FEDERAL UNIVERSITY OF TECH. MINNA

CPT 214
COMPUTER ARCHITECTURE
HAND OUT

CPT214: COMPUTER ARCHITECTURE (3 UNITS)

2010-2011 SESSION

Contents

1.1 Boolean Algebra

Before going further, let us briefly recapitulate the Boolean algebra, which will be useful in the discussions on logic circuits. The Boolean algebra is an attempt to represent the true-false logic of humans in mathematical form. George Boole proposed the principles of the Boolean algebra in 1854, hence the name Boolean algebra. Boolean algebra is used for designing and analysing digital circuits. Let us first discuss the rules of Boolean algebra and thereafter we will discuss how it can be used in analyzing or designing digital circuits.

Point 1:

A variable in Boolean algebra can take only two values

1 (TRUE) or 0 (FALSE)

Point 2:

There are three basic operations in Boolean algebra, viz:

AND, OR and NOT

(These operators will be given in capitals in this module for differentiating them from normal and, or, not, etc)

A AND B or A.B or AB

A OR B or A + B

NOT A or ¬A or A ’ or ?

But how the value of A AND B changes with the values of A and B can be represented in tabular form, which is referred to as the “truth table”.

A	B	A AND B	A OR B	NOT A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

In addition three more operators have been defined for Boolean algebra:

XOR (Exlusive OR), NOR (Not OR) and NAND (Not AND)

However, for designing and analysing a logical circuit, it is convenient to use AND, NOT and OR operators because AND and OR obey many laws as of multiplication and addition in the ordinary algebra of real numbers.

We will discuss XOR, NOR and NAND operators in the next subsection.

Point 3:

The basic logical identities used in Boolean algebra are:

BASIC IDENTITIES

$A.B = B.A$	$A+B = B+A$	Commutative law
$A.(B \mid C) = (A.B) \mid (A.C)$	$A \mid (B.C) = (A \mid B).(A \mid C)$	Distributive law
$1.A = A$	$0+A = A$	Identity law
$A.\bar{A} = 0$	$A+\bar{A} = 1$	Inverse law

OTHER IDENTITIES

$0.A = 0$	$1+A = 1$	
$A.A = A$	$A+A = A$	
$A.(B.C) = (A.B).C$	$A+(B+C) = (A+B)+C$	Associative law
$A.B = \bar{A} \mid B$	$A \mid B = \bar{A}.B$	Demorgan's Theorem

We will not give any proofs of these identities. The use of some of the identities is shown in the example given after Boolean function definition. DeMorgan's law is very useful in simplifying logical expressions.

1.2 Boolean Function*:*

A Boolean function is defined as an algebraic expression formed with the binary variables, the logic operation symbols, parenthesis, and equal to sign. For example,

F = A.B+C is a Boolean function.

The value of Boolean function F can be either 0 or 1.

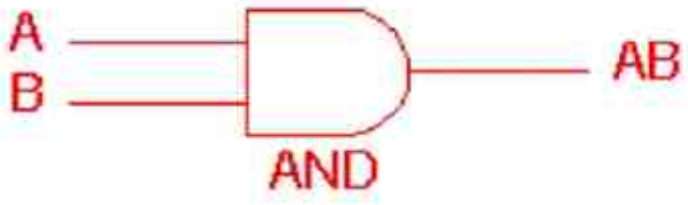
A Boolean function can be broken into logic diagram and vice versa (we will discuss this in the next section), therefore if we code the logic operations in Boolean algebraic form and simplify this expression we will design the simplified form of the logic circuits.

2.1 Logic Gates

Digital systems are said to be constructed by using logic gates. A logic gate is an electronic circuit which produces a typical output signal depending on its input signal. The output signal of a gate is a simple Boolean operation of its input signal(s). Gates are the basic logic elements. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. Any Boolean function can be represented in the form of gates.

The basic operations are described below with the aid of truth tables.

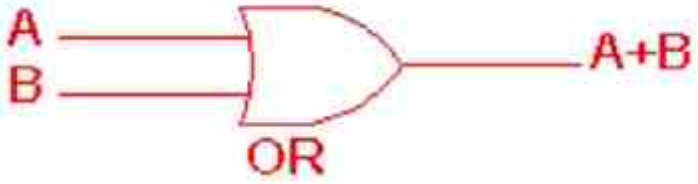
AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

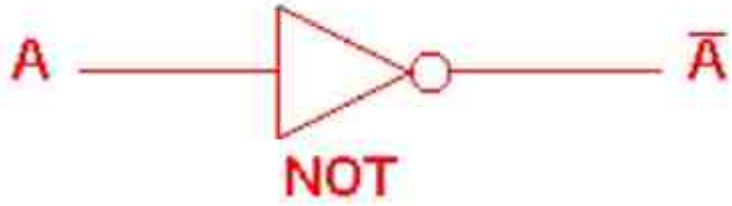
OR gate



2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

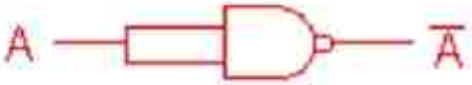
The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

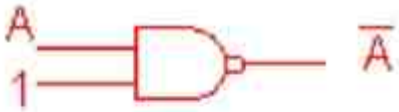
NOT gate



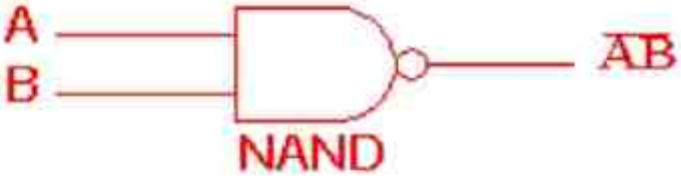
NOT gate	
A	$\overline{A}$
0	1
1	0

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an **inverter**. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways in which the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.





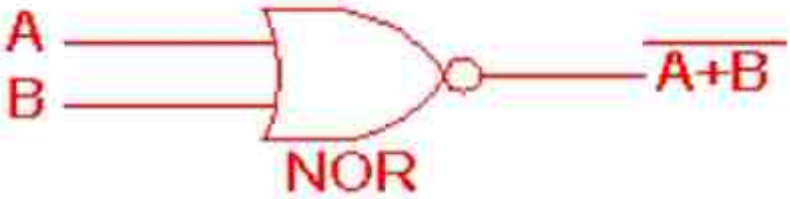
NAND gate



2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents an inversion.

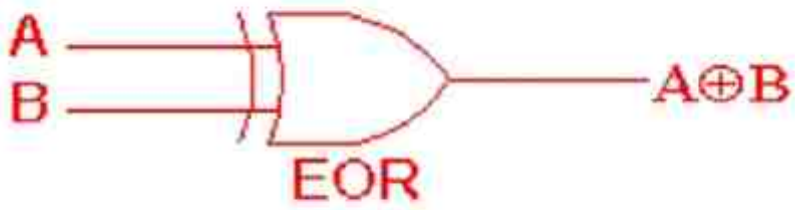
NOR gate



2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if **any** of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents an inversion.

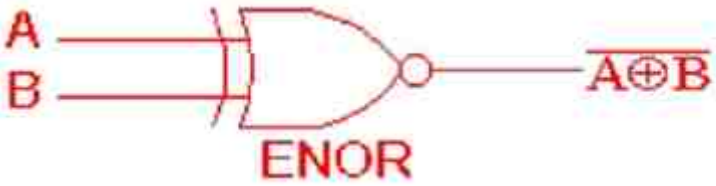
EXOR gate



2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign ( ) is used to show the EOR operation.

EXNOR gate



2 Input EXNOR gate		
A	B	$\overline{A⊕B}$
0	0	1
0	1	0
1	0	0
1	1	1

The '**Exclusive-NOR**' gate circuit does the opposite of the EOR gate. It will give a low output if **either, but not both** of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents an inversion. The NAND and NOR gates are called **universal functions** since with either one the AND and OR functions and NOT can be generated.

Note:

A function in **sum of products** form can be implemented using NAND gates by replacing all AND and OR gates by NAND gates. A function in **product of sums** form can be implemented using NOR gates by replacing all AND and OR gates by NOR gates.

Table 1: Logic gate symbols

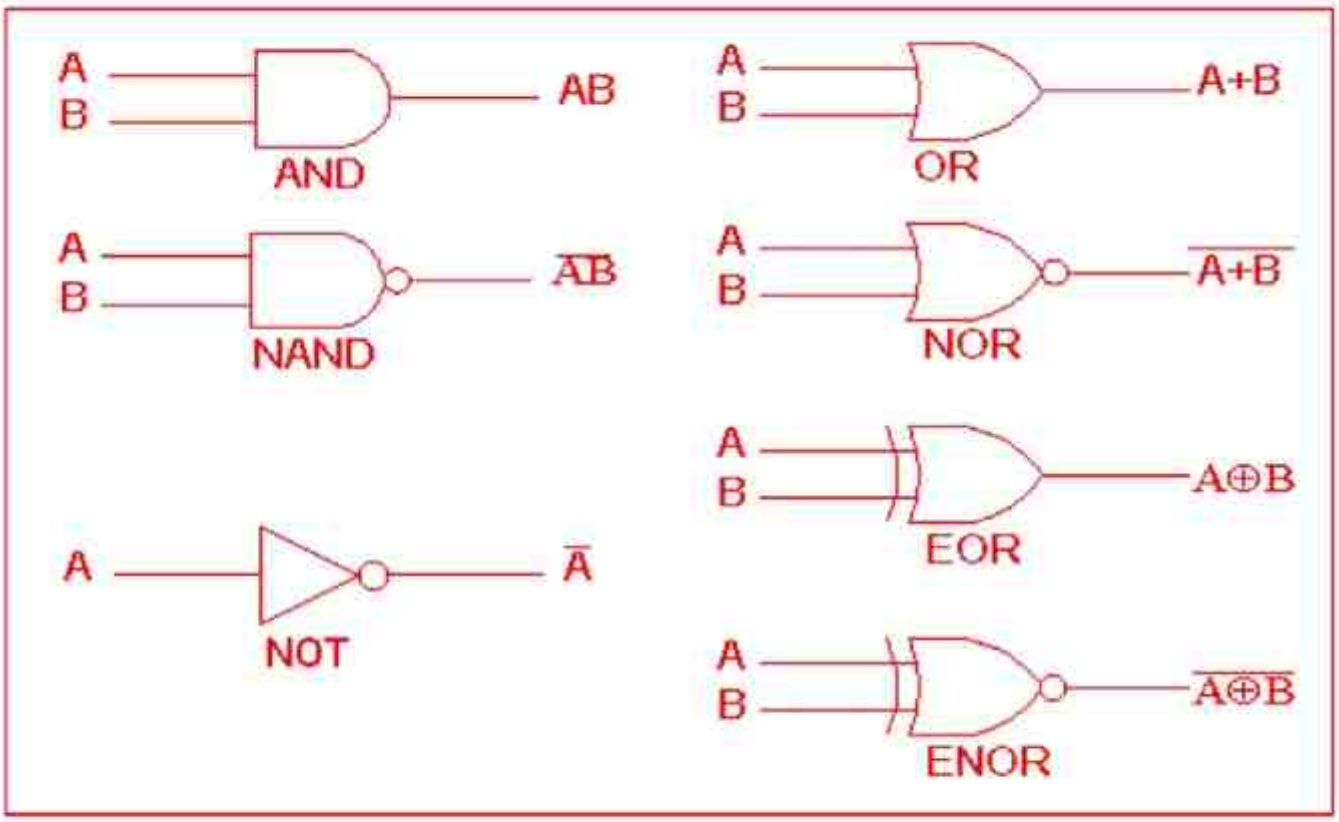


Table 2 is a summary truth table of the input/output combinations for the NOT gate together with all possible input/output combinations for the other gate functions. Also note that a truth table with 'n' inputs has 2^n rows. You can compare the outputs of different gates.

Table 2: Logic gates representation using the Truth table

NOT gate	
A	$\overline{A}$
0	1
1	0

INPUTS		OUTPUTS					
A	B	AND	NAND	OR	NOR	EXOR	EXNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

The truth table of NAND and NOR can be made from NOT (A AND B) and NOT (A OR B) respectively. Exclusive OR (XOR) is a special gate whose output is one only if the inputs are not equal. The inverse of exclusive OR can be a comparator which will produce a one output if two inputs are equal.

The digital circuit use only one or two types of gates for simplicity in fabrication purposes. Therefore, one must think in terms of functionally complete sets of gates. What does a functionally complete set imply? A set of gates by which any Boolean function can be implemented is called a functionally complete set. The functionally compete sets are: (AND, NOT), (NOR), (NAND), (OR< NOT) etc.

2.2 Combination Circuits

Combinational circuits are interconnected circuits of gates according to a certain rule to produce an output depending on its input value. A well formed combinational circuit should not have feedback loops. A combinational circuit can be represented as a network of gates and, can be expressed by a truth table or a Boolean expression.

The output of a combinational circuit is related to its input by a combinational function, which is independent of time. Therefore, for an ideal combinational circuit the output should change instantaneously according to changes in input. But in actual cases there is a slight delay. This delay is normally proportional to the depth or number of levels, i.e the maximum number of gates lying on any path from input to output.

For example, the depth of the combinational circuit in figure 3 is two

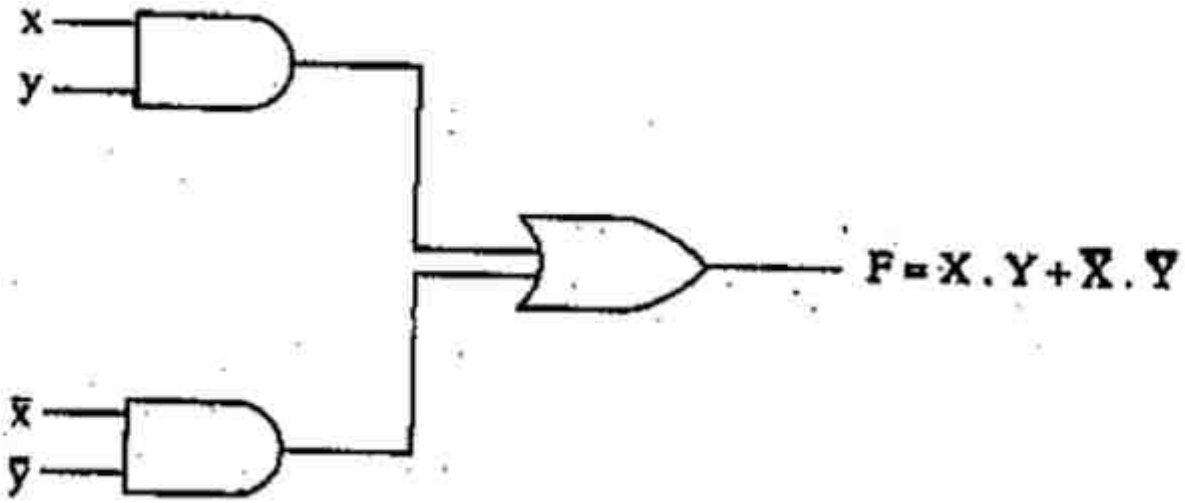


Figure 17: A two level AND-OR combinational circuit

The basic design issue related to combinational circuits is: the minimization of the number of gates. The normal circuit constraints for combinational circuit design are:

· The depth of the circuit should not exceed a specific level.

· Number of input lines to a gate (fan in) and how many gates its output can be fed (fan out) are constrained by the circuit power constraints.

2.3 Minimization of Gates

The simplification of the Boolean expression is very useful for combinational circuit designs. The following three methods are used for this.

· Algebraic simplification

· Karnaugh maps

· The Quine McCluskey method

But before defining any of the above stated methods let us discuss the forms of algebraic expressions. An algebraic expression can exist in two forms:

· Sum of products (SOP) e.g. (A.¬B) + (¬A. ¬B)

· Product of sums (POS) e.g. (¬A +¬B).(A+B)

If a product of SOP expression contains every variable of that function either in true or complement form then it is defined as a minterm. This minterm will be true only for one combination of input values of the variables. For example, in the SOP expression-

F (A,B,C) = (A.B.C) + (¬A. ¬B.C) + (A.B)

We, have three product terms namely A.B.C, ¬A. ¬B.C and A.B. But only the first two of them qualify to be a minterm as the third one does not contain variable C or its complement. In addition, the term A.B.C will be one only if A=1, B=1 and C=1 for any other combination of values of A,B,C the minterm will have zero value. Similarly, the

minterm ¬A. ¬B.C will have value 1 only if ¬A = 1 i.e. A=0, ¬B=1 i.e. B=0 and C=1.For any other combination of values the minterm will have a zero value.

A similar type of term used in POS form is called maxterm. Maxterm is a term of POS expression, which contains all the variables of the function in true or complemented form. For example,

F(A,B,C)=(A+B+C).(¬A+¬B+C) have two maxterms. A maxterm have a value 0 for only one combination of input values. The maxterm A+B+C will be 0 value only for A=0, B=0 and C=0. For all other combination of values of A, B, C it will have a value one.

Now let us come back to the problem of minimizing the number of gates.

2.4 Algebraic Simplification

We have already discussed the algebraic simplification of a logical circuit. An algebraic expression can exist in POS or SOP forms. The algebraic functions can appear in many different forms. Although the process of simplification exists yet it is cumbersome because of the absence of routes, which tell what rule to apply next. The Karnaugh map is a simple direct approach of simplification of logical expressions.

2.5 Karnaugh Maps

The Karnaugh map is a convenient way of representing and simplifying Boolean functions of 4 to 6 variables. Karnaugh maps can also be used for designing the circuits in situations where you can construct the truth table for an operation or a function. In other words, Karnaugh maps can be used to construct a circuit when the input and output to that proposed circuit are defined. For each output one Karnaugh map needs to be constructed.

2.6 Programmable Logic Array

Till now, the individual gates are treated as basic building blocks from which various logic functions can be derived. We have also learnt about the strategies of minimisation of number of

gates. But with the advancement of technology the integration provided by integrated circuit technology has increased resulting in the production of one to ten gates on a single chip (in Small Scale Integration). The gate level designs are constructed at the gate level only but if the design is to be done using these SSI chips, the design consideration needs to be changed as a number of such SSI chips may be used for developing a logic circuit. With MSI & VLSI we can put even more gates on a chip and can also make gate interconnections on a chip. This integration and connection bring the advantages of decreased cost, size, and increased speed. But the basic drawback faced in such VLSI & MSI chips is that for each logic function the layout of gates and interconnection need to be designed. The cost involved in making such a custom chip design is quite high. Thus came the concept of Programamble Logic Array (PLA), a general-purpose chip that can be readily adopted for any specific purpose.

The PLA are designed for SOP form of Boolean function and consist of a regular arrangement of NOT, AND & OR gates on a chip. Each input to the chip is passed through a NOT gate, thus, the input and their complement are available to each AND gate. The output of each AND gate is made available for each OR gate. The output of each OR gate is treated as chip output. By making appropriate connections any logic function can be implemented in these Programmable Logic Arrays.

3.0 THE MEMORY SYSTEM

The memory in a computer system is required for storage and subsequent retrieval of the instructions and data. A computer system uses a variety of devices for storing these instructions and data that are required for its operations. Normally we classify the information to be stored on computer in two basic categories: data and instructions. The storage devices along with the algorithm or information on how to control and manage these storage devices constitute the memory system of a computer.

A memory system is a very simple system yet it exhibits a wide range of technology and types. The basic objective of a computer system is to increase the speed of computation. Likewise the basic objective of a memory system is to provide fast, uninterrupted access by the processor to the memory such that the processor can operate at the speed at which it is expected to work.

But does this kind of technology where there is no speed gap between the processor and the memory speed exist? Yes, it does, but unfortunately as the access time (time taken by CPU to

access a location in memory) becomes less and less the cost per bit of memory becomes increasingly higher. In addition, normally these memories require power supply till the information needs to be stored. These things are not very convenient, but on the other hand the memories with smaller cost have very high access time that will result in slower operations of the CPU.

Thus, the cost versus access time anomaly has lead to a hierarchy of memory where we supplement fast memories with large, cheaper, slower memories. These memory units may have very different physical and operational characteristics, therefore, making the memory system very diverse in type, cost, organization, technology and performance. This memory hierarchy will work only if the frequency of access to the slower memories is significantly less than the faster memories.

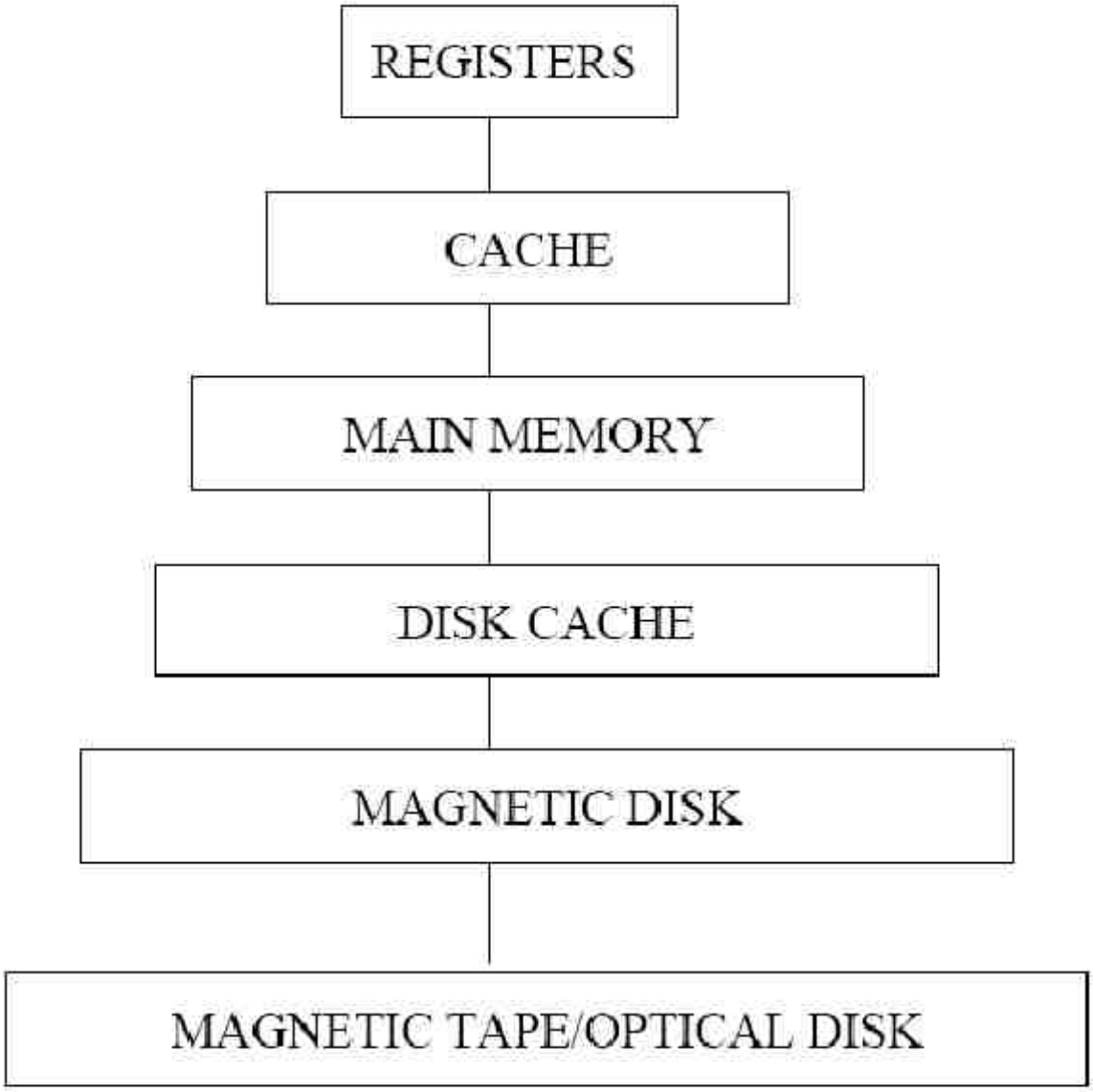


Figure 37: The memory hierarchy

You should be familiar with all the terms given in the above diagram, except the term, "disk cache". The disk cache may be a portion of RAM, sometimes called the soft disk cache that is used to speed up the access time on a disk. In some newer technologies such a memory can be a part disk drive itself; such a memory is sometimes called the hard disk cache or buffer. These hard disk caches are more effective, but they are expensive, and therefore smaller. Almost all modern disk drives include a small amount of internal cache.

3.1 Characteristics Terms for Various Memory Devices

The following terms are most commonly used for identifying the comparative behaviour of various memory devices and technologies.

Storage Capacity: It is representative of the size of the memory. The capacity of internal memory and main memory can be expressed in terms of number of words or bytes. The storage capacity of the external memory is normally measured in terms of bytes.

Unit of Transfer: A unit of transfer is the number of bits read in or out of the memory in a single read or write operation. For main memory and internal memory, the normal unit of transfer of information is equal to the word length of a processor. In fact it depends on the number of data lines in and out of the memory module. (Why?) In general, these lines are kept equal to the word size of the processor. What is a word? You have already learnt about this term in Unit 1 of this module. The unit of transfer of the external memory is normally quite large (Why? You will find the answer to this question later in this unit) and is referred to as the block of data.

Access Modes: Once we have defined the unit of transfer the next important characteristic is the access mode in which the information is accessed from the memory. A memory is considered to consist of various memory locations. The information from memory devices can be accessed in the following ways:

? ? Random Access;

? ? Sequential Access;

? ? Direct Access.

Access Time: The access time is the time required between the requests made for a read or write operation till the time the data is made available or written at the requested location. Normally it is measured for a read operation. The access time depends on the physical characteristics and access mode used for that device.

Permanence of Storage: Is it possible to lose information stored by the memory over a period of time? If yes, then what can be the reasons for the loss of information and what should be done to avoid it? There are several reasons for information destruction; these are destructive readout, dynamic storage, volatility, and hardware failure. You are familiar with all these terms. There can be some memories where the stored l looses its strength to become 0 over a period of time. These kinds of memories require refreshing. The memories, which require refreshing, are termed dynamic memories. In contrast, the memories, which do not require refreshing, are called static memories.

DRAM (Dynamic RAM)

DRAM technologies are mainly used as main memories. Presently DRAMs are available in many different forms. The basic anomaly relating to DRAMs is the cost versus speed. A faster processor needs faster memories. However, it is important to have MORE main memory rather than BETTER system memory. The performance considerations are normally associated with the performance of cache rather than the main memory.

Please note that DRAM at core is RAM. The basic difference among various acronyms of DRAM technologies are primarily because of connection of modules, configuration and addressing, or any special enhancement such as building a small portion of SRAM (Cache) in the DRAM module. A simple organisation of a DRAM chip is given in Figure 38(c).

SRAMs are mainly used as cache memories. These cache memories are discussed in more details later in this unit.

Cycle Time: It is the minimum time lapse between two consecutive read requests. Is it equal to access time? Yes, for most of the memories except the ones in which destructive readout is encountered or a refreshing cycle is needed prior to next read. Cycle time for such memories is the access time (time elapsed when a read request is made available) plus writing time as after the data has been made available, the information has to be written back in the same location as the previous value has been destroyed by reading.

Data Transfer Rate: The amount of information that can be transferred in or on the memory in a second it termed the data transfer rate or bandwidth. It is measured in bits per second. The maximum number of bits that can be transferred in a second depends on how many bits can be transferred in or out of the memory simultaneously and thus the data bus width becomes one of the controlling factors.

Physical Characteristics: In this respect the memory devices can be categorized into four main categories viz: electronic, magnetic, mechanical and optical. One of the requirements for a memory device is that it should exhibit two well-defined physical states, such that 0 and 1 can be represented in those two states. The data transfer rate of the memory depends on how quickly the state can be recognized and altered. The following table lists some of the memory technologies along with their physical and other important characteristics.

Table 3 Characteristics of some memory technologies

Technology	Access time (in Sec)	Access mode	Permanence of storage	Physical nature of storage medium	Average cost (Rs/bit) (Approx.)
Semiconductor memories	10^{-8}	Random	Volatile	Electronic	10^{-2}
Magnetic disk	10^{-2}	Direct	Non-volatile	Magnetic	10^{-5}
Magnetic tape	10^{-1}	Sequential	Non-volatile	Magnetic	10^{-6}
Compact disk ROM	1	Direct	Non-volatile	Optical	10^{-7}

Some of the main attributes that are considered for the storage devices are physical size, energy consumption and reliability. The physical size also depends on the storage density of the memories. This is also coupled with the question of portability of memory. The energy consumption is another key factor that determines the power consumption of the computer and cooling system requirements. The higher the power consumption, the costlier the equipment required for the internal cooling of the computer. Reliability is measured as mean time to failure. The storage devices, which require mechanical motion e.g., hard disks, are more prone to failure rather than the semiconductor memories, which are totally electronic in nature. Very high-speed semiconductor memories are also prone to failure as technology is moving towards its limits.

Cost: Another key factor which is of prime concern in a memory system, is cost. It is normally expressed per bit. The cost of a memory system includes not only the device but also the cost of access circuitry and peripherals essential for the operation of the memory. Table 3 also shows per bit cost of these memories. Please note that as the access time for memories is increasing, the cost is decreasing.

3.2 Cache Memories

These are small fast memories placed between the processor and the main memory (see Figure 51). Caches are faster than the main memory. Then why do we need the main memory at all?

Well once again it is the cost which determines this. The caches although are fast yet are very expensive memories and are used in only small sizes. For example, caches of sizes 128K, 256K etc. are normally used in typical Pentium based systems, whereas they can have 4 to 128 MB RAMs or even more. Thus, small cache memories are intended to provide fast speed of memory retrieval without sacrificing the size of memory (because of main memory size). If we have such a small size of fast memory how can it be advantageous in increasing the overall speed of memory references? The answer lies in the "principle of locality", which says that if a particular memory location is accessed at a time then it is highly likely that its nearby locations will be accessed in the near future.

Cache contains a copy of certain portions of main memory. The memory read or writes operation is first checked with the cache and if the desired location data is available in cache then it is used by the CPU directly. Otherwise, a block of words is read from the main memory to cache and the word is used by the CPU from cache. Since cache has limited space, for this incoming block a portion called a slot needs to be vacated in cache. The contents of this vacating block are written back to the main memory at the position it belongs to. The reason for bringing a block of words to cache is once again locality of reference. We expect that the next few addresses will be close to this address and therefore, the block of words is transferred from the main memory to cache. Thus, for the word, which is not in cache, access time is slightly more than the access time for main memory without cache. But because of locality of references, the next few words may be in the cache, thus, enhancing the overall speed of memory references. For example, if memory read cycle takes 100 ns and a cache read cycle takes 20 ns, then for four continuous references (first one brings the main memory contents to cache and next three from cache).

The time taken with cache	= (100+20) for the first read operation = 120+60 = 180	+ 20 x 3 for last three read operation
Time taken without cache	= 100x4 = 400 ns	

Thus, the closer the reference interval, the better the performance of cache and that is why a structured code is considered a good programming practice, since it provides maximum possible locality. The performance of cache is closely related to the nature of the programs

being executed; therefore, it is very difficult to say what should be the optimum size of cache, but in general a cache size between 128k to 256k is considered to be optimum for most of the cases. Another important aspect in cache is the size of the block (refer Figure 51) that is normally equivalent to 4 to 8 addressable units (which may be words, bytes etc). Figure 51 gives the cache memory organization.

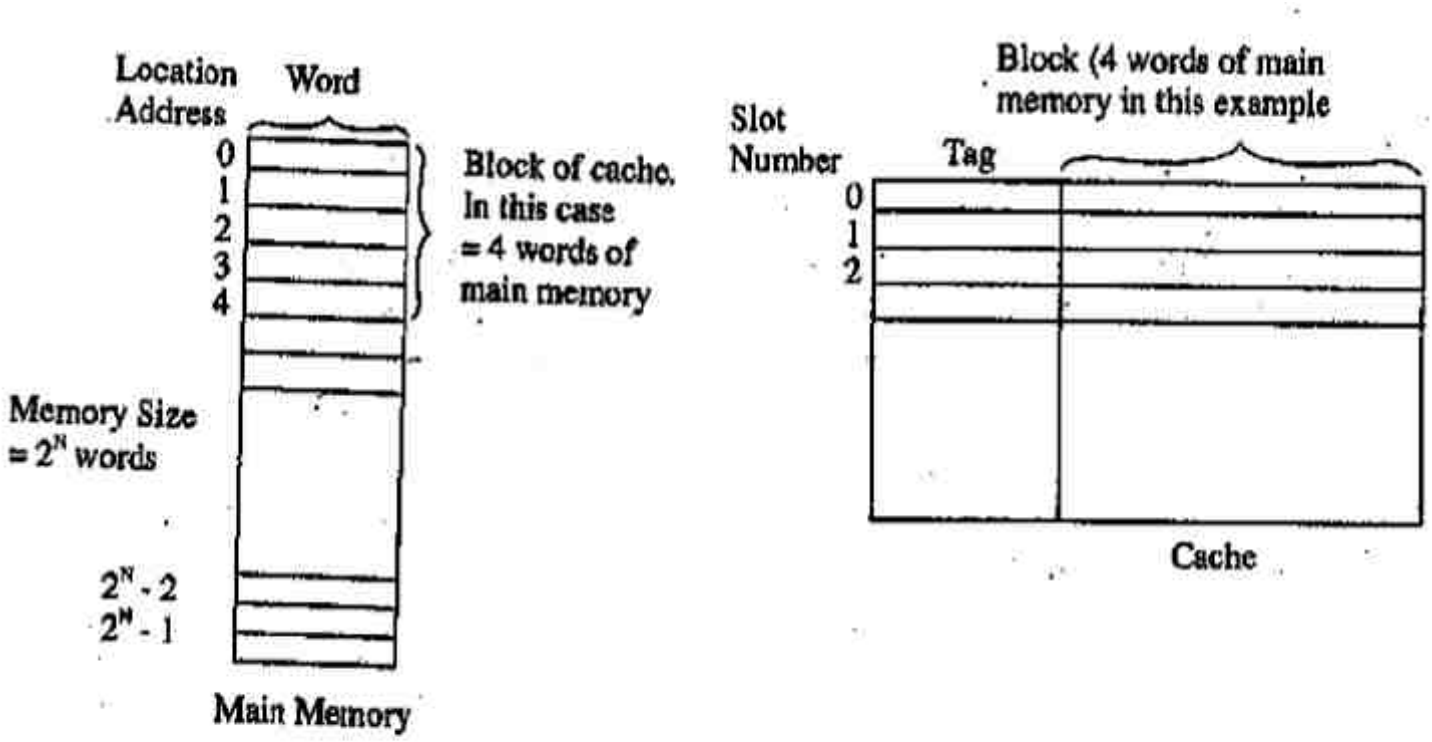


Figure 51: Cache memory organisation

A cache consists of a number of slots. As mentioned earlier, the cache size is smaller than that of the main memory; therefore, there is no possibility of one mapping of the contents of the main memory to cache.

So how will the computer identify which block is residing in cache? This is achieved by storing the block address to the tag number. In the above figure four consecutive words are put in a single block. Thus if we have 2^n words in the main memory then the total number of blocks which exist in the main memory are $=2^n/4 = 2^{n-2}$ (size of one block is 4 byte). Therefore, we need at least (n-2) bits as the size of the tag field. The important feature in cache memory is how the main memory block can be mapped in cache. There are three ways of doing so: direct, associative and set associative.

3.3 Direct Mapping: In this mapping each block of memory is mapped in a fixed slot of cache only. For example, if a cache has four slots then the main memory blocks 0 or 4 or 8 or 12 or 16…can be found in slot 0, while 1 or 5 or 9 or 13 or 17…can be found in slot 1; 2 or 6 or 10 or 14 or 18…in slot 2: and 3 or 7 or 11 15 or 19…in slot 3. This can be mathematically defined as:

Cache slot number	–	Block number of main memory	Modulo Operator	Total number of slots in cache

In this technique, it can be easily determined whether a block is in cache or not. (How?)
This technique is simple, but there is a disadvantage of this scheme. Suppose two words which are referenced alternately repeatedly are falling in the same slot then the swapping of these two blocks will take place in the cache, thus, resulting in reduced efficiency of the cache. A direct mapping example is shown in Figure 52(a). Please note the mapping of main memory address to cache tag and slot. All addresses are in hexadecimal notation.

3.4 Associative Mapping: In associative mapping any block of the memory can be mapped on to any location of the cache. But here the main difficulty is to determine "Whether a block is in cache or not." This process of determination is normally carried out simultaneously. The main disadvantage of this mapping is the complex circuitry required to examine all the cache slots in parallel to determine the presence or absence of a block in cache. An associative mapping example is shown in figure 52(b).

3.5 Set Associative Mapping: This is a compromise between the above mentioned two types of mapping. Here the advantages of both direct and associative cache can be obtained. The cache is divided in some sets, let's say t. the scheme is that a direct mapping is used to map the main memory blocks in one of the "t" sets and within this set any slot can be assigned to this block. Thus, we have associative mapping inside each set of the sets. Please refer to Figure 52(c) for an example. Another important feature for cache is the replacement algorithm. For direct mapping no algorithm is needed since only one slot can be occupied by a block in cache but in associative and set associative mapping many slots may be used by a block. So which slot should be vacated for this new block? One of the common strategies used is to vacate the least recently used slot for this new block. The reason is just the probability of accessing a block, which was used quite long ago, is less in comparison to those of blocks which are used

afterwards (or recently). This scheme is also derived from the principle of locality. Other schemes, which may not be as effective as the least frequently used, can be:

First in First Out	:	Replace the block, which has entered first in cache, and free its slot for the incoming block
Random	:	Replace any block at random
Least Frequently used	:	Replace block, which is referenced the least number of times

4.0 INPUT/OUTPUT MODULE

The input/output module (I/O module) is normally connected to the system bus of the system on one end and one or more input/output devices on the other. Normally an I/O module can control one or more peripheral devices. Is the I/O module merely a connector of input/output (I/O) devices to the system bus? No, it performs additional functions such as communicating with the CPU as well as with the peripherals. But why use the I/O module at all; why not connect peripheral devices directly to the system bus? There are three main reasons for this:

1. Diversity and variety of I/O devices makes it difficult to incorporate all the peripheral devices logic (i.e. its control commands, data format etc.) into the CPU. This in turn will also reduce the flexibility of using any new development.
2. The I/O devices are normally slower than that of the memory and the CPU, therefore, do not use them on high-speed bus directly for communication purposes.
3. The data format and word length used by the peripheral may be quite different than that of the CPU.

4.1 Functions of the I/O Module

An I/O module is a mediator between the processor and the I/O devices. It controls the data exchange between the external devices and the main memory; or external devices and CPU registers. Therefore, an I/O module provides an interface internal to the computer which connects it to the CPU and the main memory and an interface external to the computer,

connecting it to an external device or peripheral. The I/O module should not only communicate the information from the CPU to the I/O device, but it should also coordinate these two. In addition, since there are speed differences between the CPU and I/O devices, the I/O module should have facilities like buffer (storage area) and error detection mechanisms. Therefore, the functional requirements of an I/O module are:

1. It should be able to provide control and timing signals

The need of I/O from various I/O devices by the CPU is quite unpredictable. In fact it depends on I/O needs of particular programs and normally does not follow any pattern, since the I/O module also shares system bus and memory for data input/output. Therefore, control and timing are needed to coordinate the flow of data from/to external devices to/from CPU or memory. A typical control cycle for transfer of data from I/O device to CPU is this:

? ? Enquire the status of an attached device from I/O module. The status can be busy, ready or out of order.

? ? I/O module responds with the status of the device.

? ? If the device is ready, CPU commands I/O module to transfer data from the I/O device.

? ? I/O module accepts data from the I/O device.

? ? The data is then transferred from I/O module to the CPU.

2. It should communicate with the CPU

The example given above clearly specifies the need of communication between the CPU and I/O module. This communication can be:

commands such as READ SECTOR, WRITE SECTOR, SEEK track number (which are issued by the CPU to the I/O module); or data (which may be required by the CPU or transferred out); or status information such as BUSY or READY or any error condition from I/O modules. The status recognition is necessary because of the speed gap between the CPU and I/O device. An I/O device might be busy in doing the I/O of previous instructions when it is needed again.

Another important communication from the CPU is the unique address of the peripheral from which I/O is expected or is to be controlled.

3. It should communicate with the I/O device

Communication between the I/O module and the I/O device is needed to complete the I/O operation. This communication involves commands, status or data.

4. It should have a provision for data buffering

Data buffering is quite useful for the purpose of smoothing out the gaps in speed of CPU and I/O devices. The data buffers are registers, which hold the I/O information temporarily. The I/O is performed in short bursts in which data is stored in buffer a area while the device can take its own time to accept it. In I/O device to CPU transfer, data is first transferred to the buffer and then passed on to the CPU from these buffer registers. Thus, the I/O operation does not tie up the bus for the slower I/O devices.

5. Error detection mechanisms should be in-built

The error detection mechanisms may involve checking the mechanical as well as data communication errors. These errors should be reported to the CPU. Examples of the kind of mechanical errors that can occur in devices are paper jam in printer, mechanical failure, electrical failure, etc. data communication errors may be checked by using parity bit.

4.2 Structure of I/O Module

Let us now focus our attention on the question. What should be the structure of an I/O module? Although, there is no standard structure of the I/O module, let us try to visualize certain general characteristics of the structure.

? ? There is a need for I/O logic, which should interpret and execute the dialogue between the CPU and I/O module. Therefore, there need to be control lines between the CPU and this I/O module. In addition, the address lines are needed to recognize the address of the I/O module and its specific device.

? ? The data lines connecting the I/O module to the system bus must exist. These lines serve the purpose of data transfer.

? ? Data registers may act as buffer between the CPU and the I/O module.

? ? The interface of the I/O module with the device should have interface logic to control the device, to get the status information and transfer of data.

Figure 54 shows the diagram of a typical I/O module which in addition to all the above have status/control registers which might be used to pass on the status information or can store control information.

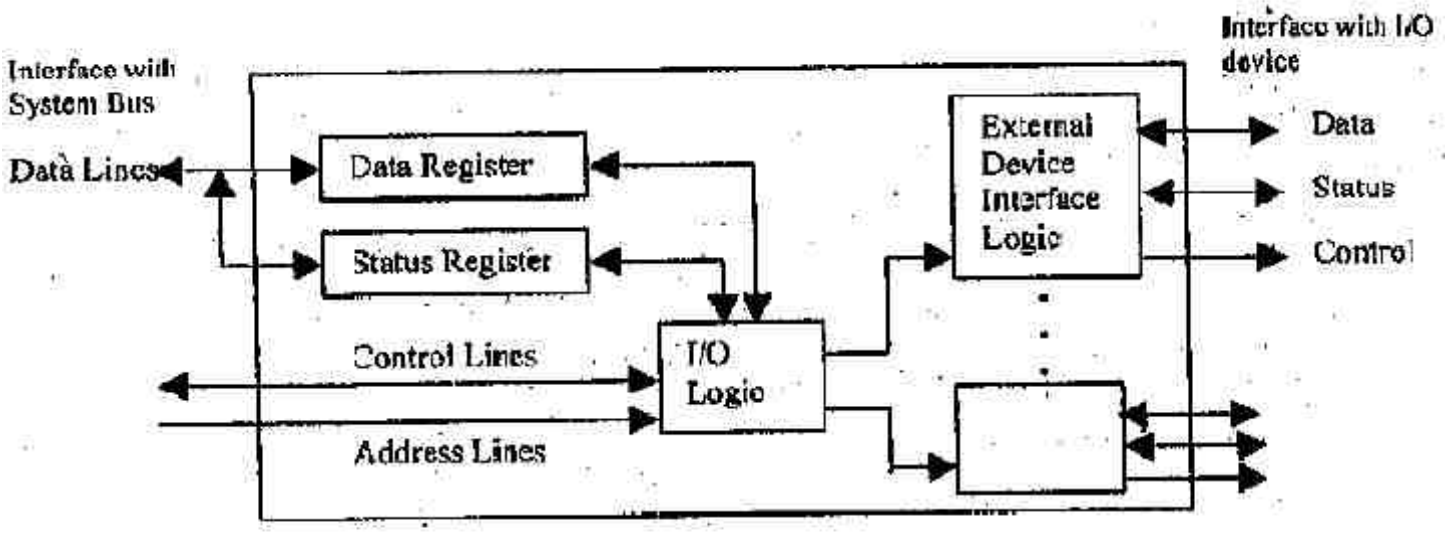


Figure 54: The general structure of the I/O Module

The data is stored or buffered in data registers. The status registers are used to indicate the current state of a device, e.g., the device is busy, the system BUS is busy, the device is out of order etc. If an I/O module takes more processing burden then it is called an I/O channel or a processor. The primitive I/O module which requires detailed control by the processor is called the I/O controller or device controller. Theses I/O controllers are normally used in micro-computers while I/O processors are mainly used for mainframes because the I/O work for the microcomputer is normally limited to single user's job, therefore, we do not expect a very huge amount of I/O to justify the investment in I/O processors, which are expensive. Once we have defined a general structure for an I/O module, the next question is how actually are the I/O operations performed? The next section tries to answer this basic query in a general way.

4.3 Input/Output Techniques

The input/output operations can be performed by three basic techniques. These are:

· Programmed input/output

· Interrupt-driven input/output

· Direct memory access

Figure 55 gives an overview of these three techniques.

	Interrupt Required	I/O Module to/from Memory Transfer
Programmed I/O	No	Through CPU
interrupt-driven I/O	Yes	Through CPU
DMA	Yes	Direct to memory

Figure 55: An overview of the three input/out techniques

In programmed I/O, the I/O operations are completely controlled by the CPU. The CPU executes programs that initiate, direct and terminate an I/O operation. It requires a little special I/O hardware, but is quite time consuming for the CPU, since the CPU has to wait for slower I/O operations.

Another technique suggested to reduce the waiting by the CPU is interrupt-driven I/O. The CPU issues the I/O command to I/O module and starts doing other work, which may be the execution of a separate program. When the I/O operation is complete, the I/O module interrupts the CPU by informing it that I/O has finished. The CPU then, may proceed with execution of this program. In both programmed I/O and interrupt-driven I/O, the CPU is responsible for extracting data from the memory for output and storing data in the memory for input. Such a requirement does not exist in DMA where the memory can be accessed directly by the I/O module. Thus, the I/O module can store or extract data in/from the memory. We will discuss programmed I/O and interrupt-driven I/O in this section and DMA in the next section.

A situation in which the I/O is solely looked after by a separate dedicated processor is referred to as I/O channel or I/O processor. The basic advantage of these devices is that they free the CPU of the burden of input/output. Thus, during this time, the CPU can do other work, therefore, effectively increasing the CPU utilization. We will discuss I/O channels and I/O processors in the next section.

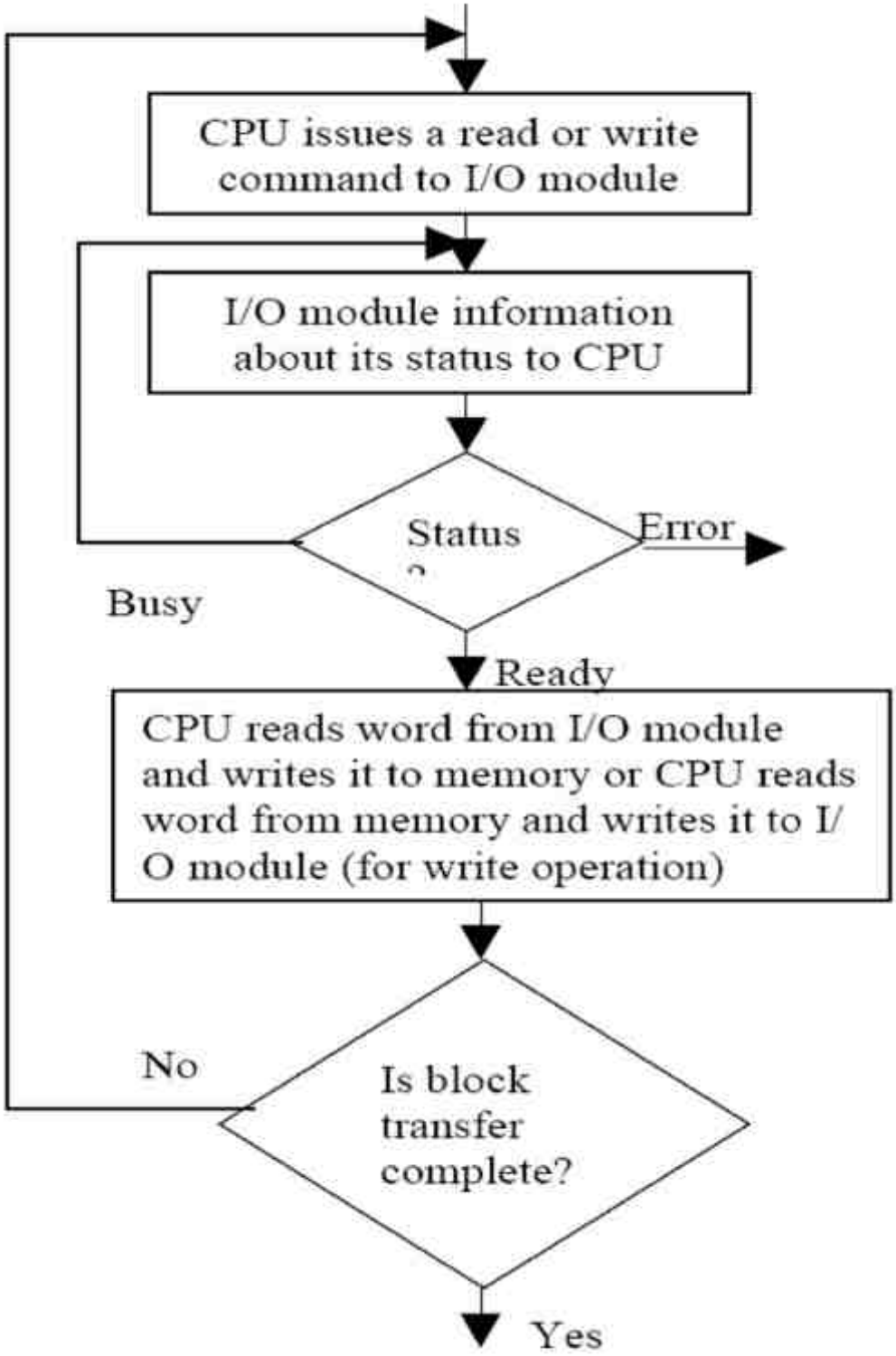
4.4 Programmed Input/Output

Programmed input/output is a useful I/O method for computers where hardware costs need to be minimized. The input or output in such cases may involve:

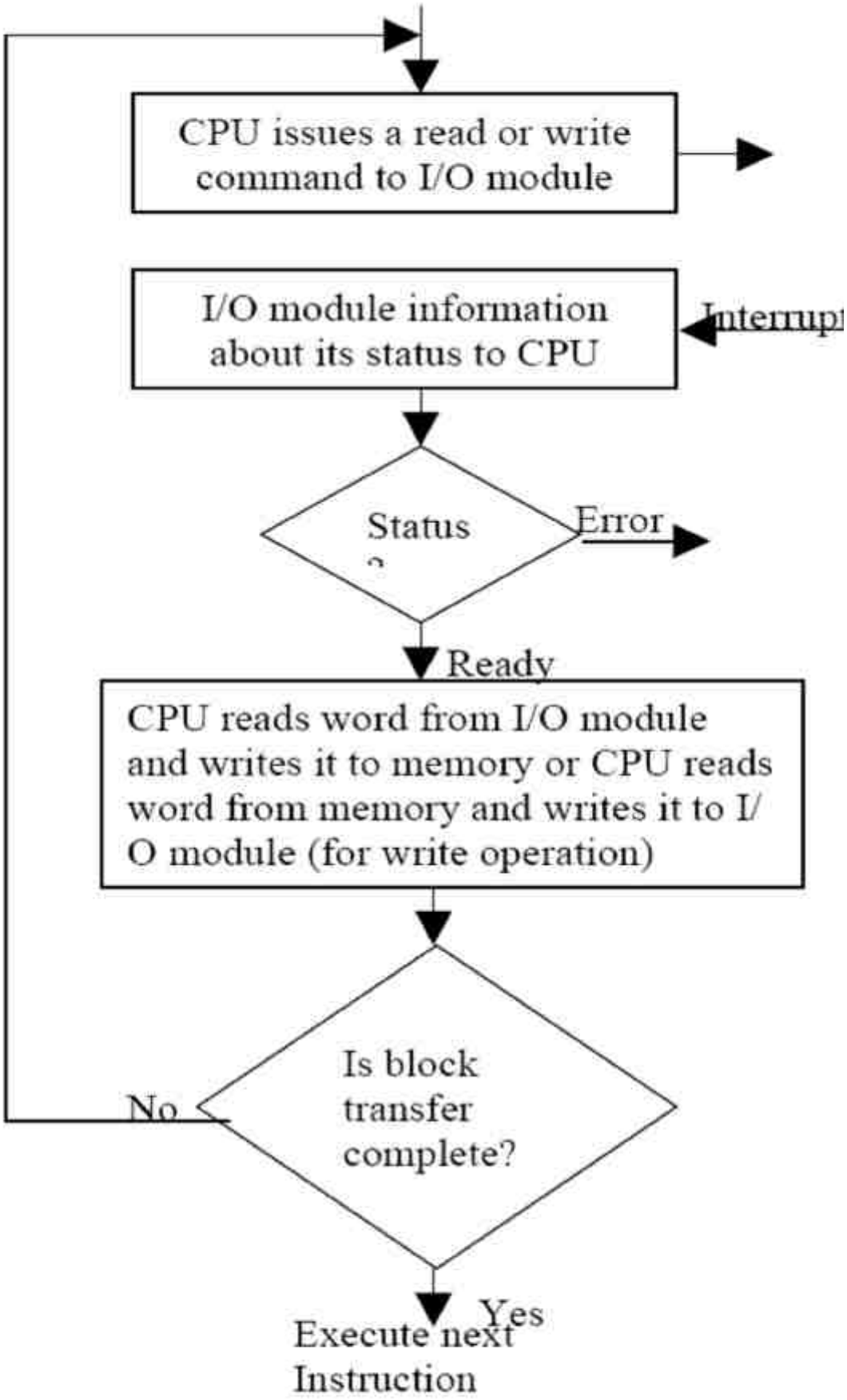
· Transfer of data from I/O device to the CPU registers.

· Transfer of data from CPU registers to memory.

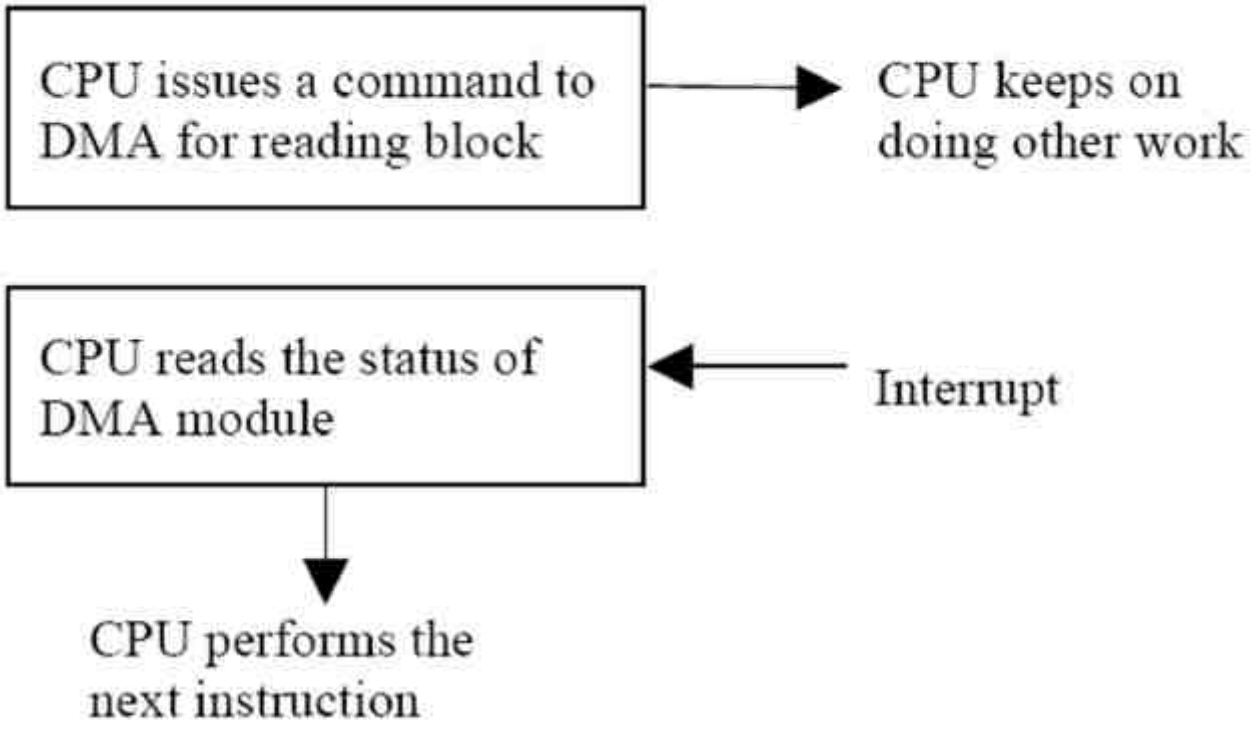
In addition, in a programmed I/O method the responsibility of the CPU is to constantly check the status of the I/O device to check whether it has become free (in case output is desired) or it has finished inputting the current series of data (in case input is going on). Thus, programmed I/O is a very time consuming method where the CPU wastes lot of time for checking and verifying the status of an I/O device. Let us now try to focus how this input-output is performed. Figure 56(a) gives the block diagram of transferring a block of data word by word using programmed I/O technique.


CPU issues a read or write command to I/O module
I/O module information about its status to CPU
Status
Error
Busy
Ready
CPU reads word from I/O module and writes it to memory or CPU reads word from memory and writes it to I/ O module (for write operation)
No
Is block transfer complete?
Yes
Execute next instruction

(a) program I/O



(b) interrupt-driven I/O



(c) DMA

Figure 56: Three techniques of input/output

I/O Instructions: To carry out input/output CPU issues I/O related instructions. These instructions consist of two components:

· The address of the input/output device specifying the I/O device and I/O module; and

· An in put/output command.

There are four types of I/O Commands, which can be classified as:

CONTROL, TEST, READ and WRITE.

CONTROL commands are specific devices and are used to control the specific instructions to the device; e.g., a magnetic tape requires rewinding or moving forward by a block. TEST command checks the status such as, if a device is ready or not or is in error condition.

The READ command is used for input of data from input device and the WRITE command is used for output of data to input device.

The other part of I/O instruction is the address of the I/O device. In systems with programmed I/O the I/O module, the main memory and the CPU normally share the system bus. Thus, each I/O module should interpret the address lines to determine if the command is for itself. Or in other words: How does CPU specify which device to access? There are two methods of doing so. These are called memory mapped I/O and I/O-mapped I/O.

If we use the single address space for memory locations and I/O devices, i.e., the CPU treats the status and data registers of the I/O module as memory locations, and then memory and I/O devices can be accessed using the same instructions. This is referred to as memory mapped I/O.

For a memory mapped I/O, only a single READ and a single WRITE line are needed for memory or I/O module read or write operations. These lines are activated by the CPU for either memory access or I/O device access.

4.5 Interrupt-driven Input/Output

What is the basic drawback of the programmed I/O? The speed of I/O devices is much slower in comparison to that of the CPU and because the CPU has to repeatedly check whether a device is free; or wait till the completion of I/O, the performance of CPU in programmed I/O goes down tremendously. What is the solution? What about the CPU going back to do other useful work without waiting for the I/O device to complete (what it is currently doing) or get freed up? But how will the CPU be intimated about the completion of I/O or that a device is ready for I/O? A well-designed mechanism was conceived for this, which is referred to as interrupt-driven I/O. In this mechanism, provision of interruption of CPU work, once I/O device has finished the I/O or when it is ready for the I/O, has been provided.

The interrupt-driven I/O mechanism for transferring a block of data is shown in Figure 56(b). Please note that after issuing a READ command (for input) the CPU goes off to do other useful work (it may be the execution of a difference program) while I/O module proceeds for reading of data from the associated device. At the completion of an instruction cycle (already discussed in Unit 1 of this module) the CPU checks for interrupts (which will occur when data is in data register of I/O module and it now needs the CPU's attention).

Now the CPU saves the important register and processor status of the executing program in a stack and requests the I/O device to provide its data, which is placed on the data bus by the I/O device. After taking the required action with the data, the CPU can go back to the program it was executing before the interrupt.

Interrupt: As discussed in Unit 1, the term interrupt is loosely used for any exceptional event that causes a temporary transfer of control of the CPU from one program to the other which is causing the interrupt. Interrupts are primarily issued on:

· the initiation of an input/output operation

· the completion of an input/output operation

· the occurrence of hardware or software errors.

An interrupt can be generated by various sources, internal and external to the CPU. An interrupt generated internally by the CPU is sometimes termed “Traps”. The traps are normally results of programming errors (such as division by zero) which occur during the execution of a program.

The two key issues in interrupt-driven input/output are:

· to determine the device which has issued an interrupt

· in the case of the occurrence of multiple interrupts, which one to be processed first.

There are several solutions to these problems. The simplest of them is to provide multiple interrupt lines, which will result in the immediate recognition of the interrupting device. The priorities can be assigned to various interrupts and the interrupt with the highest priority should be selected for service in case multiple interrupt occurs. But providing multiple interrupt lines is an impractical approach because only a few lines of the system bus can be devoted for the interrupt. Other methods for this are software poll, daisy chaining, etc.

Software Poll: In this scheme, on the occurrence of an interrupt, the CPU starts executing a software routine known as interrupt service program or routine which polls to each I/O module to determine which I/O module has caused the interrupt. This may be achieved by reading the status register of the I/O modules. The priority here can be implemented easily by defining the polling sequence, since the device polled first will have higher priority. Please note that after identifying the device, the next set of instructions to be executed will be the device service routines of that device, resulting in the desired input or output.

As far as **daisy chaining** is concerned, we have one interrupt acknowledge line, which is chained through various interrupt devices. (The mechanism is similar, as we have discussed in Unit 2). There is just one interrupt request line. On receiving an interrupt request, the interrupt acknowledge line is activated which in turn passes this signal device by device. The first device which has made the interrupt request thus grasps the signal and responds by putting a word which is normally an address of the interrupt servicing program or a unique identifier on the data lines. This word is also referred to as interrupt vector. This address or identifier in turn is used for selecting an appropriate interrupt servicing program. The daisy chaining has an in-built priority scheme, which is determined by the sequence of devices on the interrupt acknowledge line.

In bus arbitration technique, the I/O module first needs to control the bus and only after that can it request for an interrupt. In this scheme, since only one of the modules can control the bus, only one request can be made at a time. The interrupt request is acknowledge by the CPU in response to which the I/O module places the interrupt vector on the data lines. An interrupt vector normally contains the address of the interrupt servicing program. You can refer to further readings for more details on some typical interrupt structures of interrupt controllers.

4.6 Direct Memory Access (DMA)

When a large amount of data is to be transferred from the CPU, a DMA module can be used. But why? In both interrupt-driven and programmed I/O the CPU is tied up in executing input/output instructions while DMA acts as if it has taken over control from the CPU. The DMA operates in the following way:

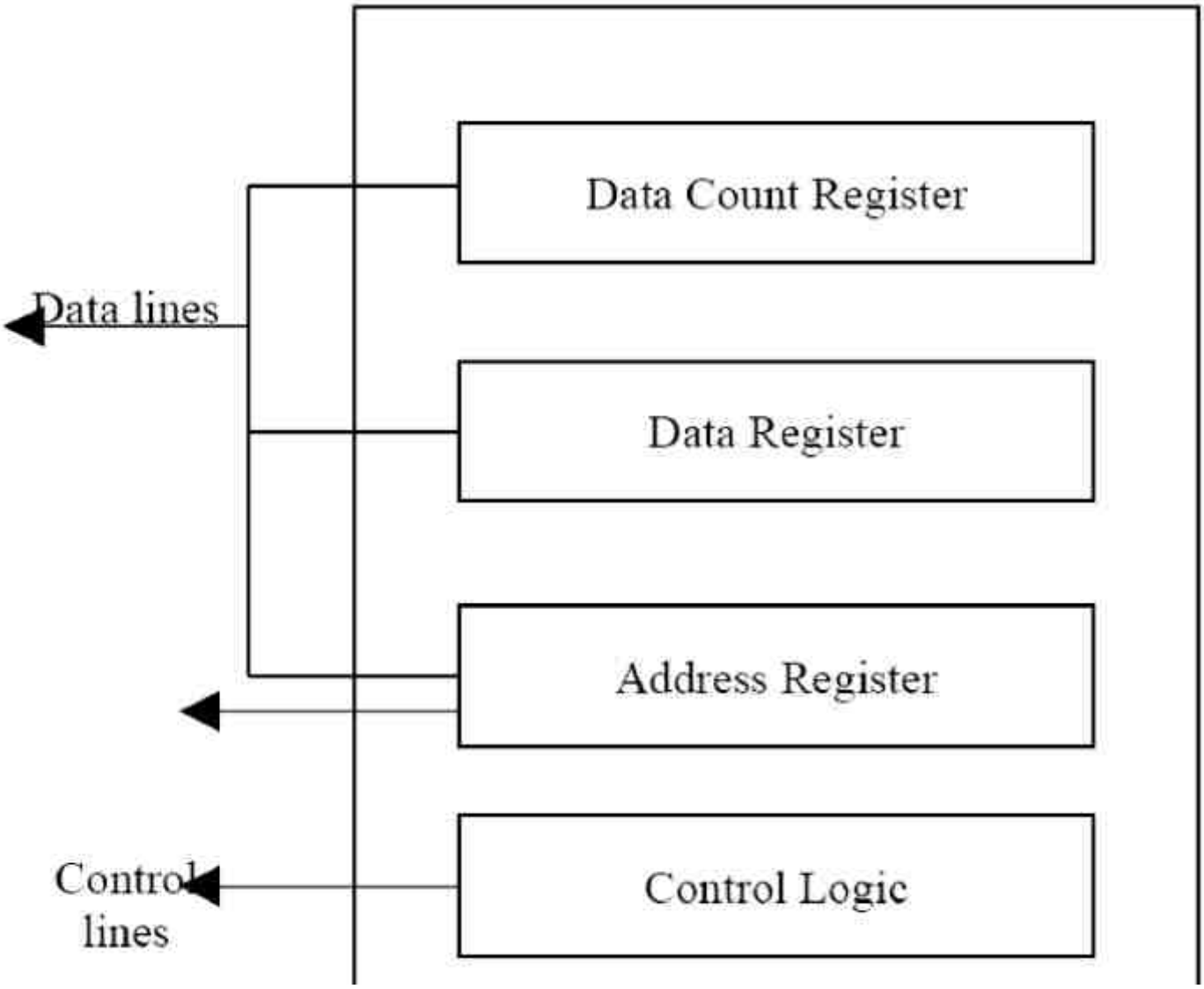
When an I/O is requested, the CPU instructs the DMA module about the operation by providing the information:

· Which operation to be performed (read or write).

· The address of the I/O device which is to be used.

· The starting location on the memory where the information will be read or written to the number of words to be written or to be read.

The DMA module transfers the requested block byte by byte directly to the memory without intervening the CPU. On completion of the request the DMA module sends an interrupt signal to the CPU. Thus, in DMA the CPU involvement can be restricted at the beginning and end of the transfer. But what does the CPU do while the DMA is doing input/output? It may execute another program or it may be another part of the same program. Figure 59 shows registers of a general DMA module. Please note that it contains additional registers for counting the data bytes and also note that address register and data count registers are fed with the data lines.


Data Count Register
Data lines
Data Register
Address Register
Control Logic
Control lines

5.0 INSTRUCTION SET CHARACTERISTICS

Till now, we have discussed instruction in an abstract way. Now let us discuss in detail various characteristics of instructions. However we will first of all find out the significance of the instructions set. One thing which should be kept in mind is that the instruction set is a boundary which is looked upon in the same fashion by a computer designer and the programmer. From the computer designer's point of view, the instruction set provides the functional requirements of the CPU. In fact, for implementing the CPU design, one of the main tasks is to implement the instruction set for that CPU. However, from the user's point of view machine instructions or assembly instructions are needed for low level programming. In addition, a user should also be aware of registers, the data types supported by the machine and the functioning of the ALU.

We will discuss the registers in Unit 2 and the ALU in Unit 3 of this module.

Explanation on the machine instruction set gives extensive details about the CPU of a machine. In fact, the operations which a CPU can perform can be determined by the machine instructions. Now, let us answer some introductory questions about the instruction set.

5.1 What is an instruction set?

An **instruction set** is a collection of all the *instructions* a CPU can execute. Therefore, if we define all the instructions of a computer, we can say we have defined the instruction set. Each instruction consists of several elements. An instruction element is a unit of information required by the CPU for execution.

What are the elements of an instruction?

· An operation code, also termed an **opcode** which specifies the operation to be performed

· A reference to the operands on which data processing is to be performed. For example, an address of an operand

· A reference to the operands which may store the results of data processing operations performed by the instruction.

· A reference for the next instruction, to be fetched and executed. The next instruction which is to be executed is normally the next instruction following the current instruction in the memory. Therefore, no explicit reference to the next instruction is provided. What if we do not want this normal flow of execution? You will find the answer to this question in this unit.

An important aspect, of the references to operands and results is: where are those operands located? In the memory or in the CPU registers or in the I/O device. If the operands are located in the registers then an instruction can be executed faster than that of the operands located in the memory. The main reason here is that memory access time is higher in comparison to the register access time.

5.2 How is an instruction represented?

Instructions are represented as sequence of bits. An instruction is divided into a number of fields. Each of these fields corresponds to a constituent element of instruction. A layout of instruction is known as **instruction format**. For example, the following is the instruction format for an IAS computer. It uses four bits for **opcode** and only two operand references are provided here. No explicit reference is provided for the next instruction to be executed.

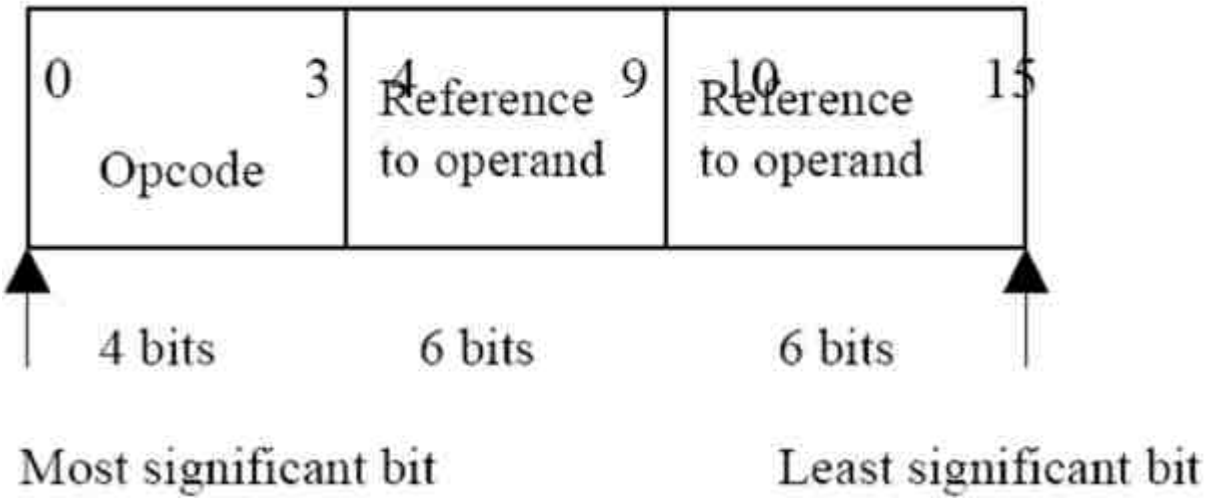


Figure63: A sample instruction format

In most instruction sets, many instruction formats are used. An instruction is first read into an instruction register (IR), and then the CPU decodes the instruction and extracts the required operands on the basis of references made through the instruction fields, and processes it.

Since the binary representation of the instruction is difficult to comprehend and is seldom used for representations, we will be using symbolic representations to these instructions in this unit along with the comments wherever desired.

5.3 What are the types of instructions?

The instructions can be categorized under the following:

· **Data Processing Instructions**: These instructions are used for arithmetical and logic operations in a machine. Examples of data processing instructions are: Arithmetic, Boolean, shift, character and string processing instructions, stack and register, manipulation instructions, vector instructions, etc.

· **Data Storage/Retrieval Instructions**: Since the data processing operations are normally performed on the data stored in CPU registers, we need instructions to bring data to and from memory to registers. These are called data storage/retrieval instructions.

Examples of data storage and retrieval instructions are load and store instructions.

· **Data Movement Instructions**: These are basically input/output instructions. They are required to bring in programs and data from various devices to memory or to communicate the results to the input/output devices. Some of these instructions can be: start input/output, halt input/output, TEST input/output etc.

· **Control Instructions**: These instructions are used for testing the status of computation through Processor Status Word (PSW). Another of such instruction is the branch instruction used for transfer of control. We will discuss in more details about these instructions.

· **Miscellaneous Instructions**: These instructions do not fit in any of the above categories. Some of these instructions are: interrupt or supervisory call, swapping, return from interrupt, halt instruction or some privileged instruction of operating systems.

What are the factors which play important roles for selection/designing of instruction sets for a machine?

Instruction set design is the most complex yet interesting and very much analyzed aspect of computer design. The instruction set plays an important role in the design of the CPU as it defines many functions of it. Since instruction sets are the means by which a programmer can control the CPU, therefore, users' views must be considered while designing the instruction set. Some of the important design issues relating to instruction design are:

· How many and what operations should be provided?

· What are the operand data types to be provided?

· What should be the instruction format? This includes issues like: instruction length, number of address, length of various elements of instructions, etc.

· What is the number of registers which can be referenced by an instruction and how are they used?

· What are the modes of specifying an operand address? We will try to analyze these issues in some detail in this and subsequent sections. However, we have kept the level of discussion very general. A special example of an instruction set is given at the end of this unit.

5.4 Operand Data Types

An operand data type specifies the type of data on which a particular operation can be performed. For example, for an arithmetical operation, numbers are to be used as data types. In general the operands which can be used in an instruction can be categorized into four general categories. These are:

· Addresses

· Numbers

· Characters

· Logical data

Addresses: Addresses are treated as a form of data which is used in the calculation of actual physical memory address of an operand. In most of the cases, the addresses provided in instruction are operand references and not the actual physical memory addresses.

Numbers: All machines provide numeric data types. One special feature of numbers used in computers is that they are limited in magnitude, and hence the underflow and overflow may occur during arithmetical operations on these numbers. The maximum and minimum magnitude is fixed for an integer number while a limit of precision of numbers and exponent exist in the floating point numbers. The three numeric data types which are common in computers are:

- Fixed point numbers or Integers (signed or unsigned)
- Floating point numbers
- Decimal numbers

All the machines provide instructions for performing arithmetical operations on fixed point and floating point numbers. Many machines provide arithmetical instructions which perform operations on packed decimal digits.

Characters: Another very common data type is the character or string of characters. The most widely used character representation is ASCII (American National Standard Code of Information Interchange). It has 7 bits for coding data pattern which implies 128 different characters. Some of these characters are control characters which may be used in data communication. The eighth bit of ASCII may be used as a parity bit. One special mention about ASCII which facilitates the conversion of a 7 bit ASCII and a 4 bit packed decimal number is that the last four digits of ASCII number are binary equivalents of digits 0-9. with packed decimal in a similar way as that of ASCII. The digits 0 through 9 in this can be represented as 1111 0000 through 1111 1001.

Logical Data: In general a data word or any other addressable unit such as byte, half word etc. are treated as a single unit of data. But can we consider an n-bit data unit consisting of n items of 1 bit each? If we treat each bit of an n-bit data as an item then it can be considered to be logical data. Each of these n items can have a value 0 or 1.

What are the advantages of such a bit oriented view of data? The advantages of such a view will be:

· We can store an array of Boolean or binary data items most efficiently.

· We will be in a position to manipulate the bits of any data item.

But where do we need to manipulate bits of a data item? The example of such a case is shifting of significance bits in a floating point operation or for converting ASCII to packed decimals where we need only the 4 rights most bits of ASCII's byte. Please note that for extracting decimal from ASCII, first the data is treated as logical data and then it can be used in arithmetical operations as numeric data. Thus, the operation performed on a unit of data determines the types of the unit of data at that instance. This statement may be true for high level language, but holds good for machine level language.

5.5 Number of Addresses in an Instruction

The fewer number of addresses in an instruction lead to reduced length of instructions; however, it also limits the range of functions that can be performed by the instructions. In a

sense this implies that a machine instruction set having less number of addresses have longer programs, which means longer execution time. However, more addresses may lead to more complex decoding and processing circuits. Most of the instructions do not require more than three operand addresses. In instructions having fewer addresses than three, normally some of the operand locations are implicitly defined. Many computers have a range of instructions of different lengths and number of addresses. The following table gives an example of zero, one, two and three address instruction along with their interpretations.

Number of Addresses	Instruction	Interpretation
3	ADD A, B, C	Operation A=B+C is executed
2	ADD A,B	Two plausible interpretations (i) AC = A + B (ii) A = A + B. In this case the original content of operand location A is lost
1	ADD A	AC = AC + A A is added to accumulator
0	ADD	Top of stack contains the addition to top two values of the stack.

- AC is an accumulator register.
  A, B, C are operand locations.

Figure 65: Examples of zero, one, two and three address instructions

The register architecture, that is a general classification which is based on register set of the computer, is sometimes classified according to the number of addresses in instructions. These classifications are:

Evaluation-Stack Architecture: These machines are Zero address machines and their operands are taken from top of the stack implicitly. The ALU of such machines directly

references a stack which can be implemented in main memory or registers or both. These machines contain instructions like PUSH and POP to load a value on stack and store a value in the memory respectively. Please note that PUSH, POP are not zero address instructions but contain one address. The main advantages of such architecture are:

· Very short instructions

· Since stacks are normally made within CPU in such an architecture machine, the operands are close to the ALU, thus effecting fast execution of instructions

· Excellent support for subroutines.

While the main disadvantages of this architecture are:

· Not very general in nature, in comparison to other architectures

· Difficult to program for applications like text and string processing.

One example of such a machine is Burroughs B6700. However, these machines are not very common today because of the general nature of machines desired. The stack machine uses Polish notations for evaluation of arithmetical expressions. Polish notation was introduced by a Polish logician, Jan Lukasiewicz. The main theme of this notation is that an arithmetical expression AxB can be expressed as:

either	xAB	(Prefix notation)
or	ABx	(Suffix or reverse polish notation)

In stacks we use suffix or reverse polish notation for evaluating expression. The rule here is to push all the variable values on the top of stack and do the operation with the top elements of stack as an operand is encountered. The priority of operand is kept in mind for generation of reverse polish notation. For example, an expression AxB+CxD/E will be represented in reverse polish notation as:

ABxCDxE/+

Thus, the execution program for evaluation of F=AxB+CxD/E will be:

PUSH A /Transfer the value of A on to th e top of stack/

PUSH B /Transfer the value of B on to the top of stack/

MULT /Multiply. It will remove the value of A and B from the stack and multiply A x B/

PUSH C /Transfer C on the top of stack/

PUSH D /Transfer D on the top of stack/

MULT /Remove the values of C & D from the stack and multiply

A x D/

PUSH E /Transfer E on the top of stack/

DIV /C x D/E /

ADD /Add the top two values on the stack/

POP F /Transfer the results back to memory location F/

Figure 66: Sample program for evaluating AxB+CxD/E using zero address instructions.

Please note that PUSH and POP are not zero-address instructions.

The execution of the above program using stack is represented diagrammatically in figure 67.

```
LOAD    A    /Transfer A to Accumulator/
MULT    B    /Multiply Accumulator with B, keep the result of/
             /multiplication in B/
STORE   T    /Store the intermediate result temporarily in a
             location T/
LOAD    C    /Transfer C to Accumulator/
MULT    D    /Accumulator = Accumulator X D/
DIV     E    /Accumulator = Accumulator/E /
ADD     T    /Accumulator = Accumulator + T/
STORE   F    /Store the Accumulator value to location F/
```

Figure 68: Sample program for evaluating AxBxCxD/E using one address instruction.

6.0 PIPELINING DESIGN TECHNIQUES

There exist two basic techniques to increase the instruction execution rate of a processor.

These are to increase the clock rate, thus decreasing the instruction execution time, or alternatively to increase the number of instructions that can be executed simultaneously. Pipelining and instruction-level parallelism are examples of the latter technique. Pipelining owes its origin to car assembly lines. The idea is to have more than one instruction being processed by the processor at the same time. Similar to the assembly line, the success of a pipeline depends upon dividing the execution of an instruction among a number of subunits (stages), each performing part of the required operations. A possible division is to consider instruction fetch (F), instruction decode (D), operand fetch (F), instruction execution (E), and store of results (S) as the subtasks needed for the execution of an instruction.

In this case, it is possible to have up to five instructions in the pipeline at the same time, thus reducing instruction execution latency.

6.1. General Concepts

Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by a given functional unit. The units are connected in a serial fashion and all of them operate simultaneously. The use of pipelining improves the performance compared to the traditional sequential execution of tasks. Figure 9.1 shows an illustration of the basic difference between executing four subtasks of a given instruction (in this case fetching F, decoding D, execution E, and writing the results W) using pipelining and sequential processing.

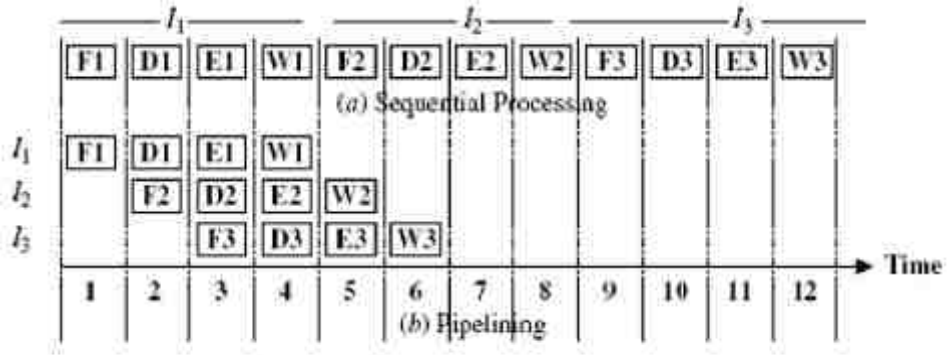


Figure 9.1 Pipelining versus sequential processing

It is clear from the figure that the total time required to process three instructions (I1, I2, I3) is only six time units if four-stage pipelining is used as compared to 12 time units if sequential

processing is used. A possible saving of up to 50% in the execution time of these three instructions is obtained. In order to formulate some performance measures for the goodness of a pipeline in processing a series of tasks, a space time chart (called the Gantt's chart) is used. The chart shows the succession of the subtasks in the pipe with respect to time. Figure 9.2 shows a Gantt's chart. In this chart, the vertical axis represents the subunits (four in this case) and the horizontal axis represents time (measured in terms of the time unit required for each unit to perform its task). In developing the Gantt's chart, we assume that the time (T) taken by each subunit to perform its task is the same; we call this the unit time.

As can be seen from the figure, 13 time units are needed to finish executing 10 instructions (I1 to I10). This is to be compared to 40 time units if sequential processing is used (ten instructions each requiring four time units).

In the following analysis, we provide three performance measures for the goodness of a pipeline. These are the Speed-up S(n), Throughput U(n), and Efficiency E(n). It should be noted that in this analysis we assume that the unit time T = t units.

1. *Speed-up* $S(n)$ Consider the execution of m tasks (instructions) using n-stages (units) pipeline. As can be seen, $n + m - 1$ time units are required

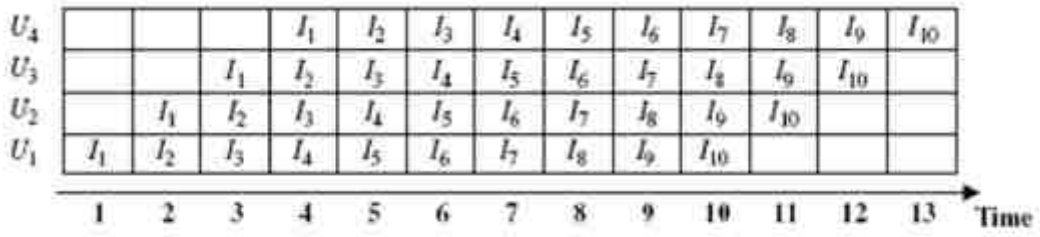


Figure 9.2 The space–time chart (Gantt chart)

to complete m tasks.

$$\text{Speed-up } S(n) = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}} = \frac{m \times n \times t}{(n+m-1) \times t}$$

$$= \frac{m \times n}{n+m-1}$$

$$\lim_{m \to \infty} S(n) = n \quad \text{(i.e., } n\text{-fold increase in speed is theoretically possible)}$$

2. *Throughput* $U(n)$

$$\text{Throughput } U(n) = \text{no. of tasks executed per unit time} = \frac{m}{(n+m-1) \times t}$$

$$\lim_{m \to \infty} U(n) = 1 \text{ assuming that } t = 1 \text{ unit time}$$

3. *Efficiency* $E(n)$

$$\text{Efficiency } E(n) = \text{Ratio of the actual speed-up to the maximum speed-up}$$

$$= \frac{\text{Speed-up}}{n} = \frac{m}{n+m-1}$$

$$\lim_{m \to \infty} E(n) = 1$$

6.2. Instruction Pipeline

The simple analysis made in figure 9.1 ignores an important aspect that can affect the performance of a pipeline, that is, pipeline stall. A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units. Figure 9.3 illustrates the effect of having instruction I2 incurring a cache miss (assuming the execution of ten instructions I1 to I10).

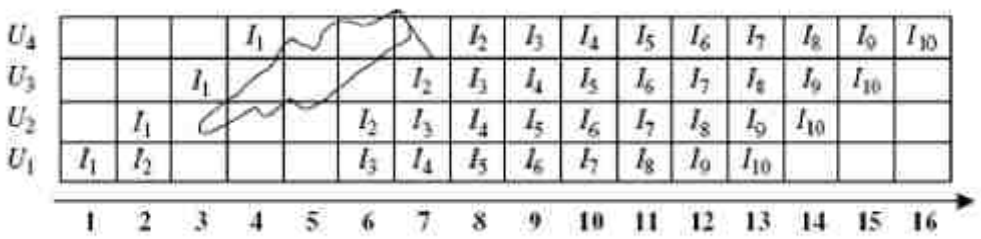


Figure 9.3 Effect of a cache miss on the pipeline

The figure shows that due to the extra time units needed for instruction I2 to be fetched, the pipeline stalls, that is, fetching of instruction I3 and subsequent instructions are delayed. Such situations create what is known as pipeline bubble (or pipeline hazards). The creation of a pipeline bubble leads to wasted unit times, thus leading to an overall increase in the number of time units needed to finish executing a given number of instructions. The number of time units needed to execute the 10 instructions shown in Figure 9.3 is now 16 time units, compared to 13 time units if there were no cache misses.

Pipeline hazards can take place for a number of other reasons. Among these are instruction dependency and data dependency. These are explained below. 9.2.1. Pipeline "Stall" Due to Instruction Dependency Correct operation of a pipeline requires that operation performed by a stage MUST NOT depend on the operation(s) performed by other stage(s). Instruction dependency refers to the case whereby fetching of an instruction depends on the results of executing a previous instruction. Instruction dependency manifests itself in the execution of a conditional branch instruction. Consider, for example, the case of a "branch if negative" instruction. In this case, the next instruction to fetch will not be known until the result of executing that "branch if negative" instruction is known. In the following discussion, we will assume that the instruction following a conditional branch instruction is not fetched until the result of executing the branch instruction is known (stored). The following example shows the effect of instruction dependency on a pipeline.

Example 1 Consider the execution of ten instructions I1–I10 on a pipeline consisting of four pipeline stages: IF (instruction fetch), ID (instruction decode), IE (instruction execute), and IS (instruction results store). Assume that the instruction I4 is a conditional branch instruction and that when it is executed, the branch is not taken, that is, the branch condition(s) is(are) not satisfied. Assume also that when the branch instruction is fetched, the pipeline stalls until the result of executing the branch instruction is stored. Show the succession of instructions in the pipeline; that is, show the Gantt's chart. Figure 9.4 shows the required Gantt's chart. The bubble created due to the pipeline stall is clearly shown in the figure.

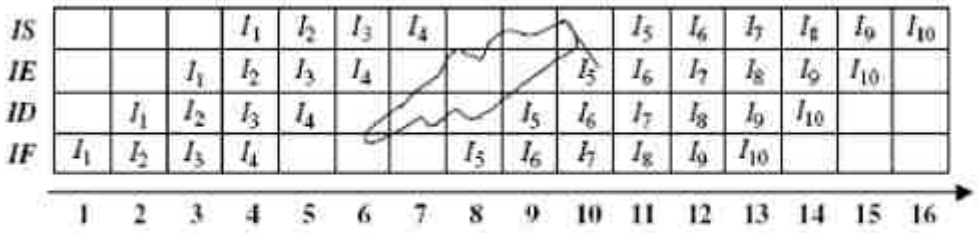


Figure 9.4 Instruction dependency effect on a pipeline

6.3 Pipeline "Stall" Due to Data Dependency

Data dependency in a pipeline occurs when a source operand of instruction Ii depends on the results of executing a preceding instruction, Ij, i > j. It should be noted that although instruction Ii can be fetched, its operand(s) may not be available until the results of instruction Ij are stored. The following example shows the effect of data dependency on a pipeline.

Example 2 Consider the execution of the following piece of code:

•

ADD	R_1, R_2, R_3;	$R_3 \leftarrow R_1 + R_2$
SL	R_3;	$R_3 \leftarrow SL(R_3)$
SUB	R_5, R_6, R_4;	$R_4 \leftarrow R_5 - R_6$

•

In this piece of code, the first instruction, call it Ii, adds the contents of two registers R1 and R2 and stores the result in register R3. The second instruction, call it Ii+1, shifts the contents of R3 one bit position to the left and stores the result back into R3. The third instruction, call it Ii+2 , stores the result of subtracting the content of R6 from the content of R5 in register R4. In order to show the effect of such data dependency, we will assume that the pipeline consists of five stages, IF, ID, OF, IE, and IS. In this case, the OF stage represents the operand fetch stage. The functions of the remaining four stages remain the same as explained before. Figure 9.5 shows the Gantt's chart for this piece of code. As shown in the figure, although instruction Ii+1 has been successfully decoded during time unit k + 2, this instruction cannot proceed to the OF unit during time unit k + 3. This is because the operand to be fetched by Ii+1 during time unit k+3 should be the content of register R3, which has been modified by execution of instruction Ii . However, the modified value of R3 will not be available until the end of time unit k + 4. This will require instruction Ii+1 to wait (at the output of the ID unit) until k + 5. Notice that instruction Ii+2 will

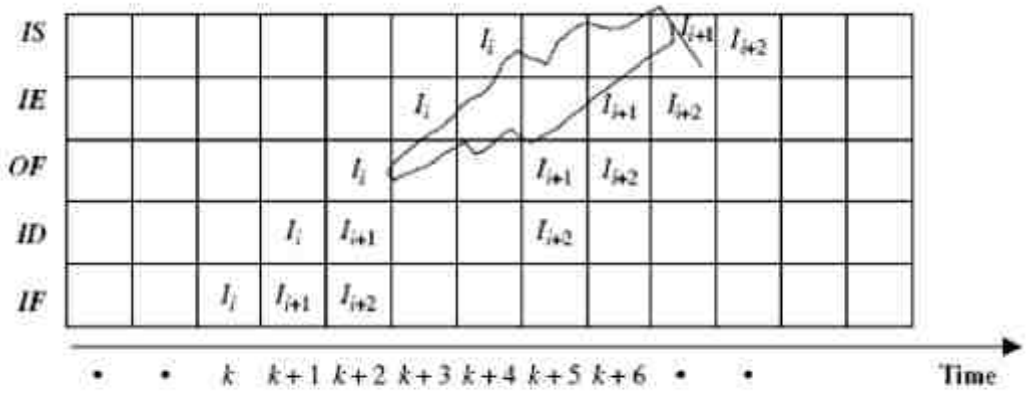


Figure 9.5 The write-after-write data dependency

have also to wait (at the output of the IF unit) until such time that instruction Iiþ1 proceeds to the ID. The net result is that pipeline stall takes place due to the data dependency that exists between instruction Ii and instruction Iiþ1. The data dependency presented in the above example resulted because register R3 is the destination for both instructions Ii and Iiþ1. This is called a write-after-write data dependency. Taking into consideration that any register can be written into (or read from), then a total of four different possibilities exist, including the write-after-write case. The other three cases are read-after-write, write-after-read, and read-after-read. Among the four cases, the read-after-read case should not lead to pipeline stall. This is because a register read operation does not change the content of the register. Among the remaining three cases, the write-after-write (see the above example) and the read-after-write lead to pipeline stall.

The following piece of code illustrates the read-after-write case:

```
•
ADD   R1, R2, R3;   R3 ← R1 + R2
SUB   R3, 1, R4;    R4 ← R3 − 1
•
```

In this case, the first instruction modifies the content of register R3 (through a write operation) while the second instruction uses the modified contents of R3 (through a read operation) to load a value into register R4. While these two instructions are proceeding within a pipeline, care should be taken so that the value of register R3 read in the second instruction is the updated value resulting from execution of the previous instruction. Figure 9.6 shows the Gantt's chart for this case assuming that the first instruction is called Ii and the second instruction is called Ii+1. It is clear that the operand of the second instruction cannot be fetched during time unit

k+3 and that it has to be delayed until time unit k + 5. This is because the modified value of the content of register R3 will not be available until time slot k + 5.

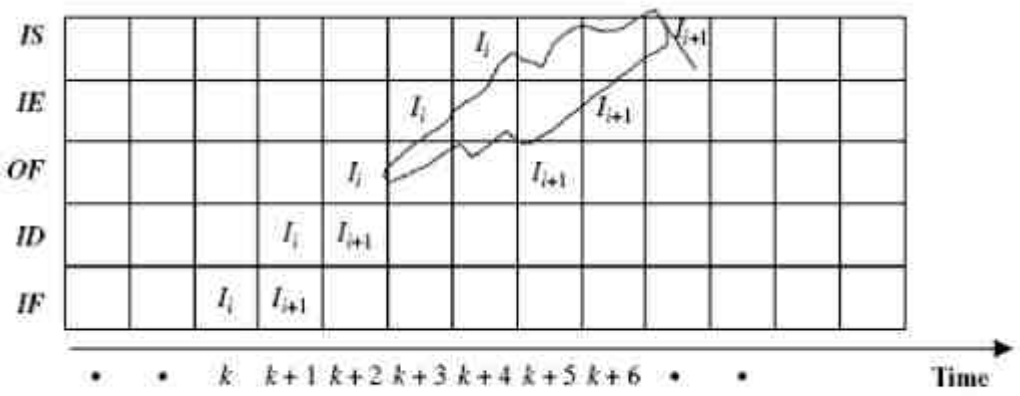


Figure 9.6 The read-after-write data dependency

Fetching the operand of the second instruction during time slot k þ 3 will lead to incorrect results.

Example 3 Consider the execution of the following sequence of instructions on a five-stage pipeline consisting of IF, ID, OF, IE, and IS. It is required to show the succession of these instructions in the pipeline.

$I_1 \rightarrow$	Load	$-1, R1;$	$R1 \leftarrow -1;$
$I_2 \rightarrow$	Load	$5, R2;$	$R2 \leftarrow 5;$
$I_3 \rightarrow$	Sub	$R2, 1, R2$	$R2 \leftarrow R2 - 1;$
$I_4 \rightarrow$	Add	$R1, R2, R3;$	$R3 \leftarrow R1 + R2;$
$I_5 \rightarrow$	Add	$R4, R5, R6;$	$R6 \leftarrow R4 + R5;$
$I_6 \rightarrow$	*SL*	$R3$	$R3 \leftarrow SL\ (R3)$
$I_7 \rightarrow$	Add	$R6, R4, R7;$	$R7 \leftarrow R4 + R6;$

In this example, the following data dependencies are observed:

Instructions	Type of data dependency
I_3 and I_2	Read-after-write and write-after-write (W-W)
I_4 and I_1	Read-after-write (R-W)
I_4 and I_3	Read-after-write (R-W)
I_6 and I_4	Read-after-write and write-after-write (W-W)
I_7 and I_5	Read-after-write (R-W)

Figure 9.7 illustrates the progression of these instructions in the pipeline taking into consideration the data dependencies mentioned above. The assumption made in constructing the Gantt's chart in Figure 9.7 is that fetching an operand by an instruction that depends on the

results of a previous instruction execution is delayed until such operand is available, that is, the result is stored. A total of 16 time units are required to execute the given seven instructions taking into consideration the data dependencies among the different instructions.

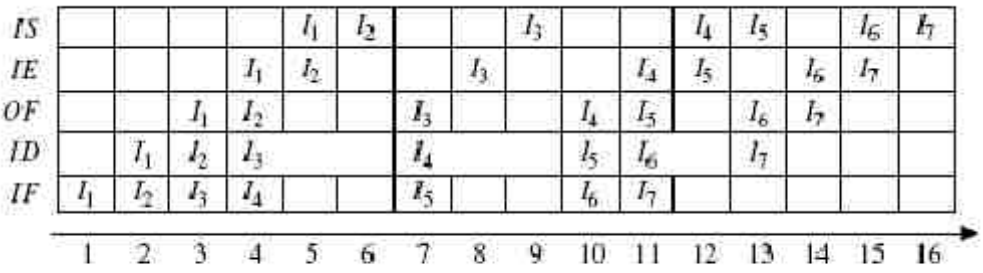


Figure 9.7 Gantt's chart for Example 3

Based on the results obtained above, we can compute the speed-up and the throughput for executing the piece of code given in Example 3 as:

$$\textit{Speed-up } S(5) = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}} = \frac{7 \times 5}{16} = 2.19$$

$$\textit{Throughput } U(5) = \text{No. of tasks executed per unit time} = \frac{7}{16} = 0.44$$

The discussion on pipeline stall due to instruction and data dependencies should reveal three main points about the problems associated with having such dependencies.

These are:

1. Both instruction and data dependencies lead to added delay in the pipeline.
2. Instruction dependency can lead to the fetching of the wrong instruction.
3. Data dependency can lead to the fetching of the wrong operand.

There exist a number of methods to deal with the problems resulting from instruction and data dependencies. Some of these methods try to prevent the fetching of the wrong instruction or the wrong operand while others try to reduce the delay incurred in the pipeline due to the existence of instruction or data dependency. A number of these methods are introduced below.

6.4. Instruction-Level Parallelism

Contrary to pipeline techniques, instruction-level parallelism (ILP) is based on the idea of multiple issue processors (MIP). An MIP has multiple pipelined datapaths for instruction execution. Each of these pipelines can issue and execute one instruction per cycle. Figure 9.17 shows the case of a processor having three pipes. For comparison purposes, we also show in the same figure the sequential and the single pipeline case. It is clear from the figure that while the limit on the number of cycles per instruction in the case of a single pipeline is CPI = 1, the MIP can achieve CPI < 1.

In order to make full use of ILP, an analysis should be made to identify the instruction and data dependencies that exist in a given program. This analysis should lead to the appropriate scheduling of the group of instructions that can be issued simultaneously while retaining the program correctness. Static scheduling results in the use of very long instruction word (VLIW) architectures, while dynamic scheduling results in the use of superscalar architectures.

In VLIW, an instruction represents a bundle of many operations to be issued simultaneously. The compiler is responsible for checking all dependencies and making the appropriate groupings/scheduling of operations. This is in contrast with superscalar architectures, which rely entirely on the hardware for scheduling of instructions.

Superscalar Architectures A scalar machine is able to perform only one arithmetic operation at once. A superscalar architecture (SPA) is able to fetch, decode, execute, and store results of several instructions at the same time. It does so by transforming a static and sequential instruction stream into a dynamic and parallel one, in order to execute a number of instructions simultaneously. Upon completion, the SPA reinforces the original sequential instruction stream such that instructions can be completed in the original order. In an SPA instruction, processing consists of the fetch, decode, issue, and commit stages. During the fetch stage, multiple instructions are fetched simultaneously.

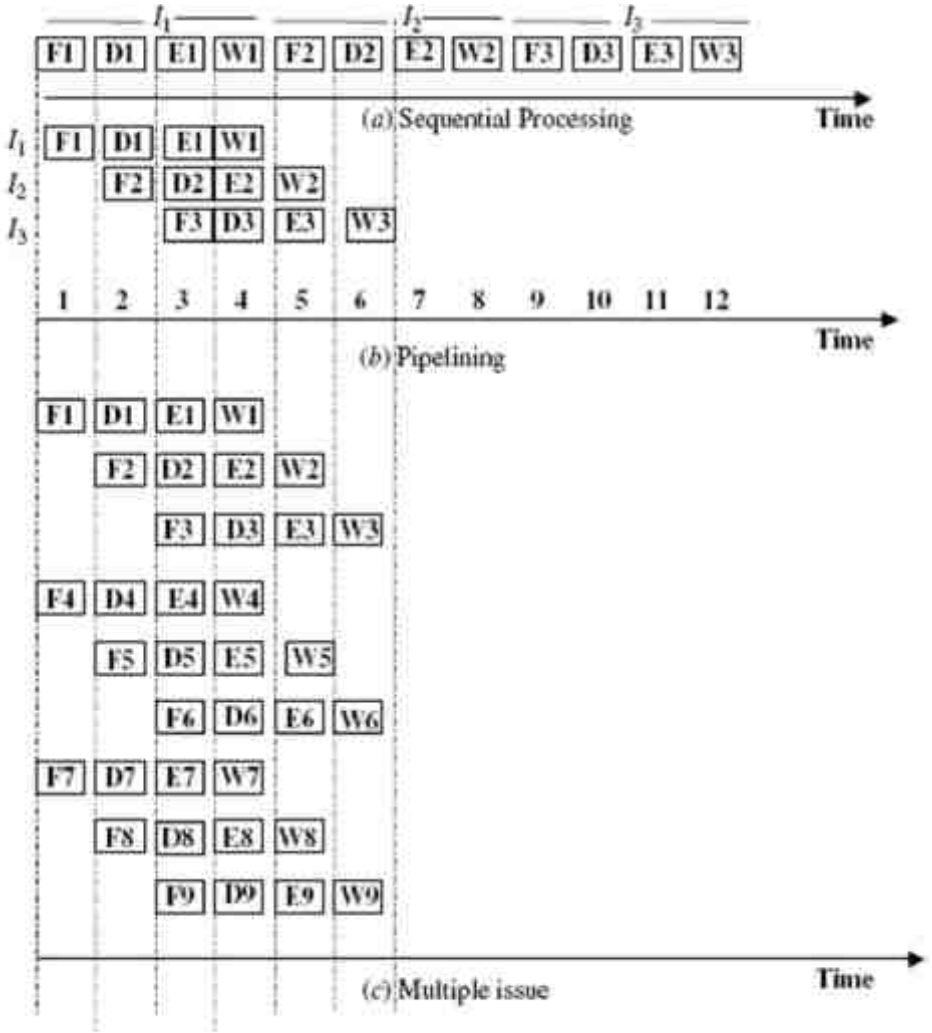


Figure 9.17 Multiple issue versus pipelining versus sequential processing

Branch prediction and speculative execution are also performed during the fetch stage. This is done in order to keep on fetching instructions beyond branch and jump instructions. Decoding is done in two steps. Predecoding is performed between the main memory and the cache and is responsible for identifying branch instructions.

Actual decoding is used to determine the following for each instruction: (1) the operation to be performed; (2) the location of the operands; and (3) the location where the results are to be stored. During the issue stage, those instructions among the dispatched ones that can start execution are identified. During the commit stage, generated values/results are written into their destination registers.

The most crucial step in processing instructions in SPAs is the dependency analysis. The complexity of such analysis grows quadratically with the instruction word size. This puts a limit on the degree of parallelism that can be achieved with SPAs such that a degree of

parallelism higher than four will be impractical. Beyond this limit, the dependence analysis and scheduling must be done by the compiler. This is the basis for the VLIW approach.

REFERENCES

Baron, Robert J. and Higbie Lee. *Computer Architecture*. Addison-Wesley Publishing Company.

Daniel P, and David G, (2009). A Practical Introduction to Computer Architecture. Text in Computer Science, Springer Dordrecht Heidelberg London New York.

Flautner, K. Kim, N.S. Martin, S. Blaauw D. and Mudge, T.N. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *International Symposium on Computer Architecture. (ISCA)*, 148–157, 2002

Hayes, John P. (1998). *Computer Architecture and Organisation* (2nd ed). McGraw-Hill International editions.

Hennessy J.L and Patterson, D.A. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2006. ISBN: 0-123-70490-1.

Mano, M. Morris, (1993). *Computer System Architecture* (3rd ed). Prentice Hall of India.

Miles J, and Vincent P. H. (1999). Principle of Computer Architecture – Class Test Edition, August 1999. http://www.cs.rutgers.edu/~murdocca/
http://ece-www.colorado.edu/faculty/heuring.html

Mostafa A. E.B and Hesham E. R, (2005). Fundamentals of Computer Organization and Architecture. Published by John Wiley & Sons, Inc., Hoboken, New Jersey. Published simultaneously in Canada.

NOUN, (2008). INTRODUCTION TO COMPUTER ORGANISATION . CIT 246 Course Material, National Open University of Nigeria, Headquarters 14/16 Ahmadu Bello Way, Victoria Island,Lagos. First Printed 2008

Stallings William. *Computer Organisation and Architecture* (3rd ed). Maxwell Macmillan International Editions.

Tanenbaum, Andrew S. (1993). *Structural Computer Organisation* (3rd ed) Prentice Hall of India.

http://websrv.cs.fsu.edu/~tyson/CDA5155/refs.html
http://www.cs.ucsd.edu/classes/wi99/cse141_B/lectures.html
http://www.cs.caltech.edu/courses/cs184/ winter2001/slides/day5_2up.pdf
http://www.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/compSched.html
http://www.mmdb.ece.ucsb.edu/~ece154/lecture5.pdf