2.5 In Figure 2.3, indicate the width, in bits, of each data path (e.g., between AC and ALU).

2.6 In the IBM 360 Models 65 and 75, addresses are staggered in two separate main memory units (e.g., all even-numbered words in one unit and all odd-numbered words in another). What might be the purpose of this technique?

2.7 With reference to Table 2.4, we see that the relative performance of the IBM 360 Model 75 is fifty times that of the 360 Model 30, yet the instruction cycle time is only five times as fast. How do you account for this discrepancy?

2.8 While browsing at Billy Bob's computer store, you overhear a customer asking Billy Bob what is the fastest computer in the store that he can buy. Billy Bob replies, "You're looking at our Macintoshes. The fastest Mac we have runs at a clock speed of 1.2 gigahertz. If you really want the fastest machine, you should buy our 2.4-gigahertz Intel Pentium IV instead." Is Billy Bob correct? What would you say to help this customer?

2.9 The ENIAC was a decimal machine, where a register was represented by a ring of 10 vacuum tubes. At any time, only one vacuum tube was in the ON state, representing one of the 10 digits. Assuming that ENIAC had the capability to have multiple vacuum tubes in the ON and OFF state simultaneously, why is this representation "wasteful" and what range of integer values could we represent using the 10 vacuum tubes?

2.10 A processor is driven by a clock with a constant frequency f or, equivalently, a constant cycle time τ, where $\tau = 1/f$. The size of a program can be measured by the number of machine instructions, or instruction count I_c, that comprise the program. Different machine instructions may require different number of clock cycles to execute. An important parameter is the average cycles per instruction *CPI* for a program. The processor time T needed to execute a given program can be expressed as:

$$T = I_c \times CPI \times \tau$$

We can refine this formulation by recognizing that during the execution of an instruction, part of the work is done by the processor, and part of the time a word is being transferred to or from memory. In this latter case, the time to transfer depends on the memory cycle time, which may be greater than the processor cycle time. We can rewrite the preceding equation as:

$$T = I_c \times [p + (m \times k)] \times \tau$$

where p is the number of processor cycles needed to decode and execute the instruction, m is the number of memory references needed, and k is the ratio between memory cycle time and processor cycle time. The five performance factors in the preceding equation (I_c, p, m, k, τ) are influenced by four system attributes: the design of the instruction set (known as *instruction set architecture*), compiler technology (how effective the compiler is in producing an efficient machine language program from a high-level language program), processor implementation, and cache and memory hierarchy. Prepare a matrix in which one dimension shows the five performance factors and the other dimension shows the four system attributes. Put an X in each cell in which a system attribute affects a performance factor.

2.11 A common measure of performance for a processor is the rate at which instructions are executed, expressed as millions of instructions per second (MIPS). Express the MIPS rate in terms of the clock rate and *CPI*.

2.12 Early examples of CISC and RISC design are the VAX 11/780 and the IBM RS/6000, respectively. Using a typical benchmark program, the following machine characteristics result:

Processor	Clock Frequency	Performance	CPU Time
VAX 11/780	5 MHz	1 MIPS	12 x seconds
IBM RS/6000	25 MHz	18 MIPS	x seconds

The final column shows that the VAX required 12 times longer than the IBM measured in CPU time.

a. What is the relative size of the instruction count of the machine code for this benchmark program running on the two machines?
b. What are the *CPI* values for the two machines?

2.13 A benchmark program is run on a 40 MHz processor. The object code consists of 100,000 instructions, with the following instruction mix and clock cycle count:

Instruction Type	Instruction Count	Clock Cycle Count
Integer arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2

Determine the effective CPI, MIPS rate, and execution time for this program.

2.14 To obtain a reliable comparison of the performance of various computers, it is preferable to run a number of different benchmark programs on each machine, and then average the results. For example, if m different benchmark program, then a simple arithmetic mean can be calculated as follows:

$$R_a = \frac{1}{m}\sum_{i=1}^{m} R_i$$

where R_i is the MIPS rating for the ith benchmark. An alternative is to take the harmonic mean:

$$R_h = \frac{m}{\sum_{i=1}^{m} \frac{1}{R_i}}$$

a. Comment on the relative advantage or disadvantage of each method. Hint: consider the mean execution time (in microseconds) per instruction for program i, $T_i = 1/R_i$.
b. Four benchmark programs are executed on three computers with the following results:

	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Program 3	500	1000	50
Program 4	100	800	100

The table shows the execution time in seconds, with 100,000,000 instructions executed in each of the four programs. Calculate the arithmetic and harmonic means, and rank the computers based on harmonic mean.

PART TWO

The Computer System

ISSUES FOR PART TWO

A computer system consists of a processor, memory, I/O, and the interconnections among these major components. With the exception of the processor, which is sufficiently complex to devote Part Three to its study, Part Two examines each of these components in detail.

ROAD MAP FOR PART TWO

Chapter 3 A Top-Level View of Computer Function and Interconnection

At a top level, a computer consists of a processor, memory, and I/O components. The functional behavior of the system consists of the exchange of data and control signals among these components. To support this exchange, these components must be interconnected. Chapter 3 begins with a brief examination of the computer's components and their input–output requirements. The chapter then looks at key issues that affect interconnection design, especially the need to support interrupts. The bulk of the chapter is devoted to a study of the most common approach to interconnection: the use of a structure of buses.

Chapter 4 Cache Memory

Computer memory exhibits a wide range of type, technology, organization, performance, and cost. The typical computer system is equipped with a hierarchy of memory subsystems, some internal (directly accessible by the processor) and some external (accessible by the processor via an I/O module). Chapter 4 begins with an overview of this hierarchy. Next, the chapter deals in detail with the design of cache memory, including separate code and data caches and two-level caches.

Chapter 5 Internal Memory

The design of a main memory system is a never-ending battle among three competing design requirements: large storage capacity, rapid access time, and low cost. As memory technology evolves, each of these three characteristics is changing, so that the design decisions in organizing main memory must be revisited anew with each new implementation. Chapter 5 focuses on design issues related to internal memory. First, the nature and organization of semiconductor main memory is examined. Then, recent advanced DRAM memory organizations are explored.

Chapter 6 External Memory

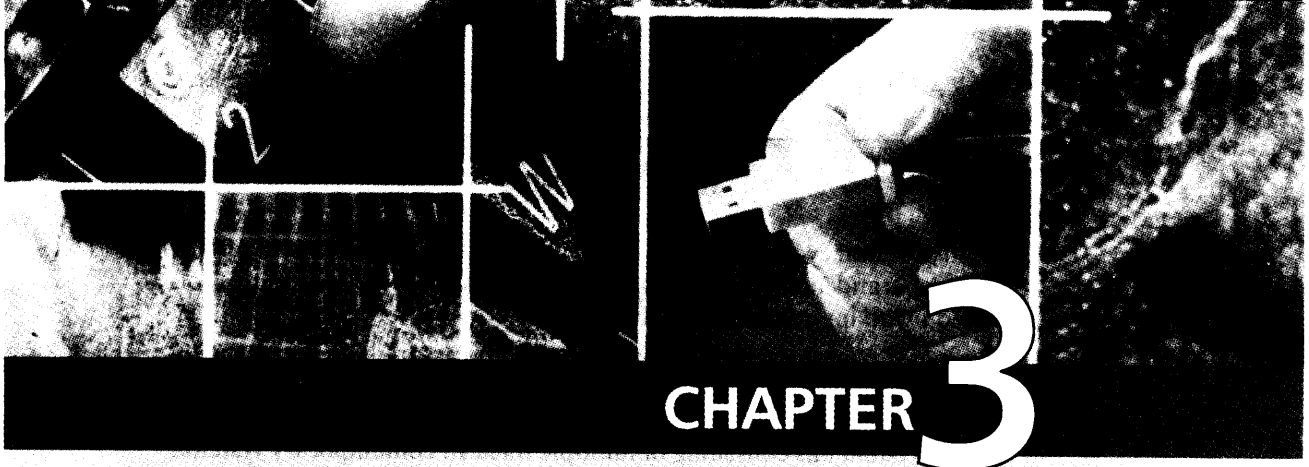
For truly large storage capacity and for more permanent storage than is available with main memory, an external memory organization is needed. The most widely used type of external memory is magnetic disk, and much of Chapter 6 concentrates on this topic. First, we look at magnetic disk technology and design considerations. Then, we look at the use of RAID organization to improve disk memory performance. Chapter 6 also examines optical and tape storage.

Chapter 7 Input/Output

I/O modules are interconnected with the processor and main memory, and each controls one or more external devices. Chapter 7 is devoted to the various aspects of I/O organization. This is a complex area, and less well understood than other areas of computer system design in terms of meeting performance demands. Chapter 7 examines the mechanisms by which an I/O module interacts with the rest of the computer system, using the techniques of programmed I/O, interrupt I/O, and direct memory access (DMA). The interface between an I/O module and external devices is also described.

Chapter 8 Operating System Support

A detailed examination of operating systems is beyond the scope of this book. However, it is important to understand the basic functions of an operating system and how the OS exploits hardware to provide the desired performance. Chapter 8 describes the basic principles of operating systems and discusses the specific design features in the computer hardware intended to provide support for the operating system. The chapter begins with a brief history, which serves to identify the major types of operating systems and to motivate their use. Next, multiprogramming is explained by examining the long-term and short-term scheduling functions. Finally, an examination of memory management includes a discussion of segmentation, paging, and virtual memory.



CHAPTER 3

A TOP-LEVEL VIEW OF COMPUTER FUNCTION AND INTERCONNECTION

KEY POINTS

- **An instruction cycle consists of an instruction fetch, followed by zero or more operand fetches, followed by zero or more operand stores, followed by an interrupt check (if interrupts are enabled).**
- **The major computer system components (processor, main memory, I/O modules) need to be interconnected in order to exchange data and control signals. The most popular means of interconnection is the use of a shared system bus consisting of multiple lines. In contemporary systems, there typically is a hierarchy of buses to improve performance.**
- **Key design elements for buses include arbitration (whether permission to send signals on bus lines is controlled centrally or in a distributed fashion); timing (whether signals on the bus are synchronized to a central clock or are sent asynchronously based on the most recent transmission); and width (number of address lines and number of data lines).**

At a top level, a computer consists of CPU (central processing unit), memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the basic function of the computer, which is to execute programs. Thus, at a top level, we can describe a computer system by (1) describing the external behavior of each component, that is, the data and control signals that it exchanges with other components; and (2) describing the interconnection structure and the controls required to manage the use of the interconnection structure.

This top-level view of structure and function is important because of its explanatory power in understanding the nature of a computer. Equally important is its use to understand the increasingly complex issues of performance evaluation. A grasp of the top-level structure and function offers insight into system bottlenecks, alternate pathways, the magnitude of system failures if a component fails, and the ease of adding performance enhancements. In many cases, requirements for greater system power and fail-safe capabilities are being met by changing the design rather than merely increasing the speed and reliability of individual components.

This chapter focuses on the basic structures used for computer component interconnection. As background, the chapter begins with a brief examination of the basic components and their interface requirements. Then a functional overview is provided. We are then prepared to examine the use of buses to interconnect system components.

3.1 COMPUTER COMPONENTS

As discussed in Chapter 2, virtually all contemporary computer designs are based on concepts developed by John von Neumann at the Institute for Advanced Studies, Princeton. Such a design is referred to as the *von Neumann architecture* and is based on three key concepts:

- Data and instructions are stored in a single read–write memory.
- The contents of this memory are addressable by location, without regard to the type of data contained there.
- Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

The reasoning behind these concepts was discussed in Chapter 2 but is worth summarizing here. There is a small set of basic logic components that can be combined in various ways to store binary data and to perform arithmetic and logical operations on that data. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting "program" is in the form of hardware and is termed a *hardwired program.*

Now consider this alternative. Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware. In the original case of customized hardware, the system accepts data and produces results (Figure 3.1a). With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, the programmer merely needs to supply a new set of control signals.

How shall control signals be supplied? The answer is simple but subtle. The entire program is actually a sequence of steps. At each step, some arithmetic or logical operation is performed on some data. For each step, a new set of control signals is needed. Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code and generate control signals (Figure 3.1b).

Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes. Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a sequence of codes or instructions is called *software.*

Figure 3.1b indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU. Several other components are needed to yield a functioning computer. Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system. A means of reporting results is needed, and this is in the form of an output module. Taken together, these are referred to as *I/O components.*

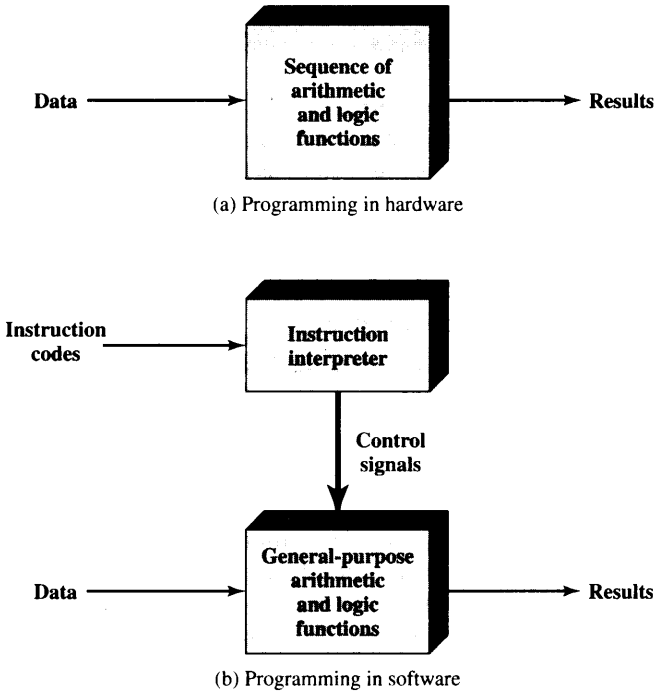


Figure 3.1 Hardware and Software Approaches

One more component is needed. An input device will bring instructions and data in sequentially. But a program is not invariably executed sequentially; it may jump around (e.g., the IAS jump instruction). Similarly, operations on data may require access to more than just one element at a time in a predetermined sequence. Thus, there must be a place to store temporarily both instructions and data. That module is called *memory*, or *main memory* to distinguish it from external storage or peripheral devices. Von Neumann pointed out that the same memory could be used to store both instructions and data.

Figure 3.2 illustrates these top-level components and suggests the interactions among them. The CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers: a memory address register (MAR), which specifies the address in memory for the next read or write, and a memory buffer register (MBR), which contains the data to be written into memory or receives the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer (I/OBR) register is used for the exchange of data between an I/O module and the CPU.

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

Having looked briefly at these major components, we now turn to an overview of how these components function together to execute programs.

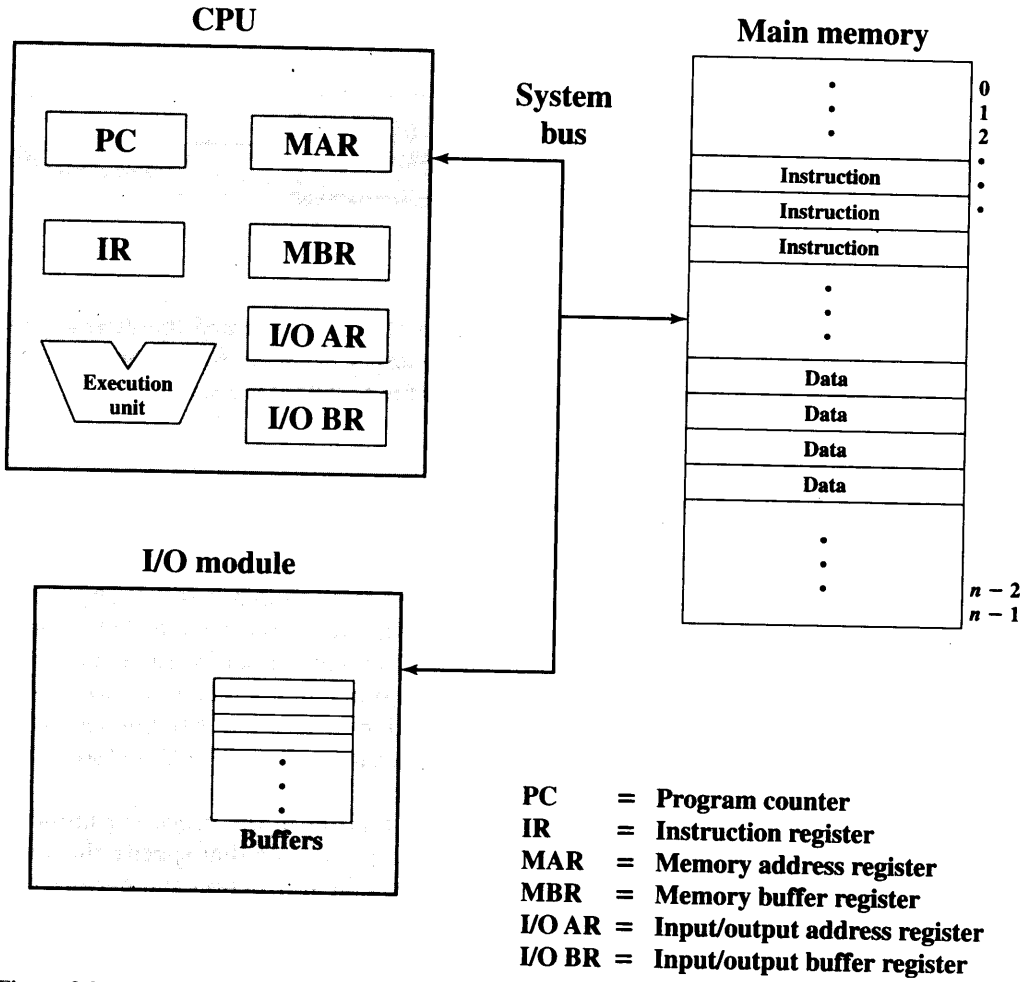


Figure 3.2 Computer Components: Top-Level View

3.2 COMPUTER FUNCTION

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. This section provides an overview of the key elements of program execution. In its simplest form, instruction processing consists of two steps: The processor reads (*fetches*) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction (see, for example, the lower portion of Figure 2.4).

The processing required for a single instruction is called an *instruction cycle*. Using the simplified two-step description given previously, the instruction cycle is depicted

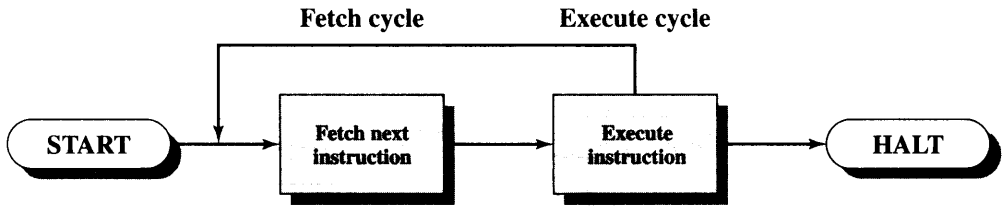


Figure 3.3 Basic Instruction Cycle

in Figure 3.3. The two steps are referred to as the *fetch cycle* and the *execute cycle*. Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

Instruction Fetch and Execute

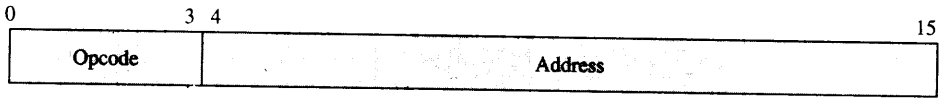
At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained presently.

The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

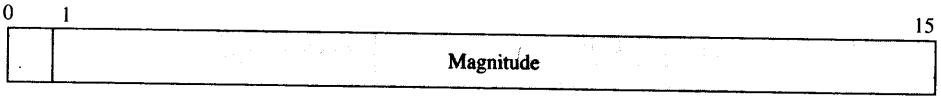
- **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor will remember this fact by setting the program counter to 182. Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150.

An instruction's execution may involve a combination of these actions.

Consider a simple example using a hypothetical machine that includes the characteristics listed in Figure 3.4. The processor contains a single data register, called an accumulator (AC). Both instructions and data are 16 bits long. Thus, it is convenient to organize memory using 16-bit words. The instruction format provides



(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 3.4 Characteristics of a Hypothetical Machine

4 bits for the opcode, so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ (4K) words of memory can be directly addressed.

Figure 3.5 illustrates a partial program execution, showing the relevant portions of memory and processor registers.[1] The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute cycles, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR and the PC is incremented. Note that this process involves the use of a memory address register (MAR) and a memory buffer register (MBR). For simplicity, these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
6. The contents of the AC are stored in location 941.

[1]Hexadecimal notation is used, in which each digit represents 4 bits. This is the most convenient notation for representing the contents of memory and registers when the word length is a multiple of 4. See Appendix A for a basic refresher on number systems (decimal, binary, hexadecimal).

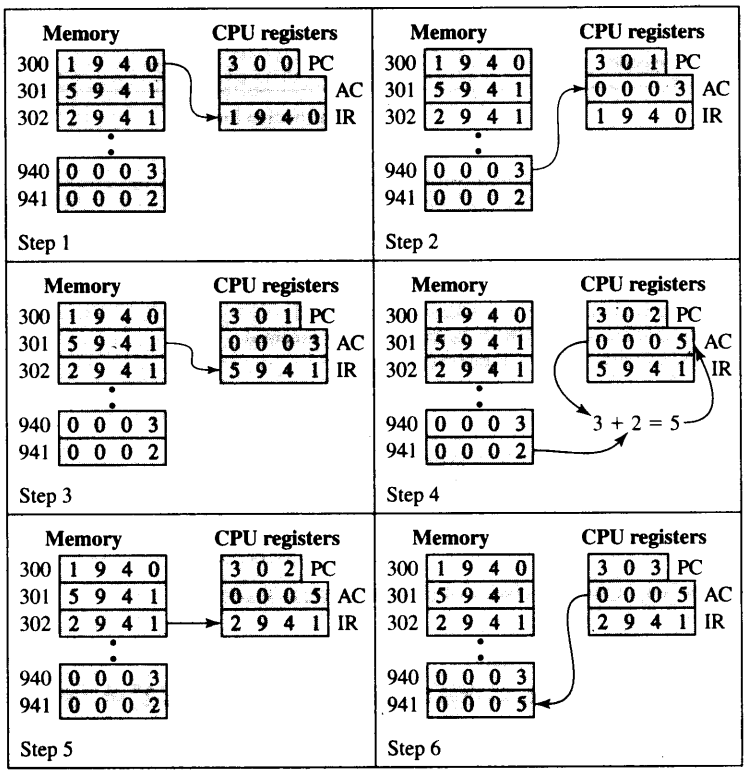


Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer cycles would be needed. Some older processors, for example, included instructions that contain more than one memory address. Thus the execution cycle for a particular instruction on such processors could involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.

For example, the PDP-11 processor includes an instruction, expressed symbolically as ADD B,A, that stores the sum of the contents of memory locations B and A into memory location A. A single instruction cycle with the following steps occurs:

- Fetch the ADD instruction.
- Read the contents of memory location A into the processor.
- Read the contents of memory location B into the processor. In order that the contents of A are not lost, the processor must have at least two registers for storing memory values, rather than a single accumulator.
- Add the two values.
- Write the result from the processor to memory location A.

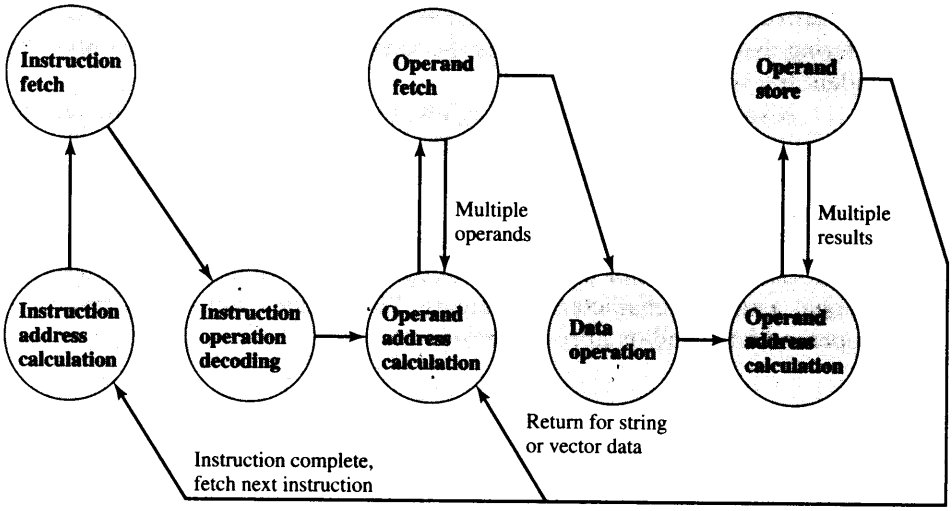


Figure 3.6 Instruction Cycle State Diagram

Thus, the execution cycle for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations in mind, Figure 3.6 provides a more detailed look at the basic instruction cycle of Figure 3.3. The figure is in the form of a state diagram. For any given instruction cycle, some states may be null and others may be visited more than once. The states can be described as follows:

- **Instruction address calculation (iac):** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.
- **Instruction fetch (if):** Read instruction from its memory location into the processor.
- **Instruction operation decoding (iod):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- **Operand fetch (of):** Fetch the operand from memory or read it in from I/O.
- **Data operation (do):** Perform the operation indicated in the instruction.
- **Operand store (os):** Write the result into memory or out to I/O.

States in the upper part of Figure 3.6 involve an exchange between the processor and either memory or an I/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because

an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the same in both cases, and so only a single state identifier is needed.

Also note that the diagram allows for multiple operands and multiple results, because some instructions on some machines require this. For example, the PDP-11 instruction ADD A,B results in the following sequence of states: iac, if, iod, oac, of, oac, of, do, oac, os.

Finally, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional array) of characters. As Figure 3.6 indicates, this would involve repetitive operand fetch and/or store operations.

Interrupts

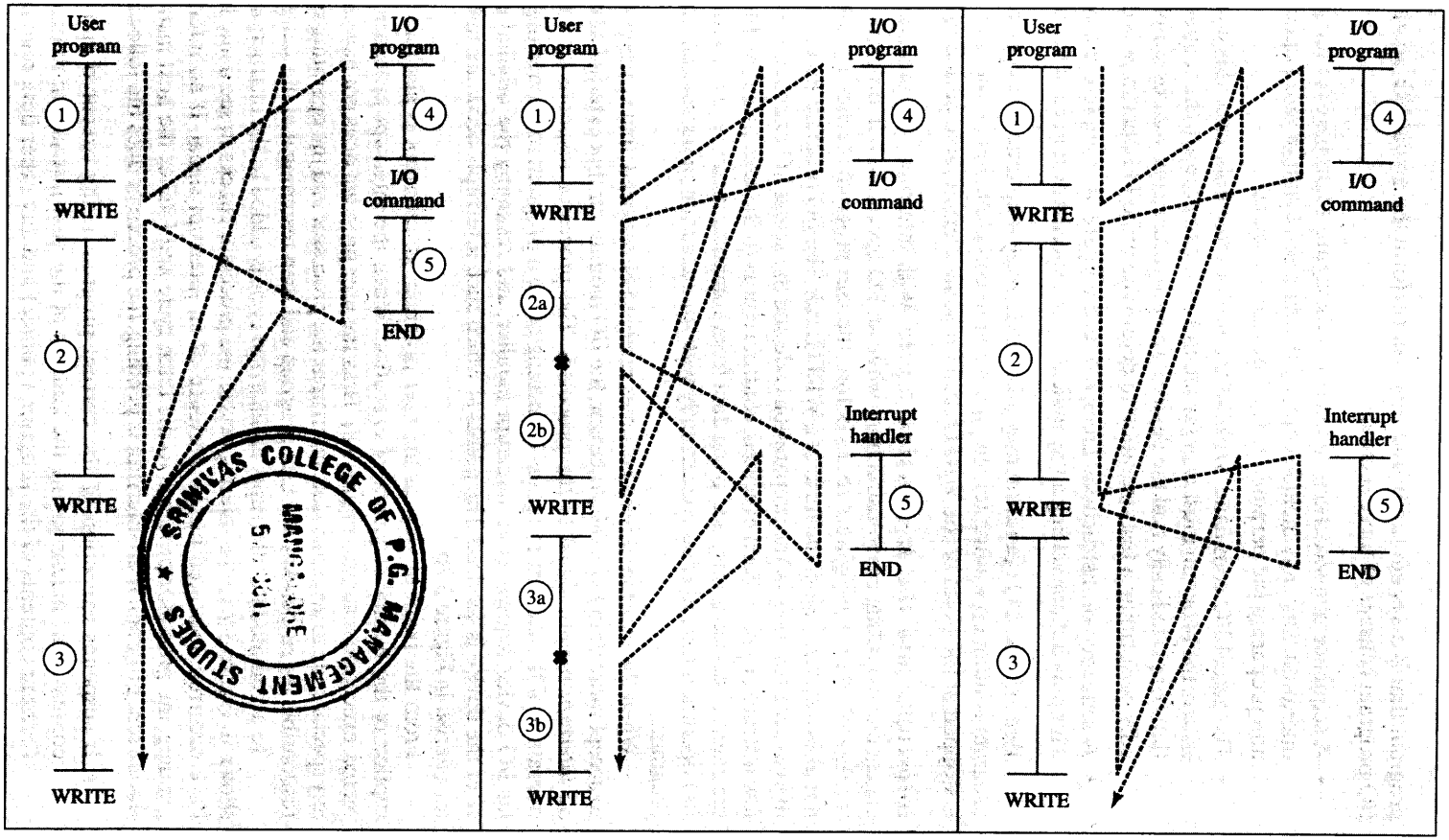
Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal processing of the processor. Table 3.1 lists the most common classes of interrupts. The specific nature of these interrupts is examined later in this book, especially in Chapters 7 and 12. However, we need to introduce the concept now to understand more clearly the nature of the instruction cycle and the implications of interrupts on the interconnection structure. The reader need not be concerned at this stage about the details of the generation and processing of interrupts, but only focus on the communication between modules that results from interrupts.

Interrupts are provided primarily as a way to improve processing efficiency. For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 3.3. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor.

Figure 3.7a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O

Table 3.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure such as power failure or memory parity error.



(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

Figure 3.7 Program Flow of Control without and with Interrupts

program that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE call for some considerable period of time.

Interrupts and the Instruction Cycle With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 3.7b. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an *interrupt request* signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service that particular I/O device, known as an interrupt handler, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk in Figure 3.7b.

From the point of view of the user program, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 3.8). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.

To accommodate interrupts, an *interrupt cycle* is added to the instruction cycle, as shown in Figure 3.9. In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
- It sets the program counter to the starting address of an *interrupt handler* routine.

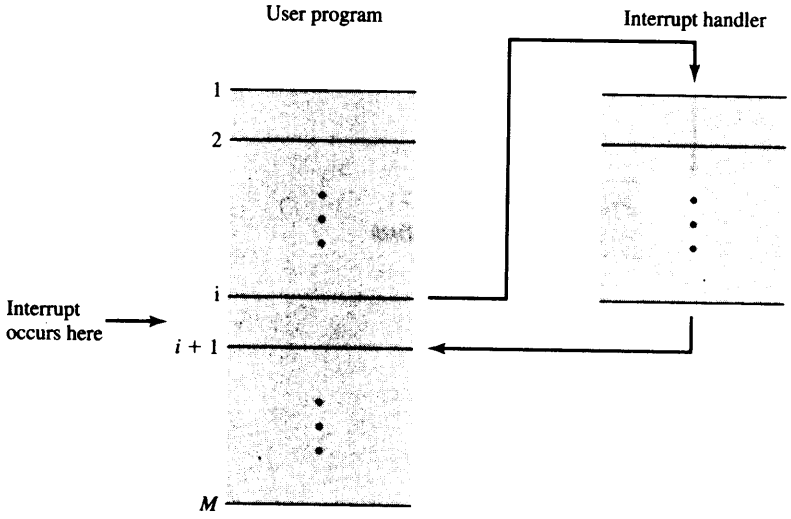


Figure 3.8 Transfer of Control via Interrupts

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt and to decide on the appropriate action. Nevertheless, because of the relatively large

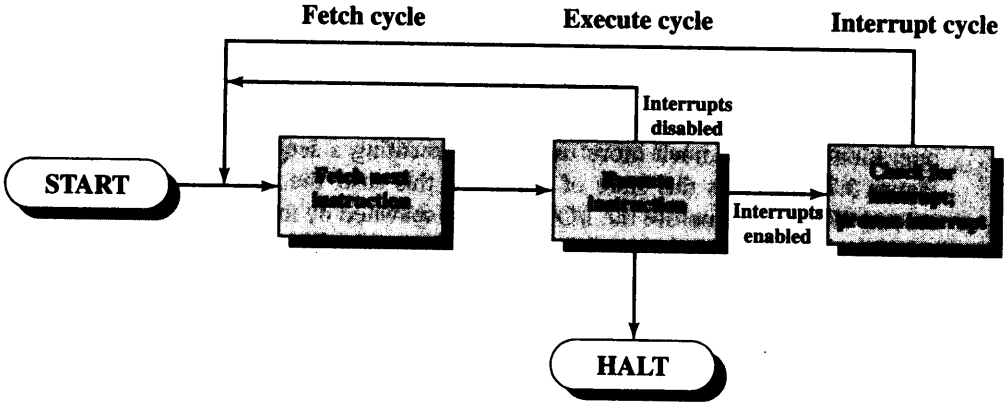


Figure 3.9 Instruction Cycle with Interrupts

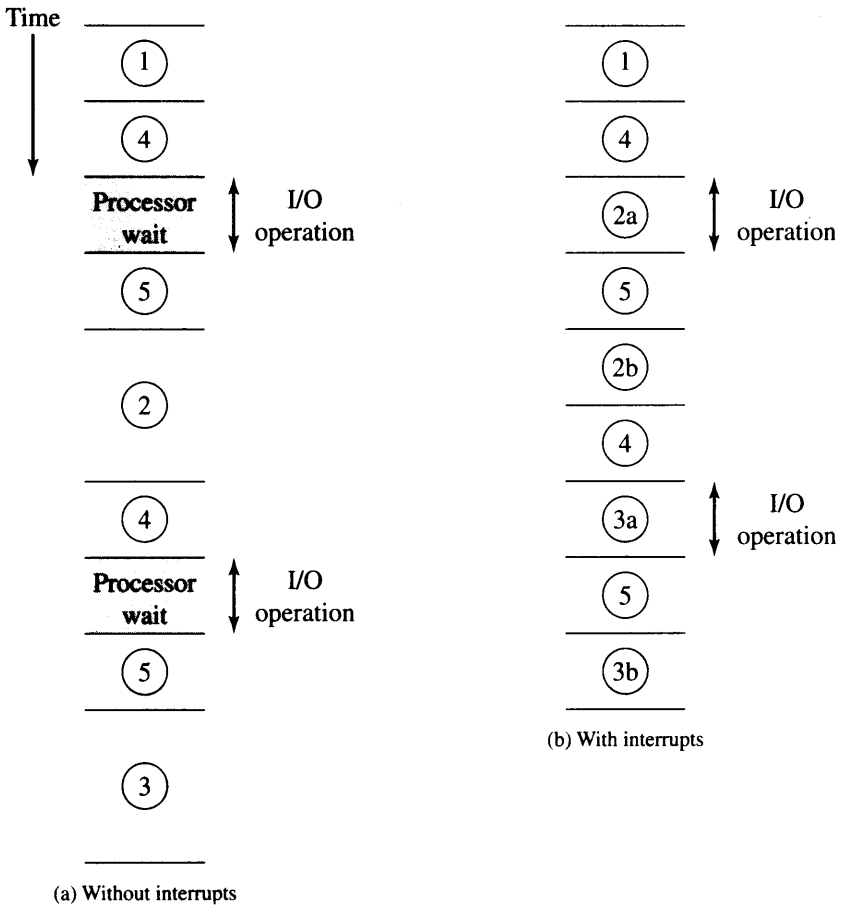


Figure 3.10 Program Timing: Short I/O Wait

amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

To appreciate the gain in efficiency, consider Figure 3.10, which is a timing diagram based on the flow of control in Figures 3.7a and 3.7b. Figures 3.7b and 3.10 assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. The more typical case, especially for a slow device such as a printer, is that the I/O operation will take much more time than executing a sequence of user instructions. Figure 3.7c indicates this state of affairs. In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O operation is completed, this new WRITE call may be processed, and a new I/O operation may be started. Figure 3.11 shows the timing for this situation with and without the use of interrupts. We can see that there is still a gain in efficiency because part of the time during which the I/O operation is underway overlaps with the execution of user instructions.

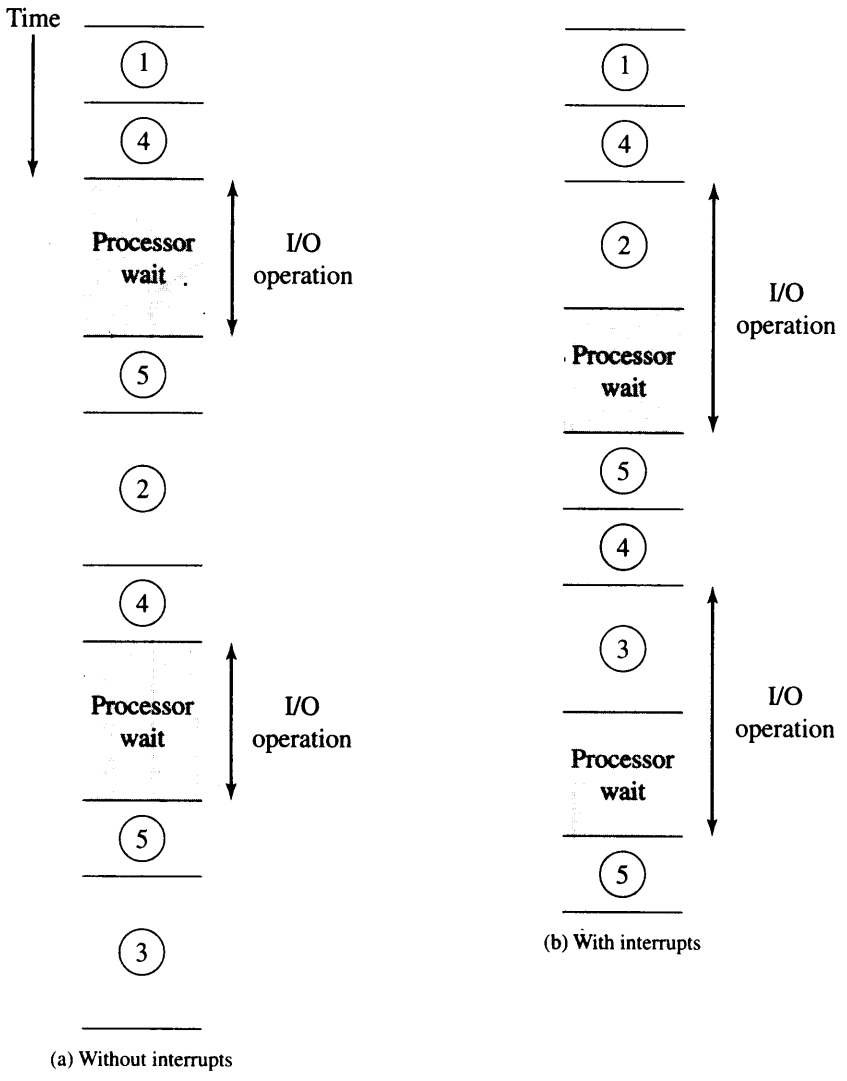


Figure 3.11 Program Timing: Long I/O Wait

Figure 3.12 shows a revised instruction cycle state diagram that includes interrupt cycle processing.

Multiple Interrupts The discussion so far has focused only on the occurrence of a single interrupt. Suppose, however, that multiple interrupts can occur. For example, a program may be receiving data from a communications line and printing results. The printer will generate an interrupt every time that it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed.

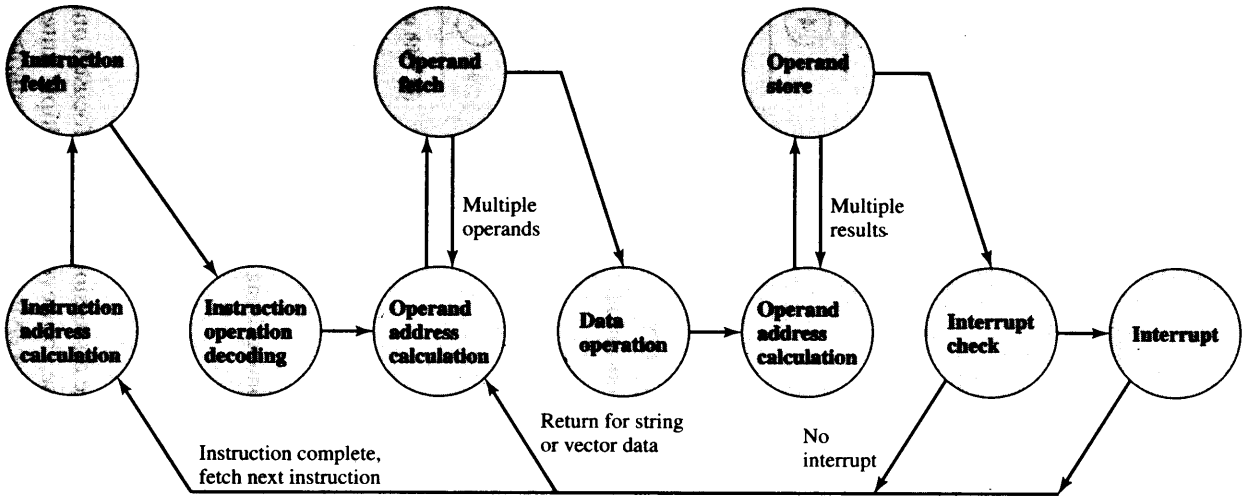


Figure 3.12 Instruction Cycle State Diagram, with Interrupts

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A *disabled interrupt* simply means that the processor can and will ignore that interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is nice and simple, as interrupts are handled in strict sequential order (Figure 3.13a).

The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted (Figure 3.13b). As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. Figure 3.14, based on an example in [TANE97], illustrates a possible sequence. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the system stack and execution continues at the printer interrupt service routine (ISR). While this routine is still executing, at $t = 15$, a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs ($t = 20$). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

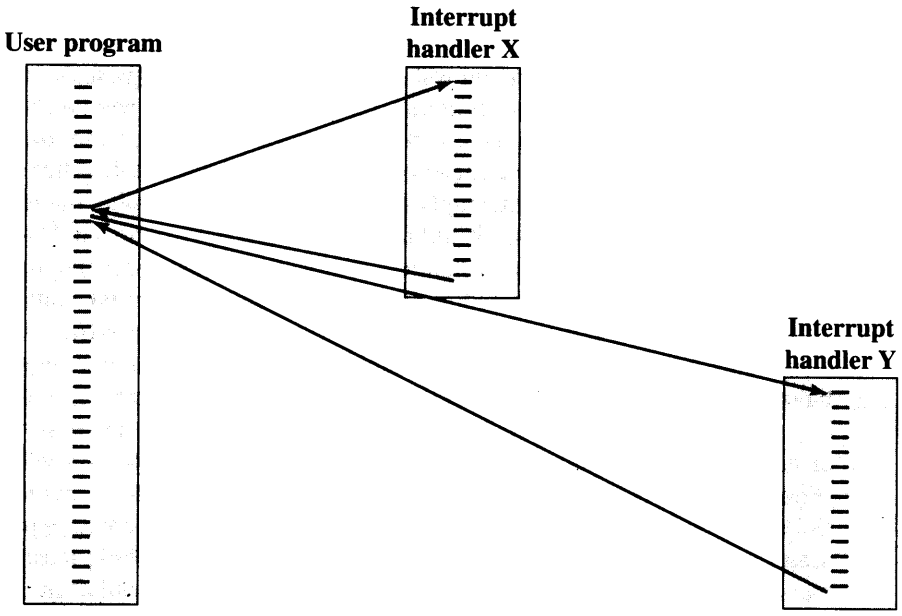
When the communications ISR is complete ($t = 25$), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and control transfers to the disk ISR. Only when that routine is complete ($t = 35$) is the printer ISR resumed. When that routine completes ($t = 40$), control finally returns to the user program.

I/O Function

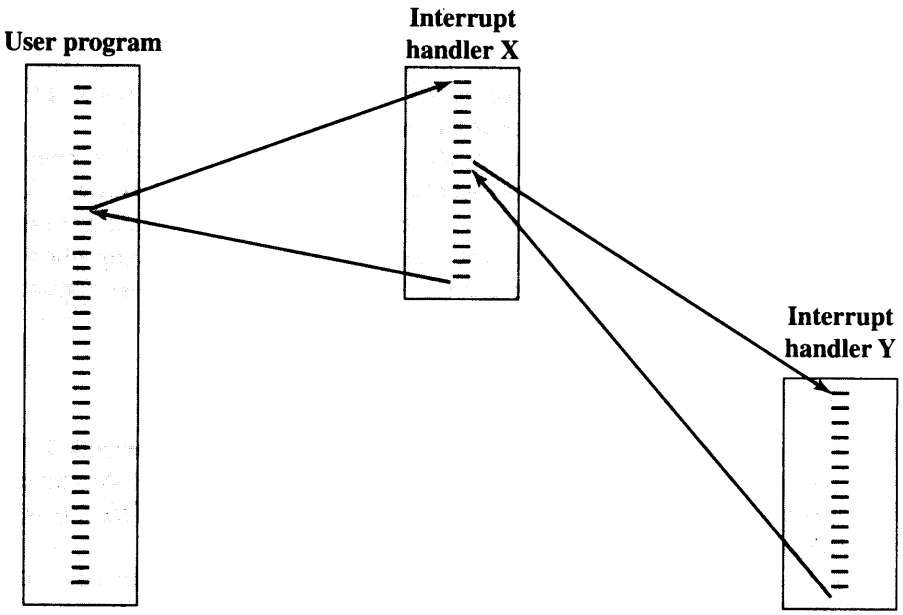
Thus far, we have discussed the operation of the computer as controlled by the processor, and we have looked primarily at the interaction of processor and memory. The discussion has only alluded to the role of the I/O component. This role is discussed in detail in Chapter 7, but a brief summary is in order here.

An I/O module (e.g., a disk controller) can exchange data directly with the processor. Just as the processor can initiate a read or write with memory, designating the address of a specific location, the processor can also read data from or write data to an I/O module. In this latter case, the processor identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence similar in form to that of Figure 3.5 could occur, with I/O instructions rather than memory-referencing instructions.

In some cases, it is desirable to allow I/O exchanges to occur directly with memory. In such a case, the processor grants to an I/O module the authority to read from



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 3.13 Transfer of Control with Multiple Interrupts

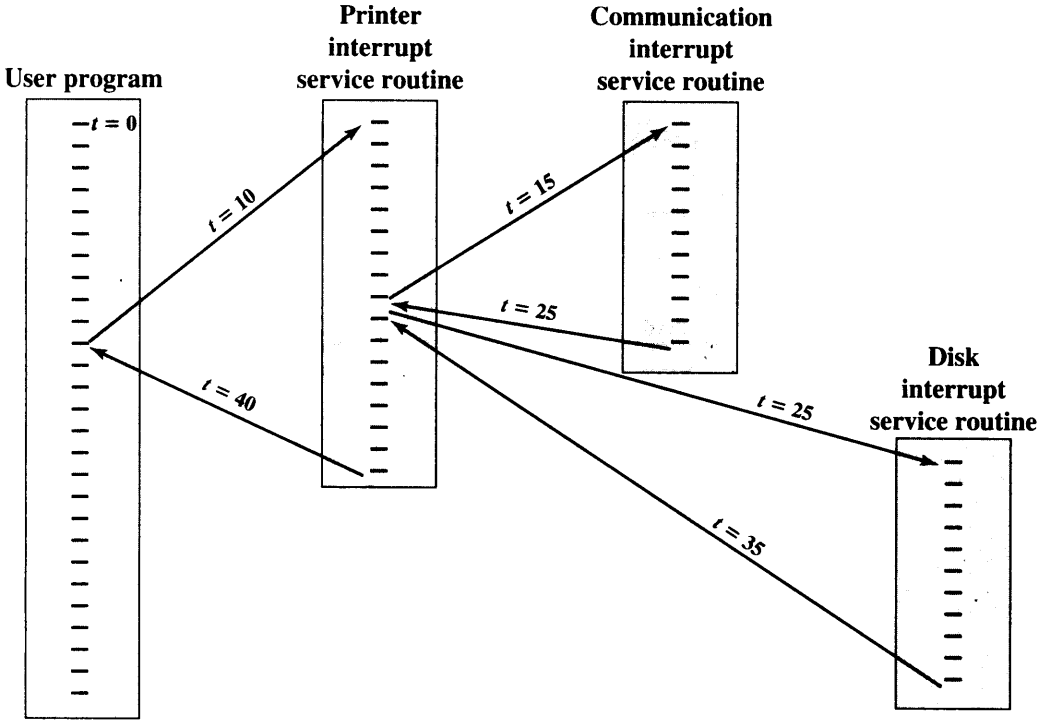


Figure 3.14 Example Time Sequence of Multiple Interrupts

or write to memory, so that the I/O-memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation is known as direct memory access (DMA) and is examined Chapter 7.

3.3 INTERCONNECTION STRUCTURES

A computer consists of a set of components or modules of three basic types (processor, memory, I/O) that communicate with each other. In effect, a computer is a network of basic modules. Thus, there must be paths for connecting the modules.

The collection of paths connecting the various modules is called the *interconnection structure.* The design of this structure will depend on the exchanges that must be made between modules.

Figure 3.15 suggests the types of exchanges that are needed by indicating the major forms of input and output for each module type:

- **Memory:** Typically, a memory module will consist of N words of equal length. Each word is assigned a unique numerical address $(0, 1, \ldots, N - 1)$. A word of data can be read from or written into the memory. The nature of the operation is indicated by read and write control signals. The location for the operation is specified by an address.

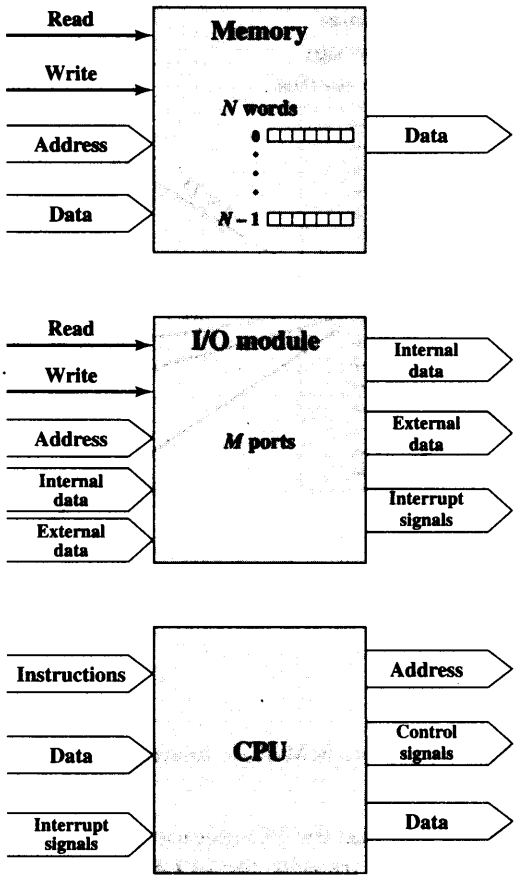


Figure 3.15 Computer Modules

- **I/O module:** From an internal (to the computer system) point of view, I/O is functionally similar to memory. There are two operations, read and write. Further, an I/O module may control more than one external device. We can refer to each of the interfaces to an external device as a *port* and give each a unique address (e.g., $0, 1, \ldots, M - 1$). In addition, there are external data paths for the input and output of data with an external device. Finally, an I/O module may be able to send interrupt signals to the processor.
- **Processor:** The processor reads in instructions and data, writes out data after processing, and uses control signals to control the overall operation of the system. It also receives interrupt signals.

The preceding list defines the data to be exchanged. The interconnection structure must support the following types of transfers:

- **Memory to processor:** The processor reads an instruction or a unit of data from memory.
- **Processor to memory:** The processor writes a unit of data to memory.

- **I/O to processor:** The processor reads data from an I/O device via an I/O module.
- **Processor to I/O:** The processor sends data to the I/O device.
- **I/O to or from memory:** For these two cases, an I/O module is allowed to exchange data directly with memory, without going through the processor, using direct memory access (DMA).

Over the years, a number of interconnection structures have been tried. By far the most common is the bus and various multiple-bus structures. The remainder of this chapter is devoted to an assessment of bus structures.

3.4 BUS INTERCONNECTION

A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus. If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Typically, a bus consists of multiple communication pathways, or lines. Each line is capable of transmitting signals representing binary 1 and binary 0. Over time, a sequence of binary digits can be transmitted across a single line. Taken together, several lines of a bus can be used to transmit binary digits simultaneously (in parallel). For example, an 8-bit unit of data can be transmitted over eight bus lines.

Computer systems contain a number of different buses that provide pathways between components at various levels of the computer system hierarchy. A bus that connects major computer components (processor, memory, I/O) is called a *system bus*. The most common computer interconnection structures are based on the use of one or more system buses.

Bus Structure

A system bus consists, typically, of from about 50 to hundreds of separate lines. Each line is assigned a particular meaning or function. Although there are many different bus designs, on any bus the lines can be classified into three functional groups (Figure 3.16):

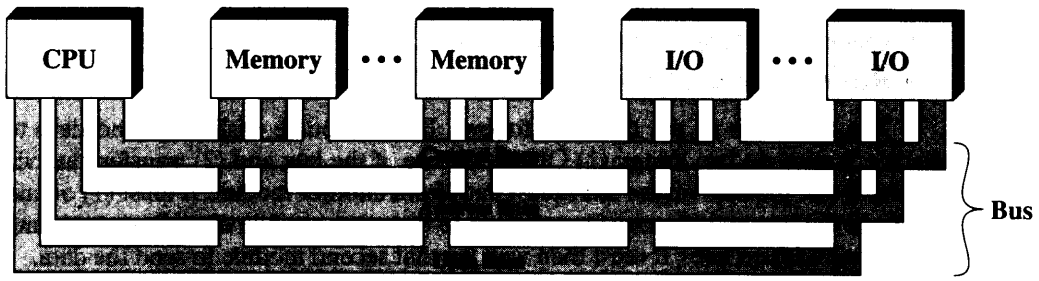


Figure 3.16 Bus Interconnection Scheme

data, address, and control lines. In addition, there may be power distribution lines that supply power to the attached modules.

The **data lines** provide a path for moving data between system modules. These lines, collectively, are called the *data bus*. The data bus may consist of from 32 to hundreds of separate lines, the number of lines being referred to as the *width* of the data bus. Because each line can carry only 1 bit at a time, the number of lines determines how many bits can be transferred at a time. The width of the data bus is a key factor in determining overall system performance. For example, if the data bus is 8 bits wide and each instruction is 16 bits long, then the processor must access the memory module twice during each instruction cycle.

The **address lines** are used to designate the source or destination of the data on the data bus. For example, if the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines. Clearly, the width of the address bus determines the maximum possible memory capacity of the system. Furthermore, the address lines are generally also used to address I/O ports. Typically, the higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module. For example, on an 8-bit address bus, address 01111111 and below might reference locations in a memory module (module 0) with 128 words of memory, and address 10000000 and above refer to devices attached to an I/O module (module 1).

The **control lines** are used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components, there must be a means of controlling their use. Control signals transmit both command and timing information between system modules. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Typical control lines include

- **Memory write:** Causes data on the bus to be written into the addressed location
- **Memory read:** Causes data from the addressed location to be placed on the bus
- **I/O write:** Causes data on the bus to be output to the addressed I/O port
- **I/O read:** Causes data from the addressed I/O port to be placed on the bus
- **Transfer ACK:** Indicates that data have been accepted from or placed on the bus
- **Bus request:** Indicates that a module needs to gain control of the bus
- **Bus grant:** Indicates that a requesting module has been granted control of the bus
- **Interrupt request:** Indicates that an interrupt is pending
- **Interrupt ACK:** Acknowledges that the pending interrupt has been recognized
- **Clock:** Used to synchronize operations
- **Reset:** Initializes all modules

The operation of the bus is as follows. If one module wishes to send data to another, it must do two things: (1) Obtain the use of the bus, and (2) transfer data via the bus. If one module wishes to request data from another module, it must (1) obtain the use of the bus, and (2) transfer a request to the other module over the appropriate control and address lines. It must then wait for that second module to send the data.

Physically, the system bus is actually a number of parallel electrical conductors. In the classic bus arrangement, these conductors are metal lines etched in a

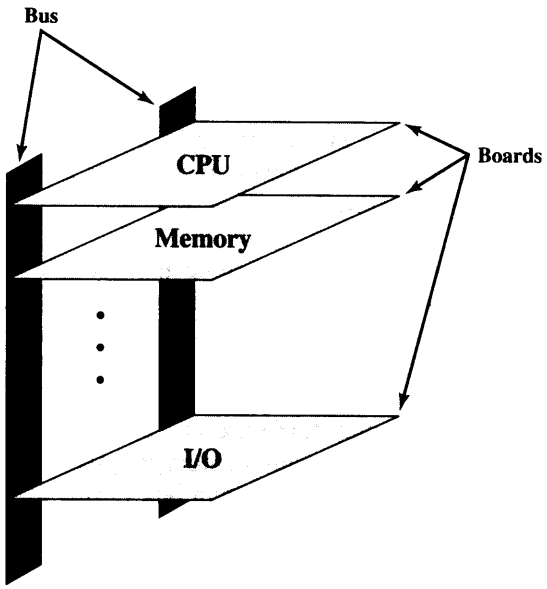


Figure 3.17 Typical Physical Realization of a Bus Architecture

card or board (printed circuit board). The bus extends across all of the system components, each of which taps into some or all of the bus lines. The classic physical arrangement is depicted in Figure 3.17. In this example, the bus consists of two vertical columns of conductors. At regular intervals along the columns, there are attachment points in the form of slots that extend out horizontally to support a printed circuit board. Each of the major system components occupies one or more boards and plugs into the bus at these slots. The entire arrangement is housed in a chassis. This scheme can still be used for some of the buses associated with a computer system. However, modern systems tend to have all of the major components on the same board with more elements on the same chip as the processor. Thus, an on-chip bus may connect the processor and cache memory, whereas an on-board bus may connect the processor to main memory and other components.

This arrangement is most convenient. A small computer system may be acquired and then expanded later (more memory, more I/O) by adding more boards. If a component on a board fails, that board can easily be removed and replaced.

Multiple-Bus Hierarchies

If a great number of devices are connected to the bus, performance will suffer. There are two main causes:

1. In general, the more devices attached to the bus, the greater the bus length and hence the greater the propagation delay. This delay determines the time it takes for devices to coordinate the use of the bus. When control of the bus passes from one device to another frequently, these propagation delays can noticeably affect performance.

2. The bus may become a bottleneck as the aggregate data transfer demand approaches the capacity of the bus. This problem can be countered to some extent by increasing the data rate that the bus can carry and by using wider buses (e.g., increasing the data bus from 32 to 64 bits). However, because the data rates generated by attached devices (e.g., graphics and video controllers, network interfaces) are growing rapidly, this is a race that a single bus is ultimately destined to lose.

Accordingly, most computer systems use multiple buses, generally laid out in a hierarchy. A typical traditional structure is shown in Figure 3.18a. There is a local bus that connects the processor to a cache memory and that may support one or more local devices. The cache memory controller connects the cache not only to this local bus, but to a system bus to which are attached all of the main memory modules. As will be discussed in Chapter 4, the use of a cache structure insulates the processor from a requirement to access main memory frequently. Hence, main memory can be moved off of the local bus onto a system bus. In this way, I/O transfers to and from the main memory across the system bus do not interfere with the processor's activity.

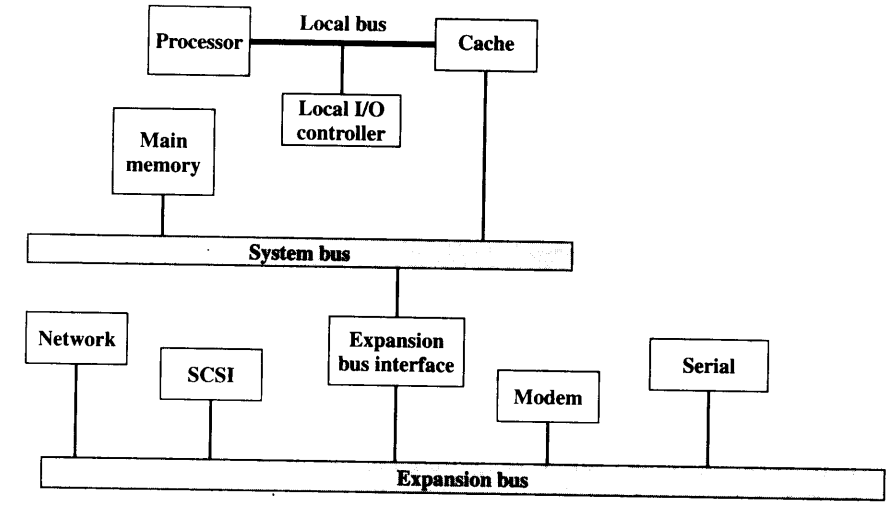
It is possible to connect I/O controllers directly onto the system bus. A more efficient solution is to make use of one or more expansion buses for this purpose. An expansion bus interface buffers data transfers between the system bus and the I/O controllers on the expansion bus. This arrangement allows the system to support a wide variety of I/O devices and at the same time insulate memory-to-processor traffic from I/O traffic.

Figure 3.18a shows some typical examples of I/O devices that might be attached to the expansion bus. Network connections include local area networks (LANs) such as a 10-Mbps Ethernet and connections to wide area networks (WANs) such as a packet-switching network. SCSI (small computer system interface) is itself a type of bus used to support local disk drives and other peripherals. A serial port could be used to support a printer or scanner.

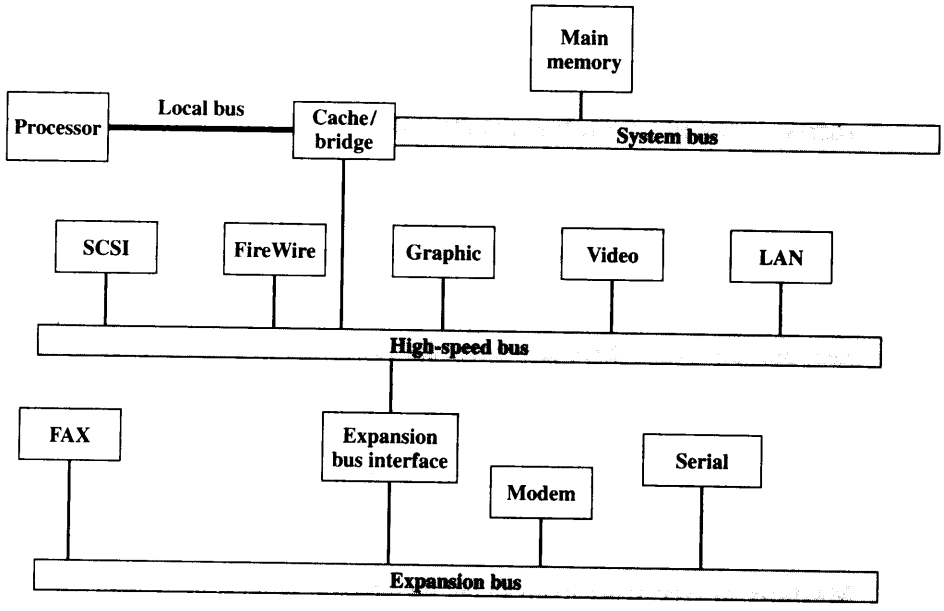
This traditional bus architecture is reasonably efficient but begins to break down as higher and higher performance is seen in the I/O devices. In response to these growing demands, a common approach taken by industry is to build a high-speed bus that is closely integrated with the rest of the system, requiring only a bridge between the processor's bus and the high-speed bus. This arrangement is sometimes known as a mezzanine architecture.

Figure 3.18b shows a typical realization of this approach. Again, there is a local bus that connects the processor to a cache controller, which is in turn connected to a system bus that supports main memory. The cache controller is integrated into a bridge, or buffering device, that connects to the high-speed bus. This bus supports connections to high-speed LANs, such as Fast Ethernet at 100 Mbps, video and graphics workstation controllers, as well as interface controllers to local peripheral buses, including SCSI and FireWire. The latter is a high-speed bus arrangement specifically designed to support high-capacity I/O devices. Lower-speed devices are still supported off an expansion bus, with an interface buffering traffic between the expansion bus and the high-speed bus.

The advantage of this arrangement is that the high-speed bus brings high-demand devices into closer integration with the processor and at the same time is independent of the processor. Thus, differences in processor and high-speed bus



(a) Traditional bus architecture



(b) High-performance architecture

Figure 3.18 Example Bus Configurations

speeds and signal line definitions are tolerated. Changes in processor architecture do not affect the high-speed bus, and vice versa.

Elements of Bus Design

Although a variety of different bus implementations exist, there are a few basic parameters or design elements that serve to classify and differentiate buses. Table 3.2 lists key elements.

Table 3.2 Elements of Bus Design

Type	**Bus Width**
Dedicated	Address
Multiplexed	Data
Method of Arbitration	**Data Transfer Type**
Centralized	Read
Distributed	Write
Timing	Read-modify-write
Synchronous	Read-after-write
Asynchronous	Block

Bus Types Bus lines can be separated into two generic types: dedicated and multiplexed. A dedicated bus line is permanently assigned either to one function or to a physical subset of computer components.

An example of functional dedication is the use of separate dedicated address and data lines, which is common on many buses. However, it is not essential. For example, address and data information may be transmitted over the same set of lines using an Address Valid control line. At the beginning of a data transfer, the address is placed on the bus and the Address Valid line is activated. At this point, each module has a specified period of time to copy the address and determine if it is the addressed module. The address is then removed from the bus, and the same bus connections are used for the subsequent read or write data transfer. This method of using the same lines for multiple purposes is known as *time multiplexing*.

The advantage of time multiplexing is the use of fewer lines, which saves space and, usually, cost. The disadvantage is that more complex circuitry is needed within each module. Also, there is a potential reduction in performance because certain events that share the same lines cannot take place in parallel.

Physical dedication refers to the use of multiple buses, each of which connects only a subset of modules. A typical example is the use of an I/O bus to interconnect all I/O modules; this bus is then connected to the main bus through some type of I/O adapter module. The potential advantage of physical dedication is high throughput, because there is less bus contention. A disadvantage is the increased size and cost of the system.

Method of Arbitration In all but the simplest systems, more than one module may need control of the bus. For example, an I/O module may need to read or write directly to memory, without sending the data to the processor. Because only one unit at a time can successfully transmit over the bus, some method of arbitration is needed. The various methods can be roughly classified as being either centralized or distributed. In a centralized scheme, a single hardware device, referred to as a *bus controller* or *arbiter,* is responsible for allocating time on the bus. The device may be a separate module or part of the processor. In a distributed scheme, there is no central controller. Rather, each module contains access control logic and the modules act together to share the bus. With both methods of arbitration, the purpose is to designate one device, either the processor or an I/O module, as master. The master may then initiate a data transfer (e.g., read or write) with some other device, which acts as slave for this particular exchange.

Timing Timing refers to the way in which events are coordinated on the bus. Buses use either synchronous timing or asynchronous timing.

With **synchronous timing**, the occurrence of events on the bus is determined by a clock. The bus includes a clock line upon which a clock transmits a regular sequence of alternating 1s and 0s of equal duration. A single 1–0 transmission is referred to as a *clock cycle* or *bus cycle* and defines a time slot. All other devices on the bus can read the clock line, and all events start at the beginning of a clock cycle. Figure 3.19 shows a typical, but simplified, timing diagram for synchronous read and write operations (see Appendix 3A for a description of timing diagrams). Other bus signals may change at the leading edge of the clock signal (with a slight reaction delay). Most events occupy a single clock cycle. In this simple example, the processor places a memory address on the address lines during the first clock cycle and may assert various status lines. Once the address lines have stabilized, the processor issues an address enable signal. For a read operation, the processor issues a read command at the start of the second cycle. A memory module recognizes the address

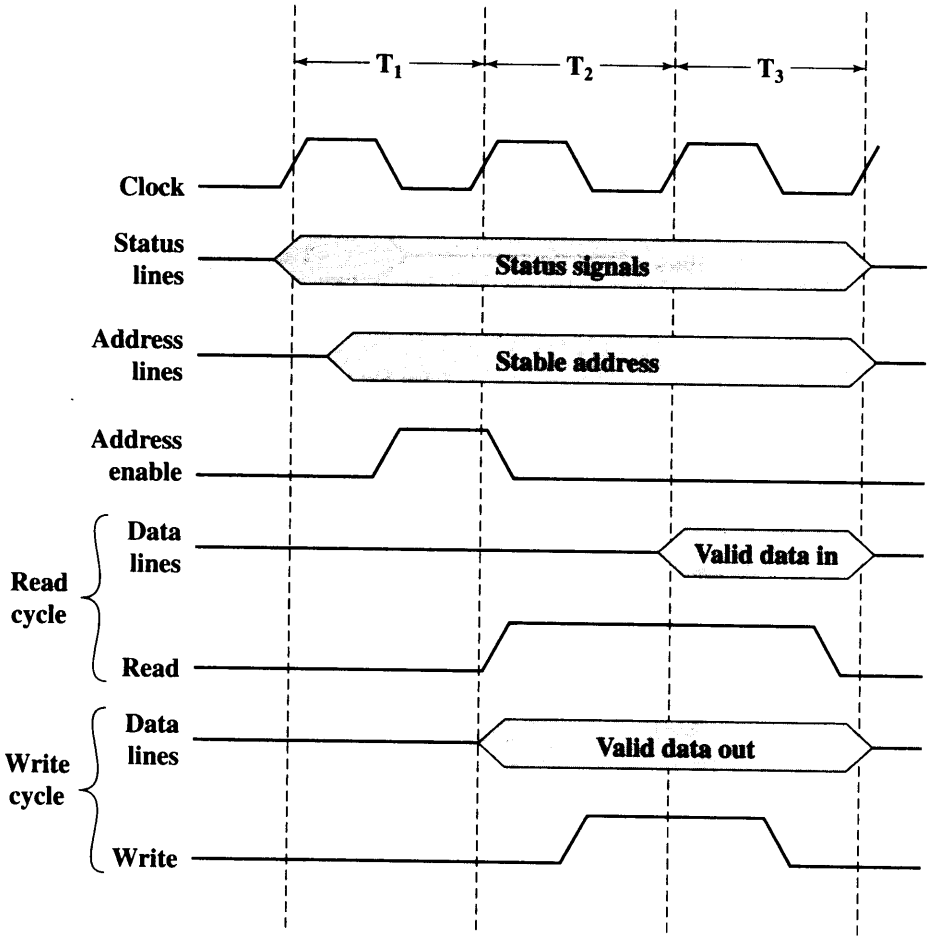
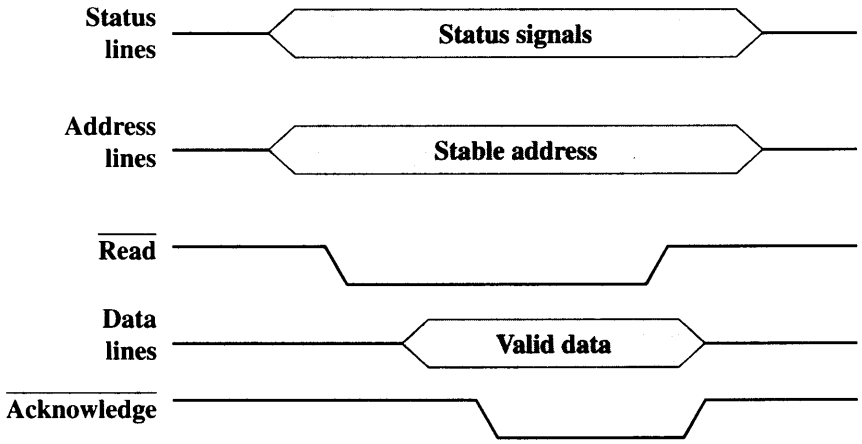


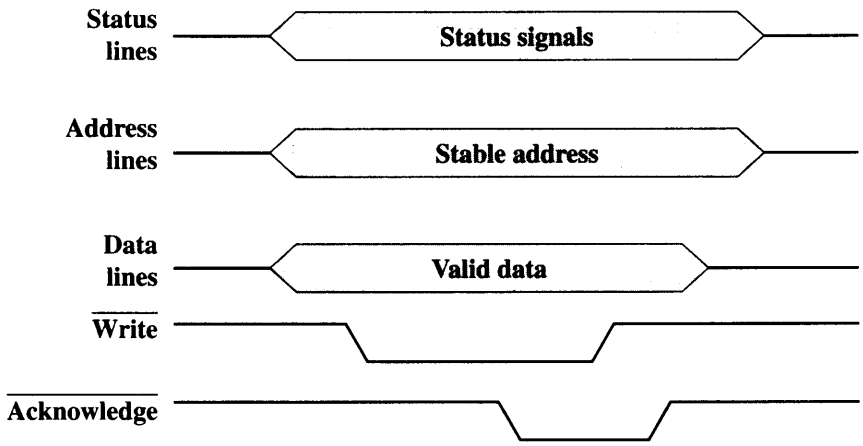
Figure 3.19 Timing of Synchronous Bus Operations

and, after a delay of one cycle, places the data on the data lines. The processor reads the data from the data lines and drops the read signal. For a write operation, the processor puts the data on the data lines at the start of the second cycle, and issues a write command after the data lines have stabilized. The memory module copies the information from the data lines during the third clock cycle.

With **asynchronous timing**, the occurrence of one event on a bus follows and depends on the occurrence of a previous event. In the simple read example of Figure 3.20a, the processor places address and status signals on the bus. After pausing for these signals to stabilize, it issues a read command, indicating the presence of valid address and control signals. The appropriate memory decodes the address and responds by placing the data on the data line. Once the data lines have stabilized, the memory module asserts the acknowledged line to signal the processor that the



(a) System bus read cycle



(b) System bus write cycle

Figure 3.20 Timing of Asynchronous Bus Operations

data are available. Once the master has read the data from the data lines, it deasserts the read signal. This causes the memory module to drop the data and acknowledge lines. Finally, once the acknowledge line is dropped, the master removes the address information.

Figure 3.20b shows a simple asynchronous write operation. In this case, the master places the data on the data line at the same time that is puts signals on the status and address lines. The memory module responds to the write command by copying the data from the data lines and then asserting the acknowledge line. The master then drops the write signal and the memory module drops the acknowledge signal.

Synchronous timing is simpler to implement and test. However, it is less flexible than asynchronous timing. Because all devices on a synchronous bus are tied to a fixed clock rate, the system cannot take advantage of advances in device performance. With asynchronous timing, a mixture of slow and fast devices, using older and newer technology, can share a bus.

Bus Width We have already addressed the concept of bus width. The width of the data bus has an impact on system performance: The wider the data bus, the greater the number of bits transferred at one time. The width of the address bus has an impact on system capacity: The wider the address bus, the greater the range of locations that can be referenced.

Data Transfer Type Finally, a bus supports various data transfer types, as illustrated in Figure 3.21. All buses support both write (master to slave) and read (slave to master) transfers. In the case of a multiplexed address/data bus, the bus is first used for specifying the address and then for transferring the data. For a read operation, there is typically a wait while the data is being fetched from the slave to be put on the bus. For either a read or a write, there may also be a delay if it is necessary to go through arbitration to gain control of the bus for the remainder of the operation (i.e., seize the bus to request a read or write, then seize the bus again to perform a read or write).

In the case of dedicated address and data buses, the address is put on the address bus and remains there while the data are put on the data bus. For a write operation, the master puts the data onto the data bus as soon as the address has stabilized and the slave has had the opportunity to recognize its address. For a read operation, the slave puts the data onto the data bus as soon as it has recognized its address and has fetched the data.

There are also several combination operations that some buses allow. A read–modify–write operation is simply a read followed immediately by a write to the same address. The address is only broadcast once at the beginning of the operation. The whole operation is typically indivisible to prevent any access to the data element by other potential bus masters. The principal purpose of this capability is to protect shared memory resources in a multiprogramming system (see Chapter 8).

Read-after-write is an indivisible operation consisting of a write followed immediately by a read from the same address. The read operation may be performed for checking purposes.

Some bus systems also support a block data transfer. In this case, one address cycle is followed by n data cycles. The first data item is transferred to or from the specified address; the remaining data items are transferred to or from subsequent addresses.

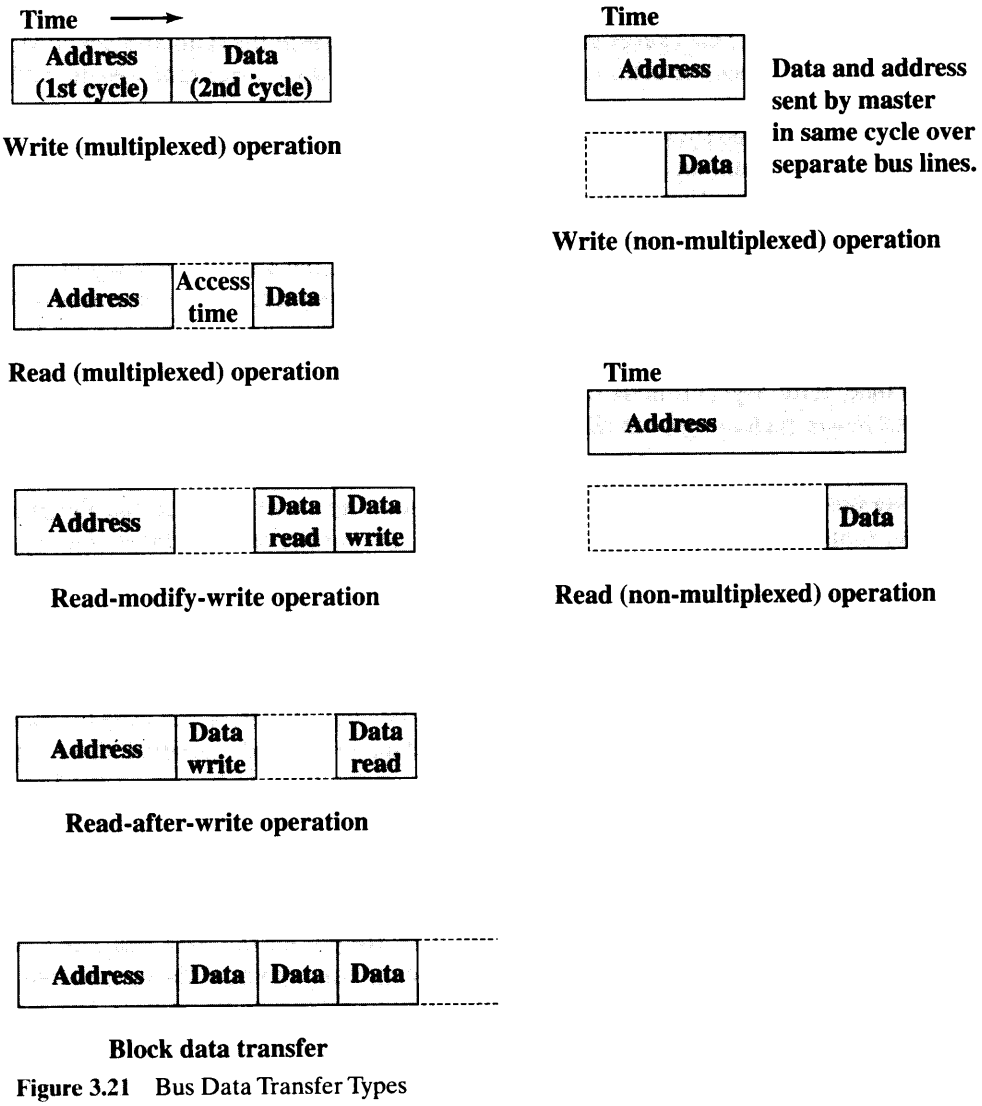


Figure 3.21 Bus Data Transfer Types

3.5 PCI

The peripheral component interconnect (PCI) is a popular high-bandwidth, processor-independent bus that can function as a mezzanine or peripheral bus. Compared with other common bus specifications, PCI delivers better system performance for high-speed I/O subsystems (e.g., graphic display adapters, network interface controllers, disk controllers, and so on). The current standard allows the use of up to 64 data lines at 66 MHz, for a raw transfer rate of 528 MByte/s, or 4.224 Gbps. But it is not just a high speed that makes PCI attractive. PCI is specifically designed to meet economically the I/O requirements of modern systems; it requires very few chips to implement and supports other buses attached to the PCI bus.

Intel began work on PCI in 1990 for its Pentium-based systems. Intel soon released all the patents to the public domain and promoted the creation of an industry association, the PCI SIG, to develop further and maintain the compatibility of the PCI specifications. The result is that PCI has been widely adopted and is finding increasing use in personal computer, workstation, and server systems. Because the specification is in the public domain and is supported by a broad cross section of the microprocessor and peripheral industry, PCI products built by different vendors are compatible.

PCI is designed to support a variety of microprocessor-based configurations, including both single- and multiple-processor systems. Accordingly, it provides a general-purpose set of functions. It makes use of synchronous timing and a centralized arbitration scheme.

Figure 3.22a shows a typical use of PCI in a single-processor system. A combined DRAM controller and bridge to the PCI bus provides tight coupling with the processor and the ability to deliver data at high speeds. The bridge acts as a data buffer so that the speed of the PCI bus may differ from that of the processor's I/O capability. In a multiprocessor system (Figure 3.22b), one or more PCI configurations may be connected by bridges to the processor's system bus. The system bus supports only the processor/cache units, main memory, and the PCI bridges. Again, the use of bridges keeps the PCI independent of the processor speed yet provides the ability to receive and deliver data rapidly.

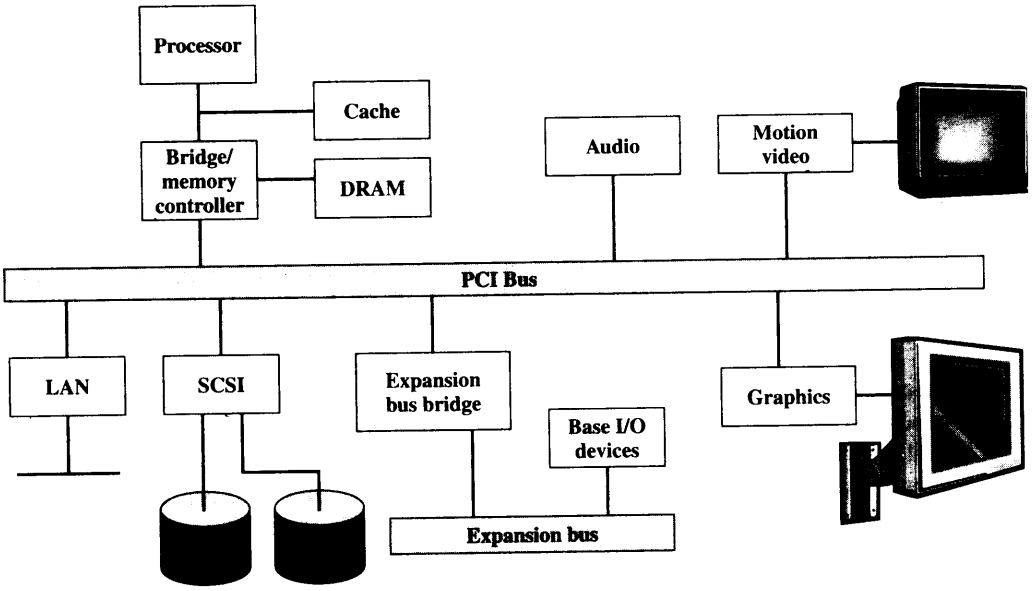
Bus Structure

PCI may be configured as a 32- or 64-bit bus. Table 3.3 defines the 49 mandatory signal lines for PCI. These are divided into the following functional groups:

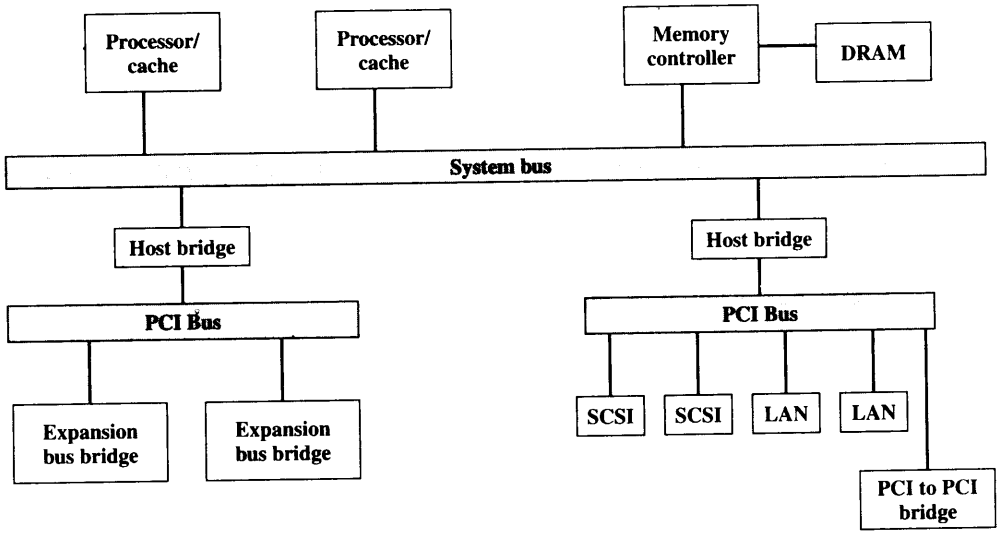
- **System pins:** Include the clock and reset pins.
- **Address and data pins:** Include 32 lines that are time multiplexed for addresses and data. The other lines in this group are used to interpret and validate the signal lines that carry the addresses and data.
- **Interface control pins:** Control the timing of transactions and provide coordination among initiators and targets.
- **Arbitration pins:** Unlike the other PCI signal lines, these are not shared lines. Rather, each PCI master has its own pair of arbitration lines that connect it directly to the PCI bus arbiter.
- **Error reporting pins:** Used to report parity and other errors.

In addition, the PCI specification defines 51 optional signal lines (Table 3.4), divided into the following functional groups:

- **Interrupt pins:** These are provided for PCI devices that must generate requests for service. As with the arbitration pins, these are not shared lines. Rather, each PCI device has its own interrupt line or lines to an interrupt controller.
- **Cache support pins:** These pins are needed to support a memory on PCI that can be cached in the processor or another device. These pins support snoopy cache protocols (see Chapter 18 for a discussion of such protocols).
- **64-bit bus extension pins:** Include 32 lines that are time multiplexed for addresses and data and that are combined with the mandatory address/data lines to form



(a) Typical desktop system



(b) Typical server system

Figure 3.22 Example PCI Configurations

a 64-bit address/data bus. Other lines in this group are used to interpret and validate the signal lines that carry the addresses and data. Finally, there are two lines that enable two PCI devices to agree to the use of the 64-bit capability.

- **JTAG/boundary scan pins:** These signal lines support testing procedures defined in IEEE Standard 1149.1.

Table 3.3 Mandatory PCI Signal Lines

Designation	Type	Description
		System Pins
CLK	in	Provides timing for all transactions and is sampled by all inputs on the rising edge. Clock rates up to 33 MHz are supported.
RST#	in	Forces all PCI-specific registers, sequencers, and signals to an initialized state.
		Address and Data Pins
AD[31::0]	t/s	Multiplexed lines used for address and data
C/BE[3::0]#	t/s	Multiplexed bus command and byte enable signals. During the data phase, the lines indicate which of the four byte lanes carry meaningful data.
PAR	t/s	Provides even parity across AD and C/BE lines one clock cycle later. The master drives PAR for address and write data phases; the target drive PAR for read data phases.
		Interface Control Pins
FRAME#	s/t/s	Driven by current master to indicate the start and duration of a transaction. It is asserted at the start and deasserted when the initiator is ready to begin the final data phase.
IRDY#	s/t/s	Initiator Ready. Driven by current bus master (initiator of transaction). During a read, indicates that the master is prepared to accept data; during a write, indicates that valid data are present on AD.
TRDY#	s/t/s	Target Ready. Driven by the target (selected device). During a read, indicates that valid data are present on AD; during a write, indicates that target is ready to accept data.
STOP#	s/t/s	Indicates that current target wishes the initiator to stop the current transaction.
IDSEL	in	Initialization Device Select. Used as a chip select during configuration read and write transactions.
DEVSEL#	in	Device Select. Asserted by target when it has recognized its address. Indicates to current initiator whether any device has been selected.
		Arbitration Pins
REQ#	t/s	Indicates to the arbiter that this device requires use of the bus. This is a device-specific point-to-point line.
GNT#	t/s	Indicates to the device that the arbiter has granted bus access. This is a device-specific point-to-point line.
		Error Reporting Pins
PERR#	s/t/s	Parity Error. Indicates a data parity error is detected by a target during a write data phase or by an initiator during a read data phase.
SERR#	o/d	System Error. May be pulsed by any device to report address parity errors and critical errors other than parity.

Table 3.4 Optional PCI Signal Lines

Designation	Type	Description
		Interrupt Pins
INTA#	o/d	Used to request an interrupt.
INTB#	o/d	Used to request an interrupt; only has meaning on a multifunction device.
INTC#	o/d	Used to request an interrupt; only has meaning on a multifunction device.
INTD#	o/d	Used to request an interrupt; only has meaning on a multifunction device.
		Cache Support Pins
SBO#	in/out	Snoop Backoff. Indicates a hit to a modified line.
SDONE	in/out	Snoop Done. Indicates the status of the snoop for the current access. Asserted when snoop has been completed.
		64-bit Bus Extension Pins
AD[63::32]	t/s	Multiplexed lines used for address and data to extend bus to 64 bits.
C/BE[7::4]#	t/s	Multiplexed bus command and byte enable signals. During the address phase, the lines provide additional bus commands. During the data phase, the lines indicate which of the four extended byte lanes carry meaningful data.
REQ64#	s/t/s	Used to request 64-bit transfer.
ACK64#	s/t/s	Indicates target is willing to perform 64-bit transfer.
PAR64	t/s	Provides even parity across extended AD and C/BE lines one clock cycle later.
		JTAG/Boundary Scan Pins
TCK	in	Test clock. Used to clock state information and test data into and out of the device during boundary scan.
TDI	in	Test input. Used to serially shift test data and instructions into the device.
TDO	out	Test output. Used to serially shift test data and instructions out of the device.
TMS	in	Test mode Select. Used to control state of test access port controller.
TRST#	in	Test reset. Used to initialize test access port controller.

in	Input-only signal
out	Output-only signal
t/s	Bidirectional, tri-state, I/O signal
s/t/s	Sustained tri-state signal driven by only one owner at a time
o/d	Open drain: allows multiple devices to share as a wire-OR
#	Signal's active state occurs at low voltage

PCI Commands

Bus activity occurs in the form of transactions between an initiator, or master, and a target. When a bus master acquires control of the bus, it determines the type of transaction that will occur next. During the address phase of the transaction, the C/BE lines are used to signal the transaction type. The commands are

- Interrupt Acknowledge
- Special Cycle
- I/O Read
- I/O Write
- Memory Read
- Memory Read Line
- Memory Read Multiple
- Memory Write
- Memory Write and Invalidate
- Configuration Read
- Configuration Write
- Dual Address Cycle

Interrupt Acknowledge is a read command intended for the device that functions as an interrupt controller on the PCI bus. The address lines are not used during the address phase, and the byte enable lines indicate the size of the interrupt identifier to be returned.

The Special Cycle command is used by the initiator to broadcast a message to one or more targets.

The I/O Read and Write commands are used to transfer data between the initiator and an I/O controller. Each I/O device has its own address space, and the address lines are used to indicate a particular device and to specify the data to be transferred to or from that device. The concept of I/O addresses is explored in Chapter 7.

The memory read and write commands are used to specify the transfer of a burst of data, occupying one or more clock cycles. The interpretation of these commands depends on whether or not the memory controller on the PCI bus supports the PCI protocol for transfers between memory and cache. If so, the transfer of data to and from the memory is typically in terms of cache lines, or blocks.[2] The three memory read commands have the uses outlined in Table 3.5. The Memory Write command is used to transfer data in one or more data cycles to memory. The Memory Write and Invalidate command transfers data in one or more cycles to memory. In addition, it guarantees that at least one cache line is written. This command supports the cache function of writing back a line to memory.

---

[2]The fundamental principles of cache memory are described in Chapter 4; bus-based cache protocols are described in Chapter 18.

Table 3.5 Interpretation of PCI Read Commands

Read Command Type	For Cachable Memory	For Noncachable Memory
Memory Read	Bursting one-half or less of a cache line	Bursting 2 data transfer cycles or less
Memory Read Line	Bursting more than one-half a cache line to three cache lines	Bursting 3 to 12 data transfers
Memory Read Multiple	Bursting more than three cache lines	Bursting more than 12 data transfers

The two configuration commands enable a master to read and update configuration parameters in a device connected to the PCI. Each PCI device may include up to 256 internal registers that are used during system initialization to configure that device.

The Dual Address Cycle command is used by an initiator to indicate that it is using 64-bit addressing.

Data Transfers

Every data transfer on the PCI bus is a single transaction consisting of one address phase and one or more data phases. In this discussion, we illustrate a typical read operation; a write operation proceeds similarly.

Figure 3.23 shows the timing of the read transaction. All events are synchronized to the falling transitions of the clock, which occur in the middle of each clock cycle. Bus devices sample the bus lines on the rising edge at the beginning of a bus cycle. The following are the significant events, labeled on the diagram:

a. Once a bus master has gained control of the bus, it may begin the transaction by asserting FRAME. This line remains asserted until the initiator is ready to complete the last data phase. The initiator also puts the start address on the address bus, and the read command on the C/BE lines.

b. At the start of clock 2, the target device will recognize its address on the AD lines.

c. The initiator ceases driving the AD bus. A turnaround cycle (indicated by the two circular arrows) is required on all signal lines that may be driven by more than one device, so that the dropping of the address signal will prepare the bus for use by the target device. The initiator changes the information on the C/BE lines to designate which AD lines are to be used for transfer for the currently addressed data (from 1 to 4 bytes). The initiator also asserts IRDY to indicate that it is ready for the first data item.

d. The selected target asserts DEVSEL to indicate that it has recognized its address and will respond. It places the requested data on the AD lines and asserts TRDY to indicate that valid data is present on the bus.

e. The initiator reads the data at the beginning of clock 4 and changes the byte enable lines as needed in preparation for the next read.

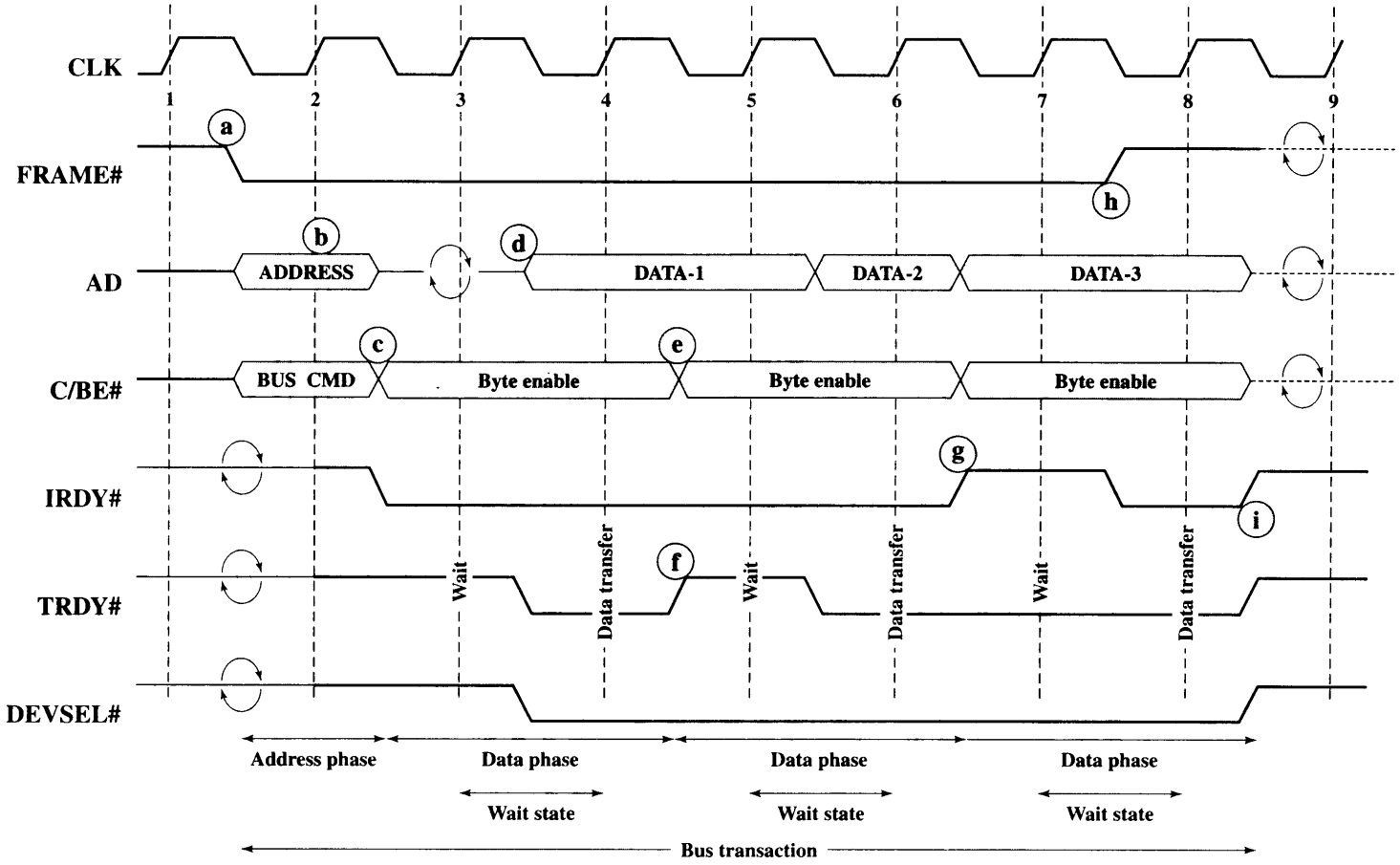


Figure 3.23 PCI Read Operation

f. In this example, the target needs some time to prepare the second block of data for transmission. Therefore, it deasserts TRDY to signal the initiator that there will not be new data during the coming cycle. Accordingly, the initiator does not read the data lines at the beginning of the fifth clock cycle and does not change byte enable during that cycle. The block of data is read at beginning of clock 6.

g. During clock 6, the target places the third data item on the bus. However, in this example, the initiator is not yet ready to read the data item (e.g., it has a temporary buffer full condition). It therefore deasserts IRDY. This will cause the target to maintain the third data item on the bus for an extra clock cycle.

h. The initiator knows that the third data transfer is the last, and so it deasserts FRAME to signal the target that this is the last data transfer. It also asserts IRDY to signal that it is ready to complete that transfer.

i. The initiator deasserts IRDY, returning the bus to the idle state, and the target deasserts TRDY and DEVSEL.

Arbitration

PCI makes use of a centralized, synchronous arbitration scheme in which each master has a unique request (REQ) and grant (GNT) signal. These signal lines are attached to a central arbiter (Figure 3.24) and a simple request–grant handshake is used to grant access to the bus.

The PCI specification does not dictate a particular arbitration algorithm. The arbiter can use a first-come-first-served approach, a round-robin approach, or some sort of priority scheme. A PCI master must arbitrate for each transaction that it wishes to perform, where a single transaction consists of an address phase followed by one or more contiguous data phases.

Figure 3.25 is an example in which devices A and B are arbitrating for the bus. The following sequence occurs:

a. At some point prior to the start of clock 1, A has asserted its REQ signal. The arbiter samples this signal at the beginning of clock cycle 1.

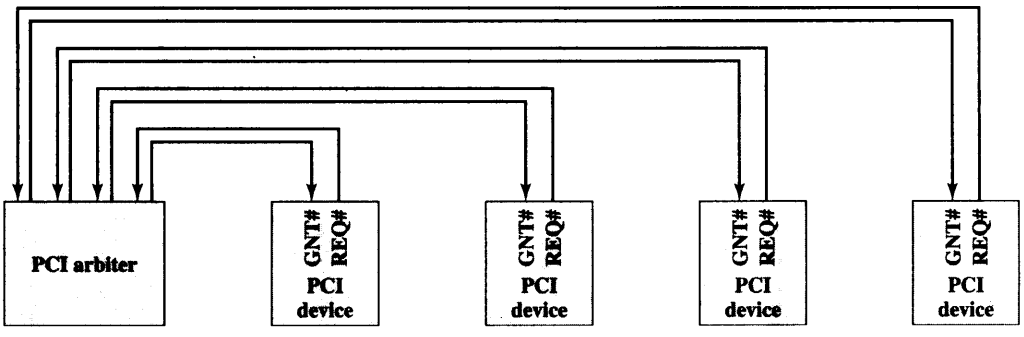


Figure 3.24 PCI Bus Arbiter

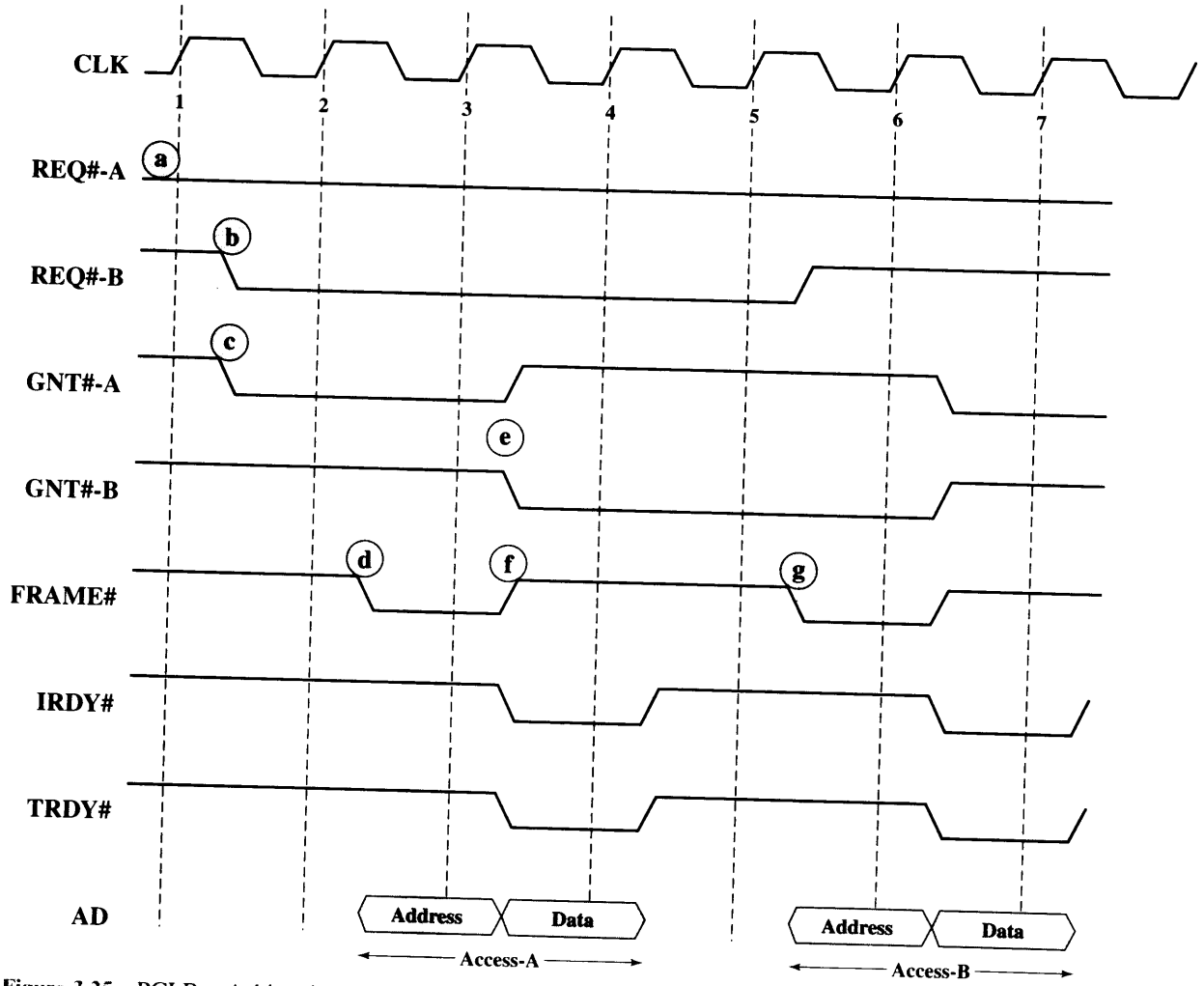


Figure 3.25 PCI Bus Arbitration between Two Masters

b. During clock cycle 1, B requests use of the bus by asserting its REQ signal.
c. At the same time, the arbiter asserts GNT-A to grant bus access to A.
d. Bus master A samples GNT-A at the beginning of clock 2 and learns that it has been granted bus access. It also finds IRDY and TRDY deasserted, indicating that the bus is idle. Accordingly, it asserts FRAME and places the address information on the address bus and the command on the C/BE bus (not shown). It also continues to assert REQ-A, because it has a second transaction to perform after this one.
e. The bus arbiter samples all REQ lines at the beginning of clock 3 and makes an arbitration decision to grant the bus to B for the next transaction. It then asserts GNT-B and deasserts GNT-A. B will not be able to use the bus until it returns to an idle state.
f. A deasserts FRAME to indicate that the last (and only) data transfer is in progress. It puts the data on the data bus and signals the target with IRDY. The target reads the data at the beginning of the next clock cycle.
g. At the beginning of clock 5, B finds IRDY and FRAME deasserted and so is able to take control of the bus by asserting FRAME. It also deasserts its REQ line, because it only wants to perform one transaction.

Subsequently, master A is granted access to the bus for its next transaction.

Notice that arbitration can take place at the same time that the current bus master is performing a data transfer. Therefore, no bus cycles are lost in performing arbitration. This is referred to as *hidden arbitration*.

3.6 RECOMMENDED READING AND WEB SITES

The clearest book-length description of PCI is [SHAN99]. [ABBO04] also contains a lot of solid information on PCI.

ABBO04 Abbot, D. *PCI Bus Demystified*. New York: Elsevier, 2004.
SHAN99 Shanley, T., and Anderson, D. *PCI Systems Architecture*. Richardson, TX: Mindshare Press, 1999.

Recommended Web Sites:

- **PCI Special Interest Group:** Information about PCI specifications and products
- **PCI Pointers:** Links to PCI vendors and other sources of information

3.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

address bus	distributed arbitration	memory address register (MAR)
asynchronous timing	instruction cycle	memory buffer register (MBR)
bus	instruction execute	peripheral component interconnect (PCI)
bus arbitration	instruction fetch	synchronous timing
bus width	interrupt	system bus
centralized arbitration	interrupt handler	
data bus	interrupt service routine	
disabled interrupt		

Review Questions

3.1 What general categories of functions are specified by computer instructions?

3.2 List and briefly define the possible states that define an instruction execution.

3.3 List and briefly define two approaches to dealing with multiple interrupts.

3.4 What types of transfers must a computer's interconnection structure (e.g., bus) support?

3.5 What is the benefit of using a multiple-bus architecture compared to a single-bus architecture?

3.6 List and briefly define the functional groups of signal lines for PCI.

Problems

3.1 The hypothetical machine of Figure 3.4 also has two I/O instructions:

0011 = Load AC from I/O
0111 = Store AC to I/O

In these cases, the 12-bit address identifies a particular I/O device. Show the program execution (using the format of Figure 3.5) for the following program:

1. Load AC from device 5.
2. Add contents of memory location 940.
3. Store AC to device 6.

Assume that the next value retrieved from device 5 is 3 and that location 940 contains a value of 2.

3.2 The program execution of Figure 3.5 is described in the text using six steps. Expand this description to show the use of the MAR and MBR.

3.3 Consider a hypothetical 32-bit microprocessor having 32-bit instructions composed of two fields: the first byte contains the opcode and the remainder the immediate operand or an operand address.

a. What is the maximum directly addressable memory capacity (in bytes)?

b. Discuss the impact on the system speed if the microprocessor bus has
   1. a 32-bit local address bus and a 16-bit local data bus, or
   2. a 16-bit local address bus and a 16-bit local data bus.

c. How many bits are needed for the program counter and the instruction register?

3.4 Consider a hypothetical microprocessor generating a 16-bit address (for example, assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus.

a. What is the maximum memory address space that the processor can access directly if it is connected to a "16-bit memory"?
b. What is the maximum memory address space that the processor can access directly if it is connected to an "8-bit memory"?
c. What architectural features will allow this microprocessor to access a separate "I/O space"?
d. If an input and an output instruction can specify an 8-bit I/O port number, how many 8-bit I/O ports can the microprocessor support? How many 16-bit I/O ports? Explain.

3.5 Consider a 32-bit microprocessor, with a 16-bit external data bus, driven by an 8-MHz input clock. Assume that this microprocessor has a bus cycle whose minimum duration equals four input clock cycles. What is the maximum data transfer rate across the bus that this microprocessor can sustain, in bytes/s? To increase its performance, would it be better to make its external data bus 32 bits or to double the external clock frequency supplied to the microprocessor? State any other assumptions you make, and explain. *Hint:* Determine the number of bytes that can be transferred per bus cycle.

3.6 Consider a computer system that contains an I/O module controlling a simple keyboard/printer teletype. The following registers are contained in the processor and connected directly to the system bus:

INPR:	Input Register, 8 bits
OUTR:	Output Register, 8 bits
FGI:	Input Flag, 1 bit
FGO:	Output Flag, 1 bit
IEN:	Interrupt Enable, 1 bit

Keystroke input from the teletype and printer output to the teletype are controlled by the I/O module. The teletype is able to encode an alphanumeric symbol to an 8-bit word and decode an 8-bit word into an alphanumeric symbol.

a. Describe how the processor, using the first four registers listed in this problem, can achieve I/O with the teletype.
b. Describe how the function can be performed more efficiently by also employing IEN.

3.7 Consider two microprocessors having 8- and 16-bit-wide external data buses, respectively. The two processors are identical otherwise and their bus cycles take just as long.

a. Suppose all instructions and operands are two bytes long. By what factor do the maximum data transfer rates differ?
b. Repeat assuming that half of the operands and instructions are one byte long.

3.8 Figure 3.26 indicates a distributed arbitration scheme that can be used with an obsolete bus scheme known as Multibus I. Agents are daisy-chained physically in priority order. The left-most agent in the diagram receives a constant *bus priority in* (BPRN) signal indicating that no higher-priority agent desires the bus. If the agent does not

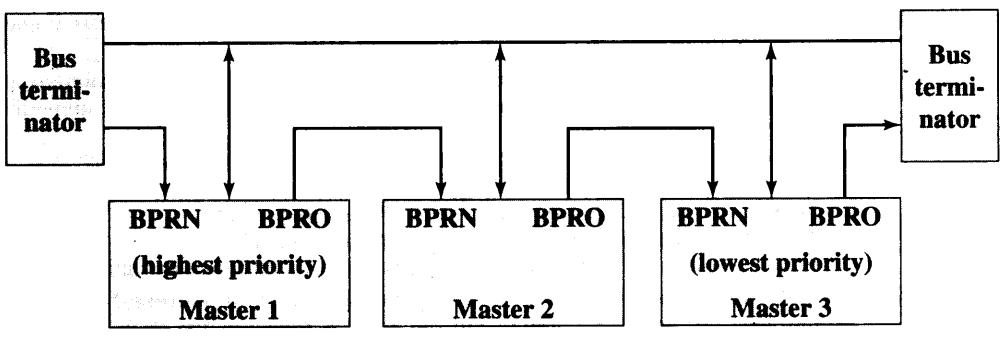


Figure 3.26 Multibus I Distributed Arbitration

wish the bus, it asserts its *bus priority out* (BPRO) line. At the beginning of a clock cycle, any agent can request control of the bus by lowering its BPRO line. This lowers the BPRN line of the next agent in the chain, which is in turn required to lower its BPRO line. Thus, the signal is propagated the length of the chain. At the end of this chain reaction, there should be only one agent whose BPRN is asserted and whose BPRO is not. This agent has priority. If, at the beginning of a bus cycle, the bus is not busy (BUSY inactive), the agent that has priority may seize control of the bus by asserting the BUSY line.

It takes a certain amount of time for the BPR signal to propagate from the highest-priority agent to the lowest. Must this time be less than the clock cycle? Explain.

3.9 The VAX SBI bus uses a distributed, synchronous arbitration scheme. Each SBI device (i.e., processor, memory, I/O module) has a unique priority and is assigned a unique transfer request (TR) line. The SBI has 16 such lines (TR0, TR1, . . . , TR15), with TR0 having the highest priority. When a device wants to use the bus, it places a reservation for a future time slot by asserting its TR line during the current time slot. At the end of the current time slot, each device with a pending reservation examines the TR lines; the highest-priority device with a reservation uses the next time slot.

A maximum of 17 devices can be attached to the bus. The device with priority 16 has no TR line. Why not?

3.10 On the VAX SBI, the lowest-priority device usually has the lowest average wait time. For this reason, the processor is usually given the lowest priority on the SBI. Why does the priority 16 device usually have the lowest average wait time? Under what circumstances would this not be true?

3.11 For a synchronous read operation (Figure 3.19), the memory module must place the data on the bus sufficiently ahead of the falling edge of the Read signal to allow for signal settling. Assume a microprocessor bus is clocked at 10 MHz and the the Read signal begins to fall in the middle of the second half of T_3.

a. Determine the length of the memory read instruction cycle.

b. When, at the latest, should memory data be placed on the bus? Allow 20 ns for the settling of data lines.

3.12 Consider a microprocessor that has a memory read timing as shown in Figure 3.19. After some analysis, a designer determines that the memory falls short of providing read data on time by about 180 ns.

a. How many wait states (clock cycles) need to be inserted for proper system operation if the bus clocking rate is 8 MHz?

b. To enforce the wait states, a Ready status line is employed. Once the processor has issued a Read command, it must wait until the Ready line is asserted before attempting to read data. At what time interval must we keep the Ready line low in order to force the processor to insert the required number of wait states?

3.13 A microprocessor has a memory write timing as shown in Figure 3.19. Its manufacturer specifies that the width of the Write signal can be determined by $T - 50$ where T is the clock period in ns.

a. What width should we expect for the Write signal if bus clocking rate is 5 MHz?

b. The data sheet for the microprocessor specifies that the data remain valid for 20 ns after the falling edge of the Write signal. What is the total duration of valid data presentation to memory?

c. How many wait states should we insert if memory requires valid data presentation for at least 190 ns?

3.14 A microprocessor has an increment memory direct instruction, which adds 1 to the value in a memory location. The instruction has five stages: fetch opcode (4 bus clock cycles), fetch operand address (3 cycles), fetch operand (3 cycles), add 1 to operand (3 cycles), and store operand (3 cycles).

a. By what amount (in percent) will the duration of the instruction increase if we have to insert two bus wait states in each memory read and memory write operation?

b. Repeat assuming that the increment operation takes 13 cycles instead of 3 cycles.

3.15 The Intel 8088 microprocessor has a read bus timing similar to that of Figure 3.19, but requires four processor clock cycles. The valid data is on the bus for an amount of time that extends into the fourth processor clock cycle. Assume a processor clock rate of 8 MHz.
   a. What is the maximum data transfer rate?
   b. Repeat but assume the need to insert one wait state per byte transferred.

3.16 The Intel 8086 is a 16-bit processor similar in many ways to the 8-bit 8088. The 8086 uses a 16-bit bus that can transfer 2 bytes at a time, provided that the lower-order byte has an even address. However, the 8086 allows both even- and odd-aligned word operands. If an odd-aligned word is referenced, two memory cycles, each consisting of four bus cycles, are required to transfer the word. Consider an instruction on the 8086 that involves two 16-bit operands. How long does it take to fetch the operands? Give the range of possible answers. Assume a clocking rate of 4 MHz and no wait states.

3.17 Consider a 32-bit microprocessor whose bus cycle is the same duration as that of a 16-bit microprocessor. Assume that, on average, 20% of the operands and instructions are 32 bits long, 40% are 16 bits long, and 40% are only 8 bits long. Calculate the improvement achieved when fetching instructions and operands with the 32-bit microprocessor.

3.18 The microprocessor of Problem 3.14 initiates the fetch operand stage of the increment memory direct instruction at the same time that a keyboard actives an interrupt request line. After how long does the processor enter the interrupt processing cycle? Assume a bus clocking rate of 10 MHz.

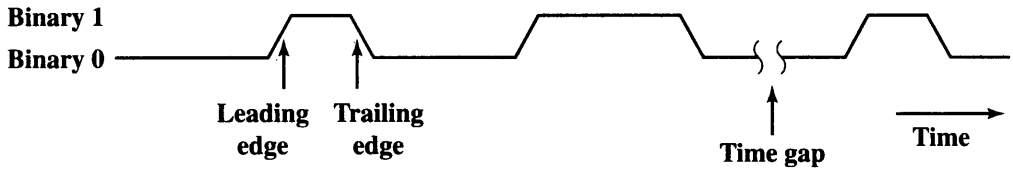
3.19 Draw and explain a timing diagram for a PCI write operation (similar to Figure 3.23).

APPENDIX 3A TIMING DIAGRAMS

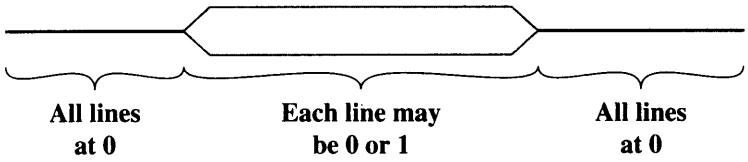
In this chapter, timing diagrams are used to illustrate sequences of events and dependencies among events. For the reader unfamiliar with timing diagrams, this appendix provides a brief explanation.

Communication among devices connected to a bus takes place along a set of lines capable of carrying signals. Two different signal levels (voltage levels), representing binary 0 and binary 1, may be transmitted. A timing diagram shows the signal level on a line as a function of time (Figure 3.27a). By convention, the binary 1 signal level is depicted as a higher level than that of binary 0. Usually, binary 0 is the default value. That is, if no data or other signal is being transmitted, then the level on a line is that which represents binary 0. A signal transition from 0 to 1 is frequently referred to as the signal's *leading edge*; a transition from 1 to 0 is referred to as a *trailing edge*. Such transitions are not instantaneous, but this transition time is usually small compared with the duration of a signal level. For clarity, the transition is usually depicted as an angled line that exaggerates the relative amount of time that the transition takes. Occasionally, you will see diagrams that use vertical lines, which incorrectly suggests that the transition is instantaneous. On a timing diagram, it may happen that a variable or at least irrelevant amount of time elapses between events of interest. This is depicted by a gap in the time line.

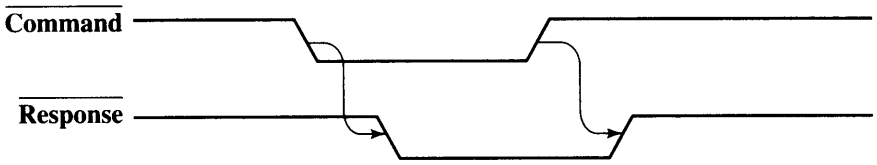
Signals are sometimes represented in groups (Figure 3.27b). For example, if data are transferred a byte at a time, then eight lines are required. Generally, it is not important to know the exact value being transferred on such a group, but rather whether signals are present or not.



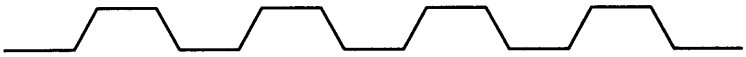
(a) Signal as a function of time



(b) Groups of lines



(c) Cause-and-effect dependencies



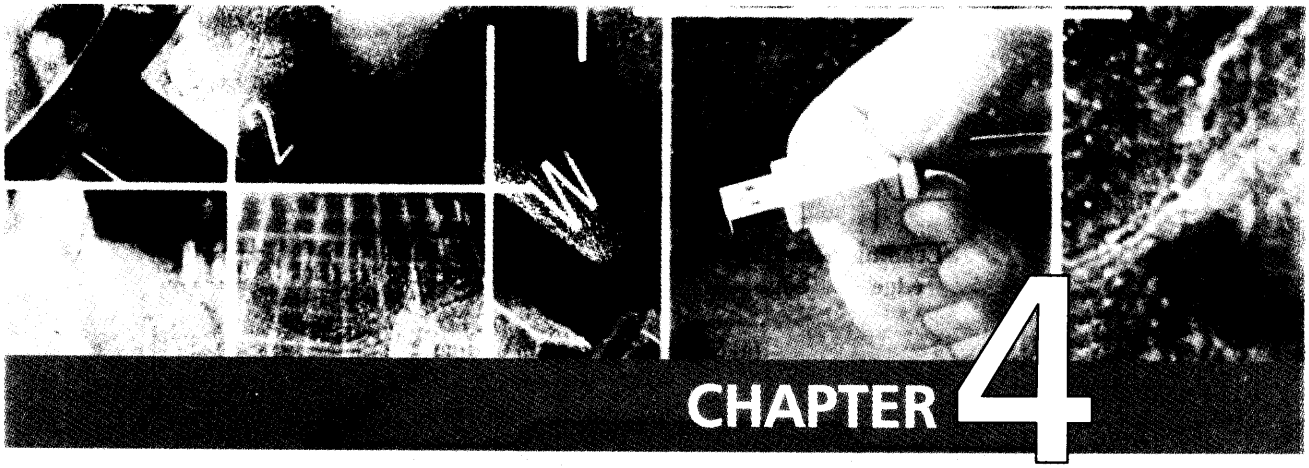
(d) Clock signal

Figure 3.27 Timing Diagrams

A signal transition on one line may trigger an attached device to make signal changes on other lines. For example, if a memory module detects a read control signal (0 or 1 transition), it will place data signals on the data lines. Such cause-and-effect relationships produce sequences of events. Arrows are used on timing diagrams to show these dependencies (Figure 3.27c).

In Figure 3.27c, the overbar over the signal name indicates that the signal is active low as shown. For example, $\overline{\text{Command}}$ is active, or asserted, at 0 volts. This means that $\overline{\text{Command}} = 0$ is interpreted as logical 1, or true.

A clock line is often part of a system bus. An electronic clock is connected to the clock line and provides a repetitive, regular sequence of transitions (Figure 3.27d). Other events may be synchronized to the clock signal.



CACHE MEMORY