, as no further input is expected. It also sets to 1 the variable EOL (End Of Line), which was initially cleared to 0. We assume that it is checked periodically by the Main program to determine when the input line is ready for processing. The EOL variable provides a means of signaling between the Main program and the interrupt-service routine.

Interrupt-service routine

```
ILOC:   Subtract              SP, SP, #8          Save registers.
        Store                 R2, 4(SP)
        Store                 R3, (SP)
        Load                  R2, PNTR            Load address pointer.
        LoadByte              R3, KBD_DATA        Read character from keyboard.
        StoreByte             R3, (R2)            Write the character into memory
        Add                   R2, R2, #1            and increment the pointer.
        Store                 R2, PNTR            Update the pointer in memory.
ECHO:   LoadByte              R2, DISP_STATUS     Wait for display to become ready.
        And                   R2, R2, #4
        Branch_if_[R2]=0      ECHO
        StoreByte             R3, DISP_DATA       Display the character just read.
        Move                  R2, #CR             ASCII code for Carriage Return.
        Branch_if_[R3]≠[R2]   RTRN                Return if not CR.
        Move                  R2, #1
        Store                 R2, EOL             Indicate end of line.
        Clear                 R2                  Disable interrupts in
        StoreByte             R2, KBD_CONT          the keyboard interface.
RTRN:   Load                  R3, (SP)            Restore registers.
        Load                  R2, 4(SP)
        Add                   SP, SP, #8
        Return-from-interrupt
```

Main program

```
START:  Move                  R2, #LINE
        Store                 R2, PNTR            Initialize buffer pointer.
        Clear                 R2
        Store                 R2, EOL             Clear end-of-line indicator.
        Move                  R2, #2              Enable interrupts in
        StoreByte             R2, KBD_CONT          the keyboard interface.
        MoveControl           R2, IENABLE
        Or                    R2, R2, #2          Enable keyboard interrupts in
        MoveControl           IENABLE, R2           the processor control register.
        MoveControl           R2, PS
        Or                    R2, R2, #1
        MoveControl           PS, R2              Set interrupt-enable bit in PS.
        next instruction
```

Figure 3.8 A RISC-style program that reads a line of characters using interrupts, and displays the line using polling.

Observe that the last three instructions in the Main program are used to set to 1 the interrupt-enable bit in PS. Since only MoveControl instructions can access the contents of a control register, the contents of PS are loaded into a general-purpose register, R2, modified and then written back into PS. Using the Or instruction to modify the contents affects only the IE bit and leaves the rest of the bits in PS unchanged.

When multiple I/O devices raise interrupt requests, it is necessary to determine which device has requested an interrupt. This can be done in software by checking the information in the IPENDING control register and choosing the interrupt-service routine that should be executed.

Example 3.3 In Example 3.2, we used interrupts with the keyboard only. The display device can also use interrupts. Suppose a program needs to display a page of text stored in the memory. This can be done by having the processor send a character whenever the display interface is ready, which may be indicated by an interrupt request. Assume that both the display and the keyboard are used by this program, and that both are enabled to raise interrupt requests. Using the register structure in Figures 3.3 and 3.7, the initialization of interrupts and the processing of requests can be done as indicated in Figure 3.9.

The Main program must initialize any variables needed by the interrupt-service routines, such as the memory buffer pointers. Then, it enables interrupts in both the keyboard and display interfaces. Next, it enables interrupts in the processor control register IENABLE. Note that the immediate value 6, which is loaded into this register, sets bits KBD and DISP to 1. Finally, the processor is enabled to respond to interrupts in general by setting to 1 the IE bit in the processor status register, PS.

Again, we assume that whenever an interrupt request arrives, the processor will automatically save the contents of the program counter (PC) and then load the address ILOC into PC. It will also save the contents of the status register (PS) by transferring them into the IPS register, and disable interrupts. Unlike Example 3.2, where we assumed that there is only one device that can raise interrupt requests, now we cannot go directly to the desired interrupt-service routine. First, it is necessary to identify the interrupting device. The needed information is found in the processor control register IPENDING. Since the interrupt-service routine uses registers R2 and R3 in this process, the contents of these registers must be saved on the stack and later restored. It is also necessary to save the contents of the subroutine linkage register, LINK_reg, because an interrupt can occur while some subroutine is being executed and the interrupt-service routine calls a subroutine. The circuit that detects interrupts sets to 1 the appropriate bit in IPENDING for each pending request. In Figure 3.9, the contents of IPENDING are loaded into general purpose register R2, and then examined to determine which interrupts are pending. If the display has a pending interrupt, then its interrupt-service routine is executed. If not, then a check is made for the keyboard. This may be followed by checking any other devices that could have pending requests. The order in which the bits in IPENDING are checked establishes a priority for the interrupting devices in case of simultaneous requests.

Interrupt handler

```
ILOC:       Subtract        SP, SP, #12         Save registers.
            Store           LINK_reg, 8(SP)
            Store           R2, 4(SP)
            Store           R3, (SP)
            MoveControl     R2, IPENDING        Check contents of IPENDING.
            And             R3, R2, #4          Check if display raised the request.
            Branch_if_[R3]=0 TESTKBD            If not, check if keyboard.
            Call            DISR                Call the display ISR.
TESTKBD:    And             R3, R2, #2          Check if keyboard raised the request.
            Branch_if_[R3]=0 NEXT               If not, then check next device.
            Call            KISR                Call the keyboard ISR.
NEXT:       ...                                 Check for other interrupts.

            Load            R3, (SP)            Restore registers.
            Load            R2, 4(SP)
            Load            LINK_reg, 8(SP)
            Add             SP, SP, #12
            Return-from-interrupt
```

Main program

```
START:      ...                                 Set up parameters for ISRs.
            Move            R2, #2              Enable interrupts in
            StoreByte       R2, KBD_CONT          the keyboard interface.
            Move            R2, #4              Enable interrupts in
            StoreByte       R2, DISP_CONT         the display interface.
            MoveControl     R2, IENABLE
            Or              R2, R2, #6          Enable interrupts in
            MoveControl     IENABLE, R2           the processor control register.
            MoveControl     R2, PS
            Or              R2, R2, #1
            MoveControl     PS, R2              Set interrupt-enable bit in PS.
            next instruction
```

Keyboard interrupt-service routine

```
KISR:       ...
            ⋮
            Return
```

Display interrupt-service routine

```
DISR:       ...
            ⋮
            Return
```

Figure 3.9 A RISC-style program that initializes and handles interrupts.

The program parts that handle interrupt requests and provide the corresponding service to the requesting devices are often referred to as the *interrupt handler*. Note that while the interrupt handler starts at the fixed address ILOC, the individual interrupt-service routines are just subroutines that can be placed anywhere in the memory.

In Figure 3.9, we used a software approach to determine the interrupting device. In processors that use vectored interrupts, the circuit that detects interrupt requests automatically loads a different address into the program counter for each interrupt that is assigned a specific location in the interrupt-vector table. A separate interrupt-service routine is executed to completion for each pending request, even if multiple interrupt requests are raised at the same time.

CISC-style Examples of Interrupts

The above tasks can be implemented using CISC-style instructions using the same basic approach. The main difference is that some operations, such as testing a bit in an I/O register, can be done directly. The tasks in Examples 3.2 and 3.3 can be realized using the programs in Figures 3.10 and 3.11, respectively. The TestBit instruction is used to test the status flags. The SetBit and ClearBit instructions are used to set an individual bit in an I/O register to 1 and 0, respectively. The comments in the programs provide explanations of how the desired tasks are realized.

Input/output operations in a computer system are usually much more involved than our simple examples suggest. As we will describe in Chapter 4, the operating system of the computer performs these operations on behalf of user programs. In Chapter 7, we will discuss in detail the hardware used in I/O operations.

3.2.6 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 events associated with 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 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

Interrupt-service routine

```
ILOC:   Move          –(SP), R2          Save register.
        Move          R2, PNTR           Load address pointer.
        MoveByte      (R2), KBD_DATA     Write the character into memory
        Add           PNTR, #1             and increment the pointer.
ECHO:   TestBit       DISP_STATUS, #2    Wait for the display to become ready.
        Branch=0      ECHO
        MoveByte      DISP_DATA, (R2)    Display the character just read.
        CompareByte   (R2), #CR          Check if the character just read is CR.
        Branch≠0      RTRN               Return if not CR.
        Move          EOL, #1            Indicate end of line.
        ClearBit      KBD_CONT, #1       Disable interrupts in keyboard interface.
RTRN:   Move          R2, (SP)+          Restore register.
        Return-from-interrupt
```

Main program

```
START:  Move          PNTR, #LINE        Initialize buffer pointer.
        Clear         EOL                Clear end-of-line indicator.
        SetBit        KBD_CONT, #1       Enable interrupts in keyboard interface.
        Move          R2, #2             Enable keyboard interrupts in
        MoveControl   IENABLE, R2          the processor control register.
        MoveControl   R2, PS
        Or            R2, #1
        MoveControl   PS, R2             Set interrupt-enable bit in PS.
        next instruction
```

Figure 3.10 A CISC-style program that reads a line of characters using interrupts, and displays the line using polling.

being executed and starts an exception-service routine, which 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, we assumed that the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error associated with the current instruction, that 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: trace mode and breakpoints. These facilities are described in detail in Chapter 4.

Interrupt handler

```
ILOC:       Move            –(SP), R2          Save registers.
            Move            –(SP), LINK_reg
            MoveControl     R2, IPENDING       Check contents of IPENDING.
            TestBit         R2, #2             Check if display raised the request.
            Branch=0        TESTKBD            If not, check if keyboard.
            Call            DISR               Call the display ISR.
TESTKBD:    TestBit         R2, #1             Check if keyboard raised the request.
            Branch=0        NEXT               If not, then check next device.
            Call            KISR               Call the keyboard ISR.
NEXT:       ...                                Check for other interrupts.

            Move            LINK_reg, (SP)+    Restore registers.
            Move            R2, (SP)+
            Return-from-interrupt
```

Main program

```
START:      ...                                Set up parameters for ISRs.
            SetBit          KBD_CONT, #1       Enable interrupts in keyboard interface.
            SetBit          DISP_CONT, #2      Enable interrupts in display interface.
            MoveControl     R2, IENABLE
            Or              R2, #6             Enable interrupts in
            MoveControl     IENABLE, R2          the processor control register.
            MoveControl     R2, PS
            Or              R2, #1
            MoveControl     PS, R2             Set interrupt-enable bit in PS.
            next instruction
```

Keyboard interrupt-service routine

```
KISR:       ...
            ⋮
            Return
```

Display interrupt-service routine

```
DISR:       ...
            ⋮
            Return
```

Figure 3.11 A CISC-style program that initializes and handles interrupts.

Use of Exceptions in Operating Systems

The operating system (OS) software coordinates the activities within a computer. It uses exceptions to communicate with and control the execution of user programs. It uses hardware interrupts to perform I/O operations. This topic is discussed in Chapter 4.

3.3 CONCLUDING REMARKS

In this chapter, we discussed two basic approaches to I/O transfers. The simplest technique is programmed I/O, in which the processor performs all of the necessary functions under direct control of program instructions. The second approach is based on the use of interrupts; this mechanism makes it possible to interrupt the normal execution of programs in order to service higher-priority requests that require more urgent attention. Although all computers have a mechanism for dealing with such situations, the complexity and sophistication of interrupt-handling schemes vary from one computer to another.

We dealt with the I/O issues from the programmer's point of view. In Chapter 7 we will consider the hardware aspects and some commonly used I/O standards.

3.4 SOLVED PROBLEMS

This section presents some examples of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 3.4

Problem: Assume that a memory location BINARY contains a 32-bit pattern. It is desired to display these bits as eight hexadecimal digits on a display device that has the interface depicted in Figure 3.3. Write a program that accomplishes this task.

Solution: First it is necessary to convert the 32-bit pattern into hex digits that are represented as ASCII-encoded characters. A simple way of doing the conversion is to use the *table-lookup* approach. A 16-entry table has to be constructed to provide the ASCII code for each possible hex digit. Then, for each four-bit segment of the pattern in BINARY, the corresponding character can be looked up in the table and stored in a block of memory bytes starting at location HEX. Finally, the eight characters starting at HEX are sent to the display.

Figures 3.12 and 3.13 give RISC- and CISC-style programs, respectively, for the required task. The comments describe the detailed actions taken.

```
        Load        R2, BINARY      Load the binary number.
        Move        R3, #8          R3 is a digit counter that is set to 8.
        Move        R4, #HEX        R4 points to the hex digits.
LOOP:   RotateL     R2, R2, #4      Rotate the high-order digit
                                      into low-order position.
        And         R5, R2, #0xF    Extract next digit.
        LoadByte    R6, TABLE(R5)   Get ASCII code for the digit and
        StoreByte   R6, (R4)          store it in HEX number location.
        Subtract    R3, R3, #1      Decrement the digit counter.
        Add         R4, R4, #1      Increment the pointer to hex digits.
        Branch_if_[R3]>0 LOOP       Loop back if not the last digit.
DISPLAY: Move       R3, #8
        Move        R4, #HEX
DLOOP:  LoadByte    R5, DISP_STATUS Wait for display to become ready.
        And         R5, R5, #4      Check the DOUT flag.
        Branch_if_[R5]=0 DLOOP
        LoadByte    R6, (R4)        Get the next ASCII character
        StoreByte   R6, DISP_DATA     and send it to the display.
        Subtract    R3, R3, #1      Decrement the counter.
        Add         R4, R4, #1      Increment the character pointer.
        Branch_if_[R3]>0 DLOOP      Loop until all characters displayed.
        next instruction

        ORIGIN      1000
HEX:    RESERVE     8               Space for ASCII-encoded digits.
TABLE:  DATABYTE    0x30,0x31,0x32,0x33 Table for conversion
        DATABYTE    0x34,0x35,0x36,0x37   to ASCII code.
        DATABYTE    0x38,0x39,0x41,0x42
        DATABYTE    0x43,0x44,0x45,0x46
```

Figure 3.12 A RISC-style program for Example 3.4.

Example 3.5 **Problem:** Consider the task described in Example 3.1. Assume that the timer circuit includes a 32-bit up/down counter driven by a 100-MHz clock. The counter can be set to count from a specified initial count value. The timer I/O interface is shown in Figure 3.14. It contains four registers.

- TIM_STATUS indicates the current status of the timer where:
  - The TON bit is set to 1 when the counter is running.
  - The ZERO bit is set to 1 when the counter reaches the count of zero.
  - The TIRQ bit is set to 1 when the timer raises an interrupt request, which happens when the counter contents reach zero and the timer interrupts are enabled.

```
            Move        R2, BINARY          Load the binary number.
            Move        R3, #8              R3 is a digit counter that is set to 8.
            Move        R4, #HEX            R4 points to the hex digits.
LOOP:       RotateL     R2, #4              Rotate the high-order digit
                                              into low-order position.
            Move        R5, R2
            And         R5, #0xF            Extract next digit.
            MoveByte    (R4)+, TABLE(R5)    Get ASCII code for the digit and
                                              store it in HEX number location.
            Subtract    R3, #1              Decrement the digit counter.
            Branch>0    LOOP                Loop back if not the last digit.
DISPLAY:    Move        R3, #8
            Move        R4, #HEX
DLOOP:      TestBit     DISP_STATUS, #2     Wait for display to become ready.
            Branch=0    DLOOP
            MoveByte    DISP_DATA, (R4)+    Send next character to display.
            Subtract    R3, #1              Decrement the counter.
            Branch>0    DLOOP               Loop until all characters displayed.
            next instruction

            ORIGIN      1000
HEX:        RESERVE     8                   Space for ASCII-encoded digits.
TABLE:      DATABYTE    0x30,0x31,0x32,0x33 Table for conversion
            DATABYTE    0x34,0x35,0x36,0x37   to ASCII code.
            DATABYTE    0x38,0x39,0x41,0x42
            DATABYTE    0x43,0x44,0x45,0x46
```

Figure 3.13 A CISC-style program for Example 3.4.

The action of reading the status register automatically clears the ZERO and TIRQ bits to 0.

- TIM_CONT controls the mode of operation, where:
  - The UP bit is set to 1 to cause the counter to count by incrementing its contents; when this bit is cleared to zero, the counter contents are decremented.
  - The FREE bit is set to 1 to cause a continuously running mode, where the counter is automatically reloaded with the initial count value whenever the actual count reaches zero.
  - The RUN bit is set to 1 to cause the counter to count; it is cleared to 0 to stop the counter.
  - The TIE bit is set to 1 to enable timer interrupts.
- TIM_INIT holds the initial count value.
- TIM_COUNT holds the current count value.

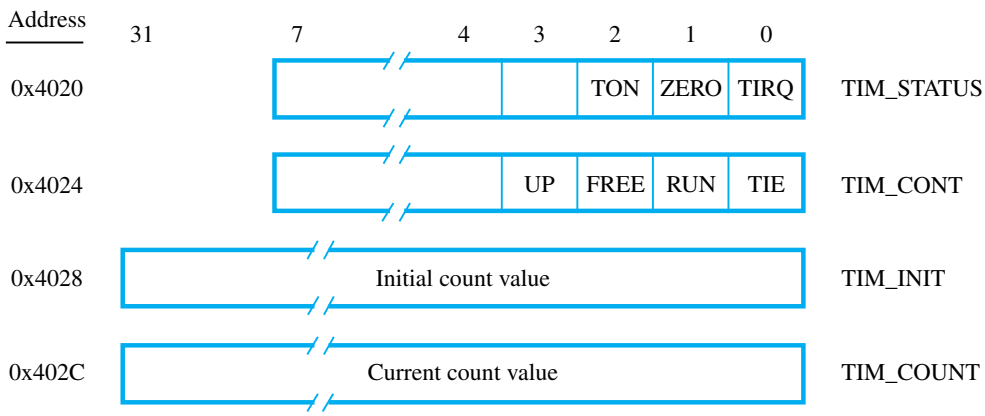


Figure 3.14 Registers in the timer interface.

Write a program to implement the desired task. Use the processor control registers depicted in Figure 3.7.

Solution: To obtain an interrupt request every ten seconds, it is necessary to count 10^9 clock cycles. This can be accomplished by writing this value into the TIM_INIT register, and then making the counter decrement its count and raise an interrupt when the count reaches zero. The value 10^9 can be represented by the hexadecimal number 3B9ACA00. To achieve the desired operation the FREE, RUN, and TIE bits must be set to 1, while the UP bit must be equal to 0.

Using the scheme outlined in Figure 3.9, we can implement the required task using a RISC-style program shown in Figure 3.15. Note that the initial count, which is a 32-bit immediate value, is loaded into R2 using the approach explained in Section 2.9.

Figure 3.16 gives a CISC-style program that uses the scheme outlined in Figure 3.11. In this case, the 32-bit immediate operand can be specified in a single instruction.

Example 3.6 **Problem:** A commonly used output device in digital systems is a seven-segment display, depicted in Figure 3.17. The device consists of seven independent segments which can be illuminated by applying electrical signals to them. Assume that each segment is illuminated when a logic value 1 is applied to it. The figure shows the bit patterns needed to display numbers 0 to 9.

Write a program that displays the number represented by an ASCII-encoded character stored in memory location DIGIT at address 0x800. Assume that the display has an I/O interface consisting of an eight-bit data register, SEVEN, where the segments *a* to *g* are connected to bits $SEVEN_{6-0}$. Let the bit $SEVEN_7$ be equal to 0. Also, assume that the address of register SEVEN is 0x4030. If the ASCII code in location DIGIT represents a

Interrupt handler

```
ILOC:   Subtract          SP, SP, #8          Save registers.
        Store             LINK_reg, 4(SP)
        Store             R2, (SP)
        MoveControl       R2, IPENDING        Check contents of IPENDING.
        And               R2, R2, #8          Check if request from timer.
        Branch_if_[R2]=0  NEXT
        LoadByte          R2, TIM_STATUS      Clear TIRQ and ZERO bits.
        Call              DISPLAY             Call the DISPLAY routine.
NEXT:   ...                                   Check for other interrupts.

        Load              R2, (SP)            Restore registers.
        Load              LINK_reg, 4(SP)
        Add               SP, SP, #8
        Return-from-interrupt
```

Main program

```
START:  ...                                   Set up parameters for ISRs.
        OrHigh            R2, R0, #0x3B9A     Prepare the initial
        Or                R2, R2, #0xCA00       count value.
        Store             R2, TIM_INIT        Set the initial count value.
        Move              R2, #7              Set the timer to free run
        StoreByte         R2, TIM_CONT          and enable interupts.
        MoveControl       R2, IENABLE
        Or                R2, R2, #8          Enable timer interrupts in
        MoveControl       IENABLE, R2           the processor control register.
        MoveControl       R2, PS
        Or                R2, R2, #1
        MoveControl       PS, R2              Set interrupt-enable bit in PS.
COMPUTE: next instruction
```

Figure 3.15 A RISC-style program for Example 3.5.

character that is not a number in the range 0 to 9, then the display should be blank, where all segments are turned off.

Solution: A look-up table can be used to hold the seven-segment bit patterns that correspond to the numbers 0 to 9. The ASCII-encoded digit is converted into a four-bit number that is used as an index into the table, by using the AND operation. Also, it is necessary to check that the high-order four bits of ASCII code are 0011. Note that all three addresses DIGIT, SEVEN, and TABLE can be represented in 16 bits.

Figures 3.18 and 3.19 give possible RISC- and CISC-style programs, respectively.

Interrupt handler

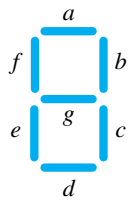
```
ILOC:   Move            –(SP), R2           Save registers.
        Move            –(SP), LINK_reg
        MoveControl     R2, IPENDING        Check contents of IPENDING.
        TestBit         R2, #3              Check if request from timer.
        Branch=0        NEXT
        MoveByte        R2, TIM_STATUS      Clear TIRQ and ZERO bits.
        Call            DISPLAY             Call the DISPLAY routine.
NEXT:   ···                                 Check for other interrupts.

        Move            LINK_reg, (SP)+     Restore registers.
        Move            R2, (SP)+
        Return-from-interrupt
```

Main program

```
START:  ···                                 Set up parameters for ISRs.
        Move            TIM_INIT, #0x3B9ACA00   Set the initial count value.
        MoveByte        TIM_CON, #7         Set the timer to free run
                                              and enable interupts.
        MoveControl     R2, IENABLE
        Or              R2, #8              Enable timer interrupts in
        MoveControl     IENABLE, R2           the processor control register.
        MoveControl     R2, PS
        Or              R2, #1
        MoveControl     PS, R2              Set interrupt-enable bit in PS.
COMPUTE: next instruction
```

Figure 3.16 A CISC-style program for Example 3.5.



Number	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Figure 3.17 Seven-segment display.

```
DIGIT   EQU                0x800          Location of ASCII-encoded digit.
SEVEN   EQU                0x4030         Address of 7-segment display.
        LoadByte           R2, DIGIT      Load the ASCII-encoded digit.
        And                R3, R2, #0xF0  Extract high-order bits of ASCII.
        And                R2, R2, #0x0F  Extract the decimal number.
        Move               R4, #0x30      Check if high-order bits of
        Branch_if_[R3]=[R4] HIGH3           ASCII code are 0011.
        Move               R2, #0x0F      Not a digit, display a blank.
HIGH3:  LoadByte           R5, TABLE(R2)  Get the 7-segment pattern.
        StoreByte          R5, SEVEN      Display the digit.

        ORIGIN             0x1000
TABLE:  DATABYTE           0x7E,0x30,0x6D,0x79  Table that contains
        DATABYTE           0x33,0x5B,0x5F,0x70    the necessary
        DATABYTE           0x7F,0x7B,0x00,0x00    7-segment patterns.
        DATABYTE           0x00,0x00,0x00,0x00
```

Figure 3.18 A RISC-style program for Example 3.6.

```
DIGIT   EQU          0x800          Location of ASCII-encoded digit.
SEVEN   EQU          0x4030         Address of 7-segment display.
        Move         R2, DIGIT      Load the ASCII-encoded digit.
        Move         R3, R2
        And          R3, #0xF0      Extract high-order bits of ASCII.
        And          R2, #0x0F      Extract the decimal number.
        CompareByte  R3, #0x30      Check if high-order bits of
        Branch=0     HIGH3            ASCII code are 0011.
        Move         R2, #0x0F      Not a digit, display a blank.
HIGH3:  MoveByte     SEVEN, TABLE(R2)  Display the digit.

        ORIGIN       0x1000
TABLE:  DATABYTE     0x7E,0x30,0x6D,0x79  Table that contains
        DATABYTE     0x33,0x5B,0x5F,0x70    the necessary
        DATABYTE     0x7F,0x7B,0x00,0x00    7-segment patterns.
        DATABYTE     0x00,0x00,0x00,0x00
```

Figure 3.19 A CISC-style program for Example 3.6.

PROBLEMS

3.1 **[E]** The input status bit in an interface circuit is cleared as soon as the input data register is read. Why is this important?

3.2 **[E]** Write a program that displays the contents of ten bytes of the main memory in hexadecimal format on a line of a display device. The ten bytes start at location LOC in the memory, and there are two hex characters per byte. The contents of successive bytes should be separated by a space when displayed.

3.3 **[E]** What is the difference between a subroutine and an interrupt-service routine?

3.4 **[E]** In the first And instruction in Figure 3.4 the immediate value 2 is used when checking the KIN flag, but in Figure 3.5 the immediate value 1 is used in the first TestBit instruction when checking the same flag. Explain the difference.

3.5 **[D]** A computer is required to accept characters from the keyboard input of 20 terminals. The main memory area to be used for storing data for each terminal is pointed to by a pointer PNTRn, where $n = 1$ through 20. Input data must be collected from the terminals while another program PROG is being executed. This may be accomplished in one of two ways:

(a) Every T seconds, program PROG calls a polling subroutine POLL. This subroutine checks the status of each of the 20 terminals in sequence and transfers any input characters to the memory. Then it returns to PROG.

(b) Whenever a character is ready in any of the interface buffers of the terminals, an interrupt request is generated. This causes the interrupt routine INTERRUPT to be executed. INTERRUPT polls the status registers to find the first ready character, transfers it, and then returns to PROG.

Write the routines POLL and INTERRUPT. Let the maximum character rate for any terminal be c characters per second, with an average rate equal to rc, where $r \leq 1$. In method (a), what is the maximum value of T for which it is still possible to guarantee that no input characters will be lost? What is the equivalent value for method (b)? Estimate, on the average, the percentage of time spent in servicing the terminals for methods (a) and (b), for $c = 100$ characters per second and $r = 0.01$, 0.1, 0.5, and 1. Assume that POLL takes 800 ns to poll all 20 devices and that an interrupt from a device requires 200 ns to process.

3.6 **[E]** In Figure 3.9, the interrupt-enable bit in the PS is set last in the START section of the Main program. Why? Does the order matter for earlier operations in START? Why or why not?

3.7 **[E]** Even if multiple interrupt requests are pending, only one request will be handled for each entry into ILOC in Figure 3.9. True or false? Explain.

3.8 **[E]** A user program could check for a zero divisor immediately preceding each division operation, and then take appropriate action without invoking the OS. Give reasons why this may or may not be preferable to allowing an exception interrupt to occur on an actual divide by zero situation in a user program.

3.9 **[M]** Assume that a memory location BINARY contains a 16-bit pattern. It is desired to display these bits as a string of 0s and 1s on a display device that has the interface depicted in Figure 3.3. Write a RISC-style program that accomplishes this task.

3.10 **[M]** Write a CISC-style program for the task in Problem 3.9.

3.11 **[E]** Modify the program in Figure 3.18 if the address of TABLE is 0x10100.

3.12 **[E]** Modify the program in Figure 3.19 if the address of TABLE is 0x10100.

3.13 **[M]** Using the seven-segment display in Figure 3.17 and the timer interface registers in Figure 3.14, write a RISC-style program that flashes decimal digits in the repeating sequence 0, 1, 2, . . . , 9, 0, ... Each digit is to be displayed for one second. Assume that the counter in the timer circuit is driven by a 100-MHz clock.

3.14 **[M]** Write a CISC-style program for the task in Problem 3.13.

3.15 **[D]** Using two 7-segment displays of the type shown in Figure 3.17, and the timer interface registers in Figure 3.14, write a RISC-style program that flashes the repeating sequence of numbers 0, 1, 2, . . . , 98, 99, 0, ... Each number is to be displayed for one second. Assume that the counter in the timer circuit is driven by a 100-MHz clock.

3.16 **[D]** Write a CISC-style program for the task in Problem 3.15.

3.17 **[D]** Write a RISC-style program that computes wall clock time and displays the time in hours (0 to 23) and minutes (0 to 59). The display consists of four 7-segment display devices of the type shown in Figure 3.17. A timer circuit that has the interface registers given in Figure 3.14 is available. Its counter is driven by a 100-MHz clock.

3.18 **[D]** Write a CISC-style program for the task in Problem 3.17.

3.19 **[M]** Write a RISC-style program that displays the name of the user backwards. The program should display a prompt requesting that the characters in the user's name be entered on the keyboard, followed by the carriage return (CR). The program should accept a sequence of characters and store them in the main memory. It should then display a message to indicate that the user's name will be displayed backwards, followed by the display of the characters from the user's name in reverse order.

3.20 **[M]** Write a CISC-style program for the task in Problem 3.19.

3.21 **[M]** Write a RISC-style program that determines whether a word entered by a user on the keyboard is a palindrome, i.e., a word that is same when its characters are written in normal and reverse order. The program should display a prompt requesting that the user enter the characters of an arbitrary word on the keyboard, followed by the carriage return (CR). The program should read the characters and store them in the main memory. It should then analyze the word to determine whether it is a palindrome. Finally, the program should display a message to indicate the result of the analysis.

3.22 **[M]** Write a CISC-style program for the task in Problem 3.21.

3.23 **[D]** Write a RISC-style program that displays a string of characters centered horizontally on a standard 80-character line and enclosed in a box, as shown below:

```
+------------+
|sample text|
+------------+
```

The string of characters is located in the main memory beginning at address STRING. There is a NUL control character (value 0) at the end of the string of characters. If the string has more than 78 characters (including spaces), the program should truncate the displayed string to 78 characters. The program for determining the length of a character string in Example 2.1 can be adapted as a subroutine for use by the program in this problem. Assume that the display device has the interface depicted in Figure 3.3.

3.24 **[D]** Write a CISC-style program for the task in Problem 3.23.

3.25 **[D]** Write a RISC-style program that displays a long sequence of text encoded in ASCII characters with automatic wraparound to fit within 80-character lines. Before displaying the next word, the program must determine whether there is sufficient space remaining on the line. If not, the word should appear at the beginning of the next line. The display process must continue until the NUL control character (value 0) is reached at the end of the sequence of characters to be displayed. Assume that the sequence of characters uses no control characters other than the NUL character at the end, hence words are separated only by a space character. Assume that the display device has the interface depicted in Figure 3.3.

3.26 **[D]** Write a CISC-style program for the task in Problem 3.25.

chapter

7

Input/Output Organization

Chapter Objectives

In this chapter you will learn about:

- Hardware needed to access I/O devices
- Synchronous and asynchronous bus operation
- Interface circuits
- Commercial standards, such as USB, SAS, and PCI Express

One of the basic features of a computer is its ability to transfer data to and from I/O 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 embedded applications, computers are less visible but equally important. They are an integral part of home appliances, manufacturing equipment, vehicle systems, cell phones, and banking and point-of-sale terminals. In such applications, input to a computer may come from a touch panel, a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be characters or numbers to be displayed, 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.

A computer should have the ability to exchange information with a wide variety of devices. In many cases, the processor is fully involved in these exchanges. However, data transfers may also take place directly between I/O devices, such as magnetic hard disks, and the main memory, with only minimal involvement of the processor. This possibility will be explored in the next chapter on the memory system.

Chapter 3 presents the programmer's view of input/output data transfers that take place between the processor and the registers in I/O device interfaces. In this chapter, we discuss the details of the hardware needed to make such transfers possible.

An interconnection network is used to transfer data among the processor, memory, and I/O devices. We describe below a commonly used interconnection network called a *bus*.

7.1 BUS STRUCTURE

The bus shown in Figure 7.1 is a simple structure that implements the interconnection network in Figure 3.1. Only one source/destination pair of units can use this bus to transfer data at any one time.

The bus consists of three sets of lines used to carry address, data, and control signals. I/O device interfaces are connected to these lines, as shown in Figure 7.2 for an input device. Each I/O device is assigned a unique set of addresses for the registers in its interface. When the processor places a particular address on the address lines, it is examined by the address

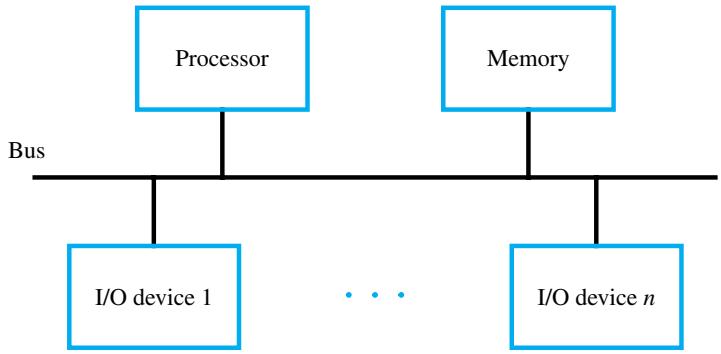


Figure 7.1 A single-bus structure.

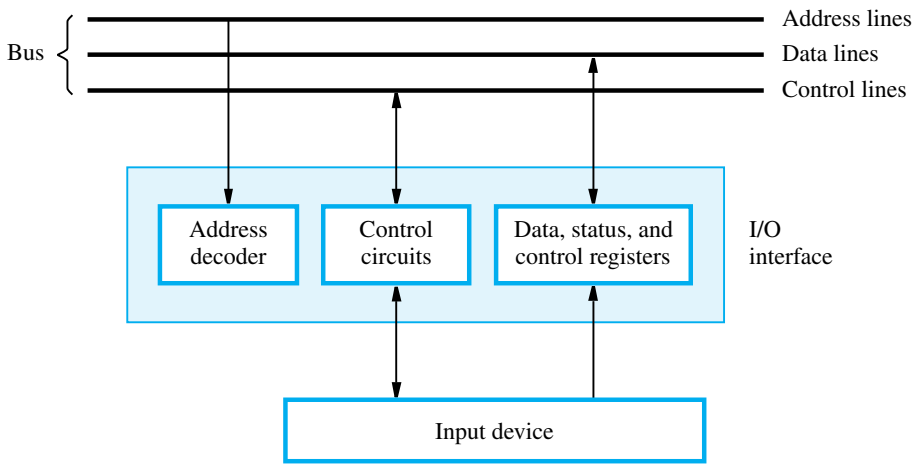


Figure 7.2 I/O interface for an input device.

decoders of all devices on the bus. The device that recognizes this address responds to the commands issued on the control lines. The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines.

When I/O devices and the memory share the same address space, the arrangement is called *memory-mapped I/O,* as described in Section 3.1. Any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if the input device in Figure 7.2 is a keyboard and if DATAIN is its data register, the instruction

```
Load    R2, DATAIN
```

reads the data from DATAIN and stores them into processor register R2. Similarly, the instruction

```
Store   R2, DATAOUT
```

sends the contents of register R2 to location DATAOUT, which may be the data register of a display device interface. The status and control registers contain information relevant to the operation of the I/O device. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

7.2 BUS OPERATION

A bus requires a set of rules, often called a *bus protocol*, that govern how the bus is used by various devices. The bus protocol determines when a device may place information on the bus, when it may load the data on the bus into one of its registers, and so on. These rules are implemented by control signals that indicate what and when actions are to be taken.

One control line, usually labelled R/$\overline{W}$, specifies whether a Read or a Write operation is to be performed. As the label suggests, it specifies Read when set to 1 and Write when set to 0. When several data sizes are possible, such as byte, halfword, or word, the required size is indicated by other control lines. The bus control lines also carry timing information. They specify the times at which the processor and the I/O devices may place data on or receive data from the data lines. 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.

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. Normally, the processor acts as the master, but other devices may also become masters as we will see in Section 7.3. The device addressed by the master is referred to as a *slave*.

7.2.1 SYNCHRONOUS BUS

On a *synchronous* bus, all devices derive timing information from a control line called the *bus clock*, shown at the top of Figure 7.3. The signal on this line has two *phases*: a high level followed by a low level. The two phases constitute a *clock cycle*. The first half of the cycle between the low-to-high and high-to-low transitions is often referred to as a clock pulse.

The address and data lines in Figure 7.3 are shown as if they are carrying both high and low signal levels at the same time. This is a common convention for indicating that

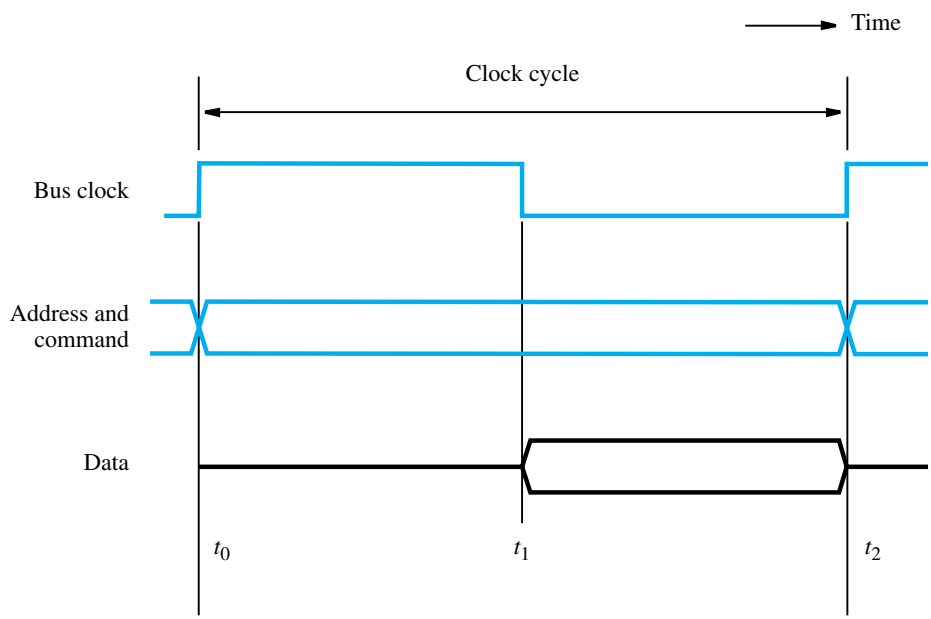


Figure 7.3 Timing of an input transfer on a synchronous bus.

some lines are high and some low, depending on the particular address or data values being transmitted. The crossing points indicate the times at which these patterns change. A signal line at a level half-way between the low and high signal levels indicates periods during which the signal is unreliable, and must be ignored by all devices.

Let us consider the sequence of signal events during an input (Read) operation. At time t_0, the master places the device address on the address lines and sends a command on the control lines indicating a Read operation. The command may also specify the length of the operand to be read. 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 over the bus. Also, it must 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 by placing the requested input data on the data lines. At the end of the clock cycle, at time t_2, the master loads the data on the data lines into one of its registers. To be loaded correctly into a register, data must be available for a period greater than the setup time of the register (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 master's register.

A similar procedure is followed for a Write 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 loads the data into its data register.

The timing diagram in Figure 7.3 is an idealized representation of the actions that take place on the bus lines. The exact times at which signals change state are somewhat different from those shown, because of propagation delays on bus wires and in the circuits of the devices. Figure 7.4 gives a more realistic picture of what actually happens. 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. The top view shows the signals as seen by the master and the bottom view as seen by the slave. We assume that the clock changes are seen at the same time by all devices connected to the bus. System designers spend considerable effort to ensure that the clock signal satisfies this requirement.

The master sends the address and command signals on the rising edge of the clock at the beginning of the clock cycle (at t_0). However, these signals do not actually appear on the bus until t_{AM}, largely due to the delay in the electronic circuit output from the master to the bus lines. A short 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 register. Hence the period $t_2 - t_{DM}$ must be greater than the setup time of that register. The data must continue to be valid after t_2 for a period equal to the hold time requirement of the register (see Appendix A for hold time).

Timing diagrams often show only the simplified picture in Figure 7.3, particularly when the intent is to give the basic idea of how data are transferred. But, actual signals will always involve delays as shown in Figure 7.4.

Multiple-Cycle Data Transfer

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,

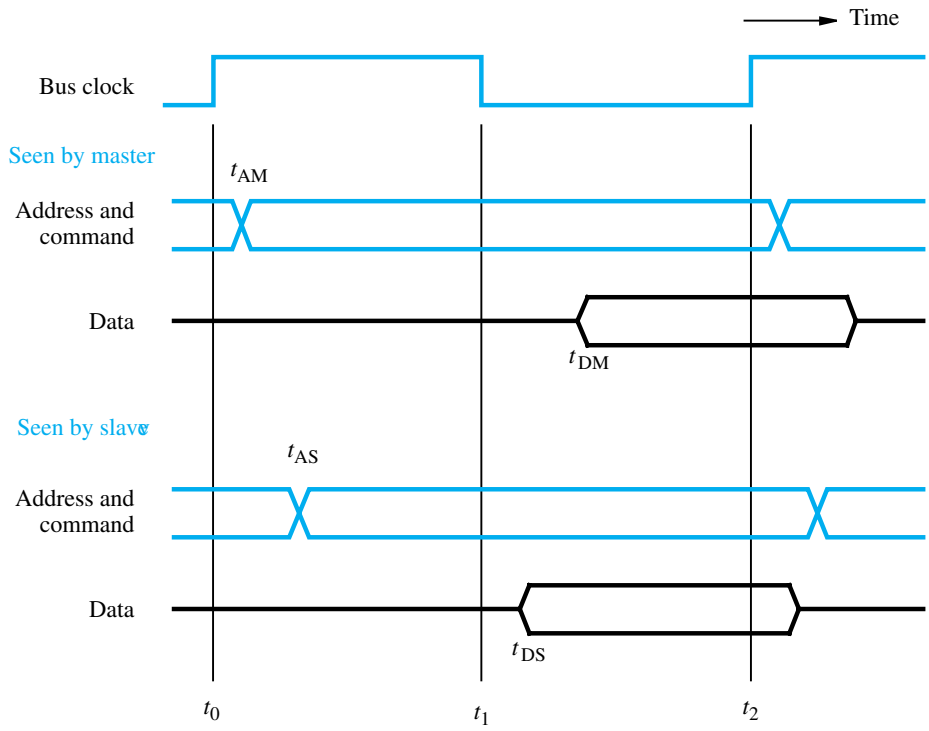


Figure 7.4 A detailed timing diagram for the input transfer of Figure 7.3.

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. At t_2, it simply assumes that the input data are available on the data lines in a Read operation, or that the output data have been received by the I/O device in a Write operation. If, because of a malfunction, a device does not operate correctly, 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 match the response speeds of different devices. This is often accomplished by allowing a complete data transfer operation to 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 7.5. 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. It begins to access the requested data on the active

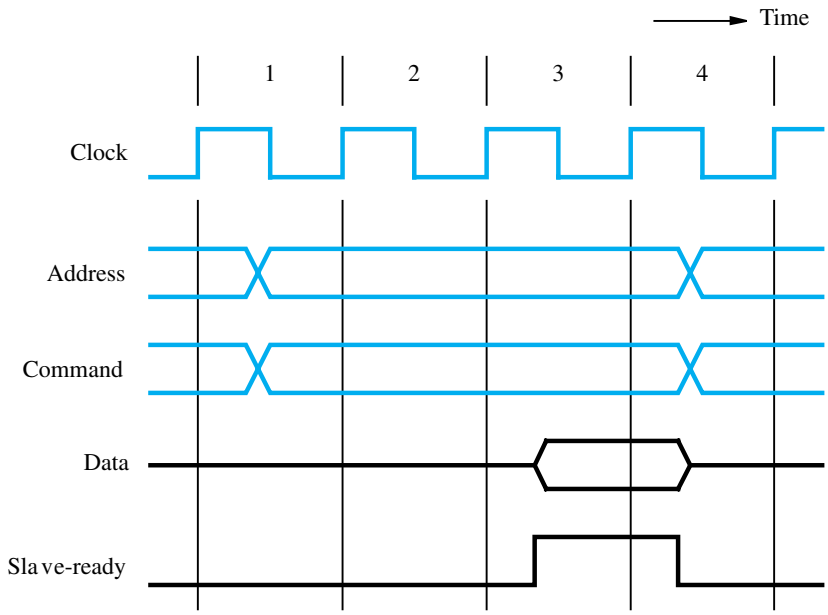


Figure 7.5 An input transfer using multiple clock cycles.

edge of the clock at the beginning of clock cycle 2. We have assumed that due to the delay involved in getting the data, the slave cannot respond immediately. The data become ready and are placed on the bus during clock cycle 3. The slave asserts a control signal called Slave-ready at the same time. The master, which has been waiting for this signal, loads the data into its register at the end of the clock cycle. The slave removes its data signals from the bus and returns its Slave-ready signal to the low level at the end of cycle 3. The bus transfer operation is now complete, and the master may send new address and command signals to start a new transfer in clock cycle 4.

The Slave-ready signal is an acknowledgment from the slave to the master, confirming that the requested data have been placed on the bus. It also allows the duration of a bus transfer to change from one device to another. In the example in Figure 7.5, the slave responds in cycle 3. A different device may respond in an earlier or a later cycle. 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.

We will now present a different approach that does not use a clock signal at all.

7.2.2 ASYNCHRONOUS BUS

An alternative scheme for controlling data transfers on a bus is based on the use of a *handshake* protocol between the master and the slave. A handshake is an exchange of

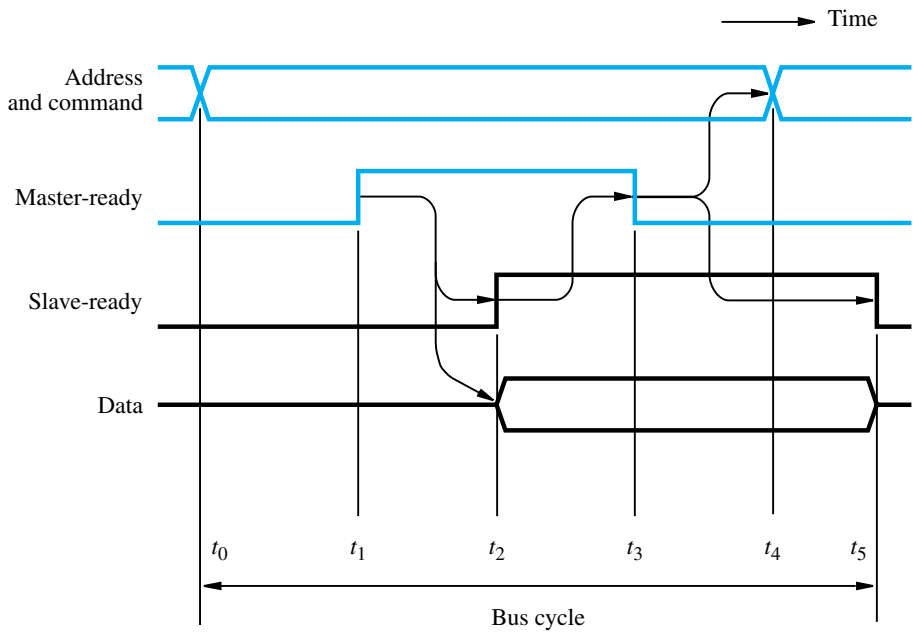


Figure 7.6 Handshake control of data transfer during an input operation.

command and response signals between the master and the slave. It is a generalization of the way the Slave-ready signal is used in Figure 7.5. A control line called Master-ready is asserted by the master to indicate that it is ready to start a data transfer. The Slave responds by asserting Slave-ready.

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 to all devices that it has done so by activating the Master-ready line. This causes all devices to decode the address. The selected slave performs the required operation and informs the processor that 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 loads the data into one of its registers.

An example of the timing of an input data transfer using the handshake protocol is given in Figure 7.6, 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 decode this information.

t_1—The master sets the Master-ready line to 1 to inform the 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 transmitted simultaneously 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 longer than the maximum possible bus skew. (Note that bus skew is a part of the maximum propagation delay in the synchronous case.) Sufficient time should be allowed for the device 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 its data 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.

t_3—The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. The master must allow for bus skew. It must also allow for the setup time needed by its register. After a delay equivalent to the maximum bus skew and the minimum setup time, the master loads the data into its register. Then, 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 7.7, is essentially the same as for an input operation. In this case, the master places the output data on the data lines at the same time that it transmits the address and command information. The selected slave loads the data into its data register 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 similar to the input operation.

The handshake signals in Figures 7.6 and 7.7 are said to be *fully interlocked*, because a change in one signal is always in response to a change in the other. Hence, this scheme is known as a *full handshake*. It provides the highest degree of flexibility and reliability.

Discussion

Many variations of the bus protocols just described are found in commercial computers. 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

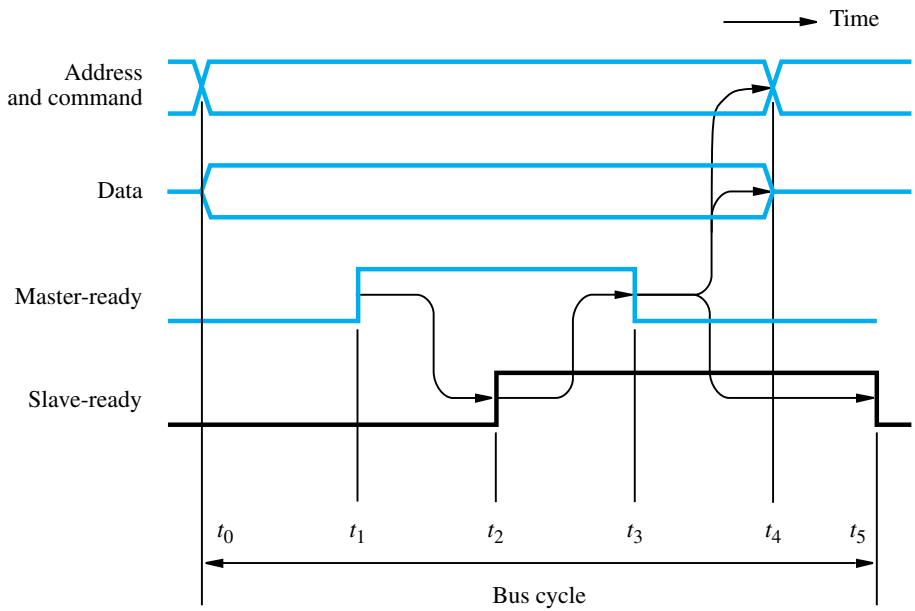


Figure 7.7 Handshake control of data transfer during an output operation.

The main advantage of the asynchronous bus is that the handshake protocol eliminates the need for distribution of a single clock signal whose edges should be seen by all devices at about the same time. This simplifies timing design. Delays, whether introduced by the interface circuits or by propagation over the bus wires, are readily accommodated. These delays are likely to differ from one device to another, but the timing of data transfers adjusts automatically. For a synchronous bus, clock circuitry must be designed carefully to ensure proper timing, and delays must be kept within strict bounds.

The rate of data transfer on an asynchronous bus controlled by the handshake protocol is limited by the fact that each transfer involves two round-trip delays (four end-to-end delays). This can be seen in Figures 7.6 and 7.7 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 round trip 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 the synchronous approach.

7.2.3 ELECTRICAL CONSIDERATIONS

A bus is an interconnection medium to which several devices may be connected. It is essential to ensure that only one device can place data on the bus at any given time. A logic gate that places data on the bus is called a *bus driver*. All devices connected to the bus, except the one that is currently sending data, must have their bus drivers turned off. A special type of logic gate, known as a *tri-state gate*, is used for this purpose. A tri-state gate

has a control input that is used to turn the gate on or off. When turned on, or enabled, it drives the bus with 1 or 0, corresponding to the value of its input signal. When turned off, or disabled, it is effectively disconnected from the bus. From an electrical point of view, its output goes into a high-impedance state that does not affect the signal on the bus.

7.3 ARBITRATION

There are occasions when two or more entities contend for the use of a single resource in a computer system. For example, two devices may need to access a given slave at the same time. In such cases, it is necessary to decide which device will access the slave first. The decision is usually made in an arbitration process performed by an *arbiter* circuit. The arbitration process starts by each device sending a *request* to use the shared resource. The arbiter associates priorities with individual requests. If it receives two requests at the same time, it *grants* the use of the slave to the device having the higher priority first.

To illustrate the arbitration process, we consider the case where a single bus is the shared resource. The device that initiates data transfer requests on the bus is the bus master. In Section 7.2, the discussion involved only one bus master—the processor. It is possible that several devices in a computer system need to be bus masters to transfer data. For example, an I/O device needs to be a bus master to transfer data directly to or from the computer's memory. Since the bus is a single shared facility, it is essential to provide orderly access to it by the bus masters.

A device that wishes to use the bus sends a request to the arbiter. When multiple requests arrive at the same time, the arbiter selects one request and grants the bus to the corresponding device. For some devices, a delay in gaining access to the bus may lead to an error. Such devices must be given high priority. If there is no particular urgency among requests, the arbiter may grant the bus using a simple round-robin scheme.

Figure 7.8 illustrates an arrangement for bus arbitration involving two masters. There are two Bus-request lines, BR1 and BR2, and two Bus-grant lines, BG1 and BG2, connecting

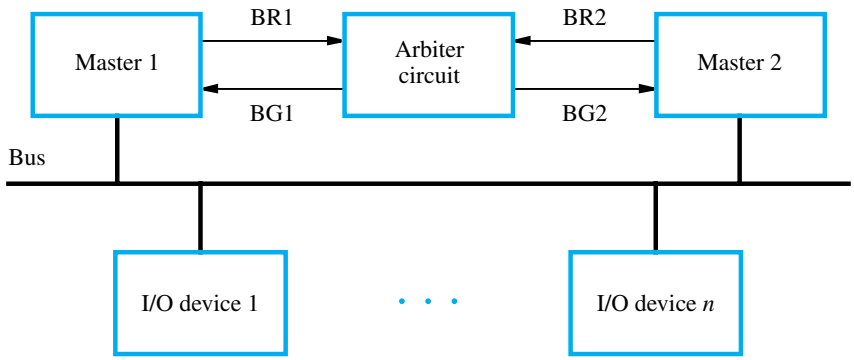


Figure 7.8 Bus arbitration.

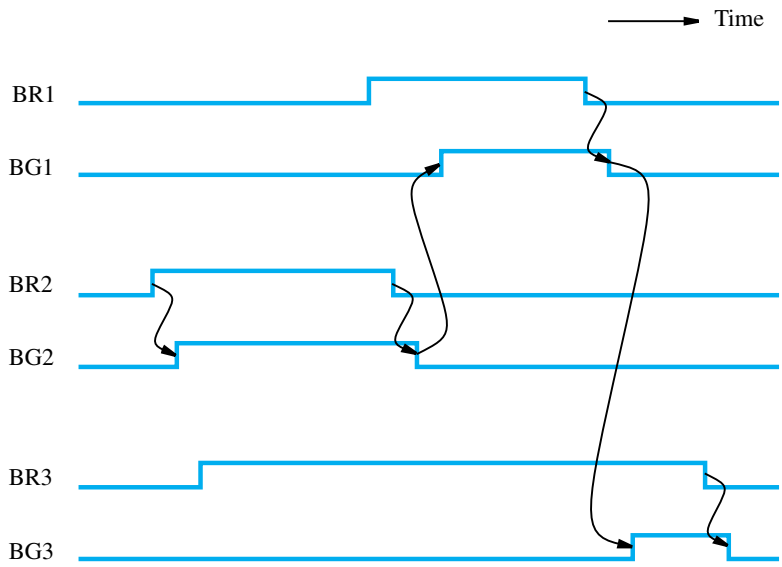


Figure 7.9 Granting use of the bus based on priorities.

the arbiter to the masters. A master requests use of the bus by activating its Bus-request line. If a single Bus-request is activated, the arbiter activates the corresponding Bus-grant. This indicates to the selected master that it may now use the bus for transferring data. When the transfer is completed, that master deactivates its Bus-request, and the arbiter deactivates its Bus-grant.

Figure 7.9 illustrates a possible sequence of events for the case of three masters. Assume that master 1 has the highest priority, followed by the others in increasing numerical order. Master 2 sends a request to use the bus first. Since there are no other requests, the arbiter grants the bus to this master by asserting BG2. When master 2 completes its data transfer operation, it releases the bus by deactivating BR2. By that time, both masters 1 and 3 have activated their request lines. Since device 1 has a higher priority, the arbiter activates BG1 after it deactivates BG2, thus granting the bus to master 1. Later, when master 1 releases the bus by deactivating BR1, the arbiter deactivates BG1 and activates BG3 to grant the bus to master 3. Note that the bus is granted to master 1 before master 3 even though master 3 activated its request line before master 1.

7.4 INTERFACE CIRCUITS

The I/O interface of a device consists of the circuitry needed to connect that device to the bus. On one side of the interface are the bus lines for address, data, and control. On the other side are the connections needed to transfer data between the interface and the I/O

device. This side is called a *port,* and it can be either a parallel or a serial port. A parallel port transfers multiple bits of data simultaneously to or from the device. A serial port sends and receives data one bit at a time. Communication with the processor is the same for both formats; the conversion from a parallel to a serial format and vice versa takes place inside the interface circuit.

Before we present specific circuit examples, let us recall the functions of an I/O interface. According to the discussion in Section 3.1, an I/O interface does the following:

1. Provides a register for temporary storage of data
2. Includes a status register containing status information that can be accessed by the processor
3. Includes a control register that holds the information governing the behavior of the interface
4. Contains address-decoding circuitry to determine when it is being addressed by the processor
5. Generates the required timing signals
6. Performs any format conversion that may be necessary to transfer data between the processor and the I/O device, such as parallel-to-serial conversion in the case of a serial port

7.4.1 PARALLEL INTERFACE

We now explain the key aspects of interface design by means of examples. First, we describe an interface circuit for an 8-bit input port that can be used for connecting a simple input device, such as a keyboard. Then, we describe an interface circuit for an 8-bit output port, which can be used with an output device such as a display. We assume that these interface circuits are connected to a 32-bit processor that uses memory-mapped I/O and the asynchronous bus protocol depicted in Figures 7.6 and 7.7.

Input Interface

Figure 7.10 shows a circuit that can be used to connect a keyboard to a processor. The registers in this circuit correspond to those given in Figure 3.3. Assume that interrupts are not used, so there is no need for a control register. There are only two registers: a data register, KBD_DATA, and a status register, KBD_STATUS. The latter contains the keyboard status flag, KIN.

A typical keyboard consists of mechanical switches that are normally 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 mechanical pushbutton switches is that the contacts *bounce* when a key is pressed, resulting in the electrical connection being made then broken several times before the switch settles in the closed position. Although bouncing may last only one or two milliseconds, this is long enough for the computer to erroneously interpret a single pressing of a key as the key being pressed and released several times. The effect of bouncing can be eliminated using a simple debouncing circuit, which could be part of the keyboard hardware

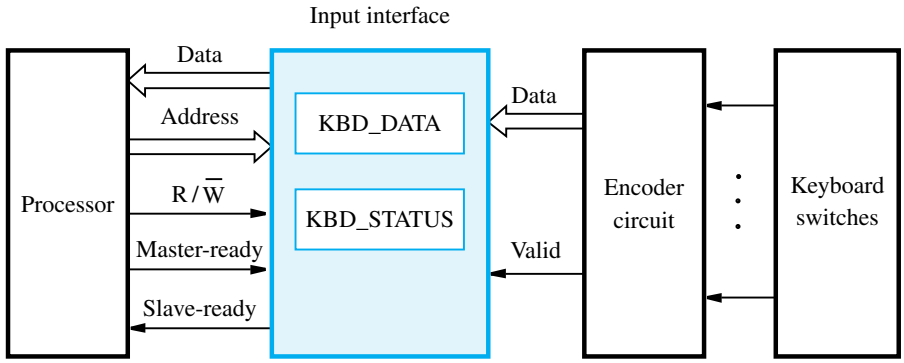


Figure 7.10 Keyboard to processor connection.

or may be incorporated in the encoder circuit. Alternatively, switch bouncing can be dealt with in software. The software detects that a key has been pressed when it observes that the keyboard status flag, KIN, has been set to 1. The I/O routine can then introduce sufficient delay before reading the contents of the input buffer, KBD_DATA, to ensure that bouncing has subsided. When debouncing is implemented in hardware, the I/O routine can read the input character as soon as it detects that KIN is equal to 1.

The output of the encoder in Figure 7.10 consists of one byte of data representing the encoded character and one control signal called Valid. When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code of the corresponding character to be loaded into the KBD_DATA register and the status flag KIN to be set to 1. The status flag is cleared to 0 when the processor reads the contents of the KBD_DATA register. The interface circuit is shown connected to an asynchronous bus on which transfers are controlled by the handshake signals Master-ready and Slave-ready, as in Figure 7.6. The bus has one other control line, $R/\overline{W}$, which indicates a Read operation when equal to 1.

Figure 7.11 shows a possible circuit for the input interface. There are two addressable locations in this interface, KBD_DATA and KBD_STATUS. They occupy adjacent word locations in the address space, as in Figure 3.3. Only one bit, b_1, in the status register actually contains useful information. This is the keyboard status flag, KIN. When the status register is read by the processor, all other bit locations appear as containing zeros.

When the processor requests a Read operation, it places the address of the appropriate register on the address lines of the bus. The address decoder in the interface circuit examines bits A_{31-3}, and asserts its output, My-address, when one of the two registers KBD_DATA or KBD_STATUS is being addressed. Bit A_2 determines which of the two registers is involved. Hence, a multiplexer is used to select the register to be connected to the bus based on address bit A_2. The two least-significant address bits, A_1 and A_0, are not used, because we have assumed that all addresses are word-aligned.

The output of the multiplexer is connected to the data lines of the bus through a set of tri-state gates. The interface circuit turns the tri-state gates on only when the three signals Master-ready, My_address, and $R/\overline{W}$ are all equal to 1, indicating a Read operation. The

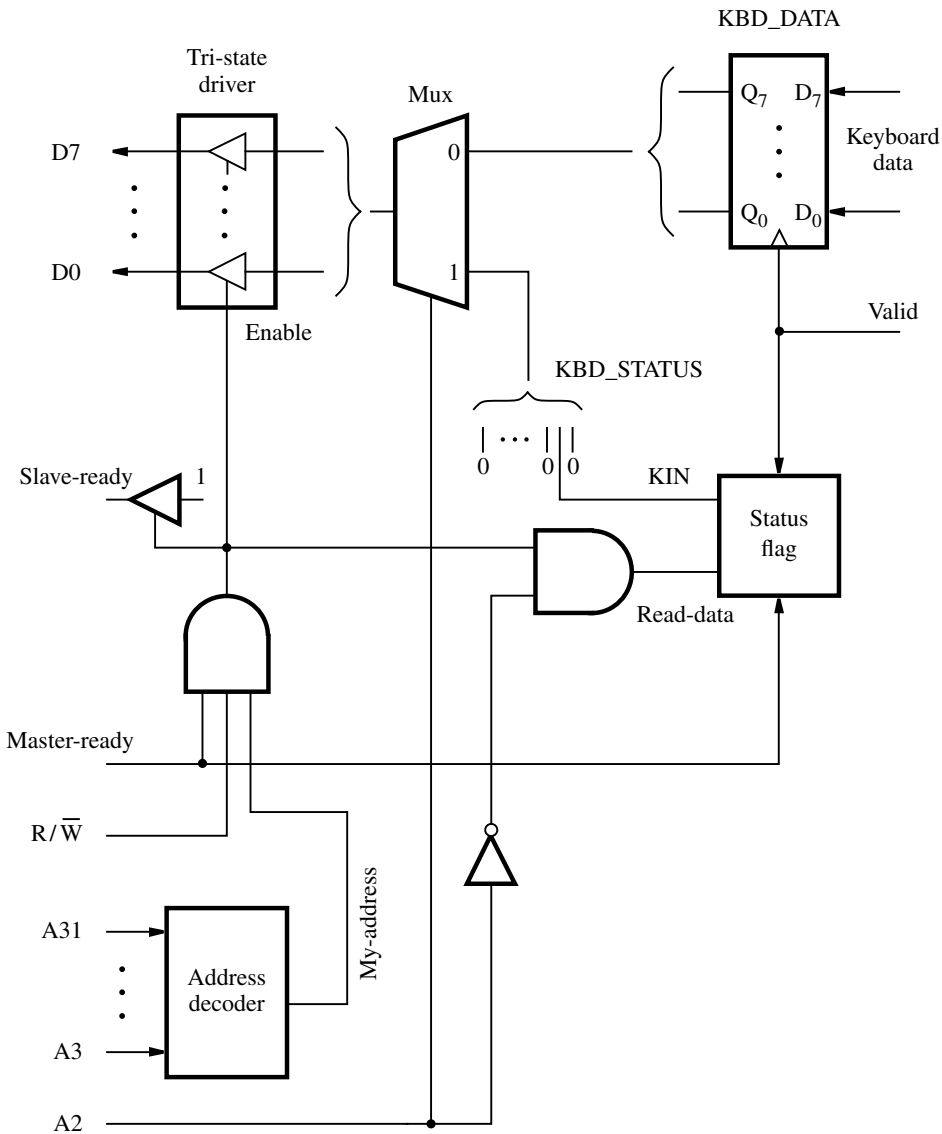


Figure 7.11 An input interface circuit.

Slave-ready signal is asserted at the same time, to inform the processor that the requested data or status information has been placed on the data lines. When address bit A_2 is equal to 0, Read-data is also asserted. This signal is used to reset the KIN flag.

A possible implementation of the status flag circuit is given in Figure 7.12. The KIN flag is the output of a NOR latch connected as shown. A flip-flop is set to 1 by the rising edge on the Valid signal line. This event changes the state of the NOR latch to set KIN to

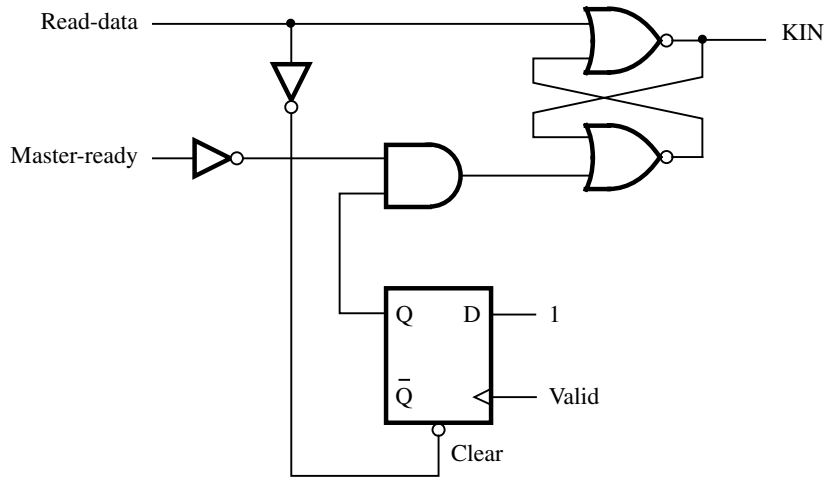


Figure 7.12 Circuit for the status flag block in Figure 7.11.

1, but only when Master-ready is low. The reason for this additional condition is to ensure that KIN does not change state while being read by the processor. Both the flip-flop and the latch are reset to 0 when Read-data becomes equal to 1, indicating that KBD_DATA is being read.

The circuits shown in Figures 7.11 and 7.12 are intended to illustrate the various functions that an interface circuit needs to perform. A designer using modern computer-aided design tools would specify these functions using a hardware description language such as VHDL or Verilog. The resulting circuits would depend on the technology used and may or may not be the same as the circuits shown in these figures.

Output Interface

Let us now consider the output interface shown in Figure 7.13, which can be used to connect an output device such as a display. We have assumed that the display uses two handshake signals, New-data and Ready, in a manner similar to the handshake between the bus signals Master-ready and Slave-ready. When the display is ready to accept a character, it asserts its Ready signal, which causes the DOUT flag in the DISP_STATUS register to be set to 1. When the I/O routine checks DOUT and finds it equal to 1, it sends a character to DISP_DATA. This clears the DOUT flag to 0 and sets the New-data signal to 1. In response, the display returns Ready to 0 and accepts and displays the character in DISP_DATA. When it is ready to receive another character, it asserts Ready again, and the cycle repeats.

Figure 7.14 shows an implementation of this interface. Its operation is similar to that of the input interface of Figure 7.11, except that it responds to both Read and Write operations. A Write operation in which $A_2 = 0$ loads a byte of data into register DISP_DATA. A Read operation in which $A_2 = 1$ reads the contents of the status register DISP_STATUS. In this case, only the DOUT flag, which is bit b_2 of the status register, is sent by the interface. The remaining bits of DISP_STATUS are not used. The state of the status flag is determined

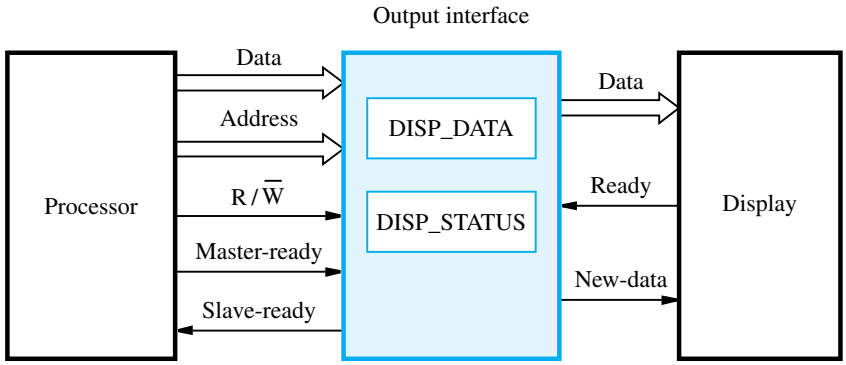


Figure 7.13 Display to processor connection.

by the handshake control circuit. A state diagram describing the behavior of this circuit is given as Example 7.4 at the end of the chapter.

7.4.2 SERIAL INTERFACE

A serial interface is used to connect the processor to I/O devices that transmit data one bit at a time. Data are transferred in a bit-serial fashion on the device side and in a bit-parallel fashion on the processor 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 7.15. 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 transferred to 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. Two status flags, which we will refer to as SIN and SOUT, are maintained by the Status and control block. The SIN flag is set to 1 when new data are loaded into DATAIN from the shift register, and cleared to 0 when these data are read by the processor. The SOUT flag indicates whether the DATAOUT register is available. It is cleared to 0 when the processor writes new data into DATAOUT and set to 1 when data are transferred from DATAOUT to the output shift register.

The double buffering used in the input and output paths in Figure 7.15 is important. It is possible to implement DATAIN and DATAOUT themselves as shift registers, thus obviating the need for separate shift registers. However, this would impose awkward restrictions on the operation of the I/O device. After receiving one character from the serial line, the interface would not be able to start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to give the processor time to read the input data. With double buffering, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the

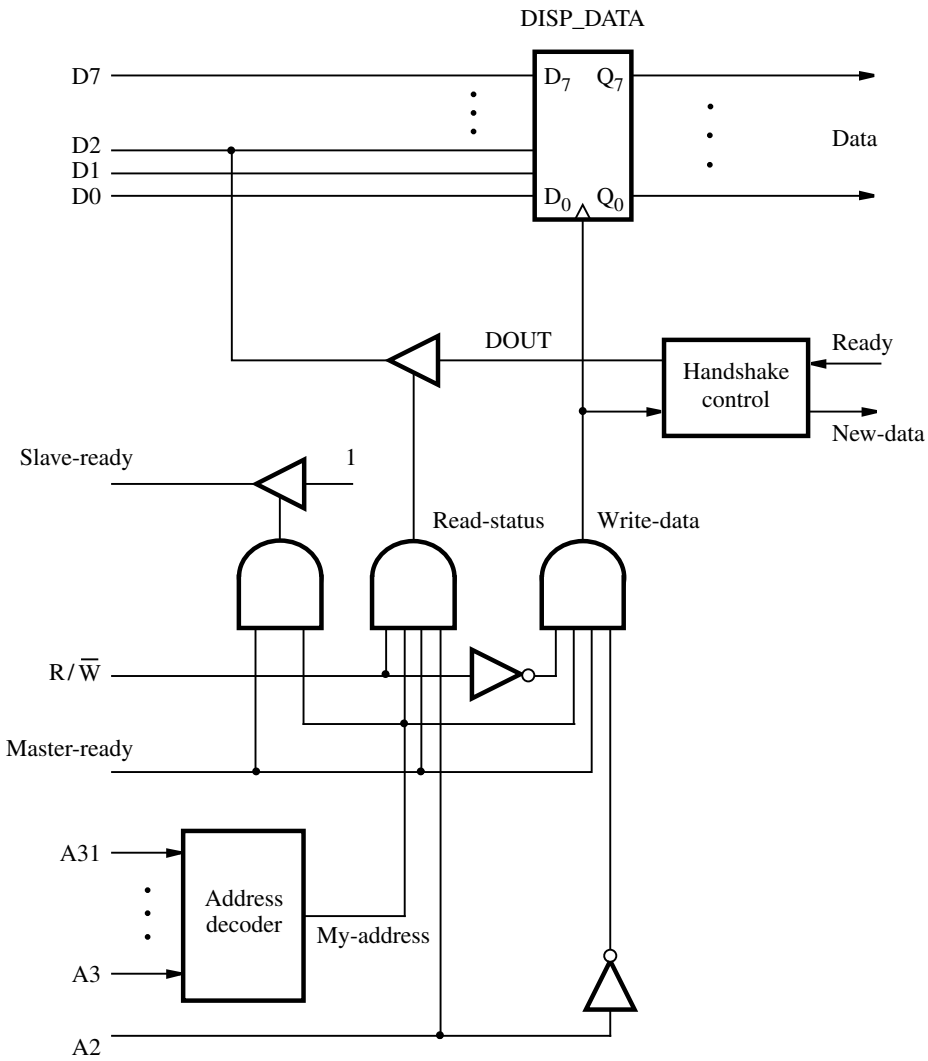


Figure 7.14 An output interface circuit.

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 a continuous stream of input data over the serial line. An analogous situation occurs in the output path of the interface.

During serial transmission, the receiver needs to know when to shift each bit into its input shift register. Since there is no separate line to carry a clock signal from the transmitter to the receiver, the timing information needed must be embedded into the transmitted data using an encoding scheme. There are two basic approaches. The first is known as

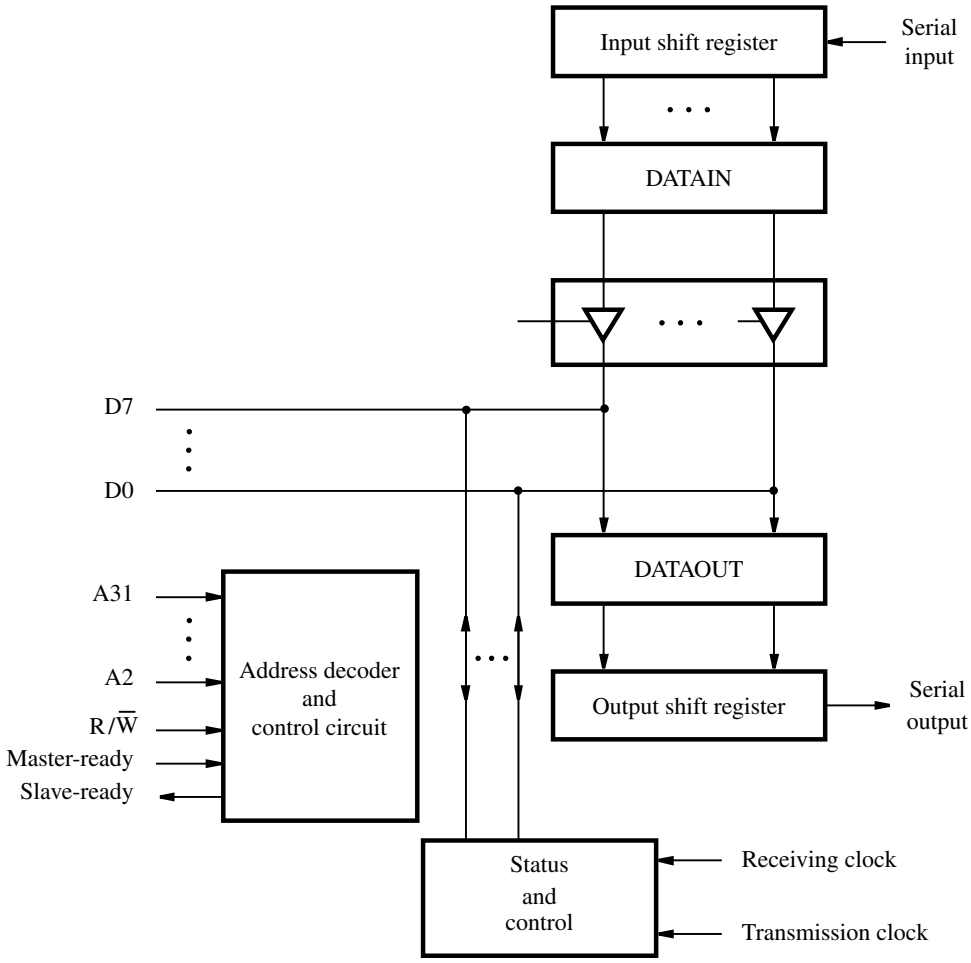


Figure 7.15 A serial interface.

asynchronous transmission, because the receiver uses a clock that is not synchronized with the transmitter clock. In the second approach, the receiver is able to generate a clock that is synchronized with the transmitter clock. Hence it is called synchronous transmission. These approaches are described briefly below.

Asynchronous Transmission

This approach uses a technique called *start-stop* transmission. Data are organized in small groups of 6 to 8 bits, with a well-defined beginning and end. In a typical arrangement, alphanumeric characters encoded in 8 bits are transmitted as shown in Figure 7.16. The line connecting the transmitter and the receiver is in the 1 state when idle. A character is transmitted as a 0 bit, referred to as the Start bit, followed by 8 data bits and 1 or 2 Stop bits. The Stop bits have a logic value of 1. The 1-to-0 transition at the beginning of the

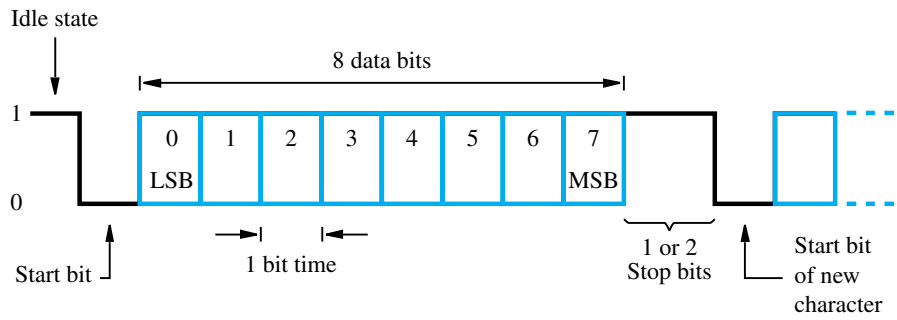


Figure 7.16 Asynchronous serial character transmission.

Start bit alerts the receiver that data transmission is about to begin. Using its own clock, the receiver determines the position of the next 8 bits, which it loads into its input register. The Stop bits following the transmitted character, which are equal to 1, ensure that the Start bit of the next character will be recognized. When transmission stops, the line remains in the 1 state until another character is transmitted.

To ensure correct reception, the receiver needs to sample the incoming data as close to the center of each bit as possible. It does so by using a clock signal whose frequency, f_R, is substantially higher than the transmission clock, f_T. Typically, $f_R = 16f_T$. This means that 16 pulses of the local clock occur during each data bit interval. This clock is used to increment a modulo-16 counter, which is cleared to 0 when the leading edge of a Start bit is detected. The middle of the Start bit is reached at the count of 8. The state of the input line is sampled again at this point to confirm that it is a valid Start bit (a zero), and the counter is cleared to 0. From this point onward, the incoming data signal is sampled whenever the count reaches 16, which should be close to the middle of each incoming bit. Therefore, as long as $f_R/16$ is sufficiently close to f_T, the receiver will correctly load the bits of the incoming character.

Synchronous Transmission

In the start-stop scheme described above, the position of the 1-to-0 transition at the beginning of the start bit in Figure 7.16 is the key to obtaining correct timing information. This scheme is useful only where the speed of transmission is sufficiently low and the conditions on the transmission link are such that the square waveforms shown in the figure maintain their shape. For higher speed a more reliable method is needed for the receiver to recover the timing information.

In synchronous transmission, the receiver generates a clock that is synchronized to that of the transmitter by observing successive 1-to-0 and 0-to-1 transitions in the received signal. It adjusts the position of the active edge of the clock to be in the center of the bit position. A variety of encoding schemes are used to ensure that enough signal transitions occur to enable the receiver to generate a synchronized clock and to maintain synchronization. Once synchronization is achieved, data transmission can continue indefinitely. Encoded data are usually transmitted in large blocks consisting of several hundreds or several thousands of bits. The beginning and end of each block are marked by appropriate codes, and data within

a block are organized according to an agreed upon set of rules. Synchronous transmission enables very high data transfer rates.

7.5 INTERCONNECTION STANDARDS

A typical desktop or notebook computer has several ports that can be used to connect I/O devices, such as a mouse, a memory key, or a disk drive. Standard interfaces have been developed to enable I/O devices to use interfaces that are independent of any particular processor. For example, a memory key that has a USB connector can be used with any computer that has a USB port. In this section, we describe briefly some of the widely used interconnection standards.

Most standards are developed by a collaborative effort among a number of companies. In many cases, the IEEE (Institute of Electrical and Electronics Engineers) develops these standards further and publishes them as IEEE Standards.

7.5.1 UNIVERSAL SERIAL BUS (USB)

The Universal Serial Bus (USB) [1] is the most widely used interconnection standard. A large variety of devices are available with a USB connector, including mice, memory keys, disk drives, printers, cameras, and many more. The commercial success of the USB is due to its simplicity and low cost. The original USB specification supports two speeds of operation, called low-speed (1.5 Megabits/s) and full-speed (12 Megabits/s). Later, USB 2, called High-Speed USB, was introduced. It enables data transfers at speeds up to 480 Megabits/s. As I/O devices continued to evolve with even higher speed requirements, USB 3 (called Superspeed) was developed. It supports data transfer rates up to 5 Gigabits/s.

The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system
- Accommodate a wide range of I/O devices and bit rates, including Internet connections, and audio and video applications
- Enhance user convenience through a "plug-and-play" mode of operation

We will elaborate on some of these objectives before discussing the technical details of the USB.

Device Characteristics

The kinds of devices that may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from these devices vary significantly.

In the case of a keyboard, one byte of data is generated every time a key is pressed, which may happen at any time. These data should be transferred to the computer promptly. Since the event of pressing a key is not synchronized to any other event in a computer system, the data generated by the keyboard are called *asynchronous*. Furthermore, the rate

at which the data are generated is quite low. It is limited by the speed of the human operator to about 10 bytes per second, which is less than 100 bits per second.

A variety of simple devices that may be attached to a computer generate data of a similar nature—low speed and asynchronous. Computer mice and some of the controls and manipulators used with video games are good examples.

Consider now a different source of data. Many computers have a microphone, either externally attached or built in. The sound picked up by the microphone produces an analog electrical signal, which must be converted into a digital form before it can be handled by the computer. This is accomplished by sampling the analog signal periodically. For each sample, an analog-to-digital (A/D) converter generates an n-bit number representing the magnitude of the sample. The number of bits, n, is selected based on the desired precision with which to represent each sample. Later, when these data are sent to a speaker, a digital-to-analog (D/A) converter is used to restore the original analog signal from the digital format. A similar approach is used with video information from a camera.

The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called *isochronous,* meaning that successive events are separated by equal periods of time. A signal must be sampled quickly enough to track its highest-frequency components. In general, if the sampling rate is s samples per second, the maximum frequency component captured by the sampling process is $s/2$. For example, human speech can be captured adequately with a sampling rate of 8 kHz, which will record sound signals having frequencies up to 4 kHz. For higher-quality sound, as needed in a music system, higher sampling rates are used. A standard sampling rate for digital sound is 44.1 kHz. Each sample is represented by 4 bytes of data to accommodate the wide range in sound volume (dynamic range) that is necessary for high-quality sound reproduction. This yields a data rate of about 1.4 Megabits/s.

An important requirement in dealing with sampled voice or music is to maintain precise timing in the sampling and replay processes. A high degree of jitter (variability in sample timing) is unacceptable. Hence, the data transfer mechanism between a computer and a music system must maintain consistent delays from one sample to the next. Otherwise, complex buffering and retiming circuitry would be needed. On the other hand, occasional errors or missed samples can be tolerated. They either go unnoticed by the listener or they may cause an unobtrusive click. No sophisticated mechanisms are needed to ensure perfectly correct data delivery.

Data transfers for images and video have similar requirements, but require much higher data transfer rates. To maintain the picture quality of commercial television, an image should be represented by about 160 kilobytes and transmitted 30 times per second. Together with control information, this yields a total bit rate of 44 Megabits/s. Higher-quality images, as in HDTV (High Definition TV), require higher rates.

Large storage devices such as magnetic and optical disks present different requirements. These devices are part of the computer's memory hierarchy, as will be discussed in Chapter 8. Their connection to the computer requires a data transfer bandwidth of at least 40 or 50 Megabits/s. Delays on the order of milliseconds are introduced by the movement of the mechanical components in the disk mechanism. Hence, a small additional delay introduced while transferring data to or from the computer is not important, and jitter is not an issue. However, the transfer mechanism must guarantee data correctness.

Plug-and-Play

When an I/O device is connected to a computer, the operating system needs some information about it. It needs to know what type of device it is so that it can use the appropriate device driver. It also needs to know the addresses of the registers in the device's interface to be able to communicate with it. The USB standard defines both the USB hardware and the software that communicates with it. Its *plug-and-play* feature means that when a new device is connected, the system detects its existence automatically. The software determines the kind of device and how to communicate with it, as well as any special requirements it might have. As a result, the user simply plugs in a USB device and begins to use it, without having to get involved in any of these details.

The USB is also hot-pluggable, which means a device can be plugged into or removed from a USB port while power is turned on.

USB Architecture

The USB uses point-to-point connections and a serial transmission format. When multiple devices are connected, they are arranged in a tree structure as shown in Figure 7.17. Each node of the tree has a device called a *hub,* which acts as an intermediate transfer point between the host computer and the I/O devices. At the root of the tree, a *root hub* connects the entire tree to the host computer. The leaves of the tree are the I/O devices: a mouse, a keyboard, a printer, an Internet connection, a camera, or a speaker. The tree structure makes it possible to connect many devices using simple point-to-point serial links.

If I/O devices are allowed to send messages at any time, two messages may reach the hub at the same time and interfere with each other. For this reason, the USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host processor. Hence, no two devices can send messages at the same time. This restriction allows hubs to be simple, low-cost devices.

Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the processor's address space. The root hub of the USB, which is attached to the processor, appears as a single device. The host software communicates with individual devices by sending information to the root hub, which it forwards to the appropriate device in the USB tree.

When a device is first connected to a hub, or when it is powered on, it has the address 0. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected. When the host is informed that a new device has been connected, it reads the information in a special memory in the device's USB interface to learn about the device's capabilities. It then assigns the device a unique USB address and writes that address in one of the device's interface registers. It is this initial connection procedure that gives the USB its plug-and-play capability.

Isochronous Traffic on USB

An important feature of the USB is its ability to support the transfer of isochronous data in a simple manner. As mentioned earlier, isochronous data need to be transferred at precisely timed regular intervals. To accommodate this type of traffic, the root hub transmits a uniquely recognizable sequence of bits over the USB tree every millisecond. This sequence of bits, called a Start of Frame character, acts as a marker indicating the

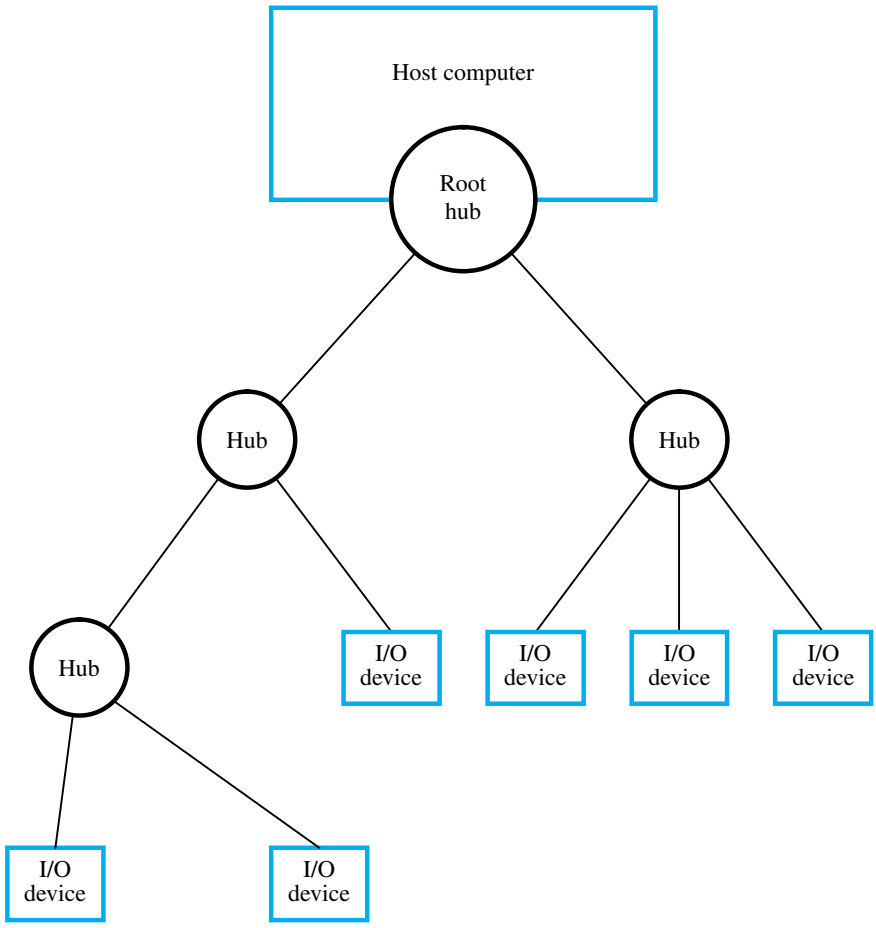


Figure 7.17 Universal Serial Bus tree structure.

beginning of isochronous data, which are transmitted after this character. Thus, digitized audio and video signals can be transferred in a regular and precisely timed manner.

Electrical Characteristics

USB connections consist of four wires, of which two carry power, +5 V and Ground, and two carry data. Thus, I/O devices that do not have large power requirements can be powered directly from the USB. This obviates the need for a separate power supply for simple devices such as a memory key or a mouse.

Two methods are used to send data over a USB cable. When sending data at low speed, a high voltage relative to Ground is transmitted on one of the two data wires to represent a 0 and on the other to represent a 1. The Ground wire carries the return current in both cases. Such a scheme in which a signal is injected on a wire relative to ground is referred to as *single-ended* transmission.

The speed at which data can be sent on any cable is limited by the amount of electrical noise present. The term *noise* refers to any signal that interferes with the desired data signal and hence could cause errors. Single-ended transmission is highly susceptible to noise. The voltage on the ground wire is common to all the devices connected to the computer. Signals sent by one device can cause small variations in the voltage on the ground wire, and hence can interfere with signals sent by another device. Interference can also be caused by one wire picking up noise from nearby wires.

The High-Speed USB uses an alternative arrangement known as *differential signaling*. The data signal is injected between two data wires twisted together. The ground wire is not involved. The receiver senses the voltage difference between the two signal wires directly, without reference to ground. This arrangement is very effective in reducing the noise seen by the receiver, because any noise injected on one of the two wires of the twisted pair is also injected on the other. Since the receiver is sensitive only to the voltage difference between the two wires, the noise component is cancelled out. The ground wire acts as a shield for the data on the twisted pair against interference from nearby wires. Differential signaling allows much lower voltages and much higher speeds to be used compared to single-ended signaling.

7.5.2 FIREWIRE

FireWire is another popular interconnection standard. It was originally developed by Apple and has been adopted as IEEE Standard 1394 [2]. Like the USB, it uses differential point-to-point serial links. The following are some of the salient differences between FireWire and USB.

- Devices are organized in a daisy chain manner on a FireWire bus, instead of the tree structure of USB. One device is connected to the computer, a second device is connected to the first one, a third device is connected to the second one, and so on.
- FireWire is well suited for connecting audio and video equipment. It can be operated in an isochronous mode that is highly optimized for carrying high-speed isochronous traffic.
- I/O devices connected to the USB communicate with the host computer. If data are to be transferred from one device to another, for example from a camera to a display or printer, they are first read by the host then sent to the display or printer. FireWire, on the other hand, supports a mode of operation called *peer-to-peer*. This means that data may be transferred directly from one I/O device to another, without the host's involvement.
- The basic FireWire connector has six pins. There are two pairs of data wires, one for transmission in each direction, and two for power and ground. Higher-speed versions use a nine-pin connector, with three ground wires added to shield the data wires against interference.
- The FireWire bus can deliver considerably more power than the USB. Hence, it can support devices with moderate power requirements.

FireWire is widely used with audio and video devices. For example, most camcorders have a FireWire port. Several versions of the standard have been defined, which can operate at speeds ranging from 400 Megabits/s to 3.6 Gigabits/s.

7.5.3 PCI Bus

The PCI (Peripheral Component Interconnect) bus [3] was developed as a low-cost, processor-independent bus. It is housed on the motherboard of a computer and used to connect I/O interfaces for a wide variety of devices. A device connected to the PCI bus appears to the processor as if it is connected directly to the processor bus. Its interface registers are assigned addresses in the address space of the processor.

We will start by describing how the PCI bus operates, then discuss some of its features.

Bus Structure

The use of the PCI bus in a computer system is illustrated in Figure 7.18. The PCI bus is connected to the processor bus via a controller called a bridge. The bridge has a special port for connecting the computer's main memory. It may also have another special high-speed port for connecting graphics devices. The bridge translates and relays commands and responses from one bus to the other and transfers data between them. For example, when

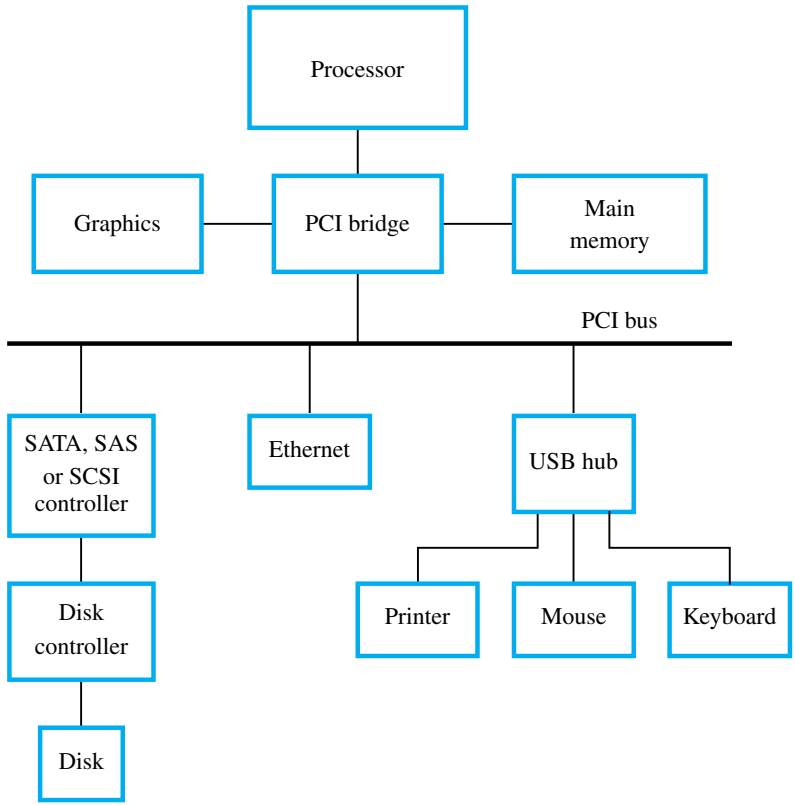


Figure 7.18 Use of a PCI bus in a computer system.

the processor sends a Read request to an I/O device, the bridge forwards the command and address to the PCI bus. When the bridge receives the device's response, it forwards the data to the processor using the processor bus. I/O devices are connected to the PCI bus, possibly through ports that use standards such as Ethernet, USB, SATA, SCSI, or SAS.

The PCI bus supports three independent address spaces: memory, I/O, and configuration. The system designer may choose to use memory-mapped I/O even with a processor that has a separate I/O address space. In fact, this is the approach recommended by the PCI standard for wider compatibility. The configuration space is intended to give the PCI its plug-and-play capability, as we will explain shortly. A 4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation.

Data transfers on a computer bus often involve bursts of data rather than individual words. Words stored in successive memory locations are transferred directly between the memory and an I/O device such as a disk or an Ethernet connection. Data transfers are initiated by the interface of the I/O device, which acts as a bus master. This way of transferring data directly between the memory and I/O devices is discussed in detail in Chapter 8. The PCI bus is designed primarily to support multiple-word transfers. A Read or a Write operation involving a single word is simply treated as a burst of length one.

The signaling convention on the PCI bus is similar to that used in Figure 7.5, with one important difference. The PCI bus uses the same lines to transfer both address and data. In Figure 7.5, we assumed that the master maintains the address information on the bus until the data transfer is completed. But, this is not necessary. The address is needed only long enough for the slave to be selected, freeing the lines for sending data in subsequent clock cycles. For transfers involving multiple words, the slave can store the address in an internal register and increment it to access successive address locations. A significant cost reduction can be realized in this manner, because the number of bus lines is an important factor affecting the cost of a computer system.

Data Transfer

To understand the operation of the bus and its various features, we will examine a typical bus transaction. The bus master, which is the device that initiates data transfers by issuing Read and Write commands, is called the *initiator* in PCI terminology. The addressed device that responds to these commands is called a *target*. The main bus signals used for transferring data are listed in Table 7.1. There are 32 or 64 lines that carry address and data using a synchronous signaling scheme similar to that of Figure 7.5. The target-ready, TRDY#, signal is equivalent to the Slave-ready signal in that figure. In addition, PCI uses an initiator-ready signal, IRDY#, to support burst transfers. We will describe these signals briefly, to provide the reader with an appreciation of the main features of the bus.

A complete transfer operation on the PCI bus, involving an address and a burst of data, is called a *transaction*. Consider a bus transaction in which an initiator reads four consecutive 32-bit words from the memory. The sequence of events on the bus is illustrated in Figure 7.19. All signal transitions are triggered by the rising edge of the clock. As in the case of Figure 7.5, we show the signals changing later in the clock cycle to indicate the delays they encounter. A signal whose name ends with the symbol # is asserted when in the low-voltage state.

Table 7.1 Data transfer signals on the PCI bus.

Name	Function
CLK	A 33-MHz or 66-MHz clock
FRAME#	Sent by the initiator to indicate the duration of a transmission
AD	32 address/data lines, which may be optionally increased to 64
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus)
IRDY#, TRDY#	Initiator-ready and Target-ready signals
DEVSEL#	A response from the device indicating that it has recognized its address and is ready for a data transfer transaction
IDSEL#	Initialization Device Select

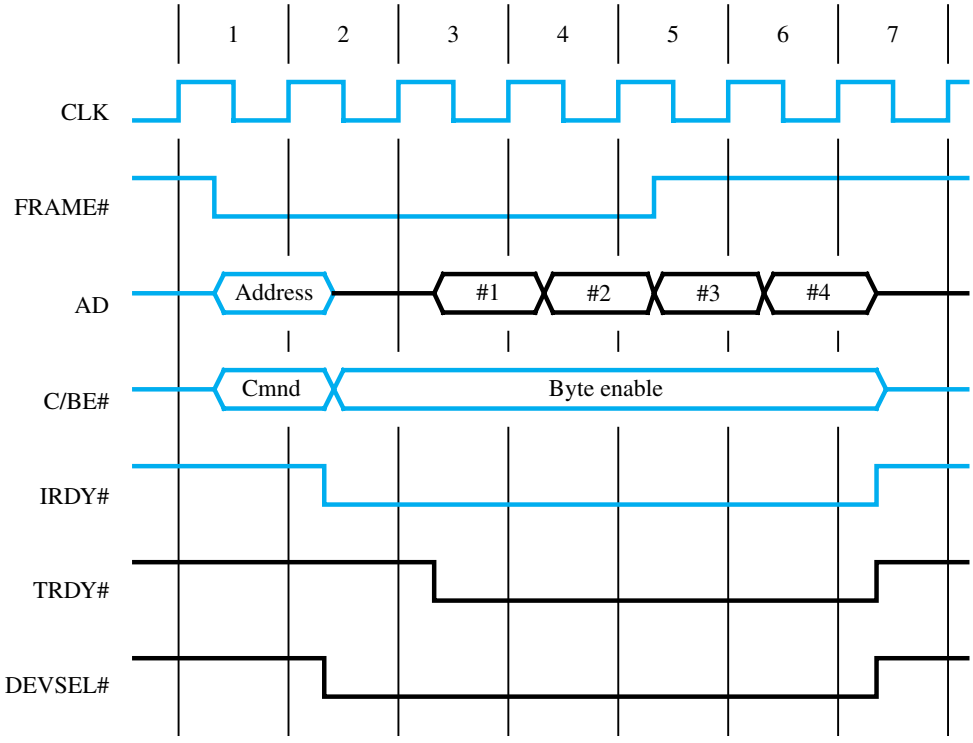


Figure 7.19 A Read operation on the PCI bus.

The bus master, acting as the initiator, asserts FRAME# in clock cycle 1 to indicate the beginning of a transaction. At the same time, it sends the address on the AD lines and a command on the C/BE# lines. In this case, the command will indicate that a Read operation is requested and that the memory address space is being used.

In clock cycle 2, the initiator removes the address, disconnects its drivers from the AD lines, and asserts IRDY# to indicate that it is ready to receive data. The selected target asserts DEVSEL# to indicate that it has recognized its address and is ready to respond. At the same time, it enables its drivers on the AD lines, so that it can send data to the initiator in subsequent cycles. Clock cycle 2 is used to accommodate the delays involved in turning the AD lines around, as the initiator turns its drivers off and the target turns its drivers on. The target asserts TRDY# in clock cycle 3 and begins to send data. It maintains DEVSEL# in the asserted state until the end of the transaction.

We have assumed that the target is ready to send data in clock cycle 3. If not, it would delay asserting TRDY# until it is ready. The entire burst of data need not be sent in successive clock cycles. Either the initiator or the target may introduce a pause by deactivating its ready signal, then asserting it again when it is ready to resume the transfer of data.

The C/BE# lines, which are used to send a bus command in clock cycle 1, are used for a different purpose during the rest of the transaction. Each of these four lines is associated with one byte on the AD lines. The initiator asserts one or more of the C/BE# lines to indicate which byte lines are to be used for transferring data.

The initiator uses the FRAME# signal to indicate the duration of the burst. It deactivates this signal during the second-last word of the transfer. In Figure 7.19, the initiator maintains FRAME# in the asserted state until clock cycle 5, the cycle in which it receives the third word. In response, the target sends one more word in clock cycle 6, then stops. After sending the fourth word, the target deactivates TRDY# and DEVSEL# and disconnects its drivers on the AD lines.

Device Configuration

When an I/O device is connected to a computer, several actions are needed to configure both the device interface and the software that communicates with it. Like USB, PCI has a plug-and-play capability that greatly simplifies this process. In fact, the plug-and-play feature was pioneered by the PCI standard. A PCI interface includes a small configuration ROM memory that stores information about the I/O device connected to it. The configuration ROMs of all devices are accessible in the configuration address space, where they are read by the PCI initialization software whenever the system is powered up or reset. By reading the information in the configuration ROM, the software determines whether the device is a printer, a camera, an Ethernet interface, or a disk controller. It can further learn about various device options and characteristics.

Devices connected to the PCI bus are not assigned permanent addresses that are built into their I/O interface hardware. Instead, device addresses are assigned by software during the initial configuration process. This means that when power is turned on, devices cannot be accessed using their addresses in the usual way, as they have not yet been assigned any address. A different mechanism is used to select I/O devices at that time.

The PCI bus may have up to 21 connectors for I/O device interface cards to be plugged into. Each connector has a pin called Initialization Device Select (IDSEL#). This pin is connected to one of the upper 21 address/data lines, AD11 to AD31. A device interface responds to a configuration command if its IDSEL# input is asserted. The configuration software scans all 21 locations to identify where I/O device interfaces are present. For each location, it issues a configuration command using an address in which the AD line corresponding to that location is set to 1 and the remaining 20 lines are set to 0. If a device interface responds, it is assigned an address and that address is writen into one of its registers designated for this purpose. Using the same addressing mechanism, the processor reads the device's configuration ROM and carries out any necessary initialization. It uses the low-order address bits, AD0 to AD10, to access locations within the configuration ROM. This automated process means that the user simply plugs in the interface board and turns on the power. The software does the rest.

The PCI bus has gained great popularity, particularly in the PC world. It is also used in many other computers, to benefit from the wide range of I/O devices for which a PCI interface is available. Both a 32-bit and a 64-bit configuration are available, using either a 33-MHz or 66-MHz clock. A high-performance variant known as PCI-X is also available. It is a 64-bit bus that runs at 133 MHz. Yet higher performance versions of PCI-X run at speeds up to 533 MHz.

7.5.4 SCSI BUS

The acronym SCSI stands for Small Computer System Interface [4]. It refers to a standard bus defined by the American National Standards Institute (ANSI). The SCSI bus may be used to connect a variety of devices to a computer. It is particularly well-suited for use with disk drives. It is often found in installations such as institutional databases or email systems where many disks drives are used.

In the original specifications of the SCSI standard, devices are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates of up to 5 Megabytes/s. The standard has undergone many revisions, and its data transfer capability has increased rapidly. SCSI-2 and SCSI-3 have been defined, and each has several options. Data are transferred either 8 bits or 16 bits in parallel, using clock speeds of up to 80 MHz. There are also several options for the electrical signaling scheme used. The bus may use single-ended transmission, where each signal uses one wire, with a common ground return for all signals. In another option, differential signaling is used, with a pair of wires for each signal.

Data Transfer

Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus or to the PCI bus. A SCSI bus may be connected directly to the processor bus, or more likely to another standard I/O bus such as PCI, through a SCSI controller. Data and commands are transferred in the form of multi-byte messages called packets. To send commands or data to a device, the processor assembles the information in the memory then instructs the SCSI controller to transfer it to

the device. Similarly, when data are read from a device, the controller transfers the data to the memory and then informs the processor by raising an interrupt.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory. Data are stored on a disk in blocks called *sectors,* where each sector may contain several hundred bytes. When a data file is written on a disk, it is not always stored in contiguous sectors. Some sectors may already contain previously stored information; others may be defective and must be skipped. Hence, a Read or Write request may result in accessing several disk sectors that are not necessarily contiguous. Because of the constraints of the mechanical motion of the disk, there is a long delay, on the order of several milliseconds, before reaching the first sector to or from which data are to be transferred. Then, a burst of data are transferred at high speed. Another delay may ensue to reach the next sector, followed by a burst of data. A single Read or Write request may involve several such bursts. The SCSI protocol is designed to facilitate this mode of operation.

Let us examine a complete Read operation as an example. The following is a simplified high-level description, ignoring details and signaling conventions. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller contends for control of the SCSI bus.
2. When it wins the arbitration process, the SCSI controller sends a command to the disk controller, specifying the required Read operation.
3. The disk controller cannot start to transfer data immediately. It must first move the read head of the disk to the required sector. Hence, it sends a message to the SCSI controller indicating that it will temporarily suspend the connection between them. The SCSI bus is now free to be used by other devices.
4. The disk controller sends a command to the disk drive to move the read head to the first sector involved in the requested Read operation. It reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data, it requests control of the bus. After it wins arbitration, it re-establishes the connection with the SCSI controller, sends the contents of the data buffer, then suspends the connection again.
5. The process is repeated to read and transfer the contents of the second disk sector.
6. The SCSI controller transfers the requested data to the main memory and sends an interrupt to the processor indicating that the data are now available.

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. Messages refer to more complex operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of the disk's operation and how it moves from one sector to the next.

The SCSI bus standard defines a wide range of control messages that can be used to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

7.5.5 SATA

In the early days of the personal computer, the bus of a popular IBM computer called AT, which was based on Intel's 8080 microprocessor bus, became an industry standard. It was named ISA, for Industry Standard Architecture. An enhanced version, including a definition of the basic software needed to support disk drives, was later named ATA, for AT Attachment bus. A serial version of the same architecture became known as SATA [5], which is now widely used as an interface for disks. Like all standards, several versions of SATA have been developed with added features and higher speeds. The original parallel version has been renamed PATA, but it is no longer used in new equipment.

The basic SATA connector has 7 pins, connecting two twisted pairs and three ground wires. Differential transmission is used, with clock frequencies ranging from 1.5 to 6.0 Gigabits/s. Some of the recent versions provide an isochronous transmission feature to support audio and video devices.

7.5.6 SAS

This is a serial implementation of the SCSI bus, hence its name: Serially Attached SCSI [6]. It is primarily intended for connecting magnetic disks and CD and DVD drives. It uses serial, point-to-point links that are similar to SATA. A SAS link can transfer data in both directions simultaneously, at speeds up to 12 Gigabits/s. At the software level, SAS is fully compatible with SCSI.

7.5.7 PCI EXPRESS

The demands placed on I/O interconnections are ever increasing. Internet connections, sophisticated graphics devices, streaming video and high-definition television are examples of applications that involve data transfers at very high speed. The PCI Express interconnection standard (often called PCIe) [7] has been developed to meet these needs and to anticipate further increases in data transfer rates, which are inevitable as new applications are introduced.

PCI Express uses serial, point-to-point links interconnected via switches to form a tree structure, as shown in Figure 7.20. The root node of the tree, called the Root complex, is connected to the processor bus. The Root complex has a special port to connect the main memory. All other connections emanating from the Root complex are serial links to I/O devices. Some of these links may connect to a switch that leads to more serial branches, as shown in the figure. The switch may also connect to bridging interfaces that support other standards, such as PCI or USB. For example, one of the tree branches could be a PCI bus, to take advantage of the wide variety of devices for which PCI interfaces already exist.

The basic PCI Express link consists of two twisted pairs, one for each direction of transmission. Data are transmitted at the rate of of 2.5 Gigabits/s over each twisted pair, using the differential signaling scheme described in Section 7.5.1. Data may be transmitted in both directions at the same time. Also, links to different devices may be carrying data at the same time, because there is no shared bus as in the case of PCI or SCSI. Furthermore,

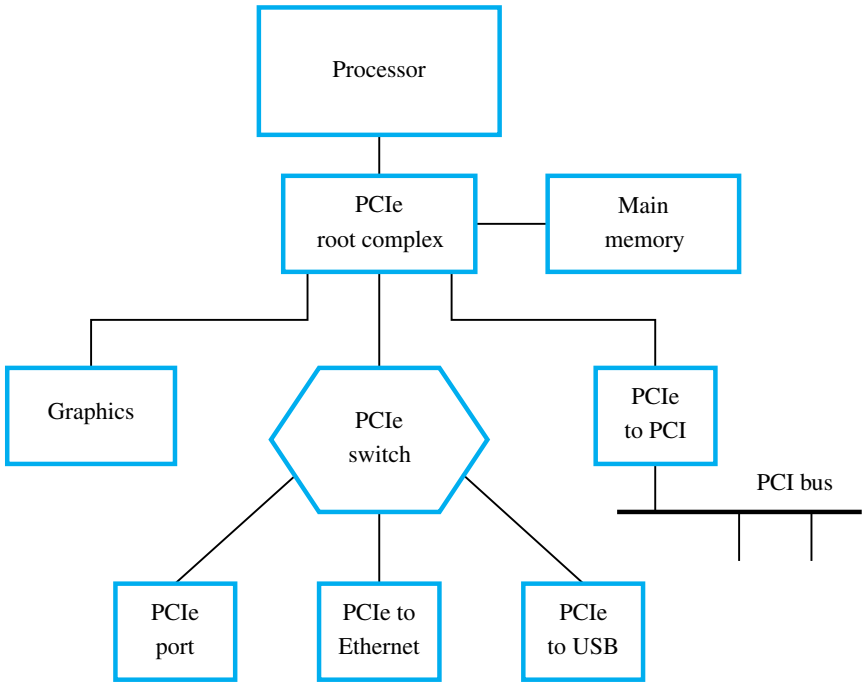


Figure 7.20 PCI Express connections.

a link may use more than one twisted pair in each direction. The basic arrangement with one twisted pair for each direction is called a lane and referred to as a X1 (read as by 1) connection. A link may use 2, 4, 8, or 16 lanes, in which case it is called a X2, X4, X8, or X16 link.

The receiver on a synchronous transmission link must synchronize its clock with that of the sender, as described in Section 7.4.2. To make this possible, the transmitted data are encoded to ensure that 0-to-1 and 1-to-0 transitions occur frequently enough. In the case of PCIe, each 8 bits of data are encoded using 10 bits. Other bits are inserted in the stream to perform various control functions, such as delineating address and data information. After accounting for the additional bits, a single twisted pair on which data are transmitted at 2.5 Gigabits/s actually delivers 1.6 Gigabits/s or 200 MByte/s of useful information. A X16 link transfers data at the rate of 3.2 Gigabyte/s in each direction. By comparison, a 64-bit PCI bus operating at 64 MHz has a peak aggregate data transfer rate of 512 Megabytes/s. PCI Express has the additional advantage of using a small number of wires, resulting in lower-cost hardware.

The PCI Express protocols are fully compatible with those of PCI. For example, the same initial configuration procedures are used. Thus, a computer that uses PCI Express can use existing operating systems and applications software that were developed for a PCI-based system.

7.6 CONCLUDING REMARKS

This chapter introduced the I/O structure of a computer from a hardware point of view. I/O devices connected to a bus are used as examples to illustrate the synchronous and asynchronous schemes for transferring data.

The architecture of interconnection networks for input and output devices has been a major area of development, driven by an ever-increasing need for transferring data at high speed, for reduced cost, and for features that enhance user convenience such as plug-and-play. Several I/O standards are described briefly in this chapter, illustrating the approaches used to meet these objectives. The current trend is to move away from parallel buses to serial point-to-point links. Serial links have lower cost and can transfer data at high speed.

7.7 SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 7.1

Problem: The I/O bus of a computer uses the synchronous protocol shown in Figure 7.4. Maximum propagation delay on this bus is 4 ns. The bus master takes 1.5 ns to place an address on the address lines. Slave devices require 3 ns to decode the address and a maximum of 5 ns to place the requested data on the data lines. Input registers connected to the bus have a minimum setup time of 1 ns. Assume that the bus clock has a 50% duty cycle; that is, the high and low phases of the clock are of equal duration. What is the maximum clock frequency for this bus?

Solution: The minimum time for the high phase of the clock is the time for the address to arrive and be decoded by the slave, which is $1.5 + 4 + 3 = 8.5$ ns. The minimum time for the low phase of the clock is the time for the slave to place data on the bus and for the master to load the data into a register, which is $5 + 4 + 1 = 10$ ns. Then, the minimum clock period is $2 \times 10 = 20$ ns, and the maximum clock frequency is 50 MHz.

Example 7.2

Problem: An arbiter receives three request signals, R1, R2, R3, and generates three grant signals, G1, G2, G3. Request R1 has the highest priority and request R3 the lowest priority. An example of the operation of such an arbiter is given in Figure 7.9. Give a state diagram that describes the behavior of this arbiter.

Solution: A state diagram is given in Figure 7.21. The arbiter starts in the idle state, A. When one or more of the request signals is asserted, the arbiter moves to one of the three states, B, C, or D, depending on which of the active requests has the highest priority. When it enters the new state, it asserts the corresponding grant signal. The arbiter remains in that state

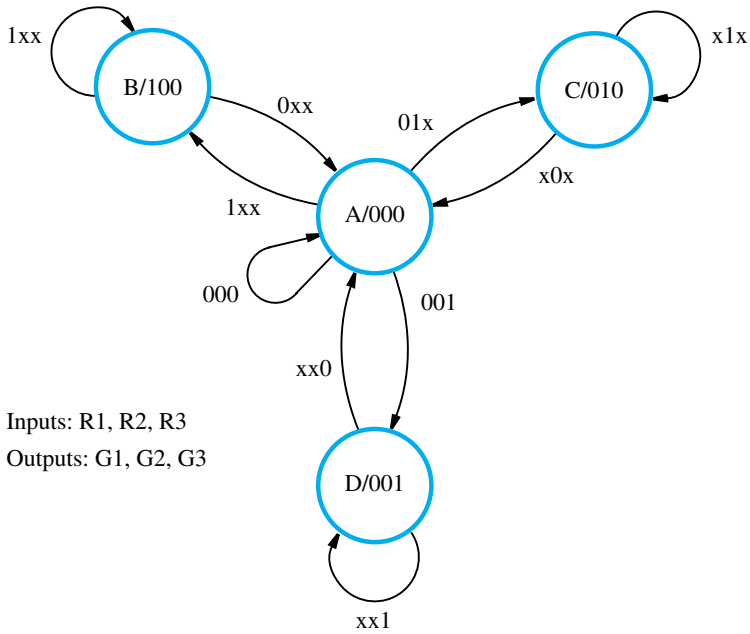


Figure 7.21 State diagram for Example 7.2.

until the device being served drops its request, at which time the arbiter returns to state A. Once it is back in state A, it will respond to any request that may be active at that time, or wait for a new request to be asserted.

Example 7.3

Problem: Design an output interface circuit for a synchronous bus that uses the protocol of Figure 7.4. When data are written into the data register of this interface, the interface sends a pulse with a width of one clock cycle on a line called New-data. This pulse lets the output device connected to the interface know that new data are available.

Solution: All events in a synchronous circuit are driven by a clock signal. A possible circuit for the interface is shown in Figure 7.22. The Write-data signal enables the data register, and data are loaded into it at the clock edge at the end of the clock cycle. At the same time, the New-data flip-flop is set to 1. The feedback connection from the $\overline{Q}$ output of the flip-flop clears the flip-flop to 0 on the following clock edge.

Example 7.4

Problem: Draw a state diagram for a finite-state machine (FSM) that represents the behavior of the handshake control circuit in Figure 7.14.

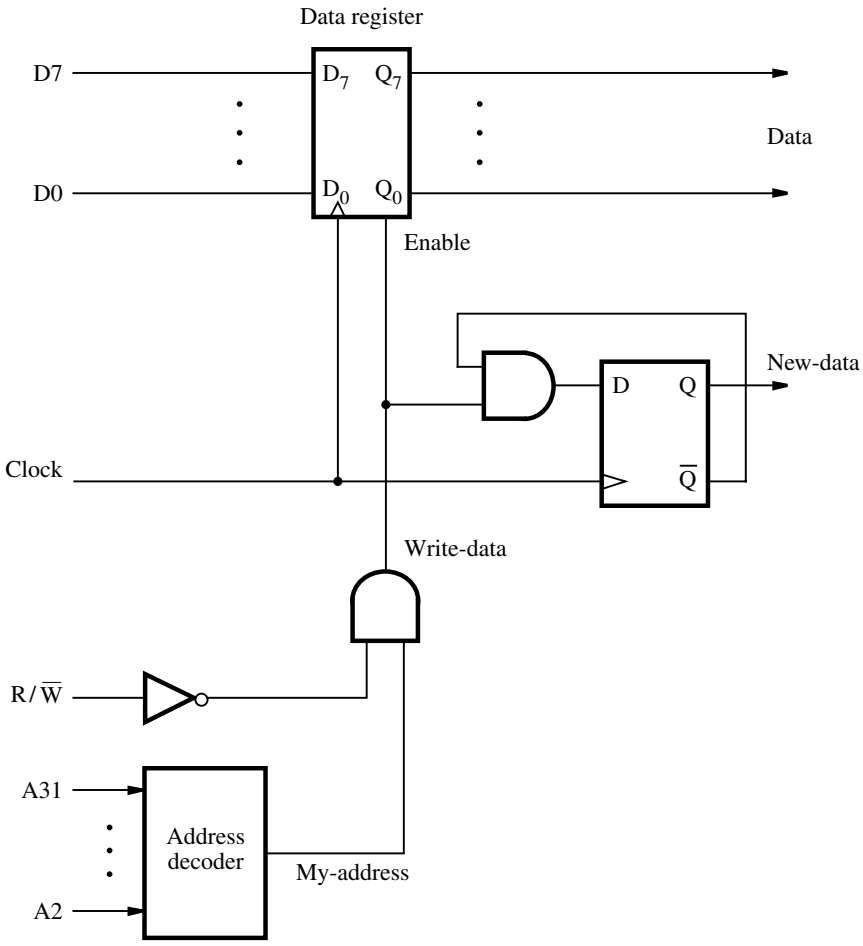


Figure 7.22 A synchronous output interface circuit for Example 7.3.

Solution: A state diagram is given in Figure 7.23. The circuit starts in state A, with the display device ready to receive new data. Thus, New-data = 0 and DOUT = 1. A Write operation causes Write-data to change to 1. This causes the state machine to move to state B, and its outputs change to 10. The machine stays in state B until Ready drops to 0, indicating that the display device recognized that new data are available. When that happens, the machine moves to state C to wait for the display to become ready again. It must also wait for Write-data to return to zero, if it has not done so already.

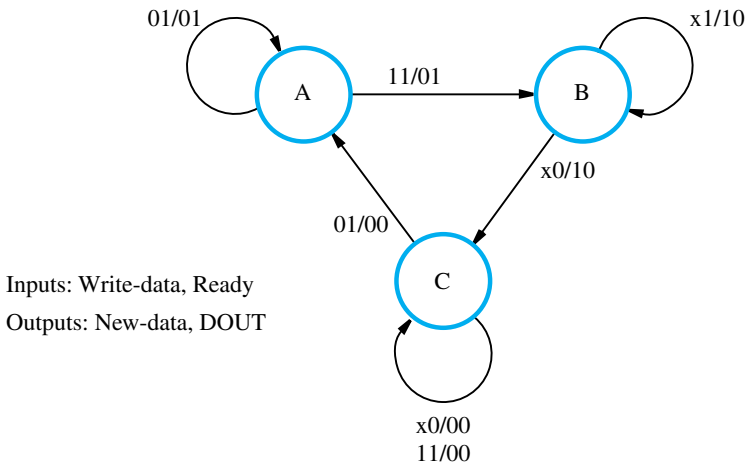


Figure 7.23 State diagram for Example 7.4.

PROBLEMS

7.1 **[E]** The input status bit in an interface circuit, which indicates that new data are available, is cleared as soon as the input data register is read. Why is this important?

7.2 **[E]** The address bus of a computer has 16 address lines, A_{15-0}. If the hexadecimal address assigned to one device is 7CA4 and the address decoder for that device ignores lines A_8 and A_9, what are all the addresses to which this device will respond?

7.3 **[M]** A processor has seven interrupt-request lines, INTR1 to INTR7. Line INTR7 has the highest priority and INTR1 the lowest priority. Design a priority encoder circuit that generates a 3-bit code representing the request with the highest priority.

7.4 **[M]** Figures 7.4, 7.5, and 7.6 show three protocols for transferring data between a master and a slave. What happens in each case if the addressed device does not respond due to a malfunction during a Read operation? What problems would this cause and what remedies are possible?

7.5 **[E]** In the timing diagram in Figure 7.5, the processor maintains the address on the bus until it receives a response from the device. Is this necessary? What additions are needed on the device side if the processor sends an address for one cycle only?

7.6 **[E]** How is the timing diagram in Figure 7.6 affected as the distance between the processor and the I/O device increases? How is increased distance accommodated in the case of Figure 7.4?

7.7 **[E]** Consider a synchronous bus that operates according to the timing diagram in Figure 7.5. The bus and the interface circuitry connected to it have the following parameters:

Bus driver delay	2 ns
Propagation delay on the bus	5 to 10 ns
Address decoder delay	6 ns
Time to fetch the requested data	0 to 25 ns
Setup time	1.5 ns

(*a*) What is the maximum clock speed at which this bus can operate?

(*b*) How many clock cycles are needed to complete an input operation?

7.8 **[M]** Consider the asynchronous bus protocol shown in Figure 7.6. Using the same parameters as in Problem 7.7, what are the minimum and maximum times to complete one bus transfer? Allow 1 ns for bus skew.

7.9 **[M]** The asynchronous bus protocol in Figure 7.6 uses a full-handshake, in which the master maintains an asserted signal on Master-ready until it receives Slave-ready, the slave keeps Slave-ready asserted until Master-ready becomes inactive, and so on. Consider an alternative protocol in which each of these signals is a pulse of a fixed width of 4 ns. Devices take action only on the rising edge of the pulse. Using the same parameters as in Problem 7.7, what is the minimum and maximum time to complete one bus transfer?

7.10 **[M]** In the arbiter protocol example depicted in Figure 7.9, the master that receives a bus grant maintains its request line in the asserted state until it is ready to relinquish bus mastership. Assume that a common line called Busy is available, which is asserted by the master that is currently using the bus. The arbiter grants the bus only when Busy is inactive. Once a master receives a grant, it asserts Busy and drops its request, and in response the arbiter drops the grant. The master deactivates Busy when it is finished using the bus. Draw a timing diagram equivalent to Figure 7.9 for this mode of operation.

7.11 **[M]** Modify the state diagram given in Example 7.2 for the mode of operation described in Problem 7.10.

7.12 **[D]** The arbiter of Example 7.2 controls access to a common resource. It does not allow preemption. This means that if a high-priority request is received after a lower-priority request has been granted, it must wait until service to the device that is currently using the common resource is completed. In some cases, it is desirable to allow preemption, to provide service to a high-priority device more quickly. Devices in such a system, must be able to stop and relinquish the use of the common resource when asked to do so by the arbiter. This must be done in a safe manner. A device that is using the resource must be allowed to reach a safe point at which service can be terminated. It would then signal to the arbiter that it has stopped using the resource.

(*a*) Suggest a suitable modification to the signaling protocol that enables the service in progress to be terminated safely.

(*b*) Modify the state diagram of the arbiter to implement the revised protocol.

7.13 **[E]** An arbiter controls access to a common resource. It uses a rotating-priority scheme in responding to requests on lines R1 through R4. Initially, R1 has the highest priority and R4 the lowest priority. After a request on one of the lines receives service, that line drops to the lowest priority, and the next line in sequence becomes the highest-priority line. For example, after R2 has been serviced, the priority order, starting with the highest, becomes R3, R4, R1, R2. What will be the sequence of grants for the following sequence of requests: R3, R1, R4, R2? Assume that the last three requests arrive while the first one is being serviced.

7.14 **[E]** Consider an arbiter that uses the priority scheme described in Problem 7.13. What happens if one device requests service repeatedly. Compare the behavior of this arbiter to one that uses a fixed-priority scheme.

7.15 **[E]** Give the logic expression for an address decoder that recognizes the 16-bit hexadecimal address FA68.

7.16 **[M]** An industrial plant uses several sensors to monitor temperature, pressure, and other factors. Each sensor includes a switch that moves to the ON position when the corresponding parameter exceeds a preset limit. Eight such sensors need to be connected to the bus of a 16-bit computer. Design an appropriate interface to enable the state of all eight switches to be read simultaneously as a single byte. Assume the bus is synchronous and that it uses the timing sequence of Figure 7.4.

7.17 **[E]** The bus protocol of Figure 7.4 specifies that the slave device should send its data only in the second phase of the clock.

(*a*) It is possible that some device may recognize its address and is ready to send data sooner. Why is it not allowed to do so? Would the processor receive wrong data?

(*b*) Would any other problem arise?

7.18 **[M]** Data are stored in a small memory in an input interface connected to a synchronous bus that uses the protocol of Figure 7.5. Read and Write operations on the bus are indicated by a Command line called R/$\overline{\text{W}}$. The speed of the memory is such that two clock cycles are required to read data from the memory. Design a circuit to generate the Slave-ready response of this interface.

7.19 **[E]** Each of the two signals DEVSEL# and TRDY# of the PCI protocol in Figure 7.19 represents a response from the initiator. How do the functions of these two signals differ?

7.20 **[E]** Consider the data transfer operation shown in Figure 7.19 for the PCI bus. How would this bus protocol handle a situation in which the target needs a delay of two clock cycles between words 2 and 3?

7.21 **[E]** Draw a timing diagram for transferring three words to an output device connected to the PCI bus.

REFERENCES

1. *Universal Serial Bus Specification,* available at www.usb.org/developers.
2. *IEEE Standard for a High-Performance Serial Bus*, IEEE Std. 1394-2008, October 2008.
3. Specifications and other information about the *PCI Local Bus* and *PCI Express* are available at www.pcisig.com/developers.
4. *SCSI-3 Architecture Model (SAM),* ANSI Standard X3.270, 1996. This and other SCSI documents are available on the web at www.ansi.org.
5. *SATA* specifications and related material are available at www.serialata.org.
6. Information about the *Serial SCSI (SAS)* standard is available at www.scsita.org.
7. A. Wilen, J. Schade, and R. Thornburg, *Introduction to PCI Express, A Hardware and Software Developer's Guide,* Intel Press, 2003.