# Chapter 1
# Data Representation

## 1.1 Data Types

→ The term "data" refers to factual information used for analysis or reasoning.

→ The data types found in the registers of digital computers may be classified as being one of the following categories: (1) Numbers used in arithmetic computation,
(2) Letters of the alphabet used in data processing,
(3) Other discrete symbols used for specific purposes.

→ Data are numbers and other binary-coded information that are operated on to achieve required computational results.

**Number System**

**i) Decimal**

→ Base, or radix (r) = 10

→ 10 symbols: 0,1,2,3,4,5,6,7,8,9

→ E.g. $630.5 = 6*10^2 + 3*10^1 + 0*10^0 + 5*10^{-1}$

**ii) Binary**

→ r = 2

→ 2 symbols: 0,1

→ E.g. $110010.01 = 1*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$

**iii) Octal**

→ r = 8

→ 8 symbols: 0,1,2,3,4,5,6,7

→ E.g. $(736.4)_8 = 7*8^2 + 3*8^1 + 6*8^0 + 4*8^{-1}$

**iv) Hexadecimal**

→ r = 16

→ 16 symbols: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

→ E.g. $A9D = A*16^2 + 9*16^1 + D*16^0$

**Conversion**

**i) Base r to decimal**

→ Multiply each coefficient with corresponding power of r and add.

**ii) Decimal to base r**

→ Divide by base r and collect remainder upwards.

But,

→ Converting decimal fraction to binary, multiplication is used instead of division.

→ And, integers are calculated instead of remainders.

→ Those remainders are collected downwards.

→ E.g. $(0.625)_{10} = (?)_2$

| | Integer | Fraction | Coefficient |
|---|---|---|---|
| 0.625*2 | 1 | 0.25 | 1 |
| 0.25*2 | 0 | 0.5 | 0 |
| 0.5*2 | 1 | 0 | 1 |

→ Therefore, $(0.625)_{10} = (0.101)_2$

→ H/W, $(65.553)_{10} = (?)_2$

**iii) Binary to octal (hexadecimal) & vice versa**

→ Conversion from binary to octal (hexadecimal) is accomplished by partitioning the binary number into group of 3 (for octal) or 4 (for hexadecimal) digits each starting from binary point (.) & proceeding to the left and to the right.

→ E.g. $(\underline{1}\ \underline{101}\ \underline{001}\ .\ \underline{111})_2 = (151.7)_8$
$(\underline{0110}\ \underline{1001}\ .\ \underline{1101}\ \underline{0})_2 = (69.D)_{16}$

→ Converting from octal (hexadecimal) to binary is done by process in reverse above.

→ E.g. $(673.124)_8 = (110\ 111\ 011\ .\ 001\ 010\ 100)_2$
$(306.D)_{16} = (11\ 0000\ 0110\ .\ 1101)_2$

**H/W Alphanumeric code (ASCII)**

## 1.2 Complements

→ Complements are used in digital systems for simplifying the subtraction operation and for logic manipulation.

→ Two types of complements are:
i) r's complement
ii) (r-1)'s complement

**i) r's complement**

→ r's complement of N (positive number with base r) is defined as

$r^n - N$, for $N \neq 0$
and 0, for N=0
where,
n=number of digits (integer parts only)

→ E.g. $(147.53)_{10, c} = 10^3 - 147.53 = 852.47$ Ans.

**ii) (r-1)'s complement**

→ (r-1)'s complement of N (positive number with base r) is defined as

$(r^n - r^{-m}) - N$, for number with fraction
$(r^n - 1) - N$, for number with integer only
where,
n →number of digits in integer part of N
m→number of digits in fraction part of N
r→base or radix

→ E.g. $(147.53)_{9, c} = 10^3 - 10^{-2} - 147.53 = 852.46$ Ans.

→ To find (r-1)'s complement, subtract all digit from (r-1).
E.g. for 1's complement, subtract each digit from 1.
for 7's complement, subtract each digit from 7.
for 9's complement, subtract each digit from 9.
for 15's complement, subtract each digit from 15.

And,

→ To find r's complement, subtract each digit from (r-1), and then add 1 to the last digit.
E.g. for 2's complement, subtract each digit from 1, and then add 1 to the last digit.
for 8's complement, subtract each digit from 7, and then add 1 to the last digit.
for 10's complement, subtract each digit from 9, and then add 1 to the last digit.

for 16's complement, subtract each digit from 15, and then add 1 to the last digit.

→ C/W, Give some examples on 1's complement and 2's complement.

**#Subtraction with r's complement**

→ Subtraction of two positive numbers (M - N), both of base r, is done as follows:

i) Add M to r's complement of N.
ii) If an end carry occurs, discard it. What is left is result.
iii) If an end carry does not occur, take r's complement of the number obtained in step (i) and place a negative sign in front. That is the result.

**#Subtraction with (r-1)'s complement**

→ Subtraction of two positive numbers (M - N), both of base r, is done as follows:

i) Add M to (r-1)'s complement of N.
ii) If an end carry occurs, add 1 to LSB. What is received is result.
iii) If an end carry does not occur, take (r-1)'s complement of the number obtained in step (i) and place a negative sign in front. That is the result.

## 1.3 Fixed-Point Representation

→ Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number.

→ There will be problem in storing +ve/-ve sign (+/-) and binary (or decimal) point.

→ Convention is 0 for +ve sign and 1 for –ve sign.

**#Integer Representation**

→ Suppose, we have 8-bit register.

→ For +ve number,

i) Signed-magnitude representation
When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number.
E.g. +14 = **0** 0001110

→ For –ve number,

i) Signed-magnitude representation
1 for the -ve sign followed by binary equivalent of that number. E.g. -14 = **1** 0001110
ii) Signed-1's complement representation
1 for the -ve sign followed by 1's complement of that number. E.g. -14 = **1** 1110001
iii) Signed-2's complement representation
1 for the -ve sign followed by 2's complement of that number. E.g. -14 = **1** 1110010

→ Signed-magnitude system is used in ordinary arithmetic while the signed-complement is used in computer arithmetic.

→ The 1's complement has two representations of 0 (+0 and -0), hence brings difficulties and used in some older computers.

→ The 2's complement is widely used.

**#Overflow**

→ When two numbers of n digits each are added and the sum occupies n + 1 digits, we say that an overflow occurred.

→ An overflow is a problem in digital computers because the width of registers is finite. A result that contains n + 1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

→ The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.

→ When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.

→ In the case of signed numbers, the leftmost bit always represents the sign, and negative numbers are in 2's complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

→ An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers.

→ An overflow may occur if the two numbers added are both either positive or both negative.

→ E.g.

| carries: 0 1 | | carries: 1 0 | |
|---|---|---|---|
| +70 | 0 1000110 | −70 | 1 0111010 |
| +80 | 0 1010000 | −80 | 1 0110000 |
| +150 | 1 0010110 | −150 | 0 1101010 |

→ An overflow condition can be detected by observing that if these two carries are not equal, an overflow condition is produced.

→ If the two carries are applied to an exclusive OR (XOR) gate, an overflow will be detected when the output of the gate is equal to 1.

## 1.4 Floating-Point Representation

→ Floating-Point Representation has two parts:

i) Mantissa (a signed number, may be a fraction or an integer)

ii) Exponent

→ E.g. $+62345.54 = +(0.6234554)*10^{+5}$

| Fraction | Exponent |
|---|---|
| +0.6234554 | +5 |

→ General form,

$m * r^e$, where r = radix

→ A floating point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

$m * 2^e = +(.100111)_2 * 2^{+4}$

| Fraction | Exponent |
|---|---|
| 01001110 | 000100 |

→ The fraction has a 0 in the leftmost position to denote positive.

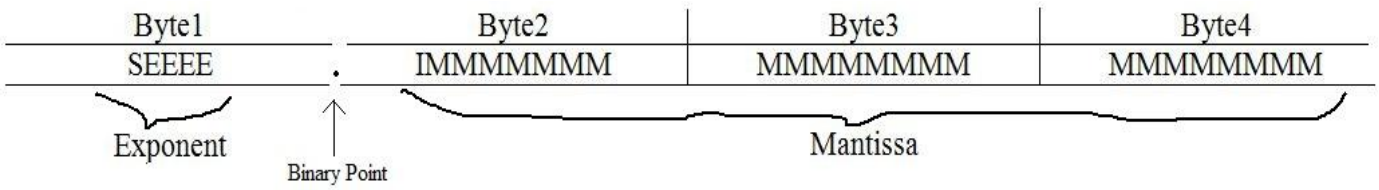
→ The binary point of the fraction follows the sign bit but is not shown in the register.

→ A floating point number is said to be normalized if the most significant digit of the mantissa is nonzero.

→ Two main standard forms of the floating point number by

- ANSI (American National Standards Institute)
- IEEE (Institute of Electrical and Electronic Engineers)

→ The ANSI 32-bit floating-point number in byte format is:




where,
S = sign of Mantissa
E = exponent bits (in 2's complement)
M = mantissa bits

→ E.g.

13 = 1101 = +(.1101) * $2^4$ = 00000100 11010000 00000000 00000000
-17 = -10001 = -(.10001) * $2^5$ = 10000101 10001000 00000000 00000000
$-0.125_{10} = -0.001_2 = -(.1) * 2^{-2}$ = 11111110 10000000 00000000 00000000

# Chapter 2
# Register Transfer and Microoperations

## 2.1 Register and Register Transfer Language (RTL)

**Register**

→ Register is the storage device, inside CPU, of data on which microoperations are performed.

→ The operations executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear and load.

→ The internal hardware organization of a digital computer is best defined by specifying:
- The set of registers it contains and their function.
- The sequence of microoperations performed on the binary information stored in the registers.
- The control that initiates the sequence of microoperations.

→ The language, which is basically used to express the transfer of data among the registers, is called ***Register Transfer Language*** **(RTL).** It is the symbolic notation used to describe the microoperation transfers among registers. In such transfer, one of the source or destination should be register (not necessarily both).

**Register Transfer**

→ Computer registers are designated by capital letters.

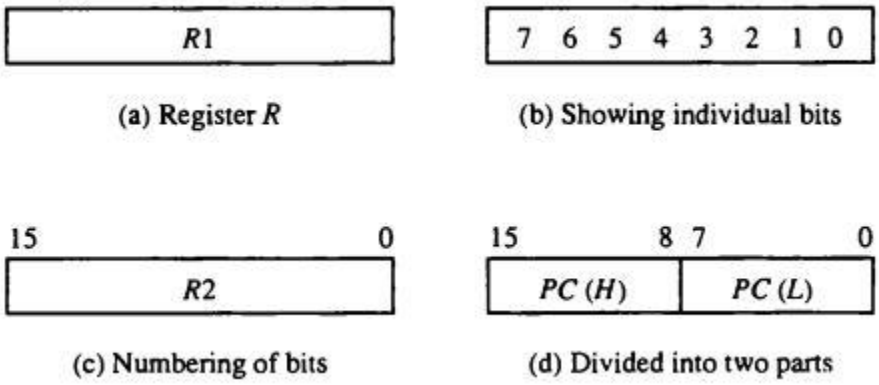
→ For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR. Other designations for registers are PC (for program counter), IR (for instruction register, and R1 (for processor register).

→ The most common way to represent a register is by a rectangular box with the name of the register inside. Fig (a).

→ The individual flip-flops in an n-bit register are numbered in sequence from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left. For e.g. 8-bit register numbered: Fig (b).

→ The numbering of bits in a 8-bit register can be marked on top of the box. Fig (c).

→ A 16-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.




→ Information transfer from one register to another is designated in symbolic form by means of a replacement operator.

$$R2 \leftarrow R1$$

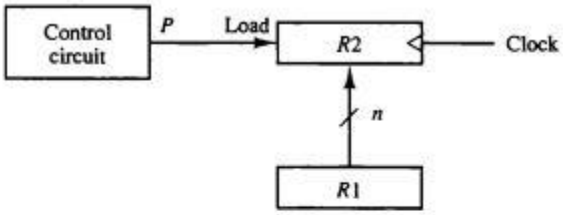
→ If there is predetermined control condition like

$$\text{If } (P=1) \text{ then } (R2 \leftarrow R1)$$

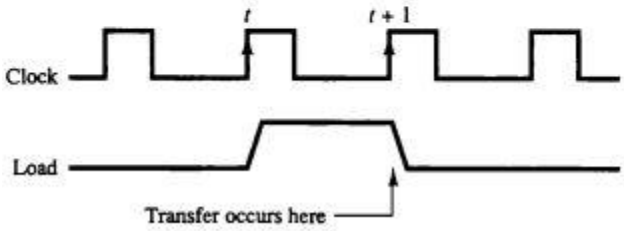
then we can write the statement as

P: R2 ← R1

where P is control signal usually a control function which is Boolean variable that is equal to 1 or 0.




(a) Block diagram




(b) Timing diagram

→ The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register.

→ It is assumed that all transfers occur during a clock edge transition. 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 next positive transition of the clock at time t + 1.

→ A comma is used to separate two or more operations that are executed at the same time. The statement

T: R2 ← R1, R1 ← R2

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

| 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* ← *R1* |
| Comma , | Separates two microoperations | *R2* ← *R1*, *R1* ← *R2* |

Fig: Basic Symbols for Register Transfers

→ For example, RTL of fetch cycle can be written as:

T1: MAR←PC
T2: MBR←[MAR]
T3: IR←MBR
T4: unspecified; PC←PC+1

The notation (T1, T2, T3, T4) represents successive time units. All three units are of equal duration. A time unit is defined by regularly spaced clock pulses. The operations performed within this single unit of time are called microoperations. A single time unit can contain one or more microoperations. Since each microoperation can specifies the transfer of data into or out of a register, such type is called **RTL**.

## 2.2 Bus and Memory Transfer

**Bus**

→ A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system.

→ A more efficient scheme for transferring information between registers in a multiple register configuration is a common bus system.

→ A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

→ For example,

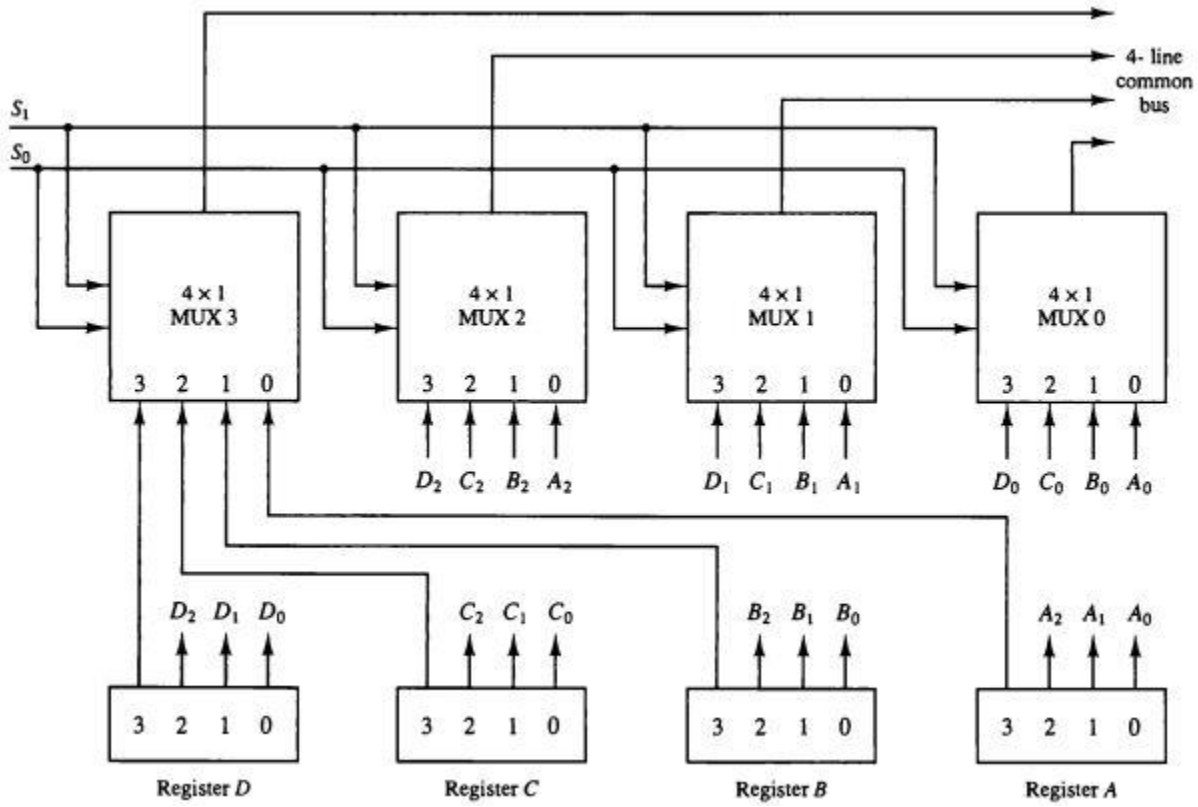



Fig: Bus system for four registers.

| $S_1$ | $S_0$ | Register selected |
|---|---|---|
| 0 | 0 | *A* |
| 0 | 1 | *B* |
| 1 | 0 | *C* |
| 1 | 1 | *D* |

Fig: Function table for Bus

→ In general, a bus system will multiplex k registers of n bits each to produce an n line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be k X 1 since it multiplexes k data lines.

→ For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

→ The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected.

**Memory Transfer**

→ 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 the memory address during the transfer.

→ 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.

## 2.3 Microoperations

→ The operations on the data in registers are called microoperations.

→ Alternatively we can say that an elementary operation performed during one clock pulse on the information stored in one or more registers is called micro-operation.

→ The result of the operation may replace the previous binary information of the resister or may be transferred to another resister.

→ Register Transfer Language (RTL) can be used to describe the (sequence of) micro-operations.

→ The microoperations most often encountered in digital computers are classified into 4 categories:

i) Register transfer microoperations
ii) Arithmetic microoperations
iii) Logic microoperations
iv) Shift microoperations

**Register transfer microoperations**

Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR). Often the names indicate function:

| | |
|---|---|
| MAR | memory address register |
| PC | program counter |
| IR | instruction register |

Information transfer from one register to another is described in symbolic form by replacement operator. The statement "R2 ←R1"

denotes a transfer of the content of the R1 into resister R2.

Control Function

Often actions need to only occur if a certain condition is true. In digital systems, this is often done via a control signal, called a control function.

Example: P: R2 ← R1 i.e. if (P = 1) then (R2← R1)

Which means "if P = 1, then load the contents of register R1 into register R2".

If two or more operations are to occur simultaneously, they are separated with commas.

Example: P: R3 ← R5, MAR ← IR

**Arithmetic microoperations**

The basic arithmetic microoperations are

– Addition

- Subtraction
- Increment
- Decrement

The additional arithmetic microoperations are

- Add with carry
- Subtract with borrow
- Transfer/Load

Summary of typical arithmetic microoperations

| **Symbolic Designation** | **Description** |
|---|---|
| R3 ← R1 + R2 | Contents of R1 plus R2 transferred to R3 |
| R3 ← R1 + R2' + 1 | Contents of R1 minus R2 transferred to R3 |
| R1 ← R1 + 1 | Increment the contents of R1 by one |
| R1 ← R1 - 1 | Decrement the contents of R1 by one |
| R3 ← R1 + R2 + 1 | Add with carry |
| R3 ← R1 + R2' | Subtract with borrow |
| R1 ← R1' +1 | 2's complement the contents of R1 (negate) |

**Logic microoperations**

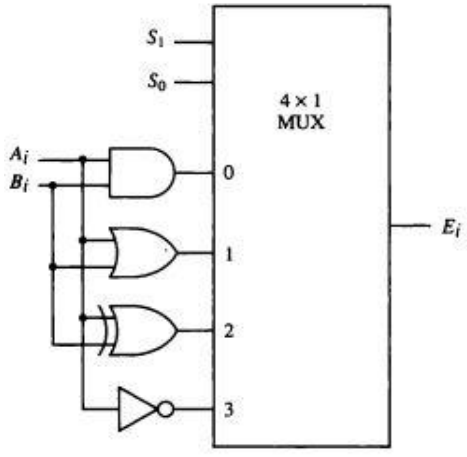
Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data. Useful for bit manipulations on binary data and for making logical decisions based on the bit value. There are, in principle, 16 different logic functions that can be defined over two binary input variables. However, most systems only implement four of these

- AND (/\), OR (\/), XOR (⊕), Complement/NOT

The others can be created from combination of these four functions.

| **Microoperation** | **Name** |
|---|---|
| F ← R1 /\ R2 | AND |
| F ← R1 \/ R2 | OR |
| F ← R1 ⊕ R2 | XOR |
| F ← R1' | Complement (NOT) |

Hardware Implementation




(a) Logic diagram

| $S_1$ | $S_0$ | Output | Operation |
|---|---|---|---|
| 0 | 0 | $E = A \wedge B$ | AND |
| 0 | 1 | $E = A \vee B$ | OR |
| 1 | 0 | $E = A \oplus B$ | XOR |
| 1 | 1 | $E = \overline{A}$ | Complement |

(b) Function table

**Shift microoperations**

The operation that changes the adjacent bit position of the binary values stored in the register is known as shift microoperation. They are used for serial transfer of data. The shift microoperations are classified into 3 types:

i) **Logical shift**: A logical shift transfer 0 through the serial input. It can be defined in RTL by:

R←shl R shift-left register R
R←shr R shift-right register R

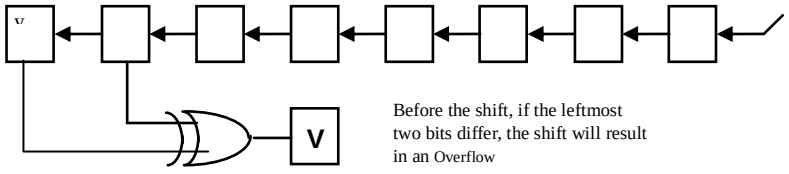

ii) **Circular shift**: A circular shift rotates the bit from one end of the register to another end of the register. It can be defined in RTL by:

R←cil R circular shift-left register R
R←cir R circular shift-right register R

iii) **Arithmetic shift**: It shifts signed-binary number left or right. For shift left the content of the register is multiplied by 2 whereas For shift right the content of the register is divided by 2. The arithmetic shift must leave the sign bit unchanged. It can be defined in RTL by:

R←ashl R arithmetic shift-left register R
R←ashr R arithmetic shift-right register R


Overflow case during arithmetic shift-left:
If a bit in $R_{n-1}$ changes in value after the shift, sign reversal occurs in the result. This happens if the multiplication by 2 causes an overflow.
Thus, left arithmetic shift operation must be checked for the overflow: an overflow occurs after an arithmetic shift-left if before shift $R_{n-1} \neq R_{n-2}$.




An overflow flip-flop V can be used to detect an arithmetic shift-left overflow.

$$V = R_{n-1} \oplus R_{n-2}$$

If V = 0, there is no overflow but if V = 1, overflow is detected.

## 2.4 Arithmetic Logic Shift Unit

Arithmetic logic shift unit is a digital circuit that performs arithmetic calculations, logical manipulation and shift operation. It is often abbreviated as ALU. The above figure shows the one stage of arithmetic logic shift unit.
The block diagram of ALU includes one stage of arithmetic circuit, one stage of logic circuit and one 4*1 multiplexer. The subscript i designates a typical stage.

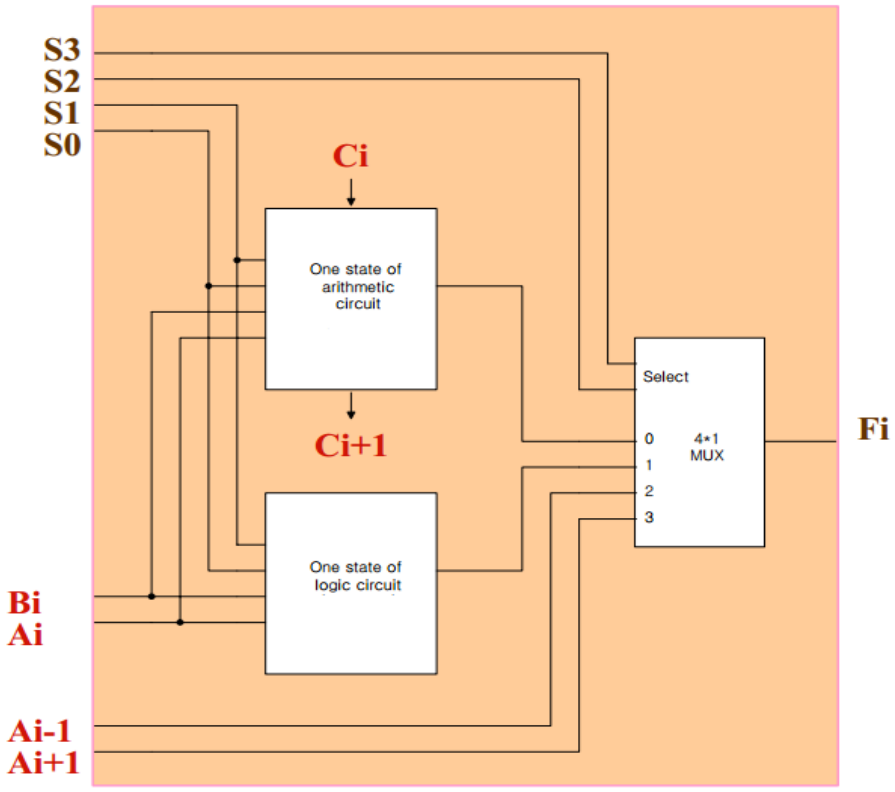



Fig: one stage of arithmetic logic shift unit

Inputs $A_i$ and $B_i$ are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs $S_1$ and $S_0$. A 4*1 MUX selects the final output. The two inputs of the MUX are received from the output of the arithmetic circuit and logic circuit. The other two is $A_{i-1}$ for the shift-right operation and $A_{i+1}$ for the shift left operation. The circuit is repeated n times for n-bit ALU. The output carry $C_{i+1}$ is connected to the input carry $C_{in}$. In every stage the circuit specifies 8 arithmetic operations, 4 logical operations and 2 shift operations, where each operation is selected by the five variables $S_3$, $S_2$, $S_1$, $S_0$ and $C_{in}$.

The operations of ALU can be summarized in table below:

| Operation select | | | | | Operation | Function |
|---|---|---|---|---|---|---|
| $S_3$ | $S_2$ | $S_1$ | $S_0$ | $C_{in}$ | | |
| 0 | 0 | 0 | 0 | 0 | F=A | Transfer A |
| 0 | 0 | 0 | 0 | 1 | F=A+1 | Increment A |
| 0 | 0 | 0 | 1 | 0 | F=A+B | Addition |
| 0 | 0 | 0 | 1 | 1 | F=A+B+1 | Add with carry |
| 0 | 0 | 1 | 0 | 0 | F=A+B’ | Subtract with borrow |
| 0 | 0 | 1 | 0 | 1 | F= A+B’+1 | Subtraction |
| 0 | 0 | 1 | 1 | 0 | F=A-1 | Decrement A |
| 0 | 0 | 1 | 1 | 1 | F=A | Transfer A |
| 0 | 1 | 0 | 0 | ╳ | F=A∧B | AND |
| 0 | 1 | 0 | 1 | ╳ | F=A∨B | OR |
| 0 | 1 | 1 | 0 | ╳ | F=A⊕B | XOR |
| 0 | 1 | 1 | 1 | ╳ | F=A’ | Complement A |
| 1 | 0 | ╳ | ╳ | ╳ | F=shr A | Shift right A into F |
| 1 | 1 | ╳ | ╳ | ╳ | F= shl A | Shift left A into F |

# Chapter 3
# Basic Computer Organization and Design

## 3.1 Introduction: Description of Basic Computer

We introduce here a basic computer whose operation can be specified by the resister transfer statements. Internal organization of the computer is defined by the sequence of microoperations it performs on data stored in its resisters. Every different processor type has its own design (different registers, buses, microoperations, machine instructions, etc). Modern processor is a very complex device. It contains:

- Many registers
- Multiple arithmetic units, for both integer and floating point calculations
- The ability to pipeline several consecutive instructions for execution speedup.

However, to understand how processors work, we will start with a simplified processor model. M. Morris Mano introduces a simple processor model; he calls it a "Basic Computer". The Basic Computer has two components, a processor and memory.

- The memory has 4096 words in it
  - $4096 = 2^{12}$, so it takes 12 bits to select an address in memory
- Each word is 16 bits long

**Stored Program Organization**

The program (instruction) as well as data (operand) is stored in the same memory. If the instruction needs data, the data is found in the same memory and accessed. This feature is called stored program organization.

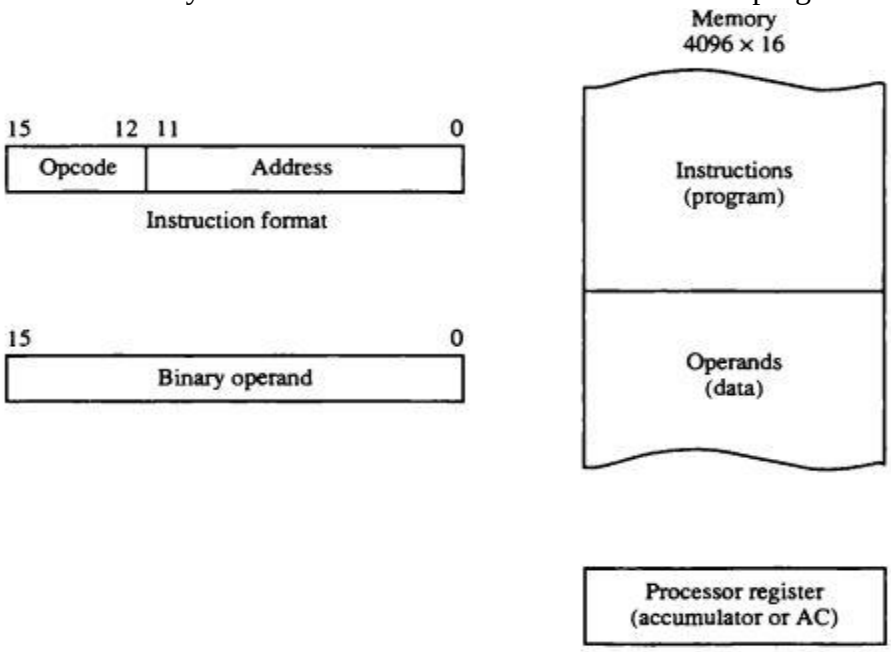



Fig: Stored Program Organization

**Instruction Format**

A computer instruction is often divided into two parts

- An *op-code* (Operation Code) that specifies the operation for that instruction
- An *address* that specifies the registers and/or locations in memory to use for that operation

In the Basic Computer, since the memory contains 4096 (= $2^{12}$) words, we needs 12 bit to specify the memory address that is used by this instruction. In the Basic Computer, bit 15 of the instruction specifies the *addressing mode* (0: direct addressing, 1: indirect addressing). Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's op-code.

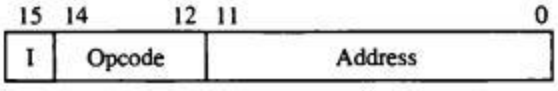



Fig: Instruction Format

**Addressing Modes**

The address field of an instruction can represent either

- Direct address: the address operand field is effective address (the address of the operand).
- Indirect address: the address operand field contains the memory address of effective address.

**Basic Computer Registers**

Computer instructions are normally stored in the consecutive memory locations and are executed sequentially one at a time. Thus computer needs processor registers for manipulating data and holding memory address which are shown in the following table:

| Symbol | Size | Register Name | Description |
|---|---|---|---|
| DR | 16 | Data Register | Holds memory operand |
| AR | 12 | Address Register | Holds address for memory |
| AC | 16 | Accumulator | Processor register |
| IR | 16 | Instruction Register | Holds instruction code |
| PC | 12 | Program Counter | Holds address of instruction |
| TR | 16 | Temporary Register | Holds temporary data |
| INPR | 8 | Input Register | Holds input character |
| OUTR | 8 | Output Register | Holds output character |

Since the memory in the Basic Computer only has 4096 ($=2^{12}$) locations, PC and AR only needs 12 bits.
Since the word size of Basic Computer only has 16 bit, the DR, AC, IR and TR needs 16 bits. The Basic Computer uses a very simple model of input/output (I/O) operations.

- Input devices are considered to send 8 bits of character data to the processor
- The processor can send 8 bits of character data to output devices

The Input Register (INPR) holds an 8-bit character gotten from an input device and the Output Register (OUTR) holds an 8-bit character to be sent to an output device.

## 3.2 Common Bus System

→ The basic computer has eight registers, a memory unit, and a control unit.

→ These registers, memory and control unit are connected using a path (bus) so that information can be transferred to each other.

→ If separate buses are used for connecting each registers, it will cost high.

→ The cost and use of extra buses can be reduced using a special scheme in which many registers use a common bus, called *common bus system*.

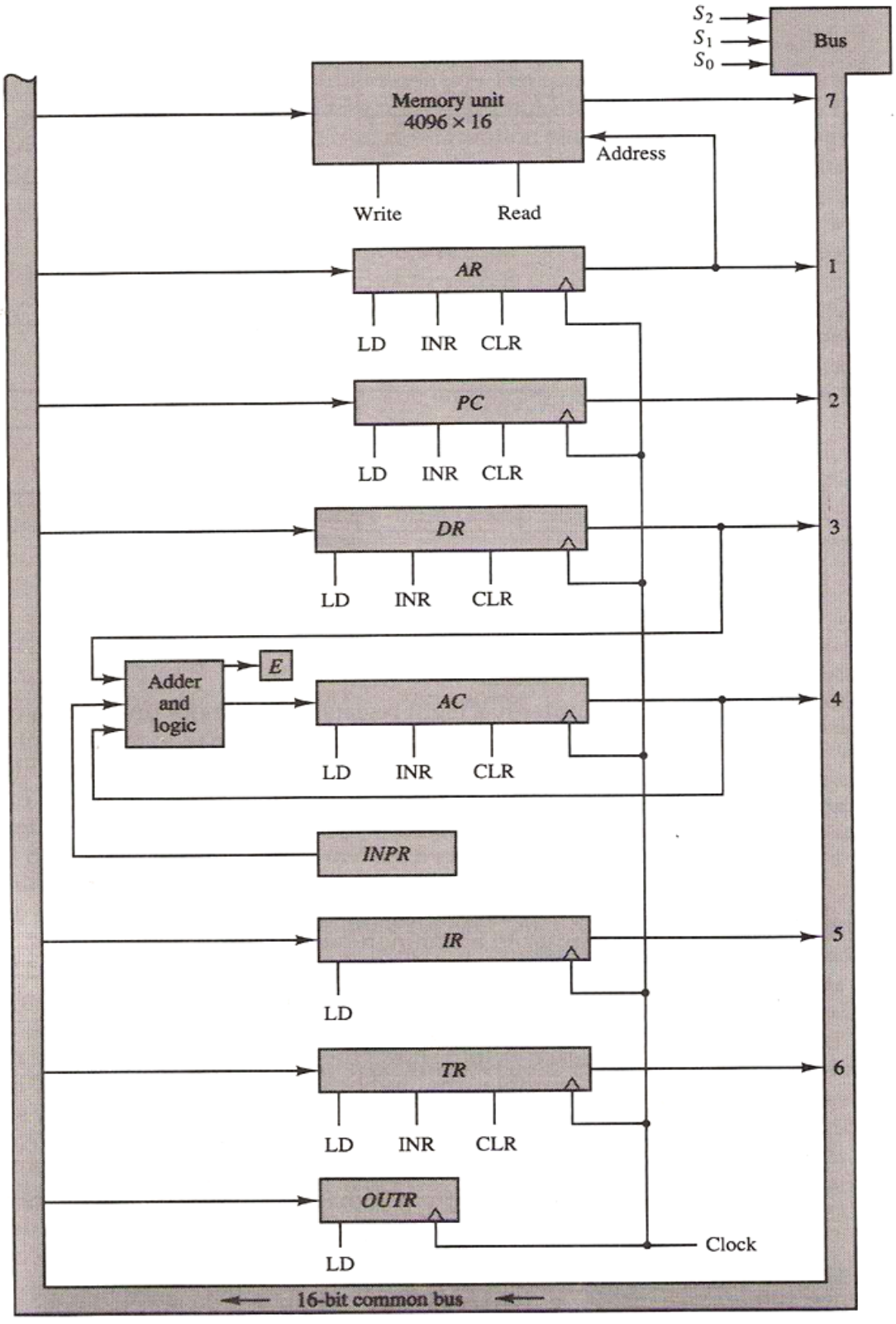



Fig: Common Bus System

→ Three control lines S2, S1 and S0 control the register to be selected as the input by the bus.

| S2 | S1 | S0 | Register |
|---|---|---|---|
| 0 | 0 | 0 | X (nothing) |
| 0 | 0 | 1 | AR |

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | PC |
| 0 | 1 | 1 | DR |
| 1 | 0 | 0 | AC |
| 1 | 0 | 1 | IR |
| 1 | 1 | 0 | TR |
| 1 | 1 | 1 | Memory |

→ The lines from the common bus are connected to the inputs of each register and the data inputs of the memory.
→ The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition.
→ The memory receives the contents of the bus when its write input is activated.
→ The memory places its 16 bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

## 3.3 Instruction Formats and their Execution

→ The instruction length of basic computer is 16-bit.
→ 16-bit instruction of basic computer has three fields:

| Mode | Op-code | Operand (Address) |
|---|---|---|

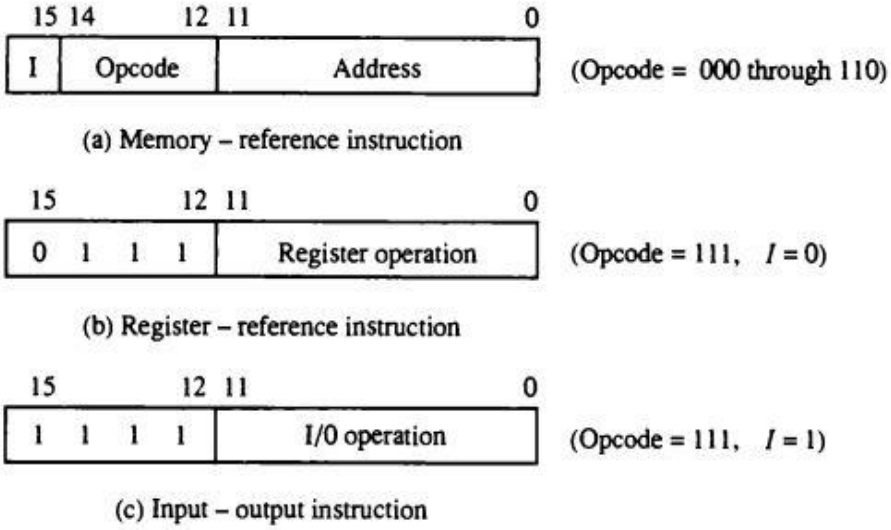
**i) Mode field**
→ MSB (bit 15) of the 16-bit instruction.
→ Value 0 = direct addressing, value 1 = indirect addressing mode.

**ii) Op-code field**
→ Defines what operation to be performed.
→ Contains 3 bits (bits 14-12)

**iii) Operand (Address) field**
→ Contains 12 bits (bits 11-0)

The basic computer has 3 instruction code formats. Type of the instruction is recognized by the computer control from 4-bit positions 12 through 15 of the instruction.

**i) Memory-Reference Instructions**
**ii) Register-Reference Instructions**
**iii) Input-Output Instructions**



| Symbol | Hexadecimal code $I = 0$ | Hexadecimal code $I = 1$ | Description |
|---|---|---|---|
| AND | 0xxx | 8xxx | AND memory word to $AC$ |
| ADD | 1xxx | 9xxx | Add memory word to $AC$ |
| LDA | 2xxx | Axxx | Load memory word to $AC$ |
| STA | 3xxx | Bxxx | Store content of $AC$ in memory |
| BUN | 4xxx | Cxxx | Branch unconditionally |
| BSA | 5xxx | Dxxx | Branch and save return address |
| ISZ | 6xxx | Exxx | Increment and skip if zero |
| CLA | 7800 | | Clear $AC$ |
| CLE | 7400 | | Clear $E$ |
| CMA | 7200 | | Complement $AC$ |
| CME | 7100 | | Complement $E$ |
| CIR | 7080 | | Circulate right $AC$ and $E$ |
| CIL | 7040 | | Circulate left $AC$ and $E$ |
| INC | 7020 | | Increment $AC$ |
| SPA | 7010 | | Skip next instruction if $AC$ positive |
| SNA | 7008 | | Skip next instruction if $AC$ negative |
| SZA | 7004 | | Skip next instruction if $AC$ zero |
| SZE | 7002 | | Skip next instruction if $E$ is 0 |
| HLT | 7001 | | Halt computer |
| INP | F800 | | Input character to $AC$ |
| OUT | F400 | | Output character from $AC$ |
| SKI | F200 | | Skip on input flag |
| SKO | F100 | | Skip on output flag |
| ION | F080 | | Interrupt on |
| IOF | F040 | | Interrupt off |

Below is the complete operation that takes place during the instruction:

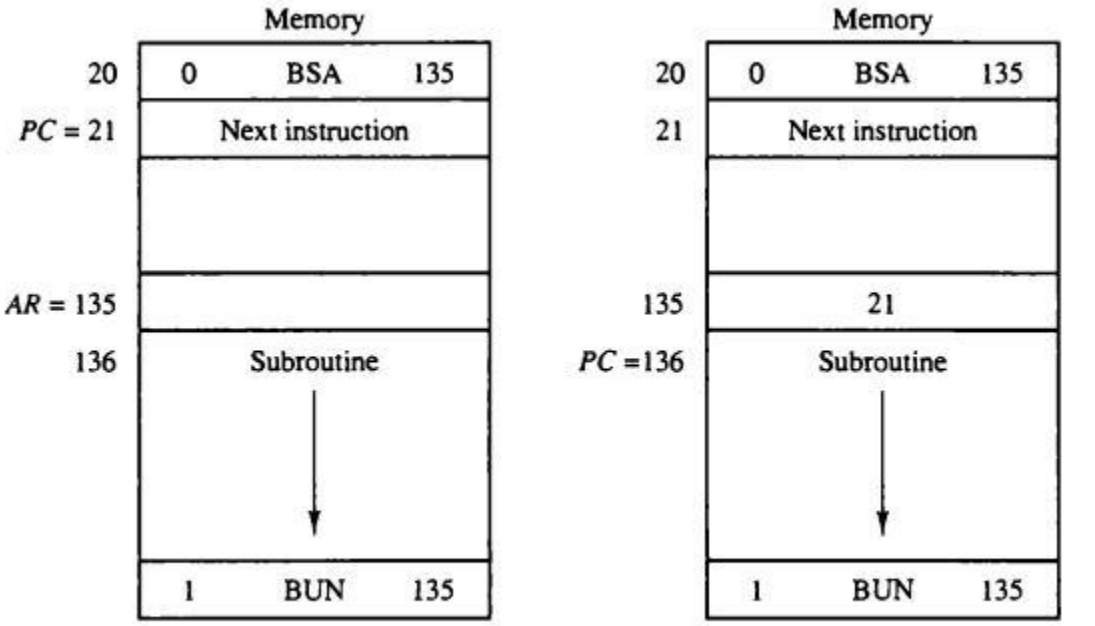
| Symbol | Operation decoder | Symbolic description |
|---|---|---|
| AND | $D_0$ | $AC \leftarrow AC \wedge M[AR]$ |
| ADD | $D_1$ | $AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$ |
| LDA | $D_2$ | $AC \leftarrow M[AR]$ |
| STA | $D_3$ | $M[AR] \leftarrow AC$ |
| BUN | $D_4$ | $PC \leftarrow AR$ |
| BSA | $D_5$ | $M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$ |
| ISZ | $D_6$ | $M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$ |

Fig: Memory-Reference Instructions

Branch and Save Return Address (BSA)

D5T4: M[AR] <— PC, AR <— AR + 1

D5T5: PC <— AR, SC <— 0




For this example:
M[135] <— 21, AR <— 135 + 1
PC <— 136

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0\text{–}11)$ that specifies the operation]

| | | | |
|---|---|---|---|
| | $r$: | $SC \leftarrow 0$ | Clear $SC$ |
| CLA | $rB_{11}$: | $AC \leftarrow 0$ | Clear $AC$ |
| CLE | $rB_{10}$: | $E \leftarrow 0$ | Clear $E$ |
| CMA | $rB_9$: | $AC \leftarrow \overline{AC}$ | Complement $AC$ |
| CME | $rB_8$: | $E \leftarrow \overline{E}$ | Complement $E$ |
| CIR | $rB_7$: | $AC \leftarrow$ shr $AC$, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$ | Circulate right |
| CIL | $rB_6$: | $AC \leftarrow$ shl $AC$, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$ | Circulate left |
| INC | $rB_5$: | $AC \leftarrow AC + 1$ | Increment $AC$ |
| SPA | $rB_4$: | If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$ | Skip if positive |
| SNA | $rB_3$: | If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$ | Skip if negative |
| SZA | $rB_2$: | If $(AC = 0)$ then $PC \leftarrow PC + 1$ | Skip if $AC$ zero |
| SZE | $rB_1$: | If $(E = 0)$ then $(PC \leftarrow PC + 1)$ | Skip if $E$ zero |
| HLT | $rB_0$: | $S \leftarrow 0$ ($S$ is a start–stop flip-flop) | Halt computer |

Fig: Register-Reference Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

| | | | |
|---|---|---|---|
| | $p$: | $SC \leftarrow 0$ | Clear $SC$ |
| INP | $pB_{11}$: | $AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$ | Input character |
| OUT | $pB_{10}$: | $OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$ | Output character |
| SKI | $pB_9$: | If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$ | Skip on input flag |
| SKO | $pB_8$: | If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$ | Skip on output flag |
| ION | $pB_7$: | $IEN \leftarrow 1$ | Interrupt enable on |
| IOF | $pB_6$: | $IEN \leftarrow 0$ | Interrupt enable off |

Fig: Input-Output Instructions

**Instruction Set Completeness**

An instruction set is said to be complete if it contains sufficient instructions to perform operations in following categories:

Functional Instructions

- Arithmetic, logic, and shift instructions
- Examples: ADD, CMA, INC, CIR, CIL, AND, CLA

Transfer Instructions

- Data transfers between the main memory and the processor registers
- Examples: LDA, STA

Control Instructions

- Program sequencing and control
- Examples: BUN, BSA, ISZ

Input/output Instructions

- Input and output
- Examples: INP, OUT

***Instruction set of Basic computer is complete*** because:

- ADD, CMA (complement), INC can be used to perform addition and subtraction and CIR (circular right shift), CIL (circular left shift) instructions can be used to achieve any kind of shift operations. Addition, subtraction and shifting can be used together to achieve multiplication and division. AND, CMA and CLA (clear accumulator) can be used to achieve any logical operations.
- LDA instruction moves data from memory to register and STA instruction moves data from register to memory.
- The branch instructions BUN, BSA and ISZ together with skip instruction provide the mechanism of program control and sequencing.
- INP instruction is used to read data from input device and OUT instruction is used to send data from processor to output device.

## 3.4 Timing and Control Unit

**Control Unit**

Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them. There are two types of control organization:

*Hardwired* Control

- CU is made up of sequential and combinational circuits to generate the control signals.
- If logic is changed, we need to change the whole circuitry.

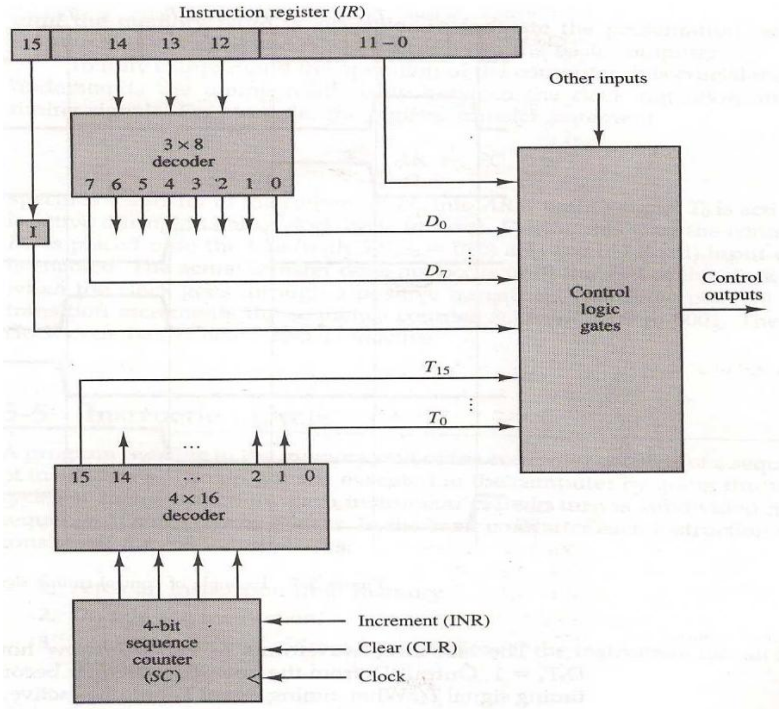
- Expensive
- Fast

*Microprogrammed Control*

- A control memory on the processor contains microprograms that activate the necessary control signals.
- If logic is changed, we only need to change the microprogram.
- Cheap
- Slow

NOTE: Microprogrammed control unit will be discussed in next chapter.

The block diagram of a hardwired control unit is shown below. It consists of two decoders, a sequence counter, and a number of control logic gates.




**Mechanism**:

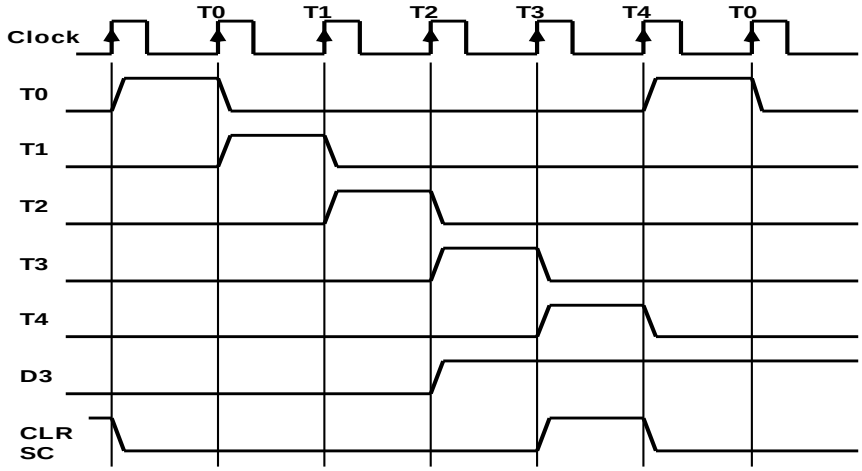
- An instruction read from memory is placed in the instruction resister (IR) where it is decoded into three parts: **I** bit, **operation code** and bits **0 through 11**.
- The operation code bit is decoded with 3 x 8 decoder producing 8 outputs $D_0$ through $D_7$.
- Bit 15 of the instruction is transferred to a flip-flop I.
- And operand bits are applied to control logic gates.
- The 16 outputs of 4-bit sequence counter (SC) are decoded into 16 timing signals $T_0$ through $T_{15}$.

This means instruction cycle of basic computer cannot take more than 16.

Fig: Control unit of a basic computer

**Timing signals**

- Generated by 4-bit sequence counter and 4x16 decoder.
- The SC can be incremented or cleared.
- Example: $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_0$, $T_1$ . . .
  Assume: At time T4, SC is cleared to 0 if decoder output D3 is active: $D_3T_4$: SC ← 0



## 3.5 Instruction Cycle

→ Processing required for complete execution of an instruction is called instruction cycle.

→ In Basic Computer, a machine instruction is executed in the following cycle:

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, control goes back to step 1 to fetch, decode and execute the next instruction. This process is continued indefinitely until HALT instruction is encountered.

**Fetch and Decode**

→ Sequence of steps required for fetching instruction from memory to CPU internal register is known as fetch cycle.

```
T0: AR ← PC  (S0S1S2=010, T0=1)
T1: IR ← M [AR],  PC ← PC + 1   (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)
```

→ For fetching and decoding, the steps are:

i) Initially, PC holds the address of next instruction to fetch. With timing signal $T_0$, address pointed by PC is transferred to the AR.

ii) The processor fetches instruction to IR from memory location referenced by AR and increment PC for next instruction. This happens with timing signal $T_1$.

iii) Processor interprets instruction and performs required action i.e. decoding during time period $T_2$.

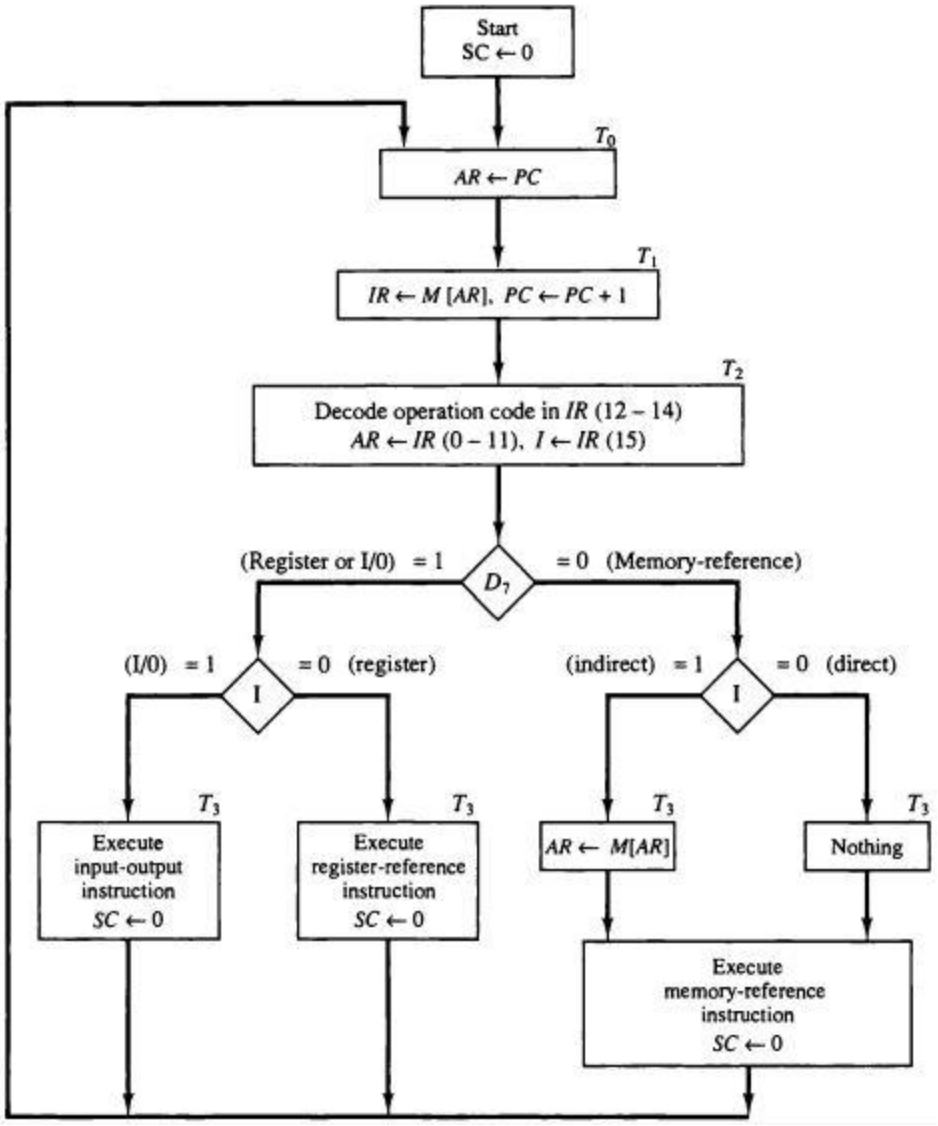



Fig: Flowchart of Instruction Cycle

→ Then, among decoded, D7 determines which type of instruction.

i) If D7 = 1, it will be either register-reference or input-output instruction.
   a) If I = 1, input-output instruction is executed during T3.
   b) If I = 0, register-reference instruction is executed during T3.

ii) If D7 = 0, it will be memory-reference instruction.
   a) If I = 1, indirect addressing mode instruction during T3.
   b) If I = 0, direct addressing mode instruction during T3.

→ The SC is reset after executing each instruction.

## 3.6 Input-Output Configuration

The terminal sends and receives serial information. Each quantity of information has 8 bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input—output configuration is shown in figure. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

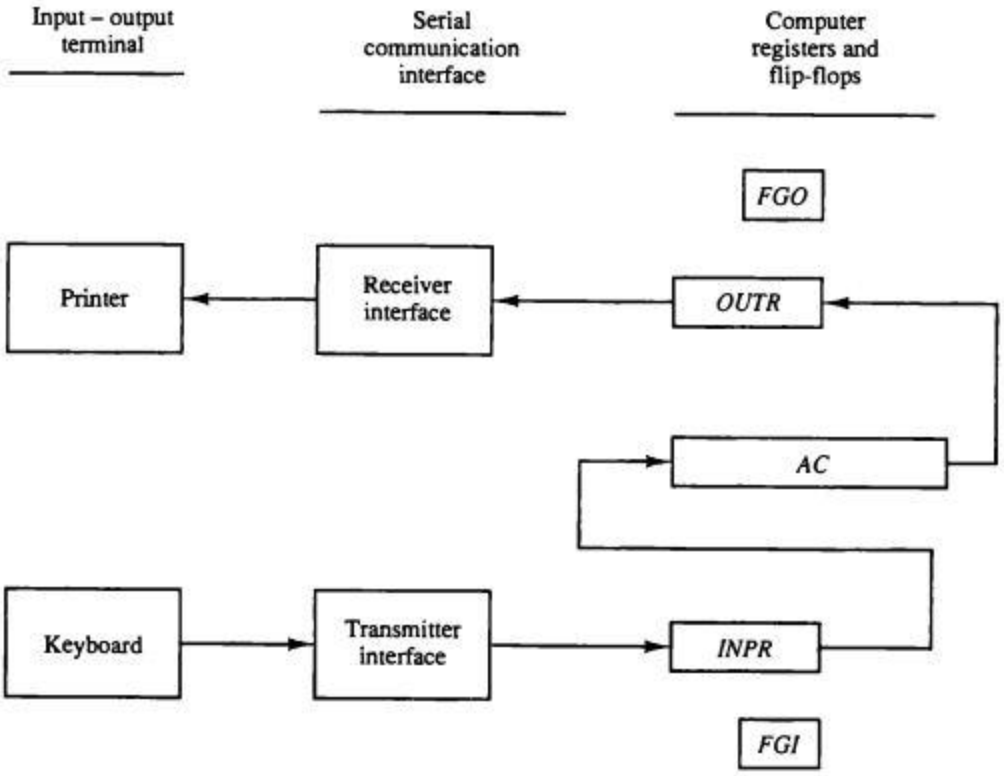



Fig: Input-Output Configuration

Scenario1: when a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The control checks the flag bit, if 1, contents of INPR is transferred in parallel to AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

Scenario2: OUTR works similarly but the direction of information flow is reversed. Initially FGO is set to 1. The computer checks the flag bit; if it is 1, the information is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character and when operation is completed, it sets FGO to 1.

# Chapter 4
# Programming the Basic Computer

## 4.1 Introduction

→ A total computer system includes both hardware and software. Hardware consists of the physical components and all associated equipments. Software refers to the programs that are written for the computer.

→ Writing a program for a computer consists of specifying, directly or indirectly, a sequence of machine instructions (which is in binary pattern).

→ Machine instructions are in binary pattern which may be quite difficult to understand and write. So, it is more preferable to write programs in more familiar symbols of the alphanumeric character set.

→ Thus, there is a need for translating user-oriented symbolic programs into binary programs recognized by the hardware.

→ A program written by user may either be dependent or independent of the physical computer that runs his program.

→ For e.g., a program written in C language is independent (you don't have to take care of hardware) because there is another program called *compiler*, which translates the C program, is machine dependent (it must translate the C program to the binary code recognized by the hardware of the particular computer used).

→ Here, we have 25 instructions of basic computer, discussed in the previous chapter, along with its hardware configuration. There is three-letter symbol for instruction to make people know what it means. We explore ideas on how the programs are written.

→ There are 7 memory-reference instructions which consist of mode (bit 15), op-code (bit 14-12) and address (bit 11-0).

→ The other 18 instructions are register-reference and input-output. They have 16-bit op-code.

→ The letter M refers to the memory word (operand) found at the effective address.

## 4.2 Machine Language

Programs written for a computer may be in one of the following categories:

i) **Binary code**. This is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.

ii) **Octal or hexadecimal code**. This is an equivalent translation of the binary code to octal or hexadecimal representation.

iii) **Symbolic code**. The user employs symbols (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code. Each symbolic instruction can be translated into one binary coded instruction. This translation is done by a special program called an ***assembler***. Because an assembler translates the symbols, this type of symbolic program is referred to as an ***assembly language program***.

iv) **High level programming languages**. These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior. An example of a high level programming language is C. It employs problem oriented symbols and formats. The program is written in a sequence of statements in a form that people prefer to think in when solving a problem. However, each statement must be translated into a sequence of binary instructions before the program can be executed in a computer. The program that translates a high level language program to binary is called a ***compiler***.

Category (i) is of *machine language*. Binary and octal or hexadecimal codes are equivalent so that category (ii) can be called machine language as well. Because of the one-to-one relationship between a symbolic instruction and its binary equivalent, an assembly language is considered to be a machine level language.

C Program to Add Two Numbers

```
main()
 {
        int a,b,c;
        a = 83;
        b = -23;
        c = a + b;
 }
```

| Symbol | Hexadecimal code | Description |
|---|---|---|
| AND | 0 or 8 | AND $M$ to $AC$ |
| ADD | 1 or 9 | Add $M$ to $AC$, carry to $E$ |
| LDA | 2 or A | Load $AC$ from $M$ |
| STA | 3 or B | Store $AC$ in $M$ |
| BUN | 4 or C | Branch unconditionally to $m$ |
| BSA | 5 or D | Save return address in $m$ and branch to $m + 1$ |
| ISZ | 6 or E | Increment $M$ and skip if zero |
| CLA | 7800 | Clear $AC$ |
| CLE | 7400 | Clear $E$ |
| CMA | 7200 | Complement $AC$ |
| CME | 7100 | Complement $E$ |
| CIR | 7080 | Circulate right $E$ and $AC$ |
| CIL | 7040 | Circulate left $E$ and $AC$ |
| INC | 7020 | Increment $AC$, |
| SPA | 7010 | Skip if $AC$ is positive |
| SNA | 7008 | Skip if $AC$ is negative |
| SZA | 7004 | Skip if $AC$ is zero |
| SZE | 7002 | Skip if $E$ is zero |
| HLT | 7001 | Halt computer |
| INP | F800 | Input information and clear flag |
| OUT | F400 | Output information and clear flag |
| SKI | F200 | Skip if input flag is on |
| SKO | F100 | Skip if output flag is on |
| ION | F080 | Turn interrupt on |
| IOF | F040 | Turn interrupt off |

Some Examples

Assembly Language Program to Add Two Numbers

```
        ORG 0       /Origin of program is location 0
        LDA A       /Load operand from location A
        ADD B       /Add operand from location B
        STA C       /Store sum in location C
        HLT         /Halt computer
A,      DEC 83      /Decimal operand
B,      DEC -23     /Decimal operand
C,      DEC 0       /Sum stored in location C
        END         /End of symbolic program
```

Program with Symbolic Operation Codes

| Location | Instruction | Comments |
|---|---|---|
| 000 | LDA 004 | Load first operand into *AC* |
| 001 | ADD 005 | Add second operand to *AC* |
| 002 | STA 006 | Store sum in location 006 |
| 003 | HLT | Halt computer |
| 004 | 0053 | First operand |
| 005 | FFE9 | Second operand (negative) |
| 006 | 0000 | Store sum here |

Hexadecimal Program to Add Two Numbers

| Location | Instruction |
|---|---|
| 000 | 2004 |
| 001 | 1005 |
| 002 | 3006 |
| 003 | 7001 |
| 004 | 0053 |
| 005 | FFE9 |
| 006 | 0000 |

Binary Program to Add Two Numbers

| Location | Instruction code |
|---|---|
| 0 | 0010 0000 0000 0100 |
| 1 | 0001 0000 0000 0101 |
| 10 | 0011 0000 0000 0110 |
| 11 | 0111 0000 0000 0001 |
| 100 | 0000 0000 0101 0011 |
| 101 | 1111 1111 1110 1001 |
| 110 | 0000 0000 0000 0000 |

## 4.3 Assembly Language

→ The basic unit of an assembly language program is a line of code. The specific language is defined by a set of rules that specify the symbols that can be used and how they may be combined to form a line of code.

**Rules of the Language**

Each line of an assembly language program is arranged in three columns called fields which specify the following information:

i) The ***label field*** may be empty or it may specify a symbolic address.
ii) The ***instruction field*** specifies a machine instruction or a pseudo instruction. '
iii) The ***comment field*** may be empty or it may include a comment.

→ A symbolic address, which is terminated by a comma, consists of one, two, or three, but not more than three alphanumeric characters. The first character must be a letter; the next two may be letters or numerals. A symbolic address in the instruction field specifies the memory location of an operand.

→ The instruction field in an assembly language program may specify one of the following items:

i) A memory reference instruction (**MRI**)
   E.g. ADD OPR
   ADD PTR I
ii) A register reference or input-output instruction (**non MRI**)
   E.g. CLA
iii) A pseudo instruction with or without an operand
   E.g. ORG 0
   END

→ A MRI occupies two or three symbols separated by spaces. The first must be a three letter symbol defining an MRI operation code, the second is symbolic address and the third symbol, which may or may not be

present, is the letter ***I***. If I is missing, it denotes direct address instruction. If I is present, it denotes indirect address instruction.

→ A non MRI is defined as an instruction that does not have an address part. It consists of only operation code.

→ A pseudo instruction is not a machine instruction but rather an instruction to the assembler giving information about some phase of the translation.

Definition of Pseudo instructions

| Symbol | Information for the Assembler |
|---|---|
| ORG N | Hexadecimal number N is the memory location for the instruction or operand listed in the following line |
| END | Denotes the end of symbolic program |
| DEC N | Signed decimal number N to be converted to binary |
| HEX N | Hexadecimal number N to be converted to binary |

→ A line of code may or may not have a comment, but if it has, it must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are inserted for explanation purposes only and are neglected during the binary translation process.

→ Example:

Assembly Language Program to Subtract Two Numbers

```
        ORG 100     /Origin of program is location 100
        LDA SUB     /Load subtrahend to AC
        CMA         /Complement AC
        INC         /Increment AC
        ADD MIN     /Add minuend to AC
        STA DIF     /Store difference
        HLT         /Halt computer
MIN,    DEC 83      /Minuend
SUB,    DEC -23     /Subtrahend
DIF,    HEX 0       /Difference stored here
        END         /End of symbolic program
```

Listing of Translated Program

| Hexadecimal code | | | |
|---|---|---|---|
| Location | Content | | Symbolic program |
| | | | ORG 100 |
| 100 | 2107 | | LDA SUB |
| 101 | 7200 | | CMA |
| 102 | 7020 | | INC |
| 103 | 1106 | | ADD MIN |
| 104 | 3108 | | STA DIF |
| 105 | 7001 | | HLT |
| 106 | 0053 | MIN, | DEC 83 |
| 107 | FFE9 | SUB, | DEC −23 |
| 108 | 0000 | DIF, | HEX 0 |
| | | | END |

## 4.4 **The Assembler**

An *assembler* is a program that accepts a symbolic language program and produces its binary machine language equivalent. The input symbolic program is called the *source program* and the resulting binary program is called the *object program*. The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

**Representation of Symbolic Program in Memory**

→ The user types the symbolic program on a terminal. A loader program is used to input the characters of the symbolic program into memory.

→ In the basic computer, each character is represented by an 8-bit code. The high-order bit is always 0 and the other seven bits are as specified by ASCII.

Table: ASCII code (in Hexadecimal code) for characters

| Character | Code | Character | Code | Character | Code | |
|---|---|---|---|---|---|---|
| A | 41 | Q | 51 | 6 | 36 | |
| B | 42 | R | 52 | 7 | 37 | |
| C | 43 | S | 53 | 8 | 38 | |
| D | 44 | T | 54 | 9 | 39 | |
| E | 45 | U | 55 | space | 20 | |
| F | 46 | V | 56 | ( | 28 | |
| G | 47 | W | 57 | ) | 29 | |
| H | 48 | X | 58 | * | 2A | |
| I | 49 | Y | 59 | + | 2B | |
| J | 4A | Z | 5A | , | 2C | |
| K | 4B | 0 | 30 | − | 2D | |
| L | 4C | 1 | 31 | . | 2E | |
| M | 4D | 2 | 32 | / | 2F | |
| N | 4E | 3 | 33 | = | 3D | |
| O | 4F | 4 | 34 | CR | 0D | (carriage |
| P | 50 | 5 | 35 | | | return) |

→ The code for CR is produced when the return key is depressed. This causes the "carriage" to return to its position to start typing a new line. The assembler recognizes a CR code as the end of a line of code.

→ A line of code is stored in consecutive memory locations with two characters in each location since a memory word has a capacity of 16 bits.

→ A label symbol is terminated with a comma. Operation and address symbols are terminated with a space and the end of the line is recognized by the CR code.

→ For example, the following line of code:

PL3, LDA SUB I

| Memory word | Symbol | Hexadecimal code | Binary representation |
|---|---|---|---|
| 1 | P L | 50 4C | 0101 0000 0100 1100 |
| 2 | 3 , | 33 2C | 0011 0011 0010 1100 |
| 3 | L D | 4C 44 | 0100 1100 0100 0100 |
| 4 | A | 41 20 | 0100 0001 0010 0000 |
| 5 | S U | 53 55 | 0101 0011 0101 0101 |
| 6 | B | 42 20 | 0100 0010 0010 0000 |
| 7 | I CR | 49 0D | 0100 1001 0000 1101 |

Fig: Computer Representation of the Line of Code: PL3, LDA SUB I

→ Each symbol is terminated by the code for space (20) except for the last symbol, which is terminated by the code of carriage return (OD).

→ If the line of code has a comment, the assembler recognizes it by the code for a slash (2F). The assembler neglects all characters in the comment field and keeps checking for a CR code. When this code is encountered, it replaces the space code after the last symbol in the line of code.

→ This input is scanned by the assembler twice to produce the equivalent binary program. The binary program constitutes the output generated by the assembler.

**First Pass**

→ A two pass assembler scans the entire symbolic program twice. During the first pass, it generates a table that correlates all user defined address symbols with their binary equivalent value.

→ The binary translation is done during the second pass.

→ To keep track of the location of instructions, the assembler uses a memory word called a location counter (abbreviated LC). The content of LC stores the value of the memory location assigned to the instruction or operand presently being processed.

→ The ORG pseudo-instruction initializes the location counter to the value of the first location. Since instructions are stored in sequential locations, the content of LC is incremented by 1 after processing each line of code. To avoid ambiguity in case ORG is missing, the assembler sets the location counter to 0 initially.

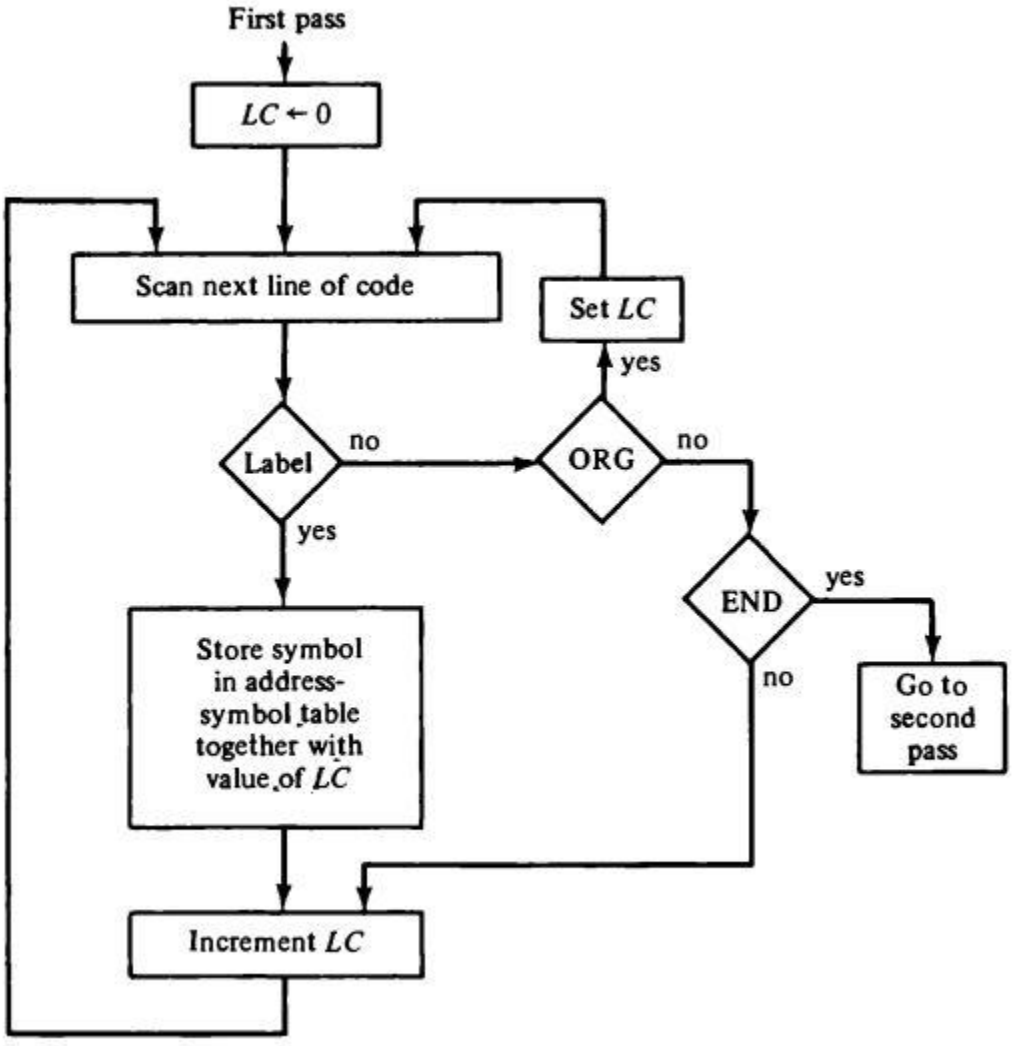



Fig: Flowchart for first pass of assembler.

| Memory word | Symbol or (LC)* | Hexadecimal code | Binary representation |
|---|---|---|---|
| 1 | M I | 4D 49 | 0100 1101 0100 1001 |
| 2 | N , | 4E 2C | 0100 1110 0010 1100 |
| 3 | (LC) | 01 06 | 0000 0001 0000 0110 |
| 4 | S U | 53 55 | 0101 0011 0101 0101 |
| 5 | B , | 42 2C | 0100 0010 0010 1100 |
| 6 | (LC) | 01 07 | 0000 0001 0000 0111 |
| 7 | D I | 44 49 | 0100 0100 0100 1001 |
| 8 | F , | 46 2C | 0100 0110 0010 1100 |
| 9 | (LC) | 01 08 | 0000 0001 0000 1000 |

* (LC) designates content of location counter.

Table: Address Symbol Table for Above Subtraction Program

→ There are three symbols MIN, SUB, DIF each followed by comma (,).

**Second Pass**

→ Machine instructions are translated during the second pass by means of table lookup procedures.

→ A table lookup procedure is a search of table entries to determine whether a specific item matches one of the items stored in the table. The assembler uses four tables.

i) Pseudo-instruction table.
The entries of the pseudo-instruction table are the four symbols ORG, END, DEC, and HEX. Each entry refers the assembler to a subroutine that processes the pseudo-instruction when encountered in the program.

ii) MRI table.
The MRI table contains the seven symbols of the memory reference instructions and their 3 bit operation code equivalent.

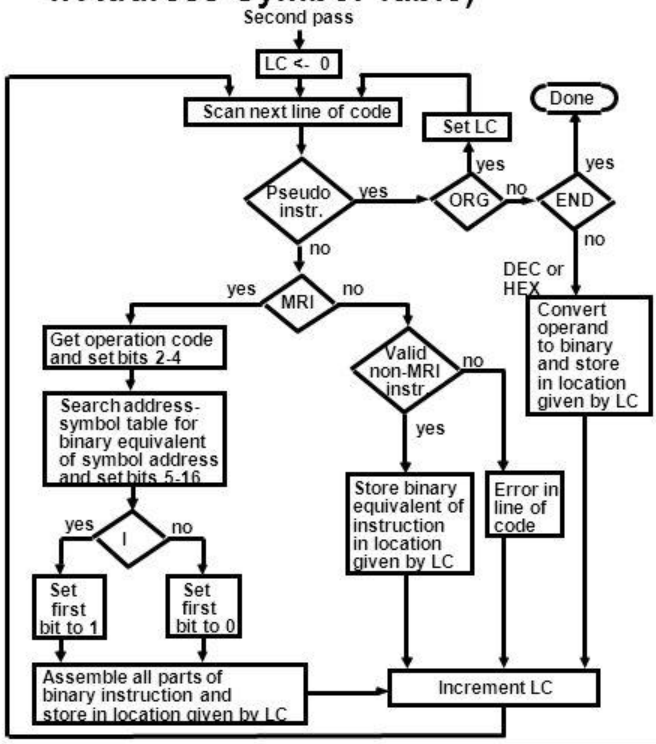
iii) Non MRI table.
The non MRI table contains the symbols for the 18 register reference and input output instructions and their 16 bit binary code equivalent.

iv) Address symbol table.
The address symbol table is generated during the first pass of the assembly process. The assembler searches these tables to find the symbol that it is currently processing in order to determine its binary value.

## ASSEMBLER - SECOND PASS -

**Second Pass**

**Machine instructions are translated by means of table-lookup procedures;**
**(1. Pseudo-Instruction Table, 2. MRI Table, 3. Non-MRI Table**
**4. Address Symbol Table)**






## 4.5 Program Loops

An example of a C program that forms the sum of 100 integer numbers:

```
int  i, sum=0, a[100];
for (i=1; i<=100; i++)
{
        sum = sum + a[i];
}
```

The same program in Symbolic code:

```
Line
 1          ORG 100      /Origin of program is HEX 100
 2          LDA ADS      /Load first address of operands
 3          STA PTR      /Store in pointer
 4          LDA NBR      /Load minus 100
 5          STA CTR      /Store in counter
 6          CLA          /Clear accumulator
 7    LOP,  ADD PTR I    /Add an operand to AC
 8          ISZ PTR      /Increment pointer
 9          ISZ CTR      /Increment counter
10          BUN LOP      /Repeat loop again
11          STA SUM      /Store sum
12          HLT          /Halt
13    ADS,  HEX 150      /First address of operands
14    PTR,  HEX 0        /This location reserved for a pointer
15    NBR,  DEC -100     /Constant to initialized counter
16    CTR,  HEX 0        /This location reserved for a counter
17    SUM,  HEX 0        /Sum is stored here
18          ORG 150      /Origin of operands is HEX 150
19          DEC 75       /First operand
 .
 .
 .
118         DEC 23       /Last operand
119         END          /End of symbolic program
```

→ Suppose that the assembler reserves locations $(150)_{16}$ to $(1B3)_{16}$ for the 100 operands which are listed in lines 19 to 118. This is done by the ORG pseudo-instruction in line 18, which specifies the origin of the operands.

→ The first and last operands are listed with a specific decimal number, although these values are not known during compilation.

→ Try to understand the program with reference to your book. Or, let's discuss in class.

## 4.6 Programming Arithmetic and Logic Operations

→ Some computers have machine instructions to add, subtract, multiply, and divide. Others, such as the basic computer, have only one arithmetic instruction, such as ADD. Operations not included in the set of machine instructions must be implemented by a program.

→ Operations that are implemented in a computer with one machine instruction are said to be implemented by hardware. Operations implemented by a set of instructions that constitute a program are said to be implemented by software.

→ Hardware implementation is more costly because of the additional circuits needed to implement the operation. Software implementation results in long programs both in number of instructions and in execution time.

→ This section demonstrates the software implementation of a few arithmetic and logic operations. Programs can be developed for any arithmetic operation and not only for fixed point binary data but for decimal and

floating-point data as well. The hardware implementation of arithmetic operations is carried out in coming chapter.

**Multiplication Program**

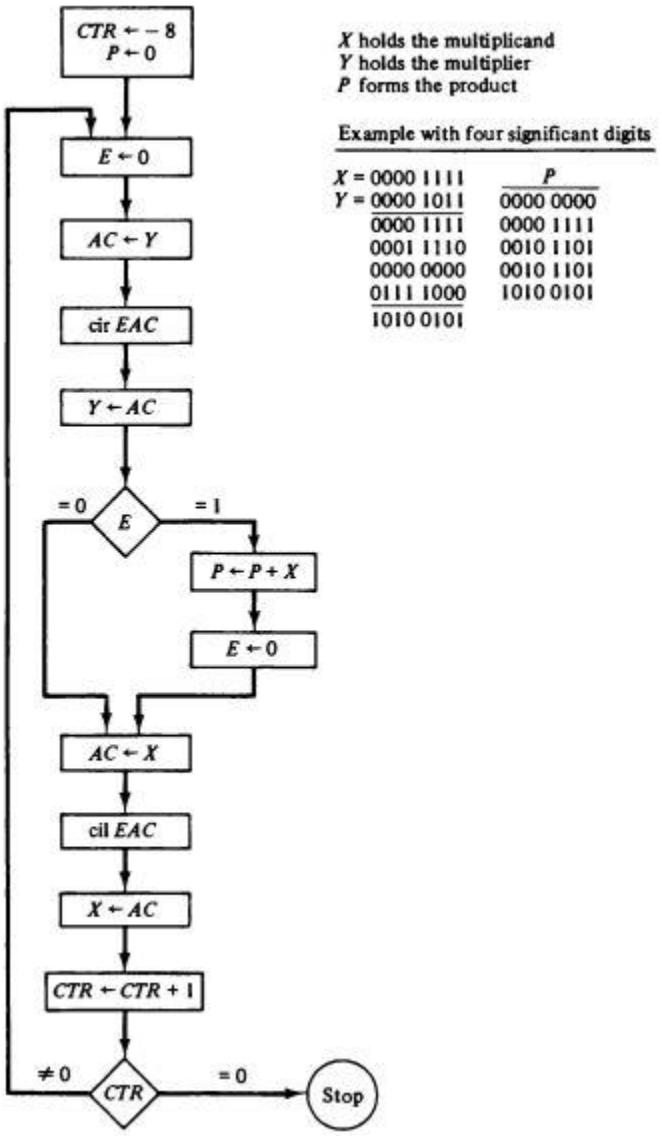



Fig: Flowchart for multiplication program.

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /Transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDA X       /Load multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```

Program to Multiply Two Positive Numbers

**Double-precision Addition**

→ Memory unit can store 16-bit data only. So, addition of 16-bit is easy. The 32-bit data, called double precision data, of which addition can be performed through programming.

→ Let's take an example:

2AB4E479 + ACD1792E = (?)

| | |
|---|---|
| AL | E479 |
| AH | 2AB4 |
| BL | 792E |
| BH | ACD1 |
| CL | Result from Addition of lower 16-bit |
| CH | Result from Addition of lower 16-bit |

Program to Add Two Double Precision Numbers

```
        LDA AL      /Load A low
        ADD BL      /Add B low, carry in E
        STA CL      /Store in C low
        CLA         /Clear AC
        CIL         /Circulate to bring carry into AC(16)
        ADD AH      /Add A high and carry
        ADD BH      /Add B high
        STA CH      /Store in C high
        HLT
AL,     —           /Location of operands
AH,     —
BL,     —
BH,     —
CL,     —
CH,     —
```

**Logic Operations**

→ The basic computer has three machine instructions that perform logic operations: AND, CMA, and CLA.

→ The LDA instruction may be considered as a logic operation that transfers a logic operand into the AC.

→ All 16 logic operations can be implemented by software means because any logic function can be implemented using the AND and complement operations. For example, the OR operation is not available as a machine instruction in the basic computer.

Program to perform A' ANDing with B'

```
LDA A     Load first operand A
CMA       Complement to get Ā
STA TMP   Store in a temporary location
LDA B     Load second operand B
CMA       Complement to get B̄
AND TMP   AND with Ā to get Ā ∧ B̄
```

Q. Program to perform A ORing with B

(Hint: From DeMorgan's theorem, we recognize the relation x + y = (x'y')'.)

→ All 16 logic operations can be implemented by software means because any logic function can be implemented using the AND and complement operations.

**Shift Operations**

→ The circular shift operations are machine instructions in the basic computer.

→ The other shifts of interest are the logical shifts and arithmetic shifts. These two shifts can be programmed with a small number of instructions.

→ The logical shift requires that zeros be added to the extreme positions. This is easily accomplished by clearing E and circulating the AC and E. Thus, for a logical shift right operation we need the two instructions.

```
CLE
CIR
```

Q. Do for logical shift left.

→ For the arithmetic right shift requires that we set E to the same value as the sign bit and circulate right, thus:

```
CLE    /Clear E to 0
SPA    /Skip if AC is positive; E remains 0
CME    /AC is negative; set E to 1
CIR    /Circulate E and AC
```

Q. Do for arithmetic left shift.

## 4.7 Subroutines

→ A set of common instructions that can be used in a program many times is called a *subroutine*.

→ Each time that a subroutine is used in the main part of the program, a branch is executed to the beginning of the subroutine. After the subroutine has been executed, a branch is made back to the main program.

→ A subroutine consists of a self contained sequence of instructions that carries out a given task. A branch can be made to the subroutine from any part of the main program.

→ This poses the problem of how the subroutine knows which location to return to, since many different locations in the main program may make branches to the same subroutine. It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return.
→ In the basic computer, the link between the main program and a subroutine is the BSA instruction (branch and save return address).
→ The last instruction in the subroutine returns the computer to the main program. This is accomplished by the indirect branch instruction with an address symbol identical to the symbol used for the subroutine name. This is done by the BUN instruction (branch unconditionally).
→ The procedure for branching to a subroutine and returning to the main program is referred to as a subroutine *linkage*. The BSA instruction performs an operation commonly called subroutine *call*. The last instruction of the subroutine performs an operation commonly called subroutine *return*.

**An example**

```
Location
                      ORG 100     /Main program
100                   LDA X       /Load X
101                   BSA SH4     /Branch to subroutine
102                   STA X       /Store shifted number
103                   LDA Y       /Load Y
104                   BSA SH4     /Branch to subroutine again
105                   STA Y       /Store shifted number
106                   HLT
107         X,        HEX 1234
108         Y,        HEX 4321
                                  /Subroutine to shift left 4 times
109         SH4,      HEX 0       /Store return address here
10A                   CIL         /Circulate left once
10B                   CIL
10C                   CIL
10D                   CIL         /Circulate left fourth time
10E                   AND MSK     /Set AC(13–16) to zero
10F                   BUN SH4 I   /Return to main program
110         MSK,      HEX FFF0    /Mask operand
                      END
```

## 4.8 Input Output Programming

→ A binary-coded character enters the computer when an INP (input) instruction is executed.
→ A binary-coded character is transferred to the output device when an OUT (output) instruction is executed. The output device detects the binary code and types the corresponding character.
→ The SKI instruction checks the input flag to see if a character is available for transfer. The next instruction is skipped if the input flag bit is 1. The INP instruction transfers the binary coded character into AC(0-7).
→ The character is then printed by means of the OUT instruction.
→ If the SKI instruction finds the flag bit at 0, the next instruction in sequence is executed. This instruction is a branch to return and check the flag bit again.
→ If the output flag is 0, the computer remains in a two instruction loop checking the flag bit. When the flag changes to 1, the character is transferred from the accumulator to the printer.

**Programs to Input and Output One Character**

(a) Input a character:

```
CIF,    SKI       /Check input flag
        BUN CIF   /Flag=0, branch to check again
        INP       /Flag=1, input character
        OUT       /Print character
        STA CHR   /Store character
        HLT
CHR,    —         /Store character here
```

(b) Output one character:

```
        LDA CHR   /Load character into AC
COF,    SKO       /Check output flag
        BUN COF   /Flag=0, branch to check again
        OUT       /Flag=1, output character
        HLT
CHR,    HEX 0057  /Character is "W"
```

# Chapter 5
# Microprogrammed Control

The function of the control unit in a digital computer is to initiate sequences of microoperations. The number of different types of rnicrooperations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of microoperations that are performed. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired. Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the microoperation sequences in a digital computer.
A computer that employs a microprogrammed control unit will have two separate memories: a main memory and a control memory.

## 5.1 Control Memory

**Control Memory (Control Storage: CS):** Storage in the microprogrammed control unit to store the microprogram.

**Control word:** It is a string of control variables (0's and 1's) occupying a word in control memory.

**Microprogram**
- ✓ Program stored in control memory that generates all the control signals required to execute the instruction set correctly
- ✓ Consists of microinstructions

**Microinstruction**
- ✓ Contains a control word and a sequencing word
- ✓ Control Word – contains all the control information required for one clock cycle
- ✓ Sequencing Word - Contains information needed to decide the next microinstruction address

**Microoperation**
- ✓ A microinstruction contains one or more microoperations to be completed.

**Writable Control Memory (Writable Control Storage: WCS)**
- ✓ CS whose contents can be modified:
  - Microprogram can be changed
  - Instruction set can be changed or modified

A computer that employs a microprogrammed control unit will have two separate memories: main memory and a control memory. The user's program in main memory consists of machine instructions and data whereas control memory holds a fixed microprogram that cannot be altered by the user. Each machine instruction initiates a series of microinstructions in control memory.
The general configuration of a microprogrammed control unit is demonstrated in the following block diagram:

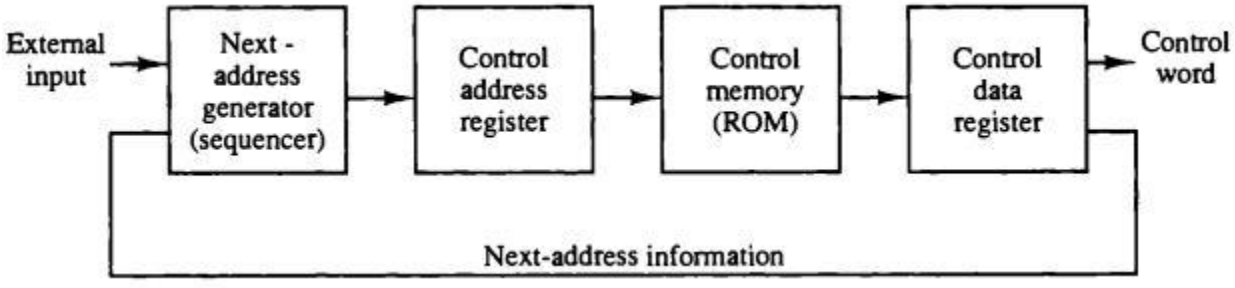



Fig: Microprogrammed control organization

**Dynamic Microprogramming**
- ✓ Computer system whose control unit is implemented with a microprogram in WCS.
- ✓ Microprogram can be changed by a systems programmer or a user.

**Sequencer:** The device or program that generates address of next microinstruction to be executed is called sequencer. While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.

The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

**Control Address Register:** CAR contains address of microinstruction.

**Control Data Register:** CDR contains microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations. The data register is sometimes called a *pipeline register.*
It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

## 5.2 Address Sequencing

Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. Process of finding address of next microinstruction to be executed is called ***address sequencing***. The address sequencing capabilities required in a control memory are:

i. Incrementing of the control address register.
ii. Unconditional branch or conditional branch, depending on status bit conditions.
iii. A mapping process from the bits of the instruction to an address for control memory.
iv. A facility for subroutine call and return.

Following is the block diagram for control memory and the associated hardware needed for selecting the next microinstruction address.

The diagram shows four different paths from which the control address register (CAR) receives the address. The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return from the subroutine.

Control address register receives address of next microinstruction from different sources.
- ✓ Incrementer simply increments the address by one
- ✓ In case of branching, branch address is specified in one of the field of microinstruction.
- ✓ In case of subroutine call, return address is stored in the register SBR which is used when returning from called subroutine.

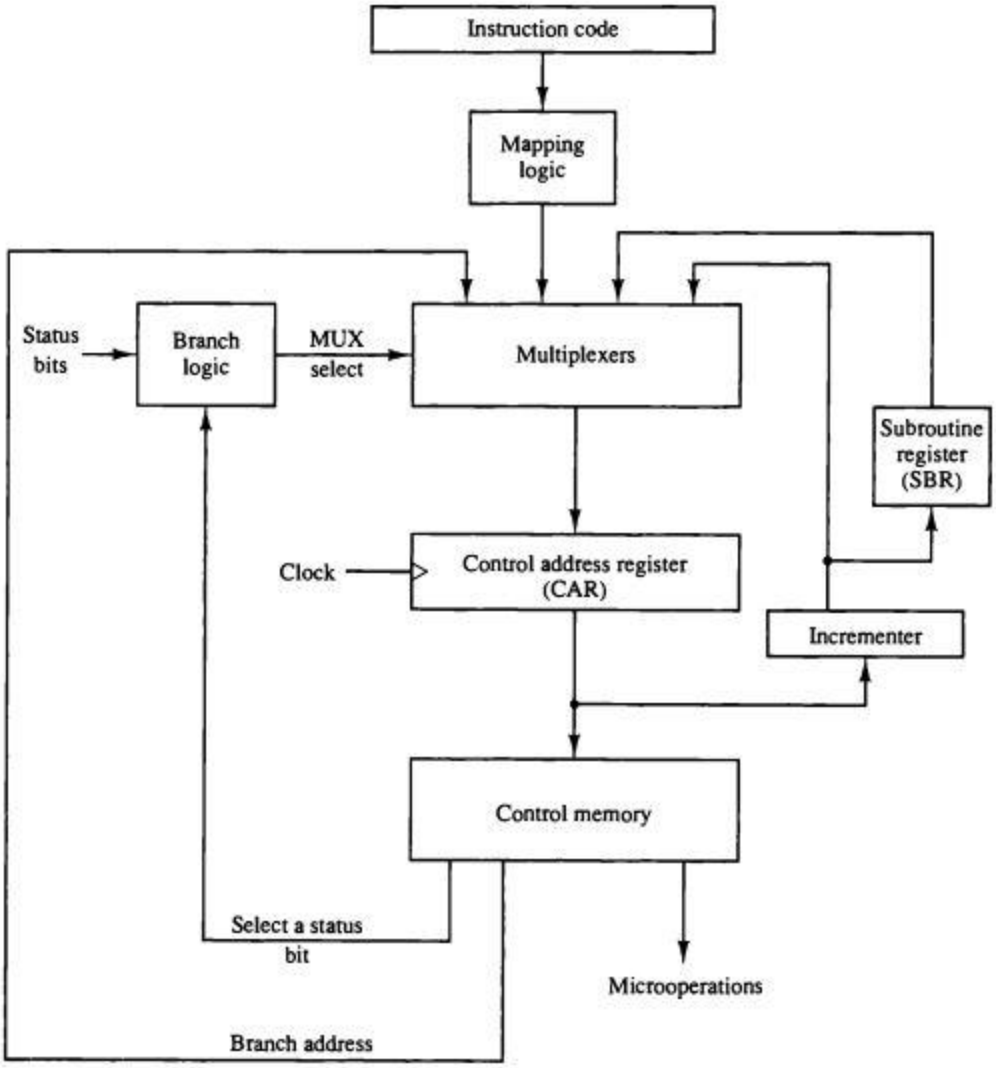



Fig: Block diagram of address sequencer.

**Conditional Branch**

Simplest way of implementing branch logic hardware is to test the specified condition and branch to the indicated address if condition is met otherwise address resister is simply incremented. If Condition is true, hardware set the appropriate field of status register to 1. Conditions are tested for O (overflow), N (negative), Z (zero), C (carry), etc.

**Unconditional Branch**

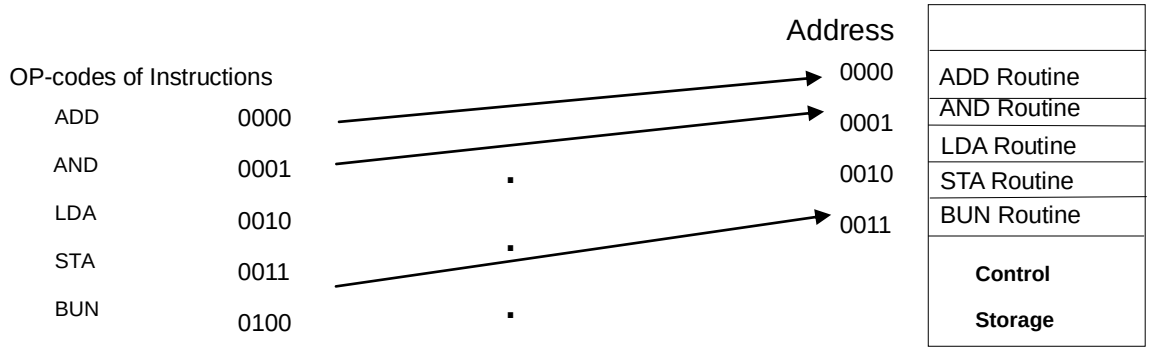
Fix the value of one status bit at the input of the multiplexer to 1. So that, branching can always be done.

**Mapping**

Assuming operation code of 4-bits which can specify 16 ($2^4$) distinct instructions. Assume further and control memory has 128 words, requiring an address of 7-bits. Now we have to map 4-bit operation code into 7-bit control memory address. Thus, we have to map Op-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its subroutine in memory.

**Direct mapping:**

Directly use op-code as address of Control memory




**Another approach of direct mapping:**

Transfer Op-code bits to use it as an address of control memory. In this mapping, one 0 is placed in the MSB and two 0s in the LSB as shown in figure:

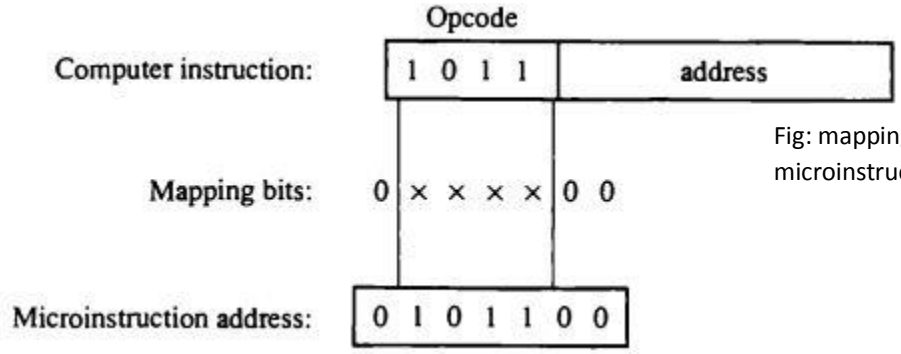
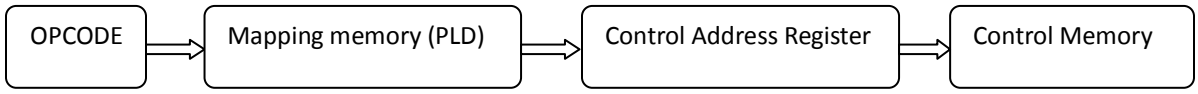



Fig: mapping from instruction code to microinstruction address

**Extended idea: Mapping function implemented by ROM or PLD (Programmable Logic Device)**

Use op-code as address of ROM where address of control memory is stored and then use that address as an address of control memory. This provides flexibility to add instructions for control memory as the need arises.




**Subroutines**

Subroutines are programs that are used by another program to accomplish a particular task. Microinstructions can be saved by employing subroutines that use common sections of micro code.

Example: the sequence of microoperations needed to generate the effective address is common to all memory reference instructions. Thus, this sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Subroutine register is used to save a return address during a subroutine call which is organized in LIFO (last in, first out) stack.

## 5.3 Microprogram Example

Once the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming.

**Computer Configuration**

→ It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram.

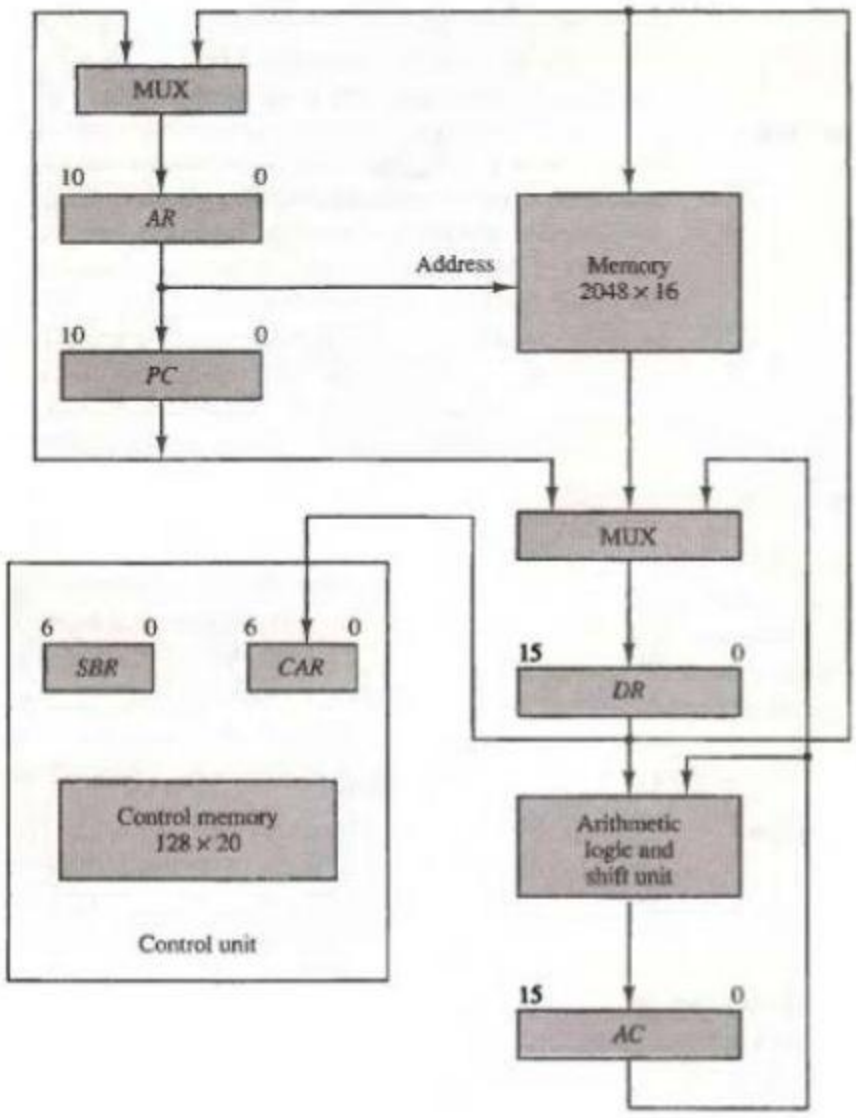



Fig: computer hardware organization

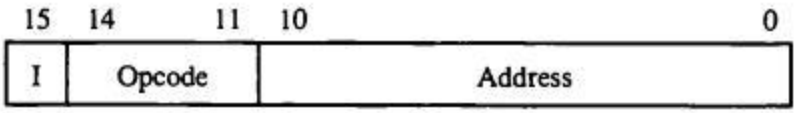
→ Four registers are associated with the processor unit and two with the control unit. The processor registers are PC, AR, DR and AC.

→ The control unit has control address register CAR and subroutine register SBR.

→ The transfer of information among the registers in processor is done through multiplexer rather than a common bus. DR can receive information from AC, PC or memory. AR can receive information from PC or DR. PC can receive information only from AR.

→ The arithmetic, logic and shift unit performs microoperations with data from AC and DR and places the result in AC. Note that memory receives its address from AR. Input data written to memory come from DR, and data read from memory can go only to DR.

→ The computer instruction format has three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit operation code (op-code), and an 11-bit address field. The figure below lists four of the 16 possible memory reference instructions.




(a) Instruction format

| Symbol | Opcode | Description |
|---|---|---|
| ADD | 0000 | $AC \leftarrow AC + M[EA]$ |
| BRANCH | 0001 | If $(AC < 0)$ then $(PC \leftarrow EA)$ |
| STORE | 0010 | $M[EA] \leftarrow AC$ |
| EXCHANGE | 0011 | $AC \leftarrow M[EA], M[EA] \leftarrow AC$ |

EA is the effective address

(b) Four computer instructions

Fig: computer instructions

→ The ADD instruction adds the content of the operand found in the effective address to the content of AC.

→ The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit is a 1.

→ The STORE instruction transfers the content of AC into the memory word specified by the effective address.

→ The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

**Microinstruction Format and Description**

We know the computer instruction format (explained in previous chapter) for different set of instruction in main memory. Similarly, microinstruction in control memory has 20-bit format divided into 4 functional parts as shown below.

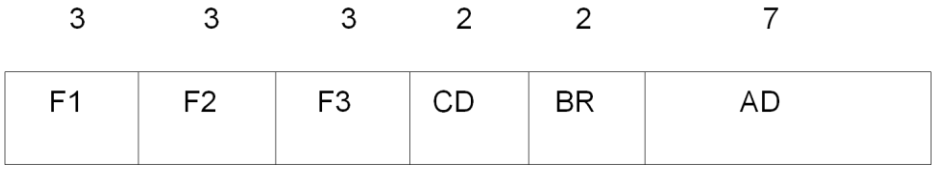



Fig: Microinstruction code format (20 bits)

F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Each microoperation below is defined using resister transfer statements and is assigned a symbol for use in symbolic microprogram.

**Description of CD**

| CD | Condition | Symbol | Comments |
|---|---|---|---|
| 00 | Always = 1 | U | Unconditional branch |
| 01 | $DR(15)$ | I | Indirect address bit |
| 10 | $AC(15)$ | S | Sign bit of $AC$ |
| 11 | $AC = 0$ | Z | Zero value in $AC$ |

**Description of BR**

| BR | Symbol | Function |
|---|---|---|
| 00 | JMP | $CAR \leftarrow AD$ if condition = 1<br>$CAR \leftarrow CAR + 1$ if condition = 0 |
| 01 | CALL | $CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1<br>$CAR \leftarrow CAR + 1$ if condition = 0 |
| 10 | RET | $CAR \leftarrow SBR$ (Return from subroutine) |
| 11 | MAP | $CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$ |

CD (condition) field consists of two bits representing 4 status bits and BR (branch) field (2-bits) used together with address field AD, to choose the address of the next microinstruction.

**Microinstruction fields (F1, F2, F3)**

| F1 | Microoperation | Symbol |
|---|---|---|
| 000 | None | NOP |
| 001 | $AC \leftarrow AC + DR$ | ADD |
| 010 | $AC \leftarrow 0$ | CLRAC |
| 011 | $AC \leftarrow AC + 1$ | INCAC |
| 100 | $AC \leftarrow DR$ | DRTAC |
| 101 | $AR \leftarrow DR(0\text{–}10)$ | DRTAR |
| 110 | $AR \leftarrow PC$ | PCTAR |
| 111 | $M[AR] \leftarrow DR$ | WRITE |

| F2 | Microoperation | Symbol |
|---|---|---|
| 000 | None | NOP |
| 001 | $AC \leftarrow AC - DR$ | SUB |
| 010 | $AC \leftarrow AC \vee DR$ | OR |
| 011 | $AC \leftarrow AC \wedge DR$ | AND |
| 100 | $DR \leftarrow M[AR]$ | READ |
| 101 | $DR \leftarrow AC$ | ACTDR |
| 110 | $DR \leftarrow DR + 1$ | INCDR |
| 111 | $DR(0\text{-}10) \leftarrow PC$ | PCTDR |

| F3 | Microoperation | Symbol |
|---|---|---|
| 000 | None | NOP |
| 001 | $AC \leftarrow AC \oplus DR$ | XOR |
| 010 | $AC \leftarrow \overline{AC}$ | COM |
| 011 | $AC \leftarrow$ shl $AC$ | SHL |
| 100 | $AC \leftarrow$ shr $AC$ | SHR |
| 101 | $PC \leftarrow PC + 1$ | INCPC |
| 110 | $PC \leftarrow AR$ | ARTPC |
| 111 | Reserved | |

Here, microoperations are subdivided into three fields of 3-bits each. These 3 bits are used to encode 7 different microoperations. No more than 3 microoperations can be chosen for a microinstruction, one for each field. If fewer than 3 microoperations are used, one or more fields will contain 000 for no operation.

**E.g. the Fetch Routine**

$$AR \leftarrow PC$$

$$DR \leftarrow M[AR], \quad PC \leftarrow PC + 1$$

$$AR \leftarrow DR(0\text{–}10), \quad CAR(2\text{–}5) \leftarrow DR(11\text{–}14), \quad CAR(0,1,6) \leftarrow 0$$

Microinstructions for Fetch routine

```
        ORG 64
FETCH:  PCTAR          U    JMP    NEXT
        READ, INCPC    U    JMP    NEXT
        DRTAR          U    MAP
```

Assembly language convention

| Binary Address | F1 | F2 | F3 | CD | BR | AD |
|---|---|---|---|---|---|---|
| 1000000 | 110 | 000 | 000 | 00 | 00 | 1000001 |
| 1000001 | 000 | 100 | 101 | 00 | 00 | 1000010 |
| 1000010 | 101 | 000 | 000 | 00 | 11 | 0000000 |

Binary microprogram

# Chapter 6
# Central Processing Unit

→ The part of the computer that performs the bulk of data processing operation is called central processing unit (CPU) which consists of ALU, control unit and register array.

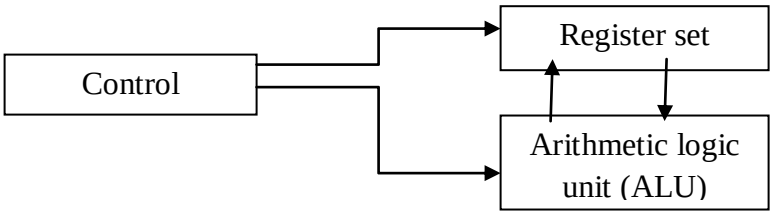



Fig: Major components of CPU

→ CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.
→ The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control (CU) unit supervises the transfer of information among the registers and instructs the ALU as to which operation to be performed.

## 6.1 CPU Organizations

There are three types of CPU organization based on the instruction format

1. Single accumulator organization
   - ✓ In this type of organization all the operations are performed with an implied accumulator register.
   - ✓ Basic computer is the good example of single accumulator organization.
   - ✓ The instruction of this type of organization has an address field
     E.g. ADD X ⇨ AC←AC+M[X] where X is the address of the operand

2. General register organization
   - ✓ When a large number of processor registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfer, but also while performing various microoperations. Hence, it is necessary to provide a common unit that can perform all the arithmetic, logic and shift microoperations in the processor.
   - ✓ In this type of organization the instruction has two or three address field.
     E.g. ADD R1, X ⇨ R1←R1+M[X]
     ADD R1, R2 ⇨ R1←R1+R2
     ADD R1, R2, R3 ⇨ R1←R2+R3

3. Stack organization
   - ✓ Last-in, first-out (LIFO) mechanism.
   - ✓ A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
   - ✓ In this type of organization of CPU, all the operations are performed with stack.
   - ✓ The PUSH and POP instruction only need address field. The operation-type instructions do not need address field.
     E.g. PUSH X ⇨ TOS←M[X]
     POP X ⇨ M[X] ←TOS
     ADD

- ✓ This ADD instruction in the stack organization performs addition of two top of the stack element and stores the result in the top of the stack. First pops two operands from the top of the stack; adds them and stores the result in the top of the stack.

## 6.2 Instruction Formats

| Mode | Op-code | Address |
|---|---|---|

Mode bit: It specifies the way the operand or the effective address is determined.
Op-code field: It specifies the operation to be performed.
Address field: It designates a memory address or a processor register.
→ The length of instruction varies with the variation of number of address in an address field.

There are four types of instruction on the basis of address field they used.

**1.** Three-Address Instruction

- ✓ Computer with three address instruction can use each address field to specify either processor register or memory operand.
- ✓ Advantage –it minimize the size of program
- ✓ Disadvantage –binary coded instruction requires too many bits to specify three address fields
- ✓ E.g. ADD R1, A, B / R1←M[A]+M[B]

Program to evaluate the following arithmetic statement

X = (A+B) * (C+D) using three address fields instruction

```
ADD R1, A, B        / R1←M[A] +M[B]
ADD R2, C, D        / R2←M[C] +M[D]
MUL X, R1, R2       / M[X]←R1*R2
```

**2.** Two-Address Instruction

- ✓ Computer with two address instruction can use each address field to specify either processer register or memory operand
- ✓ Advantage –it minimize the size of instruction
- ✓ Disadvantage –the size of program is relatively larger

Program to evaluate the following arithmetic statement

X = (A+B)*(C+D) using two address field instruction

```
MOV R1, A           / R1←M[A]
ADD R1, B           / R1←R1+M[B]
MOV R2, C           / R2←M[C]
ADD R2, D           / R2←R2+M[D]
MUL R1, R2          / R1←R1*R2
MOV X, R1           / M[X]←R1
```

**3.** One-Address Instruction

- ✓ Execution of one address field instruction use an implied accumulator register for All data manipulation
- ✓ Advantage –relatively small instruction size
- ✓ Disadvantage –relatively large program size

Program to evaluate the following arithmetic statement

X = (A+B)*(C+D) using one address field instruction

```
LOAD A              / AC←M[A]
ADD B               / AC←AC +M[B]
```

```
STORE T          / M[T]←AC
LOAD C           / AC←M[C]
ADD D            / AC←AC+M[D]
MUL T            / AC←AC*M[T]
STORE X          / M[X]←AC
```

**4.** Zero-Address Instruction

- ✓ This type of instruction is used in stack organization computer. There is no address field in this type of instruction except PUSH and POP.
- ✓ Advantage –small instruction size
- ✓ Disadvantages –large the program size

Program to evaluate the following arithmetic statement

X = (A+B)*(C+D) using zero address field instruction

```
PUSH A           / TOS←M[A]
PUSH B           / TOS←M[B]
ADD              / TOS← (A+B)
PUSH C           / TOS←M[C]
PUSH D           / TOS←M[D]
ADD              / TOS←(C+D)
MUL              / TOS← (A+B)*(C+D)
POP X            / M[X] ←TOS
```

## 6.3 Addressing Modes

→ The method of calculating or finding the effective address of the operand in the instruction is called addressing mode.

→ Effective address means the memory address where the required operand is located.

→ The addressing mode specifies the rule interpreting or modifying the address field of the instruction before the operand is actually referenced.

The various addressing modes are:

- **i.** Implied Mode
- **ii.** Immediate Mode
- **iii.** Register Mode
- **iv.** Register Indirect Mode
- **v.** Auto increment or Auto decrement Mode
- **vi.** Direct Address Mode
- **vii.** Indirect Address Mode
- **viii.** Relative Address Mode
- **ix.** Indexed Addressing Mode
- **x.** Base Register Addressing Mode

**Implied Mode**

→ In this type of addressing mode, operands specified implicitly in the definition of instruction.

→ All the register reference instructions that use an accumulator and zero-address instruction in a stack organized computer are implied mode instruction.

E.g. CMA (complement accumulator)

**Immediate Mode**

→ In this addressing mode, the operand is specified in the instruction itself i.e. there is no any address field to represent the operand

→ Immediate mode instructions are useful for initializing register to a constant value.

E.g. LD #NBR / AC←NBR

**Register Mode**

→ In this type of addressing mode, the operands are in the register which is within the CPU.

E.g. LD R1 / AC←R1

**Register Indirect Mode**

→ In this addressing mode, the content of register present in the instruction specifies the effective address of operand.

→ The advantage of this addressing mode is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

E.g. LD (R1) / AC←M[R1]

**Auto Increment or Auto Decrement Mode**

→ In auto increment mode, the content of CPU register is incremented by 1, which gives the effective address of the operand in memory.

E.g. LD (R1)+ / AC←M[R1], R1←R1 + 1

→ In auto decrement mode, the content of CPU register is decremented by 1, which gives the effective address of the operand in memory.

E.g. LD (R1)- / AC←M[R1 - 1]

**Direct Address Mode**

→ In this addressing mode, the address field of an instruction gives the effective address of operand.

E.g. LD ADR / AC←M[ADR]

**Indirect Address Mode**

→ In this addressing mode, the address field of the instruction gives the address of effective address.

E.g. LD @ADR / AC←M[M[ADR]]

**Relative Address Mode**

→ In this addressing mode, the content of program counter is added to the address part of the instruction which gives the effective address of the operand.

→ Assume that the PC is 500 and the address part of instruction is 50. The instruction at location 500 is read from memory during fetch phase and PC is then incremented by 1. Hence, PC is 501, then effective address is 501+50=551.

E.g. LD $ADR / AC←M[PC + ADR]

**Indexed Addressing Mode**

→ In this addressing mode, the content of index register is added to the address field of the instruction which gives the effective address of operand.

→ This type of addressing mode is useful to access the data array, where the address field of an instruction gives the start address of data array and content of index register gives how far the operand from the start address is.

E.g. LD ADR(X) / AC←M[ADR + XR]

**Base Register Addressing Mode**

→ In this addressing mode, the content of the base register is added to the address part of the instruction which gives the effective address of the operand.

E.g. LD ADR(B) / AC←M[ADR + BR]

**Numerical Example**

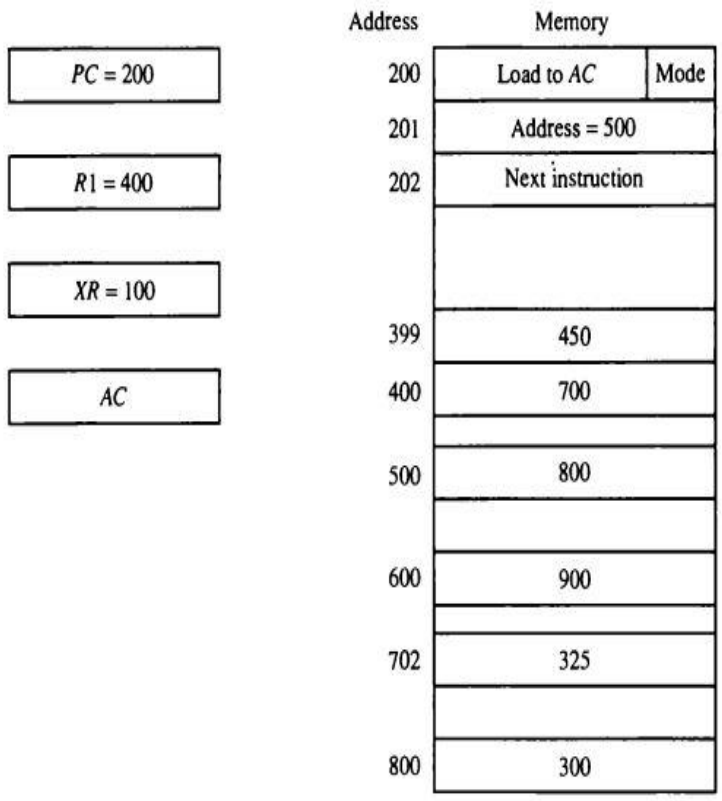



Fig: Numerical example for addressing modes

| Addressing Mode | Effective Address | Content of *AC* |
|---|---|---|
| Direct address | 500 | 800 |
| Immediate operand | 201 | 500 |
| Indirect address | 800 | 300 |
| Relative address | 702 | 325 |
| Indexed address | 600 | 900 |
| Register | — | 400 |
| Register indirect | 400 | 700 |
| Autoincrement | 400 | 700 |
| Autodecrement | 399 | 450 |

Fig: Content of AC after each addressing modes

## 6.4 RISC and CISC characteristics

**RISC (reduced instruction set computer) characteristics**

- → Relatively few instructions
- → Relatively few addressing modes
- → Memory access limited to load and store instructions
- → All operations done within the registers of the CPU
- → Fixed-length, easily decoded instruction format
- → Single-cycle instruction execution
- → The control unit is hardwired rather than micro programmed
- → Relatively large number of registers in the processor unit
- → Efficient instruction pipeline

**CISC (complex instruction set computer) characteristics**

- → A large number of instructions - typically from 100 to 250 instructions
- → Some instructions that perform specialized tasks and are used infrequently
- → A large variety of addressing modes – typically from 5 to 20 different modes
- → Variable-length instruction format
- → Uses memory to load and store instruction and operand as well
- → Instructions that manipulate operands in memory

# Chapter 7
# Pipeline and Vector Processing

## 7.1 Parallel Processing, Flynn's Classification of Computers

**Parallel Processing**

→ Parallel processing is a technique that is used to provide simultaneously data processing task for the purpose of increasing the conceptual speed of computer system.

→ The system may have two or more ALUs and be able to execute two or more instructions at the same time.

→ The purpose of parallel processing is to speed up the computer processing capability and increase its throughput (the amount of task those are completed during given interval of time). It is called parallel computing.

**Flynn's Classification of Computers**

→ There are four groups of computers according to the Flynn's classification based on the number of concurrent instruction and data streams manipulated by the computer system.

→ The normal operation of a computer is to fetch instructions from memory and execute them in processor.

→ The sequence of instructions read from memory constitutes an *instruction stream*.

→ The operations performed on the data in the processor constitute a *data stream*.

→ Parallel processing may occur in the instruction stream, in the data stream, or in both.

→ The Flynn's classification of computer are:

i. single instruction stream, single data stream (**SISD**)
ii. single instruction stream, multiple data stream (**SIMD**)
iii. multiple instruction stream, single data stream (**MISD**)
iv. multiple instruction stream, multiple data stream (**MIMD**)

→ SISD has a processor, a control unit, and a memory unit. There is a single processor which executes a single instruction stream to operate on the data stored in a single memory in this system. The parallel processing in this case may be achieved by means of multiple functional units or pipeline processing.

→ SIMD executes a single machine instruction on different set of data by different processors.

- Each processing element has associated data memory.
- All the processor receives the same instruction from control unit but operate on different items of data.
- Application of SIMD is vector and array processing.

→ MISD has many functional units which perform different operations on the same data. It is a theoretical model of computer.

→ MIMD consists of a set of processors which simultaneously execute different instruction sequences on the different set of data.

## 7.2 Pipelining

→ Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.

→ Each segment performs partial processing dictated by the way the task partitioned.

→ 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.

→ It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.

→ The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Suppose we want to perform the combined multiply and add operations with a stream of numbers.

E.g. $A_i * B_i + C_i$ for i = 1 , 2, 3,……,7

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

| | |
|---|---|
| $R1 \leftarrow A_i$, $R2 \leftarrow B_i$ | Input $A_i$ and $B_i$ |
| $R3 \leftarrow R1*R2$, $R4 \leftarrow C_i$ | Multiply and input $C_i$ |
| $R5 \leftarrow R3+R4$ | Add $C_i$ to product |

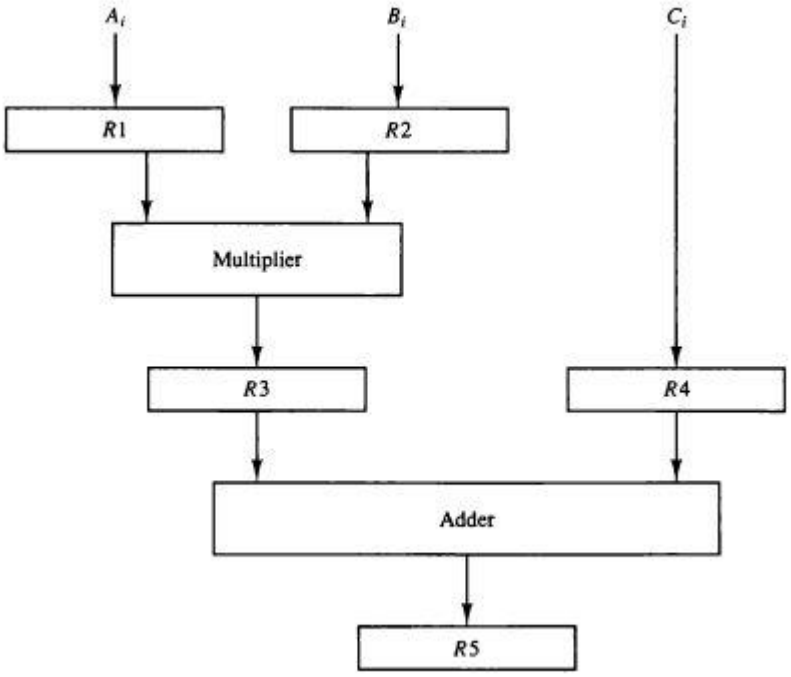



Fig: Example of pipeline processing

| 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*B | C | $A_1*B_1+C_1$ |
| 4 | $A_4$ | $B_4$ | A*B | C | A*B+C |
| 5 | $A_5$ | $B_5$ | A*B | C | A*B+C |
| 6 | $A_6$ | $B_6$ | A*B | C | A*B+C |
| 7 | $A_7$ | $B_7$ | A*B | C | A*B+C |
| 8 | - | - | A*B | C | A*B+C |
| 9 | - | - | - | - | A*B+C |

**Speedup Equation**

→ Consider a K-segment pipeline with a clock cycle time $t_p$ to execute n tasks. The first task $T_1$ require time $Kt_p$ to complete. The remaining (n-1) tasks finish at the rate of one task per click cycle and will be completed after time (n-1) $t_p$. The total time to complete the n task is $[Kt_p + (n - 1) t_p] = (K + n - 1) t_p$.

→ Consider a non–pipeline unit that performs the same operation and takes $t_n$ time to complete each task. The total time required for n tasks would be $nt_n$.

→ The speedup of pipeline processing over an equivalent non–pipeline processing is defined by the ratio

$$\text{Speedup (S)} = \frac{\text{Total time taken by non–pipeline structure to complete n tasks}}{\text{Total time taken by pipeline structure to complete n tasks}}$$

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

→ As number of tasks increases, n becomes much larger than K - 1, and (K + n – 1) approaches to n then

$$S = \frac{t_n}{t_p}$$

→ If we assume that the time to process a task is same in both circuits then $t_n = Kt_p$

$$S = \frac{Kt_p}{t_p} = K$$

Q. Calculate pipeline speedup if time taken to complete a task in conventional machine is 25 ns. In the pipeline machine, one task is divided into 5 segments and each sub operation tasks take 4 ns. Number of tasks to be completed is 100.

Solution: Here,

Time taken to complete a task ($t_n$) = 25 ns
Number of segment (K) = 5
Clock cycle time ($t_p$) = 4 ns
No. of tasks (n) = 100

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

Q. Calculate the speed up rate of 5-segment pipeline with a clock cycle time 25ns to execute 100 tasks.

Solution: Here,
n = 100
Number of segment (K) = 5
Time to complete a task ($t_p$) = 25 ns
$t_n = Kt_p$

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

Q. A non–pipeline system takes 100 ns to process a task. The same task can be processed in a six-segment pipeline with time delay of each segment in the pipeline is as follows; 20 ns, 25 ns, 30 ns. Determine the speed of ratio of pipeline for 100 tasks.

$t_n$ = 100 ns
K = 6
$t_p$ = 30 ns
n = 100

$$S = \frac{nt_n}{(K+n-1)t_n}$$

Q. Suppose that time delays of four segments are $t_1$ = 60ns, $t_2$ = 70 ns, $t_3$ = 100 ns, $t_4$ = 80 ns and interface register have a delay of 10 ns. Determine the speed up ratio.

Solution: Here,

$t_p$ = 100 + 10 = 110 ns
$t_n = t_1 + t_2 + t_3 + t_4$
=60+70+100+80 = 320 ns

$$S = \frac{t_n}{t_p}$$

## 7.3 Arithmetic Pipeline

→ Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations, multiplication of fixed point numbers, and similar computations encountered in scientific problems.

→ We will now show an example of a pipeline unit for floating point addition and subtraction. The inputs to the floating point adder pipeline are two normalized floating point binary numbers.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

→ The floating point addition and subtraction can be performed in four segments, as shown in figure. The registers labeled R are placed between the segments to store intermediate results.

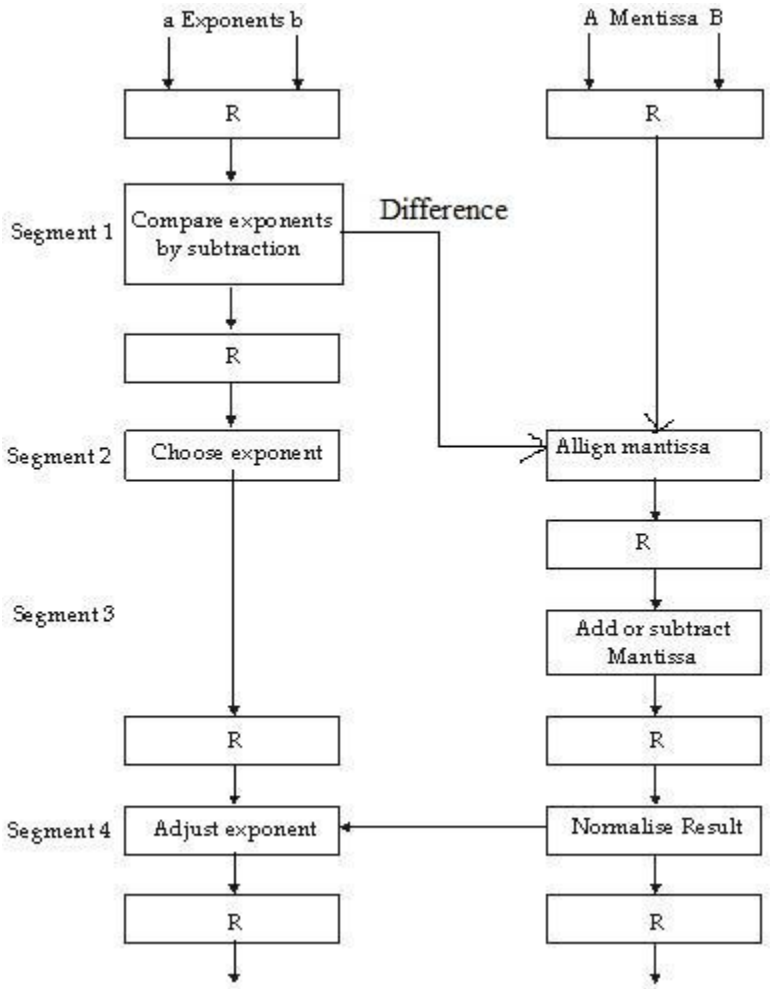



Fig: Pipeline for floating-point addition and subtraction

→ The suboperations that are performed in the four segments are:

i) Compare the exponents.
ii) Align the mantissas.
iii) Add or subtract the mantissas.
iv) Normalize the result.

→ Procedure: The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent

incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

→ For simplicity, we use decimal numbers, the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$
$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain 3-2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9504 \times 10^3$$
$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

## 7.4 Instruction Pipeline

→ Pipeline processing can not only occur in the *data stream* but in the *instruction stream* as well.

→ An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. This technique is called instruction pipelining.

→ Consider sub dividing instruction processing into two ways:

— fetch instruction
— execute instruction

→ During execution, there is time when main memory is not being accessed. During this time, the next instruction could be fetched and buffered (called instruction pre-fetch or fetch overlap).

→ In the most general case, the computer needs to process each instruction with the following sequence of steps:

i. Fetch the instruction from memory.
ii. Decode the instruction.
iii. Calculate the effective address.
iv. Fetch the operands from memory.
v. Execute the instruction.
vi. Store the result in the proper place.

→ To again further speed up, the pipeline must have more stages consider the following decomposition of instruction processing.

a) Fetch instruction (FI)
b) Decode instruction (DI)
c) Fetch operands (FO)
d) Execute instruction (EI)

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Insruction1 | FI | DI | FO | EI | | |
| Instruction2 | | FI | DI | FO | EI | |
| Instruction3 | | | FI | DI | FO | EI |

Fig: Timing diagram for 4-segment instruction pipeline

## 7.5 Pipeline Hazards and their Solution

**Hazards**

1. **resource conflicts**

→ If one common memory is used for both data and instruction, and there is need to read/write data and fetch the instruction at the same time, the resource conflicts occur.

2. **data dependency conflict**

→ Data dependency conflicts arise when an instruction depends on the result of a previous instruction but result of that instruction is not yet available.

3. **branch difficulties**

→ Branch difficulties arise from program control instructions that may change the value of PC.

→ A program is not a straight flow of sequential instructions. There may be branch instructions after the normal flow of program which delay the pipelining executions and affects the programs.

**Solution of pipeline hazards**

1. Resource conflicts can be solved by using separate instruction and data memory.
2. To solve the data dependency conflict, we have following methods:
   - i) **hardware interlock**
     Hardware interlock is a circuit that is responsible to detect the data dependency. After detecting the particular instruction needs data from instruction which is being executed, hardware interface delays the instruction till the data is not available.
   - ii) **operand forwarding**
     Operand forwarding uses special hardware to detect a conflict and then avoid by routing the data through special paths between pipeline segments.
   - iii) **delayed load**
     Delayed load is a procedure that gives the responsibilities for solving data conflicts to the compiler. The compiler is designed to detect conflict & reorder the instructions as necessary to delay the loading of the conflicting data by inserting no operation instructions.
3. Solution to branch difficulties
   - i) **pre fetch target instruction**
     Both the branch target instruction & the instruction following the branch are pre fetched and are saved until the branch instruction is executed. If branching occurs then branch target instruction is continuous.
   - ii) **branch target buffer (BTB)**
     A branch target buffer is an associative memory included in fetch segment of branch instruction that stores the target situation for the previously executed branch. It also stores the next few instructions after the branch target instruction. This way, the branch instructions that have occurred previously are readily available in the pipeline without interruption.
   - iii) **loop buffer**
     The loop buffer is similar to BTB but its speed is high. Program loops are stored in the loop buffer. The program loop then can be executed directly without having access to memory.
   - iv) **branch prediction**
     Special hardware is used to detect the branch in the conditional branch instruction. On the basis of prediction, the instructions are pre fetched.
   - v) **delayed branch**
     Compiler detects the branch instructions, so it re-arranges the instruction to make delay by inserting no operation instruction.

## 7.5 Array and Vector Processing

**Vector Processing**

There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems characterized by the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete. In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.

Computers with vector processing capabilities are in demand in specialized applications. The following are representative application areas where vector processing is of the utmost importance.

Long-range weather forecasting
Petroleum explorations
Seismic data analysis
Medical diagnosis
Aerodynamics and space flight simulations
Artificial intelligence and expert systems
Mapping the human genome
Image processing

To achieve the required level of high performance, it is necessary to utilize the fastest and most reliable hardware and apply innovative procedures from vector and parallel processing techniques.

**Array Processor**

→ An array processor is a processor that performs computations on large arrays of data. The term is used to refer to two different types of processors:

i) **Attached Array Processor**
   — It is the auxiliary processor attached to a general purpose computer which is intended to improve the performance of the host computer in specific numerical computation tasks.

ii) **SIMD Array Processor**
   — It is a processor that has a single instruction multiple data organization. It manipulates vector instruction by means of multiple functional units responding to a common instruction.

# Chapter 8
# Computer Arithmetic

Arithmetic instructions manipulate data to produce solution for computational problems. The four basic arithmetic operations are addition, subtraction, multiplication and division. From these 4, it is possible to formulate other specific problems by means of numerical analysis methods.

There are three ways of representing negative fixed-point binary numbers: signed magnitude, signed 1's complement and signed 2's complement. Signed 2's complemented form is used most but occasionally we deal with signed magnitude representation.

## 8.1 Addition and Subtraction of Signed-Magnitude Data

When two signed numbers A and B are added and subtracted, we find 8 different conditions to consider as described in the following table:

| **Operation** | **ADD Magnitudes** | **SUBTRACT Magnitudes** | | |
|---|---|---|---|---|
| | | **A > B** | **A < B** | **A = B** |
| (+A) + (+B) | + (A + B) | | | |
| (+A) + (-B) | | + (A – B ) | - (B – A ) | + (A – B ) |
| (-A) + (+B) | | - (A – B ) | + (B – A ) | + (A – B ) |
| (-A) + (-B) | - ( A + B) | | | |
| (+A) - (+B) | | + (A – B ) | - (B – A ) | + (A – B ) |
| (+A) - (-B) | + (A + B) | | | |
| (-A) - (+B) | - ( A + B) | | | |
| (-A) - (-B) | | - (A – B ) | + (B – A ) | + (A – B ) |

Table: addition and subtraction of signed-magnitude numbers

Addition (subtraction) algorithm:

- When the signs of A and B are identical (different), add magnitudes and attach the sign of A to the result.
- When the signs of A and B are different (identical), compare the magnitudes and subtract the smaller from larger.

**Hardware implementation**

To implement the two arithmetic operations with hardware, we have to store numbers into two registers A and B. Let $A_s$ and $B_s$ be two flip-flops that hold the corresponding signs. The results are transferred to A and $A_s$. A and $A_s$ together form an accumulator.

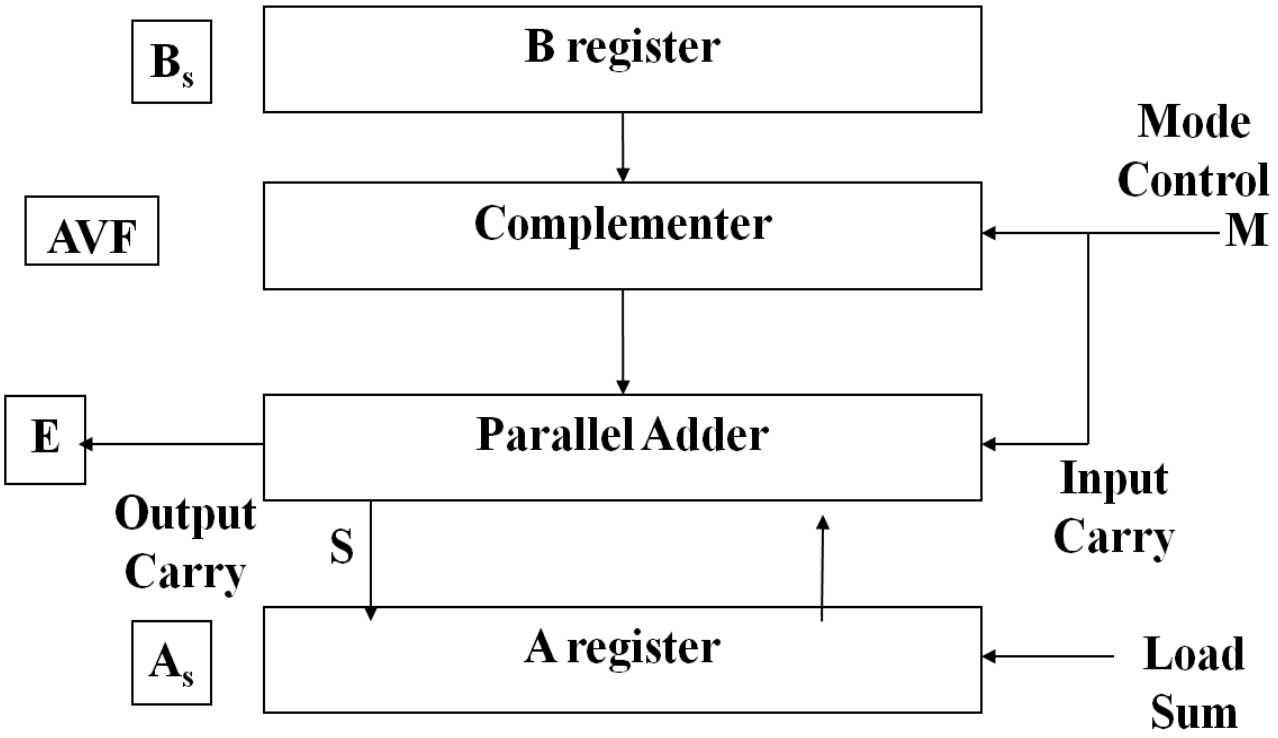



Fig: Hardware for signed-magnitude data addition and subtraction

We need:

- Two registers A and B and sign flip-flops $A_s$ and $B_s$
- A magnitude comparator: to check if A>B, A<B, or A=B
- A parallel adder: to perform A+B
- Two parallel subtractors: for A-B and B-A
- The sign relationships are determined from an X-OR gate with $A_s$ and $B_s$ as inputs.

Block Diagram Description

- Hardware above consists of registers A and B and sign flip-flops $A_s$ and $B_s$.
- The complementor provides an output of B or B' depending on mode input M.
- When M=0, the output of B is transferred to the adder, the input carry is 0 and thus output of adder is A+B. Output carry is transferred to flip-flop E, where it can be checked to determine overflow. When E=1, it is overflow, otherwise, not. The E=1 is transferred to the AVF (add-overflow-flip-flop) which holds the overflow bit when A ns B are added.
- When M=1, 1's complement of B is applied to the adder, input carry is 1 and output is S=A+B'+1 (i.e. A-B). Output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitude of two numbers.

**Hardware Algorithm**

The flowchart for the hardware algorithm is given below:

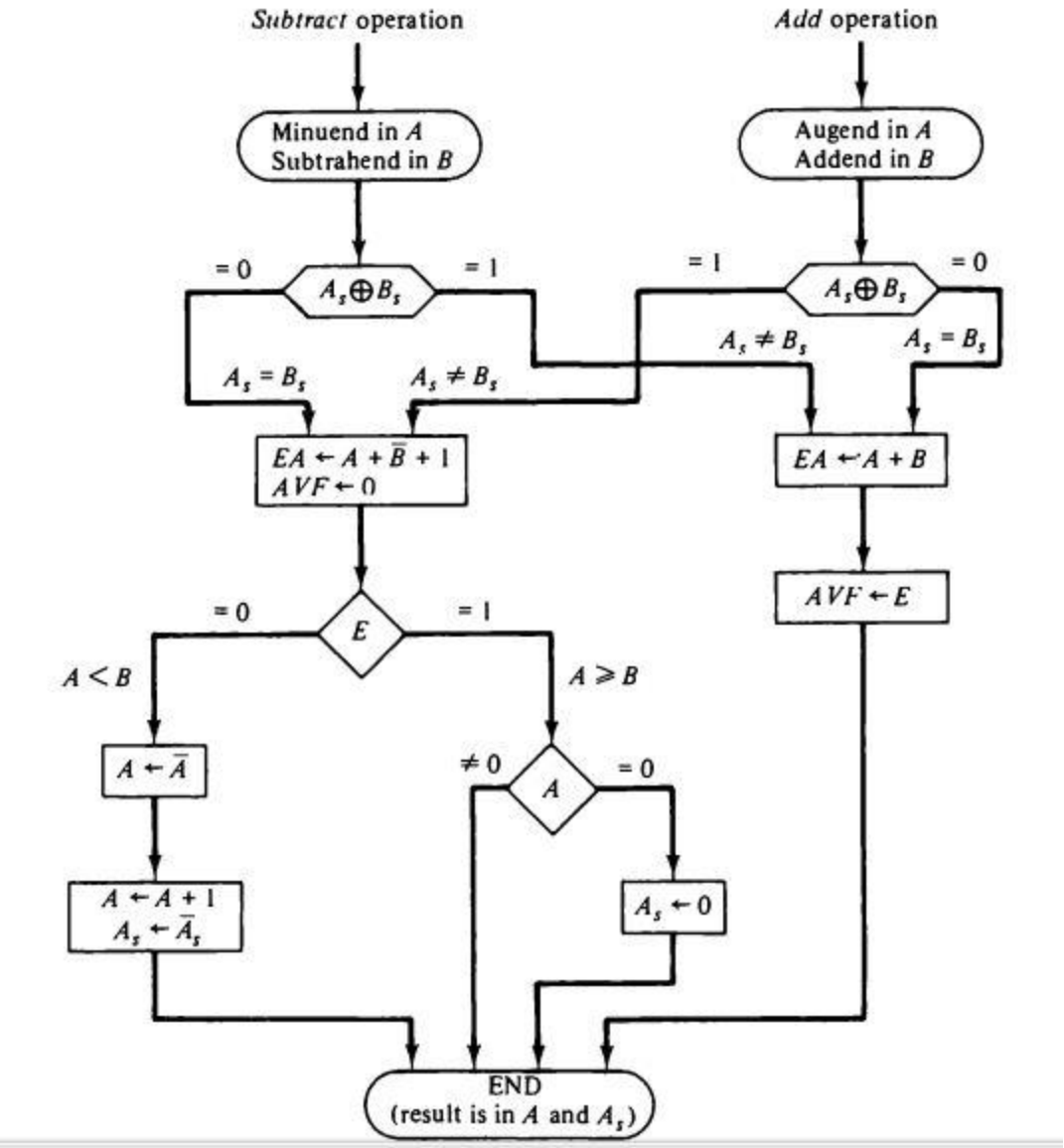



Fig: Flowchart for add and subtract operations with signed-magnitude data

→ As and Bs are compared by an X-OR gate. If output=0, signs are identical, if 1, signs are different.

→ For *add* operation, **identical** signs dictate addition of magnitudes. For *subtraction*, **different** signs dictate magnitudes be added.

- Magnitudes are with a micro operation E A ← A+B (E A is register that combines E and A). If E=1, overflow occurs and is transferred to AVF.

→ Two magnitudes are subtracted if signs are different for add operation and identical for subtract operation.

- Magnitudes are subtracted with a micro operation E A ← A+B'+1.
- No overflow occurs if the numbers are subtracted, so AVF is cleared to 0.
- If E=1, it indicates A≥B and the result in A is correct. If the number in A is 0, the sign $A_s$ must be made positive ($A_s$=0) to avoid negative zero.
- If E=0, it indicates that A<B for which it is necessary to take 2's complement of value in A. the micro operation is A ←A'+1. The sign of $A_s$ must be sign of Bs (i.e. $A_s \leftarrow A_s'$).

## 8.2 Addition and Subtraction of Signed 2's Complement Data

In signed 2's complement representation, the leftmost bit represents sign (0 for positive and 1 for negative). If sign bit is 1, entire number is represented in 2's complement form.

**Addition**: Sign bit is treated as other bits of the number. Carry out of the sign bit is discarded.

**Subtraction**: It consists of first taking 2's complement of subtrahend and then adding it to minuend.
When two numbers of n-digit each are added, the sum occupies n+1 bits. Overflow occurs which is detected by applying last two carries out of the addition to XOR gate. The overflow occurs only if output of the gate is 1.

**Hardware Implementation**

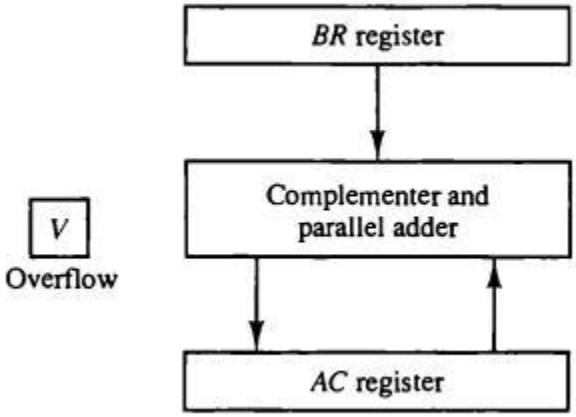



Fig: Hardware for signed-2's complement data addition and subtraction

→ Register configuration is same as signed-magnitude representation except sign bits are not separated. The leftmost bits in AC and BR represent sign bits.
→ Significant Difference: Sign bits are added together with the other bits in complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. Output carry in this case is discarded.

**Hardware Algorithm**

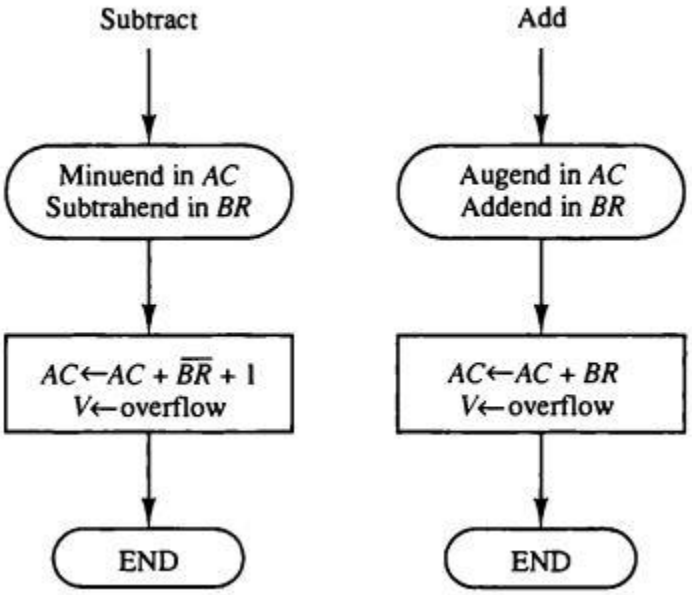



Fig: Algorithm for signed-2's complement data addition and subtraction

Comparing this with its signed-magnitude counterpart, it is much easier to add and subtract numbers. For this reason, most computers adopt this representation over the more familiar signed-magnitude.
Example: 33+(-35)
AC = 33 = 00100001
BR = -35 = 2's complement of 35 = 11011101
AC+BR = 11111110 = -2

## 8.3 Multiplication of Signed-Magnitude Data

For this representation, multiplication is done by a process of successive shift and adds operations. As an example:

```
 23      10111    Multiplicand
 19    × 10011    Multiplier
         10111
        10111
       00000   +
      00000
     10111
437  110110101    Product
```

Process consists of looking successive bits of the multiplier, least significant bits first.

- If the multiplier bit is 1, the multiplicand bit is copied down; otherwise zeroes are copied down.
- Numbers copies down in successive lines are shifted one position left. Finally, numbers are added to form a product.

The sign of the product is determined from the signs of the multiplicand and multiplier.

- If they are alike, the sign of the product is positive.
- If they are unlike, the sign of the product is negative.

**Hardware Implementation**

It needs same hardware as that of addition and subtraction of signed-magnitude. In addition, it needs two more registers Q and SC.

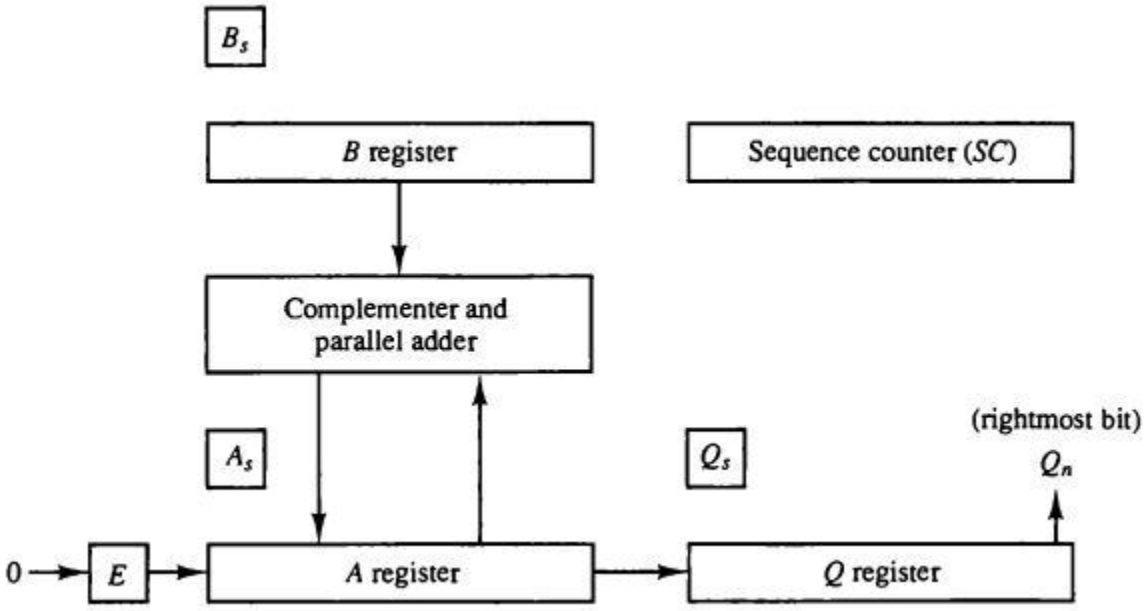
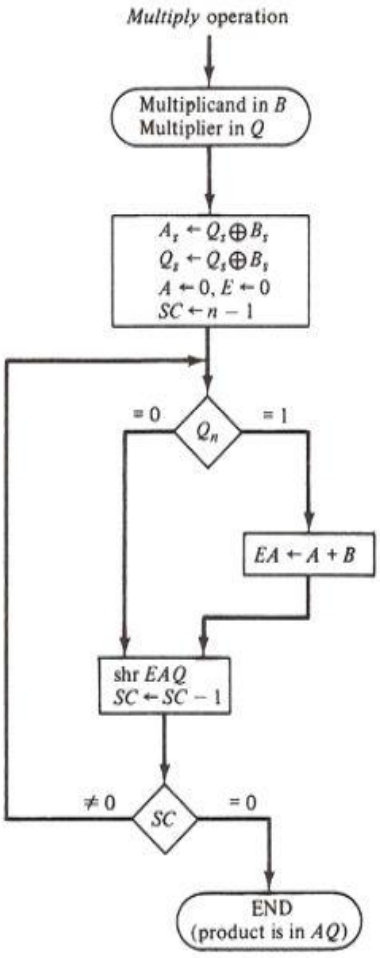



Fig: Hardware for signed-magnitude multiply operation

→ B←multiplicand, $B_s$←sign
→ Q←multiplier, $Q_s$←sign
→ Successively accumulate partial products and shift it right
→ SC←no. of bits in multiplier (magnitude only)
→ SC is decremented after forming each partial product. When SC is 0, process halts and final product is formed.
→ Sum of A and B forms a partial product.

**Hardware Algorithm**




Example: B = 10111 (**Multiplicand**)
Q = 10011 (**Multiplier**)

| Operation | E | A | Q | SC |
|---|---|---|---|---|
| Initial conf. | 0 | 00000 | 10011 | 101 |
| Iteration 1 ($Q_n$ = 1)<br>EA← A+B<br>PP1--><br>shr EAQ, SC←SC-1 | <br><br>0<br>0 | 00000<br>+ 10111<br>10111<br>01011 | <br><br><br>11001 | <br><br><br>100 |
| Iteration 2 ($Q_n$ = 1)<br>EA← A+B<br>PP2--><br>shr EAQ, SC←SC-1 | <br><br>1<br>0 | 01011<br>+ 10111<br>00010<br>10001 | <br><br>11001<br>01100 | <br><br><br>011 |
| Iteration 3 ($Q_n$ = 0)<br>shr EAQ, SC←SC-1 | <br>0 | <br>01000 | <br>10110 | <br>010 |
| Iteration 4 ($Q_n$ = 0)<br>shr EAQ, SC←SC-1 | <br>0 | <br>00100 | <br>01011 | <br>001 |
| Iteration 5 ($Q_n$ = 1)<br>EA← A+B<br>PP3--><br>shr EAQ, SC←SC-1 | <br><br>0<br>0 | 00100<br>+ 10111<br>11011<br>01101 | <br><br>01011<br>10101 | <br><br><br>000 |
| Final Product in AQ | 0110110101 | | | |

Fig: Algorithm for signed-magnitude multiply operation

## 8.4 Multiplication of Signed 2's Complement Data (Booth Multiplication Algorithm)

Booth algorithm is used to multiply binary numbers in signed-2's complement form.

Rules:

i) The partial product doesn't change when the multiplier bit is identical to the previous multiplier bit. ( $Q_nQ_{n+1}$ = 00 or 11)

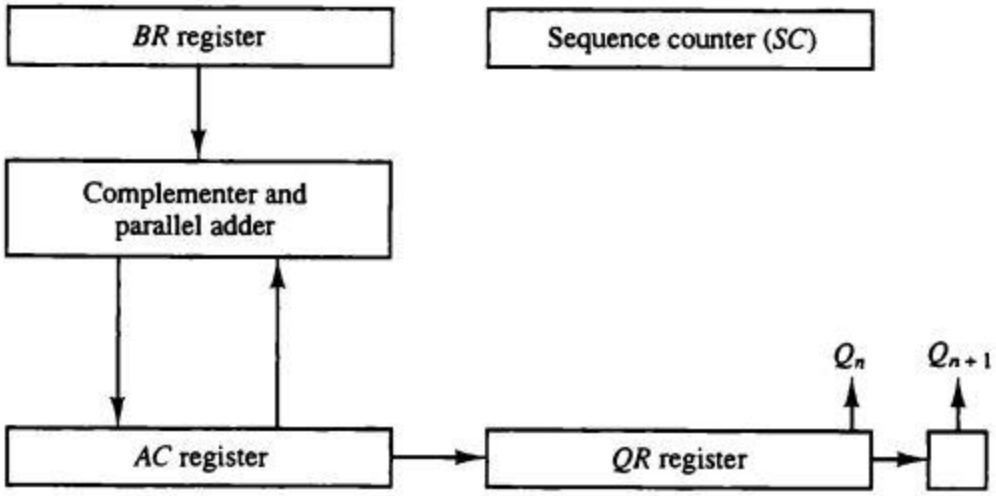
ii) The multiplicand is added to the partial product if LSB is 0 in the string of 0's in the multiplier. ( $Q_nQ_{n+1}$ = 01)

iii) The multiplicand is subtracted from the partial product if LSB is 1 in the string of 1's in the multiplier. ( $Q_nQ_{n+1}$ = 10)

iv) After each addition/subtraction, the partial product is shifted right using arithmetic shift.

This algorithm can be used for both the positive and negative multiplier in 2's complement form.

**Hardware for Booth Algorithm**




- Here, sign bits are not separated.
- Registers A, B and Q are renamed to AC, BR and QR respectively.
- Extra flip-flop $Q_{n+1}$ appended to QR is needed to store almost lost right shifted bit to the multiplier (which along with current $Q_n$ gives information about bit sequencing of multiplier).
- Pair $Q_nQ_{n+1}$ inspect double bits of the multiplier.

**Hardware Booth Algorithm**

E.g. (-9)*(-13) = +117
10111*10011

Numerical Example: **Booth algorithm**

BR = 10111 (**Multiplicand**)
QR = 10011 (**Multiplier**)

| $Q_nQ_{n+1}$ | $BR$ = 10111<br>$\overline{BR}$ + 1 = 01001 | $AC$ | $QR$ | $Q_{n+1}$ | $SC$ |
|---|---|---|---|---|---|
| | Initial | 00000 | 10011 | 0 | 101 |
| 1 0 | Subtract $BR$ | 01001 | | | |
| | | 01001 | | | |
| | ashr | 00100 | 11001 | 1 | 100 |
| 1 1 | ashr | 00010 | 01100 | 1 | 011 |
| 0 1 | Add $BR$ | 10111 | | | |
| | | 11001 | | | |
| | ashr | 11100 | 10110 | 0 | 010 |
| 0 0 | ashr | 11110 | 01011 | 0 | 001 |
| 1 0 | Subtract $BR$ | 01001 | | | |
| | | 00111 | | | |
| | ashr | 00011 | 10101 | 1 | 000 |

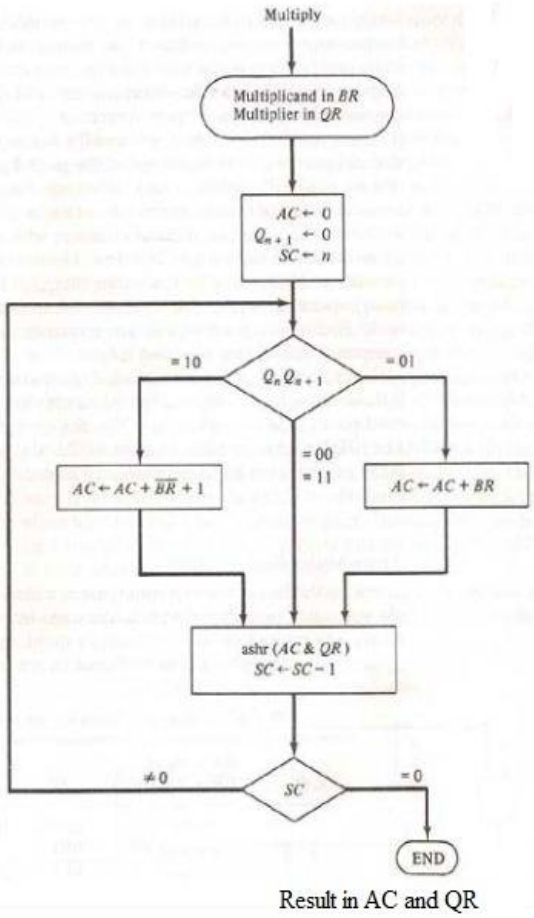



Fig: Flowchart for Booth Algorithm for signed-2's complement data multiplication

# Chapter 9
# Input and Output Organization

## 9.1 Peripheral Devices

→ Those devices which are connected to computer are called peripheral devices. E.g. keyboard, mouse, printer, magnetic tape, magnetic disk etc.

→ Input output operations are accomplished through peripheral device that provides a means of exchanging data between the external environment and the computer.

## 9.2 Input Output Interface

→ The computer system includes special hardware components between the CPU and peripheral devices to supervise and synchronize all input and output transfer. These components are called interface unit (I/O module).

→ There are several problems while we are trying to connect the peripheral devices directly with the CPU. They are:

- There is wide variety of peripherals with various methods of operation. It would be impractical to incorporate the necessary logic within the processor to control a range of devices.
- The data transfer rate of peripherals is often much slower than that of the memory and processor. Thus, it is impractical to use the high speed system bus to communicate directly with a peripheral.
- Peripheral devices often use different data formats and word lengths than the computer to which they are attached.
- Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

→ Because of these problems (differences), we cannot connect peripheral directly with CPU. And to resolve these problems, we need I/O interface.

→ The main functions of input-output interface circuit are data conversion, synchronization and device selection. Data conversion refers to conversion between digital and analog signals, and conversion between serial and parallel data formats. Synchronization refers to matching of operating speeds of CPU and other peripherals. Device selection refers to the selection of I/O device by CPU in a queue manner

**I/O bus and Interface module**

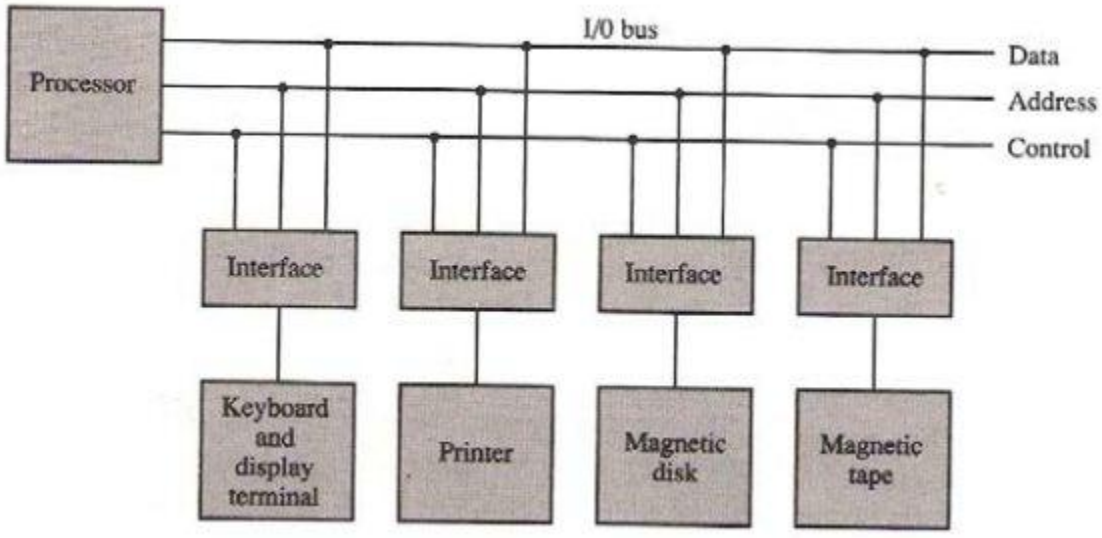



Fig: Connection of I/O bus to I/O devices

→ Above figure shows the I/O bus and I/O device connected with interface unit.

→ Each interface attached to I/O bus contains address decoder that monitors the address line.

→ To communicate with a particular device, the processor places a device address on the address lines. When the interface detects its own address, then it activates the path between the bus lines and the device that it controls. All the peripherals whose addresses don't correspond to the address in the address bus are disabled by their interface.

→ At the same time, the address is made available in the address line; the processor provides a function code in the control line which is also called I/O command. The interface selected responds to the function code and proceeds to execute it.

→ There are four types of I/O command:

*i)* *Control command*
The control command is issued to activate and inform the peripheral devices what they have to do.

*ii)* *Status command*
The processor issues status command to test the status condition of interface and peripherals.

*iii)* *Output data command*
It is issued to transfer data from system bus to one of storage register in I/O module.

*iv)* *Input data command*
It is issued to transfer data from peripheral to one of its register in I/O module.

**I/O and Memory bus**

→ The CPU communicates with the memory and I/O. Both I/O and memory have data, address and control buses. There are three ways that computer buses can be used to communicate with memory and I/O. they are:

i) Use two separate buses, one for memory and the other for I/O.
ii) Use one common bus for both memory and I/O but have separate control lines for each.
iii) Use one common bus for both memory and I/O with common control lines.

**Isolated I/O**

→ The I/O in which one common bus is used for memory and I/O but there are separate read and write controls for I/O and memory transfer is called isolated I/O.

→ This configuration isolates all I/O interface address from the address assigned to memory. Therefore, this method is called isolated I/O.

→ The I/O read and I/O write control lines are enabled during an I/O transfer. The memory read and memory write control lines are enabled during a memory transfer.

→ When the CPU doing I/O read and write operation, the address associated with information (instruction or data) is placed in common address lines. At the same time, the I/O read and I/O write line is enabled. This informs the interface that the address in the address bus is for I/O, not for memory.

→ When the CPU doing memory read and write operation, the address associated with information (instruction or data) is placed in common address lines. At the same time, the memory read and memory write line is enabled. This informs the interface that the address in the address bus is for memory, not for I/O.

Advantage

- Same address can be used for either memory and I/O transfer, only control line identifies whether the transfer is I/O or memory.

Disadvantage

- Needed separate input-output read/write and memory read/write instructions for I/O and memory transfer.

**Memory mapped I/O**

→ The I/O in which one common bus is used for memory and I/O bus with common control lines is called memory mapped I/O.

→ It uses common read/write instruction for memory and I/O operation.

→ It uses same address space for both memory and I/O and treats interface register as a part of memory.

→ The addresses used by interface register cannot be used for memory space. So, if CPU places the register addresses and data on common bus, the memory system ignores the operation. So, I/O operation is performed.

Advantage

- Same instructions are used for memory and I/O.
- No need of separate control lines for I/O and memory operation.

Disadvantage

- No full memory address can be used.

## 9.3 Asynchronous Data Transfer

→ Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted.

→ One way of achieving this is by means of a ***strobe*** pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.

→ Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as ***handshaking***.

→ Strobe technique and Handshaking technique are combined and used extensively on numerous occasions requiring the transfer of data between two independent units.

**Strobe Control**

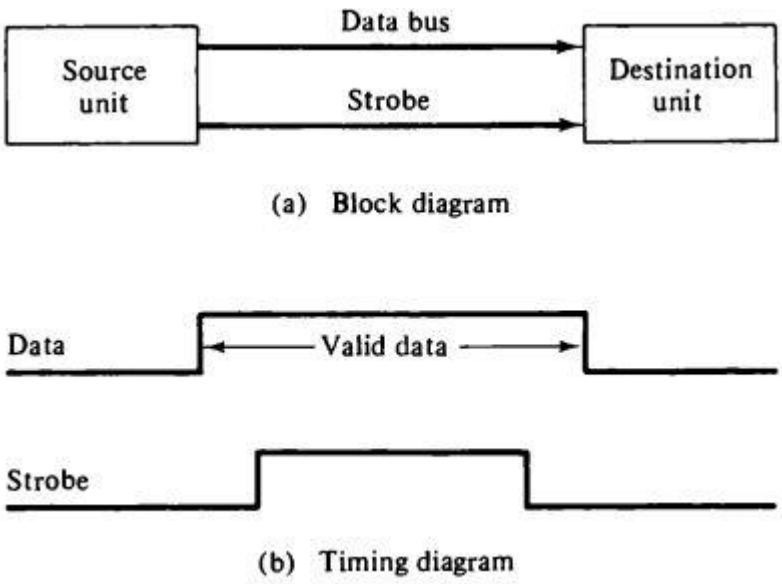



Fig: Source-initiated strobe for data transfer

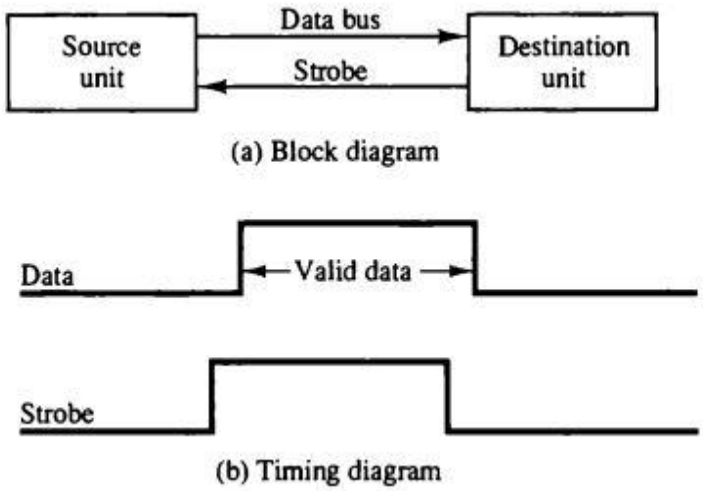
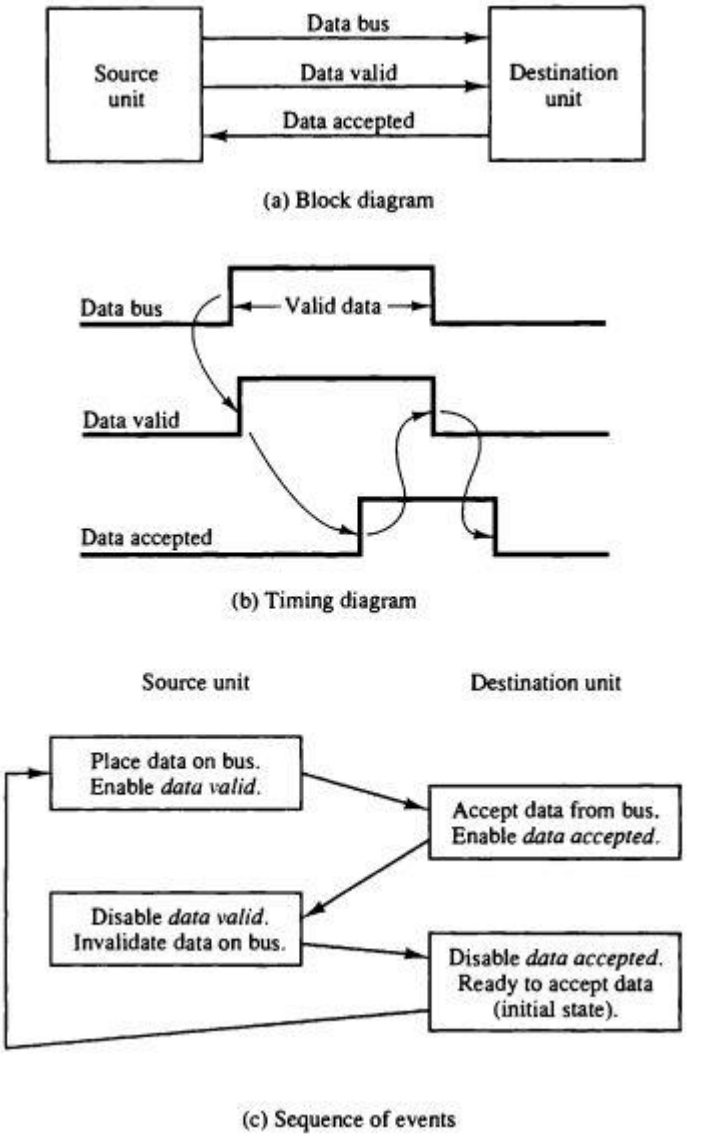
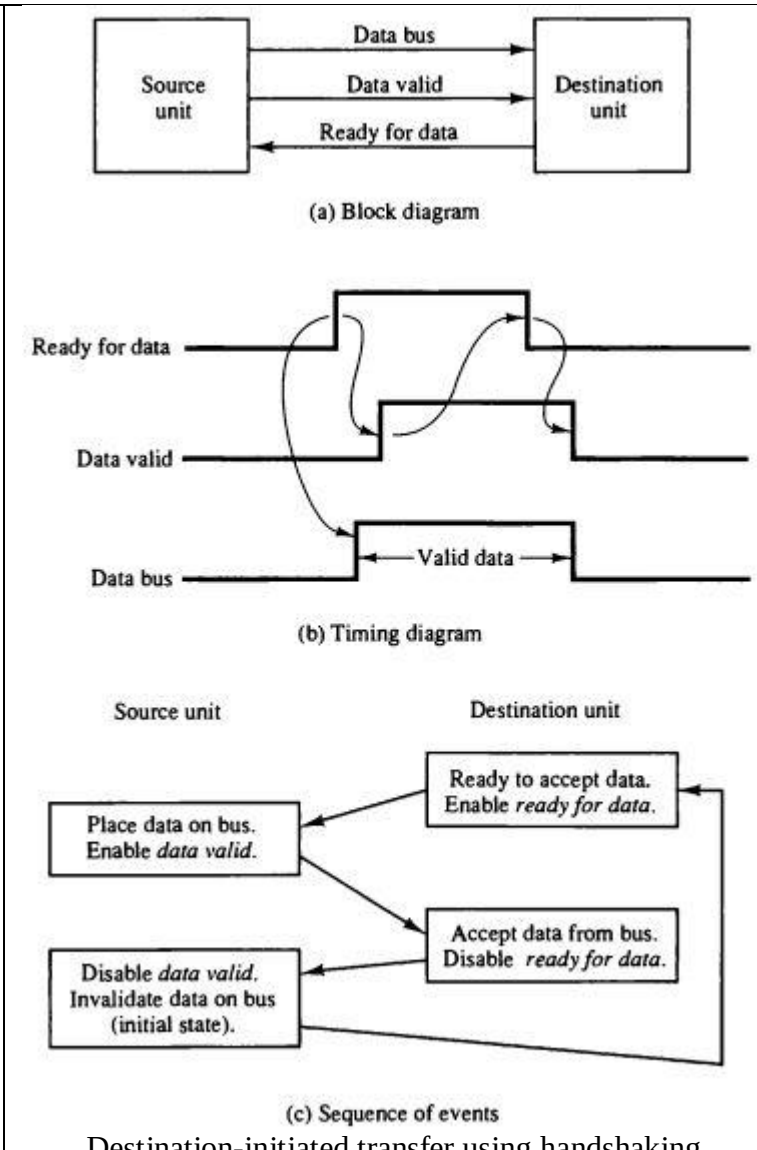



Fig: Destination-initiated strobe for data transfer

**Handshaking**




Source-initiated transfer using handshaking




Destination-initiated transfer using handshaking

H\W, Describe Asynchronous Serial Transfer.

## 9.4 Modes of transfer (techniques for I/O operations)

The possible data transfer between to and from peripherals are:

**i. Programmed I/O**

→ In programmed I/O, the processor executes a program that gives it direct control of the I/O operation, including sensing I/O device status, sending a read or write command, and transferring the data.

→ The execution of I/O related instructions are performed by issuing a command to the appropriate I/O module.

→ I/O module performs the requested action and set the appropriate bits in the I/O status register.

→ The processor periodically checks the status of the I/O module until it finds that the operation is complete.

→ In programmed I/O, CPU stays in programming loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process since it keeps the processor busy needlessly.

**ii. Interrupt-initiated I/O**

→ To reduce the time spent on I/O operations for periodically checking the status of I/O device, the CPU can use an interrupt-driven I/O approach.

→ In this method, CPU uses an interrupt and commands to inform the interface to issue an interrupt signal when the data are available from the device and CPU does other work.

→ When I/O module determines that the device is ready for data transfer, it interrupts the CPU. When CPU detects the external interrupt signal, it immediately stops the task it is processing, and jumps to a service routine to process the I/O transfer and then returns to the task it was originally performing.

**Example of Interrupt initiated I/O:**

1. Vectored interrupt
2. Non-vectored interrupt

**Vectored interrupt:**

- In vectored interrupt, the source, that interrupts, supplies the branch information to the computer. This information is called the interrupt vector.

**Non-vectored interrupt:**

- In a non vectored interrupt, the branch address is assigned to a fixed location in memory.

### iii. DMA transfer

→ Drawbacks of programmed and interrupt-driven I/O:

- The I/O transfer rate is limited by the speed with which the processor can test and service the device.
- The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

→ To overcome these drawbacks, the DMA is used to transfer the data between memory and I/O device with less involvement of CPU.

→ Large amount of data can be transferred between memory and peripheral, severely impacting CPU performance.

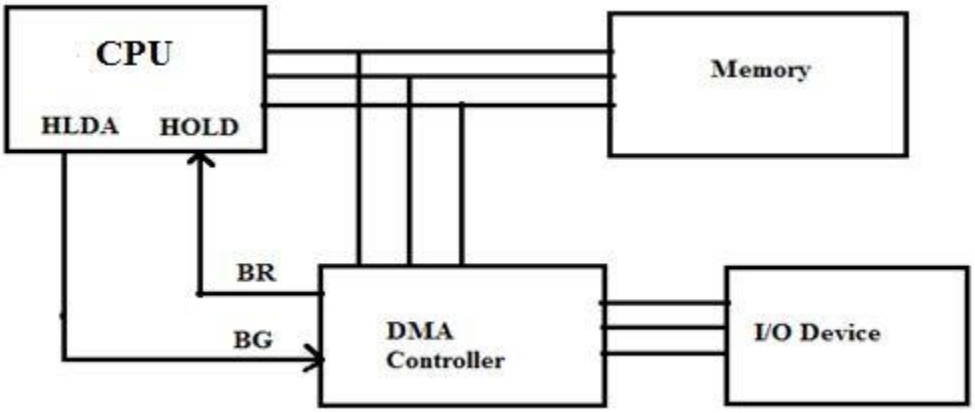



Fig: DMA Transfer

→ In this transfer mode, DMA controller sends Bus Request (BR) to CPU enabling BR line.

→ After receiving BR request, CPU acknowledges the DMA controller by sending Read/Write control signal, Device address, Starting address of memory block for data and amount of data to be transferred and disables its control over system bus by issuing Bus Grant (BG) signal to DMA controller. And CPU does other works.

→ DMA controller then start transfer of data between memory and I/O. After completing all transfer of data, the controller sends interrupt to CPU informing that it has completed the job assigned to it.

→ The CPU finalizes the DMA operation and resumes it operation.

## 9.5 Interrupt

When a Process is executed by the CPU and when a user Request for another Process, then this will create disturbance for the Running Process. This is also called as the **Interrupt**.

Interrupts can be generated by User, Some Error Conditions and also by Software's and the hardware's. But CPU will handle all the Interrupts very carefully because when Interrupts are generated, then the CPU must

handle all the Interrupts very carefully means the CPU will also provide Response to the various Interrupts those are generated so that when an interrupt has occurred, then the CPU will handle by using the Fetch, Decode and Execute Operations.

Types of Interrupts

Generally there are three types o Interrupts those are occurred. For Example

a) Internal Interrupt
b) Software Interrupt.
c) External Interrupt.

The Internal Interrupts are those which are occurred due to some problem in the execution. For example, when a user performing any operation which contains any type of error so that internal interrupts are those which are occurred by some operations or by some instructions and the operations those are not possible but a user is trying for that operation.

The software interrupts are those which are made some call to the system. For example, while we are processing some instructions and when we want to execute one more application programs.

The External Interrupt occurs when any input and output devices request for any operation and the CPU will execute those instructions first. For example, when a program is executed and when we move the mouse on the screen, then the CPU will handle this external interrupt first and after that he will resume with his operation.

**Priority Interrupt**

A priority interrupt is a system that determines which condition is to be serviced first when two or more requests arrive simultaneously. Highest priority interrupts are serviced first. Devices with high speed transfers are given high priority and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time the computer services the device, with the higher priority first. Establishing the priority of simultaneous interrupts can be done by software or hardware.

The hardware priority function can be established by either a serial or a parallel connection of interrupt lines.

**Daisy-Chaining Priority**

The serial connection is called daisy chaining method. In daisy chaining method, all the devices are connected in serial. The device with the highest priority is placed in the first position, followed by lower priority devices.

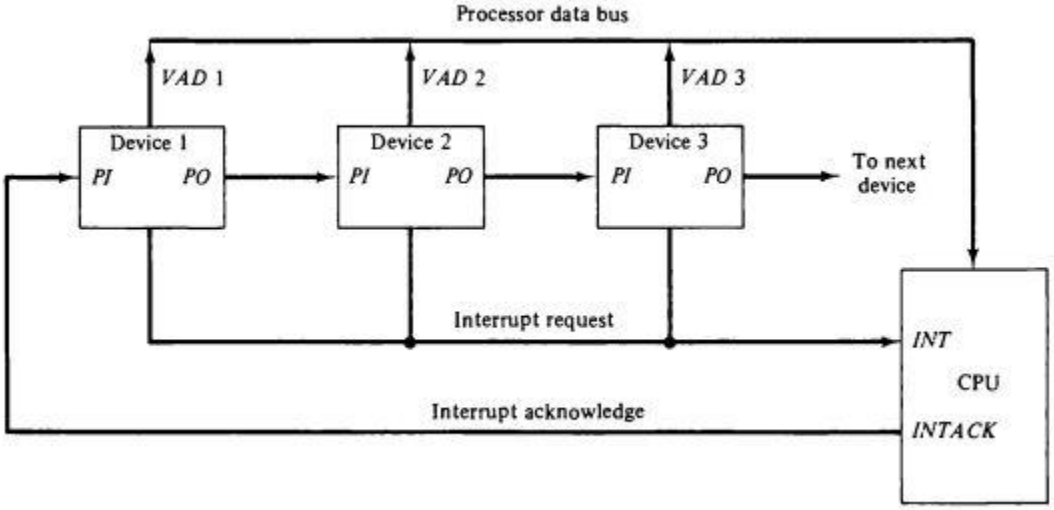



Fig: Daisy-Chain Priority Interrupt

**Parallel Priority Interrupt**

The parallel priority interrupt hardware is shown in figure. It has an *interrupt register* whose bits are connected to the interrupt request lines of different devices in the system. It also has a *mask register* whose bits can be

used to control the status of each interrupt request. The mask register has the same number of bits as the interrupt register. Each interrupt bit and its corresponding mask bit are applied to an AND gate.

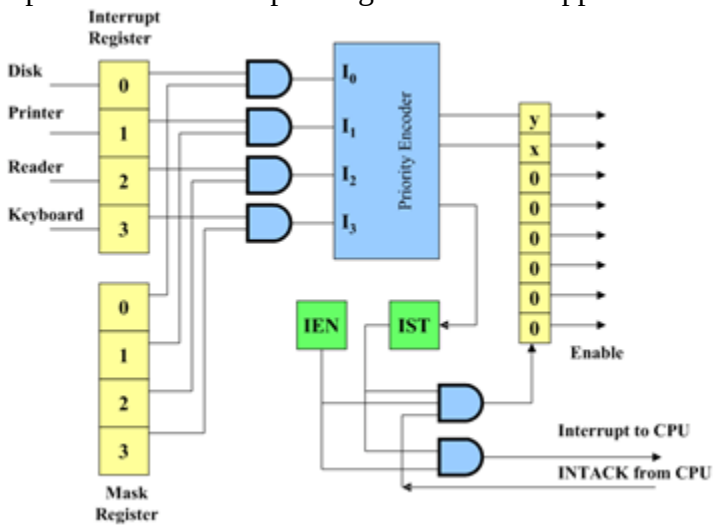



Fig: Parallel Priority Interrupt Hardware

This produces the four inputs to a priority encoder. The priority encoder generates two bits of the vector address. This is transferred to the CPU. Another output from the encoder sets an interrupt status flip flop IST. The outputs of interrupt enable flip-flop IEN and IST are applied to an AND gate. The outputs of this AND gate provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

**Priority Encoder**

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given below.

| Inputs | | | | Outputs | | | |
|---|---|---|---|---|---|---|---|
| $I_0$ | $I_1$ | $I_2$ | $I_3$ | $x$ | $y$ | $IST$ | Boolean functions |
| 1 | × | × | × | 0 | 0 | 1 | |
| 0 | 1 | × | × | 0 | 1 | 1 | $x = I_0'I_1'$ |
| 0 | 0 | 1 | × | 1 | 0 | 1 | $y = I_0'I_1 + I_0'I_2'$ |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | $(IST) = I_0 + I_1 + I_2 + I_3$ |
| 0 | 0 | 0 | 0 | × | × | 0 | |

Fig: Priority Encoder Truth Table

The X's in the table designate don't care conditions. Input $I_0$ has the highest priority. When $I_0$ input is 1, the output generates an output xy=00. $I_1$ has the next priority level. The output is 01 if $I_1$=1 and $I_0$=0. The output for $I_2$ is generated only if higher priority inputs are 0 and so on. The interrupt status IST is set only when one or more inputs are equal to 1. If all inputs are 0, IST is cleared to 0 and the other outputs of the encoder are not used, so they are marked with don't care conditions.

## 9.6 Direct Memory Access (DMA)

DMA is a sophisticated I/O technique in which a DMA controller replaces the CPU and takes care of the access of both, the I/O device and memory, for fast data transfers. Using DMA you get the fastest data transfer rates possible.

Momentum behind the DMA: interrupt-driven and Programmed I/O require active CPU intervention (All data must pass through CPU).transfer rate is limited by processor's ability to service the device and hence CPU is

tied up managing I/O transfer. Removing CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer.

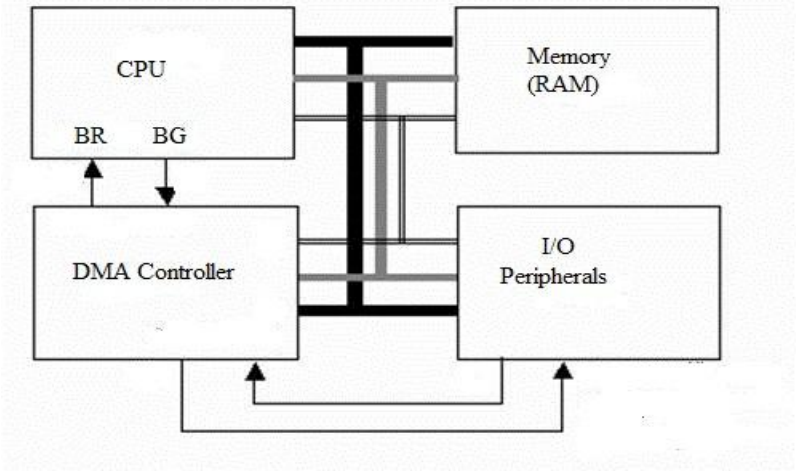



Fig: DMA

Extensively used method to capture buses is through special control signals:

- Bus Request (BR): used by DMA controller to request the CPU for buses. When this input is active, CPU terminates the execution of the current instruction and places the address bus; data bus and read & write lines into high impedance state.
- Bus Grant (BG): CPU activates BG output to inform DMA that buses are available (in high impedance state). DMA now take control over buses to conduct memory transfers without processor intervention. When DMA terminates the transfer, it disables the BR line and CPU disables BG and returns to normal operation.

When DMA takes control of bus system, the transfer with memory can be made in the following ways:

a) Burst transfer mode
   - Entire block of data is transferred to one contiguous sequence.
   - Useful for loading programs or data file into memory, but it makes CPU inactive for long periods of time.

b) Cycle stealing mode
   - DMA controller transfers one byte of data then returns control of system buses to CPU.
   - CPU performs an instruction, and then DMA controller transfers a data value by stealing one machine cycle of CPU.

## 9.7 Input-Output Processor (IOP)

Instead of having each interface communicate with the CPU, computer may have one or more external processors for the task of communicating directly with all I/O devices. Such processor is called Input-Output processor. I/O processors also have direct memory access capability.

→ IOP is a special processor dedicated to communicate directly with all I/O devices.

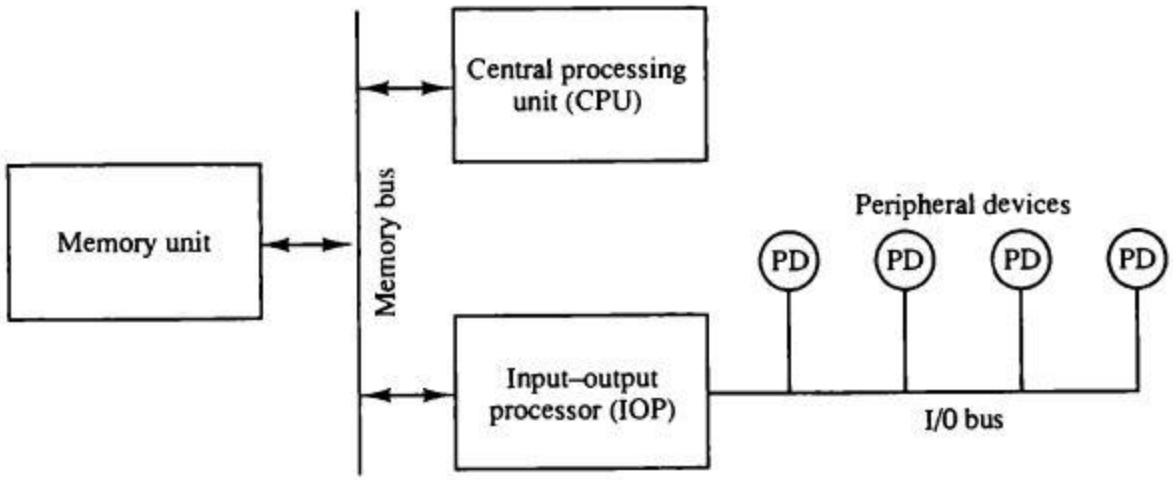



Fig: Block Diagram of computer with I/O Processor

→ Each IOP takes care of I/O task keeping CPU free from involvement of I/O transfer.
→ IOP has ability to execute I/O instruction which gives it complete control over I/O operation. It has its own instruction set with I/O instructions and a local memory in its own right.
→ IOP accesses memory by cycle stealing.
→ CPU directs IOP to execute an I/O programs in memory. The IOP fetches and executes these instructions without CPU intervention. IOP interrupts CPU when entire operation has been performed.
→ The major difference between DMA and IOP is that IOP can fetch and execute I/O instruction from memory but DMA cannot fetch and execute the I/O instruction.

# Chapter 10
# Memory Organization

## 10.1 Memory Hierarchy

→ Memory unit is an essential component in any digital computer since it is needed for storing programs and data.

→ The memory unit that communicates directly with the CPU is called the ***main memory***. Devices that provide backup storage are called ***auxiliary memory***.

→ Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

→ The memory hierarchy consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster ***cache memory*** accessible to the high-speed processing logic.

→ The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor,

→ Access time of CPU and main memory are different. So, to co-ordinate the speed between these, a fast memory is needed called as cache memory.

→ While the I/O processor manages data transfer between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU.

→ As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer.

→ The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory.

→ The cache memory is very small, relatively very small, relatively expensive, and has very high access speed.

→ CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data.

→ Many OS are designed to enable the CPU to process a number of independent programs concurrently. This concept, called ***multiprogramming***, refers to the existence of two or more programs in different parts of memory hierarchy at the same time. In this way, it is possible to keep all parts of the computer busy by working with several programs in sequence.

→ The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called ***memory management system***.

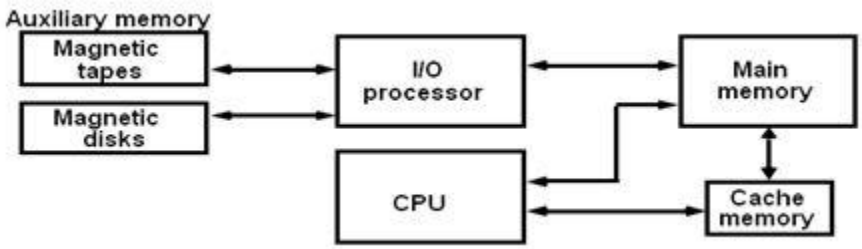
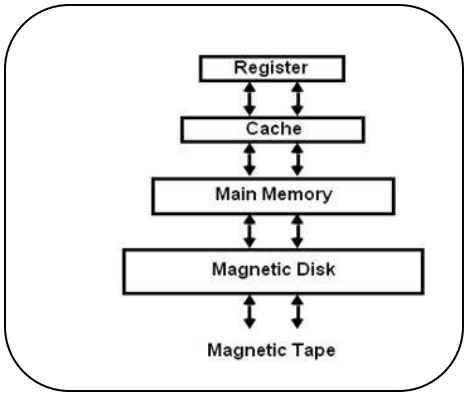



Fig: Memory hierarchy in a computer system




## 10.2 Main Memory

→ The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation.

→ Integrated circuit RAM chips are available in two possible modes: ***static*** and ***dynamic***.

→ The static RAM consists essentially of internal flip-flops. The stored information remains valid as long as power is applied to the unit.

→ The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the

capacitor tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory.

→ The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip.

→ Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips.

→ RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in the value once the production of the computer is completed.

→ Among other things, the ROM portion of main memory is needed for storing an initial program called as ***bootstrap loader***. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on.

→ Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again.

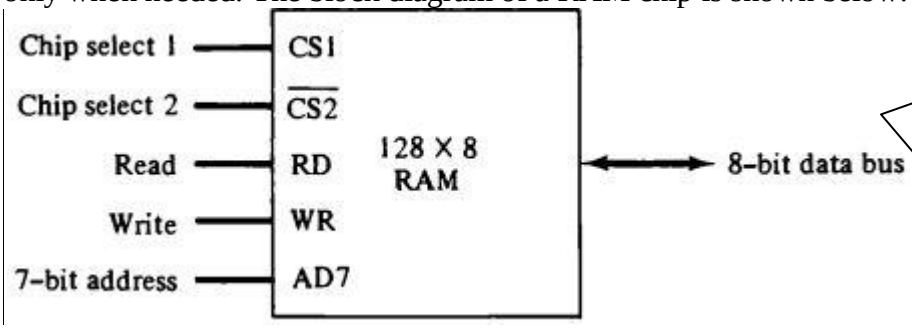
→ When power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader which loads a portion of the OS from disk to main memory and control is then transferred to the OS, which prepares the computer for general use.

**RAM and ROM chips**

RAM and ROM chips are available in a variety of sizes. If we need larger memory for the system, it is necessary to combine a number of chips to form the required memory size.

**RAM Chips**

A RAM chip is better suited to communicate with CPU if it has one or more control inputs that select the chip only when needed. The block diagram of a RAM chip is shown below:




→ Requires 7-bit address and an 8-bit bidirectional data bus

→ Chip select (CS) control inputs are for enabling the chip only when it is selected by CPU.

Fig: Typical RAM chip (128 words of eight bits each)

| CS1 | $\overline{CS2}$ | RD | WR | Memory function | State of data bus |
|---|---|---|---|---|---|
| 0 | 0 | × | × | Inhibit | High-impedance |
| 0 | 1 | × | × | Inhibit | High-impedance |
| 1 | 0 | 0 | 0 | Inhibit | High-impedance |
| 1 | 0 | 0 | 1 | Write | Input data to RAM |
| 1 | 0 | 1 | × | Read | Output data from RAM |
| 1 | 1 | × | × | Inhibit | High-impedance |

→ The unit is in operation only when CS1=1 and (CS2)'=0.

→ High impedance state indicates open circuit i.e. output does not carry a signal and has no logic significance.

Fig: Function table for RAM chip

**ROM Chips**

Since a ROM chip can only read, data bus is unidirectional (output mode only).

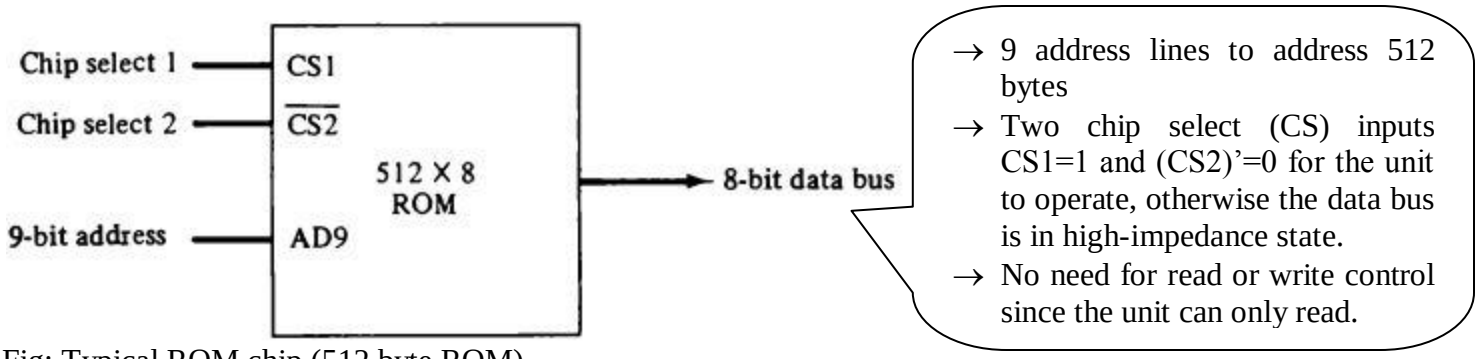



Fig: Typical ROM chip (512 byte ROM)

**Memory Address Map**

The addressing of memory can be established by means of a table that specifies the memory address assigned to each RAM or ROM chip. This table is called memory address map and is a pictorial representation of assigned address space for particular chip.

Example: Suppose computer system needs 512 bytes of RAM and 512 bytes of ROM.

| Component | Hexadecimal address | Address bus 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| RAM 1 | 0000–007F | 0 | 0 | 0 | x | x | x | x | x | x | x |
| RAM 2 | 0080–00FF | 0 | 0 | 1 | x | x | x | x | x | x | x |
| RAM 3 | 0100–017F | 0 | 1 | 0 | x | x | x | x | x | x | x |
| RAM 4 | 0180–01FF | 0 | 1 | 1 | x | x | x | x | x | x | x |
| ROM | 0200–03FF | 1 | x | x | x | x | x | x | x | x | x |

→ Component column specifies RAM or ROM chip. We use four 128 words RAM to make 512 byte size.
→ Hexadecimal address column assigns a range of addresses for each chip.
→ 10 lines in address bus column: lines 1 through 7 for RAM and 1 through 9 for ROM. Distinction between RAM and ROM chip is made by line 10. When line 10 is 1, it selects ROM and when it is 0, CPU selects RAM.
→ X's represents a binary number ranging from all-0's to all-1's.

## 10.3 Associative Memory

→ Also called as Content-addressable memory (CAM), associative storage, or associative array
→ Content-addressed or associative memory refers to a memory organization in which the memory is accessed by its content (as opposed to an explicit address).
→ It is a special type of computer memory used in certain very high speed searching applications.
→ In standard computer memory (random access memory or RAM), the user supplies a memory address and the RAM returns the data word stored at that address.
→ In CAM, the user supplies a data word and then CAM searches its entire memory to see if that data word is stored anywhere in it. If the data word is found, the CAM returns a list of one or more storage addresses where the word was found.
→ CAM is designed to search its entire memory in a single operation.
→ It is much faster than RAM in virtually all search applications.
→ An associative memory is more expensive than RAM, as each cell must have storage capability as well as logic circuits for matching its content with an external argument.
→ Associative memories are used in applications where the search time is very critical and short.

→ Associative memories are expensive compared to RAMs because of the add logic associated with each cell.

**Hardware Organization**

→ Associative memory consists of a memory array and logic for m words and n bits per word.

→ The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory words. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.

→ Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

→ The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus, the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

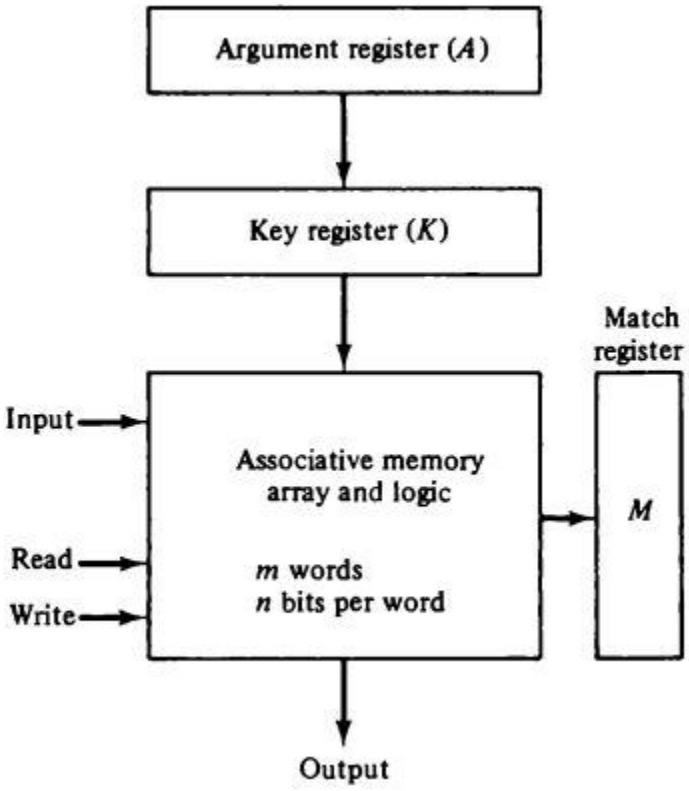


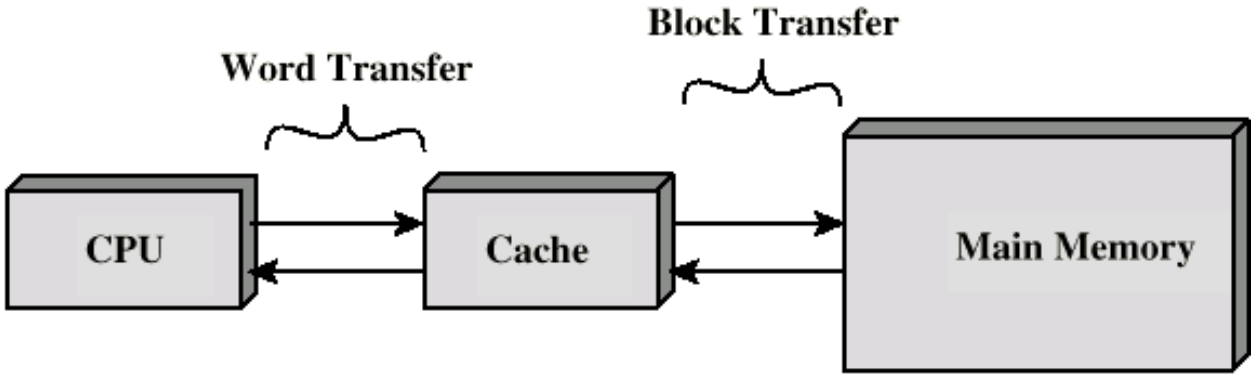

Fig: Block Diagram of Associative Memory

E.g. A=10111100
K=111000000

| | | |
|---|---|---|
| Word1 | 100111100 | no match |
| Word2 | 101000001 | match |

## 10.4 Cache Memory: cache mapping techniques

Cache (pronounced cash) memory is extremely fast memory that is built into a computer's central processing unit (CPU), or located next to it on a separate chip. The CPU uses cache memory to store instructions that are repeatedly required to run programs, improving overall system speed. The advantage of cache memory is that the CPU does not have to use the motherboard's system bus for data transfer. Whenever data must be passed through the system bus, the data transfer speed slows to the motherboard's capability. The CPU can process data much faster by avoiding the bottleneck created by the system bus.




→ A cache is a small amount of very fast associative memory.
→ It sits between normal main memory and CPU.
→ When CPU needs to access contents of memory location, then cache is examined for this data.
  - If present, get from cache (fast).
  - If not present, read required block from main memory to cache, then deliver from cache to CPU.
→ Cache includes tags to identify which block of main memory is in each cache slot.
→ The performance of cache memory is frequently measured in terms of a quantity called **hit ratio.** When CPU refers to memory and finds the word in cache, then it is said to produce hit. If word is not found in cache, it is in main memory and it counts as a miss. The ratio of the number of hits (success in finding the words in cache) to the total CPU references to memory (hits + misses) is known as hit ratio.
→ The basic characteristic of cache memory is its fast access time.

**Cache Mapping**

The process of transferring the data from main memory to cache is known as mapping process. There are three types of cache mapping techniques:

- ❖ Associative mapping
- ❖ Direct mapping
- ❖ Set-associative mapping

→ The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

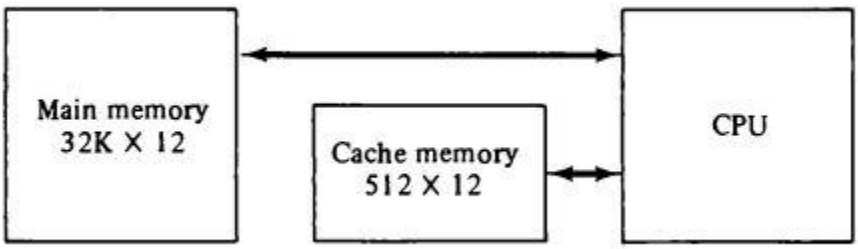



Fig: Example of Cache memory

**Associative Mapping**

The fastest and most flexible cache organization uses an associative memory. This organization is illustrated in Fig. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five digit octal number and its corresponding 12 bit word is shown as a four digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12 bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The address-data pair

is then transferred to the associative cache memory. If the cache is full, an address-data pair must be displaced to make room for a pair that is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round robin order whenever a new word is requested from main memory. This constitutes a first in first out (FIFO) replacement policy.

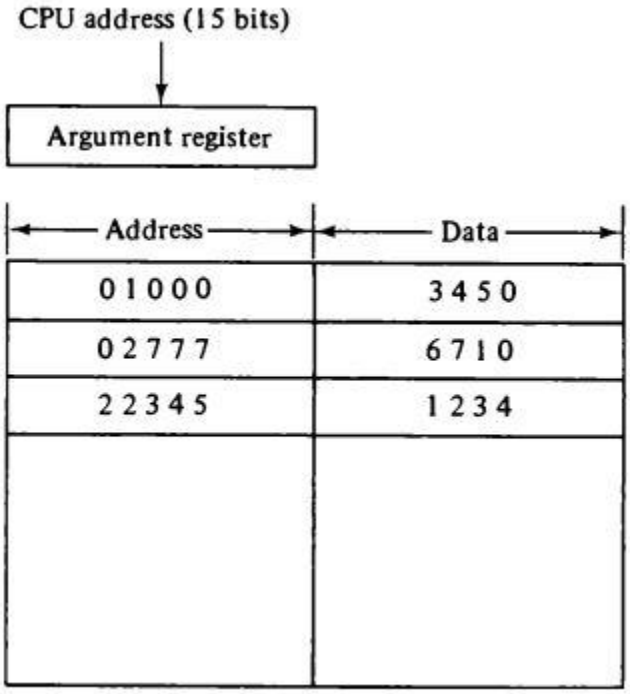



Fig: Associative mapping cache (all numbers in octal)

**Direct Mapping**

→ The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the *index field* and the remaining six bits form the *tag field*.

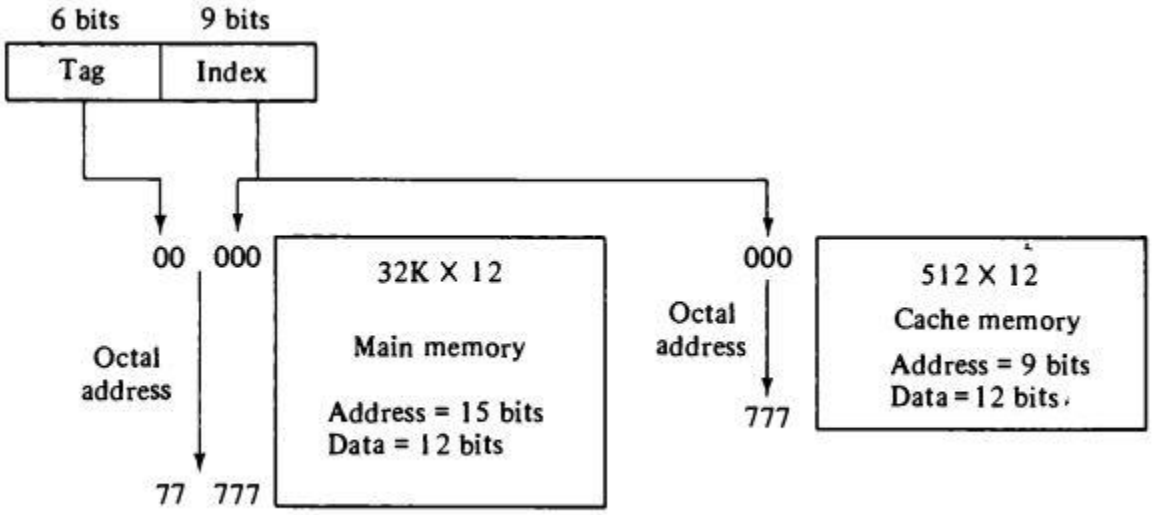



Fig: Addressing relationships between main and cache memories.

→ The figure shows that main memory needs an address that includes both the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

→ The n bit memory address is divided into two fields: k bits for the index field and n-k bits for the tag field. The direct mapping cache organization uses the n bit address to access the main memory and the k bit index to access the cache.

→ Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.

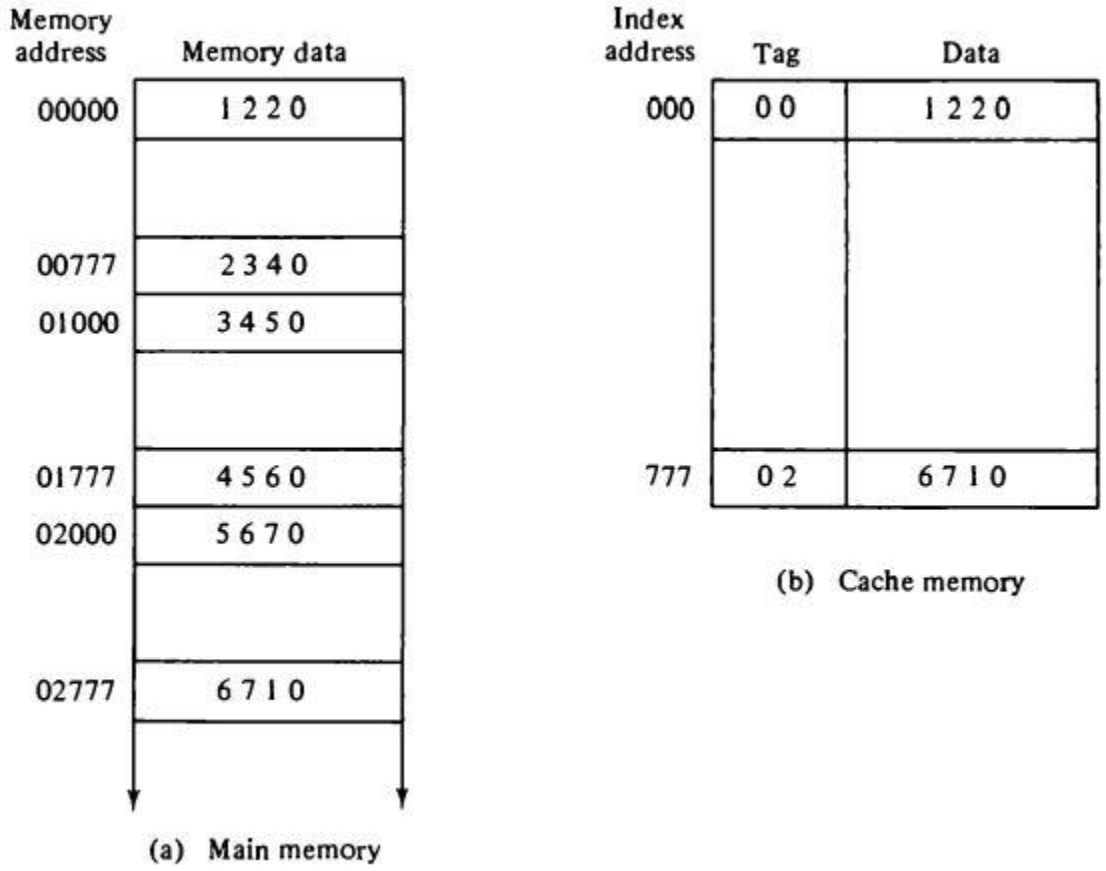



Fig: Direct mapping cache organization.

→ The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value.

→ The disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.

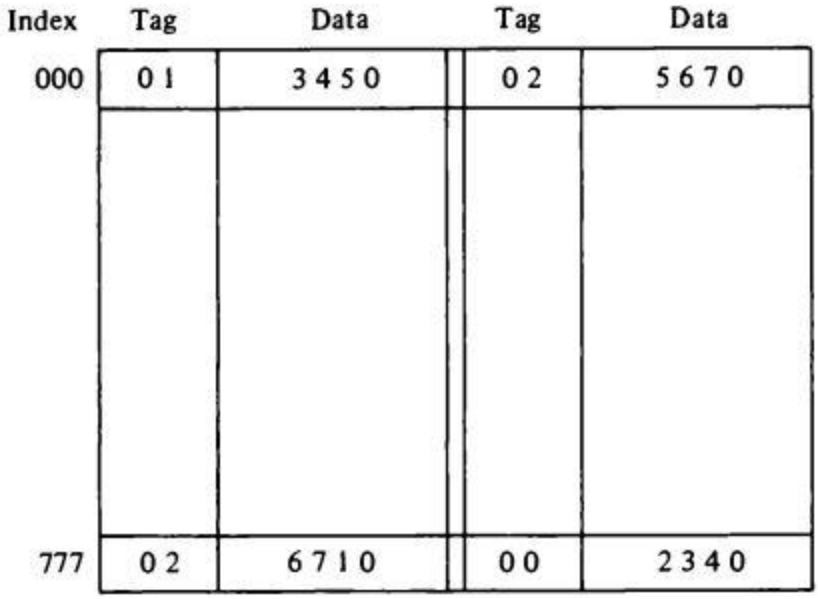
**Set-Associative Mapping**

→ It is an improvement over the direct mapping organization in that each word of cache can store two or more words of memory under the same index address.

→ Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set.

→ Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits.

→ An index address of nine bits can accommodate 512 words. Thus, the size of cache memory is 512*36. It can accommodate 1024 words of main memory since each word of cache contains two data words.



| Index | Tag | Data | Tag | Data |
|---|---|---|---|---|
| 000 | 01 | 3450 | 02 | 5670 |
| | | | | |
| 777 | 02 | 6710 | 00 | 2340 |

Fig: Two-way set associative mapping cache.

→ The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777.

→ When a miss occurs in a set associative cache and the set is full, it is necessary to replace one of the tag data items with a new value.

→ The most common replacement algorithms used are: *random replacement*, *first in first out (FIFO)*, and *least recently used (LRU).*

## 10.5 Virtual Memory

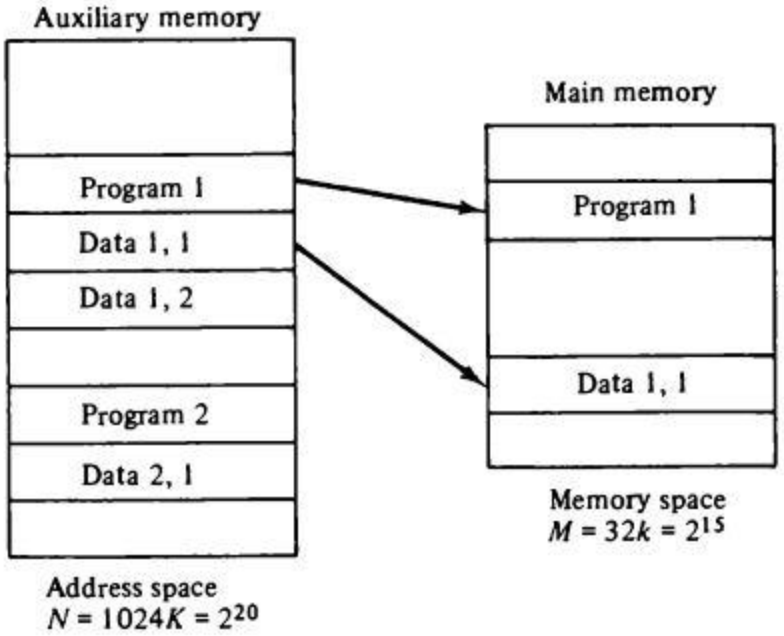
→ A virtual memory system attempts to optimize the use of the main memory (the higher speed portion) with the hard disk (the lower speed portion). In effect, **virtual memory** is a technique for using the secondary storage to extend the apparent limited size of the physical memory beyond its actual physical size. It is usually the case that the available physical memory space will not be enough to host all the parts of a given active program.

→ Virtual memory gives programmers the illusion that they have a very large memory and provides mechanism for dynamically translating program-generated addresses into correct main memory locations. The translation or mapping is handled automatically by the hardware by means of a mapping table.

**Address Space and Memory Space**

An address used by the programmer is a virtual address (virtual memory addresses) and the set of such addresses is the **Address Space**. An address in main memory is called a location or physical address. The set of such locations is called the **Memory Space**. Thus the address space is the set of addresses generated by the programs as they reference instructions and data; the memory space consists of actual main memory locations directly addressable for processing. Generally, the address space is larger than the memory space.

Example: consider main memory: 32K words (K = 1024) = $2^{15}$ and auxiliary memory 1024K words = $2^{20}$. Thus we need 15 bits to address physical memory and 20 bits for virtual memory (virtual memory can be as large as we have auxiliary storage).




→ Here auxiliary memory has the capacity of storing information equivalent to 32 main memories.
→ Address space N = 1024K
→ Memory space M = 32K
→ In multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on the demands imposed by the CPU.

Fig: Relation between address and memory space in a virtual memory system

In virtual memory system, address field of an instruction code has a sufficient number of bits to specify all virtual addresses. In our example above we have 20-bit address of an instruction (to refer 20-bit virtual address) but physical memory addresses are specified with 15-bits. So a table is needed to map a virtual address of 20-bits to a physical address of 15-bits. Mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

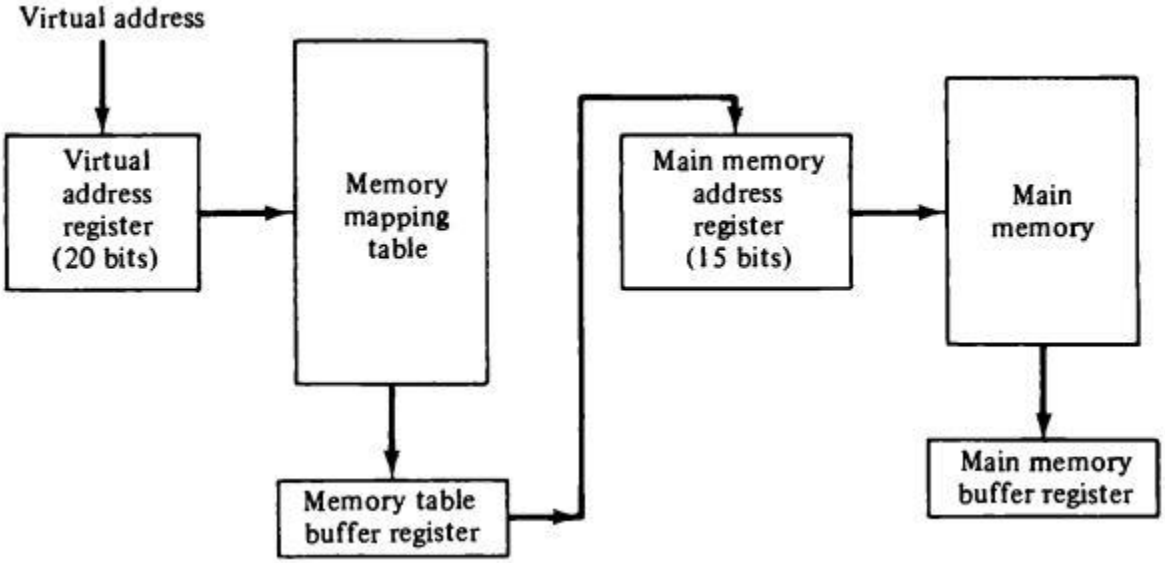



Fig: Memory table for mapping a virtual address

## 10.6 Memory Management Hardware

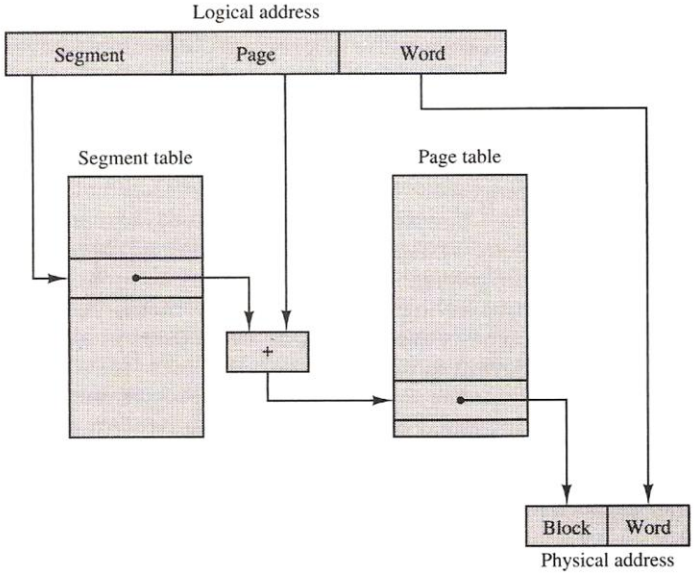
A memory management system is a collection of hardware and software procedures for managing various programs (effect of multiprogramming support) residing in memory. Basic components of memory management unit (MMU) are:

→ A facility for dynamic storage relocation that maps logical memory references into physical memory addresses.
→ A provision for sharing common programs by multiple users.
→ Protection of information against unauthorized access.

The dynamic storage relocation hardware is a mapping process similar to paging system.
**Segment**: It is more convenient to divide programs and data into logical parts called segments despite of fixed-size pages. A **segment** is a set of logically related instructions or data elements. Segments may be generated by the programmer or by OS. Examples are: a subroutine, an array of data, a table of symbols or user's program.

**Logical address**: The address generated by the segmented program is called a *logical address*. This is similar to virtual address except that logical address space is associated with variable-length segments rather than fixed-length pages.

**Segmented-Page Mapping**
The length of each segment is allowed to grow and contract according to the needs of the program being executed. One way of specifying the length of a segment is by associating with it a number of equal-sized pages.
Consider diagram below:
Logical address = Segment + Page + Word
Where **segment** specifies segment number, **page** field specifies page within the segment and **word** field specifies specific word within the page.




→ Mapping of logical address to physical address is done by using two tables: segment and page table.
→ The entry in the segment table is a pointer address for the page table base, which is then added to page number (given in logical address). The sum point to some entry in page table and content of that page is the address of physical block. The concatenation of block field with the word field produces final **physical mapped address**.

Fig: Logical to Physical address mapping

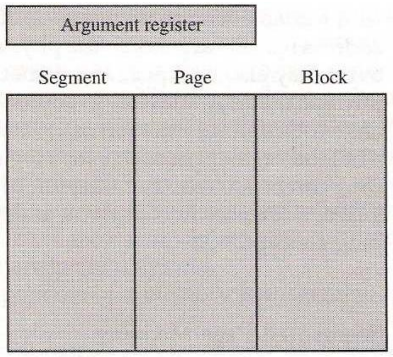



Fig: Associative Memory: Translation Lookaside Buffer (TLB)

→ This is a fast associative memory (TLB) and holds most recently referenced entries.
(Alternatively we could store above two tables: segment table and page table, in two separate small memories which really increases the CPU access time).

**H\W,** *See Numerical example to clear the concept of MMU (Computer System Architecture, 3rd edition, page no. 481, Morris Mano)*

# Chapter 11
# Multiprocessor

## 11.1 Introduction and Characteristics of Multiprocessor

→ The system in which two or more processing units (CPU or IOP) are connected to the memory and I/O devices is known as multiprocessor system.

→ 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.

**Characteristics of multiprocessors**

- Multiple processing elements
- System is controlled by single operating system
- System is more reliable

The system derives its high performance from the fact that computations can proceed in parallel in one of two ways:

- Multiple independent jobs can be made to operate in parallel.
- A single job can be partitioned into multiple parallel tasks.

**Types of multiprocessor**

→ Multiprocessors are classified by the way their memory is organized. They are:

→ **Shared memory or tightly-coupled multiprocessor:** the multiprocessor system in which all processing elements share a common memory. In this type, there is no local memory with processor but they have their own cache memory.

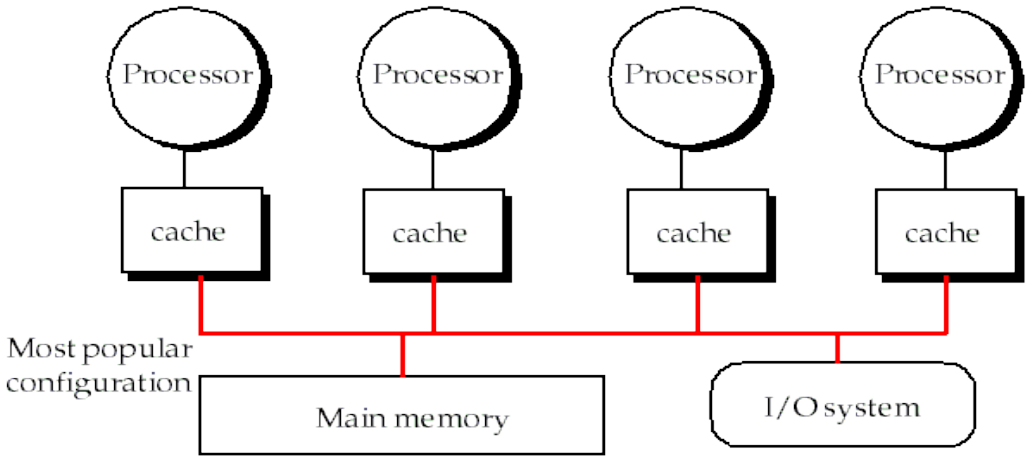



Fig: Shared Memory Architecture

→ **Distributed memory or loosely-coupled multiprocessor:** the multiprocessor system in which each processing element has its own private local memory and all the processors are tied together by a switching mechanism to route information from one processor to another through a message passing scheme.

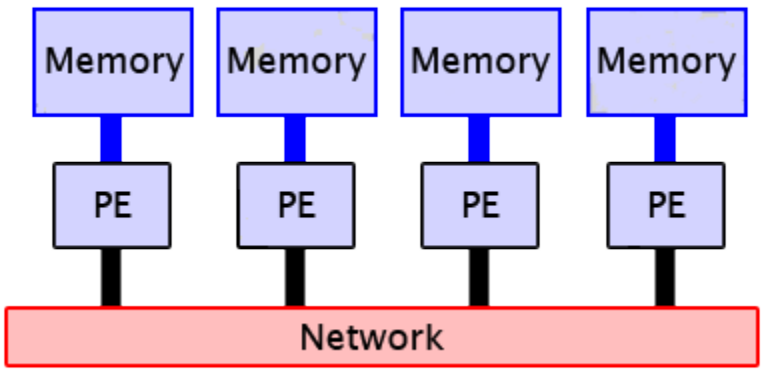


Fig: Distributed Memory Architecture

## 11.2 Interconnection Structures

→ The components that form a multiprocessor system are CPUs, IOPs connected to I/O devices, and a memory unit that may be partitioned into a number of separate modules.

→ The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system.

→ There are several interconnection networks. Some of these schemes are given as:

1. Time shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

**1. Time-shared common bus**

→ A common bus is used for all CPU to communicate with shared memory. At any given time, only one processor can communicate with the memory or another processor but all the processors are either busy with their internal operation or idle waiting for the bus.

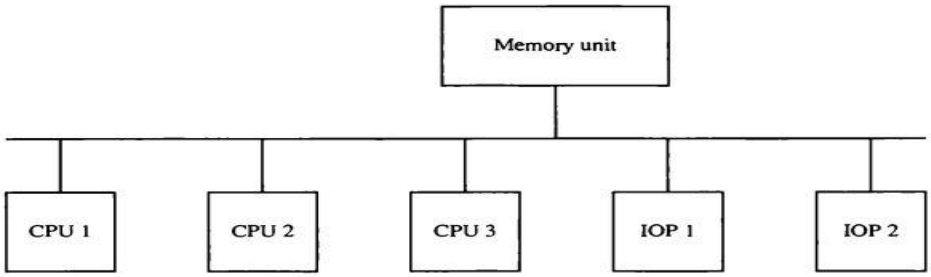



Fig: Time-shared Common Bus Organization.

→ Advantages:

- Simplicity
- Flexibility
- Reliability

→ Disadvantages

- Performance limited by bus cycle time
- Each processor should have local cache
- Reduce number of bus accesses
- Leads to problems with cache coherence

**2. Multiport Memory**

→ It uses separate buses between each memory module and each CPU.

→ Each processor has direct independent access of memory modules by their own bus connected to each module.

→ The modules must have internal control logic to determine which port will have access to memory at any given time.

→ Memory access conflicts are resolved by assigning fixed priorities to each memory port. CPU1 will have priority over CPU2, CPU2 will have priority over CPU3 and CPU4 will have the lowest priority.

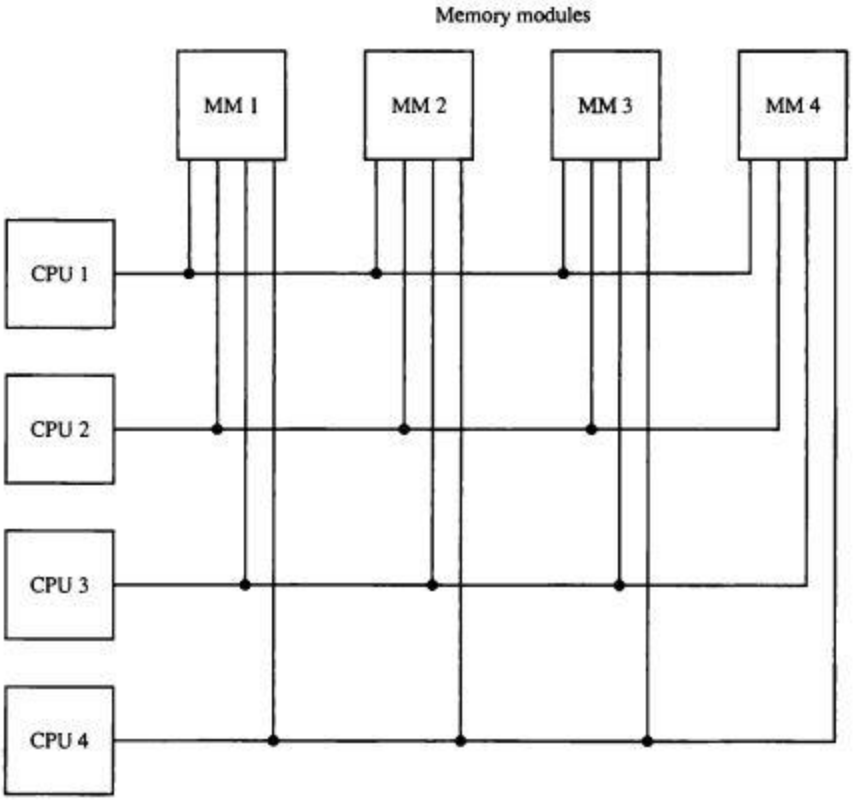



Fig: Multiport memory organization.

→ Advantages:
- Better performance because each processor has dedicated path to each module.
- Can configure portions of memory as private to one or more processors so increased security.

→ Disadvantages:
- Requires extra control logic so more complex and increase the cost.
- Requires large connection wires.

**3. Crossbar Switch**

→ A crossbar switch (also known as cross-point switch or matrix switch) is a switch connecting multiple inputs to multiple outputs in a matrix manner.

→ The crossbar switch organization consists of a number of cross points that are placed at interconnection between processor bus and memory module path.

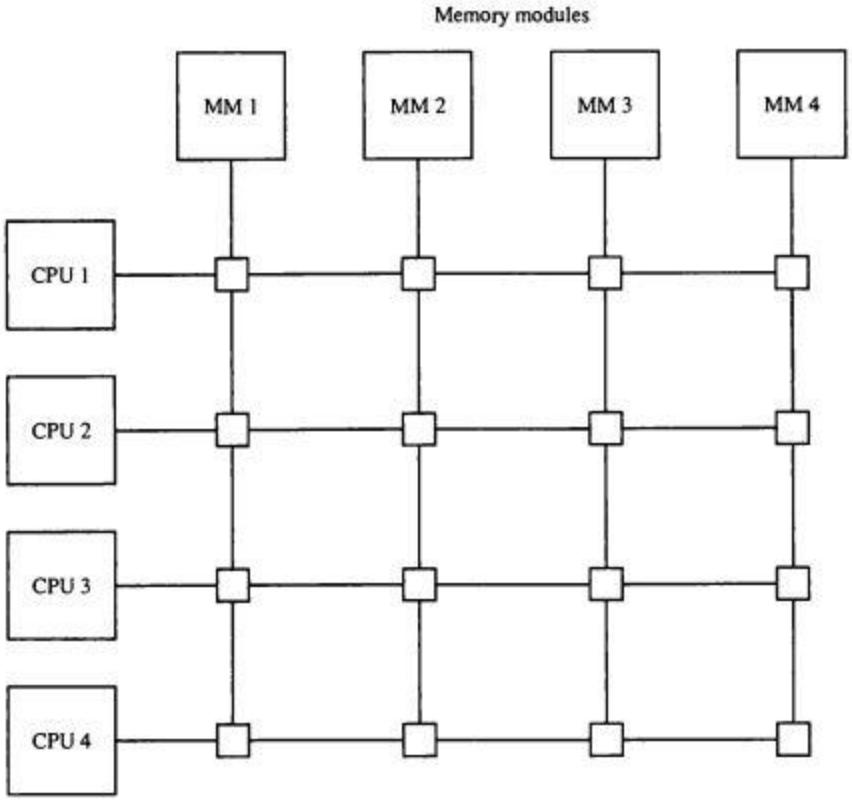



Fig: Crossbar switch.

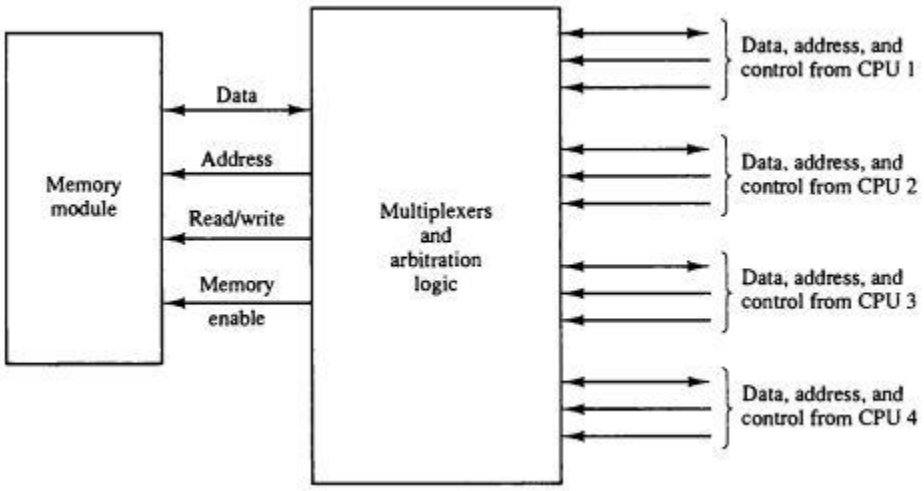



Fig: Block diagram of crossbar switch.

→ Above figure shows a crossbar switch interconnection of four CPUs and four memory modules. The small square in each cross point is a switch that determines the path from a processor to a memory module.

→ Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed. It also resolves multiple requests for access to the same memory module on a predetermined priority basis.

**4. Multistage Switching Network**

→ Controls the communication between a number of resources and destinations.

→ Basic components of a multistage switching network are two-input, two-output interchange switch.

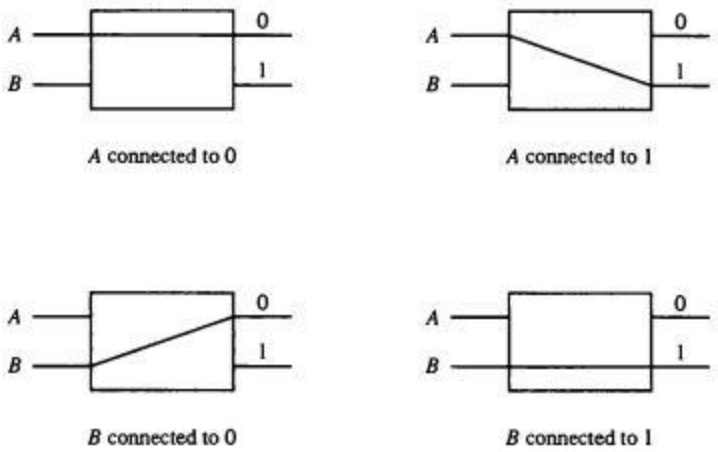



Fig: Operation of a 2×2 interchange switch.

→ As shown in above figure, the 2*2 switch has two inputs labeled A and B, and two outputs 0 and 1. There are control signals associated with the switch that establish the interconnection between the input and output terminals.

→ The switch has capacity of connecting input A to either of the outputs. Terminal B of the switch behaves in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If inputs A and B both request the same output terminal, only one of them will be connected, the other will be blocked.

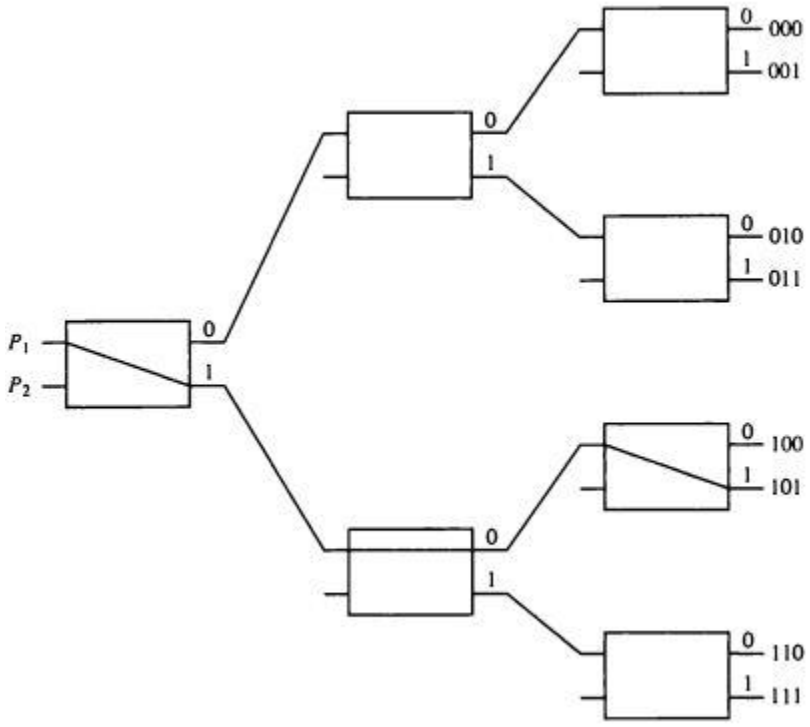



Fig: Binary tree with 2×2 switches.

→ Using the 2*2 switch as a building block, it is possible to build a multistage network as in figure.

**5. Hypercube Interconnection**

→ The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N=2^n$ processors interconnected in an n-dimensional binary cube.

→ Each processor forms a node of the cube. Each processor has direct communication paths to n other neighbor processors. These paths correspond to the edges of the cube.

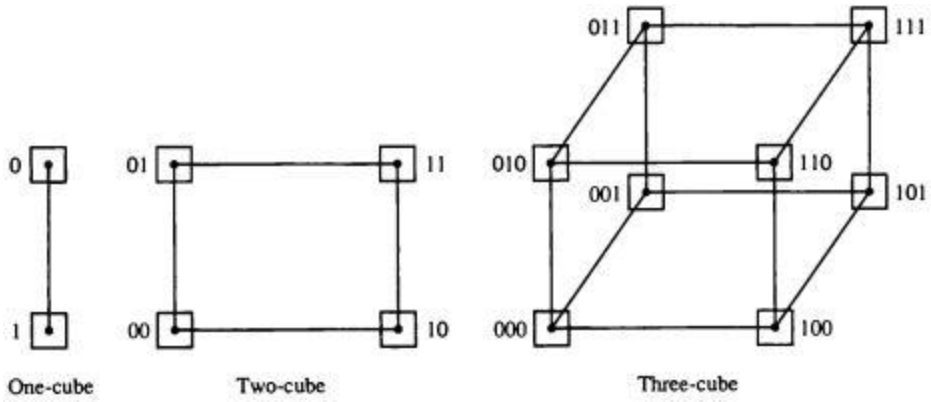



Fig: Hypercube structures for n = 1, 2, 3.

→ Above figure shows the hypercube structure for n = 1, 2, and 3.

→ A one-cube structure has n = 1 and $2^1 = 2$. It contains two processors interconnected by a single path.

→ A two-cube structure has n = 2 and $2^2 = 4$. It contains four nodes interconnected as a square.

→ A three-cube structure has eight nodes interconnected as a cube. An n-cube structure has $2^n$ nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three cube structure are 000, 110, and 101. Each of these binary numbers differs from address 100 by one bit value.

→ Routing messages through an n cube structure may take from one to n links from a source node to a destination node. For example, in a three cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011 (from 000 to O01 to 011 or from 000 to 010 to 011). It is necessary to go through at least three 1ines to communicate from node 000 to node 111.

→ A routing procedure can be developed by computing the exclusive OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three cube structure, a message at 010 going to 001 produces an exclusive OR of the two addresses equal to O11. The message can be sent along the second axis to 000 and then through the third axis to 001.

## 11.3 Interprocessor Arbitration

→ The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately.

→ However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time.

→ Arbitration must then be performed to resolve this multiple contention for the shared resources. The arbitration logic would be part of the system bus controller placed between the local bus and the system bus.

→ Some arbitration processes are:

### 1. Serial Arbitration Procedure (Daisy-Chain Arbitration)

→ Each processor has its bus arbiter logic with priority-in (PI) priority-out (PO) lines.

→ The priority out (PO) of each arbiter is connected to the priority in (Pl) of the next lower priority arbiter. The PI of the highest priority unit is maintained at logic 1 value.

→ The highest priority unit in the system will always receive access to the system bus when it requests it.

→ The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus. This is the way that priority is passed to the next unit in the chain. If the processor requests control of the bus and the corresponding arbiter finds its PI input equal to 1, it sets its PO output to 0. Lower priority arbiters receive a 0 in PI and generate a O in PO. Thus the processor whose arbiter has a PI = 1 and PO = 0 is the one that is given control of the system bus.

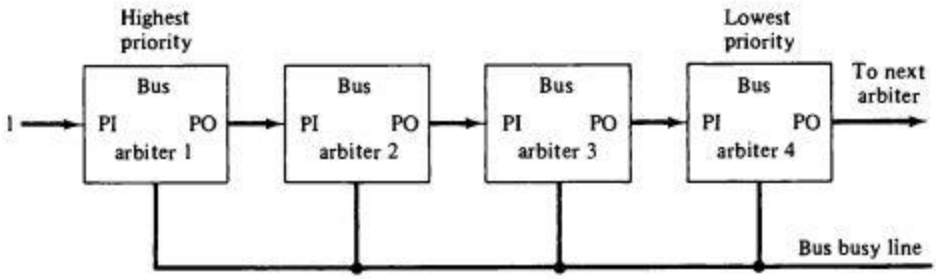



Fig: Serial (daisy chain) arbitration.

**2. Parallel Arbitration Process**

→ The parallel bus arbitration technique uses an external priority encoder and a decoder. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line.

→ The processor takes control of the bus if its acknowledged input line is enabled. The bus busy line provides an orderly transfer of control, as in the daisy chaining case.

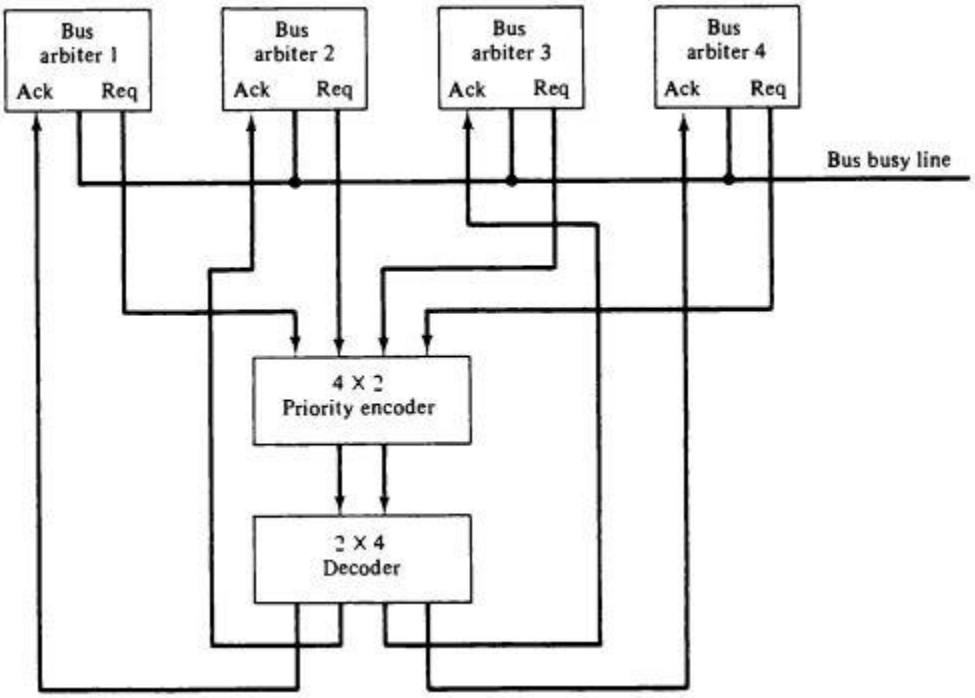



Fig: Parallel arbitration.

→ Above figure shows the 2-bit code from the encoder output drives a 2 X 4 decoder which enables the proper acknowledge line to grant bus access to the highest priority unit.

**3. Dynamic Arbitration Algorithm**

→ Serial and Parallel arbitration procedures use a static priority algorithm since the priority of each device is fixed by the way it is connected to the bus.

→ In contrast, a **DAA** gives the system the capability for changing the priority of the devices while the system is in operation.

→ There are 5 types of DAA:

i) **Time Slice**: The time slice algorithm allocates a fixed length time slice of bus time that is offered sequentially to each processor, in round robin fashion. The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.

ii) **Polling**: In a bus system that uses polling, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. After a number of bus cycles, the polling process continues by

choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.

iii) **Least Recently Used (LRU)**: The least recently used (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. With this, no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

iv) **First-Come, First-Serve (FIFO) scheme**: In the first come, first serve scheme, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive.

v) **Rotating Daisy-Chain**: The rotating daisy chain procedure is a dynamic extension of the daisy chain algorithm. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter whose processor is currently controlling the bus. Once an arbiter releases the bus, it has the lowest priority.

## 11.4 Interprocessor Communication and Synchronization

### Interprocessor Communication

→ Communication refers to the exchange of data between different processes. For example, parameters passed to a procedure in a different processor constitute inter processor communication.

→ The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common input output channels. In a shared memory multiprocessor system, the most common procedure is to set aside a portion of memory that is accessible to all processors.

→ In addition to shared memory, a multiprocessor system may have other shared resources. For example, a magnetic disk storage unit connected to an IOP may be available to all CPUs.

→ To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. This task is given to the operating system. There are three organizations that have been used in the design of operating system for multiprocessors: **1) master slave configuration, 2) separate operating system, and 3) distributed operating system**.

→ In a **master slave mode**, one processor, designated the master, always executes the operating system functions. The remaining processors, denoted as slaves, do not perform operating system functions. If a slave processor needs an operating system service, it must request it by interrupting the master and waiting until the current program can be interrupted.

→ In the **separate operating system organization**, each processor can execute the operating system routines it needs. This organization is more suitable for loosely coupled systems where every processor may have its own copy of the entire operating system.

→ In the **distributed operating system organization**, the operating system routines are distributed among the available processors. However, each particular operating system function is assigned to only one processor at a time. This type of organization is also referred to as a floating operating system since the routines float from one processor to another and the execution of the routines may be assigned to different processors at different times.

→ In a loosely coupled multiprocessor system the memory is distributed among the processors and there is no shared memory for passing information. The communication between processors is by means of message passing through l/O channels.

### Interprocessor Synchronization

→ The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes.

→ Synchronization refers to the special case where the data used to communicate between processors is control information. Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.

→ A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is through the use of a binary **semaphore**.
→ A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources. This is necessary to protect data from being changed simultaneously by two or more processors. This mechanism has been termed **mutual exclusion**.
→ Mutual exclusion must be provided in a multiprocessor system to enable one processor to exclude or lock out access to a shared resource by other processors when it is in a critical section. A **critical section** is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource.
→ A binary variable called a semaphore is often used to indicate whether or not a processor is executing a critical section. A semaphore is a software controlled flag that is stored in a memory location that all processors can access. When the semaphore is equal to 1, it means that a processor is executing a critical program, so that the shared memory is not available to other processors.
→ When the semaphore is equal to 0, the shared memory is available to any requesting processor. Processors that share the same memory segment agree by convention not to use the memory segment unless the semaphore is equal to 0, indicating that memory is available.
→ They also agree to set the semaphore to 1 when they are executing a critical section and to clear it to 0 when they are finished.
→ A semaphore can be initialized by means of a test and set instruction in conjunction with a hardware **lock** mechanism.
→ Assume that the semaphore is a bit in the least significant position of a memory word whose address is symbolized by SEM. Let the mnemonic TSL designate the "test and set while locked" operation.
→ The instruction TSL SEM will be executed in two memory cycles (the first to read and the second to write) without interference as follows:

| | |
|---|---|
| R←M [SEM] | Test semaphore |
| M [SEM]←1 | Set semaphore |

## 11.5 Cache Coherence

→ In multiprocessor system, if the update made on some data item on local cache of any processor is reflected on other processor having the same data item, then it is said to be cache coherence. Or, if same data item resides in cache of multiple processors and its value is same to all cache, then it is known as cache coherence.
→ Cache coherence arises when shared data is to be written as well as read. If one processor modifies a cached value shared in cache by other processors, then all processors must eventually agree on the updated value.
→ For example, the system shown on fig. 1 is coherent system because in all local cache, the value of X=52.

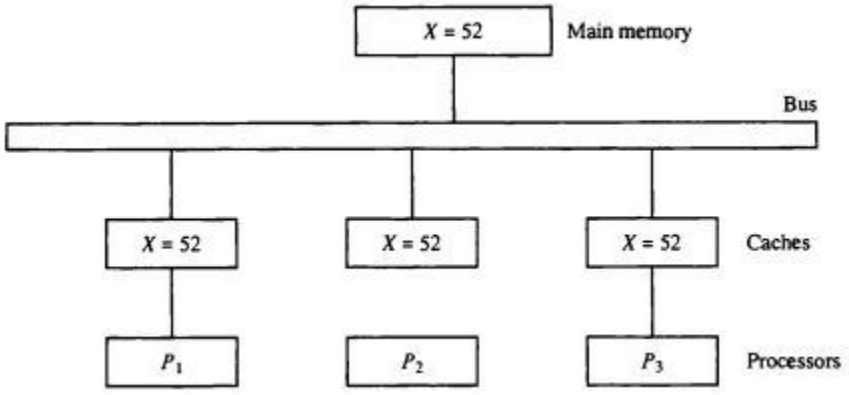



Fig: Cache configuration after a load on X.

→ But in fig. 2, when value at X is updated on local cache of processor P1 using write through cache, then it is not reflected to local caches of P2 and P3. Hence, it is not coherent system.

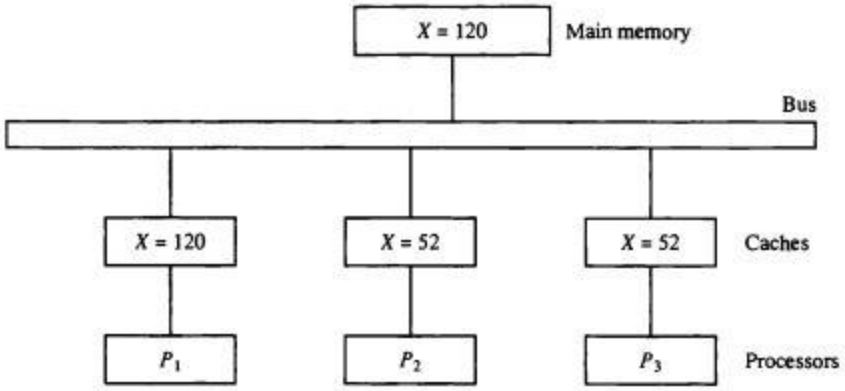



Fig: With write-through cache policy

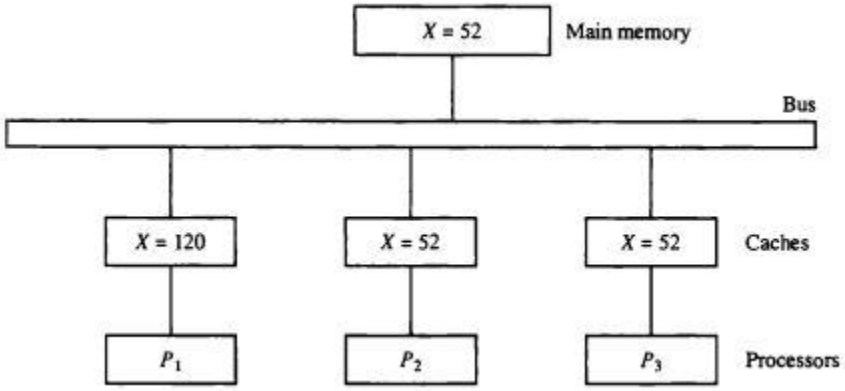



Fig: With write-back cache policy

→ Solution to cache coherence

i) Using shared cache
- All the processors use the same global cache.

ii) Non-cacheable data
- Shared writable data are made non-cacheable.
- Non-shared and readable data are made cacheable.

iii) Two separate caches
- One global cache for writable block of data.
- Local cache for readable data.

iv) Snoopy cache controller
- It is a special hardware that monitors the write operation in any local cache.
- If the cache write is observed, then main memory is updated, and every cache controller sees. If they have the same data, then they mark the data invalid.