In describing computers, a distinction is often made between *computer architecture* and *computer organization.* Although it is difficult to give precise definitions for theseterms, a consensus exists about the general areas covered by each

**Computer architecture** refers to those attributes of a system visible to a programmer or, put another way, those attributes that have a direct impact on the logical execution of a program.

Examples of architectural attributes include the instruction set, the number of bits used to represent various data types (e.g., numbers, characters), I/O mechanisms, and techniques for addressing memory.

**Computer organization** refers to the operational units and their interconnections that realize the architectural specifications. Organizational attributes include those hardware details transparent to the programmer, such as control signals; interfaces between the computer and peripherals; and the memory technology used.

For example, it is an architectural design issue whether a computer will have a multiply instruction. It is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism that makes repeated use of the add unit of the system. The organizational decision may be based on the anticipated frequency of use of the multiply instruction, the relative speed of the two approaches, and the cost and physical size of a special multiply unit.

**STRUCTURE AND FUNCTION OF A COMPUTER**

A computer is a complex system; contemporary computers contain millions of elemen-tary electronic components

**Structure:** The way in which the components are interrelated

**Function:** The operation of each individual component as part of the structure

**RUCTU**

**Function**

Both the structure and functioning of a computer are, in essence, simple. Figure 1.1 depicts the basic functions that a computer can perform. In general terms, there are only four:

Data processing

Data storage

Data movement

Control

The computer, of course, must be able to **process data**. The data may take a wide variety of forms, and the range of processing requirements is broad. However, we shall see that there are only a few fundamental methods or types of data processing.

It is also essential that a computer **store data**. Even if the computer is processing data on the fly (i.e., data come in and get processed, and the results go out immedi-ately), the computer must temporarily store at least those pieces of data that are being worked on at any given moment. Thus, there is at least a short-term data storage func-tion. Equally important, the computer performs a long-term data storage function. Files of data are stored on the computer for subsequent retrieval and update.

The computer must be able to **move data** between itself and the outside world. The computer’s operating environment consists of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process is known as *input–output* (I/O), and the device is referred to as a *peripheral.* When data are moved over longer distances, to or from a remote device, the process is known as *data communications.*

Finally, there must be **control** of these three functions. Ultimately, this control is exercised by the individual(s) who provides the computer with instructions. Within the computer, a control unit manages the computer’s resources and orchestrates the performance of its functional parts in response to those instructions.

**Structure**

The computer interacts in some fashion with its external environment. In general, all of its linkages to the external environment can be classified as peripheral devices or communication lines. We will have something to say about both types of linkages

There are four main structural components:

**Central processing unit (CPU):** Controls the operation of the computer andperforms its data processing functions; often simply referred to as **processor**.

**Main memory:** Stores data.

**I/O:** Moves data between the computer and its external environment.

**System interconnection:** Some mechanism that provides for communication among CPU, main memory, and I/O. A common example of system interconnection is by means of a **system bus**, consisting of a number of conducting wires to which all the other components attach.

The most interesting and in some ways the most complex component is the CPU. Its major structural components are as follows:

**Control unit:** Controls the operation of the CPU and hence the computer

**Arithmetic and logic unit (ALU):** Performs the computer’s data processingfunctions

**Registers:** Provides storage internal to the CPU

**CPU interconnection:** Some mechanism that provides for communicationamong the control unit, ALU, and registers

**Clock Speed and Instructions per Second**

***THE SYSTEM CLOCK***

Operations performed by a processor, such as fetching an instruction, decoding the instruction, performing an arithmetic operation, and so on, are governed by a system clock. Typically, all operations begin with the pulse of the clock. Thus, at the most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).

Typically, clock signals are generated by a quartz crystal, which generates a constant signal wave while power is applied. This wave is converted into a digital voltage pulse stream that is provided in a constant flow to the processor circuitry (Figure 2.14). For example, a 1-GHz processor receives 1 billion pulses per second. The rate of pulses is known as the **clock rate**, or **clock speed**. One increment, or pulse, of the clock is referred to as a **clock cycle**, or a **clock tick**. The time between pulses is the **cycle time**.

The execution of an instruction involves a number of discrete steps, such as fetching the instruction from memory, decoding the various portions of the instruc-tion, loading and storing data, and performing arithmetic and logical operations. Thus, most instructions on most processors require multiple clock cycles to com-plete. Some instructions may take only a few cycles, while others require dozens. In addition, when pipelining is used, multiple instructions are being executed simulta-neously. Thus, a straight comparison of clock speeds on different processors does not tell the whole story about performance.

***INSTRUCTION EXECUTION RATE*** A processor is driven by a clock with a constantfrequency f or, equivalently, a constant cycle time t, where t = 1/*f*. Define the instruction count

**Computer Components**

**Computer Function**

Instruction Fetch and Execute

Interrupts

I/O Function

**Interconnection Structures**

**Bus Interconnection**

Bus Structure

Elements of Bus Design

Data Transfers

Arbitration

**Computer Function**

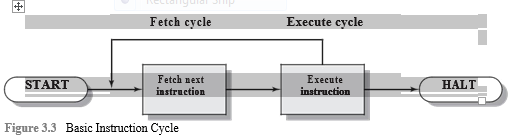
The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. This section provides an overview of the key elements of program execution. In its simplest 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

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 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 in-struction in sequence (i.e., the instruction located at the next higher memory ad-dress). 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,

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



**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 ofone 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): 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 among system modules. These lines, collectively, are called the *data bus*. The data bus may consist of 32, 64, 128, or even more 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 deter-mines how many bits can be transferred at a time. The width of the data bus is a keyfactor in determining overall system performance. For example, if the data bus is 32 bits wide and each instruction is 64 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 among system modules. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed

The operation of the bus is as follows. If one module wishes to send data to an-other, 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

**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.

|  |  |
| --- | --- |
| **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 multi-plexed. 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 ex-ample, 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 mod-ule has a specified period of time to copy the address and determine if it is the ad-dressed 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 connectsonly 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 modulemay 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 bea separate module or part of the processor. In a distributed scheme, there is no cen-tral 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. Busesuse 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 se-quence of alternating 1s and 0s of equal duration. A single 1–0 transmission is re-ferred 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 dia-grams). 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 ex-ample, 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 stabi-lized, 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 mod-ule recognizes the address 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 sig-nal. 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 stabi-lized. 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 ad-dress 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 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.

***BUS WIDTH*** We have already addressed the concept of bus width. The width of thedata 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 are 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 ad-dress 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.

**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).

**INSTRUCTION SETS:**

**CHARACTERISTICS AND FUNCTIONS**

**Machine Instruction Characteristics**

Elements of a Machine Instruction

Instruction Representation

Instruction Types

Number of Addresses

Instruction Set Design

**Types of Operands**

Numbers

Characters

Logical Data

**Types of Operations**

**KEY POINTS**

◆ The essential elements of a computer instruction are the opcode, which specifies the operation to be performed; the source and destination operand references, which specify the input and output locations for the op-eration; and a next instruction reference, which is usually implicit.

◆ Opcodes specify operations in one of the following general categories: arithmetic and logic operations; movement of data between two registers, register and memory, or two memory locations; I/O; and control.

◆ Operand references specify a register or memory location of operand data. The type of data may be addresses, numbers, characters, or logical data.

◆ A common architectural feature in processors is the use of a stack, which may or may not be visible to the programmer. Stacks are used to manage procedure calls and returns and may be provided as an alternative form of addressing memory. The basic stack operations are PUSH, POP, and opera-tions on the top one or two stack locations. Stacks typically are implemented to grow from higher addresses to lower addresses.

◆ Byte-addressable processors may be categorized as big endian, little endi-an, or bi-endian. A multibyte numerical value stored with the most signifi-cant byte in the lowest numerical address is stored in big-endian fashion. The little-endian style stores the most significant byte in the highest numeri-cal address. A bi-endian processor can handle both styles.

MACHINE INSTRUCTION CHARECTERISTICS

The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions.* The collection of different instructions that the processor can execute is referred to as the processor’s *instruction set.*

**Elements of a Machine Instruction**

Each instruction must contain the information required by the processor for execution.

The design of an instruction set is one of the most important aspects of processor design. The design of the instruction set is highly complex because it defines many of the functions performed by the CPU and therefore affects most aspects of the entire system. Each instruction must contain the information required by the CPU for execution.

With most instruction sets, more than one instruction format is used. Each instruction format consists of an opcode field and 0 to 3 operand fields, as follows:



The opcode (stands for operation code) field determines the function of the instruction, and the operand fields provide the addresses of data items (or sometimes the data items themselves). Designing this type of instruction format requires answers to the following questions:

How many instructions are provided? What type of operations are provided?

How many operand fields and what type of operands are allowed in each instruction?

A number of conflicting factors complicate the task of instruction set design. As a result, there are no simple answers to these questions, as the following discussion demonstrates.

**Size of opcode.** The question of how many instructions are provided is directly related to the size of the opcode. The opcode size reflects the number of instructions that can be provided by an architecture; as the number of bits in the opcode increases, the number of instructions will also increase. Having more instructions reduces the size of a program. Smaller programs tend to reduce the storage space and execution time. This is because a sequence of basic instructions can be reinterpreted as equivalent to one advanced instruction. For example, if an instruction set (all the instructions provided by an architecture) includes a multiplication operation, only one instruction is needed in the program to multiply instead of a sequence of add and shift instructions. To summarize, if there are only a few simple instructions to choose from, then many are required, making longer programs to perform a task. If many instructions are available, then ewer are needed, because each instruction will accomplish a longer part of the task. Fewer program instructions means a shorter program.

Although increasing the number of bits in the opcode reduces the program size, ultimately a price must be paid for such an increase. Eventually, the addition of an extra bit to the opcode field will result in increased storage space for a program, despite the initial program size reduction.

Furthermore, increasing the number of instructions will add more complexity to the processor design, which increases the cost. A larger set of instructions requires a more extensive control unit circuit and complicates the process of microprogramming design if microprogramming is used. Additionally, it is ideal to have the whole CPU design on a single chip, since a chip is much faster and less expensive than a board of chips. If design complexity increases, more gates will be needed to successfully implement the design, which could make it impossible to fit the whole design on a single chip.

From this discussion, it can be concluded that the size of an instruction set directly affects even the most fundamental issues involved in processor design. A small and simple instruction set offers the advantage of uncomplicated hardware designs, but also increases program size. A large and complex instruction set decreases program storage needs, but also increases hardware complexity. One trend in computer design is to increase the complexity of the instruction set by providing special instructions that are able to perform complex operations. Recent machines falling within such trends are termed *complex instruction set computers* (**CISC**s). The CISC design approach emphasizes reducing the number of instructions in the program and, as a result, increases overall performance.

Another trend in computer design is to simplify the instruction set rather than make it more complex. As a result, the terminology *reduced instruction set computer* (**RISC**) has been introduced for this type of design. The basic concept of the RISC design approach is based on the observation that in a large number of programs many complex instructions are seldom used. It has been shown that programmers or compilerwriters often use only a subset of the instruction set. Therefore, in contrast to CISC, the RISC design approach employs a simpler instruction set to improve the rate at which instructions can be executed. It emphasizes reducing the average number of clock cycles required to execute an instruction, rather than reducing the number of instructions in the program. In RISC design, most instructions are executed within a single cycle. This innovative approach to computer architecture is covered in more detail in chapters 3 and 4.

TYPES OF INTRASCTION OPERATIONS

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

Data transfer

Arithmetic

Logical

Conversion

I/O

System control

Transfer of control

|  |  |  |
| --- | --- | --- |
| **Type** | **Operation Name** | **Description** |
|  |  |  |
|  | Move (transfer) | Transfer word or block from source to |
|  |  | destination |
|  | Store | Transfer word from processor to memory |
|  | Load (fetch) | Transfer word from memory to processor |
| Data Transfer | Exchange | Swap contents of source and destination |
|  | Clear (reset) | Transfer word of 0s to destination |
|  | Set | Transfer word of 1s to destination |
|  | Push | Transfer word from source to top of stack |
|  | Pop | Transfer word from top of stack to |
|  |  | destination |
|  | Add | Compute sum of two operands |
|  | Subtract | Compute difference of two operands |
|  | Multiply | Compute product of two operands |
| Arithmetic | Divide | Compute quotient of two operands |
| Absolute | Replace operand by its absolute value |
|  |
|  | Negate | Change sign of operand |
|  | Increment | Add 1 to operand |
|  | Decrement | Subtract 1 from operand |
|  |  |  |

**ADDRESSING MODES AND FORMATS**

**Addressing**

Immediate Addressing

Direct Addressing

Indirect Addressing

Register Addressing

Register Indirect Addressing

Displacement Addressing

Stack Addressing

**KEY POINTS**

◆ An operand reference in an instruction either contains the actual value of the operand (immediate) or a reference to the address of the operand. A wide variety of addressing modes is used in various instruction sets. These include direct (operand address is in address field), indirect (address field points to a location that contains the operand address), register, register indirect, and various forms of displacement, in which a register value is added to an address value to produce the operand address.

◆ The instruction format defines the layout fields in the instruction. Instruction format design is a complex undertaking, including such con-sideration as instruction length, fixed or variable length, number of bits assigned to opcode and each operand reference, and how addressing mode is determined.

**Addressing**

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other. In this section, we examine the most common addressing techniques:

Immediate

Direct

Indirect

Register

Register indirect

Displacement

Stack

**Immediate Addressing**

The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

Operand = A

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in twos complement form; the left-most bit of the operand field is used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the left to the full data word size. In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer.

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length

**Direct Addressing**

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

EA = A

The technique was common in earlier generations of computers but is not common on contemporary architectures. It requires only one memory reference and no special calculation. The obvious limitation is that it provides only a limited address space.

**Indirect Addressing**

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as *indirect addressing:*

EA = (A)

:

**Register Addressing**

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

EA = R

To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the

**Register Indirect Addressing**

Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register.Thus, for register indirect address,

EA = (R)

The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

**Displacement Addressing**

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use, but the basic mechanism is the same. We will refer to this as *displacement* *addressing:*

EA = A + (R)

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit.

**ODES AND FORMATS**

is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

We will describe three of the most common uses of displacement addressing:

Relative addressing

Base-register addressing

Indexing

***RELATIVE ADDRESSING*** For relative addressing, also called PC-relative addressing,the implicitly referenced register is the program counter (PC). That is, the next instruction address is added to the address field to produce the EA. Typically, the address field is treated as a twos complement number for this operation. Thus, the effective address is a displacement relative to the address of the instruction.

Relative addressing exploits the concept of locality that was discussed in Chapters 4 and 8. If most memory references are relatively near to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

***BASE-REGISTER ADDRESSING*** For base-register addressing, the interpretation isthe following: The referenced register contains a main memory address, and the ad-dress field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.

Base-register addressing also exploits the locality of memory references. It is a convenient means of implementing segmentation, which was discussed in Chapter 8. In some implementations, a single segment-base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. In this latter case, if the length of the address field is K and the number of possible registers is N*,* then one instruction can reference any one of N areas of 2*K* words.

***INDEXING*** For indexing, the interpretation is typically the following: The addressfield references a main memory address, and the referenced register contains a pos-itive displacement from that address. Note that this usage is just the opposite of the interpretation for base-register addressing. Of course, it is more than just a matter of user interpretation. Because the address field is considered to be a memory address in indexing, it generally contains more bits than an address field in a comparable base-register instruction. Also, we shall see that there are some refinements to in-dexing that would not be as useful in the base-register context. Nevertheless, the method of calculating the EA is the same for both base-register addressing and in-dexing, and in both cases the register reference is sometimes explicit and sometimes implicit (for different processor types).

An important use of indexing is to provide an efficient mechanism for per-forming iterative operations. Consider, for example, a list of numbers stored starting at location A. Suppose that we would like to add 1 to each element on the list. We need to fetch each value, add 1 to it, and store it back. The sequence of effective ad-dresses that we need is A, A + 1, A + 2, . . ., up to the last location on the list. With indexing, this is easily done. The value A is stored in the instruction’s address field, and the chosen register, called an *index register,* is initialized to 0. After each operation, the index register is incremented by 1.

Because index registers are commonly used for such iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some systems will automatically do this as part of the same instruction cycle. This is known as *autoindexing.* If certain registers are devoted exclusively to indexing, then autoindexing can be invoked implicitly and automatically. If general-purpose registers are used, the autoindex operation may need to be signaled by a bit in the instruction. Auto indexing using increment can be depicted as follows.

EA = A + (R)

(R) ; (R) + 1

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: the indexing is performed either before or after the indirection.

If indexing is performed after the indirection, it is termed *postindexing:*

EA = (A) + (R)

First, the contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value. This technique is useful for accessing one of a number of blocks of data of a fixed format. For example, it was described in Chapter 8 that the operating system needs to employ a process control block for each process. The operations performed are the same regardless of which block is being manipulated. Thus, the addresses in the instructions that reference the block could point to a location (value = A) containing a variable pointer to the start of a process control block. The index register contains the displacement within the block.

With *preindexing,* the indexing is performed before the indirection:

EA = (A + (R))

An address is calculated as with simple indexing. In this case, however, the calculated address contains not the operand, but the address of the operand. An example of the use of this technique is to construct a multiway branch table. At a particular point in a program, there may be a branch to one of a number of locations depend-ing on conditions. A table of addresses can be set up starting at location A. By indexing into this table, the required location can be found.

Typically, an instruction set will not include both pre indexing and post indexing.

**Stack Addressing**

The final addressing mode that we consider is stack addressing. As defined in Appendix 9A, a stack is a linear array of locations. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue.* The stack is a reserved block of locations.Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack (Figure 10.14b). The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

**CACHE MEMORY**

**Computer Memory System Overview**

Characteristics of Memory Systems

The Memory Hierarchy

**Cache Memory Principles**

Replacement Algorithms

Write Policy

**KEY POINTS**

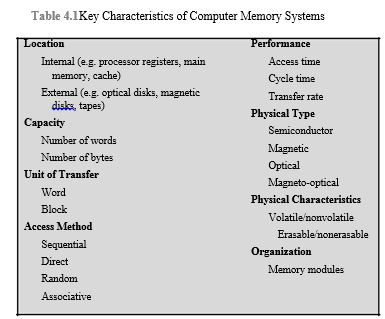
◆ Computer memory is organized into a hierarchy. At the highest level (clos-est to the processor) are the processor registers. Next comes one or more levels of cache, When multiple levels are used, they are denoted L1, L2, and so on. Next comes main memory, which is usually made out of dynamic random-access memory (DRAM). All of these are considered internal to the computer system. The hierarchy continues with external memory, with the next level typically being a fixed hard disk, and one or more levels below that consisting of removable media such as optical disks and tape.

◆ As one goes down the memory hierarchy, one finds decreasing cost/bit, in-creasing capacity, and slower access time. It would be nice to use only the fastest memory, but because that is the most expensive memory, we trade off access time for cost by using more of the slower memory. The design challenge is to organize the data and programs in memory so that the ac-cessed memory words are usually in the faster memory.

◆ In general, it is likely that most future accesses to main memory by the processor will be to locations recently accessed. So the cache automatically retains a copy of some of the recently used words from the DRAM. If the cache is designed properly, then most of the time the processor will request memory words that are already in the cache.

**Characteristics of Memory Systems**

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 4.1.



The term **location** in Table 4.1 refers to whether memory is internal and exter-nal to the computer. Internal memory is often equated with main memory. But there are other forms of internal memory. The processor requires its own local memory, in the form of registers (e.g., see Figure 2.3). Further, as we shall see, the control unit portion of the processor may also require its own internal memory. We will defer dis-cussion of these latter two types of internal memory to later chapters. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.

An obvious characteristic of memory is its **capacity**. For internal memory, this is typically expressed in terms of bytes (1 byte = 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

A related concept is the **unit of transfer**. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

**Word:** The “natural” unit of organization of memory. The size of the word istypically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.

**Addressable units:** In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is 2A = N*.*

**Unit of transfer:** For main memory, this is the number of bits read out of orwritten into memory at a time. The unit of transfer need not equal a word or an

addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

**Sequential access:** Memory is organized into units of data, called records. Ac-cess must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read– write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units, discussed in Chapter 6, are sequential access.

**Direct access:** As with sequential access, direct access involves a sharedread–write mechanism. However, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units, discussed in Chapter 6, are direct access.

**Random access:** Each addressable location in memory has a unique, physicallywired-in addressing mechanism. The time to access a given location is inde-pendent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.

**Associative:** This is a random access type of memory that enables one to makea comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

From a user’s point of view, the two most important characteristics of memory are capacity and **performance**. Three performance parameters are used:

**Access time (latency):** For random-access memory, this is the time it takes toperform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.

**Memory cycle time:** This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.

**Transfer rate:** This is the rate at which data can be transferred into or out of amemory unit. For random-access memory, it is equal to 1/(cycle time).

For non-random-access memory, the following relationship holds:

|  |  |  |
| --- | --- | --- |
| TN = TA + | N |  |
| R |

where

TN = Average time to read or write N bits

TA = Average access time

= Number of bits

= Transfer rate, in bits per second (bps)

A variety of **physical types** of memory have been employed. The most com-mon today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.

Several **physical characteristics** of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are nonvolatile. Semiconductor memory may be either volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as *read-only memory* (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.

For random-access memory, the **organization** is a key design issue. By *organi-zation* is meant the physical arrangement of bits to form words. The obviousarrangement is not always used, as is explained in Chapter 5.

**The Memory Hierarchy**

The design constraints on a computer’s memory can be summed up by three ques-tions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reason-able in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: namely, capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

Faster access time, greater cost per bit

Greater capacity, smaller cost per bit

Greater capacity, slower access time

The dilemma facing the designer is clear. The designer would like to use mem-ory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memo-ries with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 4.1. As one goes down the hierarchy, the following occur:

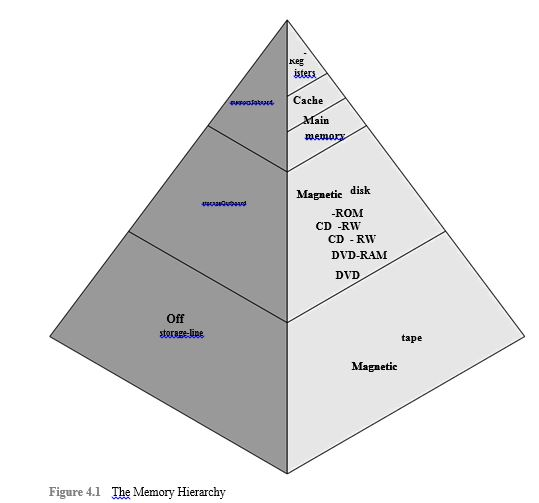
Decreasing cost per bit

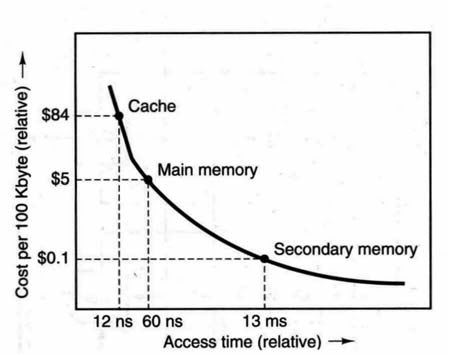
Increasing capacity

Increasing access time

Decreasing frequency of access of the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is item (d): decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter, and virtual memory in Chapter 8. A brief explanation is provided at this point.





**Replacement Algorithms**

Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware. A number of algorithms have been tried. We mention four of the most common. Probably the most effective is least recently used (LRU): Replace that block in the set that has been in the cache longest with no reference to it. LRU should give the best hit ratio.

Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references.

**Write Policy**

When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem oc-curs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably in-validate a word in other caches.

The simplest technique is called **write through**. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck. An alternative technique, known as **write back**, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck. Experience has shown that the percentage of memory references that are writes is on the order of 15%. However, for HPC applications, this number may approach 33% (vector-vector multiplication) and can go as high as 50% (matrix transposition).

**Locality of reference**

By statistical analysis of typical programs, it has been established that at any given interval of time the references to memory tend to be confined within local areas of memory.

This phenomenon is known as the property of *locality of reference*. Three concepts are associated with locality of reference: *temporal, spatial,* and *sequential*. Each of these concepts is defined next.

**Temporal locality.** Items (data or instructions) recently referenced have a good chance to be referenced in the near future. For example, a set of instructions in an iterative loop or in a subroutine may be referenced repeatedly many times.

**Spatial locality.** A program often references items whose addresses are close to each other in the address space. For example, references to the elements of an array always occur within a certain bounded area in the address space.

**Sequential locality.** Most of the instructions in a program are executed in a sequential order. The instructions that might cause out-of-order execution are branches. However, branches construct about 20% to 30% of all instructions; therefore, about 70% to 80% of instructions are often accessed in the same order as they are stored in memory.

**INTERNAL MEMORY**

DRAM and SRAM

Interleaved Memory

**DRAM and SRAM**

All of the memory types that we will explore in this chapter are random access. That is, individual words of memory are directly accessed through wired-in addressing logic.

Table 5.1 lists the major types of semiconductor memory. The most common is referred to as *random-access memory* (RAM). This is, of course, a misuse of the term, because all of the types listed in the table are random access. One distinguish-ing characteristic of RAM is that it is possible both to read data from the memory and to write new data into the memory easily and rapidly. Both the reading and writing are accomplished through the use of electrical signals.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | **Write** |  |
| **Memory Type** | **Category** | **Erasure** | **Mechanism** | **Volatility** |
|  |  |  |  |  |
| Random-access memory | Read-write | Electrically, | Electrically | Volatile |
| (RAM) | memory | byte-level |
|  |  |
| Read-only memory (ROM) | Read-only | Not possible | Masks |  |
| Programmable ROM (PROM) | memory |  |  |
|  |  |  |
| Erasable PROM (EPROM) |  | UV light, |  |  |
|  | chip-level |  | Nonvolatile |
|  |  |  |
| Electrically Erasable PROM | Read-mostly | Electrically, | Electrically |  |
| (EEPROM) | memory | byte-level |  |  |
| Flash memory |  | Electrically, |  |  |
|  | block-level |  |  |
|  |  |  |  |
|  |  |  |  |  |

The other distinguishing characteristic of RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage. The two traditional forms of RAM used in computers are DRAM and SRAM.

***DYNAMIC RAM*** RAM technology is divided into two technologies: dynamic andstatic. A dynamic RAM (DRAM) is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a bi-nary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAMs require periodic charge refreshing to maintain data storage. The term *dynamic* refers to this tendency of the stored charge to leak away, even with powercontinuously applied.

***STATIC RAM*** In contrast, a static RAM (SRAM) is a digital device that uses thesame logic elements used in the processor. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations (see Chapter 20 for a description of flip-flops). A static RAM will hold its data as long as power is supplied to it.

***SRAM VERSUS DRAM*** Both static and dynamic RAMs are volatile; that is, powermust be continuously supplied to the memory to preserve the bit values. A dynamic memory cell is simpler and smaller than a static memory cell. Thus, a DRAM is more dense (smaller cells = more cells per unit area) and less expensive than a corresponding SRAM. On the other hand, a DRAM requires the supporting refresh cir-cuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the smaller variable cost of DRAM cells. Thus, DRAMs tend to be favored for large memory requirements. A final point is that SRAMs are generally somewhat faster than DRAMs. Because of these relative characteristics, SRAM is used for cache memory (both on and off chip), and DRAM is used for main memory.

**Interleaved Memory**

Main memory is composed of a collection of DRAM memory chips. A number of chips can be grouped together to form a *memory bank*. It is possible to organize the memory banks in a way known as interleaved memory. Each bank is independently able to service a memory read or write request, so that a system with K banks can service K re-quests simultaneously, increasing memory read or write rates by a factor of K. If consecutive words of memory are stored in different banks, then the transfer of a block of memory is speeded up. Appendix E explores the topic of interleaved memory.

**EXTERNAL MEMORY**

Disk Performance Parameters

**Raid**

RAID Level 0

RAID Level 1

RAID Level 2

RAID Level 3

RAID Level 4

RAID Level 5

RAID Level 6

**KEY POINTS**

◆ Magnetic disks remain the most important component of external memory. Both removable and fixed, or hard, disks are used in systems ranging from personal computers to mainframes and supercomputers.

◆ To achieve greater performance and higher availability, servers and larger systems use RAID disk technology. RAID is a family of techniques for using multiple disks as a parallel array of data storage devices, with redundancy built in to compensate for disk failure.

◆ Optical storage technology has become increasingly important in all types of computer systems. While CD-ROM has been widely used for many years, more recent technologies, such as writable CD and DVD, are becoming increasingly important.

**Disk Performance Parameters**

The actual details of disk I/O operation depend on the computer system, the operating system, and the nature of the I/O channel and disk controller hardware.

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek** **time**. In either case, once the track is selected, the disk controller waits until theappropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **rotational delay**, or *rotational latency*. The sum of the seek time, if any, and the rotational delay equals the **access time**, which is the time it takes to get into position to read or write. Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the **transfer time**.

In addition to the access time and transfer time, there are several queuing delays normally associated with a disk I/O operation. When a process issues an I/O request, it must first wait in a queue for the device to be available. At that time, the device is assigned to the process. If the device shares a single I/O channel or a set of I/O channels with other disk drives, then there may be an additional wait for the channel to be available. At that point, the seek is performed to begin disk access.

In some high-end systems for servers, a technique known as rotational positional sensing (RPS) is used. This works as follows: When the seek command has been issued, the channel is released to handle other I/O operations. When the seek is completed, the device determines when the data will rotate under the head. As that sector approaches the head, the device tries to reestablish the communication path back to the host. If either the control unit or the channel is busy with another I/O, then the reconnection attempt fails and the device must rotate one whole revolution before it can attempt to reconnect, which is called an RPS miss. This is an extra delay element that must be added to the timeline of Figure 6.7.

***SEEK TIME*** Seek time is the time required to move the disk arm to the requiredtrack. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components: the initial startup time, and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed. Unfortunately, the traversal time is not a linear function of the number of tracks, but includes a settling time (time after positioning the head over the target track until track identification is confirmed).

Much improvement comes from smaller and lighter disk components. Some years ago, a typical disk was 14 inches (36 cm) in diameter, whereas the most com-mon size today is 3.5 inches (8.9 cm), reducing the distance that the arm has to travel. A typical average seek time on contemporary hard disks is under 10 ms.

***ROTATIONAL DELAY*** Disks, other than floppy disks, rotate at speeds ranging from3600 rpm (for handheld devices such as digital cameras) up to, as of this writing, 20,000 rpm; at this latter speed, there is one revolution per 3 ms. Thus, on the aver-age, the rotational delay will be 1.5 ms.

***TRANSFER TIME*** The transfer time to or from the disk depends on the rotationspeed of the disk in the following fashion:

T = rNb

where

1. = transfer time
2. = number of bytes to be transferred
3. = number of bytes on a track
4. = rotation speed, in revolutions per second

Thus the total average access time can be expressed as

Ta = Ts + 21r + rNb

where *Ts* is the average seek time. Note that on a zoned drive, the number of bytes per track is variable, complicating the calculation.2

***A TIMING COMPARISON*** With the foregoing parameters defined, let us look at twodifferent I/O operations that illustrate the danger of relying on average values. Consider a disk with an advertised average seek time of 4 ms, rotation speed of 15,000 rpm, and 512-byte sectors with 500 sectors per track. Suppose that we wish to read a file consisting of 2500 sectors for a total of 1.28 Mbytes. We would like to estimate the total time for the transfer.

First, let us assume that the file is stored as compactly as possible on the disk. That is, the file occupies all of the sectors on 5 adjacent tracks (5 tracks × 500 sectors/ track = 2500 sectors). This is known as *sequential organization*. Now, the time to read the first track is as follows:

|  |  |
| --- | --- |
| Average seek | 4 ms |
| Average rotational delay | 2 ms |
| Read 500 sectors | 4 ms |
|  | 10 ms |

Suppose that the remaining tracks can now be read with essentially no seek time. That is, the I/O operation can keep up with the flow from the disk. Then, at most, we need to deal with rotational delay for each succeeding track. Thus each successive track is read in 2 + 4 = 6 ms. To read the entire file,

Total time = 10 + (4 \* 6) = 34 ms = 0.034 seconds

Now let us calculate the time required to read the same data using random access rather than sequential access; that is, accesses to the sectors are distributed randomly over the disk. For each sector, we have

|  |  |  |
| --- | --- | --- |
| Average seek | 4 | ms |
| Rotational delay | 2 | ms |
| Read 1 sectors | 0.008 ms | |
|  |  | |
|  | 6.008 ms | |

Total time = 2500 \* 6.008 = 15020 ms = 15.02 seconds

**RAID**

The rate in improvement in secondary storage performance has been considerably less than the rate for processors and main memory. This mismatch has made the disk storage system perhaps the main focus of concern in improving overall computer system performance.

As in other areas of computer performance, disk storage designers recognize that if one component can only be pushed so far, additional gains in performance are to be had by using multiple parallel components. In the case of disk storage, this leads to the development of arrays of disks that operate independently and in parallel. With multiple disks, separate I/O requests can be handled in parallel, as long as the data required reside on separate disks. Further, a single I/O request can be executed in parallel if the block of data to be accessed is distributed across multiple disks.

With the use of multiple disks, there is a wide variety of ways in which the data can be organized and in which redundancy can be added to improve reliability. This could make it difficult to develop database schemes that are usable on a number of platforms and operating systems. Fortunately, industry has agreed on a standardized scheme for multiple-disk database design, known as RAID (Redundant Array of Independent Disks). The RAID scheme consists of seven levels, zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

RAID is a set of physical disk drives viewed by the operating system as a single logical drive.

1. Data are distributed across the physical drives of an array in a scheme known as striping, described subsequently.
2. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

The details of the second and third characteristics differ for the different RAID levels. RAID 0 and RAID 1 do not support the third characteristic.

The term *RAID* was originally coined in a paper by a group of researchers at the University of California at Berkeley The paper outlined various RAID configurations and applications and introduced the definitions of the RAID levels that are still used. The RAID strategy employs multiple disk drives and dis-tributes data in such a way as to enable simultaneous access to data from multiple drives, thereby improving I/O performance and allowing easier incremental in-creases in capacity.

The unique contribution of the RAID proposal is to address effectively the need for redundancy. Although allowing multiple heads and actuators to operate simultaneously achieves higher I/O and transfer rates, the use of multiple devices increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

We now examine each of the RAID levels. Table 6.3 provides a rough guide to the seven levels. In the table, I/O performance is shown both in terms of data trans-fer capacity, or ability to move data, and I/O request rate, or ability to satisfy I/O re-quests, since these RAID levels inherently perform differently relative to these two

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Table 6.3** | RAID Levels | |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | **Disks** |  | **Large I/O Data** | **Small I/O** |
| **Category** |  | **Level** | **Description** | **Required** | **Data Availability** | **Transfer Capacity** | **Request Rate** |
|  |  |  |  |  |  |  |  |
| Striping |  | 0 | No redundant | N | Lower than | Very high | Very high for both read |
|  | single disk | and write |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  | Higher than RAID 2, | Higher than single disk | Up to twice that of a |
|  |  |  |  |  | single disk for read; |
| Mirroring |  | 1 | Mirrored | 2N | 3, 4, or 5; lower than | for read; similar to sin- |
|  | similar to single disk |
|  |  |  |  |  | RAID 6 | gle disk for write |
|  |  |  |  |  | for write |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | Redundant via Ham |  | Much higher than single | Highest of all listed | Approximately twice |
|  |  | 2 | N + m | disk; comparable to |
|  |  | ming code | alternatives | that of a single disk |
|  |  |  |  | RAID 3, 4, or 5 |
| Parallel access | |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  | Much higher than single | Highest of all listed | Approximately twice |
|  |  |  |  |  |
|  |  | 3 | Bit-interleaved parity | N + 1 | disk; comparable to |
|  |  | alternatives | that of a single disk |
|  |  |  |  |  | RAID 2, 4, or 5 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  | Much higher than single | Similar to RAID 0 for | Similar to RAID 0 for |
|  |  |  | Block-interleaved |  | read; significantly lower | read; significantly lower |
|  |  | 4 | N + 1 | disk; comparable to |
|  |  | parity | than single disk for | than single disk for |
|  |  |  |  | RAID 2, 3, or 5 |
|  |  |  |  |  | write | write |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Independent | |  |  |  | Much higher than single | Similar to RAID 0 for | Similar to RAID 0 for |
|  | Block-interleaved |  | read; generally lower |
| access |  | 5 | N + 1 | disk; comparable to | read; lower than single |
|  | distributed parity | than single disk for |
|  |  |  |  | RAID 2, 3, or 4 | disk for write |
|  |  |  |  |  | write |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | Block-interleaved |  | Highest of all listed | Similar to RAID 0 for | Similar to RAID 0 for |
|  |  | 6 | dual distributed | N + 2 | read; lower than RAID | read; significantly lower |
|  |  | alternatives |
|  |  |  | parity |  | 5 for write | than RAID 5 for write |
|  |  |  |  |  |
|  |  |  |  |  |  |  |  |

**INPUT/OUTPUT**

**7.1 External Devices**

Keyboard/Monitor

Disk Drive

**7.2 I/O Modules**

Module Function

I/O Module Structure

**7.3 Programmed I/O**

Overview of Programmed I/O

I/O Commands

I/O Instructions

**7.4 Interrupt-Driven I/O**

Interrupt Processing

Design Issues

**7.5 Direct Memory Access**

Drawbacks of Programmed and Interrupt-Driven I/O

DMA Function

**KEY POINTS**

◆ The computer system’s I/O architecture is its interface to the outside world. This architecture provides a systematic means of controlling interaction with the outside world and provides the operating system with the information it needs to manage I/O activity effectively.

◆ The are three principal I/O techniques: **programmed I/O**, in which I/O occurs under the direct and continuous control of the program requesting the I/O operation; **interrupt-driven I/O**, in which a program issues an I/O command and then continues to execute, until it is interrupted by the I/O hard-ware to signal the end of the I/O operation; and **direct memory access** **(DMA)**, in which a specialized I/O processor takes over control of an I/Ooperation to move a large block of data.

◆ Two important examples of external I/O interfaces are **FireWire** and **Infiniband**.

In addition to the processor and a set of memory modules, the third key element of a computer system is a set of I/O modules. Each module interfaces to the system bus or central switch and controls one or more peripheral devices. An I/O module is not simply a set of mechanical connectors that wire a device into the system bus. Rather, the I/O module contains logic for performing a communication function between the peripheral and the bus.

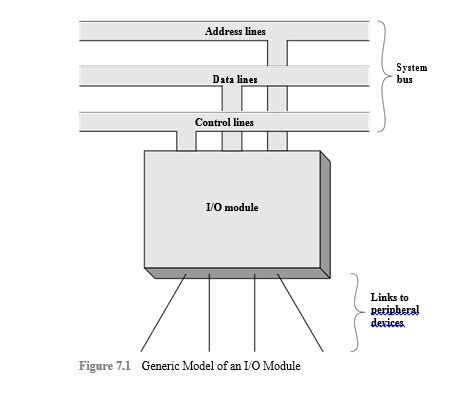
The reader may wonder why one does not connect peripherals directly to the system bus. The reasons are as follows:

There are a wide variety of peripherals with various methods of operation. It would be impractical to incorporate the necessary logic within the processor to control a range of devices.

The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.

On the other hand, the data transfer rate of some peripherals is faster than that of the memory or processor. Again, the mismatch would lead to inefficiencies if not managed properly.

Peripherals often use different data formats and word lengths than the computer to which they are attached.



Thus, an I/O module is required. This module has two major functions (Figure 7.1):

* Interface to the processor and memory via the system bus or central switch
* Interface to one or more peripheral devices by tailored data links

**EXTERNAL DEVICES**

I/O operations are accomplished through a wide assortment of external devices that provide a means of exchanging data between the external environment and the computer. An external device attaches to the computer by a link to an I/O module (Figure 7.1). The link is used to exchange control, status, and data between the I/O module and the external device. An external device connected to an I/O module is often referred to as a *peripheral device* or, simply, a *peripheral.*

We can broadly classify external devices into three categories:

1. **Human readable:** Suitable for communicating with the computer user
2. **Machine readable:** Suitable for communicating with equipment
3. **Communication:** Suitable for communicating with remote devices

Examples of human-readable devices are video display terminals (VDTs) and printers. Examples of machine-readable devices are magnetic disk and tape systems, and sensors and actuators, such as are used in a robotics application. Note that we are viewing disk and tape systems as I/O devices in this chapter, whereas in Chapter 6 we viewed them as memory devices. From a functional point of view, these devices are part of the memory hierarchy, and their use is appropriately discussed in Chapter 6. From a structural point of view, these devices are controlled by I/O modules and are hence to be considered in this chapter.

Communication devices allow a computer to exchange data with a remote device, which may be a human-readable device, such as a terminal, a machine-readable device, or even another computer.

**I/O MODULE**

**Module Function**

The major functions or requirements for an I/O module fall into the following categories:

* 1. Control and timing
  2. Processor communication
  3. Device communication
  4. Data buffering
  5. Error detection

During any period of time, the processor may communicate with one or more external devices in unpredictable patterns, depending on the program’s need for I/O. The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O. Thus, the I/O function includes a **control and timing** requirement, **to coordinate the flow of traffic between internal resources and external devices.** For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps:

The processor interrogates the I/O module to check the status of the attached device.

* + The I/O module returns the device status.
  + If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
  + The I/O module obtains a unit of data (e.g., 8 or 16 bits) from the external device.
  + The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations

The preceding simplified scenario also illustrates that the I/O module must communicate with the processor and with the external device. **Processor communication** involves the following:

1. **Command decoding:** The I/O module accepts commands from the processor,typically sent as signals on the control bus. For example, an I/O module for a disk drive might accept the following commands: READ SECTOR, WRITE SECTOR, SEEK track number, and SCAN record ID. The latter two commands each include a parameter that is sent on the data bus.
2. **Data:** Data are exchanged between the processor and the I/O module over thedata bus.
3. **Status reporting:** Because peripherals are so slow, it is important to know thestatus of the I/O module. For example, if an I/O module is asked to send data to the processor (read), it may not be ready to do so because it is still working on the previous I/O command. This fact can be reported with a status signal. Common status signals are **BUSY** and **READY.** There may also be signals to report various error conditions.
4. **Address recognition:** Just as each word of memory has an address, so doeseach I/O device. Thus, an I/O module must recognize one unique address for each peripheral it controls.

On the other side, the I/O module must be able to perform **device communication**.

This communication involves commands, status information, and data

**Data buffering.** Whereas the transfer rate into and out of main memory or the processor is quite high, the rate is orders of magnitude lower for many peripheral devices and covers a wide range. Data coming from main memory are sent to an I/O module in a rapid burst. The data are buffered in the I/O module and then sent to the peripheral device at its data rate. In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation. Thus, the I/O module must be able to operate at both device and memory speeds. Similarly, if the I/O device operates at a rate higher than the memory access rate, then the I/O module performs the needed buffering operation.

Finally, an I/O module is often responsible for **error detection** and for subsequently reporting errors to the processor. One class of errors includes mechanical and electrical malfunctions reported by the device (e.g., paper jam, bad disk track). Another class consists of unintentional changes to the bit pattern as it is transmitted from device to I/O module. Some form of error-detecting code is often used to detect transmission errors. A simple example is the use of a parity bit on each character of data. For example, the IRA character code occupies 7 bits of a byte. The eighth bit is set so that the total number of 1s in the byte is even (even parity) or odd (odd parity). When a byte is received, the I/O module checks the parity to determine whether an error has occurred.

**I/O OPERATION**

Three techniques are possible for I/O operations.

*Programmed I/O,* data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is wasteful of processor time.

*Interrupt-driven I/O,* the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work. With both programmed and interrupt I/O, the processor responsible for extracting data from main memory for output and storing data in main memory for input.

*Direct memory access* (DMA). In this mode, the I/O module and main memory exchange data directly, without processor involvement

**Design Issues**

Two design issues arise in implementing interrupt I/O. First, because there will al-most invariably be multiple I/O modules, how does the processor determine which device issued the interrupt? And second, if multiple interrupts have occurred, how does the processor decide which one to process?

The most straightforward approach to the problem is to provide **multiple interrupt lines** between the processor and the I/O modules. However, it is impractical todedicate more than a few bus lines or processor pins to interrupt lines. Consequently, even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it. Thus, one of the other three techniques must be used on each line

One alternative is the **software poll**. When the processor detects an interrupt, it branches to an interrupt-service routine whose job it is to poll each I/O module to determine which module caused the interrupt. The poll could be in the form of a separate command line (e.g., TESTI/O). In this case, the processor raises TESTI/O and places the address of a particular I/O module on the address lines. The I/O mod-ule responds positively if it set the interrupt. Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module. Once the correct module is identified, the processor branches to a device-service routine specific to that device.

The disadvantage of the software poll is that it is time consuming. A more efficient technique is to use a **daisy chain**, which provides, in effect, a hardware poll. An example of a daisy-chain configuration is shown in Figure 3.26. For interrupts, all I/O modules share a common interrupt request line. The interrupt acknowledge line is daisy chained through the modules. When the processor senses an interrupt, it sends out an interrupt acknowledge. This signal propagates through a series of I/O modules until it gets to a requesting module. The requesting module typically re-sponds by placing a word on the data lines. This word is referred to as a *vector* and is either the address of the I/O module or some other unique identifier. In either case, the processor uses the vector as a pointer to the appropriate device-service routine. This avoids the need to execute a general interrupt-service routine first. This technique is called a *vectored interrupt.*

There is another technique that makes use of vectored interrupts, and that is **bus arbitration**. With bus arbitration, an I/O module must first gain control of thebus before it can raise the interrupt request line. Thus, only one module can raise the line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places its vector on the data lines.

The aforementioned techniques serve to identify the requesting I/O module. They also provide a way of assigning priorities when more than one device is re-questing interrupt service. With multiple lines, the processor just picks the interrupt line with the highest priority. With software polling, the order in which modules are polled determines their priority. Similarly, the order of modules on a daisy chain determines their priority.