MODULE 1

CHAPTER 2

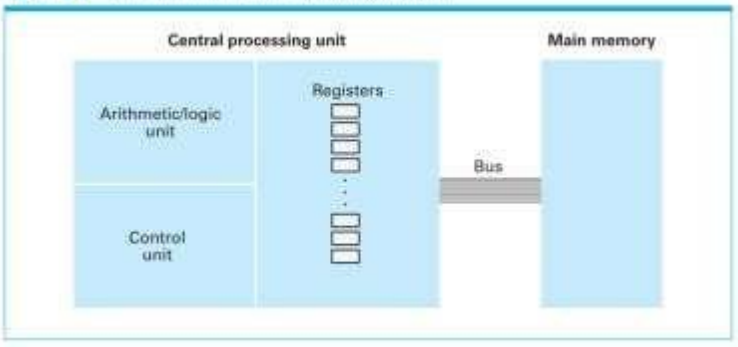
2.1 Computer Architecture

- The circuitry in a computer that controls the manipulation of data is called the central processing unit, or CPU (often referred to as merely the processor).
- In the machines of the midtwentieth century, CPUs were large units comprised of perhaps several racks of electronic circuitry that reflected the significance of the unit.
- However, technology has shrunk these devices drastically.
- The CPUs found in today's desktop computers and notebooks are packaged as small flat squares (approximately two inches by two inches) whose connecting pins plug into a socket mounted on the machine's main circuit board (called the motherboard).
- In smartphones, mininotebooks, and other Mobile Internet Devices (MID), CPUs are around half the size of a postage stamp.
- Due to their small size, these processors are called microprocessors.

CPU Basics

- A CPU consists of three parts (Figure 2.1): the **arithmetic/logic unit**, which contains the circuitry that performs **operations on data** (such as **addition** and **subtraction**); the **control unit**, which contains the circuitry for **coordinating the machine's activities**; and the **register unit**, which contains **data storage cells** (similar to **main memory cells**), called **registers**, that are used for **temporary storage of information** within the CPU.
- Some of the **registers** within the **register unit** are considered **general-purpose registers**, whereas others are **special-purpose registers**.

Figure 2.1 CPU and main memory connected via a bus



- **Generalpurpose registers** serve as **temporary holding places** for **data** being manipulated by the **CPU**.
- These **registers** hold the **inputs** to the **arithmetic/logic unit's circuitry** and provide **storage space** for **results** produced by that unit.
- To perform an **operation on data stored in main memory**, the **control unit** transfers the **data** from **memory** into the **generalpurpose registers**.
- The **control unit** informs the **arithmetic/logic unit** which **registers** hold the **data**.
- The **control unit** activates the **appropriate circuitry** within the **arithmetic/logic unit**.
- The **control unit** tells the **arithmetic/logic unit** which **register** should receive the **result**.
- For the purpose of **transferring bit patterns**, a machine's **CPU** and **main memory** are connected by a collection of wires called a **bus** (see again Figure 2.1).
- Through this **bus**, the **CPU** extracts (**reads**) **data** from **main memory** by supplying the **address** of the pertinent **memory cell** along with an **electronic signal** telling the **memory circuitry** that it is supposed to **retrieve the data** in the indicated **cell**.
- In a similar manner, the **CPU** places (**writes**) **data** in **memory** by providing the **address** of the **destination cell** and the **data** to be **stored** together with the appropriate **electronic signal** telling **main memory** that it is supposed to **store the data** being sent to it.
- Based on this design, the task of **adding two values stored in main memory** involves more than the mere execution of the **addition operation**.
- The **data** must be **transferred** from **main memory** to **registers** within the **CPU**, the **values** must be **added** with the **result** being placed in a **register**, and the **result** must then be **stored** in a **memory cell**.
- The entire process is summarized by the **five steps** listed in **Figure 2.2**.

Figure 2.2 Adding values stored in memory

The Stored-Program Concept

- **Early computers were not known for their flexibility**—the steps that each device executed were **built into the control unit** as a part of the machine.
- To gain more **flexibility**, some of the **early electronic computers** were designed so that the **CPU** could be conveniently **rewired**.
- This **flexibility** was accomplished by means of a **pegboard arrangement** similar to old **telephone switchboards** in which the ends of **jumper wires** were plugged into **holes**.
- A breakthrough (credited, apparently incorrectly, to John von Neumann) came with the realization that a program, just like data, can be encoded and stored in main memory.
- If the control unit is designed to extract the program from memory, decode the instructions, and execute them, the program that the machine follows can be changed merely by changing the contents of the computer's memory instead of rewiring the CPU.
- The idea of storing a computer's program in its main memory is called the **stored-program concept** and has become the standard approach used today—so standard, in fact, that it seems obvious.
- What made it difficult originally was that everyone thought of programs and data as different entities.
- Data were stored in memory; programs were part of the CPU.
- The result was a prime example of not seeing the forest for the trees.

- It is easy to be caught in such ruts, and the development of computer science might still be in many of them today without our knowing it.
- Indeed, part of the excitement of the science is that new insights are constantly opening doors to new theories and applications.

2.2 Machine Language

- CPUs are designed to recognize instructions encoded as bit patterns. This collection of instructions along with the encoding system is called the machine language.
- An instruction expressed in this language is called a machine level instruction or, more commonly, a machine instruction.

The Instruction Repertoire

- The degree to which machine designs should take advantage of this fact has led to two philosophies of CPU architecture.
- One is that a CPU should be designed to execute a minimal set of machine instructions.
- This approach leads to what is called a reduced instruction set computer (RISC).
- The argument in favor of RISC architecture is that such a machine is efficient, fast, and less expensive to manufacture.
- On the other hand, others argue in favor of CPUs with the ability to execute a large number of complex instructions, even though many of them are technically redundant.
- The result of this approach is known as a complex instruction set computer (CISC).
- The argument in favor of CISC architecture is that the more complex CPU can better cope with the everincreasing complexities of today's software.
- With CISC, programs can exploit a powerful rich set of instructions, many of which would require a multiinstruction sequence in a RISC design.
- In the 1990s and into the millennium, commercially available CISC and RISC processors were actively competing for dominance in desktop computing.
- Intel processors, used in PCs, are examples of CISC architecture; PowerPC processors (developed by an alliance between Apple, IBM, and Motorola) are examples of RISC architecture and were used in the Apple Macintosh.

- As time progressed, the manufacturing cost of CISC was drastically reduced; thus Intel's processors (or their equivalent from AMD—Advanced Micro Devices, Inc.) are now found in virtually all desktop and laptop computers (even Apple is now building computers based on Intel products).
- While CISC secured its place in desktop computers, it has an insatiable thirst for electrical power.
- In contrast, the company Advanced RISC Machine (ARM) has designed a RISC architecture specifically for low power consumption.
- (Advanced RISC Machine was originally Acorn Computers and is now ARM Holdings.)
- Thus, ARMbased processors, manufactured by a host of vendors including Qualcomm and Texas Instruments, are readily found in game controllers, digital TVs, navigation systems, automotive modules, cellular telephones, smartphones, and other consumer electronics.
- Regardless of the choice between RISC and CISC, a machine's instructions can be categorized into three groupings:
  (1) the data transfer group
  (2) the arithmetic/logic group
  (3) the control group.

1. Data Transfer:

- The **data transfer group** consists of **instructions** that request the **movement of data** from one **location** to another.
- **Steps 1, 2, and 4 in Figure 2.2** fall into this category.
- We should note that using terms such as **transfer** or **move** to identify this group of **instructions** is actually a **misnomer**.
- It is rare that the **data** being **transferred** is erased from its **original location**.
- The **process** involved in a **transfer instruction** is more like **copying the data** rather than moving it.
- Thus terms such as **copy** or **clone** better describe the actions of this group of **instructions**.
- While on the subject of **terminology**, we should mention that special terms are used when referring to the **transfer of data** between the **CPU** and **main memory**.

- A request to fill a **general-purpose register** with the contents of a **memory cell** is commonly referred to as a **LOAD instruction**.
- Conversely, a request to transfer the contents of a **register** to a **memory cell** is called a **STORE instruction**.
- In **Figure 2.2**, **Steps 1 and 2** are **LOAD instructions**, and **Step 4** is a **STORE instruction**.
- An important group of **instructions** within the **data transfer category** consists of the **commands** for communicating with **devices outside the CPU-main memory context** (printers, keyboards, display screens, disk drives, etc.).
- Since these **instructions** handle the **input/output (I/O) activities** of the machine, they are called **I/O instructions** and are sometimes considered as a **category in their own right**.

2. Arithmetic/Logic:

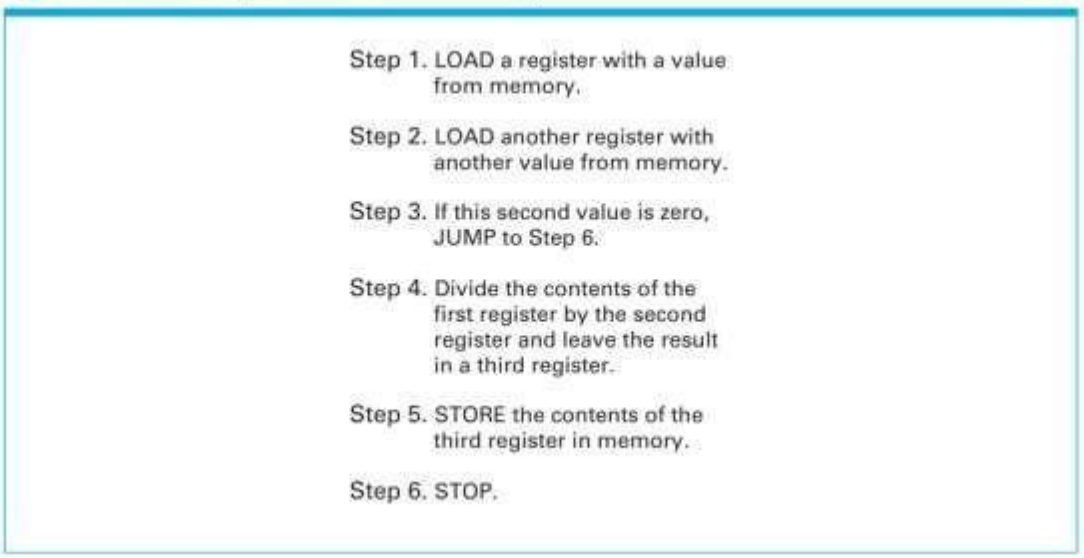
- The **arithmetic/logic group** consists of the **instructions** that tell the **control unit** to request an **activity** within the **arithmetic/logic unit**.
- **Step 3 in Figure 2.2** falls into this group.
- As its name suggests, the **arithmetic/logic unit** is capable of performing **operations** other than the **basic arithmetic operations**.
- Some of these additional **operations** are the **Boolean operations AND, OR, and XOR**.
- Another collection of **operations** available within most **arithmetic/logic units** allows the **contents of registers** to be moved to the **right** or the **left** within the **register**.
- These **operations** are known as either **SHIFT** or **ROTATE operations**, depending on whether the **bits** that "fall off the end" of the **register** are merely **discarded (SHIFT)** or are used to **fill the holes** left at the other end (**ROTATE**).

3. Control:

- The control group consists of those instructions that direct the execution of the program rather than the manipulation of data.
- Step 5 in Figure 2.2 falls into this category, although it is an extremely elementary example.

- This group contains many of the more interesting instructions in a machine's repertoire, such as the family of JUMP (or BRANCH) instructions used to direct the CPU to execute an instruction other than the next one in the list.
- These JUMP instructions appear in two varieties: unconditional jumps and conditional jumps.
- An example of the former would be the instruction "Skip to Step 5"; an example of the latter would be, "If the value obtained is 0, then skip to Step 5."
- The distinction is that a conditional jump results in a "change of venue" only if a certain condition is satisfied.
- As an example, the sequence of instructions in Figure 2.3 represents an algorithm for dividing two values where Step 3 is a conditional jump that protects against the possibility of division by zero.

Figure 2.3 Dividing values stored in memory

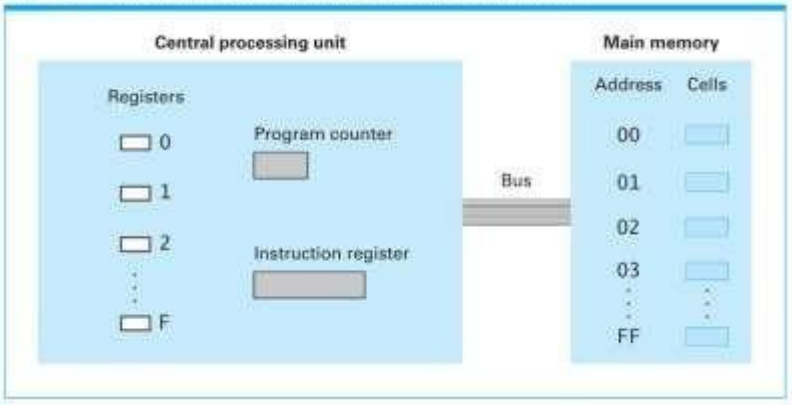


An Illustrative Machine Language

- The machine that we will use for our discussion is summarized in Figure 2.4.
- It has 16 general-purpose registers and 256 main memory cells, each with a capacity of 8 bits.
- For referencing purposes, we label the registers with the values 0 through 15 and address the memory cells with the values 0 through 255.
- For convenience we think of these labels and addresses as values represented in base two and compress the resulting bit patterns using hexadecimal notation.

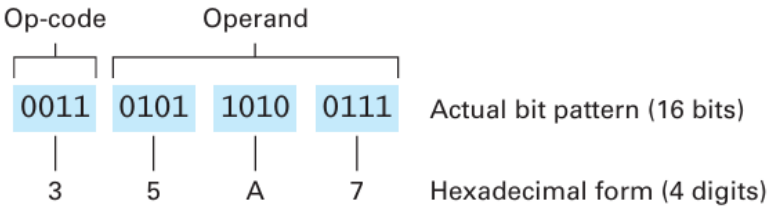
- Thus, the registers are labeled 0 through F, and the memory cells are addressed 00 through FF.
- The encoded version of a machine instruction consists of two parts: the op-code (short for operation code) field and the operand field.
- The bit pattern appearing in the opcode field indicates which of the elementary operations, such as STORE, SHIFT, XOR, and JUMP, is requested by the instruction.
- The bit patterns found in the operand field provide more detailed information about the operation specified by the opcode.
- For example, in the case of a STORE operation, the information in the operand field indicates which register contains the data to be stored and which memory cell is to receive the data.

Figure 2.4 The architecture of the machine described in Appendix C



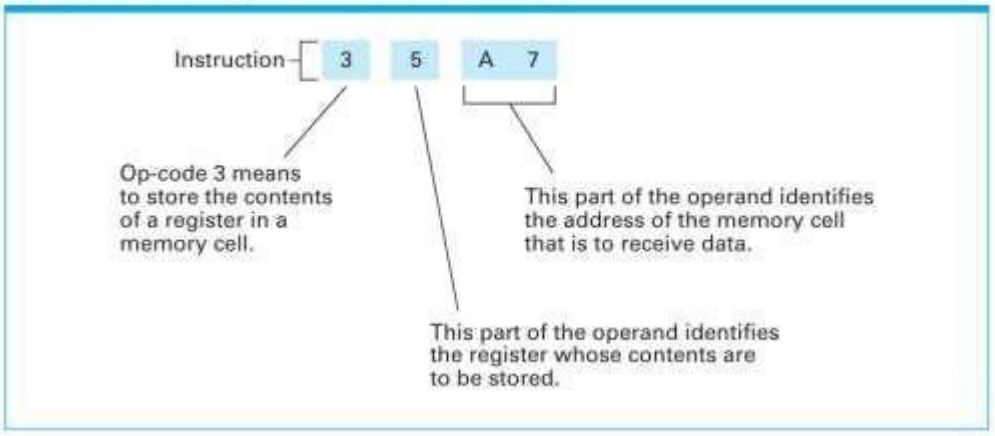
- The entire machine language of our illustrative machine (Appendix C) consists of only twelve basic instructions.
- Each of these instructions is encoded using a total of 16 bits, represented by four hexadecimal digits (Figure 2.5).
- The opcode for each instruction consists of the first 4 bits or, equivalently, the first hexadecimal digit.
- Note (Appendix C) that these opcodes are represented by the hexadecimal digits 1 through C.
- In particular, the table in Appendix C shows us that an instruction beginning with the hexadecimal digit 3 refers to a STORE instruction, and an instruction beginning with hexadecimal A refers to a ROTATE instruction.

Figure 2.5 The composition of an instruction for the machine in Appendix C



- The operand field of each instruction in our illustrative machine consists of three hexadecimal digits (12 bits), and in each case (except for the HALT instruction, which needs no further refinement) clarifies the general instruction given by the opcode.
- For example (Figure 2.6), if the first hexadecimal digit of an instruction were 3 (the opcode for storing the contents of a register), the next hexadecimal digit of the instruction would indicate which register is to be stored, and the last two hexadecimal digits would indicate which memory cell is to receive the data.
- Thus the instruction 35A7 (hexadecimal) translates to the statement "STORE the bit pattern found in register 5 in the memory cell whose address is A7."
- (Note how the use of hexadecimal notation simplifies our discussion. In reality, the instruction 35A7 is the bit pattern 0011010110100111.)
- The instruction 35A7 also provides an explicit example of why main memory capacities are measured in powers of two.
- Because 8 bits in the instruction are reserved for specifying the memory cell utilized by this instruction, it is possible to reference exactly 2^8 different memory cells.
- It behooves us therefore to build main memory with this many cells—addressed from 0 to 255.
- If main memory had more cells, we would not be able to write instructions that distinguished between them; if main memory had fewer cells, we would be able to write instructions that referenced nonexisting cells.

Figure 2.6 Decoding the instruction 35A7



- As another example of how the operand field is used to clarify the general instruction given by an opcode, consider an instruction with the opcode 7 (hexadecimal), which requests that the contents of two registers be ORed.
- In this case, the next hexadecimal digit indicates the register in which the result should be placed, while the last two hexadecimal digits indicate which two registers are to be ORed.
- Thus the instruction 70C5 translates to the statement "OR the contents of register C with the contents of register 5 and leave the result in register 0."
- A subtle distinction exists between our machine's two LOAD instructions.
- Here we see that the opcode 1 (hexadecimal) identifies an instruction that loads a register with the contents of a memory cell, whereas the opcode 2 (hexadecimal) identifies an instruction that loads a register with a particular value.
- The difference is that the operand field in an instruction of the first type contains an address, whereas in the second type the operand field contains the actual bit pattern to be loaded.
- Note that the machine has two ADD instructions: one for adding two's complement representations and one for adding floating-point representations.
- This distinction is a consequence of the fact that adding bit patterns that represent values encoded in two's complement notation requires different activities within the arithmetic/logic unit from adding values encoded in floating-point notation.
- We close this section with Figure 2.7, which contains an encoded version of the instructions in Figure 2.2.

Figure 2.7 An encoded version of the instructions in Figure 2.2

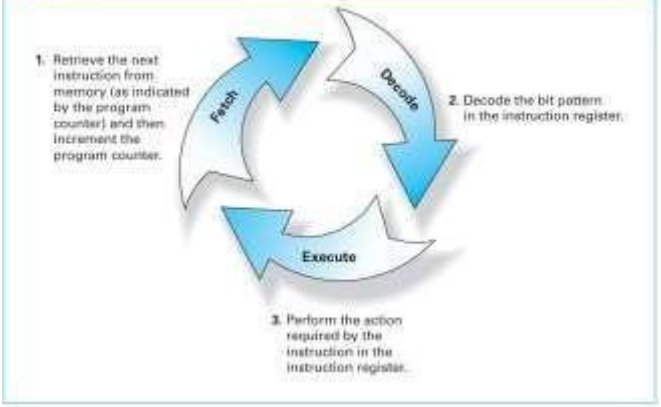
Encoded instructions	Translation
156C	Load register 5 with the bit pattern found in the memory cell at address 6C.
166D	Load register 6 with the bit pattern found in the memory cell at address 6D.
5056	Add the contents of register 5 and 6 as though they were two's complement representation and leave the result in register 0.
306E	Store the contents of register 0 in the memory cell at address 6E.
C000	Halt.

2.3 Program Execution

- A computer follows a program stored in its memory by copying the instruc tions from memory into the CPU as needed. Once in the CPU, each instruction is decoded and obeyed.
- The order in which the instructions are fetched from memory corresponds to the order in which the instructions are stored in memory unless otherwise altered by a JUMP instruction.
- It is necessary to consider two of the special purpose registers within the CPU: the instruction register and the program counter (Figure 2.4).
- The instruction register is used to hold the instruction being executed.
- The program counter contains the address of the next instruction to be executed, thereby serving as the machine's way of keeping track of where it is in the program.
- The CPU performs its job by continually repeating an algorithm that guides it through a three-step process known as the machine cycle.
- The steps in the machine cycle are fetch, decode, and execute (Figure 2.8).
- During the fetch step, the CPU requests that main memory provide it with the instruction that is stored at the address indicated by the program counter.
- Since each instruction in our machine is two bytes long, this fetch process involves retrieving the contents of two memory cells from main memory.

- The CPU places the instruction received from memory in its instruction register and then increments the program counter by two so that the counter contains the address of the next instruction stored in memory.
- With the instruction now in the instruction register, the CPU decodes the instruction, which involves breaking the operand field into its proper components based on the instruction's opcode.
- The CPU then executes the instruction by activating the appropriate circuitry to perform the requested task.
- Once the instruction in the instruction register has been executed, the CPU again begins the machine cycle with the fetch step.
- Observe that since the program counter was incremented at the end of the previous fetch, it again provides the CPU with the correct address.

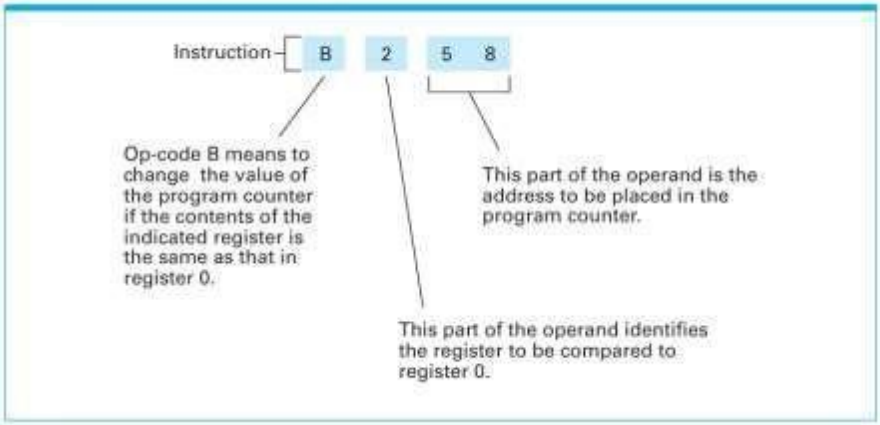
Figure 2.8 The machine cycle



- A somewhat special case is the execution of a JUMP instruction.
- Consider, for example, the instruction B258 (Figure 2.9), which means "JUMP to the instruction at address 58 (hexadecimal) if the contents of register 2 is the same as that of register 0."
- In this case, the execute step of the machine cycle begins with the comparison of registers 2 and 0.
- If they contain different bit patterns, the execute step terminates and the next machine cycle begins.
- If, however, the contents of these registers are equal, the machine places the value 58 (hexadecimal) in its program counter during the execute step.

- In this case, then, the next fetch step finds 58 in the program counter, so the instruction at that address will be the next instruction to be fetched and executed.
- Note that if the instruction had been B058, then the decision of whether the program counter should be changed would depend on whether the contents of register 0 was equal to that of register 0.
- But these are the same registers and thus must have equal content.
- In turn, any instruction of the form B0XY will cause a jump to be executed to the memory location XY regardless of the contents of register 0.

Figure 2.9 Decoding the instruction B258



An Example of Program Execution

- The machine cycle applied to the program presented in Figure 2.7, which retrieves two values from main memory, computes their sum, and stores that total in a main memory cell.
- We first need to put the program somewhere in memory.
- For our example, suppose the program is stored in consecutive addresses, starting at address A0 (hexadecimal).
- With the program stored in this manner, we can cause the machine to execute it by placing the address (A0) of the first instruction in the program counter and starting the machine (Figure 2.10).
- The CPU begins the fetch step of the machine cycle by extracting the instruction stored in main memory at location A0 and placing this instruction (156C) in its instruction register (Figure 2.11a).
- Notice that, in our machine, instructions are 16 bits (two bytes) long.

- Thus the entire instruction to be fetched occupies the memory cells at both address A0 and A1.
- The CPU is designed to take this into account so it retrieves the contents of both cells and places the bit patterns received in the instruction register, which is 16 bits long.
- The CPU then adds two to the program counter so that this register contains the address of the next instruction (Figure 2.11b).
- At the end of the fetch step of the first machine cycle, the program counter and instruction register contain the following data:

```
Program Counter: A2
Instruction Register: 156C
```

Figure 2.10 The program from Figure 2.7 stored in main memory ready for execution

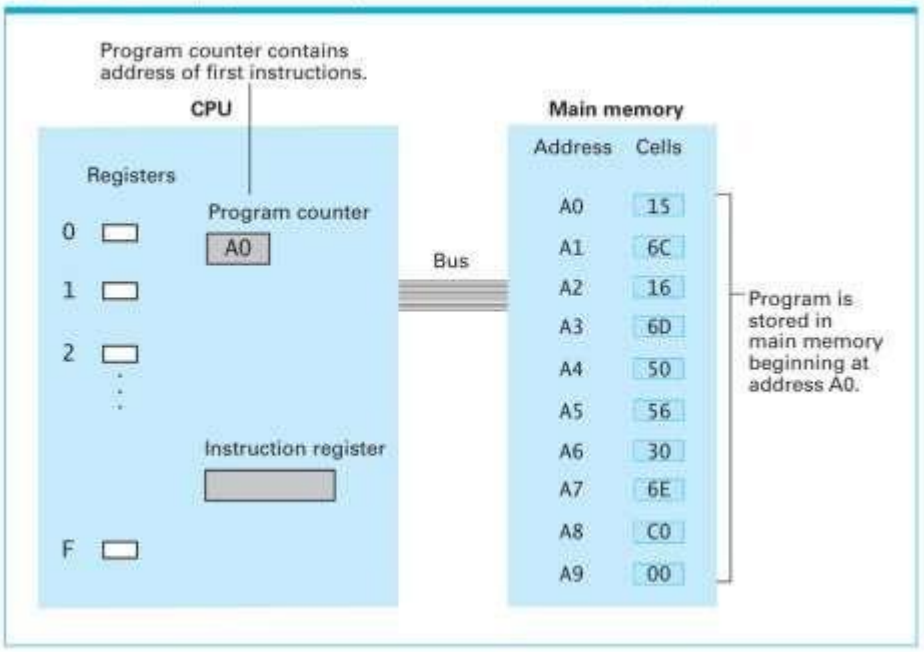
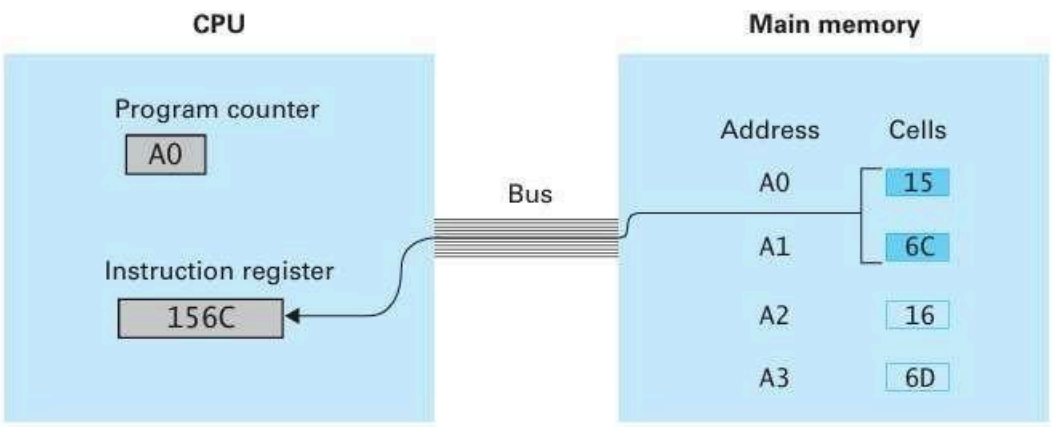
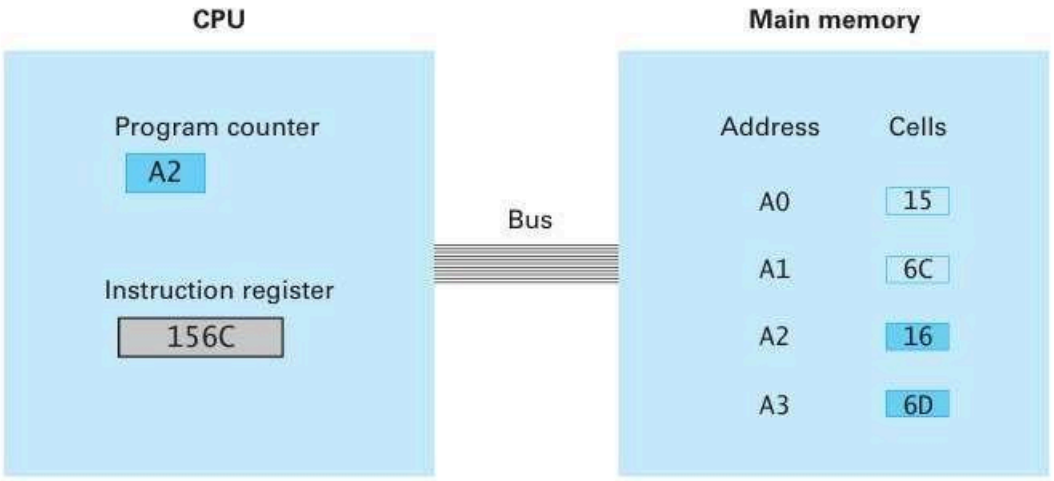


Figure 2.11 Performing the fetch step of the machine cycle



a. At the beginning of the fetch step the instruction starting at address A0 is retrieved from memory and placed in the instruction register.



b. Then the program counter is incremented so that it points to the next instruction.

- Next, the CPU analyzes the instruction in its instruction register and concludes that it is to load register 5 with the contents of the memory cell at address 6C.
- This load activity is performed during the execution step of the machine cycle, and the CPU then begins the next cycle.
- This cycle begins by fetching the instruction 166D from the two memory cells starting at address A2.
- The CPU places this instruction in the instruction register and increments the program counter to A4.

- The values in the program counter and instruction register therefore become the following:

```
Program Counter: A4
Instruction Register: 166D
```

- Now the CPU decodes the instruction 166D and determines that it is to load register 6 with the contents of memory address 6D.
- It then executes the instruction. It is at this time that register 6 is actually loaded.
- Since the program counter now contains A4, the CPU extracts the next instruction starting at this address. The result is that 5056 is placed in the instruction register, and the program counter is incremented to A6.
- The CPU now decodes the contents of its instruction register and executes it by activating the two's complement addition circuitry with inputs being registers 5 and 6.
- During this execution step, the arithmetic/logic unit performs the requested addition, leaves the result in register 0 (as requested by the control unit), and reports to the control unit that it has finished.
- The next instruction is fetched starting from memory location A8, and the program counter is incremented to AA.
- The contents of the instruction register (C000) are now decoded as the halt instruction. Consequently, the machine stops during the execute step of the machine cycle, and the program is completed.
- In summary, we see that the execution of a program stored in memory involves the same process you and I might use if we needed to follow a detailed list of instructions.
- Whereas we might keep our place by marking the instructions as we perform them, the CPU keeps its place by using the program counter.
- After determining which instruction to execute next, we would read the instruction and extract its meaning.
- Then, we would perform the task requested and return to the list for the next instruction in the same manner that the CPU executes the instruction in its instruction register and then continues with another fetch.

Programs Versus Data

- Many programs can be stored simultaneously in a computer's main memory, as long as they occupy different locations.
- Which program will be run when the machine is started can then be determined merely by setting the program counter appropriately.
- One must keep in mind, however, that because data are also contained in main memory and encoded in terms of 0s and 1s, the machine alone has no way of knowing what is data and what is program.
- If the program counter were assigned the address of data instead of the address of the desired program, the CPU, not knowing any better, would extract the data bit patterns as though they were instructions and execute them.
- The final result would depend on the data involved.
- We should not conclude, however, that providing programs and data with a common appearance in a machine's memory is bad.
- In fact, it has proved a useful attribute because it allows one program to manipulate other programs (or even itself) the same as it would data.
- Imagine, for example, a program that modifies itself in response to its interaction with its environment and thus exhibits the ability to learn, or perhaps a program that writes and executes other programs in order to solve problems presented to it.

2.4 Arithmetic/Logic Instructions

The arithmetic/logic group of instructions consists of instructions requesting arithmetic, logic, and shift operations.

Logic Operations

- The logic operations AND, OR, and XOR (exclusive or, often pro nounced, "exor") operations that combine two input bits to produce a single output bit.
- These operations can be extended to bitwise operations that combine two strings of bits to produce a single output string by applying the basic operation to individual columns.

- For example, the result of ANDing the patterns 10011010 and 11001001 results in

```
  10011010
AND 11001001
  10001000
```

- where we have merely written the result of ANDing the two bits in each column at the bottom of the column. Likewise, ORing and XORing these patterns would produce

```
  10011010
OR 11001001
  11011011
```

```
  10011010
XOR 11001001
  01010011
```

- One of the major uses of the AND operation is for placing 0s in one part of a bit pattern while not disturbing the other part.There are many applications for this in practice, such as filtering certain colors out of a digital image represented in the RGB format, as described in the previous chapter.
- Consider, for example, what happens if the byte 00001111 is the first operand of an AND operation.Without knowing the contents of the second operand, we still can conclude that the four most significant bits of the result will be 0s.Moreover, the four least significant bits of the result will be a copy of that part of the second operand, as shown in the following example:

```
  00001111
AND 10101010
  00001010
```

- This use of the AND operation is an example of the process called masking.
- Here one operand, called a mask, determines which part of the other operand will affect the result.
- In the case of the AND operation, masking produces a result that is a partial replica of one of the operands, with 0s occupying the nonduplicated positions.
- One trivial use of the AND operation in this context would be to mask off all of the bits associated with the red component of the pixels in an image, leaving only the blue and green components.
- AND operations are useful when manipulating other types of bit map besides images, whenever a string of bits is used in which each bit represents the presence or absence of a particular object.

- Suppose, then, that the eight bits in a memory cell are being used as a bit map, and we want to find out whether the object associated with the third bit from the high-order end is present. We merely need to AND the entire byte with the mask 00100000, which produces a byte of all 0s if and only if the third bit from the high-order end of the bit map is itself 0.
- A program can then act accordingly by following the AND operation with a conditional branch instruction.
- Moreover, if the third bit from the high-order end of the bit map is a 1, and we want to change it to a 0 without disturbing the other bits, we can AND the bit map with the mask 11011111 and then store the result in place of the original bit map.
- Where the AND operation can be used to duplicate a part of a bit string while placing 0s in the nonduplicated part, the OR operation can be used to duplicate a part of a string while putting 1s in the nonduplicated part. For this we again use a mask, but this time we indicate the bit positions to be duplicated with 0s and use 1s to indicate the nonduplicated positions. For example, ORing any byte with 11110000 produces a result with 1s in its most significant four bits while its remaining bits are a copy of the least significant four bits of the other operand, as demonstrated by the following example:

```
   11110000
OR 10101010
   11111010
```

- Consequently, whereas the mask 11011111 can be used with the AND operation to force a 0 in the third bit from the highorder end of a byte, the mask 00100000 can be used with the OR operation to force a 1 in that position.
- A major use of the XOR operation is in forming the complement of a bit string. XORing any byte with a mask of all 1s produces the complement of the byte. For example, note the relationship between the second operand and the result in the following example:

```
   11111111
XOR 10101010
   01010101
```

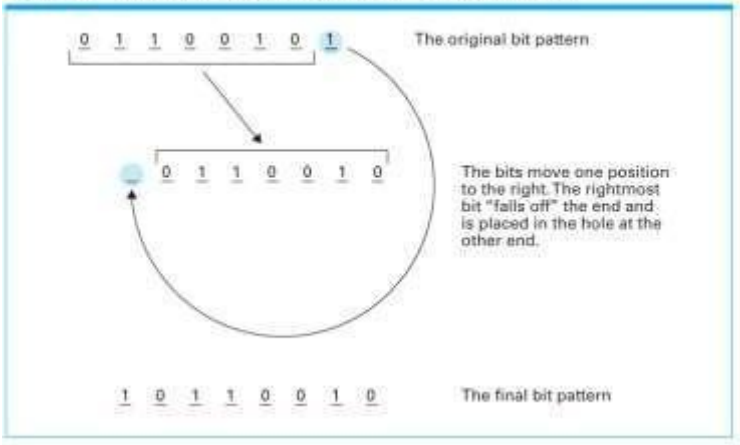
- The XOR operation can be used to invert all of the bits of an RGB bitmap image, resulting in an "inverted" color image in which light colors have been replaced by dark colors, and vice versa.
- In the machine language described in Appendix C, opcodes 7, 8, and 9 are used for the logic operations OR, AND, and XOR, respectively.
- Each requests that the corresponding logic operation be performed between the contents of two designated registers and that the result be placed in another designated register.
- For example, the instruction 7ABC requests that the result of ORing the contents of registers B and C be placed in register A.

Rotation and Shift Operations

- The operations in the class of rotation and shift operations provide a means for moving bits within a register and are often used in solving alignment problems.
- These operations are classified by the direction of motion (right or left) and whether the process is circular.
- Within these classification guidelines are numerous variations with mixed terminology.
- Consider a register containing a byte of bits.
- If we shift its contents one bit to the right, we imagine the rightmost bit falling off the edge and a hole appearing at the leftmost end.
- What happens with this extra bit and the hole is the distinguishing feature among the various shift operations.
- One technique is to place the bit that fell off the right end in the hole at the left end.
- The result is a circular shift, also called a rotation.
- Thus, if we perform a right circular shift on a byte-size bit pattern eight times, we obtain the same bit pattern we started with.
- Another technique is to discard the bit that falls off the edge and always fill the hole with a 0.
- The term logical shift is often used to refer to these operations.
- Such shifts to the left can be used for multiplying two's complement representations by two.

- Shifting binary digits to the left corresponds to multiplication by two, just as a similar shift of decimal digits corresponds to multiplication by ten.
- Division by two can be accomplished by shifting the binary string to the right. In either shift, care must be taken to preserve the sign bit when using certain notational systems.
- Among the variety of shift and rotate instructions possible, the machine language described in Appendix C contains only a right circular shift, designated by opcode A. In this case the first hexadecimal digit in the operand specifies the register to be rotated, and the rest of the operand specifies the number of bits to be rotated.
- Thus the instruction A501 means "Rotate the contents of register 5 to the right by 1 bit." In particular, if register 5 originally contained the bit pattern 65 (hexadecimal), then it would contain B2 after this instruction is executed (Figure 2.12).

Figure 2.12 Rotating the bit pattern 65 (hexadecimal) one bit to the right



Arithmetic Operations

- Although we have already mentioned the arithmetic operations of add, subtract, multiply, and divide, a few loose ends should still be connected.
- First, we have already seen that subtraction can be simulated by means of addition and negation.
- Moreover, multiplication is merely repeated addition and division is repeated subtraction. (Six divided by two is three because three twos can be subtracted from six.)
- For this reason, some small CPUs are designed with only the add or perhaps only the add and subtract instructions.
- We should also mention that numerous variations exist for each arithmetic operation.

- We have already alluded to this in relation to the add operations available on our machine in Appendix C.
- In the case of addition, for example, if the values to be added are stored in two's complement notation, the addition process must be performed as a straightforward column by column addition.
- However, if the operands are stored as floating-point values, the addition process must extract the mantissa of each, shift them right or left according to the exponent fields, check the sign bits, perform the addition, and translate the result into floating-point notation.
- Thus, although both operations are considered addition, the action of the machine is not the same.

2.5 Communicating with Other Devices

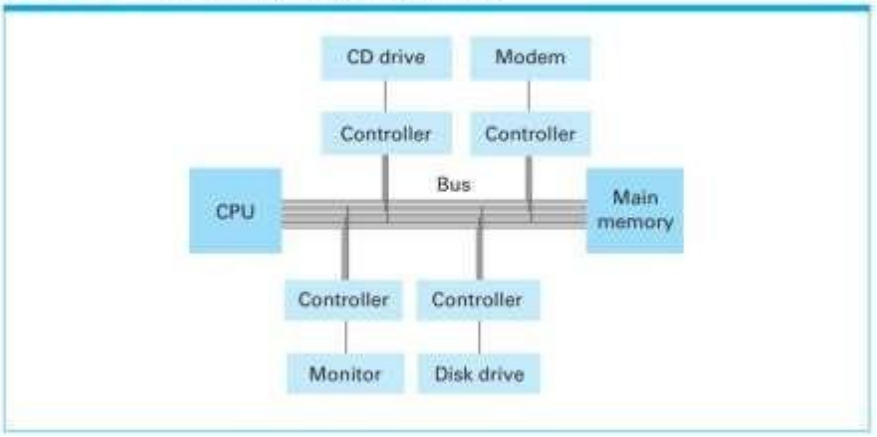
Main memory and the CPU form the core of a computer.

The Role of Controllers

- Communication between a computer and other devices is normally handled through an intermediary apparatus known as a controller.
- In the case of a personal computer, a controller may consist of circuitry permanently mounted on the computer's motherboard or, for flexibility, it may take the form of a circuit board that plugs into a slot on the motherboard.
- In either case, the controller connects via cables to peripheral devices within the computer case or perhaps to a connector, called a port, on the back of the computer where external devices can be attached.
- These controllers are sometimes small computers themselves, each with its own memory circuitry and simple CPU that performs a program directing the activities of the controller.
- A controller translates messages and data back and forth between forms compatible with the internal characteristics of the computer and those of the peripheral device to which it is attached.
- Originally, each controller was designed for a particular type of device; thus, purchasing a new peripheral device often required the purchase of a new controller as well.

- Recently, steps have been taken within the personal computer arena to develop standards, such as the universal serial bus (USB) and FireWire, by which a single controller is able to handle a variety of devices.
- For example, a single USB controller can be used as the interface between a computer and any collection of USB-compatible devices.
- The list of devices on the market today that can communicate with a USB controller includes mice, printers, scanners, mass storage devices, digital cameras, and smartphones.
- Each controller communicates with the computer itself by means of con nections to the same bus that connects the computer's CPU and main memory (Figure 2.13). From this position it is able to monitor the signals being sent between the CPU and main memory as well as to inject its own signals onto the bus.

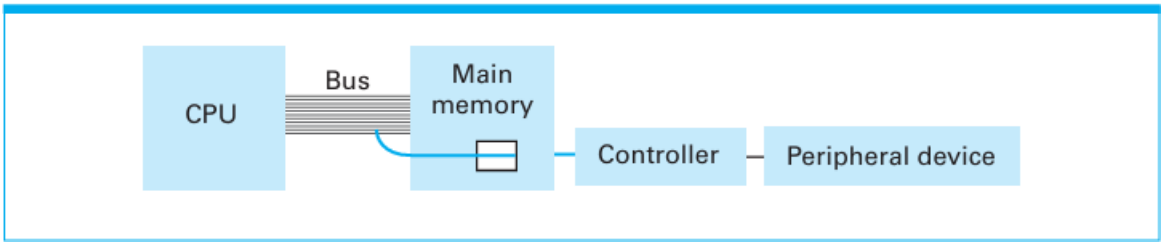
Figure 2.13 Controllers attached to a machine's bus



- With this arrangement, the CPU is able to communicate with the controllers attached to the bus in the same manner that it communicates with main memory.
- To send a bit pattern to a controller, the bit pattern is first constructed in one of the CPU's general-purpose registers.
- Then an instruction similar to a STORE instruction is executed by the CPU to "store" the bit pattern in the controller.
- Likewise, to receive a bit pattern from a controller, an instruction similar to a LOAD instruction is used.
- In some computer designs the transfer of data to and from controllers is directed by the same LOAD and STORE opcodes that are already provided for communication with main memory.

- In these cases, each controller is designed to respond to references to a unique set of addresses while main memory is designed to ignore references to these locations.
- Thus when the CPU sends a message on the bus to store a bit pattern at a memory location that is assigned to a controller, the bit pattern is actually "stored" in the controller rather than main memory.
- Likewise, if the CPU tries to read data from such a memory location, as in a LOAD instruction, it will receive a bit pattern from the controller rather than from memory.
- Such a communication system is called memory-mapped I/O because the computer's input/output devices appear to be in various memory locations (Figure 2.14).

Figure 2.14 A conceptual representation of memory-mapped I/O



- An alternative to memorymapped I/O is to provide special opcodes in the machine language to direct transfers to and from controllers. Instructions with these opcodes are called I/O instructions.

Direct Memory Access

- Since a controller is attached to a computer's bus, it can carry on its own communication with main memory during those nanoseconds in which the CPU is not using the bus.
- This ability of a controller to access main memory is known as direct memory access (DMA), and it is a significant asset to a computer's performance.
- For instance, to retrieve data from a sector of a disk, the CPU can send requests encoded as bit patterns to the controller attached to the disk asking the controller to read the sector and place the data in a specified area of main memory.
- The CPU can then continue with other tasks while the controller performs the read operation and deposits the data in main memory via DMA.
- Thus two activities will be performed at the same time. The CPU will be executing a program and the controller will be overseeing the transfer of data between the disk and main memory.

- In this manner, the computing resources of the CPU are not wasted during the relatively slow data transfer.
- The use of DMA also has the detrimental effect of complicating the communication taking place over a computer's bus.
- Bit patterns must move between the CPU and main memory, between the CPU and each controller, and between each controller and main memory.
- Coordination of all this activity on the bus is a major design issue.
- Even with excellent designs, the central bus can become an impediment as the CPU and the controllers compete for bus access.
- This impediment is known as the von Neumann bottleneck because it is a consequence of the underlying von Neumann architecture in which a CPU fetches its instructions from memory over a central bus.

Handshaking

- The transfer of data between two computer components is rarely a one-way affair.
- Even though we may think of a **printer** as a device that receives data, the truth is that a printer also **sends data back to the computer**.
- A computer can produce and send characters to a printer much faster than the printer can print them.
- If a computer blindly sent data to a printer, the printer would quickly fall behind, resulting in **lost data**.
- A process such as printing a document involves a constant **two-way dialogue, known as handshaking**, in which the computer and the peripheral device exchange information about the device's status and coordinate their activities.
- **Handshaking** often involves a **status word**, which is a bit pattern that is generated by the peripheral device and sent to the controller.
- The **status word** is a **bit map** in which the bits reflect the **conditions of the device**.
- For example, in the case of a printer:

1. The **least significant bit** of the status word may indicate whether the printer is **out of paper**.
2. The next bit may indicate whether the printer is **ready for additional data**.
3. Still another bit may indicate the presence of a **paper jam**.

- Depending on the system, the controller may **respond to this status information itself** or **make it available to the CPU**.
- In either case, the **status word provides the mechanism** by which communication with a peripheral device can be coordinated.

Popular Communication Media

- Communication between computing devices is handled over two types of paths: **parallel** and **serial**.
- These terms refer to the manner in which signals are transferred with respect to each other.
- In the case of **parallel communication**, several signals are transferred at the same time, each on a separate "line."
- Parallel communication is capable of transferring data rapidly but requires a relatively **complex communication path**.
- Examples of parallel communication include a computer's **internal bus**, where multiple wires are used to allow large blocks of data and other signals to be transferred simultaneously.
- In contrast, **serial communication** is based on transferring signals **one after the other over a single line**.
- Serial communication requires a **simpler data path** than parallel communication, which is the reason for its popularity.
- **USB** and **FireWire**, which offer relatively high-speed data transfer over short distances of only a few meters, are examples of serial communication systems.
- For slightly longer distances (within a home or office building), serial communication over **Ethernet connections**, either by wire or radio broadcast, are popular.
- For communication over greater distances, traditional **voice telephone lines** dominated the personal computer arena for many years; these are inherently serial systems.
- The transfer of digital data over these lines is accomplished by converting **bit patterns into audible tones** using a **modem (modulator-demodulator)**, transferring the tones serially, and converting them back into bits at the destination.

- For faster long-distance communication over traditional telephone lines, **DSL (Digital Subscriber Line)** is used, taking advantage of higher frequencies above the audible range for digital data while leaving lower frequencies for voice communication.
- Telephone companies are rapidly upgrading their systems to **fiber-optic lines**, which support digital communication more readily than traditional telephone lines.
- **Cable modems** are a competing technology that modulate and demodulate bit patterns over **cable television systems**, often using both fiber-optic lines and coaxial cables to provide TV signals and network access.
- **Satellite links** via high-frequency radio broadcast make computer network access possible even in **remote locations** far from high-speed telephone and cable television networks.

Communication Rates

- The rate at which bits are transferred from one computing component to another is measured in **bits per second (bps)**.
- Common units include **Kbps (kilobps, equal to one thousand bps)**, **Mbps (megabps, equal to one million bps)**, and **Gbps (gigabps, equal to one billion bps)**.
- Note the distinction between **bits and bytes**—8 Kbps is equal to 1 KB per second.
- In abbreviations, a lowercase **b** usually means bit whereas an uppercase **B** means byte.
- For short distance communication, **USB 2.0** and **FireWire** provide transfer rates of several hundred Mbps.
- These rates are sufficient for most **multimedia applications**.
- Their **convenience and relatively low cost** is why USB and FireWire are popular for communication between home computers and local peripherals such as printers, external disk drives, and cameras.
- By combining **multiplexing** and **data compression techniques**, traditional voice telephone systems were able to support transfer rates of 57.6 Kbps.
- This falls short of the needs of today's **multimedia and Internet applications**, such as high definition video streaming from sites like Netflix or YouTube.
- To play **MP3 music recordings** requires a transfer rate of about 64 Kbps.
- To play even **low quality video clips** requires transfer rates measured in units of Mbps.

- Alternatives such as **DSL, cable, and satellite links**, which provide transfer rates well into the Mbps range, have replaced traditional audio telephone systems.
- For example, **DSL offers transfer rates on the order of 54 Mbps**.
- The **maximum rate** available in a particular setting depends on the type of the communication path and the technology used in its implementation.
- This maximum rate is often loosely equated to the communication path's **bandwidth**, which also implies the capacity to carry large amounts of information simultaneously.