Unit – 1

Functional blocks of a computer:

A computer consists of five functionally independent main parts:

1. input
2. memory
3. arithmetic and logic
4. output
5. control unit

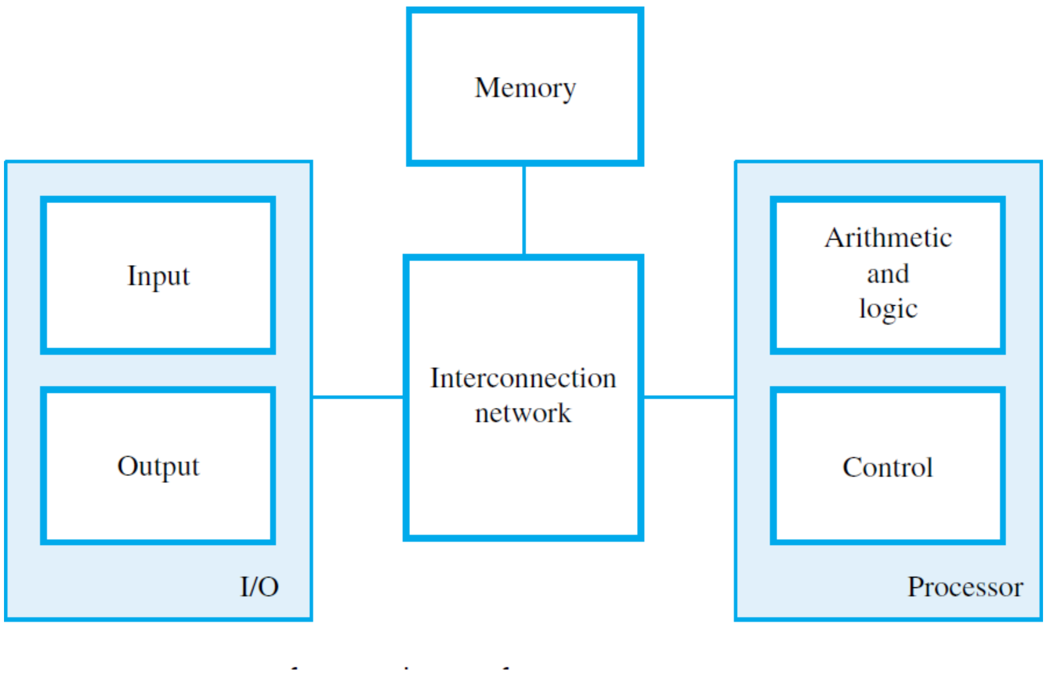


Fig:Basic functional units of a computer.

1. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines.
2. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit.
3. The processing steps are specified by a program that is also stored in the memory.
4. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit.
5. An interconnection network provides the means for the functional units to exchange information and coordinate their actions.

Aprogram is a list of instructions which performs a task. Programs are stored in the memory.The processor fetches the program instructions from the memory, one after another, and perform the desired operations. The computer is controlled by the stored program, exceptfor possible external interruption by an operator or by I/O devices.The instructions and data handled by a computer is encoded as a string of binary bits.

- **Input Unit**: Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input.

Eg: touchpad, mouse, joystick

- **Memory Unit:**

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory: Primary memory, also called main memory, is a fast memory that operates at electronicspeeds. Programs must be stored in this memory while they are being executed.The memory consists of a large number of semiconductor storage cells, each capable of storingone bit of information.The memory is organized so that one word canbe stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits.To provide easy access to any word in the memory, a distinct address is associatedwith each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing acontrol command to the memory that starts the storage or retrieval process.Instructions and data can be written into or read from the memory under the control ofthe processor. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The timerequired to access one word is called the memory access time. This time is independent ofthe location of the word being accessed.

Cache Memory: As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates. At the start of program execution, the cache is empty. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Secondary Storage: Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondarystorage is used when large amounts of data and many programs have to be stored, particularlyfor information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. Awide selection of secondary storage devices is available, including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit: Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use. When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data.

Output Unit: The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a printer. Some units, such as graphic displays, provide both an output function, showing textand graphics, and an input function, through touchscreen capability.

Control Unit: The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states. Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place. In practice, much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

Basic Operational Concepts

To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.A typical instruction might be

Load R2, LOC

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.The original contents of location LOC are preserved, whereas those of register R2 are overwritten.Execution of this instruction requires several steps.

1. . First, the instruction is fetched from the memory into the processor.
2. Next, the operation to be performed is determined by the control unit.
3. The operand at LOC is then fetched from the memory into the processor.
4. Finally, the operand is stored in register R2.

Let us consider another example

Add R4, R2, R3

This instruction adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum. After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

Store R4, LOC

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location and asserting the appropriate control signals.The data are then transferred to or from the memory.Figure 1.1 shows how the memory and the processor can be connected.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.

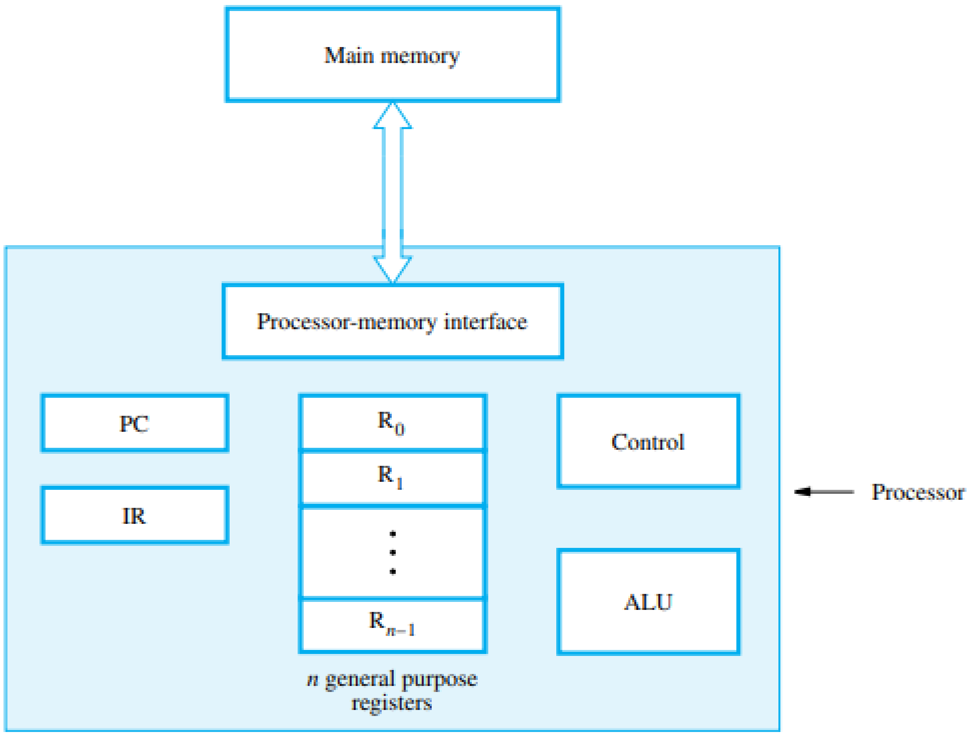


Fig 1.1: ***Connection between the processor and the main memory***

PC: The program counter (PC) is another specialized register.contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed.

General purpose Registers: There are also general-purpose registers R0 through Rn−1, often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing.

Processor Memory Interface: The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal. The interface waits for the word to be retrieved, then transfers it to the appropriate processor register.If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Following are typical operating steps:

1) A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage
2) Execution of the program begins when the PC is set to point to the first instruction of the program.
3) The contents of the PC are transferred to the memory along with a Read control signal. When the addressed word (in this case, the first instruction of the program) has been fetched from the

memory it is loaded into register IR. At this point, the instruction is ready to be decoded and executed.

4) If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register "**R**".
5) After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register.
6) If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.

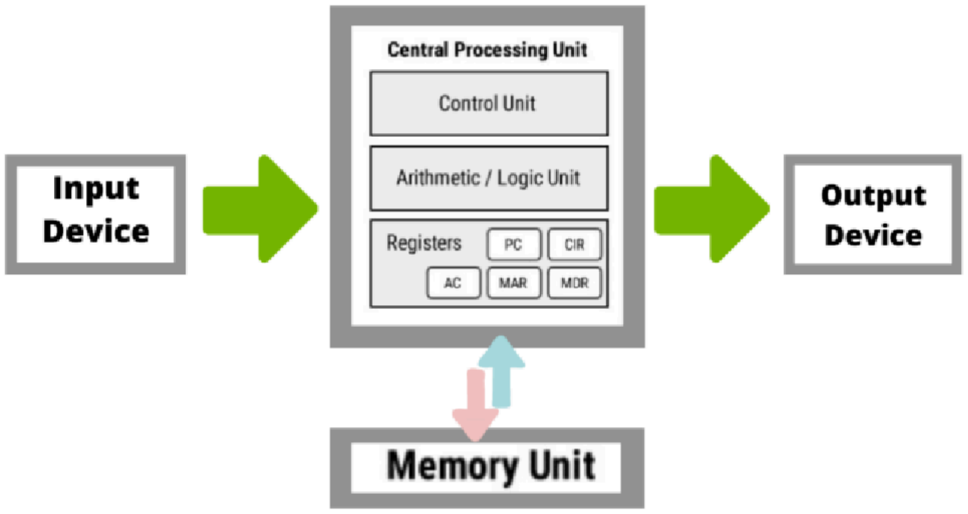
At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

Normal execution of a program may be preempted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an interrupt signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an interrupt-service routine. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

Von Neumann Architecture:

The Von-Neumann Architecture or Von-Neumann model is also known as **"Princeton Architecture"**. This architecture was published by the Mathematician **John Von Neumann** in **1945**.

Von Neumann architecture is the design upon which many general purpose computers are based. This architecture implemented the stored program concept in which the data and instructions are stored in the same memory. This architecture consists of a CPU(ALU, Registers, Control Unit), Memory and I/O unit.



Following are the components of Von Neumann Architecture:

1. CPU(Central processing unit)
   - CU(Control Unit)
   - ALU(Arithmetic and logic unit)
   - Registers
     - ✔ PC(Program Counter)
     - ✔ IR(Instruction Register)
     - ✔ AC(Accumulator)
     - ✔ MAR(Memory Address Register)
     - ✔ MDR(Memory Data Register)

2. BUSES

3. I/o Devices

4. Memory Unit

1. CPU: CPU acts as the brain of the computer and is responsible for the execution of instructions.

a) **Control Unit:** A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches code for instructions, and controls how data moves around the system.

b) **Arithmetic and Logic Unit (ALU) :**
The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic operations.

c) **Registers**: A processor based on von Neumann architecture has five special registers which it uses for processing:

- **Program counter (PC)** holds the memory address of the next instruction to be fetched from primary storage.
- The **Memory Address Register(MAR)** holds the address of the current instruction that is to be fetched from memory, or the address in memory to which data is to be transferred.
- The **Memory Data Register(MDR)** holds the contents found at the address held in MAR or data which is to be transferred to the primary storage.
- The **Current Instruction Register(CIR)** holds the instruction that is currently being decoded and executed.
- The **Accumulator** is a special purpose Register and is used by the ALU to hold the data being processed and the results of calculations.

2. **BUSES** :A bus is a subsystem that is used to connect computer components and transfer data between them. There are three types of BUSES

a) **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
b) **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
c) **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

3. **I/o Devices**:Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer. If some results are evaluated by CPU and it is stored in the computer, then with the help of output devices, we can present them to the user.

4. **Memory:**A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage. The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word(2^Mx N, eg: 128KB).

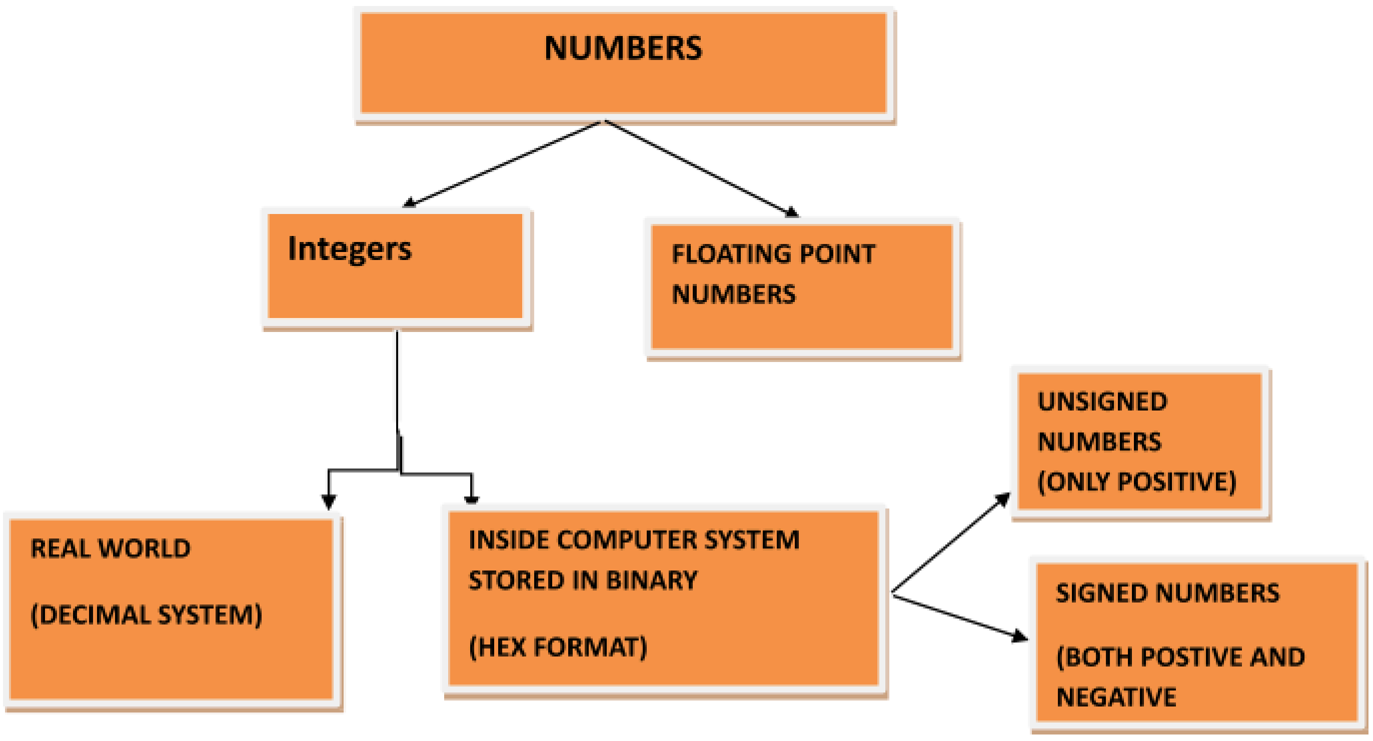
There are two types of Primary Memory:

1) RAM: VOLATILE MEMORY or temporary Memory(to store the program in execution)

2) ROM: NON-VOLATILE MEMORY or permanent Memory(to store the booting program)

Unit – 2 & 3

NUMBER REPRESENTATION:


NUMBERS
Integers
FLOATING POINT NUMBERS
REAL WORLD (DECIMAL SYSTEM)
INSIDE COMPUTER SYSTEM STORED IN BINARY (HEX FORMAT)
UNSIGNED NUMBERS (ONLY POSITIVE)
SIGNED NUMBERS (BOTH POSTIVE AND NEGATIVE

UNSIGNED INTEGERS

These are binary numbers that are always assumed to be positive.Here all available bits of the number are used to represent the magnitude of the number.No bits are used to indicate itssign, hence they are called unsigned numbers.
E.g.: Roll Numbers, Memory addresses etc

SIGNED INTEGERS

These are binary numbers that can be either positive or negative. The MSB of the number indicates whether it is positive or negative. If **MSB is 0 then the number is Positive**. If **MSB is 1 then the number is Negative**. Negative numbers are always stored in 2's complement form.

Three systems are used forrepresenting such numbers:

- **Signed magnitude**
- **1's-complement**
- **2's-complement**

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.Positive values have identical representations in all systems, but negative values have different representations.

In the **signed magnitude system**, negative values are represented by changing the mostsignificant bit from 0 to 1.For example, +5 is represented by 0101, and −5 is represented by 1101.

In **1's-complement representation**, negative values are obtained by complementing eachbit of the corresponding positive number. Thus, the representation for −3 is obtainedby complementing each bit in the vector 0011 to yield 1100.The same operation, bitcomplementing, is done to convert a negative number to the corresponding positive value.

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

Fig: Binary signed number Representations

Two's complement gives a unique representation for zero. Any other system gives a separate representation for +0 and for -0. This is absurd. In two's complement system, -(x) is stored as two's complement of (x). Applying the same rule for 0, -(0) should be stored as two's complement of 0. 0 is stored as 000. So –(0) should be stored as two's complement of 000, which again is 000. Hence two's complement gives a unique representation for 0.**It produces an additional number on the negative side.** As two's complement system produces a unique combination for 0, it has a spare combination '1000' in the above case, and can be used to represent –(8).

3 Bit Integer	
2^3 = 8 therefore 8 combinations	
Unsigned	Signed
0 ... 7	-4 ... -1 0 1 ... 3

4 Bit Integer	
2^4 = 16 therefore 16 combinations	
Unsigned	Signed
0 ... 15	-8 ... -1 0 1 ... 7

Fixed and Floating point Representations:

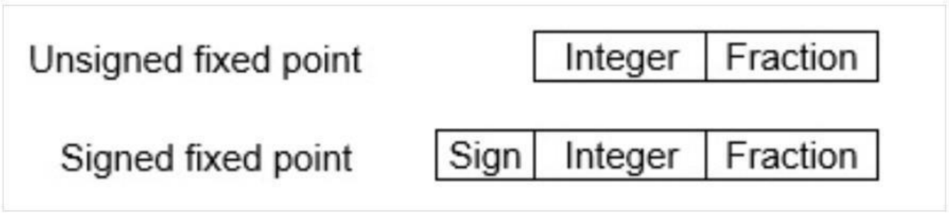
There are two major approaches to store real numbers (i.e., numbers withfractional component) in modern computing. These are

(i) Fixed Point Notation and

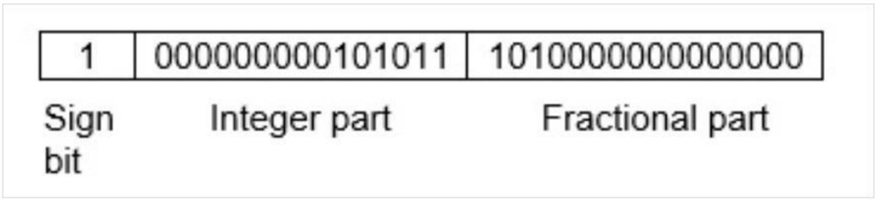
(ii) Floating Point Notation.

Fixed Point Notation: In **fixed point notation**, there are a fixed number of digits after the decimal point, whereas **floating point number** allows for avarying number of digits after the decimal point.

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.



Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part. Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent -. 000000000101011 is 15-bit binary value for decimal 43 and 1010000000000000 is 16-bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.

<u>Floating Point Representation:</u>

In some numbers, which have a fractional part, the position of the decimal point is not fixed as the number of bits before (or after) the decimal point may vary. **Eg: 0010.01001, 0.0001101, -1001001.01** etc. the position of the decimal point is not fixed, instead it**"floats"** in the number.Such numbers are called Floating Point Numbers. Floating Point Numbers are stored in a "Normalized" form.

<u>NORMALIZATION OF A FLOATING POINTNUMBER:</u>

Normalization is the process of shifting the point, left or right, so that there is only one non-zero digit to the left of the point.

01010.01 (-1)***0*** x 1.***01001*** x 2^3

11111.01 (-1)***0*** x 1.***111101*** x 2^4

-10.01 (-1)***1*** x 1.***001*** x 2^1

A normalized form of a number is:

$$\mathbf{-1^S x1.MX2^E}$$

Where: S = Sign, M = Mantissa and E = Exponent.

As Normalized numbers are of the 1.M format, the "1" is not actually stored, it is instead assumed. Also the Exponent is stored in the biased form by adding an appropriate bias value to it so that -ve exponents can be easily represented.

Advantages of Normalization.

1. Storing all numbers in a standard for makes **calculations easier** and **faster**.
2. By **not storing** the **1** (of 1.M format) for a number, considerable **storage space** is **saved**.
3. The **exponent** is **biased** so there is **no need** for **storing** its **sign bit** (as the biased exponent cannot be - ve).

<u>SHORT REAL FORMAT / SINGLE PRECISION FORMAT / IEEE 754: 32 BIT FORMAT:</u>

S	Biased Exponent	Mantissa
(1)	(8) Bias value = 127	(23 bits)

1. **32 bits** are used to store the **number**.
2. **23 bits** are used for the **Mantissa**.
3. **8 bits** are used for the Biased **Exponent**.
4. **1 bit** used for the **Sign** of the number.
5. The **Bias** value is (**127**)10.

Range: $\underline{+}1 \times 10^{-38}$ **to** $\underline{+}3 \times 10^{38}$

<u>LONG REAL FORMAT / DOUBLE PRECISION FORMAT / IEEE 754: 64 BIT FORMAT</u>

1. **64 bits** are used to store the **number**.
2. **52 bits** are used for the **Mantissa**.
3. **11 bits** are used for the Biased **Exponent**.
4. **1 bit** used for the **Sign** of the number.
5. The **Bias** value is (**1023**)10.
6. The range is **+10^{-308} to +10^{308}**approximately.

s	Biased Exponent	Mantissa
1 bit	**11-bits (Bias value:1023)**	**52-bits**

<u>Extreme cases of floating point numbers:</u>

Floating point numbers are represented in IEEE formats.Consider IEEE 754 32-bit format also called Single Precision format or Short real format.

Overflow:

For a value, 1.0 the normalized form will be

$(-1)^0 \times 1.0 \times 2^0$

Herethe True Exponent is 0.

```
If:  TE = 0,     BE = 127    Representation = 0111 1111
If:  TE = 1,     BE = 128    Representation = 1000 0000
If:  TE = 2,     BE = 129    Representation = 1000 0000
...
If:  TE = 127,   BE = 254    Representation = 1111 1110
If:  TE = 128,   BE = 255    Representation = 1111 1111
If:  TE = 129,   BE = 255    Representation = 1111 1111
If:  TE = 130,   BE = 255    Representation = 1111 1111
```

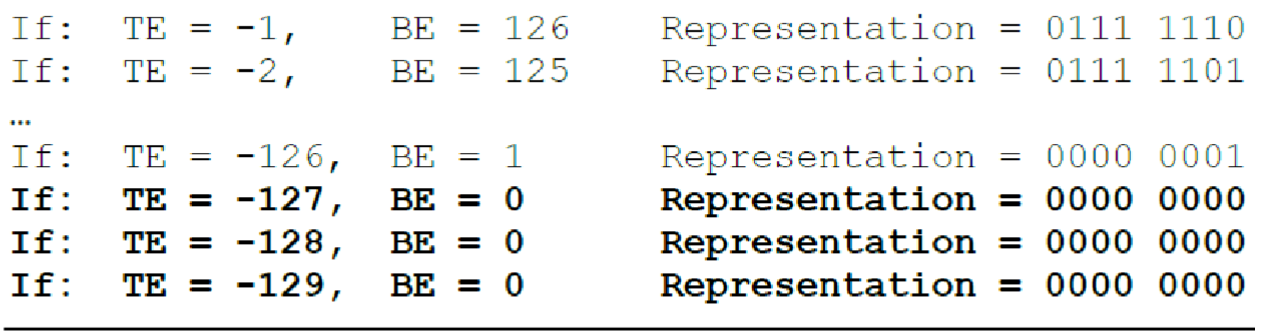
This is because the 8-bit biased exponent cannot hold a value more than 255.Hence, all cases where the TE = 128 or more, the ***BE will be represented as 1111 1111.This indicates as exception (error) called OVERFLOW. The number is called NaN (Not a Number).***It is identified by Exponent being all 1s (1111

1111).Here, the Mantissa can be anything!The **Extreme case of NaN is Infinity**.It is also an OVERFLOW and hence the Exponent will be 1111 1111.To differentiate Infinity from NaN, the Mantissa in infinity is 0000 0000.Hence **Infinity is identified as Exponent all 1s and Mantissa all 0s.**

Suppose the number is 0.1.It will be normalized

as $(-1)^0$ x 1.0 x 2^{-1}

The true exponent here is -1.



```
If:  TE = -1,     BE = 126    Representation = 0111 1110
If:  TE = -2,     BE = 125    Representation = 0111 1101
…
If:  TE = -126,   BE = 1      Representation = 0000 0001
If:  TE = -127,   BE = 0      Representation = 0000 0000
If:  TE = -128,   BE = 0      Representation = 0000 0000
If:  TE = -129,   BE = 0      Representation = 0000 0000
```

Underflow: All cases where the TE = -127 or less, the BE will be represented as 0000 0000.This indicates as exception (error) called UNDERFLOW.

The number is called De-Normal Number.It is identified by Exponent being all 0s (0000 0000).Here, the Mantissa can be anything.The **Extreme case of De-Normal Number is Zero.**

It is also an UNDERFLOW and hence the Exponent will be 0000 0000.To differentiate Zero from De-Normal Number, the Mantissa in Zero is 0000 0000.Hence **Zero is identified as Exponent all 0s and Mantissa all 0s**.This means Zero is represented as all 0s.

Example:Convert 2A3BH into Short Real format.

Soln: Converting the number into binary we get:

0010 1010 0011 1011

Normalizing the number we get:

$(-1)^0$**x** 1.**0101000111011** x 2^{13}

Here S = 0; M = 0101000111011; True Exponent = 13.

Bias value for Short Real format is 127:

Biased Exponent (BE) = True Exponent + Bias

= 13 + 127

= 140.

Converting the Biased exponent into binary we get:

Biased Exponent (BE) = (1000 1100)

Representing in the required format we get:

0	**10001100**	**010100011101100…**

S Biased Exp Mantissa

(1) (8) (23)

Computer Arithmetic

Integer Addition:

Addition of Unsigned Integers: Addition of 1-bit numbers is illustrated below.The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the sum is 0 and the carry-out is 1. In order to add multiple-bit numbers, We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the carry-in to the next bit pair to the left. The carry-in must be added to a bit pair in generating the sum and carry-out at that position. For example, if both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is 1, which represents the value 3.

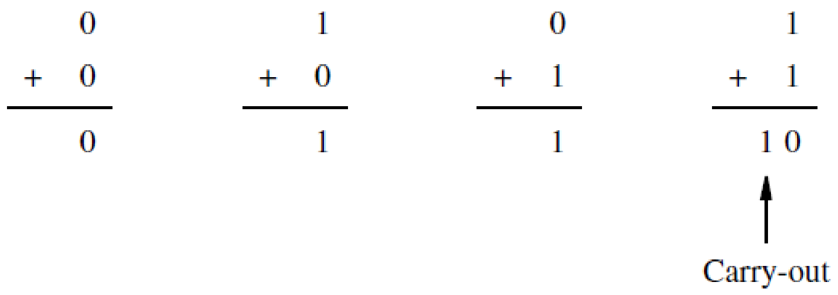


Fig: Addition of 1-bit Numbers

Addition and Subtraction of Signed Integers:

- To add two numbers, add their n-bit representations, ignoring the carry-out bit fromthe most significant bit (MSB) position. The sum will be the algebraically correct value in2's-complement representation if the actual result is in the range$-(2^{n-1})$ through$+2^{n-1}-1$.
- To subtract two numbers X and Y, that is, to perform X − Y , form the 2's-complement of Y , then add it to X using the add rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range $-(2^{n-1})$ through$+2^{n-1}$.

X-Y = X+(-Y) = X+(2'S Complement of Y)

<u>*Example:*</u> **To perform 7-3 using 2's complement addition**

$$\begin{array}{rcccc} & 0 & 1 & 1 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 \\ \uparrow & & & & \\ \text{Carry-out} & & & & \end{array}$$

If we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer.

Few more examples:

$$\begin{array}{llrr} \text{(a)} & & 0010 & (+2) \\ & + & 0011 & (+3) \\ \hline & & 0101 & (+5) \end{array} \qquad \begin{array}{llrr} \text{(b)} & & 0100 & (+4) \\ & + & 1010 & (-6) \\ \hline & & 1110 & (-2) \end{array}$$

$$\begin{array}{llrr} \text{(c)} & & 1011 & (-5) \\ & + & 1110 & (-2) \\ \hline & & 1001 & (-7) \end{array} \qquad \begin{array}{llrr} \text{(d)} & & 0111 & (+7) \\ & + & 1101 & (-3) \\ \hline & & 0100 & (+4) \end{array}$$

$$\begin{array}{llrr} \text{(e)} & & 1101 & (-3) \\ & - & 1001 & (-7) \\ \hline & & & \end{array} \quad \Longrightarrow \quad \begin{array}{lrr} & 1101 & \\ + & 0111 & \\ \hline & 0100 & (+4) \end{array}$$

Sign Extension: We often need to represent a value given in a certain number of bits by using a larger number of bits. For a positive number, this is achieved by adding 0s to the left. For a negative number in 2's-complement representation, the leftmost bit, which indicates the sign of the number, is a 1. A longer number with the same value is obtained by replicating the sign bit to the left as many times as needed.

Overflow in Integer Arithmetic: Using 2's-complement representation, n bits can represent values in the range $-(2^{n-1})$ through$+2^{n-1}$.For example, the range of numbers that can be represented by 4 bits is −8through +7.When the actualresult of an arithmetic operation isoutside the representable range, an arithmetic overflow has occurred.

Introduction to adder circuits:

ONE BIT ADDITION: FULL ADDER

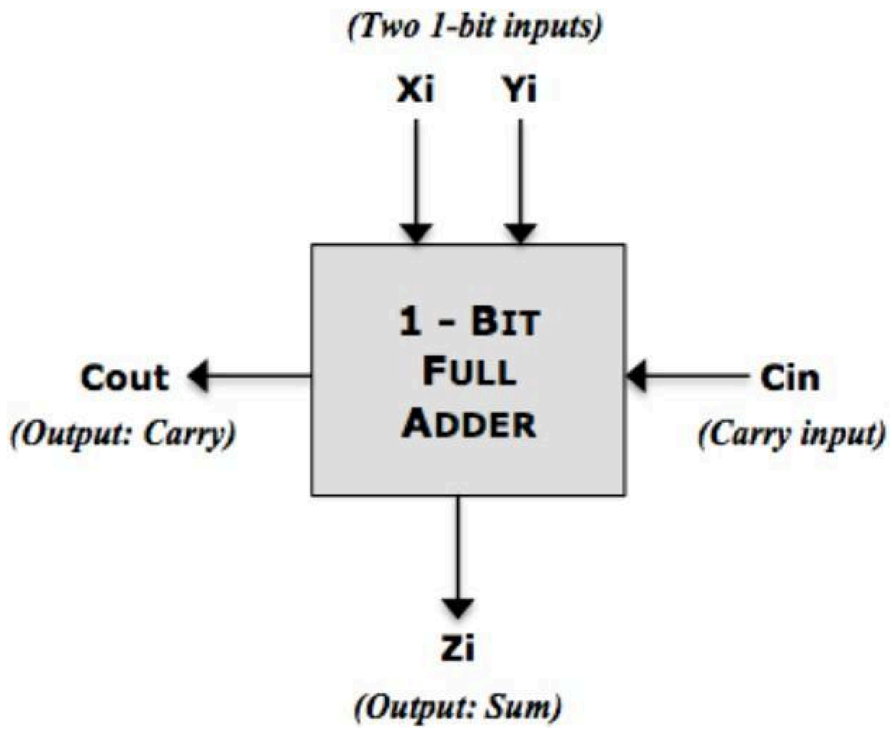
1) It is a 1-bit adder circuit.

2) It adds two 1-bit inputs Xi & Yi, along with a Carry Input Cin.

3) It produces a sum Zi and a Carry output Cout.

4) As it considers a carry input, it can be used in combination to add large numbers.

5) Hence it is called a Full Adder.



Inputs bits: Xi and Yi.
Input Carry: Cin

Output (Sum): Zi
Output (Carry): Cout

<u>Formula for Sum (Zi)</u>

$$Zi = Xi \oplus Yi \oplus Cin$$
$$\therefore Zi = Xi \cdot Yi \cdot Cin + Xi \cdot Yi \cdot \overline{Cin} + Xi \cdot \overline{Yi} \cdot Cin + \overline{Xi} \cdot Yi \cdot Cin$$

<u>Formula for Carry (Cout)</u>

$$Cout = Xi \cdot Yi + Xi \cdot Cin + Yi \cdot Cin$$

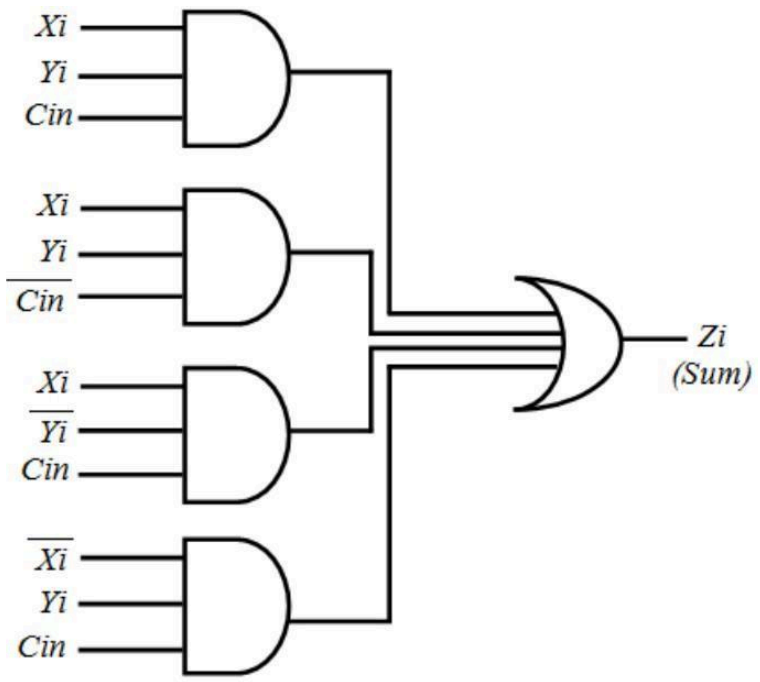


Fig: Circuit for Sum

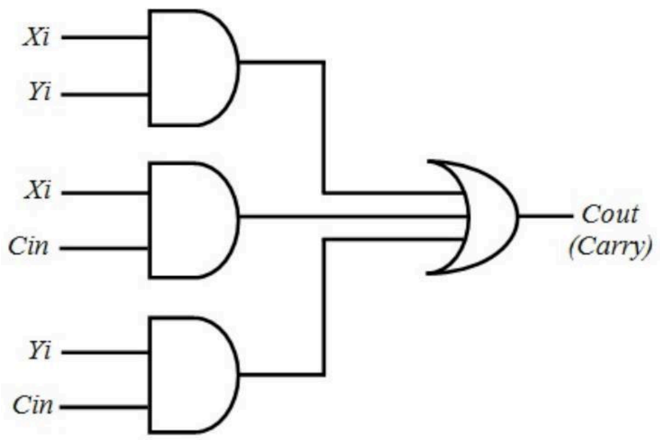


Fig: Circuit for carry

RIPPLE CARRY ADDER(For Multiple bit addition):

1) A Full Adder can add two "1-bit" numbers with a Carry input.

2) It produces a "1-bit" Sum and a Carry output.

3) Combining many of these Full Adders, we can add multiple bits.

4) One such method is called Serial Adder.

5) Here, bits are added one-by-one from Least significant bit(LSB) onwards.

6) The carries are connected in a chain through the full adders. The Carry of each stage is propagated (Rippled) into the next stage.

7) Hence, these adders are also called Ripple Carry Adders.

Advantage: They are very easy to construct.

Drawback: As addition happens bit-by-bit, they are slow.

8) Number of cycles needed for the addition is equal to the number of bits to be added.

Inputs:

Assume X and Y are two "4-bit" numbers to be added, along with a Carry input CIN.

X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB)

Y = Y0 Y1 Y2 Y3 (Y0 is the MSB … Y3 is the LSB)

CIN = Carry Input

Outputs:

Assume Z to be a "4-bit" output, and COUT to be the output Carry

Z = Z0 Z1 Z2 Z3 (Z0 is the MSB … Z3 is the LSB)(Here Z represents the sum)

COUT = Carry Output

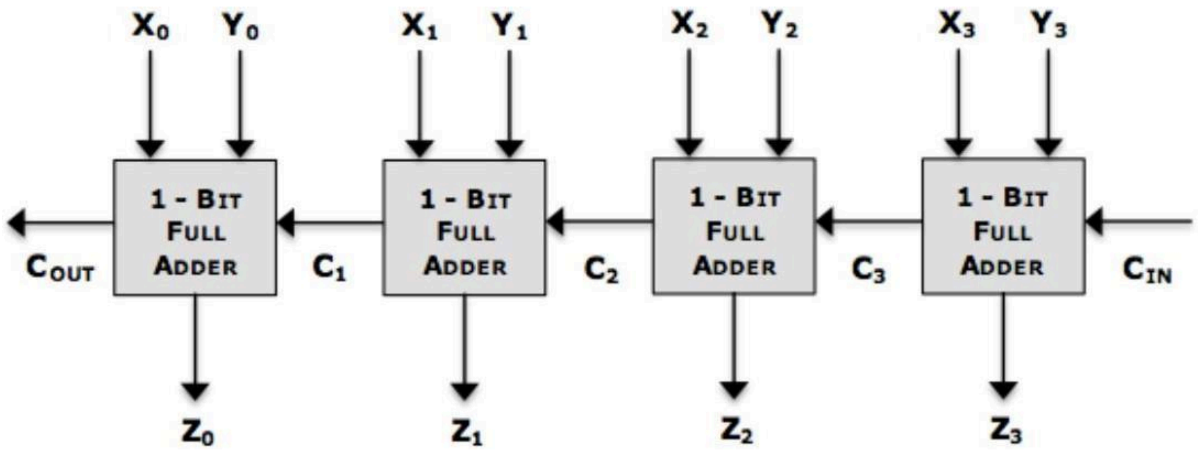


Fig:4-bit Ripple Carry Adder

<u>Carry Look ahead Adder(For multiple bit Addition):</u>

1) This is also called as parallel adder. It is used to add multiple bits simultaneously.
2) While adding multiple bits, the main issue is that of the intermediate carries.
3) In Serial Adders, we therefore added the bits one-by-one.
4) This allowed the carry at any stage to propagate to the next stage.
5) But this also made the process very slow.
6) If we "PREDICT" the intermediate carries, then all bits can be added simultaneously.
7) This is done by the Carry Look Ahead Generator Circuit.
8) Once all carries are determined beforehand, then all bits can be added simultaneously.

Advantage: This makes the addition process extremely fast.

Drawback: Circuit is complex.

Inputs:

Assume X and Y are two "4-bit" numbers to be added, along with a Carry input CIN.

X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB); Y = Y0 Y1 Y2 Y3 & CIN = Carry Input

Outputs:

Assume Z to be a “4-bit” output, and COUT to be the output Carry

Z = Z0 Z1 Z2 Z3 & COUT = Carry Output

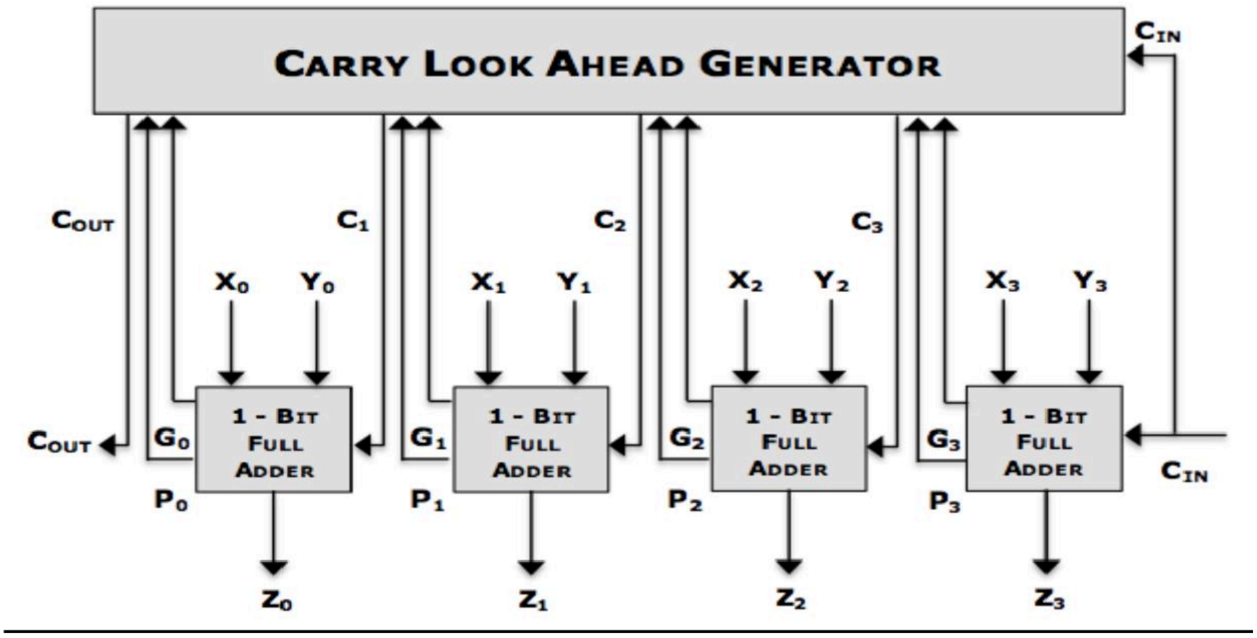


Fig: Circuit for Carry Look ahead Adder

We can “Predict” (Look Ahead) all the intermediate carries in the followingmanner:

The carry at any stage can be calculated as:

$$C_i = X_i.Y_i + X_i.C_{in} + Y_i.C_{in}$$
$$C_i = X_i.Y_i + C_{in}(X_i + Y_i)$$

This implies $C_i = G_i + P_i.C_{IN}$

Here $Gi = X_i.Y_i$ ... (Generate)
And $Pi = X_i+Y_i$ ... (Propagate)

We need to predict the Carries: C3, C2, C1 and C0

C3 = G3 + P3CIN (I)

C2 = G2 + P2C3

Substituting the value of C3, we

get: C2 = G2 + P2G3 + P2P3CIN

(II) C1 = G1 + P1C2

Substituting the value of C2, we get:

C1 = G1 + P1G2 + P1P2G3 + P1P2P3CIN (III)

$C0 = G0 + P0C1$

Substituting the value of C1, we get:

$C0 = G0 + P0G1 + P0P1G2 + P0P1P2G3 + P0P1P2P3CIN$ (IV)

From the above four equations, it is clear that the values of all the four Carries (C3, C2, C1, C0) can be determined beforehand even without doing the respective additions. To do this we need the values of all G's (Xi.Yi) and all P's (Xi+Yi) and the original carry input CIN. This is done by the Carry Look Ahead Generator Circuit.

Cycle 1: g_1, p_1, g_2, p_2, g_3, p_3, g_0, $p_0$are given to the carry look ahead generator.
Cycle 2: Input carries are given to the adders by the carry generator.
Cycle 3: Results are produced.
Total number of cycles required :3

Multiplication:

1)   Shift and Add: This method is used to multiply two unsigned numbers. When we multiply two "N- bit" numbers, the answer is "2 x N" bits. Three registers A, Q and M, are used for this process. Q contains the Multiplier and M contains the Multiplicand. A (Accumulator) is initialized with 0. At the end of the operation, the Result will be stored in (A & Q) combined. The process involves addition and shifting. That is why it is called shift and add method.

Algorithm:

The **number of steps** required is equal to the **number of bits in the multiplier**.

1) At each step, **examine** the current **multiplier bit** starting from the **LSB**.

2) If the current **multiplier bit is "1"**, then the **Partial-Product** is the **Multiplicand** itself.

3) If the current **multiplier bit is "0"**, then the **Partial-Product** is the **Zero**.

4) At each step, **ADD the Partial-Product to the Accumulator**.

5)   Now **Right-Shift the Result** produced so far (**A & Q combined**). **Repeat** steps 1 to 5 for **all bits** of the multiplier.

The **final answer** will be in **A & Q** combined.

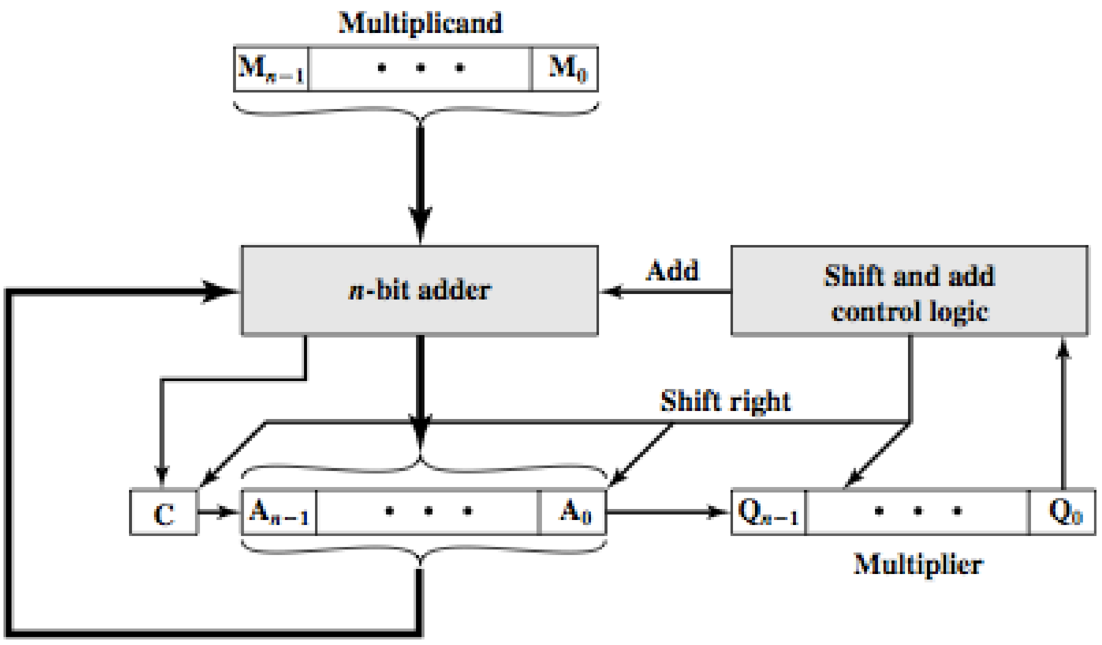


Fig: Shift and Add Multiplication

Example: Let us consider 7X6

```
                0   1   1   1   … Multiplicand (7)
        X       0   1   1   0   … Multiplier (6)
------------------------------------------------------
                0   0   0   0   … Partial-Product
            0   1   1   1   X               ”
        0   1   1   1   X   X               ”
 +  0   0   0   0   X   X   X               ”
------------------------------------------------------
    0   1   0   1   0   1   0   … Result (42)
------------------------------------------------------
```

Step	C Carry	A Accumulator	Q Multiplier	M Multiplicand	Explanation
	0	0000	0110	0111	*Initial Value*
1	**0** **0**	0000 0000	0110 0011		*Current Multiplier bit is “0” so ADD “0” to Accumulator and Right-Shift*

2	0 0	0111 0011	0011 1001		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
3	0 0	1010 0101	1001 0100		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
4	0 0	0101 ***0010***	0100 ***1010***		*Current Multiplier bit is "0" so ADD "0" to Accumulator and Right-Shift*

2) Booth Multiplier(For signed Multiplication):

Booth's Algorithm is used to **multiply two SIGNED numbers**. When we multiply two **"N-bit"** numbers, the answer is "**2 x N**" bits. Three registers A, Q and M, are used for this process.**Q** contains the **Multiplier** and **M** contains the **Multiplicand**.**A** (**Accumulator**) is initialized with 0.At the end of the operation, the **Result** will be stored in (**A & Q**) combined.The process involves **addition, subtraction** and **shifting**.

Algorithm:

The **number of steps** required is equal to the **number of bits in the multiplier**.

At the beginning, consider an **imaginary "0" beyond LSB of Multiplier**

1) At each step, **examine two adjacent Multiplier bits** from **Right to Left**.

2) If the transition is from **"0 to 1"** then **Subtract M** from **A** and **Right-Shift** (A & Q) combined.

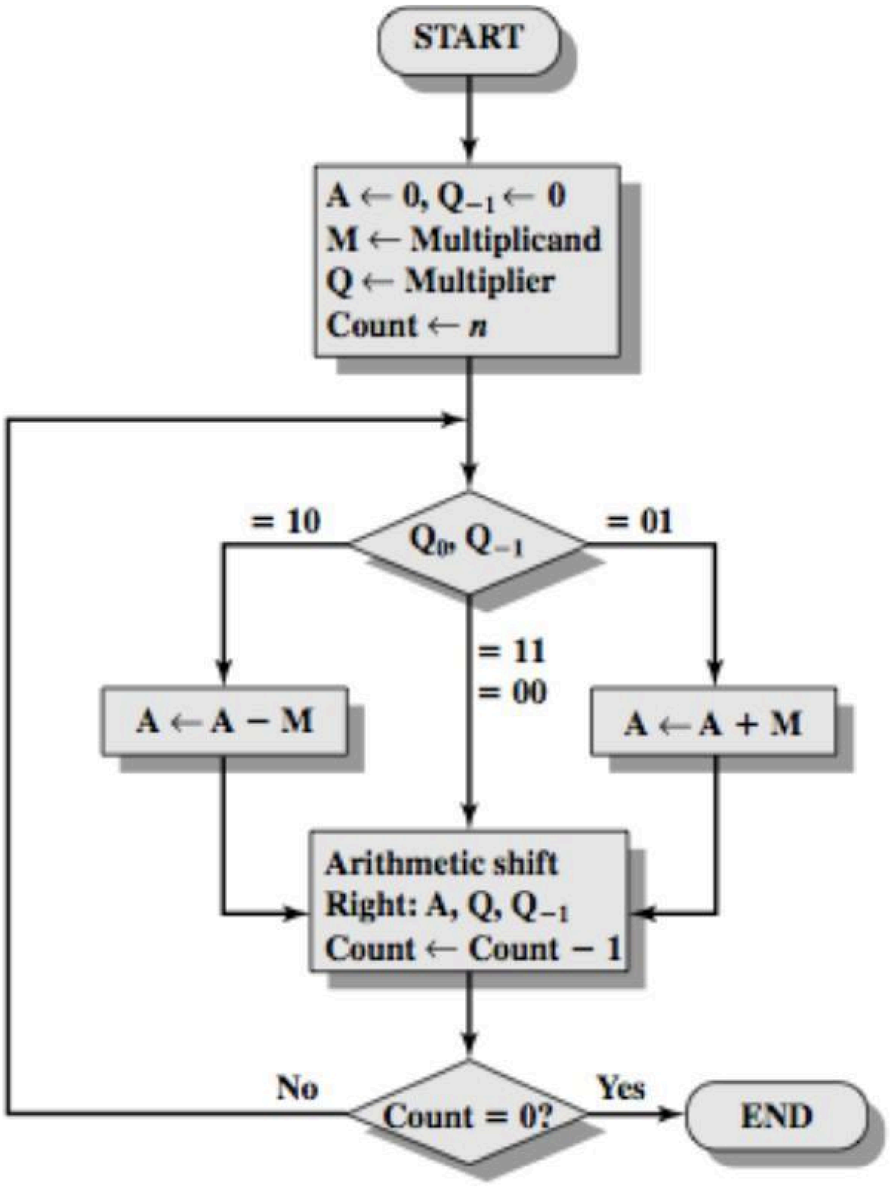
3) If the transition is from **"1 to 0"** then **ADD M** to **A** and **Right-Shift**.

4) If the transition is from **"0 to 0"** then **simply Right-Shift**.

5) If the transition is from **"1 to 1"** then **simply Right-Shift**. **Repeat** steps 1 to 5 for **all bits** of the multiplier.

The **final answer** will be in **A & Q** combined.

Flowchart for Booth's Algorithm:



Example: -9x10=-90

Multiplicand (M): **-9 = 10111**   **9 = 01001**. (Two's Complement Form)

Multiplier (Q): **10 = 01010**.   **-10 = 10110** (Two's Complement Form)

step	**A** **Accumulator**	**Q** **Multiplier**	**Q(-1)**	**M** **Multiplicand**
Initial	**00000**	**01010**	**0**	**10111**
1) (0 ç 0) No Add or Sub Right-Shift	**00000** **00000**	**01010** **00101**	**0** **0**	

2) (1 ç 0)	**01001**	**00101**	**0**	

Perform **(A - M)** Right-Shift	**00100**	**10010**	**1**	
3) (0 ç 1) Perform **(A + M)** Right-Shift	**11011** **11101**	**10010** **11001**	**1** **0**	
4) (1 ç 0) Perform **(A - M)** Right-Shift	**00110** **00011**	**11001** **01100**	**0** **1**	
5) (0 ç 1) Perform **(A + M)** Right-Shift	**11010** **11101**	**01100** **00110**	**1** **0**	

<u>Restoring and Non-Restoring Division</u>:

<u>Non Restoring Division:</u>

1) Let Q register hold the divided, M register holds the divisor and A register is 0.
2) On completion of the algorithm, Q will get the quotient and A will get the remainder.

<u>Algorithm:</u>

The number of steps required is equal to the number of bits in the Dividend.
1) At each step, left shift the dividend by 1 position.
2) Subtract the divisor from A (perform A - M).
3) If the result is positive then the step is said to be "Successful". In this case quotient bit will be "1" and Restoration is NOT Required. The Next Step will also be Subtraction.
4) If the result is negative then the step is said to be "Unsuccessful". In this case quotient bit will be "0". Here Restoration is NOT Performed. Instead, the next step will be ADDITION in place of subtraction.

As restoration is not performed, the method is called Non-Restoring Division.
Repeat steps 1 to 4 for all bits of the Dividend.

Example: (7) / (5)

Dividend (Q) = 7

Divisor (M) = 5

Accumulator (A) = 0

7 = 0111 **5** = 0101

-7 = 1001**-5** = 1011

	Accumulator **A(0)**	**Dividend** **Q(7)**	**Divisor** **M(5)**
Initial Values	**0000**	**0111**	**0101**
Step 1: Left shift A-M Unsuccessful(-ve) Next step: Add	0000 +1011 **1011**	**111_** **1110**	
Step 2: Left shift A+M Unsuccessful(-ve) Next step: Add	0111 +0101 **1100**	**110_** **1100**	
Step 3: Left shift A+M Unsuccessful(-ve) Next step: Add	1001 +0101 **1110**	**100_** **1000**	
Step 4: Left shift A+M successful(+ve)	1101 +0101 **0010**	**000_** **0001**	
	Remainder:2	**Quotient:1**	

RESTORING DIVISION (For unsigned Numbers)

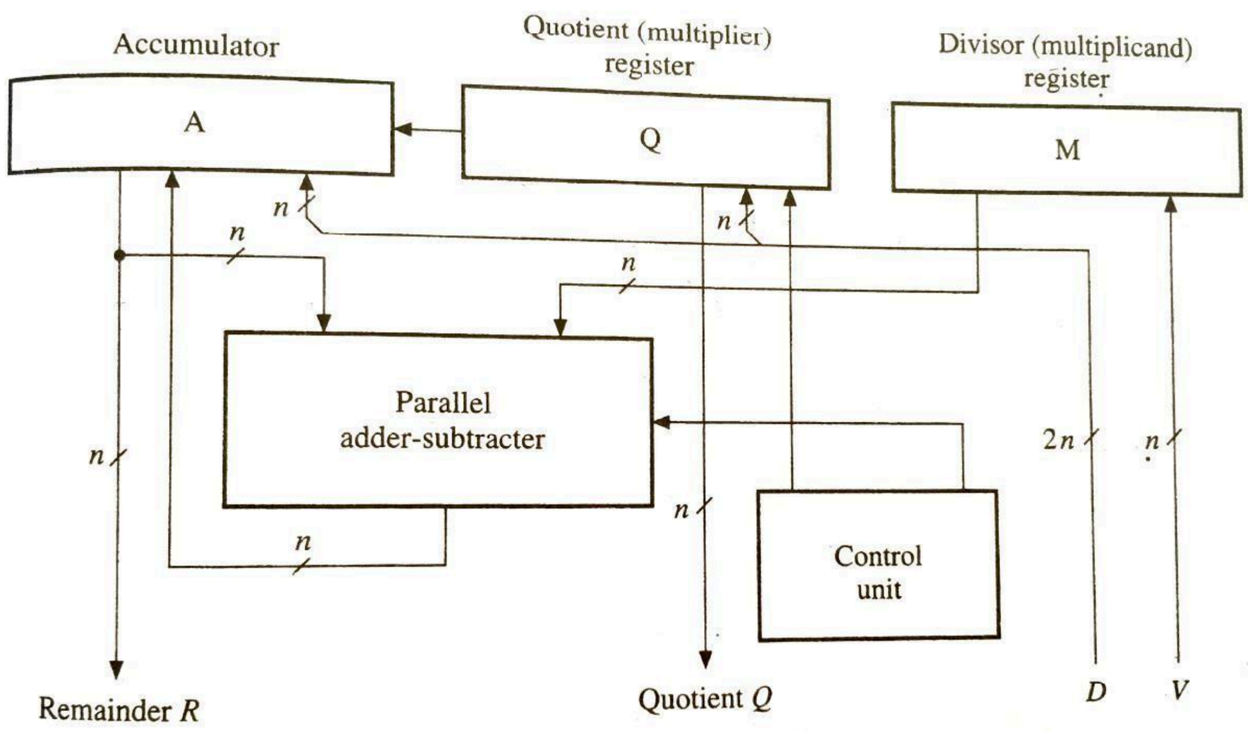
1) Let Q register hold the divided, M register holds the divisor and A register is 0.
2) On completion of the algorithm, Q will get the quotient and A will get theremainder.

Algorithm:

The number of steps required is equal to the number of bits in the Dividend.

1) At each step, left shift the dividend by 1 position.
2) Subtract the divisor from A (perform A - M).
3) If the result is positive then the step is said to be “Successful”.In this case quotient bit will be “1” and Restoration is NOT Required.
4) If the result is negative then the step is said to be “Unsuccessful”.In this case quotient bit will be “0”.Here Restoration is performed by adding back the divisor.

Hence the method is called Restoring Division.Repeat steps 1 to 4 for all bits of the Dividend.


Accumulator
Quotient (multiplier) register
Divisor (multiplicand) register
A
Q
M
n
Parallel adder-subtracter
2n
Control unit
Remainder R
Quotient Q
D
V

Example: (6) / (4)

Dividend (Q) = 6

Divisor (M) = 4

Accumulator (A) = 0

6 = 0110 4 = 0100

-6 = 1010 -4 = 1100

	Accumulator **A(0)**	**Dividend** **Q(6)**	**Divisor** **M(4)**
Initial Values	0000	0110	0100
Step 1: Left shift A-M Unsuccessful(-ve) Restoration:	0000 + 1100 **1100** 0000	110_ 1100	
Step 2: Left shift A-M Unsuccessful(-ve) Restoration:	0001 +1100 1101 0001	100_ 1000	
Step 3: Left shift A-M Unsuccessful(-ve) Restoration:	0011 +1100 1111 0011	000_ 0000	

Step 3: Left shift A-M Successful(+ve) No Restoration	0110 +1100 0010	000_ 0001	
	Remainder(2)	Quotient(1)	

RESTORING DIVISION FOR SIGNED NUMBERS:

1) Let M register hold the divisor, Q register hold the divided.
2) A register should be the signed extension of Q.
3) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:
The number of steps required is equal to the number of bits in the Dividend.
1) At each step, left shift the dividend by 1 position.
2) If Sign of A and M is the same then Subtract the divisor from A (perform A - M), Else Add M to A
3) After the operation,If Sign of A remains the same or the dividend (in A and Q) becomes zero,then the step is said to be "Successful".In this case quotient bit will be "1" and Restoration is NOT Required.
4) If Sign of A changes, then the step is said to be "Unsuccessful".In this case quotient bit will be "0".Here Restoration is Performed.Hence, the method is called Restoring Division.Repeat steps 1 to 4 for all bits of the Dividend.

Note: The result of this algorithm is such that, the quotient will always bepositive and the remainder will get the same sign as the dividend.
Example: (-19) / (7)
19 = 010011 7 = 000111
-19 = 101101 -7 = 111001

	Accumulator A(Sign Extension)	**Dividend Q(-19)**	**Divisor M(7)**
Initial Values	111111	101101	000111
Step 1: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111111 + 000111 000110 111111	01101_ 011010	
Step 2: Left-shift	111110	11010_	

Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	+ 000111 000101 111110	110100	
Step 3: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111101 + 000111 000100 111101	10100_ 101000	
Step 4: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111011 + 000111 000010 111011	01000_ 010000	
Step 5: Left-shift Sign(A,M) Different: A+M Sign still same: Successful Restoration not required	110110 + 000111 111101 111101	10000_ 100001	
Step 6: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111011 + 000111 000010 111011	00001_ 000010	
	Remainder(-5)	Quotient(2)	

Register Transfer:

Computer registers are denoted by capital letters (sometimes followed by numerals) to denote the function of the register. The register that holds an address for the memory unit is usually called a **memory address register** and is denoted by **MAR**. Other registers are PC (for program counter), IR (for instruction register, and R1 (for processor register). An n-bit register is sequence of n-flipflops numbered from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left.

The most common way to represent a register is by a rectangular box with the name of the register inside, as shown in the figure below. The individual bits can be distinguished as shown in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H(for high byte). The name of the 16-bit register is PC. The symbol PC (0-7) or PC (L) refers to the low-order byte and PC(8-15) or PC(H) to the high-order byte.

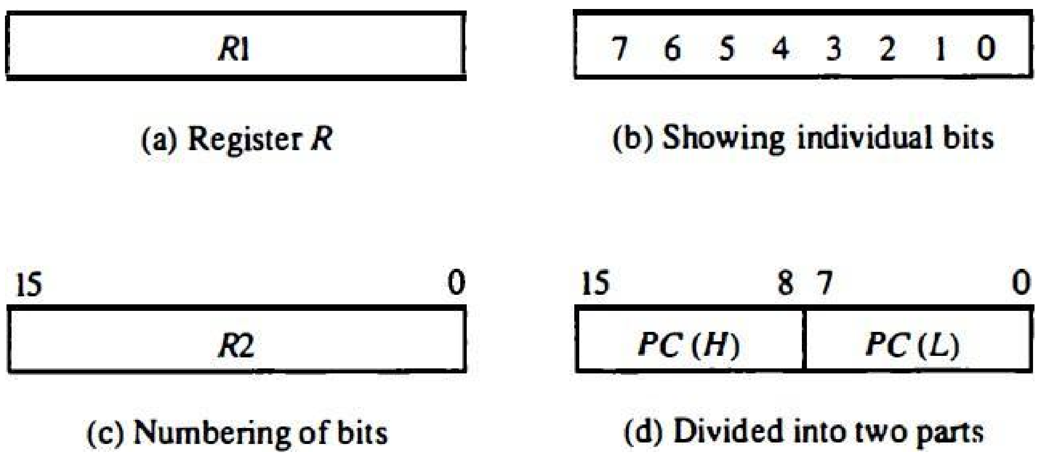


Fig: Block diagram of registers

Information transfer from one register to another is designated in symbolicform by means of a replacement operator as shown below, which denotes a transfer of the contents of register R1 into register R2.Contents of R2 are replaced by the contents of R1.By definition, thecontent of the source register R1 does not change after the transfer.register transfer implies that circuits areavailable from the outputs of the source register to the inputs of the destinationregister.

R2 <--R1

Sometimes, we may want the transfer to occur only under a predetermined controlcondition. This can be shown by means of an if-then statement

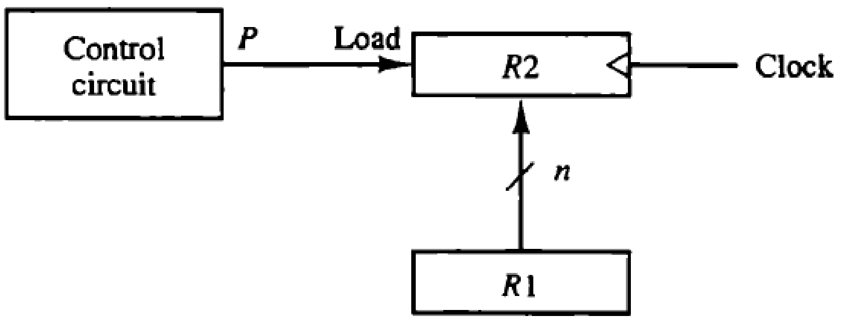
If (P = 1) then (R2 <--R1)

where P is a control signal generated in the control section.A control function is a Boolean variable that isequal to 1 or 0. The control function is included in the statement as follows

P: R2 <--R1

The control condition is terminated with a colon. It symbolizes the requirementthat the transfer operation be executed by the hardware only if P = 1.

Every statement written in a register transfer notation implies a hardwareconstruction for implementing the transfer. Figure below shows the block diagramthat depicts the transfer from R1 to R2. The n outputs of register R1 areconnected to the n inputs of register R2. The letter n will be used to indicateany number of bits for the register. It will be replaced by an actual numberwhen the length of the register is known. Register R2 has a load input that isactivated by the control variable P. It is assumed that the control variable issynchronized with the same clock as the one applied to the register.



In the timing diagram below, P is activated in the control section by the rising edgeof a clock pulse at time t . The next positive transition of the clock at time t + 1finds the load input active and the data inputs of R2 are then loaded into theregister in parallel. P may go back to 0 at time t + 1; otherwise, the transferwill occur with every clock pulse transition while P remains active.

Note:Even though the control condition such as P becomes active just after time t,the actual transfer does not occur until the register is triggered by the nextpositive transition of the clock at time t + 1 .

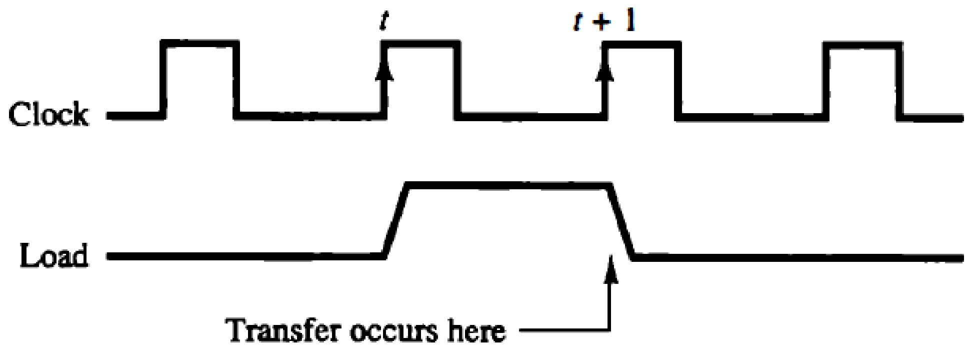


Fig: Timing Diagram

Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time.

The statement

T: R2 <- R1, R1 <- R2

It denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1.

The basic symbols of the register transfer notation are given below:

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	$R2 \leftarrow R1$
Comma ,	Separates two microoperations	$R2 \leftarrow R1, R1 \leftarrow R2$

Fig: Basic symbols of register Transfer

<u>Memory Transfer:</u>

The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation. A memory word will be symbolized by the letter M.

The particular memory word among the many available is selected by thememory address during the transfer. It is necessary to specify the address ofM when writing memory transfer operations. This will be done by enclosingthe address in square brackets following the letter M.

Memory Read: Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

Read: DR <- M [AR]

This causes a transfer of information into DR from the memory word M selected by the address in AR.

Memory Write: The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data is in register R1 and the address in AR. The write operation can be stated symbolically as follows:

Write: M[AR] ← R1

This causes the transfer of information from R1 into the memory word M selected by the address in AR.

<u>Instruction cycle:</u>

In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.

2. Decode the instruction.

3. Read the effective address from memory if the instruction has an indirect address.

4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

FETCH AND DECODE: Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal To. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2, and so on.

The Micro-operations for the fetch and decode phases can be specified by the following register transfer statements:

T_0: AR ← PC

The address from PC to AR during the clock transition associated with timing signal T0.

T_1: IR ←M[AR], PC ←PC + 1

The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program

T_2: D0, …, D7 ← Decode IR(12-14),AR ← IR(0-11), I← IR(l5)

At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR.

Decoding: The timing signal that is active after the decoding is T3. During time T3, the control unit determines the type of instruction that was just read from memory. Decoder output D7, is equal to 1 if the operation code is equal to binary 111. If D7 = 1, the instruction must be a register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 000 through 110,specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If D7 = 0 and I = 1, we have a memoryreferenceinstruction with an indirect address.The microoperation for the indirect addresscondition can be symbolized by the register transfer statement:

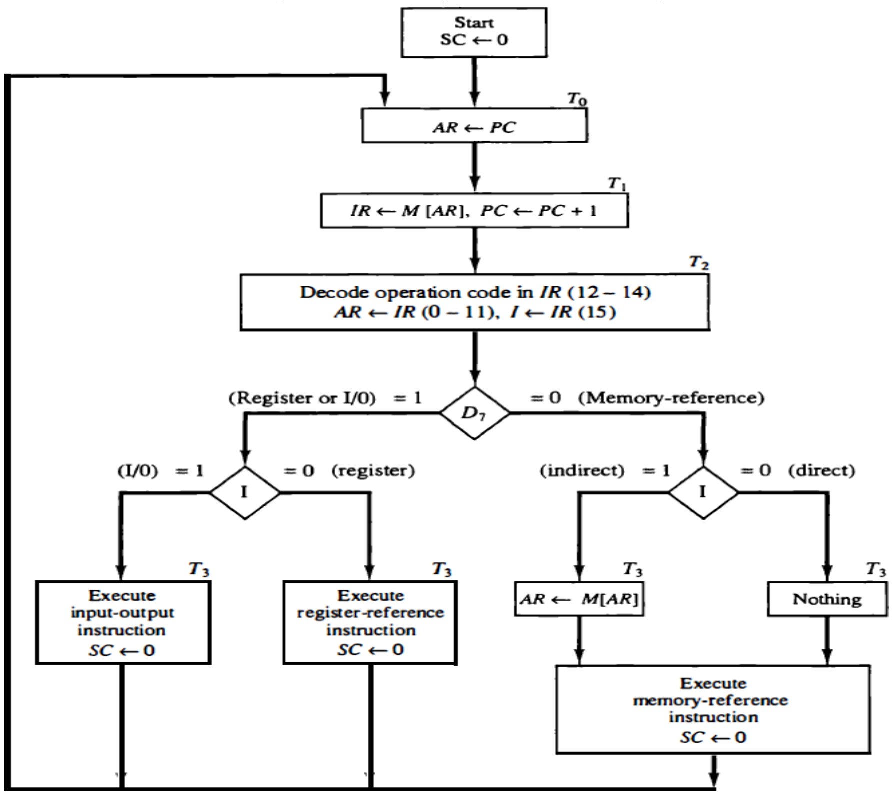
AR M [AR]

$D_7' I T_3$: $AR \leftarrow M[AR]$
$D_7' I' T_3$: Nothing
$D_7 I' T_3$: Execute a register-reference instruction
$D_7 I T_3$: Execute an input–output instruction

When a **memory-reference instruction** with I = 0 is encountered, it is notnecessary to do anything since the effective address is already in AR. However,the sequence counter SC must be incrementedso that theexecution of the memory-reference instruction can be continued with timingvariable T4.After the instruction is executed,SC is cleared to 0 and control returns to the fetch phase with T0 = 1 .

Register-reference instructions are recognized by the control when 07 = 1 andI = 0.The 12 bitsavailable in IR(0-11) are transferred to AR during time T2.These instructions are executed with the clocktransition

Fig:Flowchart for instruction cycle


Start
SC ← 0
T0
AR ← PC
T1
IR ← M [AR], PC ← PC + 1
T2
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
Execute memory-reference instruction SC ← 0

associated with timing variable T3.The execution of a register-reference instruction is completedat time T3.The sequence counter SC is cleared to 0 and the control goesback to fetch the next instruction with timing signal T0.

Arithmetic instructions :The four basic arithmetic operations are addition, subtraction, multiplication,and division. Most computers provide instructions for all four operations.Some small computers have only addition and possibly subtraction instructions.

A list of typical arithmetic instructions is given in Table given below:

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

The **increment instruction** adds 1 to the value stored in a register or memory word.The **decrement instruction** subtracts 1 from a value stored in a registeror memory word.The add, subtract, multiply, and divide instructions may be available fordifferent types of data. The data type assumed ·to be in processor registersduring the execution of these arithmetic operations is included in the definitionof the operation code. An arithmetic instruction may specify fixed-point orfloating-point data, binary or decimal data, single-precision or double-precision data.

The mnemonics for three add instructions that specifydifferent data types are shown below:

ADDI Add two binary integer numbers

ADDF Add two floating-point numbers

ADDD Add two decimal numbers in BCD

The instruction "**add with carry**" performs the addition on two operands plus the value of the carry fromthe previous computation. Similarly, the "**subtract with borrow**" instructionsubtracts two words and a borrow which may have resulted from a previoussubtract operation. The **negate instruction** forms the 2' s complement of anumber, effectively reversing the sign of an integer when represented in thesigned-2's complement form.

<u>Logical and Bit Manipulation Instructions:</u>Logical instructions **perform binary operations on strings of bits** stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The **logical instructions consider each bit of the operand separately** and treat it as a Boolean variable. By proper application of the logical instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

Some logical and bit manipulation instructions are shown in the figure below:

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

The clear instruction causes the specified operand to be replaced by D's.The complement instruction produces the 1's complement by inverting all thebits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands. Although theyperform Boolean operations, when used in computer instructions, the logicalinstructions should be considered as performing bit manipulation operations.There are three-bit manipulation operations possible: a selected bit can becleared to 0, or can be set to 1, or can be complemented. The three logicalinstructions are usually applied to do just that.

<u>**Shift Instructions**</u>: Shifts are operations in which the bits of a word are moved to the left or right. Shift instructions may specify either logicalshifts, arithmetic shifts, or rotate-type operations. In either case the shift maybe to the right or to the left.Table below lists four types of shift instructions:

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

The **logical shift** inserts 0to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.

The **arithmetic shift-right** instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus, a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

<u>**Program Control Instructions:**</u> Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter.

Some program control instructions are listed in Table below:

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP

Branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. Branch instruction is written as **BR ADR**, where ADR is a symbolic name for an address. Branch and jump instructions may be conditional or unconditional. Anunconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a conditionsuch as branch if positive or branch if zero. If the condition is met, the programcounter is loaded with the branch address and the next instruction is taken.from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

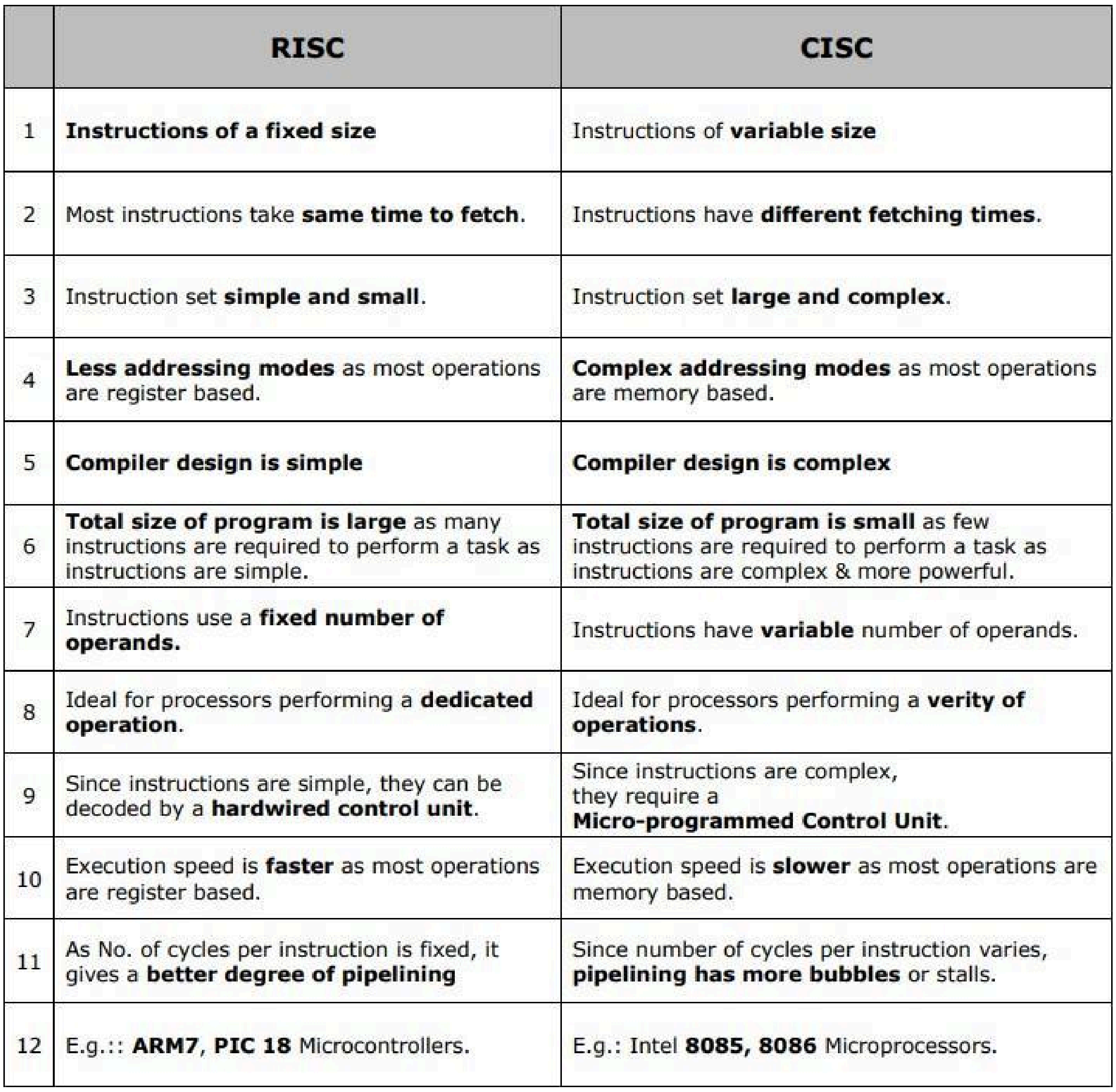
The **skip instruction** does not need an address field and is therefore azero-address instruction. A conditional skip instruction will skip the nextinstruction if the condition is met. If the condition is not met, control proceeds with thenext instruction in sequence.

The **call and return** instructions are used in conjunction with subroutines.

The **compare instruction** performs a subtraction between two operands, but the result of the operation isnot retained. However, certain status bit conditions are set as a result of theoperation. Similarly, the **test instruction** performs the logical AND of twooperands and updates certain status bits without retaining the result or changing the operands. (***Note:***The compare and test instructions do not change the program sequence directly. They are listed in Table because of their application in setting conditions for subsequent conditional branch instructions)

RISC vs CISC Architecture

RISC stands for Reduced Instruction set Computer and **CISC** stands for Complex Instruction Set Computer.RISC and CISC are the two ideologies behind making the processor.



	RISC	CISC
1	**Instructions of a fixed size**	Instructions of **variable size**
2	Most instructions take **same time to fetch**.	Instructions have **different fetching times**.
3	Instruction set **simple and small**.	Instruction set **large and complex**.
4	**Less addressing modes** as most operations are register based.	**Complex addressing modes** as most operations are memory based.
5	**Compiler design is simple**	**Compiler design is complex**
6	**Total size of program is large** as many instructions are required to perform a task as instructions are simple.	**Total size of program is small** as few instructions are required to perform a task as instructions are complex & more powerful.
7	Instructions use a **fixed number of operands.**	Instructions have **variable** number of operands.
8	Ideal for processors performing a **dedicated operation**.	Ideal for processors performing a **verity of operations**.
9	Since instructions are simple, they can be decoded by a **hardwired control unit**.	Since instructions are complex, they require a **Micro-programmed Control Unit**.
10	Execution speed is **faster** as most operations are register based.	Execution speed is **slower** as most operations are memory based.
11	As No. of cycles per instruction is fixed, it gives a **better degree of pipelining**	Since number of cycles per instruction varies, **pipelining has more bubbles** or stalls.
12	E.g.:: **ARM7**, **PIC 18** Microcontrollers.	E.g.: Intel **8085, 8086** Microprocessors.

13 They have **register based operations**. 13. **Memory based** operations.

CPU CONTROL UNIT DESIGN

- **Hardwired CU :**
  In Hardwired CU, control signals are produced by hardware. There are three types of Hardwired Control Units
  1) STATE TABLE METHOD
  2) DELAY ELEMENT METHOD
  3) SEQUENCE COUNTER METHOD

  STATE TABLE METHOD:
  1) It is the most basic type of hardwired control unit.
  2) Here the behaviour of the control unit is represented in the form of a table called the state table.
  3) The rows represent the T-states and the columns indicate the instructions.
  4) Each intersection indicates the control signal to be produced, in the corresponding T-state of everyinstruction.
  4) A circuit is then constructed based on every column of this table, for each instruction.

T-STATES	INSTRUCTIONS			
	I_1	I_2	...	I_N
T_1	$Z_{1,1}$	$Z_{1,2}$	...	$Z_{1,N}$
T_2	$Z_{2,1}$	$Z_{2,2}$	...	$Z_{2,N}$
...	...	...	...	...
T_M	$Z_{M,1}$	$Z_{M,2}$	...	$Z_{M,N}$

$Z_{1,1}$: Control Signal to be produced in T-state (T_1) of Instruction (I_1)

ADVANTAGE: It is the simplest method and is ideally suited for very small instruction sets.

DRAWBACK: As the number of instructions increase, the circuit becomes bigger and hence more complicated. As a tabular approach is used, instead of a logical approach (flowchart), there are duplications of manycircuit elements in various instructions.

Delay Element Method:

1) Here the behaviour of the control unit is represented in the form of a flowchart.

2) Each step in the flowchart represents a control signal to be produced.

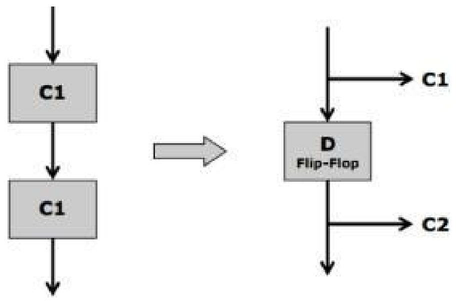
3) Once all steps of a particular instruction, are performed, the complete instruction gets executed.

4) Control signals perform Micro-Operations, which require one T-states each.

5) Hence between every two steps of the flowchart, there must be a delay element.

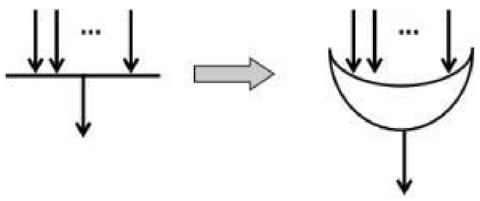
6) The delay must be exactly of one T-state. This delay is achieved by D Flip-Flops.

7) These D Flip-Flops are inserted between every two consecutive control signals.

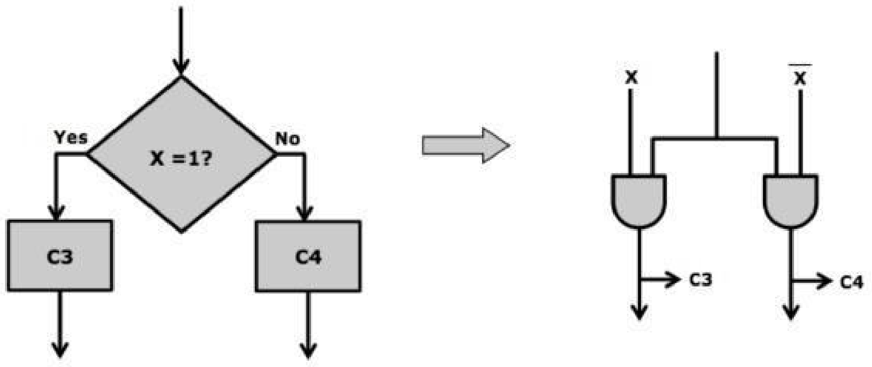


8) Of all D Flip-Flops only one will be active at a time. So the method is also called “One Hot Method”.

9) In a multiple entry point, to combine two or more paths, we use an OR gate.



10) A decision box is replaced by a set of two complementing AND gates



11) A multiple entry point is substituted by an OR

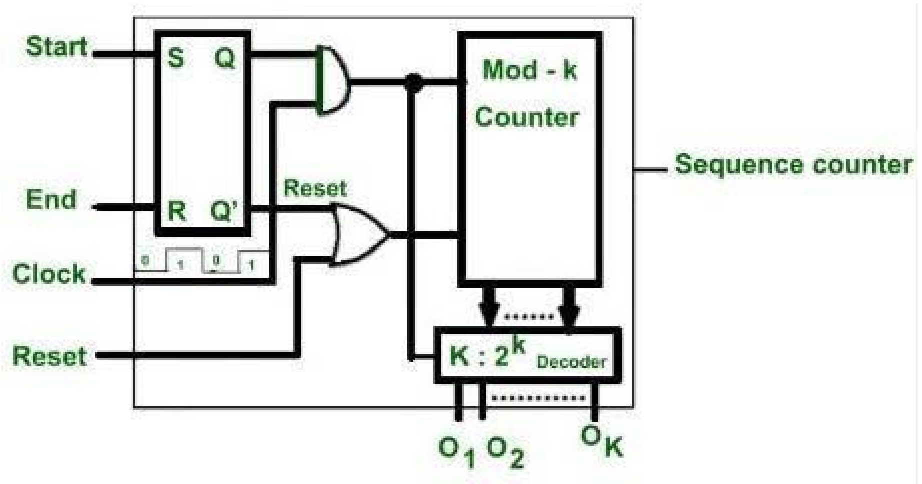
gate. ADVANTAGE:

As the method has a logical approach, it can reduce the circuit complexity.This is done by re-utilizing common elements between various instructions.

DRAWBACK:

As the no of instructions increase, the number of D Flip-Flops increase, so the cost increases.Moreover, only one of those D Flip-Flops are actually active at a time.

SEQUENCE COUNTER METHOD:



1) This is the most popular form of hardwired control unit. The goal of this circuit is to provide triggers to different parts of the circuit after gaps of 1-Tstate.

2) It follows the same logical approach of a flowchart, like the Delay element method, but does not use all those unnecessary D Flip-Flops because at any point of time only one delay element is active and a complex circuitry would involve many delay elements which is very inefficient. The D-Flip-flops are replaced by trigger points which are activated after gaps of one T-state.

Following are the steps involved in designing a CU using Sequence Counter Method.

1) First a flowchart is made representing the behaviour of a control unit.

2) It is then converted into a circuit using the same principle of AND & OR gates.

3) We need a delay of 1 T-state (one clock cycle) between every two consecutive control signals.

4) That is achieved by the above circuit.

5) If there are “k” number of distinct steps producing control signals, we employ a “mod k” and “k”output decoder.

6) The counter will start counting at the beginning of the instruction.

7) The “clock” input via an AND gate ensures each count will be generated after 1 T-state.

8) The count is given to the decoder which triggers the generation of “k” control signals, each aftera delay of 1 T-state.

9) When the instruction ends, the counter is reset so that next time, it begins from the first

count. ADVANTAGE:

Avoids the use of too many D Flip-Flops.

GENERAL DRAWBACKS OF A HARDWIRED CONTROL UNIT

1) Since they are based on hardware, as the instruction set increases, the circuit becomes more and more complex. For modern processors having hundreds of instructions, it is virtually impossible to create Hardwired Control Units.

2) Such large circuits are very difficult to debug.

3) As the processor gets upgraded, the entire Control Unit has to be redesigned, due to the rigid nature of hardware design.

Microprogrammed CU

<u>WILKES’ DESIGN FOR A MICROPROGRAMMED CONTROL UNIT:</u>

1) Microprogrammed Control Unit produces control signals by software, using micro-instructions

2) A program is a set of instructions.

3) An instruction requires a set of Micro-Operations.

4) Micro-Operations are performed by control signals.

5) Instead of generating these control signals by hardware, we use micro-instructions. This means every instruction requires a set of micro-instructions This is called its micro-program.

6) Microprograms for all instructions are stored in a small memory called “Control Memory”. The Control memory is present inside the processor.

7) Consider an Instruction that is fetched from the main memory into the Instruction Register (IR).

8) The processor uses its unique “opcode” to identify the address of the first micro-instruction. That address is loaded into CMAR (Control Memory Address Register). CMAR passes the address to the decoder.

9) The decoder identifies the corresponding micro-instruction from the Control Memory.

10) A micro-instruction has two fields: a control filed and an address field.

Control field: Indicates the control signals to be generated.

Address field: Indicates the address of the next micro-instruction.

11) This address is further loaded into CMAR to fetch the next micro-instruction.

12) For a conditional micro-instruction, there are two address fields.This is because, the address of the next micro-instruction depends on the condition.The condition (true or false) is decided by the appropriate control flag.

13) The control memory is usually implemented using FLASH ROM as it is writable yet non-volatile.

ADVANTAGES

1) The biggest advantage is flexibility.

2) Any change in the control unit can be performed by simply changing the micro-instruction.

3) This makes modifications and up gradation of the Control Unit very easy.

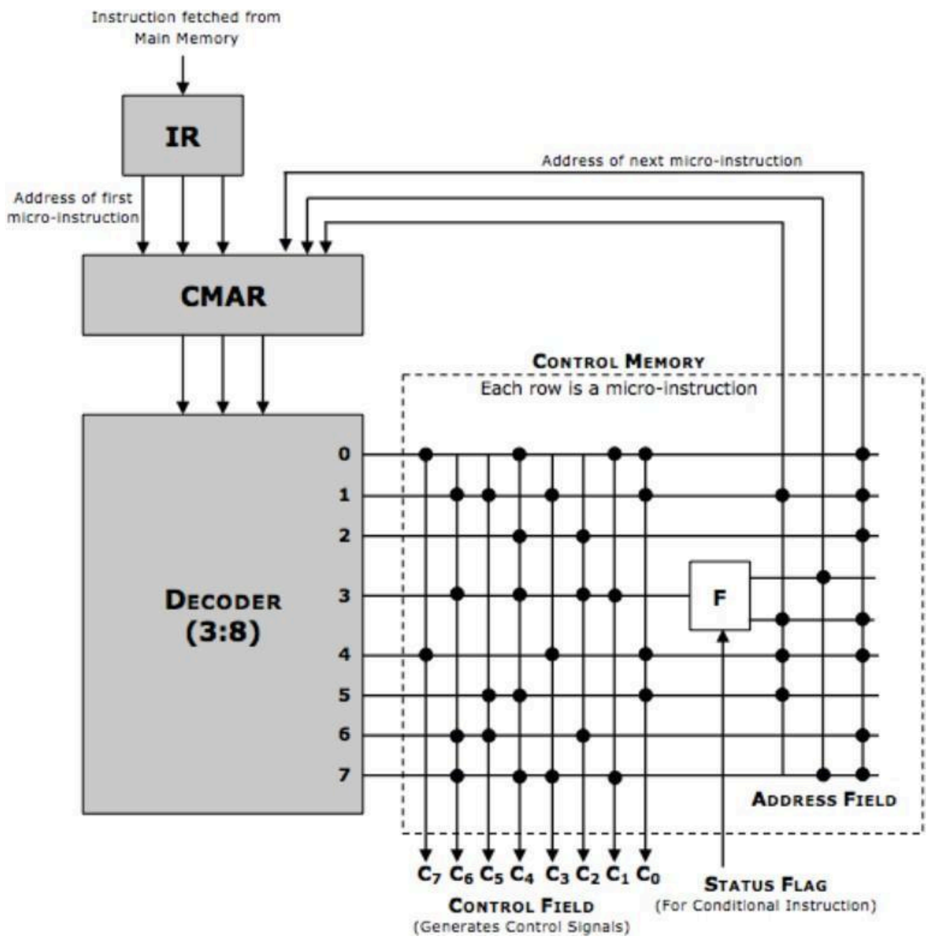
4) Moreover, software can be much easily debugged as compared to a large Hardwired Control Unit.

DRAWBACKS

1) Control memory has to be present inside the processor, increasing its size.

2) This also increases the cost of the processor.

3) The address field in every micro-instruction adds more space to the control memory. This can beeasily avoided by proper micro-instruction sequencing.


Instruction fetched from Main Memory
IR
Address of first micro-instruction
Address of next micro-instruction
CMAR
Control Memory
Each row is a micro-instruction
Decoder (3:8)
0
1
2
3
4
5
6
7
F
Address Field
C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0
Control Field
(Generates Control Signals)
Status Flag
(For Conditional Instruction)

<u>TYPICAL MICROPROGRAMMED CONTROL UNIT</u>

1) Microprogrammed Control Unit produces control signals by software, using micro-instructions.

2) A program is a set of instructions.

3) An instruction requires a set of Micro-Operations.

4) Micro-Operations are performed by control signals.

5) Here, these control signals are generated using micro-instructions.

6) This means every instruction requires a set of micro-instructions

7) This is called its micro-program.

8) Microprograms for all instructions are stored in a small memory called “Control Memory”.

9) The Control memory is present inside the processor.

10) Consider an Instruction that is fetched from the main memory into the Instruction Register (IR).

11) The processor uses its unique “opcode” to identify the address of the first micro-instruction.

12) That address is loaded into CMAR (Control Memory Address Register) also called μIR.

13) This address is decoded to identify the corresponding μ-instruction from the Control Memory.

14) There is a big improvement over Wilkes’ design, to reduce the size of micro-instructions.

15) Most micro-instructions will only have a Control field.

16) The Control field Indicates the control signals to be generated.

17) Most micro-instructions will not have an address field.

18) Instead, μPC will simply get incremented after every micro-instruction.

19) This is as long as the μ-program is executed sequentially.

20) If there is a branch μ-instruction only then there will be an address filed.

21) If the branch is unconditional, the branch address will be directly loaded into CMAR.

22) For Conditional branches, the Branch condition will check the appropriate flag.

23) This is done using a MUX which has all flag inputs.

24) If the condition is true, then the MUX will inform CMAR to load the branch address.

25) If the condition is false CMAR will simply get incremented.

26) The control memory is usually implemented using FLASH ROM as it is writable yet non-volatile.

ADVANTAGES

1) The biggest advantage is flexibility.

2) Any change in the control unit can be performed by simply changing the micro-instruction.

3) This makes modifications and up gradation of the Control Unit very easy.

4) Moreover, software can be much easily debugged as compared to a large Hardwired Control Unit.

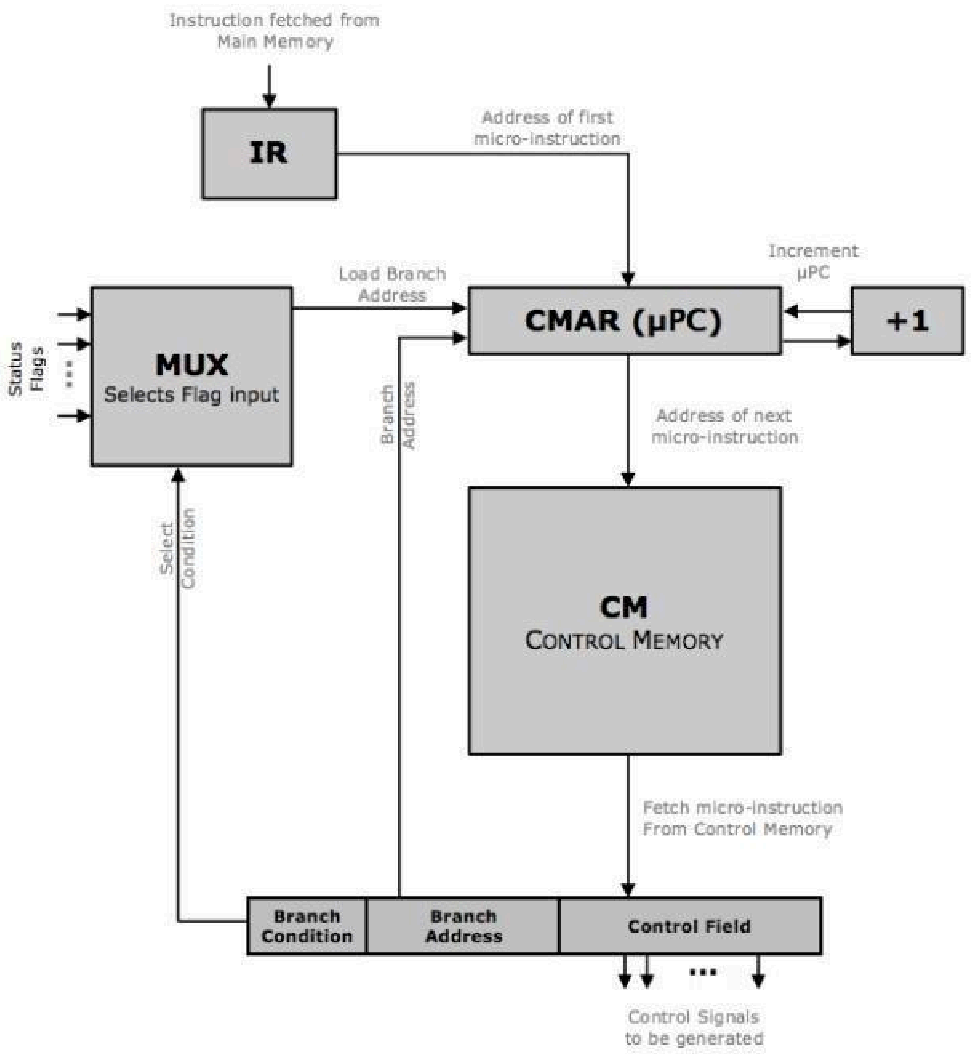
5) Since most micro-instructions are executed sequentially, they don’t need for an address field.

6) This significantly reduces the size of micro-instructions, and hence the Control Memory.

DRAWBACKS

1) Control memory has to be present inside the processor, increasing its size.

2) This also increases the cost of the processor.



Instruction fetched from Main Memory
IR
Address of first micro-instruction
Increment μPC
Load Branch Address
CMAR (μPC)
+1
Status Flags
MUX
Selects Flag input
Branch Address
Address of next micro-instruction
Select Condition
CM
CONTROL MEMORY
Fetch micro-instruction From Control Memory
Branch Condition
Branch Address
Control Field
Control Signals to be generated

Internal Memory Organization:

Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. Apossible organization is illustrated in Figure below:

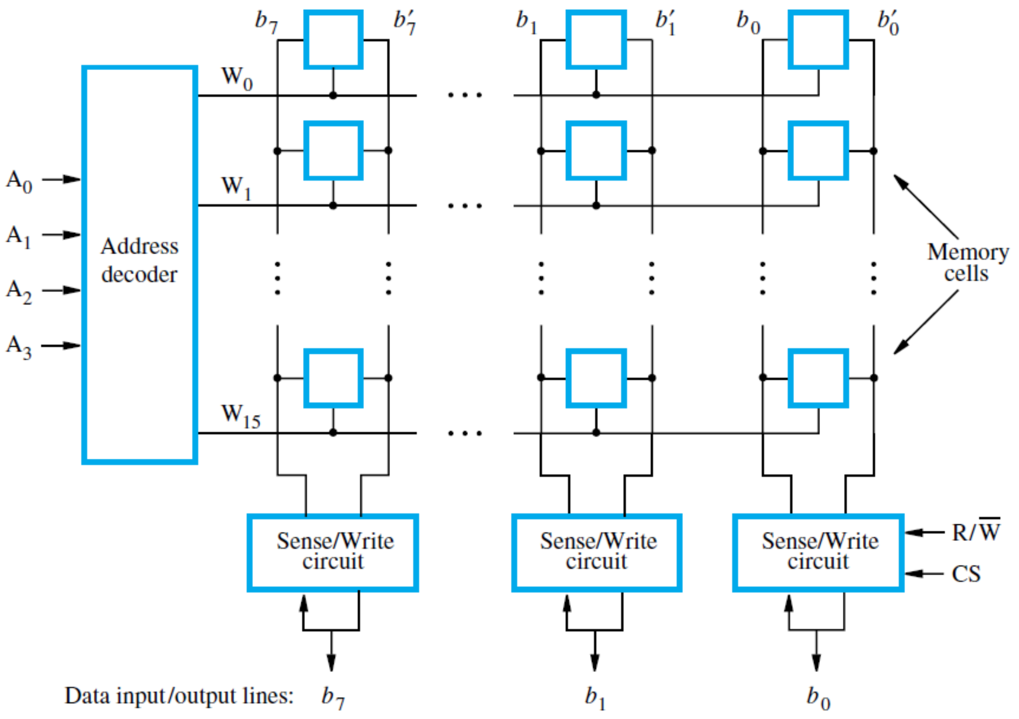


Fig: Organization of bit cells in a memory chip.

Each row of cells constitutes a memory word, and **all cells of a row are connected to a common line referred to as the word line**, which is driven by the address decoder on the chip.The **cells in each column are connected to a Sense/Write circuit by two bit lines**, and the Sense/Writecircuits are connected to the data input/output lines of the chip.**During a Read** operation,these **circuits sense, or read, the information stored in the cells selected by a word line** andplace this information on the output data lines. **During a Write operation, the Sense/Write circuits receive input data and store them in the cells of the selected word**.Above figureis an example of a very small memory circuit consisting of 16 words of 8 bitseach. This is referred to as a 16 × 8 organization. The data input and the data output of eachSense/Write circuit are connected to a single bidirectional data line that can be connectedto the data lines of a computer.Two control lines, R/W and CS, are provided. The R/W(Read/Write) input specifies the required operation, and the CS (Chip Select) input selectsa given chip in a multichip memory system.**The above memory circuit stores 128 bits and requires 14 external connections for address, data, and control lines.** It also needs two lines for power supply and groundconnections.

Consider now a slightly larger memory circuit, one that has 1K (1024) memorycells. This **circuit can be organized as a 128 × 8 memory, requiring a total of 19 external connections**. Alternatively, the same number of cells can be organized into a 1K×1 format.In this case, a 10-bit address is needed, but there is only one data line, resulting in 15 externalconnections. Figure below shows such an organization. The required 10-bit address is dividedinto two groups of 5 bits each to form the row and column addresses for the cell array. Arow address selects a row of 32 cells, all of which are accessed in parallel. But, only oneof these cells is connected to the external data line, based on the column address.For example, a 1Giga-bitchip may have a 256M × 4 organization, in which case a 28-bit address is needed and 4bits are transferred to or from the chip.

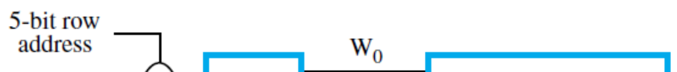


Fig: Organization of a 1K × 1 memory chip.

Memory Hierarchy

- The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of all storage devices employed in a computer systemfrom the slow but high-capacity secondary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic.
- The purpose of any memory device is to store programs and data. Several types of memory devices are used in the computer forming a Memory Hierarchy. Each plays a specific role contributing to the speed, cost effectiveness, portability etc.

Main Memory

The memory unit that communicates directly with the CPU is called the main memory.It comprises of RAM and ROM, both are Semi-Conductor memories. (chip memories). ROM is non-volatile.It is used is storing permanent information like the BIOS program.It is typically of 2 MB - 4 MB in size.RAM is writable and hence is used for day-to-day operations.Every file that we access from secondary memory, is first loaded into RAM.The main memory occupies
a central position by being able to communicate directly with the CPU andwith auxiliary memory devices through an UO processor. To provide large amount of working space RAM is typically 4 GB - 8 GB.

Secondary Memory (Auxiliary Memory):

Devices that provide backup storage arecalled auxiliary memory. The most common auxiliary memory devices used incomputer systems are magnetic disks and tapes. They are used for storingsystemprograms, large data files, and other backup information. Only pro-grams and data currently needed by the processor reside in main memory.When programs notresiding in main memory are needed by the CPU, they are brought in fromauxiliary memory.The main purpose of Secondary Memory is to increase the storage capacity, at low cost.Its biggest component is the Hard Disk.It is writeable as well as non-volatile. Typical size of a HD is 1 TB.Disk memories are much slower than chip memories but are also much cheaper.

Cache Memory:

The cache memory is employed in computersystems to compensate for the speed differential between main memory accesstime and processor logic. CPU logic is usually faster than main memory accesstime, with the result that processing speed is limited primarily by the speedof main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU andmain memory whose access time Is dose to processor logic dock cycle time.It is the fastest form of memory as it uses SRAM (Static RAM).The Main Memory uses DRAM (Dynamic RAM).SRAM uses flip-flops and hence is much faster than DRAM which uses capacitors.But SRAM is also very expensive as compared to DRAM.Hence only the current portion of the file we need to access is copied from Main Memory (DRAM)to Cache memory (SRAM), to be directly accessed by the processor.This gives maximum performance and yet keeps the cost low.While the VO processor manages data transfers between auxiliary memoryand main memory, the cache organization is concerned with the transferof information between main memory and CPU.Typical size of Cache is around 2 MB – 8MB. If code and data are in the same cache then it is unified cache else its called split cache.Depending upon the location of cache, it is of three types: L1, L2 and L3.L1 cache is present inside the processor and is a split cache typically 4-8 KB.L2 is present on the same die as the processor and is a unified cache typically 1

MB.L3 is present outside the processor. It is also unified and is typically of 2-8 MB.

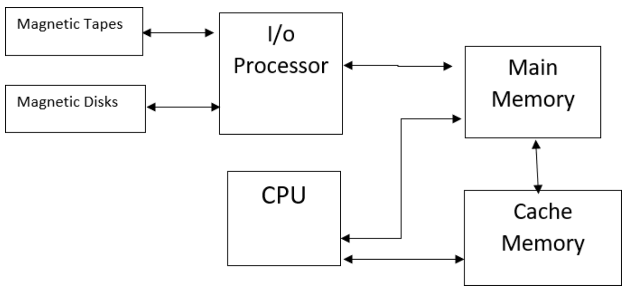


Fig: Memory Hierarchy in a computer system

The reason for having twoor three levels of memory hierarchy is economics. As the storage capacity ofthe memory increases, the cost per bit for storing binary information decreasesand the access time of the memory becomes longer. The auxiliary memory hasa large storage capacity, is relatively inexpensive, but has low access speedcompared to main memory. The cache memory is very small, relatively expensive,and has very high access speed. Thus as the memory access speedincreases, so does its relative cost. The overall goal of using a memory hierarchyis to obtain the highest-possible average access speed while minimizing thetotal cost of the entire memory system.

Memory Interleaving

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments. Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure below shows a memory unit with four modules.

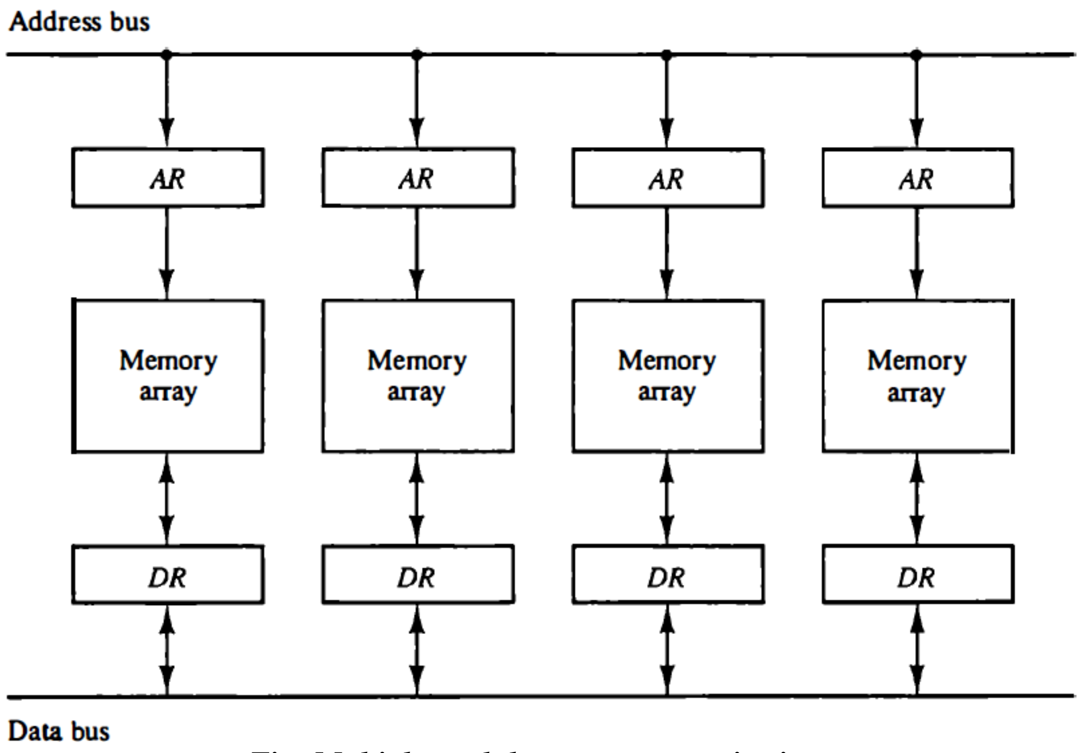


Fig: Multiple module memory organization.

Each memory array has its ownaddress register AR and data register DR . The address registers receive informationfrom a common address bus and the data registers communicate witha bidirectional data bus. The two least significant bits of the address can be usedto distinguish between the four modules. The modular system permits onemodule to initiate a memory access while other modules are in the process ofreading or writing a word and each module can honor a memory requestindependent of the state of the other modules.The advantage of a modular memory is that it allows the use of a techniquecalled interleaving. In an interleaved memory, different sets of addressesare assigned to different memory modules. For example, in a two-modulememory system, the even addresses may be in one module and the oddaddresses in the other. When the number of modules is a power of 2, the leastsignificant bits of the address select a memory module and the remaining bitsdesignate the specific location to be accessed within the selected module.A modular memory is useful in systems with pipeline and vector processing.A vector processor that uses an n-way interleaved memory can fetch noperands from n different modules. By staggering the memory access, theeffective memory cycle time can be reduced by a factor close to the number ofmodules. A CPU with instruction pipeline can take advantage of multiplememory modules so that each segment in the pipeline can access memoryindependent of memory access from other segments.

Cache Memory:

The cache is a small and very fast memory, interposed between the processor and the mainmemory. Its purpose is to make the main memory appear to the processor to be muchfaster than it actually is.

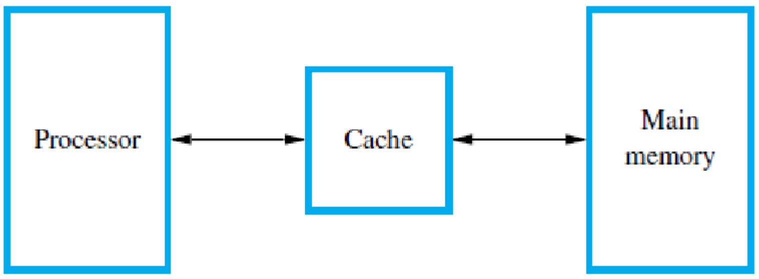


Fig: Use of Cache Memory

operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference. Temporal locality suggests that whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon. Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as well. The termcache block refers to a set of contiguous address locations of some size. Another term that is often used to refer to a cache block is a cache line.

When the processor issues a Read request, the contents of a block of memory words containing the location specified are transferred into the cache. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function. When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache's replacement algorithm.

Cache Hits:

The processor does not need to know explicitly about the existence of the cache. It simply issues Read andWrite requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a read or write hit is said to have

occurred. The main memory is not involved when there is a cache hit in a Read operation. For a Write operation, the system can proceed in one of two ways. In the first technique, called the write-through protocol, both the cache location and the main memory location are updated. The second technique is to update only the cache location and to mark the block containing it with an associated flag bit, often called the dirty or modified bit. The main memory location of the word is updated later, when the block containing this marked word is removed from the cache to make room fora new block. This technique is known as the write-back, or copy-back, protocol.
The write-through protocol is simpler than the write-back protocol, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. The write-back protocol also involves unnecessary Write operations, because all words of the block are eventually written back, even if only a single word has been changed while the block was in the cache. The write-back protocol is used most often, to take advantage of the high speed with which data blocks can be transferred to memory chips.

Cache Misses

A Read operation for a word that is not in the cache constitutes a Read miss. It causes the block of words containing the requested word to be copied from the main memory into the cache. After the entire block is loaded into the cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from the main memory. The latter approach, which is called load-through, or early restart, reduces the processor's waiting time somewhat, at the expense of more complex circuitry.
When a Write miss occurs in a computer that uses the write-through protocol, the information is written directly into the main memory. For the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.

Pipelining and Parallel Processors

Basic Concepts of Pipelining :

Introduction:

1. Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
2. A pipeline can be visualized as a collection of processing segments through which binary information flows.
3. Each segment performs partial processing dictated by the way the task is partitioned.
4. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
5. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Pipeline organization is demonstrated by means of a simple example.

Suppose that we want to perform the combined multiply and add operations with a stream of numbers. Ai* Bi + Ci for i = 1, 2, 3, . . . , 7 Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig below:

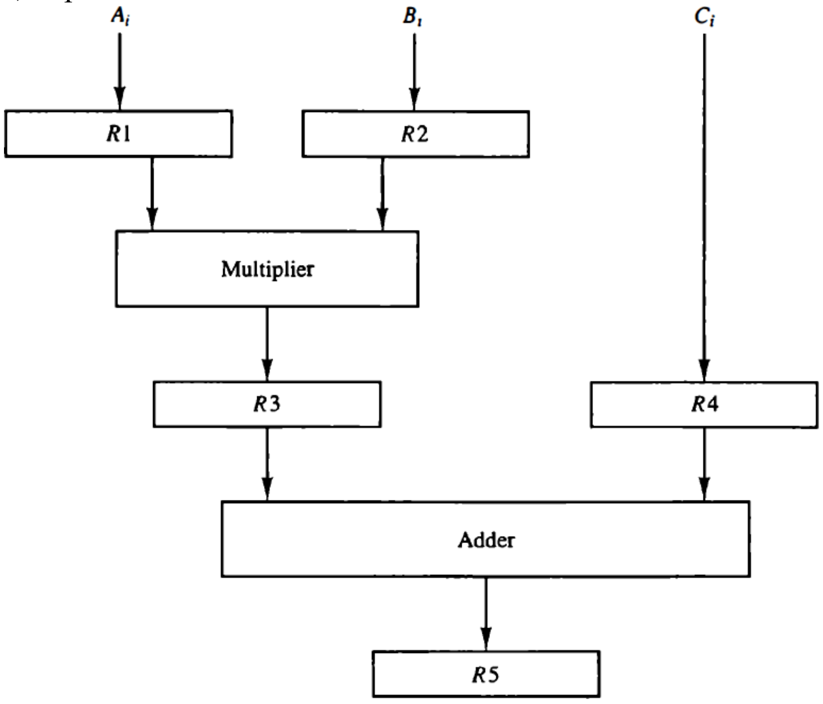
The suboperations performed in each segment of the pipeline are as follows:

R 1 <--Ai, R2 <--Bi

R3 <--R 1 * R2, R4 <--C,

R5 <--R3 + R4

- Input A, and B,
- Multiply and input C,
- Add C; to product



Example of pipeline processing.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$

<u>Table: Contents of Registers in pipeline</u>

The five registers are loaded with new data every clock pulse. The first clock pulse transfers A1 and B1 into R 1 and R2. The second dock pulse transfers the product of R 1 and R2 into

R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R 1 and R2. The third clock pulse operates on all three segments simultaneously. It places A, and B, into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

<u>Instruction Pipelining:</u>

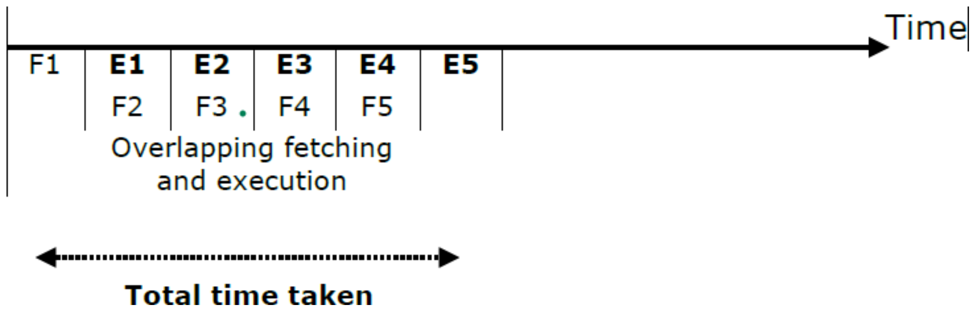
Instruction Pipelining is an implementation technique in which ***multiple instructions are overlapped in execution.*** An instruction requires several steps which mainly involve fetching, decoding and execution.

If these steps are performed one after the other, they will take a long time.

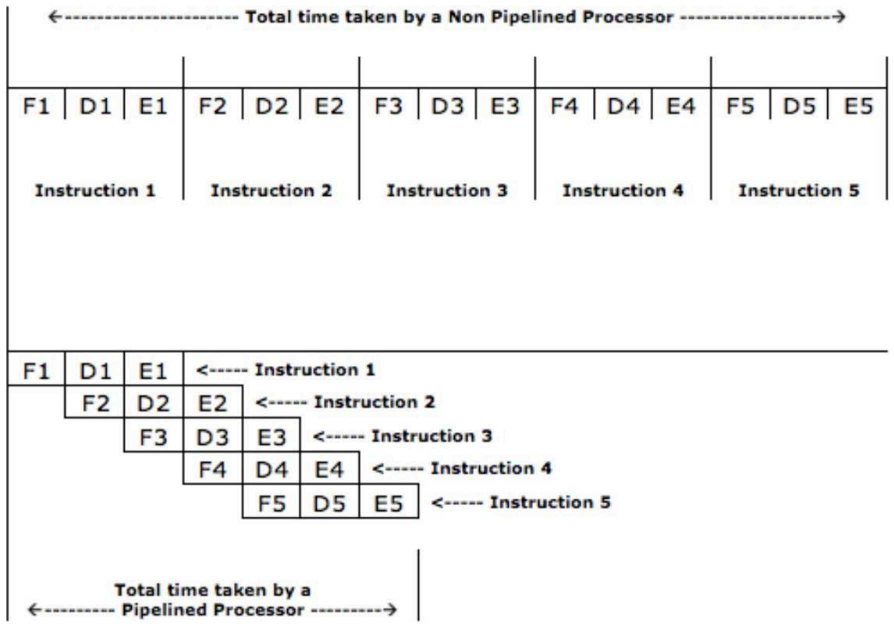
As processors became faster, several of these steps started to get overlapped, resulting in faster processing. This is done by a mechanism called pipelining.

2 STAGE PIPELINING - 8086

Here the instruction process in divided into two stages of fetching and execution. Fetching of the next instruction takes place while the current instruction is being executed. Hence two instructions are being processed at any point of time.



3 STAGE PIPELINING –ARM 7


Total time taken by a Non Pipelined Processor
F1 D1 E1 F2 D2 E2 F3 D3 E3 F4 D4 E4 F5 D5 E5
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
F1 D1 E1 <----- Instruction 1
F2 D2 E2 <----- Instruction 2
F3 D3 E3 <----- Instruction 3
F4 D4 E4 <----- Instruction 4
F5 D5 E5 <----- Instruction 5
Total time taken by a Pipelined Processor

Consider the case where a k-segment pipeline with a clock cycle time t_p ,is used to execute n tasks. The first task T1 requires a time equal to Kt_p, to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n - 1) tp . Therefore, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles.

Next consider a non-pipeline unit that performs the same operation and
takes a time equal to t_n. to complete each task. The total time required for n tasks is nt_n. The speedup of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

ADVANTAGE OF PIPELINING

The advantage of pipelining is that it increases the performance. As shown by the various examples above, deeper the pipelining, more is the level of parallelism, and hence the processor becomes much faster.

DRAWBACKS/ HAZARDS OF PIPELINING

There are various hazards of pipelining, which **cause a dip** in the performance of the processor. These hazards become even **more prominent** as the **number of pipeline stages increase**.

They may occur
due to the following reasons.

1) DATA HAZARD/ DATA DEPENDENCY HAZARD

Data Hazard is caused when **the result (destination) of one instruction becomes the operand (source) of the next instruction**.

Consider two instructions I1 and I2 (I1 being the first).

Assume I1: INC [4000H]

Assume I2: MOV BL , [4000H]

Clearly in I2, BL should get the incremented value of location [4000H].

But this can only happen once I1 has completely finished execution and also written back the result at [4000H].

In a multistage pipeline, I2 may reach execution stage before I1 has finished storing the result at location [4000H], and hence get a wrong value of data.

This is called **data dependency hazard**.

It is solved by inserting NOP (No operation) instructions between such data dependent instructions.

Because of the data hazard, there will be a delay in the pipeline. The data hazards are basically of three types:

1. RAW
2. WAR
3. WAW

To understand these hazards, we will assume we have two instructions I1 and I2, in such a way that I2 follows :

RAW:

RAW hazard can be referred to as 'Read after Write'. It is also known as Flow/True data dependency. If the later instruction tries to read on operand before earlier instruction writes it, in this case, the RAW hazards will occur. The condition to detect the RAW hazard is when O_n (Output of nth instruction) and I_{n+1}(Input of n+1th instruction) both have a minimum one common operand.

I1: add R1, R2, R3

I2: sub R5, R1, R4

WAR

WAR can be referred to as 'Write after Read'. It is also known as Anti-Data dependency. If the later instruction tries to write an operand before the earlier instruction reads it, in this case, the WAR hazards will occur. The condition to detect the WAR hazard is when I_n and O_{n+1} both have a minimum one common operand.

add R1, **R2,** R3
sub **R2,** R5, R4

In a reasonable (in-order) pipeline, the WAR hazard is very uncommon or impossible.

WAW

WAW can be referred to as 'Write after Write'. It is also known as Output Data dependency. If the later instruction tries to write on operand before earlier instruction writes it, in this case, the WAW hazards will occur. The condition to detect the WAW hazard is when O_n and O_{n+1} both have a minimum one common operand.

add **R1,** R2, R3
sub **R1,** R2, R4

2) CONTROL HAZARD/ CODE HAZARD
Pipelining assumes that the program will always flow in a sequential manner.
Hence, it performs various stages of the forthcoming instructions before-hand, while the current instruction is still being executed. While programs are sequential most of the times, it is not true always.
Sometimes, branches do occur in programs.

In such an event, all the forthcoming instructions that have been fetched/ decoded etc have to be flushed/ discarded, and the process has to start all over again, from the branch address. This causes pipeline bubbles, which simply means time of the processor is wasted. Consider the following set of instructions:
Start:
JMP Down
INC BL
MOV CL,
DL ADD AL,
BL
…
…
…
Down: DEC CH
JMP Down is a branch instruction.
After this instruction, program should jump to the location "Down" and continue with DEC CH
instruction.
But, in a multistage pipeline processor, the sequentially next instructions after JMP Down have already been fetched and decoded. These instructions will now have to be discarded and fetching will begin all over again from DEC CH. This will keep several units of the architecture idle for some time. This is called a pipeline bubble. The **problem of branching is solved** in higher processors by a method called "**Branch Prediction Algorithm**". It was introduced by **Pentium** processor. It relies on the **previous history** of the instruction as most programs are repetitive in nature. It then **makes a prediction** whether branch will be **taken or not** and hence puts the correct instructions in the pipelines.

3) STRUCTURAL HAZARD
Structural hazards are caused by **physical constraints in the architecture like the buses**. Even in the most basic form of pipelining, we want to execute one instruction and fetch the next one. Now as long as execution only involves registers, pipelining is possible. But **if execution requires to read/ write data from the memory, then it will make use of the buses, which means fetching cannot take place at the same time**. So the fetching unit will have to wait and hence a pipeline bubble is caused. This problem is solved in complex Harvard architecture processors, which use separate memories and separate buses for programs and data. This means fetching and execution can actually happen at the same time without any interference with each other.
E.g.: PIC 18 Microcontroller.

<u>Introduction to Parallel Processors:</u>

- A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" In multiprocessor can mean either a central processing unit (CPU) or an input-output processor (lOP).
- However, a system with a single CPU and one or more lOPs is usually not included in the definition of a multiprocessor system unless the lOP has computational facilities comparable to a CPU.
- Multiprocessors are classified as **multiple instruction stream, multiple data stream (MIMD) systems.**

- A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- The fact that microprocessors take **very little physical space and are very inexpensive** brings about the feasibility of interconnecting a large number of microprocessors into one composite system.
- **Very-large-scale integrated circuit technology** has reduced the cost of computer components to such a low level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.
- Multiprocessing **improves the reliability** of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.
- A multiprocessor system derives its high performance from the fact that computations can proceed in parallel in one of two ways.
  1. Multiple independent jobs can be made to operate in parallel.
  2. A single job can be partitioned into multiple parallel tasks.
- An overall function can be partitioned into a number of tasks that each processor can handle individually. System tasks may be allocated to special purpose processors whose design is optimized to perform certain types of processing efficiently.
- Example is a computer where one processor performs highspeed floating-point mathematical computations and another takes care of routine data-processing tasks.
- Multiprocessors are classified by the way their memory is organized.
    1. A multiprocessor system with common shared memory is classified as a shared memory or tightly coupled multiprocessor. This does not preclude each processor from having its own local memory. In fact, most commercial **tightly coupled multiprocessors** provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.
    2. An alternative model of microprocessor is the distributed-memory or **loosely coupled system**. Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message- passing scheme. The processors relay program and data to other processors in packets. A packet consists of an address, the data content, and some error detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used.

<u>**Shared Memory Multiprocessors:**</u>

- A **multiprocessor system consists of a number of processors capable of simultaneously executing independent tasks.** A task may encompass a few instructions for one pass through a loop, or thousands of instructions executed in a subroutine.

- In a shared-memory multiprocessor**, all processors have access to the same memory**. Tasks running in different processors can access shared variables in the memory using the same addresses. The size of the shared memory is likely to be large.
- Implementing a **large memory in a single module would create a bottleneck when many processors make requests to access the memory simultaneously**. This **problem is alleviated by distributing the memory across multiple modules** so that simultaneous requests from different processors are more likely to access different memory modules, depending on the addresses of those requests.
- **An interconnection network enables any processor to access any module that is a part of the shared memory**. When memory modules are kept physically separate from the processors, all requests to access memory must pass through the network. Below Figure shows such an arrangement.
- A **system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Access** (UMA) multiprocessor.

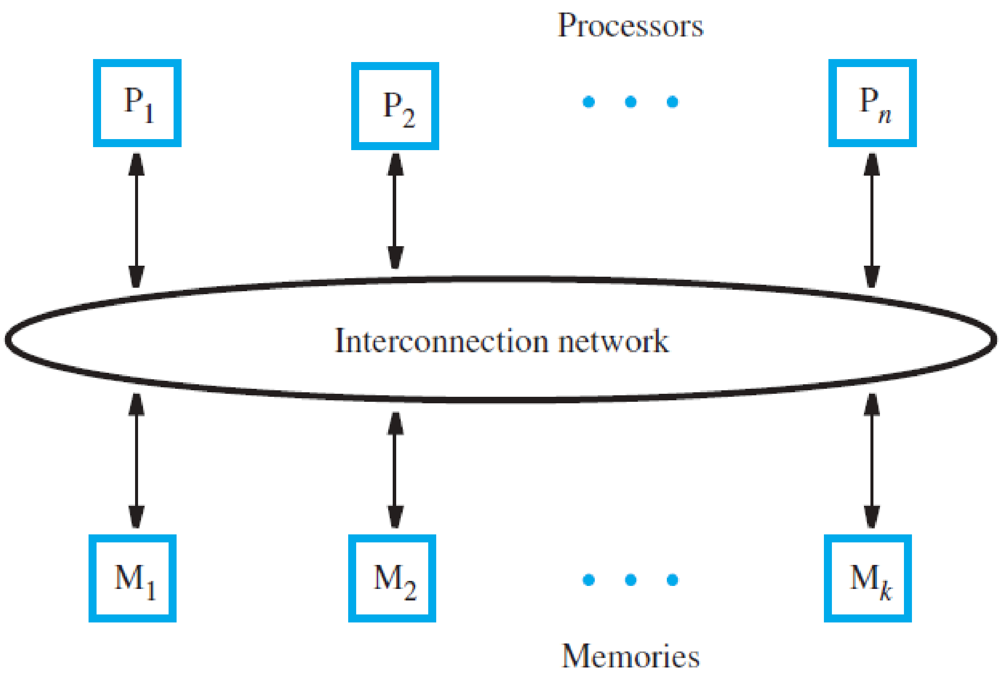


Fig: A UMA multiprocessor.

- For better performance, **it is desirable to place a memory module close to each processor.** The result is a collection of nodes, each consisting of a processor and a memory module. The nodes are then connected to the network, as shown in Figure below:

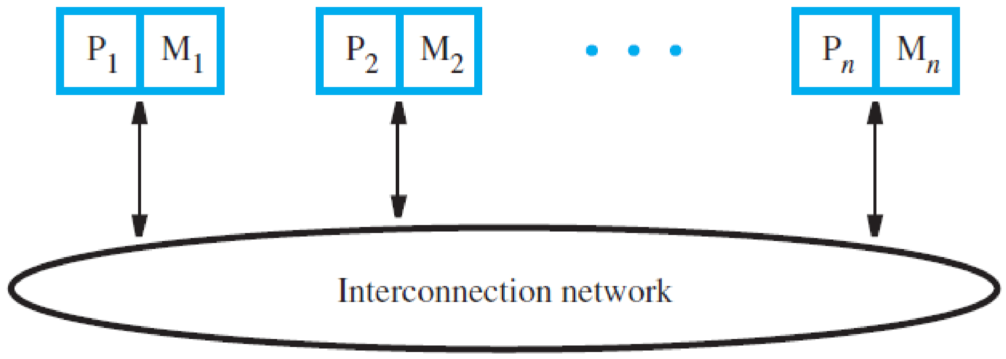


Fig: A NUMA multiprocessor.

- The network latency is avoided when a processor makes a request to access its local memory. However, a request to access a remote memory module must pass through the network. **Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called Non- Uniform Memory Access (NUMA) multiprocessors**.

Interconnection Networks:

- The interconnection network must allow information transfer between any pair of nodes in the system. The network may also be used to broadcast information from one node to many other nodes. The traffic in the network consists of requests (such as read and write)
and data transfers.
- The suitability of a particular network is judged in terms of **cost, bandwidth, effective throughput, and ease of implementation. The term bandwidth refers to the capacity of a transmission link to transfer data and is expressed in bits or bytes per second.** The effective throughput is the actual rate of data transfer. **This rate is less than the available bandwidth because a given link must also carry control information that coordinates the transfer of data.**
- **Information transfer through the network usually takes place in the form of packets of fixed length and specified format.** For example, a read request is likely to be a single packet sent from a processor to a memory module. The packet contains the node identifiers for the source and destination, the address of the location to be read, and a command field that indicates what type of read operation is required. A write request that writes one word in a memory module is also likely to be a single packet that includes the data to be written. On the other hand, a read response may involve an entire cache block requiring several packets for the data transfer.
- Ideally**, a complete packet would be handled in parallel in one clock cycle at any node or switch in the network.** This implies having wide links, comprising many wires. However, **to reduce cost and complexity, the links are often considerably narrower**. In such cases, a packet must be divided into smaller pieces, each of which can be transmitted in one clock cycle.
- The following are few of the interconnection networks that are commonly used in multiprocessors:

Bus:

A bus is a set of lines (wires) that provide a single shared path for information transfer. Buses are most commonly used in UMA multiprocessors to connect a

number of processors to several shared-memory modules. **Arbitration is necessary to ensure that only one of many possible requesters is granted use of the bus at any time.**
The bus is suitable for a relatively small number of processors because of the contention for access to the bus when many processors are connected. A simple bus does not allow a new request to appear on the bus until the response for the current request has been provided. However, if the response latency is high, there may be considerable idle time on the bus. Higher performance can be achieved by using a **split-transaction bus, in which a request and its corresponding response are treated as separate events. Other transfers may take place between them.**

<u>Ring:</u>
A ring network is formed with point-to-point connections between nodes, as shown in Figure below:

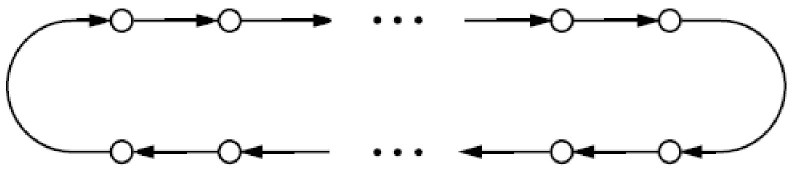
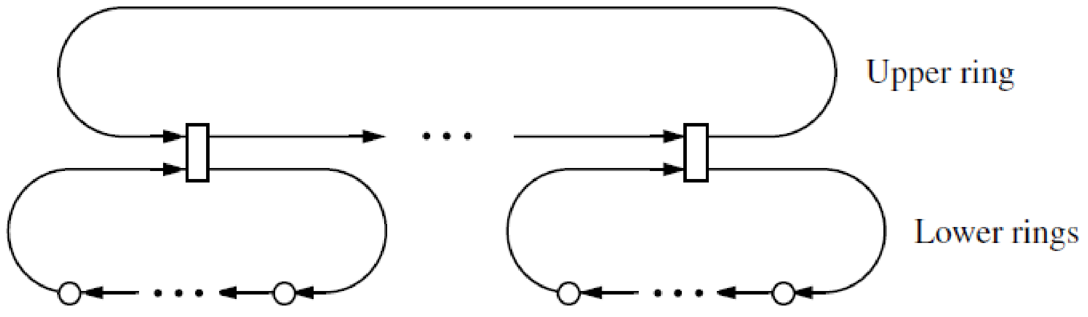


Fig: Simple Ring
A long single ring results in high average latency for communication between any two nodes. This high latency can be mitigated in two different ways. **A second ring can be added to connect the nodes in the opposite direction.** The resulting bidirectional ring halves the average latency and doubles the bandwidth. However, handling of communications is more complex.
Another approach is to use a hierarchy of rings. A two-level hierarchy is shown in Figure below: The upper-level ring connects the lower-level rings. The average latency for communication between any two nodes on lower-level rings is reduced with this arrangement. Transfers between nodes on the same lower-level ring need not traverse the
upper-level ring. Transfers between nodes on different lower-level rings include a traversal on part of the upper-level ring.



(b) Hierarchy of rings

<u>Crossbar:</u>
A crossbar is a network that provides a direct link between any pair of units connected to the network. It is typically used in UMA multiprocessors to connect

processors to memory modules. It enables many simultaneous transfers if the same destination is not the target of multiple requests. Below figure shows a crossbar that comprises a collection of switches. For n processors and k memories, n × k switches are needed.

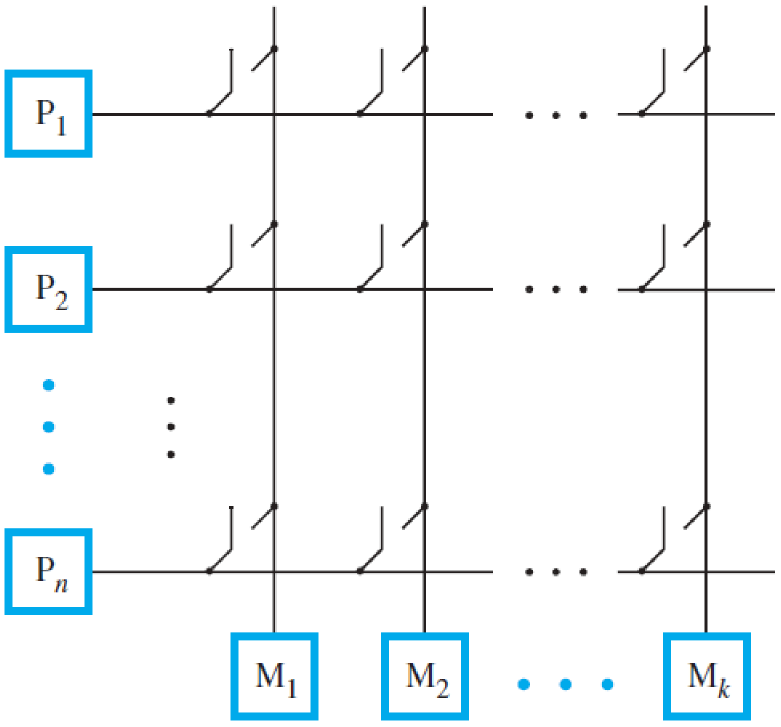


Fig: Crossbar Interconnection Network

<u>Mesh:</u>

A natural way of connecting a large number of nodes is with a two-dimensional mesh, as shown in Figure below:

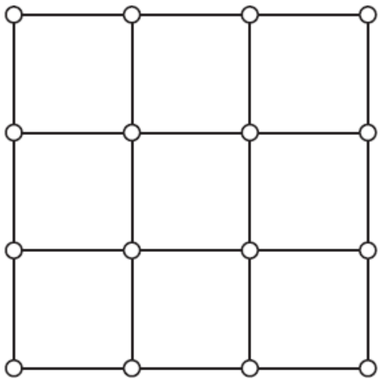


Fig : A two-dimensional mesh network.

Each internal node of the mesh has four connections, one to each of its horizontal and vertical neighbours. Nodes on the boundaries and corners of the mesh have fewer neighbours and hence fewer connections. To reduce latency for communication between nodes that would otherwise be far apart in the mesh, wrap around connections may be introduced between nodes at opposite boundaries of the mesh. **A network with**

such connections is called a torus. All nodes in a torus have four connections. Average latency is reduced, but the implementation complexity for routing requests and responses through a torus is somewhat higher than in the case of a simple mesh.

Cache Coherence:

1. A shared-memory multiprocessor is easy to program. *Each variable in a program has a unique address location in the memory, which can be accessed by any processor.* However, *each processor has its own cache*. Therefore, it is *necessary to deal with the possibility that copies of shared data may reside in several caches*.
2. When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value. They must be informed of the change so that they can either update their copy to the new value or invalidate it. This is the issue of maintaining *cache coherence, which requires having a consistent view of shared data in multiple caches*.
3. The write-through approach changes the data in both the cache and the main memory. The write-back approach changes the data only in the cache; the main memory copy is updated when a modified data block in the cache has to be replaced. Similar approaches can be used to address cache coherence in a multiprocessor system.

Write Through Protocol:

A write-through protocol can be implemented in one of two ways:

1) First version is based on updating the values in other caches. When a processor writes a new value to a block of data in its cache, the new value is also written into the memory module containing the block being modified. Since copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the Write operation. The simplest way of doing this is to broadcast the written data to the caches of all processors in the system. As each processor receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache.

2) The second version of the write-through protocol is based on invalidation of copies. When a processor writes a new value into its cache, this value is also sent to the appropriate location in memory, and all copies in other caches are invalidated. Again, broadcasting can be used to send the invalidation requests throughout the system.

Write-Back protocol:

✔ Maintaining coherence with the write-back protocol **is based on the concept of ownership of a block of data in the memory.** Initially, the memory is the owner of all blocks, and the memory retains ownership of any block that is read by a processor to place a copy in its cache.

✔ If some processor wants to write to a block in its cache, it must first become the exclusive owner of this block. To do so, **all copies in other caches must first be invalidated with a broadcast request**. The new owner of the block may then modify the contents at will without having to take any other action.

✔ **Read: When another processor wishes to read a block that has been modified, the request for the block must be forwarded to the current owner.** The data are then sent to the requesting processor by the current owner**. The data are also sent to the appropriate memory module, which reacquires ownership and updates the contents of the block in the memory.** The cache of the processor that was the previous owner retains a copy of the block. **Subsequent requests from other processors to read the same block are serviced by the memory module containing the block.**

✔ When another processor wishes to write to a block that has been modified, the current owner sends the data to the requesting processor. **It also transfers ownership of the block to the requesting processor and invalidates its cached copy**. Since the block is being modified by the new owner, the contents of the block in the memory are not updated. The next request for the same block is serviced by the new owner.

✔ The ***write-back protocol has the advantage of creating less traffic than the write-through protocol***. This is because a processor is likely to perform several writes to a cache block

before this block is needed by another processor. With the write-back protocol, these writes are performed only in the cache, once ownership is acquired with an invalidation request.

<u>**Snoopy Caches:**</u>

✔ In multiprocessors that connect a modest number of processors to the memory modules using a single bus, ***cache***

coherence can be realized using a scheme known as **snooping.**

✔ In a single-bus system, **all transactions between processors and memory modules occur via requests and responses on the bus.** Suppose that each processor cache has a controller circuit that observes, or snoops, all transactions on the bus.

✔ Below are some scenarios for the write-back protocol and how cache coherence is enforced**:**

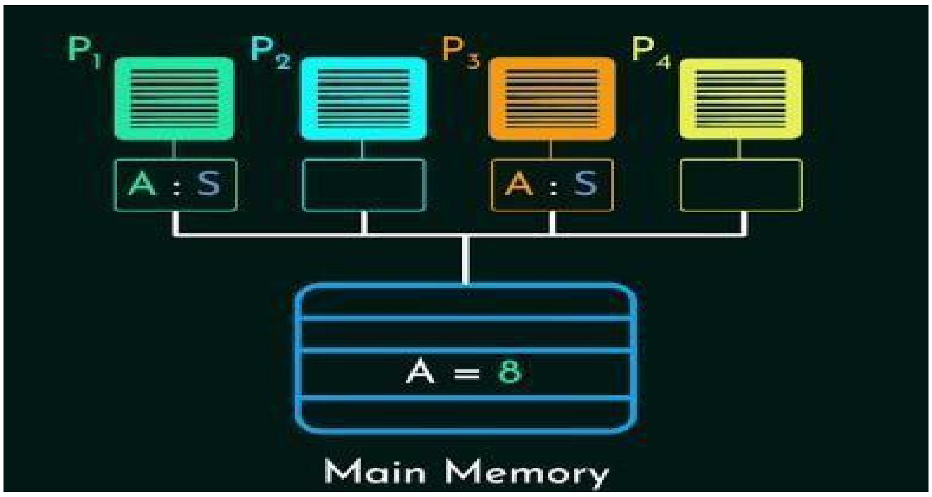
 Consider a processor that has previously read a copy of a block from the memory into its cache. Before writing to this block for the first time, **the processor must broadcast an invalidation request to all other caches, whose controllers accept the request and invalidate any copies of the same block.** This action causes the requesting processor to become the new owner of the block. The processor may then write to the block and mark it as being modified. No further broadcasts are needed from the same processor to write to the modified block in its cache.

 Now, if another processor broadcasts a read request on the bus for the same block, the memory must not respond because it is not the current owner of the block. The processor owning the requested block snoops the read request on the bus. **Because it holds a modified copy of the requested block in its cache, it asserts a special signal on the bus to prevent the memory from responding**. The owner then broadcasts a copy of the block on the bus, and marks its copy as clean (unmodified). The data response on the bus is accepted by the cache of the processor that issued the read request. **The data response is also accepted by the memory to update its copy of the block. In this case, the memory reacquires ownership of the block, and the block is said to be in a shared state because copies of it are in the caches of two processors**. Coherence is maintained because the two cached copies and the copy of the block in the memory contain the same data. **Subsequent requests from any processor are serviced by the memory**.

 Consider now the situation in which **two processors have copies of the same block in their respective caches**, and **both processors attempt to write to the same cache block at the same time**. Since the block is in the shared state, the memory is the owner of the block. Hence, **both processors request the use of the bus to broadcast an invalidation message**. One of the processors is granted the use of the bus first. **That processor broadcasts**

its invalidation request and becomes the new owner of the block. Through snooping, the copy of the block in the cache of the other processor is invalidated. When the other processor is later granted the use of the bus, it broadcasts a **read-exclusive request. This request combines a read request and an invalidation** request for the same block. The controller for the first processor snoops the read-exclusive request, provides a data response on the bus, and invalidates the copy in its cache. **Ownership of the block is therefore transferred to the second processor making the request**. The memory is not updated because the block is being modified again. Since the requests from the two processors are handled sequentially, cache coherence is maintained at all times.

- **The scheme just described is based on the ability of cache controllers to observe the activity on the bus and take appropriate actions**. Such schemes are called **snoopy-cache techniques**.



Snooping based protocol

<u>Directory-Based Cache Coherence:</u>
The concept of snoopy caches is easy to implement in single-bus systems. **Large shared memory multiprocessors use interconnection networks such as rings and meshe**s. In such systems, **broadcasting every single request to the caches of all processors is inefficient**. A scalable, but more complex, solution to this problem **uses directories** in each memory module **to indicate which nodes may have copies of a given block in the shared state. If a block is modified, the directory identifies the node that is the current owner.** Each request from a processor must be sent first to the memory module containing the relevant block. **The directory information for that block is used to determine the action that is taken. A read request is forwarded to the current owner, if the block is modified. In the case of a write request for a block that is shared, individual invalidations are sent only to nodes that may have copies of the block in question**. The cost and complexity of the directory-based approach for enforcing cache coherence limits its use to large systems. Small multiprocessors, including current multicore chips, typically use snooping.

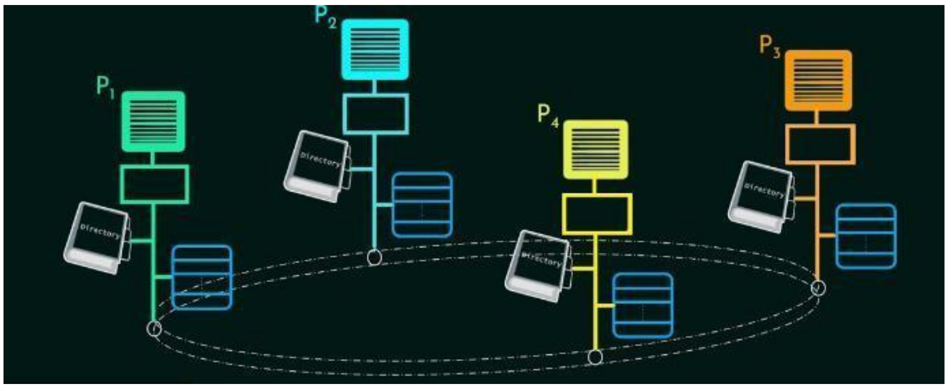
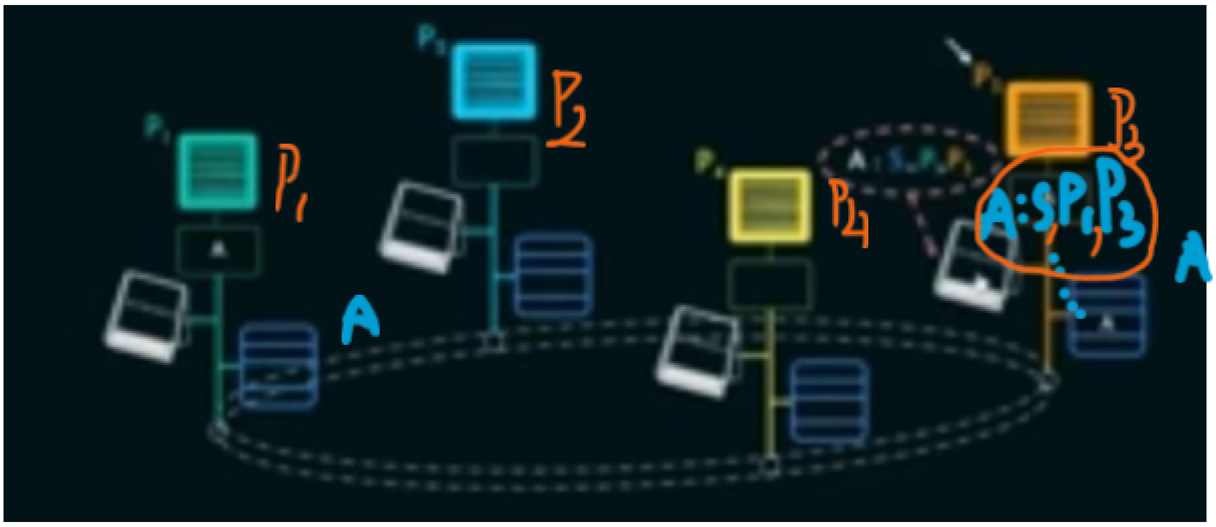


Fig: Directory based protocol



P1
P2
P4
P3
A
A:SP1,P3
A