

1-Introduction.

Although it is difficult to distinguish between the ideas belonging to computer organization and those ideas belonging to computer architecture, it is impossible to say where hardware issues end and software issues begin. Computer scientists design algorithms that usually are implemented as programs written in some computer language, such as Java or C. But what makes the algorithm run? Another algorithm, of course! And another algorithm runs that algorithm, and so on until you get down to the machine level, which can be thought of as an algorithm implemented as an electronic device.

We begin our discussion of computer hardware by looking at the components necessary to build a computing system. **At the most basic level, a computer is a device consisting of three pieces:**

1. A **processor** to interpret and execute programs
2. A **memory** to store both data and programs
3. A **mechanism** for transferring data to and from the outside world

Once you understand computers in terms of their component parts, **you should be able to understand what a system is doing at all times and how you could change its behavior if so desired.** You might even feel like you have a few things in common with it. This idea is not as far-fetched as it appears. **Consider how a student sitting in class exhibits the three components of a computer:** the student's brain is the processor, the notes being taken represent the memory, and the pencil or pen used to take notes is the I/O mechanism. **But keep in mind that your abilities far surpass those of any computer in the world today, or any that can be built in the foreseeable future.**

2- The Computer Level Hierarchy.

If a machine is to be capable of solving a wide range of problems, it must be able to execute programs written in different languages, from FORTRAN and C to Lisp and

Prolog. The only physical components we have to work with are wires and gates. A formidable open space—*a semantic gap*—exists between these physical components and a high-level language such as C++. For a system to be practical, the semantic gap must be invisible to most of the users of the system.

Programming experience teaches us that when a problem is large, we should break it down and use a "divide and conquer" approach. In programming, we divide a problem into modules and then design each module separately. Each module performs a specific task and modules need only know how to interface with other modules to make use of them.

Computer system organization can be approached in a similar manner. Through the principle of abstraction, we can imagine the machine to be built from a hierarchy of levels, in which each level has a specific function and exists as a distinct hypothetical machine. We call the hypothetical (افتراضی) computer at each level a *virtual machine*. Each level's virtual machine executes its own particular set of instructions, calling upon machines at lower levels to carry out the tasks when necessary. By studying computer organization, you will see the rationale behind the hierarchy's partitioning, as well as how these layers are implemented and interface with each other. [Figure 1](#) shows the commonly accepted layers representing the abstract virtual machines.

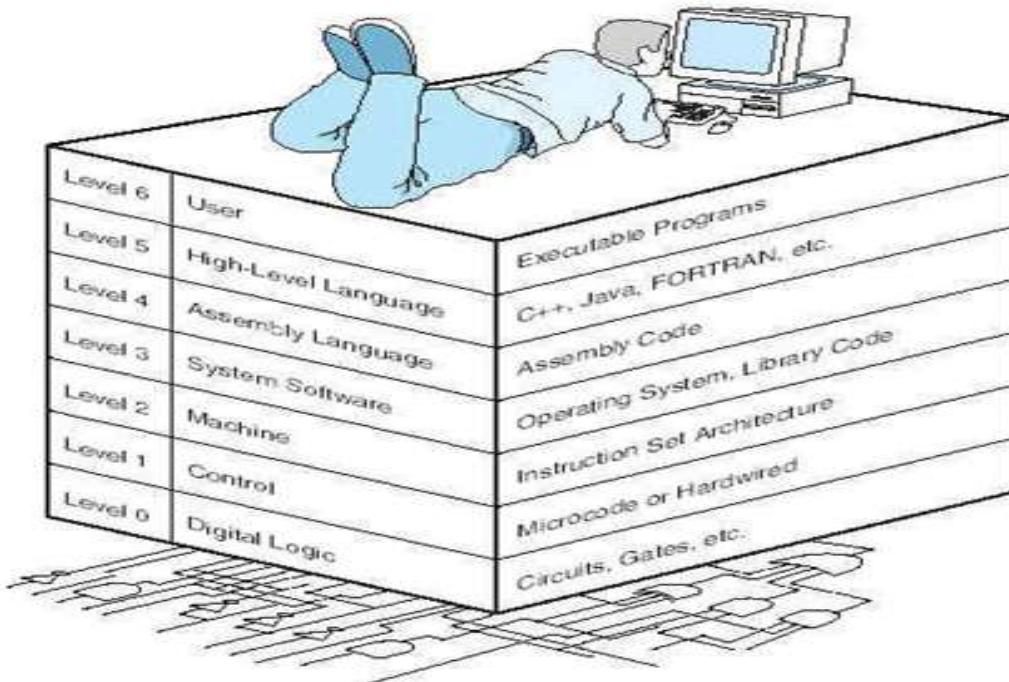


Figure 1: The Abstract Levels of Modern Computing Systems

Level 6, the User Level, is composed of applications and is the level with which everyone is most familiar. At this level, we **run programs such as word processors, graphics packages, or games**. The lower levels are nearly invisible from the User Level.

Level 5, the High-Level Language Level, consists of **languages** such as **C, C++, FORTRAN, Lisp, Pascal, and Prolog**. These **languages must be translated** (using either a compiler or an interpreter) to a language the machine can understand. Compiled languages are translated into assembly language and then assembled into machine code. (They are translated to the next lower level.) The user at this level sees very little of the lower levels. Even though a programmer must know about data types and the instructions available for those types, she need not know about how those types are actually implemented.

Level 4, the Assembly Language Level, encompasses some type **of assembly language**. As previously mentioned, compiled higher-level languages are first translated to assembly, which is then directly translated to machine language. This is a **one-to-one translation**, meaning that one assembly language instruction is translated to exactly one

machine language instruction. By having separate levels, we reduce the semantic gap between a high-level language, such as C++, and the actual machine language (which consists of 0s and 1s).

Level 3, the System Software Level, deals with operating system instructions. This level is responsible for multiprogramming, protecting memory, synchronizing processes, and various other important functions. Often, instructions translated from assembly language to machine language are passed through this level unmodified.

Level 2, the Instruction Set Architecture (ISA), or Machine Level, consists of the machine language recognized by the particular architecture of the computer system. Programs written in a computer's true machine language on a hardwired computer (see below) can be executed directly by the electronic circuits without any interpreters, translators, or compilers.

Level 1, the Control Level, is where a *control unit* makes sure that instructions are decoded and executed properly and that data is moved where and when it should be. The control unit interprets the machine instructions passed to it, one at a time, from the level above, causing the required actions to take place.

Control units can be designed in one of two ways: They can be *hardwired* or they can be *microprogrammed*. In *hardwired* control units, control signals emanate from blocks of digital logic components. These signals direct all of the data and instruction traffic to appropriate parts of the system. Hardwired control units are typically very fast because they are actually physical components. However, once implemented, they are very difficult to modify for the same reason.

The other option for control is to implement instructions using a *microprogram*. A microprogram is a program written in a low-level language that is implemented directly by the hardware. Machine instructions produced in Level 2 are fed into this microprogram, which then interprets the instructions by activating hardware suited to

execute the original instruction. One machine-level instruction is often translated into several microcode instructions. This is not the one-to-one correlation that exists between assembly language and machine language. *Microprograms are popular because they can be modified relatively easily.* The **disadvantage** of microprogramming is, of course, that the additional layer of translation typically results in slower instruction execution.

Level 0, the Digital Logic Level, is where we find the physical components of the computer system: the **gates and wires**. These are the fundamental building blocks, the implementations of the mathematical logic, that are common to all computer systems.

Memory Addressing

The memory of a computer system consists of tiny electronic switches, with each switch in one of two states: *open* or *closed*. It is, however, more convenient to think of these states as **0** and **1**, rather than open and closed. Thus, each switch can represent a bit. The memory unit consists of millions of such bits. In order to make memory more manageable, eight bits are grouped into a byte. Memory can then be viewed as consisting of an ordered sequence of bytes. Each byte in this memory is identified by its sequence number starting with 0, as shown in Figure 1. This is referred to as the *memory address* of the byte. Such memory is called *byte addressable* memory because each byte has a unique address.

Address (in decimal)	Address (in hex)
$2^{32}-1$	FFFFFFF
	FFFFFFE
	FFFFFFD
•	
•	
•	
2	00000002
1	00000001
0	00000000

Figure 1 Logical view of the system memory.

The Pentium can address up to 4 GB (2^{32} bytes) of main memory (see Figure 1). This magic number comes from the fact that the address bus of the Pentium has 32 address lines. This number is referred to as the *memory address space*. The memory address space of a system is determined by the address bus width of the processor used in the system. The actual memory in a system, however, is always less than or equal to the memory address space. The amount of memory in a system is determined by how much of this memory address space is *populated* with memory chips.

Memory chips:

It is possible to visualize a typical internal main memory structure as consisting of rows and columns of basic cells. Each cell is capable of storing one bit of information. Figure 2 provides a conceptual internal organization of a memory chip. In this figure, cells belonging to a given row can be assumed to form the bits of a given memory word. Address lines $A_0A_1 \dots A_{n-2}A_{n-1}$ are used as inputs to the address decoder in order to generate the word select lines $W_0 W_1 \dots W_{2^n-1}$. A given word select line is common to all memory cells in the same row. At any given time, the address decoder activates only one word select line while deactivating the remaining lines. A word select line is used to enable all cells in a row for read or write. Data (bit) lines are used to input or output the contents of cells. Each memory cell is connected to two data lines. A given data line is common to all cells in a given column.

The address lines $A_0 \dots A_{n-1}$ in the memory chip shown in Figure 2 contain an address, which is decoded from an ***n-bit*** address into one of ***2ⁿ locations*** (*The number of locations may be obtained from the address width of the chip*) within the chip, each of which has a ***w-bit*** word associated with it. The storage capacity of a memory chip is the product of the number of locations and the word width. The chip thus contains ***2ⁿ × w bits***.

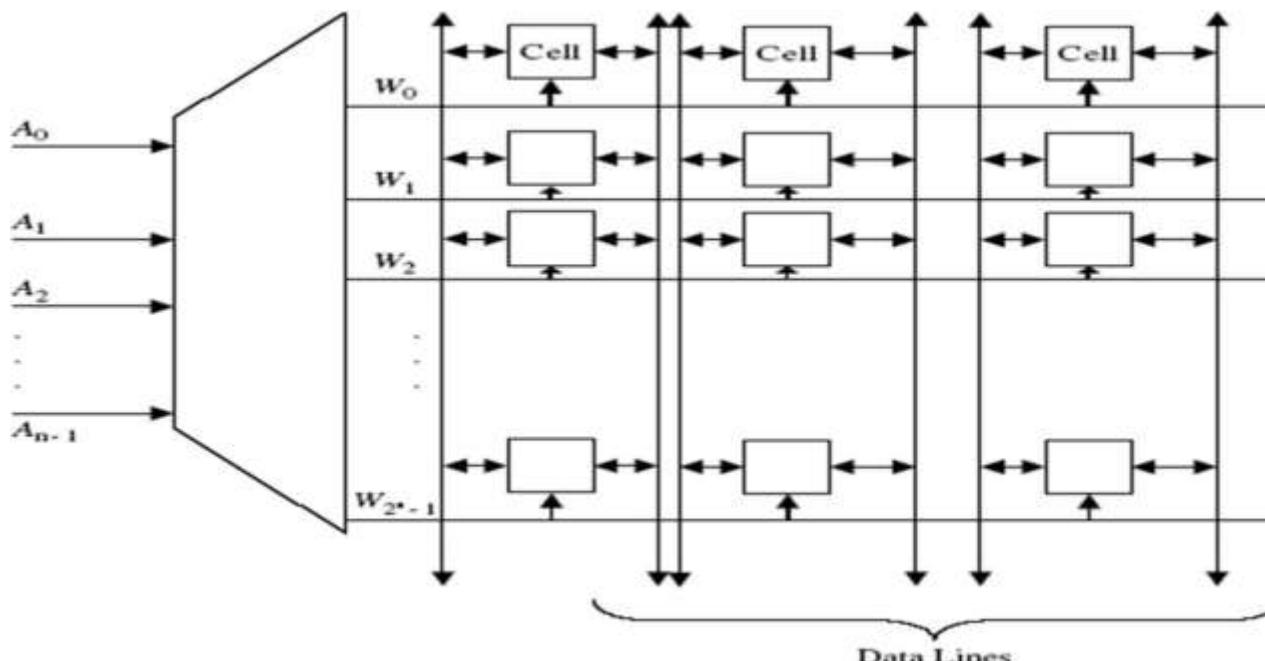


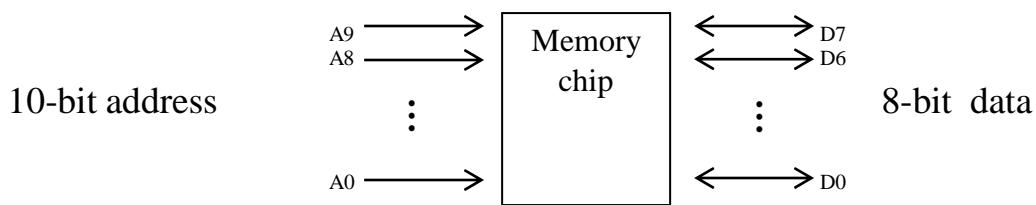
Figure 2: A conceptual internal organization of a memory chip

The computer's word size can be the size of the data bus which carries data between the CPU and memory and the CPU and I/O devices.

To access the memory, to store or retrieve a single word of information, it is necessary to have a unique address. The word address is the number that identifies the location of a word in a memory.

For example, a chip width 10 address lines has $2^{10}=1024$ locations. Given an 8 bit data width, a 10-bit address chip has a memory size of :

$$2^{10} * 8 = 1024 * 8 = 1K * 1\text{byte} = \text{Kbyte}$$



The second properties of memory chips is access time, access time is the speed with a location within the memory chip may be made available to the data bus. It is defined as the time interval between the instant that an address is sent to the memory chip and the instant that the data stored in the locations appears on the data bus. Access time is given in nanosecond's (ns) and varies from 25 ns to the relatively slow 200 ns.

In general a computer with a larger word size can execute programs of instructions at a faster rate because more data and more instructions are stuffed into word. The larger word sizes, however, mean more lines making up the data bus, and therefore more interconnections between the CPU and memory and I/O devices.

A single chip is usually insufficient to provide the memory requirements of a computer. A number of chips are therefore connected in parallel to form what is known as a ***memory bank***. Figure below shows a memory bank consisting of eight 1-bit chips. Each has 18 address lines (A₀ ... A₁₇). The total storage capacity of one chip = $2^{18} * 1 = 256 \text{ K bits}$. Total size of the memory bank = $256 \text{ K bits} * 8 = 256 \text{ KB}$

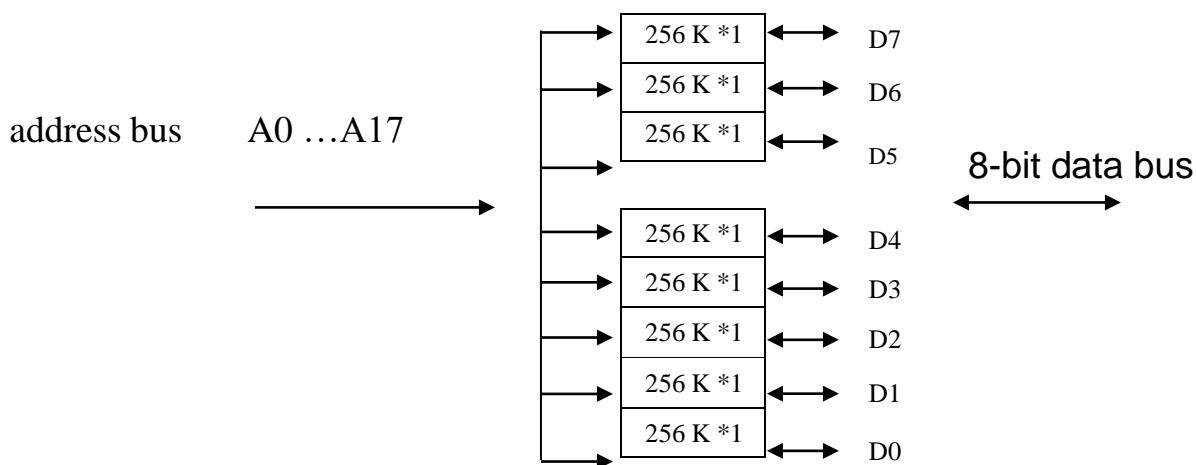


Figure 3:Memory bank

[

Ex: A certain memory chip is specified as 2Kx8

1- How many words can be stored on this chip?

2- What is the word size :

3- How many total bits can this chip store?

Solution:

1- $2K = 2 * 1024 = 2048$ words

2- The word size is 8-bits(1-byte)

3- Capacity = $2048 * 8 = 16\text{-}383\text{bits} = 16\text{Kbit}$

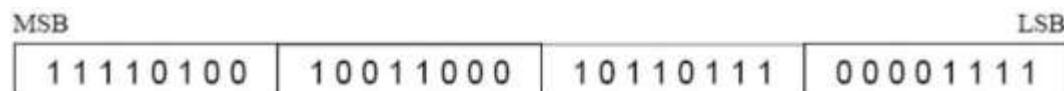
Byte Ordering

Storing data often requires more than a byte. For example, we need four bytes of memory to store an integer variable that can take a value between 0 and $2^{32}-1$. Let us assume that the value to be stored is the one in Figure 16a.

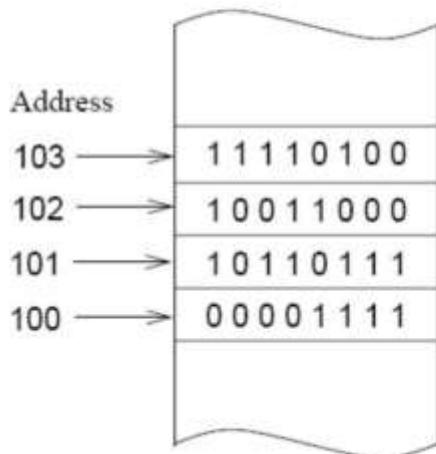
Suppose that we want to store these 4-byte data in memory at locations 100 through 103. How do we store them? Figure 16 shows two possibilities: least significant byte (Figure 16b) or most significant byte (Figure 16c) is stored at location 100. These two byte ordering schemes are referred to as the *little endian* and *big endian*. In either case, we always refer to such multi byte data by specifying the lowest memory address (100 in this example).

Is one byte ordering scheme better than the other? Not really! It is largely a matter of choice for the designers. For example, Pentium processors use the little-endian byte ordering. However, most processors leave it up to the system designer to configure the processor. For example, the MIPS and PowerPC processors use the big-endian byte ordering by default, but these processors can be configured to use the little-endian scheme.

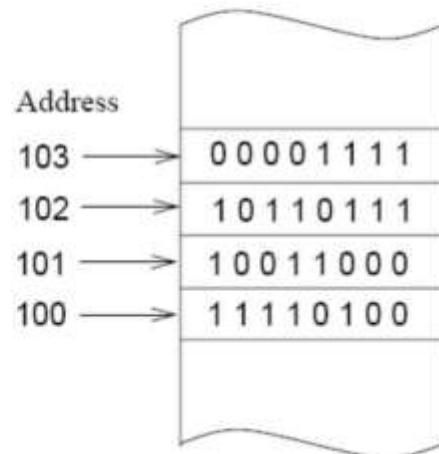
The particular byte ordering scheme used does not pose any problems as long as you are working with machines that use the same byte ordering scheme. However, difficulties arise when you want to transfer data between two machines that use different schemes. In this case, conversion from one scheme to the other is required. For example, the Pentium provides two instructions to facilitate such conversion: one to perform 16-bit data conversions and the other for 32-bit data.



(a) 32-bit data



(b) Little-endian byte ordering



(c) Big-endian byte ordering

Figure 10 Two byte ordering schemes commonly used by computer systems.

Basic Memory Operations

The memory unit supports two basic operations: *read* and *write*. The *read operation* reads previously stored data and the *write operation* stores a new value in memory. Both of these operations require a memory address. In addition, the write operation requires specification of the data to be written. The address and data of the memory unit are connected to the address and data buses of the system bus, respectively. The read and write signals come from the control bus. Two metrics are used to characterize memory. **Access time** refers to the amount of time required by the memory to retrieve the data at the addressed location. The other metric is the **memory cycle time**, which refers to the minimum time between successive memory operations.

The read operation is nondestructive in the sense that one can read a location of the memory as many times as one wishes without destroying the contents of that location. The write operation, however, is destructive, as writing a value into a location destroys the old contents of that memory location.

Steps in a Typical Read Cycle:

1. Place the address of the location to be read on the address bus,
2. Activate the memory read control signal on the control bus,
3. Wait for the memory to retrieve the data from the addressed memory location and place them on the data bus,
4. Read the data from the data bus,
5. Drop the memory read control signal to terminate the read cycle.

Steps in a Typical Write Cycle:

1. Place the address of the location to be written on the address bus,
2. Place the data to be written on the data bus,
3. Activate the memory write control signal on the control bus,
4. Wait for the memory to store the data at the addressed location,
5. Drop the memory write signal to terminate the write cycle.

Cache Memory

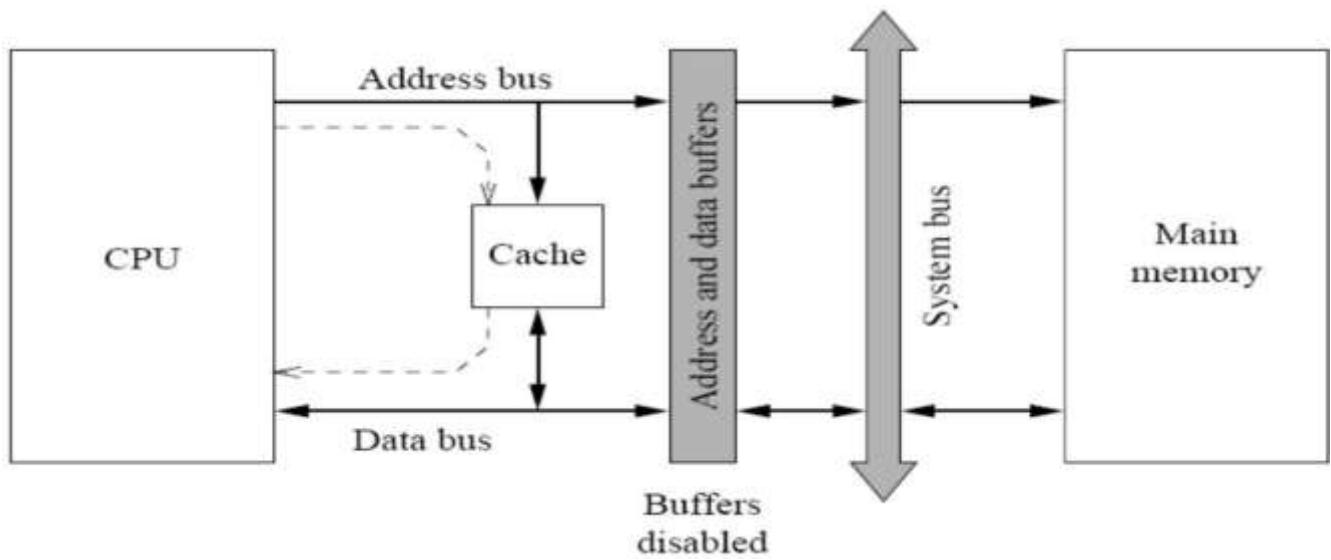
Cache memory is much smaller than the main memory. Usually implemented using SRAMs, which sits between the processor and main memory in the memory hierarchy. The cache effectively isolates the processor from the slowness of the main memory, which is DRAM-based. The principle behind the cache memories is to prefetch the data from the main memory before the processor needs them. If we are successful in predicting the data that the processor needs in the near future, we can

preload the cache and supply those data from the faster cache. It turns out that predicting the processor future access requirements is not difficult owing to a phenomenon known as *locality of reference* that programs exhibit.

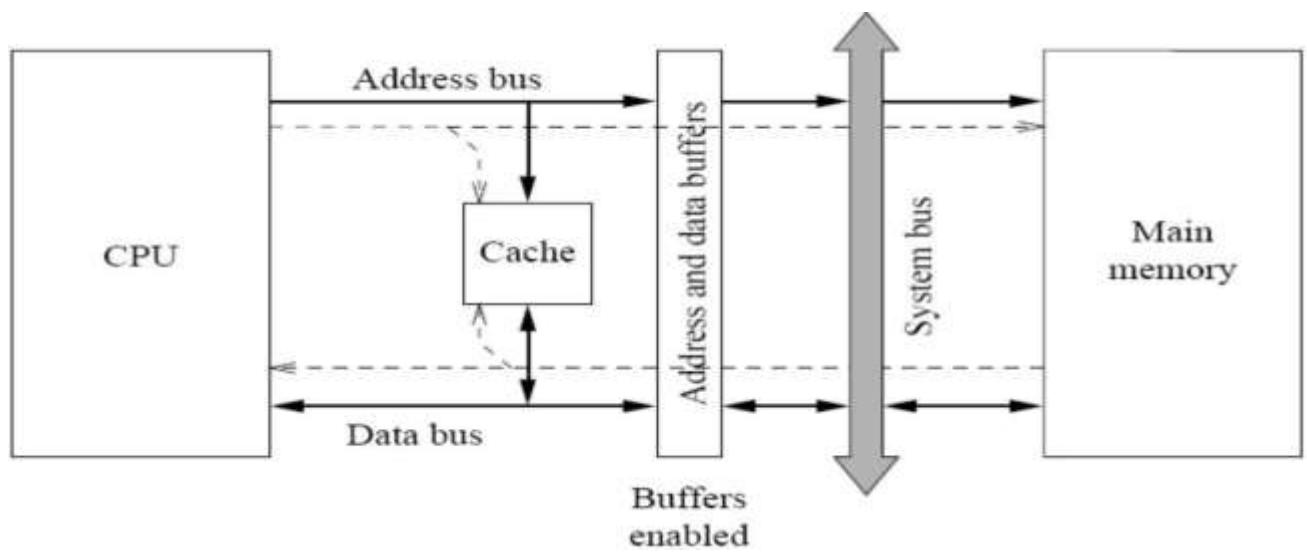
Performance of cache systems can be measured using the *hit rate* or *hit ratio*. When the processor makes a request for a memory reference, the request is first sought in the cache. If the request corresponds to an element that is currently residing in the cache, we call that a **cache hit**. On the other hand, if the request corresponds to an element that is not currently in the cache, we call that a **cache miss**. A cache hit ratio, hc , is defined as the probability of finding the requested element in the cache. A cache miss ratio ($1 - hc$) is defined as the probability of not finding the requested element in the cache. The likelihood of the processor finding what it wants in cache is as high as 95 percent.

Figure (1) shows how memory read operations are handled to include the cache memory. When the data needed by the processor are in the cache memory (“read hit”), the requested data are supplied by the cache (see Figure 1a).

The dashed line indicates the flow of information in the case of a read hit. In the case of a read miss, we have to read the data from the main memory. While supplying the data from the main memory, a copy is also placed in the cache as indicated by the bottom dashed lines in Figure 1b. Reading data on a cache miss takes more time than reading directly from the main memory as we have to first test to see if the data are in the cache and also consider the overhead involved in copying the data into the cache. This additional time is referred to as the *miss penalty*.



(a) Read hit



(a) Read miss

In the case that the requested element is not found in the cache, then it has to be brought from a subsequent memory level in the memory hierarchy. Assuming that the element exists in the next memory level, that is, the main memory, then it has to be brought and placed in the cache. In expectation that the next requested element will be residing in the neighboring locality of the current requested element (spatial locality), then upon a cache miss what is actually brought to the main memory is a block of elements that contains the requested element (a **block** as a minimum unit of data transfer). The advantage of transferring a block from the main memory to the cache will be most visible if it could be possible to transfer such a block using one main memory access time. Such a possibility could be achieved by increasing the rate at which information can be transferred between the main memory and the cache. The transferring of data is referred to as a mapping process.

Modern memory systems may have several levels of cache, referred to as Level 1 (L1), Level 2 (L2), and even, in some cases, Level 3 (L3). In most instances the L1 cache is implemented right on the CPU chip. Both the Intel Pentium and the IBM-Motorola PowerPC G3 processors have 32 Kbytes of L1 cache on the CPU chip. Processors will often have two separate L1 caches, one for instructions (**I-cache**) and one for data (**D-cache**). A level 2 (L2) high speed memory cache store up to 512 kilobytes of data.

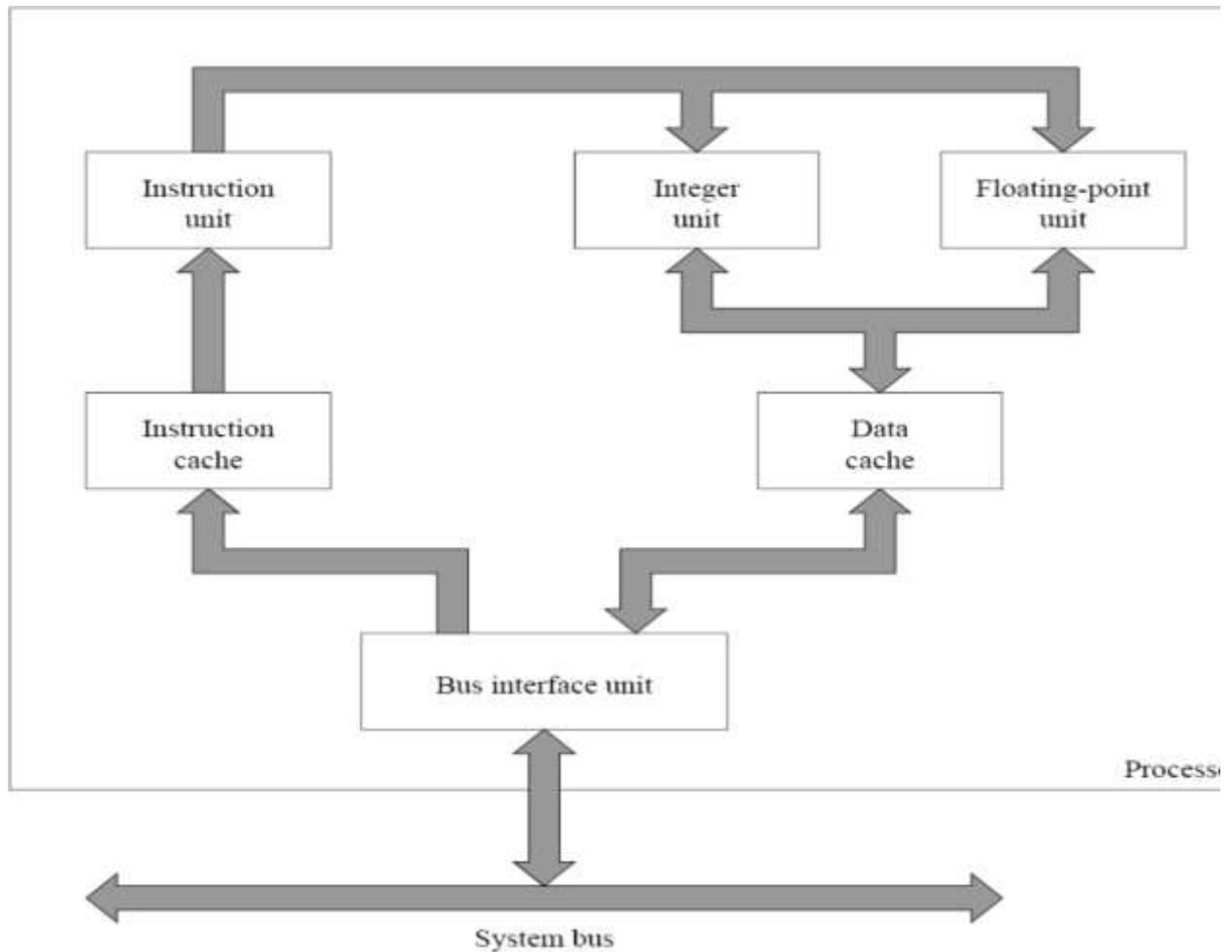


Figure 18 Most current processors use separate caches for instructions and data with separate instruction and data buses.

System bus.

A computer system consists of three major components: a processor, a memory unit, and an input/output (I/O) subsystem. Interconnection network facilitates communication among these three components, as shown in Figure 21. The interconnection network is called the *system bus*.

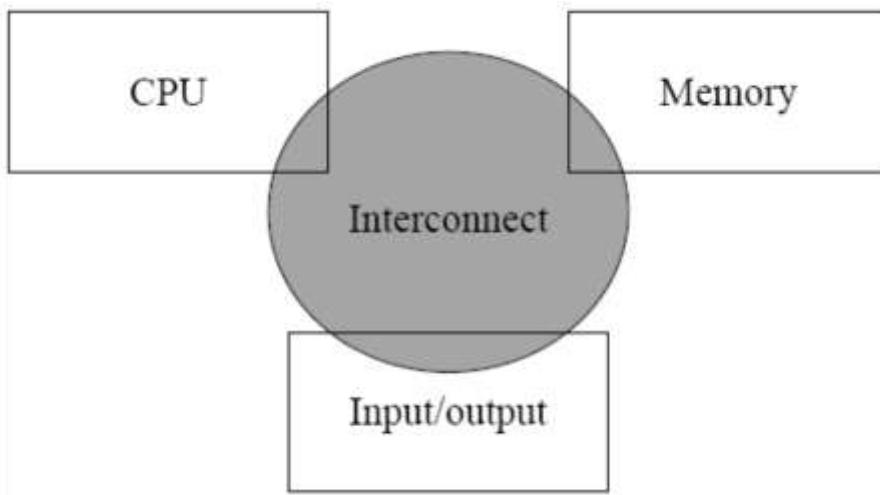


Figure 21 The three main components of a computer system are interconnected by a bus.

The term “bus” is used to represent a group of electrical signals or the wires that carry these signals. As shown in Figure 1.5, the system bus consists of three major components: an address bus, a data bus, and a control bus.

The address bus width determines the amount of physical memory addressable by the processor. The data bus width indicates the size of the data transferred between the processor and memory or an I/O device. For example, the Pentium processor has 32 address lines and 64 data lines. Thus, the Pentium can address up to 2^{32} , or 4 GB of memory. Furthermore, each data transfer can move 64 bits of data.

The control bus consists of a set of control signals. Typical control signals include memory read, memory write, I/O read, I/O write, interrupt, interrupt acknowledge, bus request, and bus grant. These control signals indicate the type of action taking place on the system bus. For example, when the processor is writing data

into the memory, the memory write signal is generated. Similarly, when the processor is reading from an I/O device, it generates the I/O read signal.

A bus connects various components in a computer system. Thus buses can be categorized into:

Internal buses: The processor uses several internal buses to interconnect registers and the ALU.

External buses: are used to interface with the devices outside a typical processor system. By our classification, serial and parallel interfaces, Universal Serial Bus (USB). These buses are typically used to connect I/O devices.

Figure 22 shows *dedicated buses* connecting the major components of a computer system. The system bus consists of address, data, and control buses. One problem with the dedicated bus design is that it requires a large number of wires. We can reduce this count by using *multiplexed buses*. For example, a single bus may be used for both address and data. In addition, the address and data bus widths play an important role in determining the address space and data transfer rate, respectively.

- **Bus Width:** Bus width refers to the *data* and *address* bus widths. System performance improves with a wider data bus as we can move more bytes in parallel. We increase the addressing capacity of the system by adding more address lines.
- **Bus Type:** As discussed in the last section, there are two basic types of buses: *dedicated* and *multiplexed*.
- **Bus Operations:** Bus systems support several types of operations to transfer data. These include the *read*, *write*, *block transfer*, *read-modify-write*, and *interrupt*.

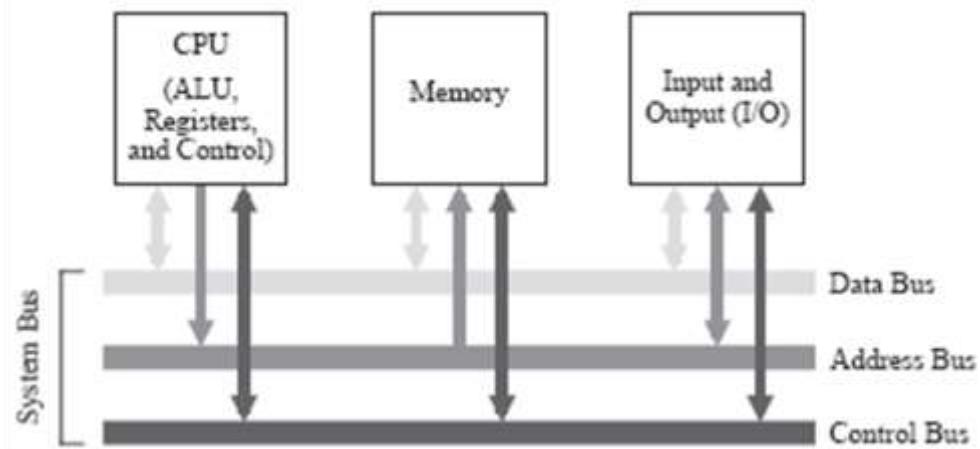


Figure 22: The system bus model of a computer system.

Consider, for example, the bus connecting a CPU and memory in a given system. The size of the memory in that system is **512M** word and each word is 32 bits. In such system, the size of the address bus should be $\log_2(512 \times 2^{20}) = 29$ lines.

23- Pipelining.

In each execution cycle the control unit would have to wait until the instruction is fetched from memory. Furthermore, the ALU would have to wait until the required operands are fetched from memory. Processor speeds are increasing at a much faster rate than the improvements in memory speeds. Thus, we would be wasting the control unit and ALU resources by keeping them idle while the system fetches instructions and data. How can we avoid this situation? Let's suppose that we can prefetch the instruction. That is, we read the instruction before the control unit needs it. These prefetched instructions are typically placed in a set of registers called the *prefetch buffers*. Then, the control unit doesn't have to wait. How do we do this prefetch? Given that the program execution is sequential, we can prefetch

the next instruction in sequence while the control unit is busy decoding the current instruction. Pipelining generalizes this concept of overlapped execution. Similarly, prefetching the required operands avoids the idle time experienced by the ALU.

Figure 23 shows how pipelining helps us improve the efficiency. The instruction execution can be divided into five parts. In pipelining terminology, each part is called a stage. For simplicity, let's assume that execution of each stage takes the same time (say, one cycle). As shown in Figure 23, each stage spends one cycle in executing its part of the execution cycle and passes the instruction on to the next stage. Let's trace the execution of this pipeline during the first few cycles. During the first cycle, the first stage S1 fetches the instruction. All other stages are idle. During Cycle 2, S1 passes the first instruction I1 to stage S2 for decoding and S1 initiates the next instruction fetch. Thus, during Cycle 2, two of the five stages are busy: S2 decodes I1 while S1 is busy with fetching I2. During Cycle 3, stage S2 passes instruction I1 to stage S3 to fetch any required operands. At the same time, S2 receives I2 from S1 for decoding while S1 fetches the third instruction. This process is repeated in each stage. As you can see, after four cycles, all five stages are busy. This state is called the pipeline full condition. From this point on, all five stages are busy.

Stage	Time (cycles) →									
	1	2	3	4	5	6	7	8	9	10
S1: IF	I1	I2	I3	I4	I5	I6	...			
S2: ID		I1	I2	I3	I4	I5	I6	...		
S3: OF			I1	I2	I3	I4	I5	I6	...	
S4: IE				I1	I2	I3	I4	I5	I6	..
S5: WB					I1	I2	I3	I4	I5	I6

Figure 23 A pipelined execution of the basic execution cycle

This figure clearly shows that the execution of instruction I1 is completed in Cycle 5. Thus, executing six instructions takes only 10 cycles. Without pipelining, it would have taken 30 cycles.

Notice from this description that pipelining does not speed up execution of individual instructions; each instruction still takes five cycles to execute. However, pipelining increases the number of instructions executed per unit time; that is, instruction throughput increases.

Pipelining a computer architecture designed so that all parts of circuit are always working, so that no part of the circuit is stalled waiting from another part. Pipelining allows overlapped execution to improve throughput.

The Von Neumann Model.

John von Neumann, along with others, proposed the concept of the stored program that we use even today. The idea was to keep a program in the memory and read the instructions from it. He also proposed an architecture that clearly identified the components we have presented previously: ALU, control, input, output, and memory as illustrated in Figure 1. This architecture is known as the *von Neumann architecture*.

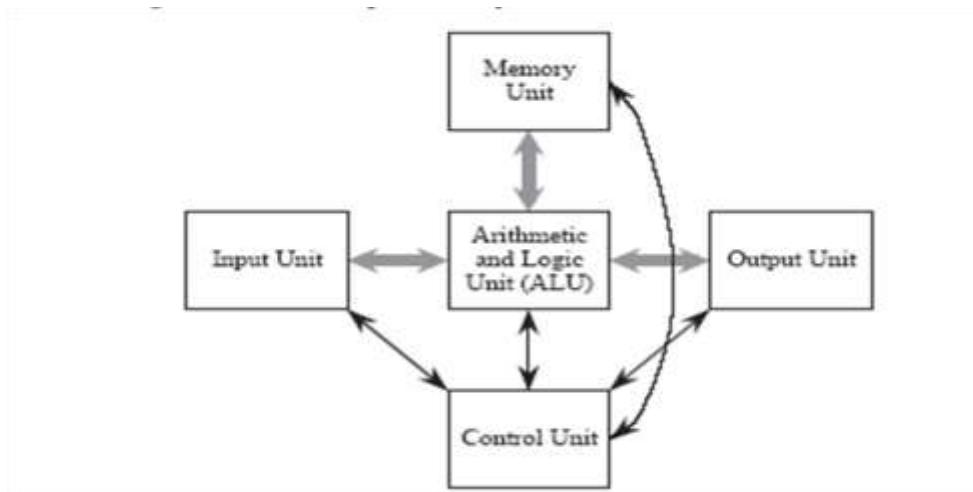


Figure 1: The von Neumann model of a digital computer. Thick arrows represent data paths. Thin arrows represent control paths.

This architecture uses what is known as the **stored program model**. In the von Neumann architecture, the **stored program** is the most important aspect of the von Neumann model. The key features of this architecture are as follows:

- There is no distinction between instructions and data. This requirement has several main implications:
 1. Instructions are represented as numbers, just like the data themselves. This uniform treatment of instructions and data simplifies the design of memory and software.
 2. Instructions and data are not stored in separate memories; a single memory is used for both. Thus, a single path from the memory can carry both data and instructions.

3. The memory is addressed by location, regardless of the type of data at that location.

- By default, instructions are executed in the sequential manner in which they are present in the stored program.

A program is stored in the computer's memory along with the data to be processed (*A computer with a von Neumann architecture has a single memory space that contains both the instructions and the data, see figure 2*). This can lead to a condition called the ***von Neumann bottleneck***, it places a limitation on how fast the processor can run. Instructions and data must share the same path to the CPU from memory, so if the CPU is writing a data value out to memory, it cannot fetch the next instruction to be executed. It must wait until the data has been written before proceeding and vice versa.

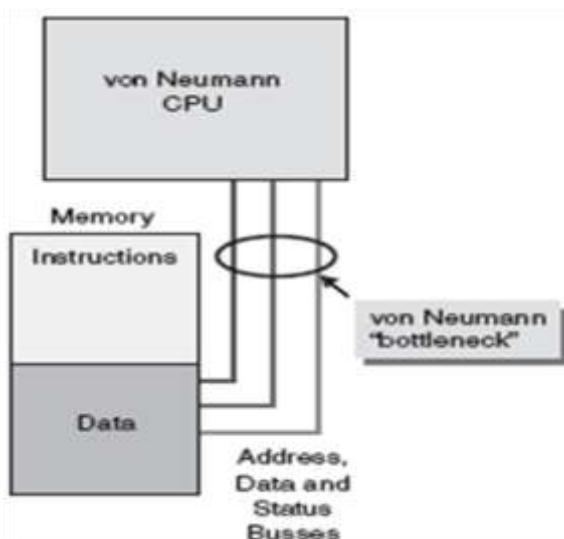


Figure 2: Memory architecture for the von Neumann

In contrast to the single memory concept used in the von Neumann architecture, the *Harvard architecture* uses separate memories for instructions and data. The term now refers to machines that have a single main memory but use separate caches for instructions and data.

College of Information Technology

Most processors now use two separate caches: one for instructions and the other for data. This design uses separate buses for instructions and data. Processors typically use the Harvard architecture only at the CPU-cache interface.

In order to avoid the von Neumann bottleneck :-

- multi-level caches used to reduce miss penalty (assuming that the L1 cache is on-chip); and
- memory system are designed to support caches with burst mode accesses.

Programmed I/O (PIO).

I/O ports provide the basic access to I/O devices via the associated I/O controller. We still will have to devise ways to transfer data between the system and I/O devices using the I/O ports. A simple way of transferring data is to ask the processor to do the transfer. In this scheme of things, the processor is responsible for transferring data word by word. Typically, it executes a loop until the data transfer is complete. This technique is called *programmed I/O* (PIO).

The CPU sends commands to the I/O device through an I/O address. Suppose the CPU wants some data from a part of one of the disk drives. When a PC accesses data from a disk, it can't take it in 1 or 2 bytes, the smallest amount of data that the CPU can ask for is a 512 byte sector.

The operations that take place for programmed I/O are shown in the flowchart in Figure 3. The CPU first checks the status of the disk by reading a special register that can be accessed in the memory space, or by issuing a special I/O instruction. If the disk is not ready to be read or written, then the process loops back and checks the status continuously until the disk is ready. This is referred to as a **busy-wait**. When the disk is finally ready, then a transfer of data is made between the disk and the CPU.

College of Information Technology

After the transfer is completed, the CPU checks to see if there is another communication request for the disk. If there is, then the process repeats, otherwise the CPU continues with another task.

One disadvantage of programmed I/O is that it wastes processor time. Another problem is that high priority devices are not checked until the CPU is finished with its current I/O task, which may have a low priority.

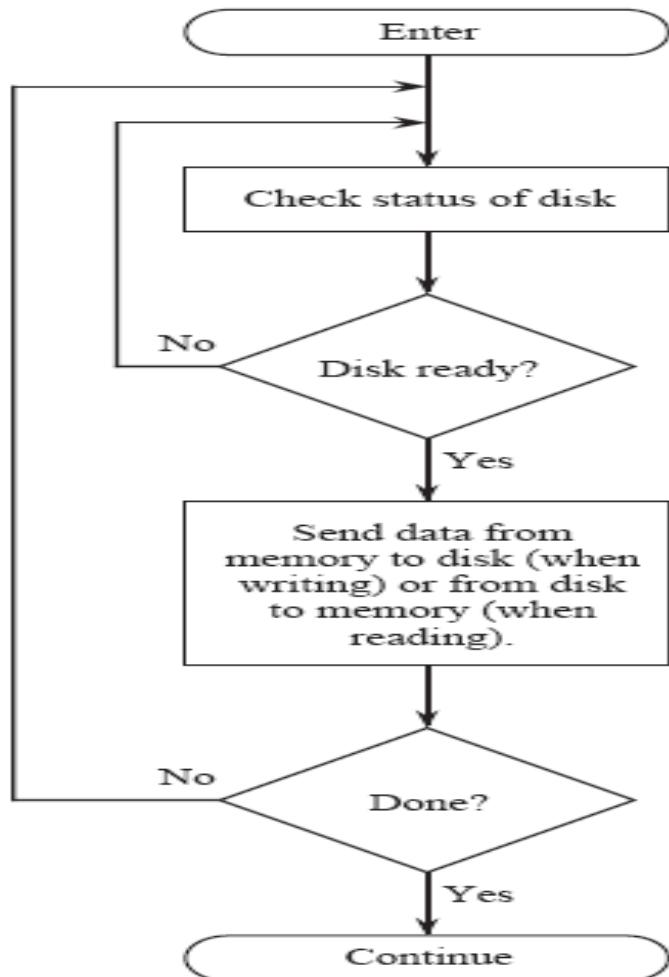


Figure 3 Programmed I/O flowchart for a disk transfer.

In addition to the Programmed I/O, two other mechanisms can be used to carry out I/O operations. These are interrupt-driven I/O and direct memory access (DMA).

DIRECT MEMORY ACCESS (DMA).

Is there another way of performing the I/O activity without wasting the processor's time? Computer systems also employ a similar technique. It is called *direct memory access* (DMA). In DMA, the processor gives the command such as "transfer 10 KB to I/O port 125" and the DMA performs the transfer without bothering the processor. Once the operation is complete, the processor is notified. This notification is done by using an interrupt mechanism. We use DMA to transfer bulk data, not for single word transfers. A special DMA controller is used to direct the DMA transfer operations.

DMA is implemented by using a DMA controller. The DMA controller acts as a slave to the processor and receives data transfer instructions from the processor. Figure 31 shows the difference between programmed I/O and DMA transfer. In programmed I/O, the system bus is used twice as shown in Figure 31a. The DMA transfer not only relieves the processor from the data transfer chore but also makes the transfer process more efficient by transferring data directly from the I/O device to memory.

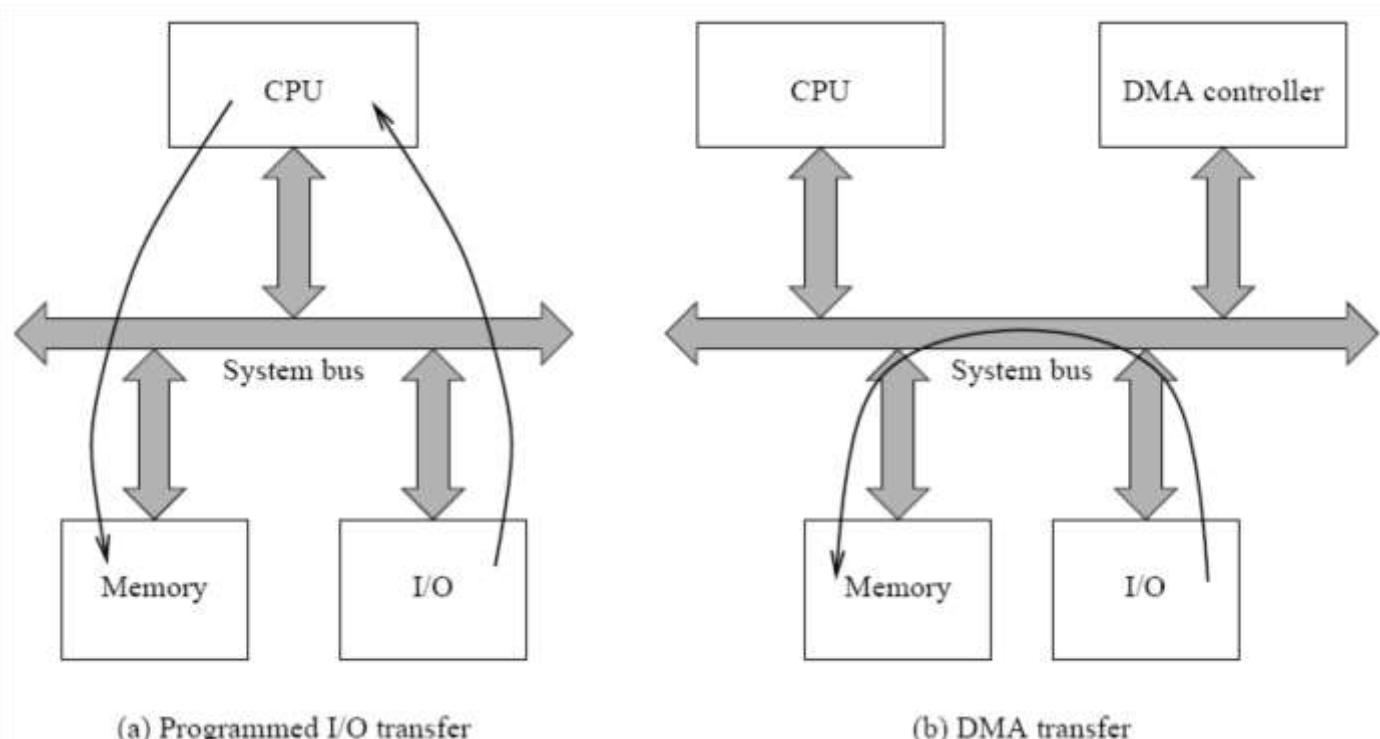


Figure 31 Data transfer from an I/O device to system memory: (a) in programmed I/O, data are read by the processor and then written to the memory; (b) in DMA transfer, the DMA controller generates the control signals to transfer data directly between the I/O device and memory.

INTERRUPT-DRIVEN I/O.

It is often necessary to have the normal flow of a program interrupted, for example, to react to abnormal events, such as power failure. An interrupt can also be used to acknowledge the completion of a particular course of action, such as a printer indicating to the computer that it has completed printing the character(s) in its input register and that it is ready to receive other character(s). The instruction sets of modern CPUs often include instruction(s) that mimic the actions of the hardware interrupts.

When the CPU is interrupted, it is required to discontinue its current activity, attend to the interrupting condition (serve the interrupt), and then resume its activity from wherever it stopped. Discontinuity of the processor's current activity requires finishing executing the current instruction, saving the processor, and transferring control (jump) to what is called the interrupt service routine (ISR). The service offered to an interrupt will depend on the source of the interrupt. For example, if the interrupt is due to power failure, then the action taken will be to save the values of all processor registers and pointers such that resumption of correct operation can be guaranteed upon power return. In the case of an I/O interrupt, serving an interrupt means to perform the required data transfer. Upon finishing serving an interrupt, the processor should restore the original status by popping the relevant values from the stack. Once the processor returns to the normal state, it can enable sources of interrupt again. A flowchart for interrupt driven I/O is shown in Figure 32.

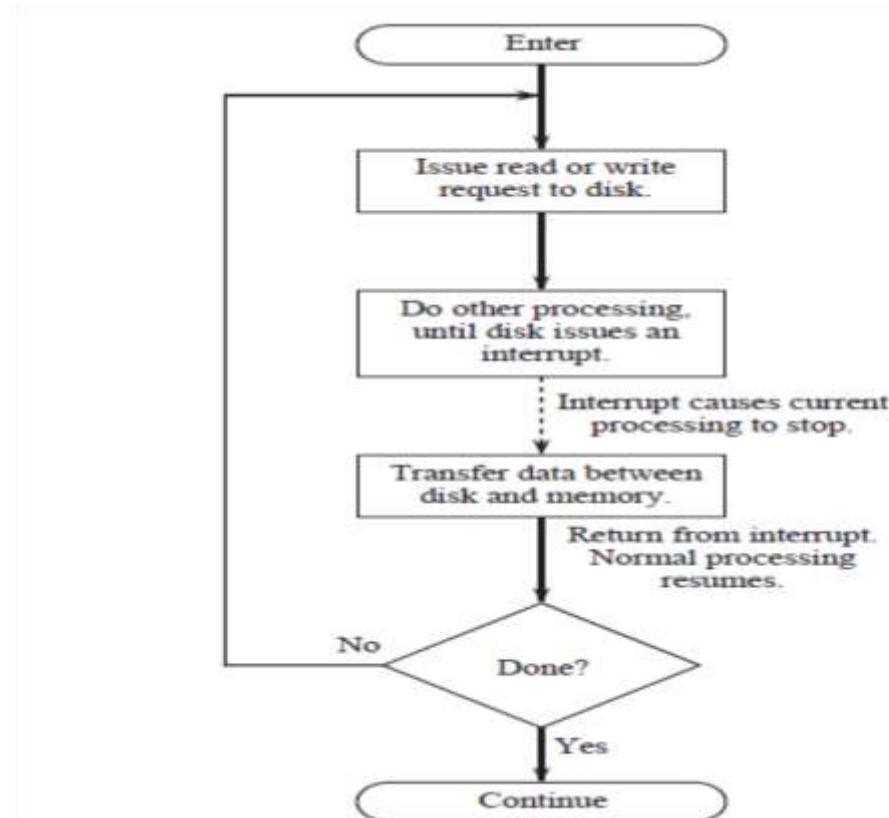


Figure 32 Interrupt driven I/O flowchart for a disk transfer.

Data Transmission.

We can interface I/O devices in one of two basic ways: using a parallel or serial interface. In parallel transmission, several bits are transmitted on parallel wires (***n* bits are transmitted on *n* wires in parallel**) as shown in Figure 33. On the other hand, in the serial transmission, only a single wire is used for data transmission (***one bit at a time***).

Parallel transmission is faster: we can send n bits at a time on an n -bit wide parallel interface. That also means it is expensive compared to the serial interface. A further problem with the parallel interface, particularly at high data transfer rates, is that skew (some bits arrive early and out of sync with the rest) on the parallel data

lines may introduce error prone delivery of data. Because of these reasons, the parallel interface is usually limited to small distances. In contrast, the serial interface is cheaper and does not cause the data skew problems.

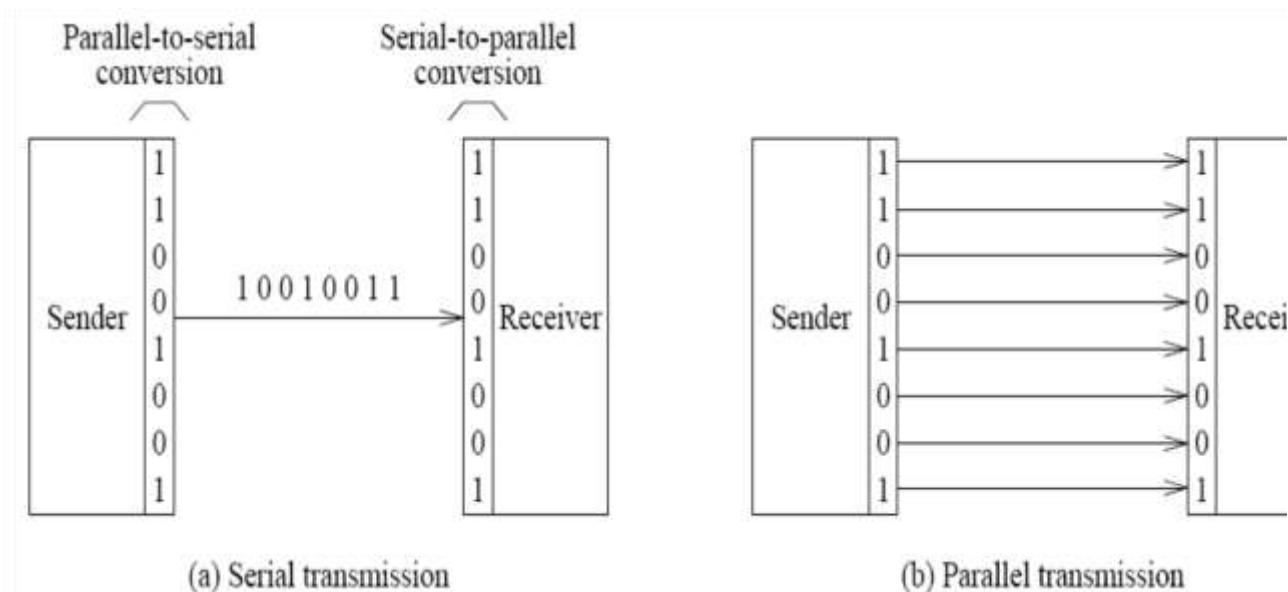


Figure 33 Two basic modes of data transmission.

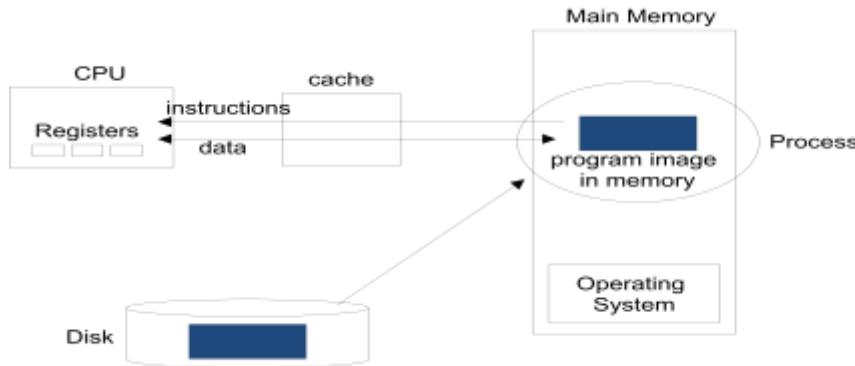
To detect simple errors during data transmission, a single parity bit is added to the 7-bit data. Assume that we are using even parity encoding. That is, every 8-bit character code transmitted will contain an even number of 1 bits. Then, the receiver can count the number of 1s in each received byte and flag transmission error if the byte contains an odd number of 1 bits. Such a simple encoding scheme can detect single bit errors (in fact, it can detect an odd number of single bit errors). To encode, the parity bit is set or cleared depending on whether the remaining 7 bits contain an odd or even number of 1s, respectively. For example, if we are transmitting character A, whose 7-bit ASCII representation is 41H, we set the parity bit to 0 so that there is an even number of 1s.

A **parity bit**, or **check bit**, is a bit added to a string of binary code. Parity bits are a simple form of error detecting code. Parity bits are generally applied to the smallest units of a communication protocol, typically 8-bit octets (bytes), although they can also be applied separately to an entire message string of bits.

The parity bit ensures that the total number of 1-bits in the string is even or odd. Accordingly, there are two variants of parity bits: **even parity bit** and **odd parity bit**. In the case of even parity, for a given set of bits, the occurrences of bits whose value is 1 are counted. If that count is odd, the parity bit value is set to 1, making the total count of occurrences of 1s in the whole set (including the parity bit) an even number. If the count of 1s in a given set of bits is already even, the parity bit's value is 0. In the case of odd parity, the coding is reversed. For a given set of bits, if the count of bits with a value of 1 is even, the parity bit value is set to 1 making the total count of 1s in the whole set (including the parity bit) an odd number. If the count of bits with a value of 1 is odd, the count is already odd so the parity bit's value is 0.

7 bits of data	(count of 1-bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010
1101001	4	11010010	11010011
1111111	7	11111111	11111110

Memory Management



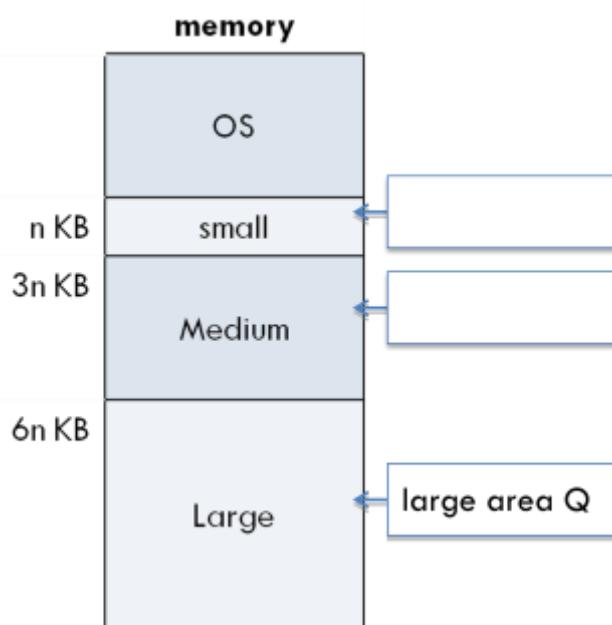
Memory Management

- In a multiprogramming system, in order to share the processor, a number of processes must be kept in memory.
- Memory management is achieved through memory management algorithms.
- Each memory management algorithm requires its own hardware support.
- In order to be able to load programs at anywhere in memory, the compiler must generate relocatable object code.((object code itself, object files may contain metadata used for linking.))
- Also we must make it sure that a program in memory, addresses only its own area, and no other program's area. Therefore, some protection mechanism is also needed

1 . Fixed Partitioning.

- In this method, memory is divided into partitions whose sizes are fixed.
- OS is placed into the lowest bytes of memory.
- Relocation of processes is not needed
- Processes are classified on entry to the system according to their memory requirements.
- We need one *Process Queue (PQ)* for each class of process.

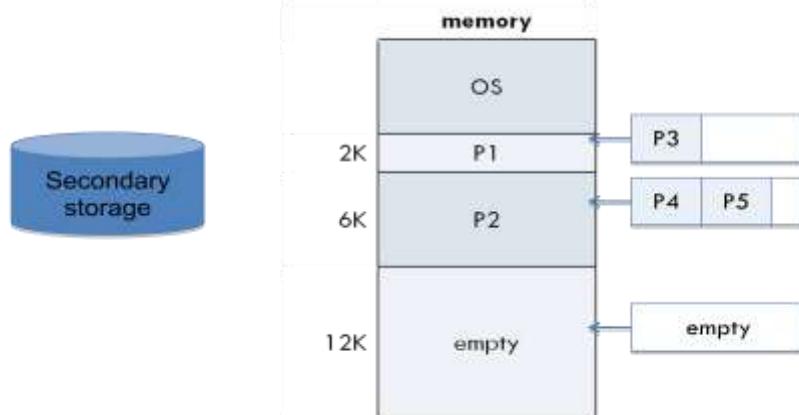
- If a process is selected to allocate memory, then it goes into memory and competes for the processor.
- The number of fixed partition gives the degree of multiprogramming.
- Since each queue has its own memory region, there is no competition between queues for the memory.
- The main problem with the fixed partitioning method is how to determine the number of partitions, and how to determine their sizes.



Fixed Partitioning with Swapping

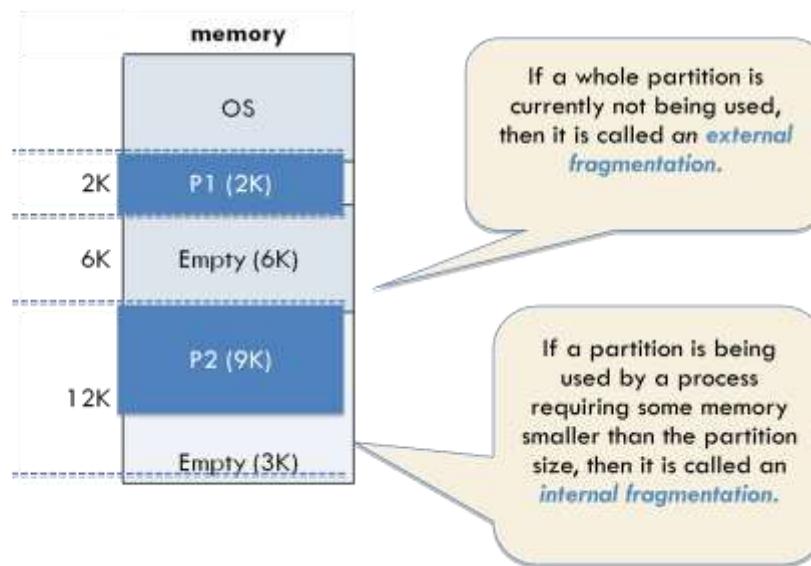
- This is a version of fixed partitioning that uses RRS with some time quantum.
- When time quantum for a process expires, it is swapped out of memory to disk and the next process in the corresponding process queue is swapped into the memory.

Fixed Partitioning with Swapping



Fragmentation

fragmentation



2. Variable Partitioning

- With fixed partitions we have to deal with the problem of determining the number and sizes of partitions to minimize internal and external fragmentation.
- If we use variable partitioning instead, then partition sizes may vary dynamically.

- In the variable partitioning method, we keep a table (linked list) indicating used/free areas in memory.
- There are three algorithms for searching the list of free blocks for a specific amount of memory.

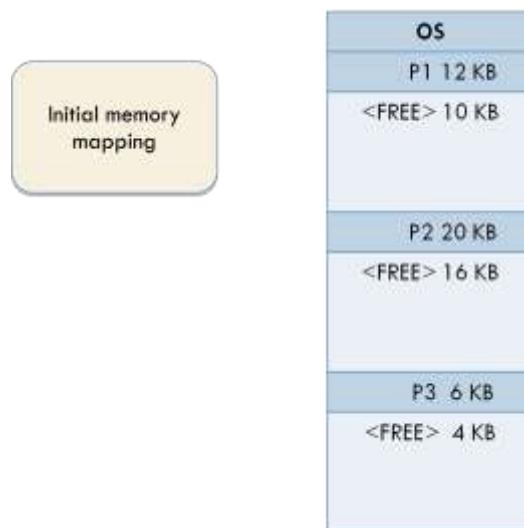
1. First Fit

2. Best Fit

3. Worst Fit

1. first fit

- First Fit : Allocate the first free block that is large enough for the new process.
- This is a fast algorithm.



2. Best fit

- Best Fit : Allocate the smallest block among those that are large enough for the new process.
- In this method, the OS has to search the entire list, or it can keep it sorted and stop when it hits an entry which has a size larger than the size of new process.
- However, it requires more time for searching all the list or sorting it
- If sorting is used, merging the area released when a process terminates to neighboring free blocks, becomes complicated.

3. worst fit

College of Computer Technology

- Worst Fit : Allocate the largest block among those that are large enough for the new process.
- Again a search of the entire list or sorting it is needed.
- This algorithm produces the largest over block.

Compaction

- Compaction is a method to overcome the external fragmentation problem.
- All free blocks are brought together as one large block of free space.
- Compaction requires dynamic relocation.
- Certainly, compaction has a cost and selection of an optimal compaction strategy is difficult.
- One method for compaction is swapping out those processes that are to be moved within the memory, and swapping them into different memory locations

Relocation

- **Static relocation:** A process may be loaded into memory, each time possibly having a different starting address
- Necessary for variable partitioning
- **Dynamic relocation:** In addition to static relocation, the starting address of the process may change while it is already loaded in memory
- Necessary for compaction

Computer Arithmetic: (Integer, Fixed-point, and Floating-point arithmetic).

A computer designer chooses methods of representation in computer on the basis of an evaluation of cost and speed considerations, and occasionally on the basis of accuracy and programmer convenience. A computer design is then chosen that has operations to handle information in those representations. Normally, just a single representation is used for character data. However, a single representation for numeric information is inadequate for a wide range of problems, so general purpose computers frequently have more than one form of number representation. Typically, there are integer and floating-point representation.

Fixed-point is one whose point (decimal or binary) is in a fixed place in relation to the digits. For example, both integers and fractions can be represented as fixed-point numbers. For integers, the point is fixed at the extreme right. For fractions, the point is fixed at the extreme left. Since the point position is fixed, it does not have to be stored in memory with the value.

Floating-point Representation.

Floating-point (FP) number is one in which the position of the point can vary from number to number.

$$\underline{\text{Ex1:- }} 125 \rightarrow 0.125 \times 10^3 \rightarrow 1.25 \times 10^2 \rightarrow 12.5 \times 10^1$$

$$\Downarrow \qquad \qquad \Downarrow$$

$$0.125 \times E3 \rightarrow 1.25 \times E2$$

$$\underline{\text{Ex2:- }} 0.00127 \rightarrow 12.7 \times 10^{-4} \rightarrow 1.27 \times 10^{-3} \rightarrow 0.0127 \times 10^{-1}$$

$$\Downarrow$$

$$12.7 \times E-4$$

In digital computers, floating-point numbers consist of three parts (figure 1):

Sign, exponent, and mantissa

After a binary number is normalized, only three pieces of information about the number are stored: sign, exponent, and mantissa (the bits to the right of the decimal point). For example, +1000111.0101 becomes:

<i>Sign</i>	<i>Exponent</i>	<i>Mantissa</i>
+	2^6	1.0001110101
1	6	0001110101

Sign

The sign of the number can be stored using 1 bit (0 or 1).

Exponent

The exponent (power of 2) defines the shifting of the decimal point. Note that the power can be negative or positive. The Excess representation (discussed later) is the method used to store the exponent.

Mantissa

The mantissa is the binary integer to the right of the decimal point. It defines the precision of the number. The mantissa is stored in fixed-point notation. If we think of the mantissa and the sign together, we can say this combination is stored as an integer in sign-andmagnitude format. However, we need to remember that it is not an integer—it is a fractional part that is stored like an integer. We emphasize this point because in a mantissa,

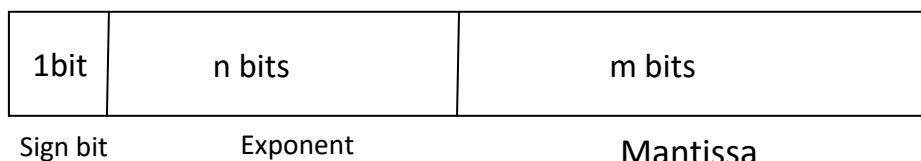


Figure 1 Representation of a floating-point number

Ex3:- (+6132.789)₁₀ can be represented in floating-point as:

$$+0.6132789 \times 10^{+4}$$

The mantissa here is considered to be a fixed-point fraction, so the decimal point is assumed to be at the left of the most significant digit. The exponent contains the decimal number +4 to indicate that the actual position of the decimal point is *four* decimal position to the right of the assumed decimal point.

The number of bits used for the **exponent** and **mantissa** depends on whether we would like to optimize for range (more bits in the exponent) or precision (more bits in the mantissa).

Suppose we will use a 14-bits model with a 5-bit exponent, an 8-bit **mantissa**, and a sign bit (see [Figure 3](#))

1bit	5 bits	8 bits
Sign bit	Exponent	Mantissa

Figure 4: Floating-Point Representation with 14 bits model

Let's say that we wish to store the decimal number 17 in our model. We know that $17 = 17.0 \times 10^0 = 1.7 \times 10^1 = 0.17 \times 10^2$. Analogously, in binary, $17_{10} = 10001_2 \times 2^0 = 1000.1_2 \times 2^1 = 100.01_2 \times 2^2 = 10.001_2 \times 2^3 = 1.0001_2 \times 2^4 = 0.10001_2 \times 2^5$. If we use this last form, our fractional part will be 10001000 and our exponent will be 00101, as shown here:

0	00101	10001000
---	-------	----------

Example/

Show the Excess_127 (single precision) representation of the decimal number 5.75.

Solution

- a. The sign is positive, so S = 0.
- b. Decimal to binary transformation: $5.75 = (101.11)_2$.
- c. Normalization: $(101.11)_2 = (1.0111)_2 \times 2^2$.
- d. $E = 2 + 127 = 129 = (10000001)_2$, M = 0111. We need to add 19 zeros at the right of M to make it 23 bits.
- e. The presentation is shown below:

S	E	M
0	10000001	0111000000000000000000000

The number is stored in the computer as 01000000101110000000000000000000.

One obvious problem with this model is that we haven't provided for negative exponents. The exponent in the above figures, can only represent positive numbers. If we wanted to store 0.25 we would have no way of doing so because $0.25 = 2^{-2}$ and the exponent -2 cannot be represented. To represent both positive and negative exponents, a fixed value, called a *bias*, is used.

The idea behind using a bias value is to convert every integer in the range into a non-negative integer, which is then stored as a binary numeral. The integers in the desired range of exponents are first adjusted by adding this fixed bias value to each exponent. *The bias value is a number near the middle of the range of possible values.* In this case, we could select 16 because it is midway between 0 and 31 (**our exponent has 5 bits, thus allowing for 2⁵ or 32 values**). Any number larger than 16 in the exponent field will represent a positive value. Values less than 16 will indicate negative values. This is called an **excess-16** representation because we have to subtract 16 to get the true value of the exponent.

Floating-Point Arithmetic Addition/Subtraction.

If we wanted to add two decimal numbers that are expressed in scientific notation, such as $1.5 \times 10^2 + 3.5 \times 10^3$, we would change one of the numbers so that both of them are expressed in the same power of the base. In our example, $1.5 \times 10^2 + 3.5 \times 10^3 = 0.15 \times 10^3 + 3.5 \times 10^3 = 3.65 \times 10^3$. Floating-point addition and subtraction work the same way, as illustrated below.

Example 5:

Add the following binary numbers as represented in a normalized 14-bit format with a bias of 16.

0	1	0	0	1	0	1	1	0	0	1	0	0	0	+
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0	1	0	0	0	0	1	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

We see that the addend is raised to the second power. Alignment of these two operands on the binary point gives us:

$$\begin{array}{r}
 11.001000 \\
 + 0.10011010 \\
 \hline
 11.10111010
 \end{array}$$

Operating system

An operating system is complex, so it is difficult to give a simple universal definition. Instead, here are some common definitions:

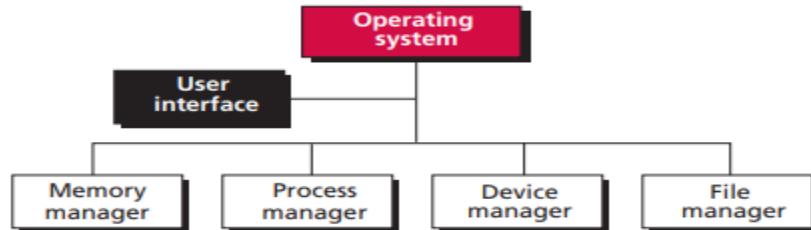
- An operating system is an interface between the hardware of a computer and the user (programs or humans).
- An operating system is a program (or a set of programs) that facilitates the execution of other programs.
- An operating system acts as a general manager supervising the activity of each component in the computer system. As a general manager, the operating system checks that hardware and software resources are used efficiently, and when there is a conflict in using a resource, the operating system mediates to solve it.

An operating system is an interface between the hardware of a computer and the user (programs or humans) that facilitates the execution of other programs and the access to hardware and software resources.

COMPONENTS

Today's operating systems are very complex. An operating system needs to manage different resources in a computer system. It resembles an organization with several managers at the top level. Each manager is responsible for managing their department, but also needs to cooperate with others and coordinate activities. A modern operating system has at least four duties:

- memory manager,
- process manager,
- device manager,
- and file manager

Components of an operating system**1- User interface**

Each operating system has a user interface, a program that accepts requests from users (processes) and interprets them for the rest of the operating system. A user interface in some operating systems, such as UNIX, is called a shell. In others, it is called a window to denote that it is menu driven and has a GUI (graphical user interface) component.

2- Memory manager

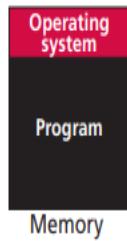
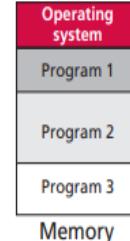
One of the responsibilities of a modern computer system is memory management. Although the memory size of computers has increased tremendously in recent years, so has the size of the programs and data to be processed. Memory allocation must be managed to prevent applications from running out of memory. Operating systems can be divided into two broad categories of memory management: monoprogramming and multiprogramming.

- Monoprogramming

Monoprogramming belongs to the past, but it is worth mentioning because it helps us to understand multiprogramming. In monoprogramming, most of the memory capacity is dedicated to a single program.

- Multiprogramming

In multiprogramming, more than one program is in memory at the same time, and they are executed concurrently, with the CPU switching rapidly between the programs.

Monoprogramming*Multiprogramming***3-Process manager**

A second function of an operating system is process management, but before discussing this concept, we need to define some terms.

Program, job, and process

Modern operating systems use three terms that refer to a set of instructions: program, job, and process. Although the terminology is vague and varies from one operating system to another, we can define these terms informally.

Program

A program is a nonactive set of instructions stored on disk (or tape). It may or may not become a job.

Job

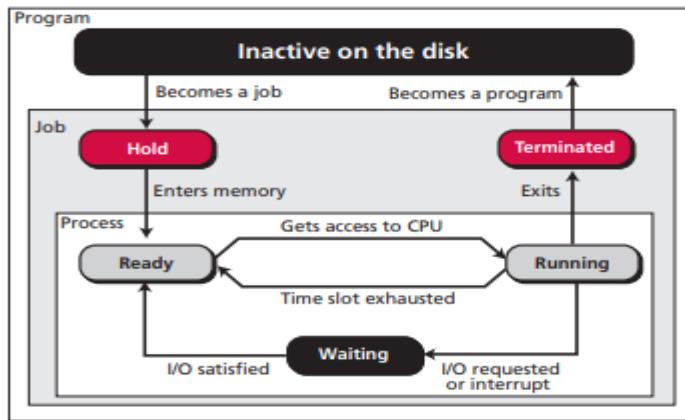
A program becomes a job from the moment it is selected for execution until it has finished running and becomes a program again. During this time a job may or may not be executed.

Process

A process is a program in execution. It is a program that has started but has not finished. In other words, a process is a job that is being run in memory. It has been selected among other waiting jobs and loaded into memory.

State diagrams

The relationship between a program, a job, and a process becomes clearer if we consider how a program becomes a job and how a job becomes a process. This can be illustrated with a state diagram that shows the different states of each of these entities.

State diagram with boundaries between program, job, and process

A program becomes a job when selected by the operating system and brought to the hold state. It remains in this state until it can be loaded into memory. When there is memory space available to load the program totally or partially, the job moves to the ready state. It now becomes a process. It remains in memory and in this state until the CPU can execute it, moving to the running state at this time. When in the running state, one of three things can happen:

- The process executes until it needs I/O resources
- The process exhausts its allocated time slot
- The process terminates

In the first case, the process goes into the waiting state and waits until I/O is complete. In the second case, it goes directly to the ready state. In the third case, it goes into the terminated state and is no longer a process. A process can move between the running, waiting, and ready states many times before it goes to the terminated state.

4- File manager

Operating systems today use a file manager to control access to files. A detailed discussion of the file manager also requires advanced knowledge of operating system principles and file access concepts.

A SURVEY OF OPERATING SYSTEMS

1- UNIX

It has been a popular operating system among computer programmers and computer scientists. It is a very powerful operating system with three outstanding features.

First, UNIX is a portable operating system that can be moved from one platform to another without many changes. The reason is that it is written mostly in the C language (instead of a machine language specific to a particular computer system).

Second, UNIX has a powerful set of utilities (commands) that can be combined (in an executable file called a script) to solve many problems that require programming in other operating systems.

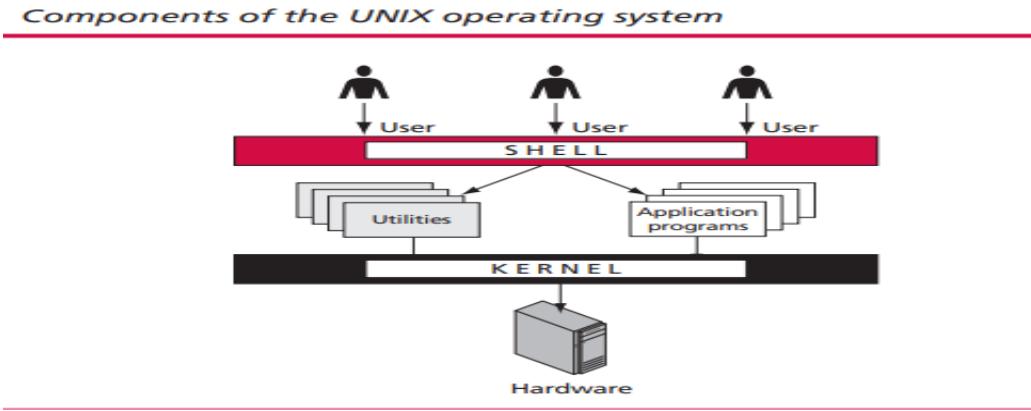
Third, it is device-independent, because it includes device drivers in the operating system itself, which means that it can be easily configured to run any device.

UNIX is a multiuser, multiprocessing, portable operating system designed to facilitate programming, text processing, communication, and many other tasks that are expected from an operating system.

UNIX is a multiuser, multiprocessing, portable operating system. It is designed to facilitate programming, text processing, and communication.

UNIX structure

UNIX consists of four major components: the kernel, the shell, a standard set of utilities, and application programs.



The kernel

The kernel is the heart of the UNIX system. It contains the most basic parts of the operating system: memory management, process management, device management, and file management.

The shell

The shell is the part of UNIX that is most visible to the user. It receives and interprets the commands entered by the user.

Utilities

There are literally hundreds of UNIX utilities. A utility is a standard UNIX program that provides a support process for users. Three common utilities are text editors, search programs, and sort programs.

Applications

Applications in UNIX are programs that are not a standard part of the operating system distribution. Written by systems administrators, professional programmers, or users, they provide extended capabilities to the system.

2- Linux

new operating system that is known today as Linux. The initial kernel, which was similar to a small subset of UNIX, has grown into a full-scale operating system today. it has all the features traditionally attributed to UNIX . the main component:-

Kernel

The kernel is responsible for all duties attributed to a kernel, such as memory management, process management, device management, and file management.

System libraries

The system libraries hold a set of functions used by the application programs, including the shell, to interact with the kernel.

System utilities

The system utilities are individual programs that use the services provided by the system libraries to perform management tasks.

3-Windows

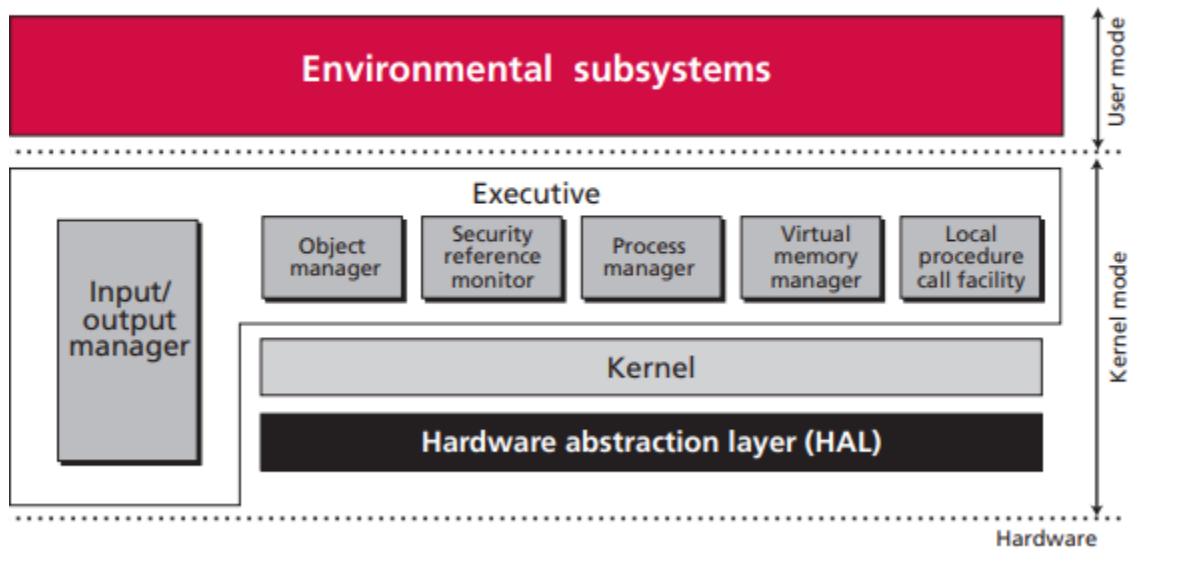
Microsoft started development of a new single-user operating system to replace MS-DOS (Microsoft Disk Operating System). Windows was the result. Several versions of Windows followed. We refer to all of these versions as Windows.

Design goals

Design goals released by Microsoft are extensibility, portability, reliability, compatibility, and performance.

Architecture

The architecture of Windows



HAL

The hardware abstraction layer (HAL) hides hardware differences from the upper layers.

Kernel

The kernel is the heart of the operating system. It is an object-oriented piece of software that sees any entity as an object.

Executive

The Windows executive provides services for the whole operating system. It is made up of six subsystems: object manager, security reference monitor, process manager, virtual memory manager, local procedure call facility, and the I/O manager. Most of these subsystems are familiar from our previous discussions of operating subsystems. Some subsystems, like the object manager, are added to Windows because of its object-oriented nature. The executive runs in kernel (privileged) mode.

Environmental subsystems

These are subsystems designed to allow Windows to run application programs designed for Windows, for other operating systems, or for earlier versions of Windows. The native subsystem that runs applications designed for Windows is called Win32. The environment subsystems run in the user mode (a non-privileged mode).

Conclusion

In this lecture we discuss all the lectures that we had it in this season, beginning from the introduction to the last lectures (operating system).

1-Introduction.

Although it is difficult to distinguish between the ideas belonging to computer organization and those ideas belonging to computer architecture, it is impossible to say where hardware issues end and software issues begin. Computer scientists design algorithms that usually are implemented as programs written in some computer language, such as Java or C. But what makes the algorithm run? Another algorithm, of course! And another algorithm runs that algorithm, and so on until you get down to the machine level, which can be thought of as an algorithm implemented as an electronic device.

We begin our discussion of computer hardware by looking at the components necessary to build a computing system. **At the most basic level, a computer is a device consisting of three pieces:**

1. A **processor** to interpret and execute programs
2. A **memory** to store both data and programs
3. A **mechanism** for transferring data to and from the outside world

2- Memory Addressing

The memory of a computer system consists of tiny electronic switches, with each switch in one of two states: ***open*** or ***closed***. It is, however, more convenient to think of these states as **0** and **1**, rather than open and closed. Thus, each switch can represent a bit. The memory unit consists of millions of such bits. In order to make memory more manageable, eight bits are grouped into a byte. Memory can

then be viewed as consisting of an ordered sequence of bytes. Each byte in this memory is identified by its sequence number starting with 0, as shown in Figure 1. This is referred to as the ***memory address*** of the byte. Such memory is called ***byte addressable*** memory because each byte has a unique address.

3- Basic Memory Operations

The memory unit supports two basic operations: *read* and *write*. The *read operation* reads previously stored data and the *write operation* stores a new value in memory. Both of these operations require a memory address. In addition, the write operation requires specification of the data to be written. The address and data of the memory unit are connected to the address and data buses of the system bus, respectively. The read and write signals come from the control bus. Two metrics are used to characterize memory. ***Access time*** refers to the amount of time required by the memory to retrieve the data at the addressed location. The other metric is the ***memory cycle time***, which refers to the minimum time between successive memory operations.

4- Cache Memory

Cache memory is much smaller than the main memory. Usually implemented using SRAMs, which sits between the processor and main memory in the memory hierarchy. The cache effectively isolates the processor from the slowness of the main memory, which is DRAM-based. The principle behind the cache memories is to prefetch the data from the main memory before the processor needs them. If we are successful in predicting the data that the processor needs in the near future, we can preload the cache and supply those data from the faster cache. It turns out predicting the processor future access requirements is not difficult owing to a phenomenon known as *locality of reference* that programs exhibit.

5- System bus.

A computer system consists of three major components: a processor, a memory unit, and an input/output (I/O) subsystem. Interconnection network facilitates communication among these three components, as shown in Figure 21. The interconnection network is called the *system bus*.

6- The Von Neumann Model.

John von Neumann, along with others, proposed the concept of the stored program that we use even today. The idea was to keep a program in the memory and read the instructions from it. He also proposed an architecture that clearly identified the components we have presented previously: ALU, control, input, output, and memory.

7- mid Exam

8- DIRECT MEMORY ACCESS (DMA).

Is there another way of performing the I/O activity without wasting the processor's time? Computer systems also employ a similar technique. It is called *direct memory access* (DMA). In DMA, the processor gives the command such as "transfer 10 KB to I/O port 125" and the DMA performs the transfer without bothering the processor. Once the operation is complete, the processor is notified. This notification is done by using an interrupt mechanism.

9- INTERRUPT-DRIVEN I/O.

It is often necessary to have the normal flow of a program interrupted, for example, to react to abnormal events, such as power failure. An interrupt can also be used to acknowledge the completion of a particular course of action, such as a printer indicating to the computer that it has completed printing the character(s) in its input register and that it is ready to receive other character(s). The instruction sets

of modern CPUs often include instruction(s) that mimic the actions of the hardware interrupts.

10- Memory Management

- In a multiprogramming system, in order to share the processor, a number of processes must be kept in memory.
- Memory management is achieved through memory management algorithms.
- Each memory management algorithm requires its own hardware support.
- In order to be able to load programs at anywhere in memory, the compiler must generate relocatable object code.((object code itself, object files may contain metadata used for linking.))
- Also we must make it sure that a program in memory, addresses only its own area, and no other program's area. Therefore, some protection mechanism is also needed

11- Computer Arithmetic: (Integer, Fixed-point)

A computer designer chooses methods of representation in computer on the basis of an evaluation of cost and speed considerations, and occasionally on the basis of accuracy and programmer convenience. A computer design is then chosen that has operations to handle information in those representations. Normally, just a single representation is used for character data. However, a single representation for numeric information is inadequate for a wide range of problems, so general purpose computers frequently have more than one form of number representation. Typically, there are integer and floating-point representation.

12- Floating-point Representation.

Floating-point (FP) number is one in which the position of the point can vary from number to number.

13- Operating system

An operating system is complex, so it is difficult to give a simple universal definition. Instead, here are some common definitions:

- An operating system is an interface between the hardware of a computer and the user (programs or humans).
- An operating system is a program (or a set of programs) that facilitates the execution of other programs.
- An operating system acts as a general manager supervising the activity of each component in the computer system. As a general manager, the operating system checks that hardware and software resources are used efficiently, and when there is a conflict in using a resource, the operating system mediates to solve it.

14- A SURVEY OF OPERATING SYSTEMS

- **Unix**
- **Linux**
- **windows**