MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

Autonomous Institution – UGC, Govt. of India

Department of COMPUTATIONAL INTELLIGENCE

CSE (AI&ML)

B.TECH (R-22 Regulation)

(II YEAR – I SEM)

2024-25

COMPUTER ORGANIZATION AND ARCHITECTURE

(R22A1261)

LECTURE NOTES

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12(B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE-Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad–500100, Telangana State, India

Department of COMPUTATIONAL INTELLIGENCE

CSE (ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

COMPUTER ORGANIZATION AND ARCHITECTURE

(R22A1261)

LECTURE NOTES

SK. Subhani

Assistant Professor

Department of Computational Intelligence
CSE (Artificial Intelligence and Machine Learning), Artificial Intelligence and Machine Learning

Vision

To be a premier centre for academic excellence and research through innovative interdisciplinary collaborations and making significant contributions to the community, organizations, and society as a whole.

Mission

- To impart cutting-edge Artificial Intelligence technology in accordance with industry norms.
- To instill in students a desire to conduct research in order to tackle challenging technical problems for industry.
- To develop effective graduates who are responsible for their professional growth, leadership qualities and are committed to lifelong learning.

QUALITY POLICY

- To provide sophisticated technical infrastructure and to inspire students to reach their full potential.
- To provide students with a solid academic and research environment for a comprehensive learning experience.
- To provide research development, consulting, testing, and customized training to satisfy specific industrial demands, thereby encouraging self-employment and entrepreneurship among students.

For more information: www.mrcet.ac.in

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF COMPUTATIONAL INTELLIGENCE

II Year B.Tech. CSE(AIML)- I Sem **L/T/P/C**
3/-/-/3

(R22A1261) COMPUTER ORGANIZATION AND ARCHITECTURE

OBJECTIVES:
The students will be able to:

1. To understand the working of a Computer System and its basic principles.
2. To learn the architecture and design of 8086 processor.
3. To know the concepts of Memory and corresponding technologies.
4. To understand the functional aspects of various peripheral devices.
5. To acquire knowledge about parallel processors.

UNIT - I:
Functional blocks of a computer: CPU, memory, input-output subsystems, control unit. Computer Organization and Architecture - Von Neumann
Data representation: signed number representation, fixed and floating-point Representations, Character representation. Computer arithmetic – integer addition and Subtraction, Ripple carry adder, carry look-ahead adder, etc. Multiplication – shift-and add, Booth multiplier.

UNIT – II:
Introduction to x86 architecture.
Instruction set architecture of a CPU: Registers, instruction execution cycle, RTL Interpretation of instructions, addressing modes, instruction set.
CPU Control unit design: Micro-programmed design approach.

UNIT – III:
Memory system design: Semiconductor memory technologies, memory organization.
Memory organization: Memory interleaving, concept of hierarchical memory organization, Cache memory, cache size vs. block size, mapping functions, Replacement algorithms, write policies.

UNIT – IV:
Peripheral devices and their characteristics: Input-output subsystems, I/O device interface, I/O transfers – program controlled, interrupt driven and DMA, privileged and non-privileged instructions, software interrupts and exceptions. Programs and processes – role of interrupts in process state transitions.

UNIT – V:

Pipelining: Basic concepts of pipelining, throughput and speedup, pipeline hazards.

Parallel Processors: Introduction to parallel processors, Concurrent access to memory and cache coherency.

TEXT BOOKS:

1."Computer System Architecture", 3rd Edition by M.Morris Mano, Pearson.

2. **"**Computer Organization and Design: The Hardware/Software Interface", 5th Edition by David A. Patterson and John L. Hennessy, Elsevier.

3."Computer Organization and Embedded Systems", 6th Edition by Carl Hamacher, McGraw Hill Higher Education.

REFERENCE BOOKS:

Course Outcomes:

At the end of the course, Students will be able to:

1. Illustrate the functional block diagram of a single bus architecture of a computer.
2. Analyze the various instruction sets and addressing modes.
3. Design a memory module and analyze its operation by interfacing with the CPU for a specific architecture.
4. Compare and contrast the peripherals and the related I/O transfers
5. Assess the performance, and apply design techniques to enhance performance using pipelining & parallelism.

COA UNIT-I (part-1)

→ **Overview of Microcomputer structure and operation:-**

Functional parts of the Micro Computer:-

the block diagram for a simple microcomputer. The major parts are the central processing unit or CPU, memory and input and output circuitry or I/O. Connecting these parts are three sets of parallel lines called buses. The three buses are the address bus, the data bus, and the control bus.

Memory:- the memory section consists of a mixture of RAM and ROM. It may also have magnetic disks magnetic hard disks, or optical disks. Memory has two purposes. the first purpose is to store the binary codes for the sequence of instructions you want the computer to carry out. The second purpose of the memory is to store the binary coded data with which the computer is going to be worked.

Input/output:- The input/output or I/O section allows the computer to take data from the outside world or send data to the outside world.

Peripherals such as keyboards, Video display terminals, printers and modems are connected to the I/O section. This allows the

user and computer to communicate with each other. The best example for the input devices is A/D converter and the best example for output device is D/A Converter.

I/O ports:- The actual physical device used to interface the computer buses to external systems are often called ports. The port is simply a group of D-flipflops.

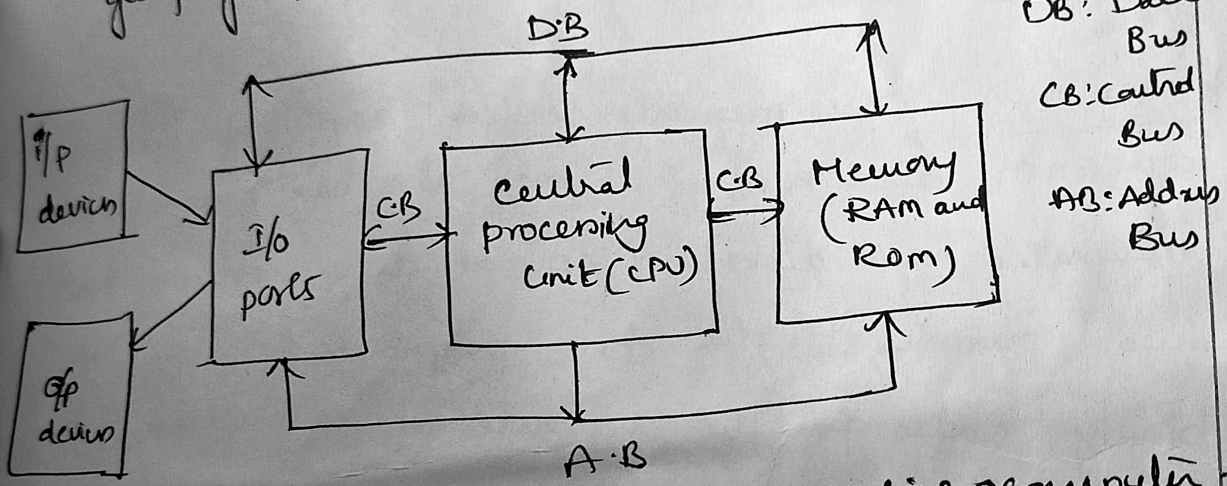


fig: Block diagram of the microcomputer

Central Processing Unit:- The central processing unit or CPU controls the operation of the computer. In a microcomputer the CPU is a microprocessor. The CPU fetches binary coded instructions from memory, decodes the instructions into a series of simple actions and carries out these actions in a sequence of steps.

The CPU also contains an address counter or instruction pointer register which holds the address of address from near next executable instruction or data item to be fetched from memory, - general purpose registers, which are used for temporary storage of binary data and circuitry which generates the control bus signals.

Address Bus:- The address Bus consists of 16, 20, 24 or 32 parallel signal lines. On these lines the CPU sends out the address of memory location that is to be written to or read from. The no. of memory locations that the CPU can address is determined by the number of address lines. If the CPU has N address lines it can address 2^N memory locations. A CPU with 16 address lines can address 2^{16} memory locations i.e., 65,536 memory location. A CPU with 20 address lines can address 2^{20} i.e., 1,048,576 locations.

Data Bus:- The Data bus consists of 8, 16 or 32 parallel signal lines. As indicated by the double ended arrows on the data bus, the data bus lines are bidirectional. This means that CPU can read data from a part on these lines or it can send data

in from memory or from a port on these lines, or it can send data out to memory or from a port on these lines.

Many devices in a system will have their outputs connected to the databus but only one device will have its o/ps connected to the data bus i.e. outputs enabled.

Control Bus :-

The control bus consists of 4 to 10 parallel signal lines. The CPU sends out signals on the control bus to enable the o/ps of addressed memory devices or port devices. Typical control bus signals are Memory Read, Memory Write, I/O Read and I/O Write.

To read a byte of data from a memory location the CPU sends out the memory address of the desired byte on the address bus and then sends out a Memory Read signal on the control bus. the memory read signal enables the addressed memory device to output a data word onto the databus

Computer Organization and Architecture

Computer Organization is concerned with the structure and behaviour of a computer system as seen by the user. Organization deals with physical components (circuit design, adders, signals & peripherals)

Computer Architecture helps us to understand the functionalities of a system. A programmer can view architecture in terms of instruction, addressing modes.

Von-Neumann Architecture :-

Von-Neumann Architecture follows stored program organization. All modern computers are based on a stored-program concept introduced by John Von Neumann. In this stored program concept, programs and data are stored in a separate storage unit called memories and are treated the same. This novel idea meant that a computer built with this architecture could be much easier to reprogram.

Data Representation :-

Number systems

1. Decimal Number System
2. Binary Number System
3. Octal Number system
4. Hexa decimal Number system
5. Binary coded decimal Number System

Binary Number system

The base (or) radix of the binary no system is 2. It consists of only two different digits '0' and '1'.

Octal Number system

0	0000	2	0010	4	0100	8	1000
1	0001	3	0011	5	0101	9	1001
				6	0110	10	1010
				7	0111	11	1011
						12	1100
						13	1101
						14	1110
						15	1111

Octal Number system:-

The base (or) radix of octal no system is '8'. It consists of 8 different digits '0' to '7'.

			70	100	
0	10	20		⋮	
1	11	21		⋮	
2	12	22		⋮	
3	13	⋮		⋮	
4	14	⋮		⋮	
5	15	⋮		⋮	
6	16	⋮			
7	17	⋮ ----	77	107 ----	777

Hexa Decimal Number system

The base (or) radix of hexadecimal no system is '16'. It consists of '16' different digits '0' to 'f' It is as shown below.

0	10	20	90	A0	100
1	11	21			
2	12	22			
3	13	23			
4	14	24			
5	15	25			
6	16	26			
7	17	27			
8	18	28			
9	19	29			
A	1A	2A			
B	1B	2B			
C	1C	2C			
D	1D	2D			
E	1E	2E			
F	1F	2F	9F	AF ..... ff	10f ... fff

Binary coded Decimal Number system:-

In this no system decimal digits are coded into binary format.

9 0 8 7 (10)

1001 0000 1000 0111 (BCD)

Every decimal digit has to be written 4 bit binary equivalent.

Conversions.

Decimal to binary:-

$75_{(10)}$

2 | 75
2 | 37 — 1
2 | 18 — 1
2 | 9 — 0
2 | 4 — 1
2 | 2 — 0
1 — 0

$1001011_{(2)}$

perform repeated division with '2' and record the remainders in the reverse order.

$75.23_{(10)}$

$75_{(10)} = 1001011_{(2)}$

$75.23_{(10)} = 1001011.00111_{(2)}$

$0.23 \times 2 = 0.46$

$0.46 \times 2 = 0.92$

$0.92 \times 2 = 1.84$

$0.84 \times 2 = 1.68$

$0.68 \times 2 = 1.36$

$.00111_{(2)}$

→ To convert the fractional part into binary multiply the fractional part by '2' leave the integer part and multiply the new fractional part by '2' and so on until fractional part becomes zero or perform multiplication atleast '5' times and record the left over integer parts in forward order

Decimal to octal
and
Decimal to Hexadecimal

} To convert integer part perform repeated division with respective radix 8 or 16. And to convert fractional part multiply it by 8 or 16 leave the integer part and continue the multiplication with fractional part. Exactly same as Decimal to binary.

Binary to Decimal :-

$2^3\ 2^2\ 2^1\ 2^0 \quad 2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}$

$1\ 0\ 1\ 0 \cdot 1\ 1\ 0\ 1_{(2)}$

$[2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 0] \cdot 2^{-1} \times 1 + 2^{-2} \times 1 + 2^{-3} \times 0 + 2^{-4} \times 1$

$[8 + 0 + 2 + 0] \cdot [0.5 + 0.25 + 0 + 0.0625]$

$10.8125_{(10)}$

Octal to Decimal :-

$8^1\ 8^0 \cdot 8^{-1}$

$2\ 3 \cdot 5_{(8)}$

$[8 \times 2 + 1 \times 3] \cdot [5 \times 0.125] = 19.625_{(10)}$

Hexa Decimal to Decimal :-

16^1 16^0 16^{-1}

A B . 4 $_{(16)}$

$[16 \times A + 1 \times B] \cdot [0.0625 \times 4]$

$171 + 2.5 = 173.5_{(10)}$

Octal to Binary

4 3 7 . 2 6 $_{(8)}$

Allocate '3' binary digits for each and every octal digit. then we get binary equivalent

100 011 111 . 010 110 $_{(2)}$

Binary to Octal :-

1011010 . 10101 $_{(2)}$

starting from binary point towards left side group '3' positions and assign octal equivalent. starting from binary point towards Right side group '3' positions and assign octal equivalent.

010|110|110 . 101|010 $_{(2)}$

2 6 6 . 5 2 $_{(8)}$

$266.52_{(8)}$

Hexadecimal to Binary

B 2 A . C $_{(16)}$

Allocate '4' binary digits for each and every o digit. Then we get binary equivalent.

1011 0010 1010 . 1100 $_{(2)}$

Binary to Hexadecimal :-

$10110101.101101_{(2)}$

Starting from Binary point towards left side group of '4' positions and assign hexa decimal equivalent. ~~and~~ starting from binary point towards Right side group '4' positions and assign hexadecimal equivalent.

$0010|1001|0110.1011|0100_{(2)}$

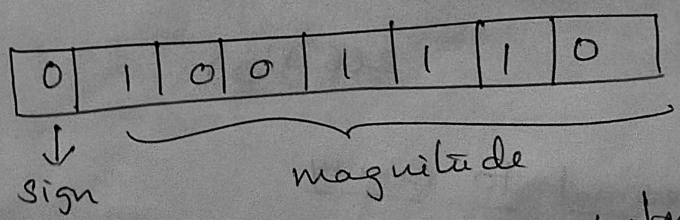
$296.B4_{(16)}$

Signed Integer Representation :-

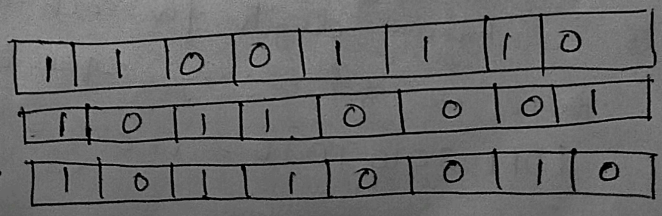
+ve integers can be represented in only one format. i.e., Sign-magnitude form.
-ve integers can be represented in ~~already~~ 3 formats.
They are

1. sign-magnitude form
2. signed-1's complement form.
3. Signed 2's complement form.

Represent $+78_{(10)}$ in sign-mag format :-



Represent $-78_{(10)}$ in Three different formats :-



Floating point Representation:-

IEEE floating point notation consists of (1) single precision floating point representation (32)
(2) double " " " (64)
(3) Extended " " " (80)

Represent the given no $-53.5_{(10)}$ in ~~signed sign~~ single precision and double precision format

$-53.5_{(10)}$

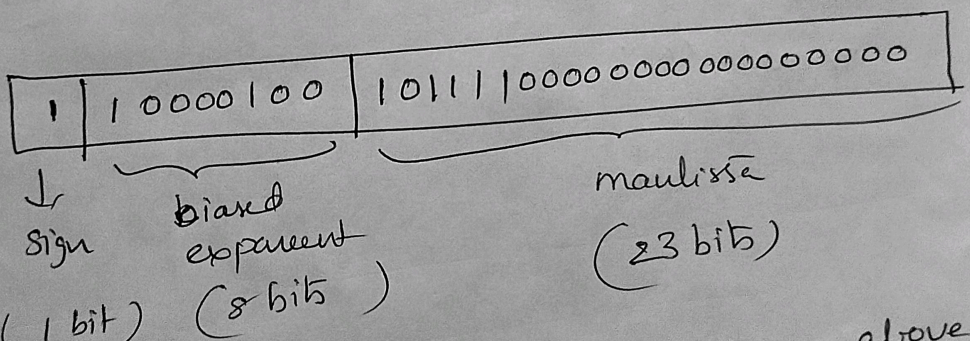
$0.5 \times 2 = 1.0$

2 | 53	
2 | 27	-1
2 | 13	-1
2 | 6	-1
2 | 3	-0
	1 -1

127	
1 5	
2 | 132	
2 | 66	-0
2 | 33	-0
2 | 16	-1
2 | 8	-0
2 | 4	-0
2 | 2	-0
	1 -0

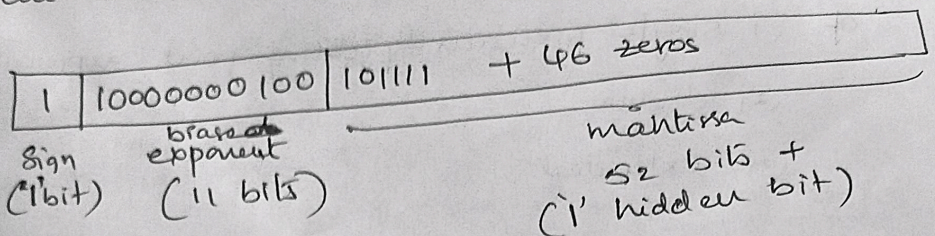
Binary equivalent is

$$-110111.1_{(2)} = (-1.101111) \times 2^5$$



In single precision format as shown above 1-bit is allocated for sign, 8 bits for biased exponent (bias value is 127 bit $(2^{K-1} - 1)$ for ~~signed~~ single precision and 23 bits for the fractional part. The leading bit '1' is not stored (as it is always '1' for a normalized number) and is referred as hidden bit

If we represent the same number $-53.5_{(10)}$ in double precision format.



The bias value in double precision is 1023

$$1023 + 5 = 1028.$$

So, biased exponent is

10000000100

→ In double precision format '1' bit is for sign, '11' bits are for biased exponent and '52' bits are allocated for mantissa and '1' hidden bit is present in mantissa.

1023
5
1028
2 | 1028
2 | 514 – 0
2 | 257 – 0
2 | 128 – 1
2 | 64 – 0
2 | 32 – 0
2 | 16 – 0
2 | 8 – 0
2 | 4 – 0
2 | 2 – 0
1 – 0

UNIT - I Part - 2

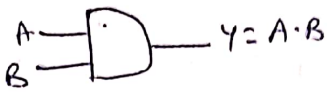
Logic Gates:-

AND

AND Gate TruthTable

A	B	Y = A·B
0	0	0
0	1	0
1	0	0
1	1	1

Symbol



Def:- If all the inputs of AND gate are equal to '1' the o/p is '1'. In all othercases o/p is equal to '0'.

OR

OR Gate TruthTable

A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

Symbol

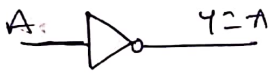
Def:- Of all the inputs of OR gate are equal to '0' the o/p is '0'. In all othercases o/p is equal to '1'.

NOT

Truth Table

A	$Y = \bar{A}$
0	1
1	0

Symbol



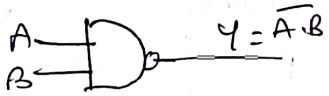
Def:- o/p of the NOT gate is the complement of i/p.

NAND:-

TruthTable

A	B	$Y = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Symbol



Def: NAND gate is the complement of AND gate. The o/p of NAND gate is complement of AND gate: If all the i/ps of NAND gate are equal to '1' the o/p is '0'.

In all the othercases the o/p is '1'.

NAND = AND + NOT.

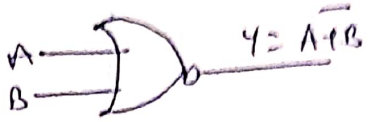
NOR Gate :-

Truth Table :-

A	B	$Y = \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

NOR = OR + NOT

Symbol



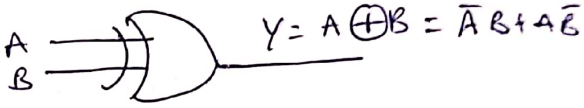
Def :- NOR gate is the complement of OR gate. The o/p of NOR is complement of OR gate. If all the inputs are equal to '0' then o/p is '1'. In all the other cases, the o/p is '0'

Ex-OR gate

Truth Table :-

A	B	$Y = \bar{A}B + A\bar{B}$
0	0	0
0	1	1
1	0	1
1	1	0

Symbol



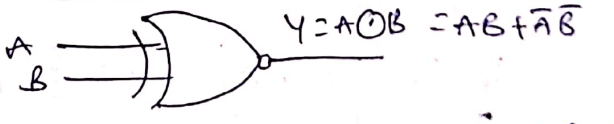
Def : If all the i/ps all Ex-OR gate are equal then o/p is '0'. If all the i/ps are different then o/p is '1'.

Ex-NOR gate

Truth Table :-

A	B	$Y = AB + \bar{A}\bar{B}$
0	0	1
0	1	0
1	0	0
1	1	1

Symbol :-



Def :- If all the i/ps of Ex-NOR gate are equal then o/p is '1'. If all the i/ps are different then o/p is '0'.

Ex-NOR = Ex-OR + NOT.

Combinational Circuits:-

Digital logic circuits are basically categorized into two types.

1. Combinational circuits
2. Sequential circuits

A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs. At any given time, the binary values of the outputs are a function of the binary combination of inputs.

Half-Adder : Half-adder is a combinational circuit which adds two binary digits namely augend and addend and produces the result as carry and sum.

Truth table of HA

S.NO	x	y	C	S
0	0	0	0	0
1	0	1	0	1
2	1	0	0	1
3	1	1	1	0

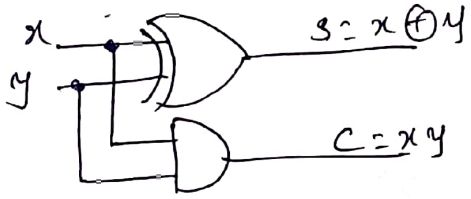
Boolean Expressions

$C = xy$

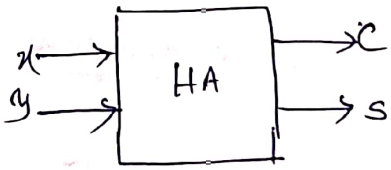
$S = \bar{x}y + x\bar{y}$

$= x \oplus y$

Implementation of Half Adder



Block diagram of Half Adder



Full-Adder : Full-Adder is a combination circuit which adds three binary digits namely augend, addened and previous carry and produces the result as carry and sum.

Truth Table of FA

S.No	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Boolean Expressions of FA

$S = \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xyz$

$= \bar{x}(\bar{y}z + y\bar{z}) + x(\bar{y}\bar{z} + yz)$

$= \bar{x}(y \oplus z) + x(y \odot z)$

$= \bar{x}(y \oplus z) + x(\overline{y \oplus z})$

Let $y \oplus z = A$

$= \bar{x}A + x\bar{A}$

$= x \oplus A$

$= x \oplus (y \oplus z)$

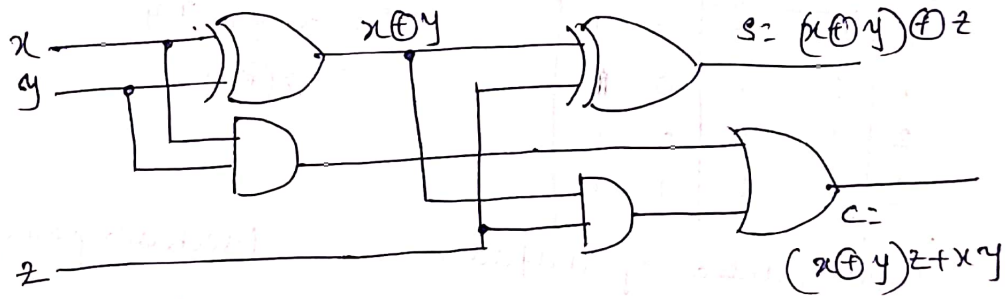
$= (x \oplus y) \oplus z$

$C = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz$

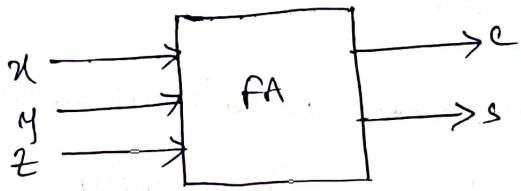
$= (\bar{x}y + x\bar{y})z + xy(\bar{z} + z)$ ($\bar{z} + z = 1$)

$= (x \oplus y)z + xy$

Implementation of Full Adder



Block diagram of FA :-



Ripple carry Adder (or) Binary Parallel Adder

A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It is a cascaded connection of full adders. A 4-bit ripple carry adder is cascaded connection of '4' full adders.

In ripple carry adder, in each section the sum output is generated only after the previous carry is produced. Thus, the sum of the most significant bit is available only after the carry signal has rippled through the adder from the least significant stage to the most significant stage. As a result, the final sum and carry bits will be available after a considerable delay. fig. below shows the 4-bit Ripple carry Adder.

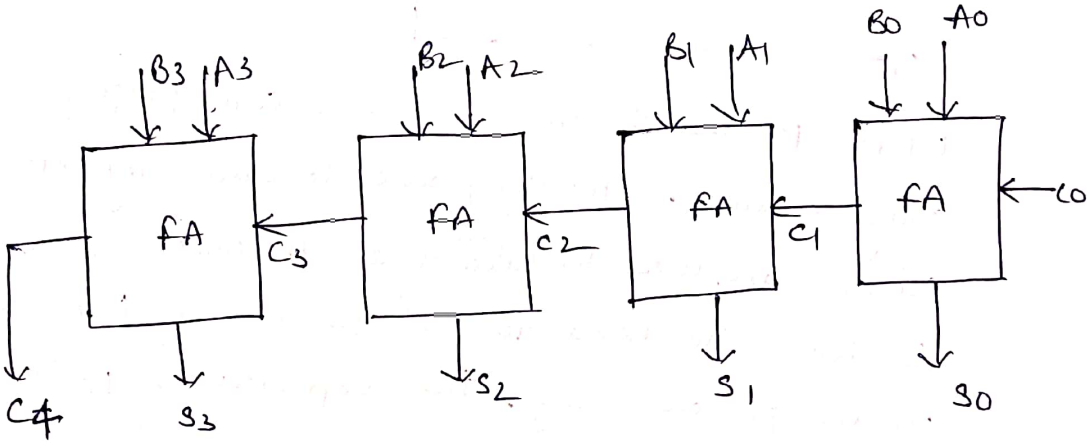


fig: 4-bit Ripple carry Adder

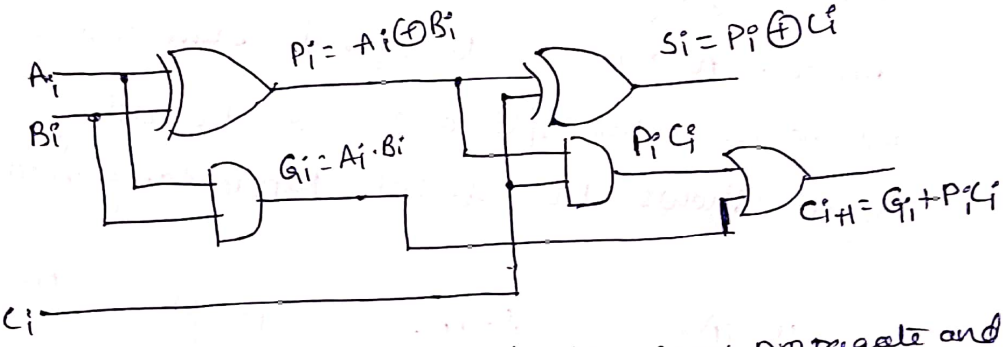
Carry lookahead Adder:-

The carry lookahead Adder (CLA) solves the carry delay problem by calculating the carry signals in advance, based on the input signals. It is based on the fact that carry signal will be generated in two cases:

1) When both bits A_i and B_i are '1',
2) when one of the two bits is '1' and carry-in is '1'.

The above two conditions are designated in the below truth table.

	A_i	B_i	C_i	C_{i+1}	S_i
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
P_i	0	1	1	1	0
P_i	1	0	0	0	1
P_i	1	0	1	1	0
G_i	1	1	0	1	0
G_i	1	1	1	1	1



where P_i and G_i are called the carry propagate and carry generate. The carry generate and carry propagate signals are used to calculate all the carries in advance. So, there is no need to wait for the carry to ripple through all the previous stages. The expressions for P_i, G_i, S_i and C_{i+1} are as follows.

$$P_i = A_i \oplus B_i$$
$$G_i = A_i \cdot B_i$$
$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

Putting $i = 0, 1, 2, 3$ in the above equation C_{i+1} we get

let $i=0$ $\quad C_1 = G_0 + P_0 C_0$

let $i=1$ $\quad C_2 = G_1 + P_1 C_1$

$G_1 + P_1 (G_0 + P_0 C_0)$

$G_1 + P_1 G_0 + P_1 P_0 C_0$

let $i=2$ $\quad C_3 = G_2 + P_2 C_2$

$G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0)$

$G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

let $i=3$ $\quad C_4 = G_3 + P_3 C_3$

$G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0)$

$G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$

4-bit Carry lookahead Generator is shown in figure below.

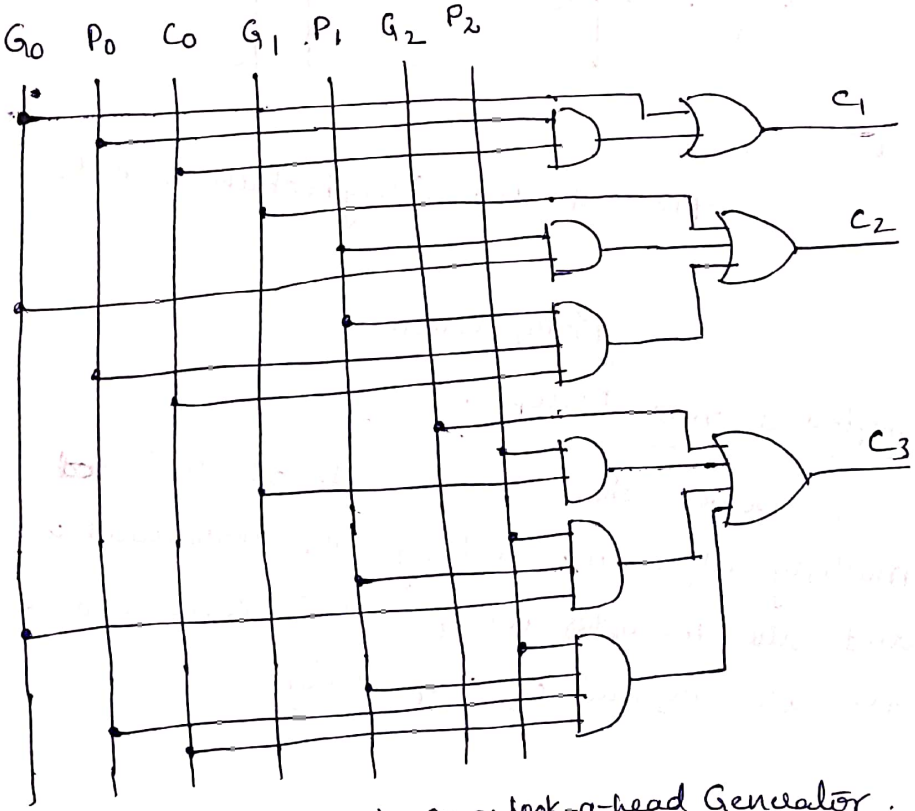


fig: 4-bit carry look-a-head Generator.

The fig. below shows Carry lookahead adder.

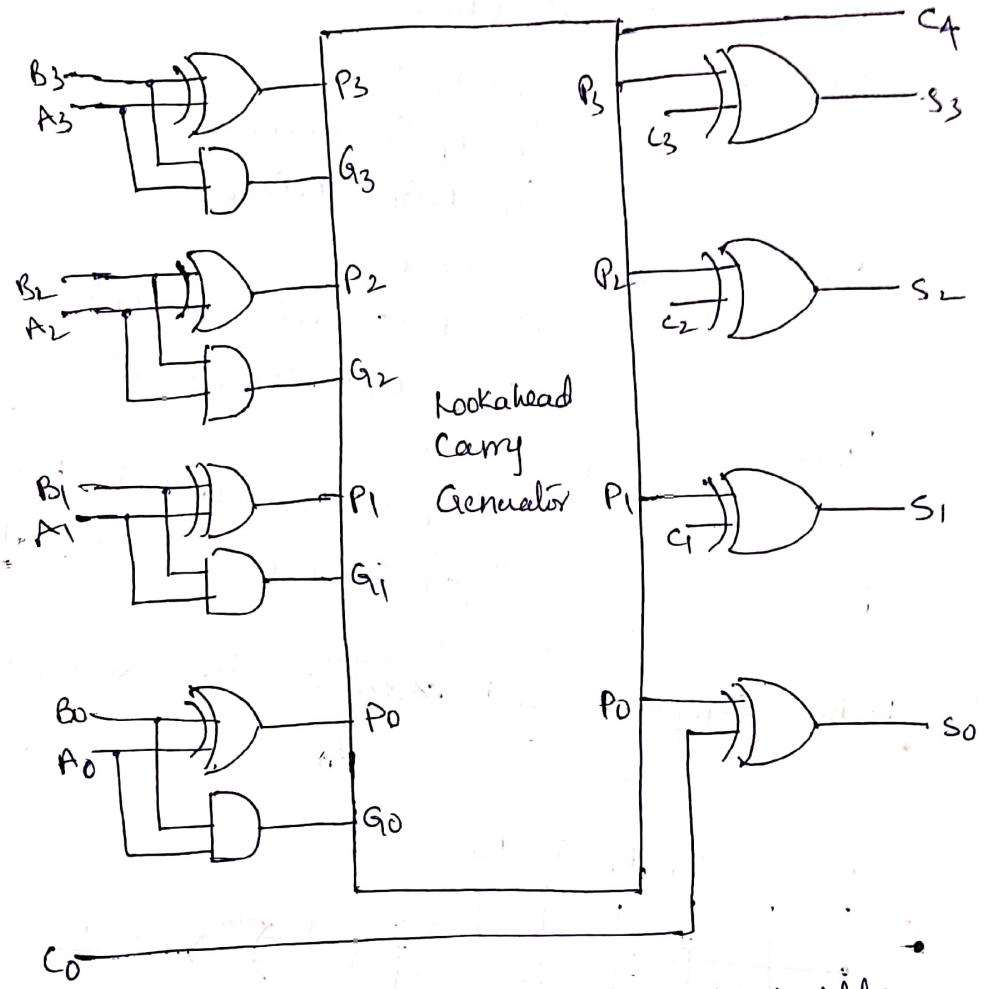


fig: 4-bit Carry Lookahead Adder

Multiplication

Shift and Add Multiplier:

figure below shows flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s respectively.

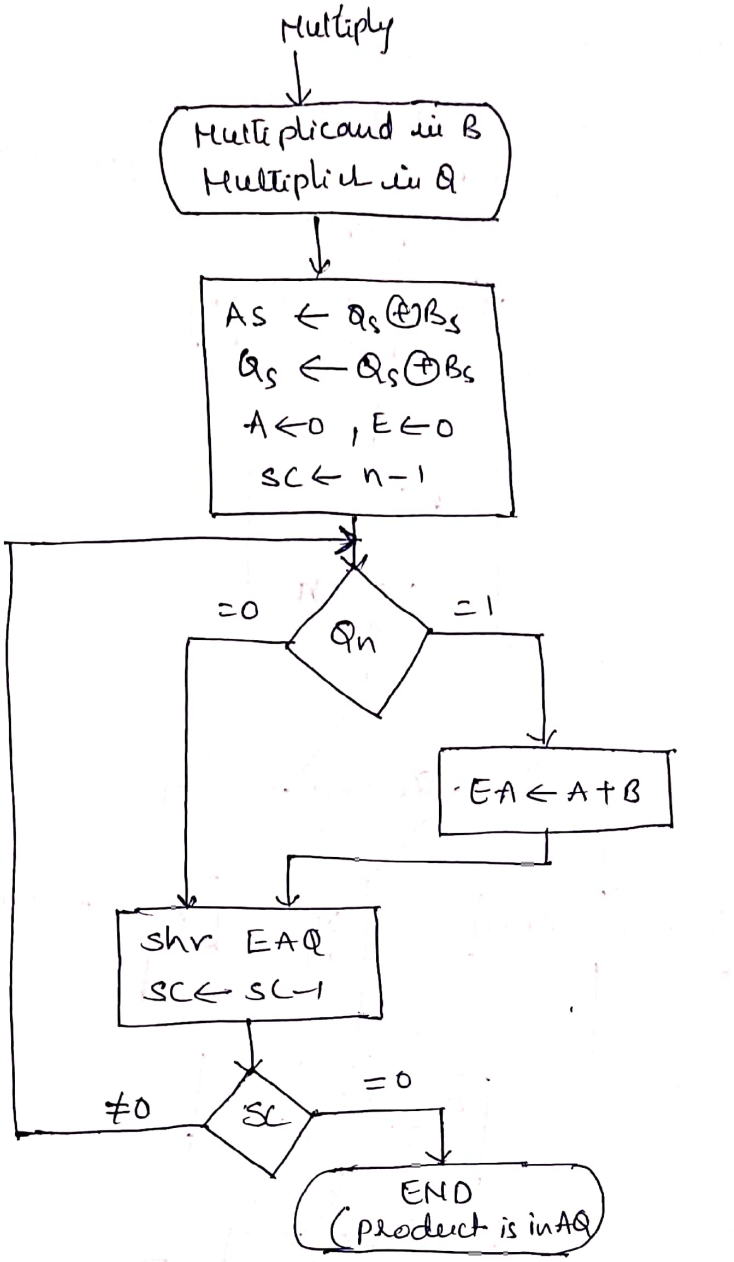


fig: flowchart for multiply operation

Multiplicand B=10111	E	A	Q	SC
Q_n=1, EA←A+B	0	00000	10011 Q_n	101
shr on EAQ, SC←SC-1		10111		
	0	10111	10011	
	0	01011	11001 Q_n	100
Q_n=1, EA←A+B		10111		
shr on EAQ, SC←SC-1	1	00010	11001 Q_n	
	0	10001	01100	011
Q_n=0, shr on EAQ	0	01000	10110 Q_n	010
SC←SC-1				
Q_n=0, shr on EAQ	0	00100	01011 Q_n	001
SC←SC-1				
Q_n=1, EA←A+B		10111		
shr on EAQ, SC←SC-1		11011	01011	
	0	01101	10101	000
	0	Product		

Booth Multiplication Algorithm:-

Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation.

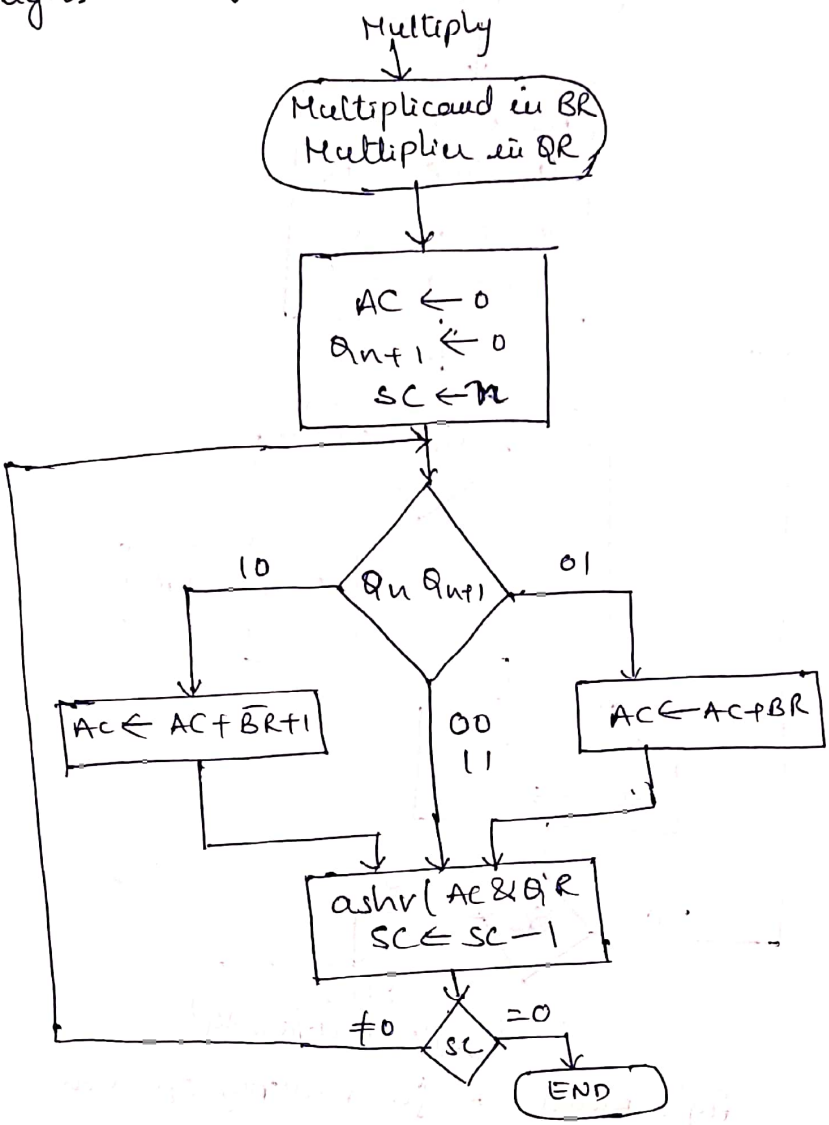


fig: Booth algorithm for multiplication of signed 2's complement numbers.

A numerical example, solved using Booth algorithm is shown below.

$(-9) \times (-13)$

sign-magnitude representation of (-9) = 11001

" " " (-13) = 11101

signed 2's complement representation of (-9) = 10111

" " " (-13) = 10011

Example of Multiplication with Booth Algorithm

Q_n Q_{n+1}	BR = 10111 $\overline{BR}+1$ = 01001	AC	QR	Q_{n+1}	SC
		00000	10011 Q_n	0	101
1 0	AC ← AC + $\overline{BR}+1$ ashr on AC & QR SC ← SC − 1	01001 01001 00100	10011 Q_n 11001 Q_n	1	100
1 1	ashr on AC & QR SC ← SC − 1	00010	01100 Q_n	1	011
0 1	AC ← AC + BR ashr on AC & QR SC ← SC − 1	10111 11001 11100	01100 10110 Q_n	0	010
0 0	ashr on AC & QR SC ← SC − 1	11110	01011 Q_n	0	001
1 0	AC ← AC + $\overline{BR}+1$ ashr on AC & QR SC ← SC − 1	01001 (1)00111 00011 Discard carry	01011 10101	1	000

Addition and subtraction :-

There are three ways of representing negative fixed-point binary numbers.

1. signed - magnitude
2. signed - 1's complement
3. signed - 2's complement

Most computers use the signed-2's complement representation when performing arithmetic operations with integers. For floating point operations, most computers use the signed-magnitude representation for the mantissa.

Addition and subtraction with signed - Magnitude Data :-

The algorithms for addition and subtraction are derived from the following table.

Addition and subtraction of Signed-Magnitude Data

Operation	Add magnitudes	Subtract magnitudes		
		When A>B	When A<B	When 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)
(-A)-(-B)				

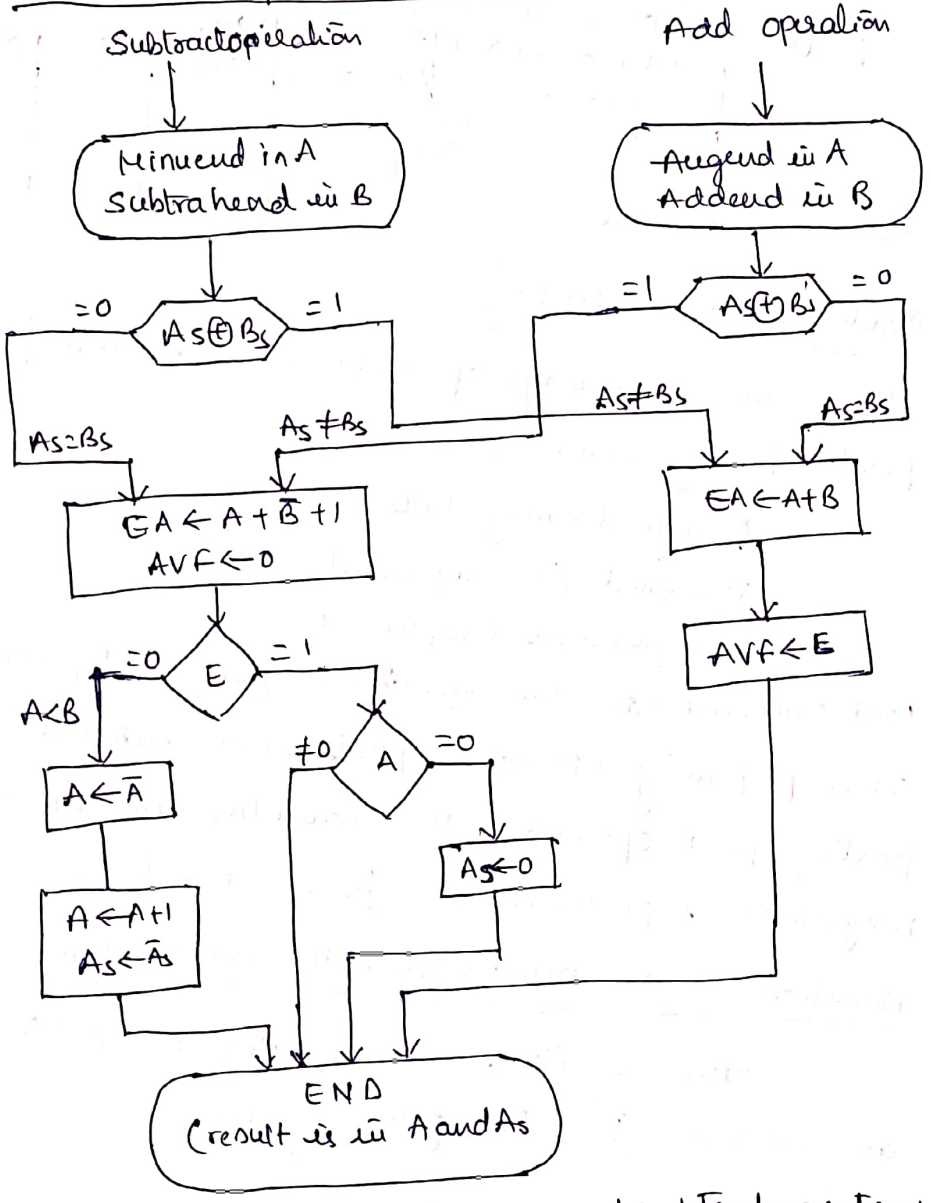


Fig: Flowchart for add and subtract operations

UNIT - II Part - I

Syllabus:-

Introduction to X86 Architecture.

Instruction set architecture of a CPU: Registers, Instruction execution cycle, RTL Interpretation of instructions, addressing modes, Instruction set.

CPU control Unit Design: Microprogrammed ~~control unit~~. design approach.

I) Introduction to X86 Architecture:-

The figure below shows the 8086 micro processor architecture. The 8086 CPU is divided into two independent functional parts.

1. BUS Interface Unit (BIU)
2. Execution Unit (EU)

Dividing the work between these two units speeds up processing

Bus Interface unit (BIU):

The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, writes data to ports and memory. In simple words, the BIU handles all transfers of data and addresses on the buses for the Execution unit.

Execution Unit (EU):-

The EU of the 8086 tells the BIU where to

fetch instructions or data from, decodes instructions and execute instructions. The EU contains control system which directs all the internal operations. A decoder in the EU translates instructions fetched from memory into a series of actions, which the EU carries out.

The EU has 16-bit arithmetic logic unit (ALU) which can perform all arithmetic, logic and shift operations also.

The main functions of EU are:

1. Decoding of instructions
2. Execution of instructions

8086 has pipelining Architecture:-

fetching the next instruction while the current instruction executes is called pipelining. While the EU is decoding an instruction or executing an instruction, which does not require use of the buses, the BIU fetches upto six instruction bytes, and places them on 6-byte instruction prefetch queue which works on the principle FIFO.

When the EU is ready to take next instruction it just grabs the instruction from instruction prefetch queue.

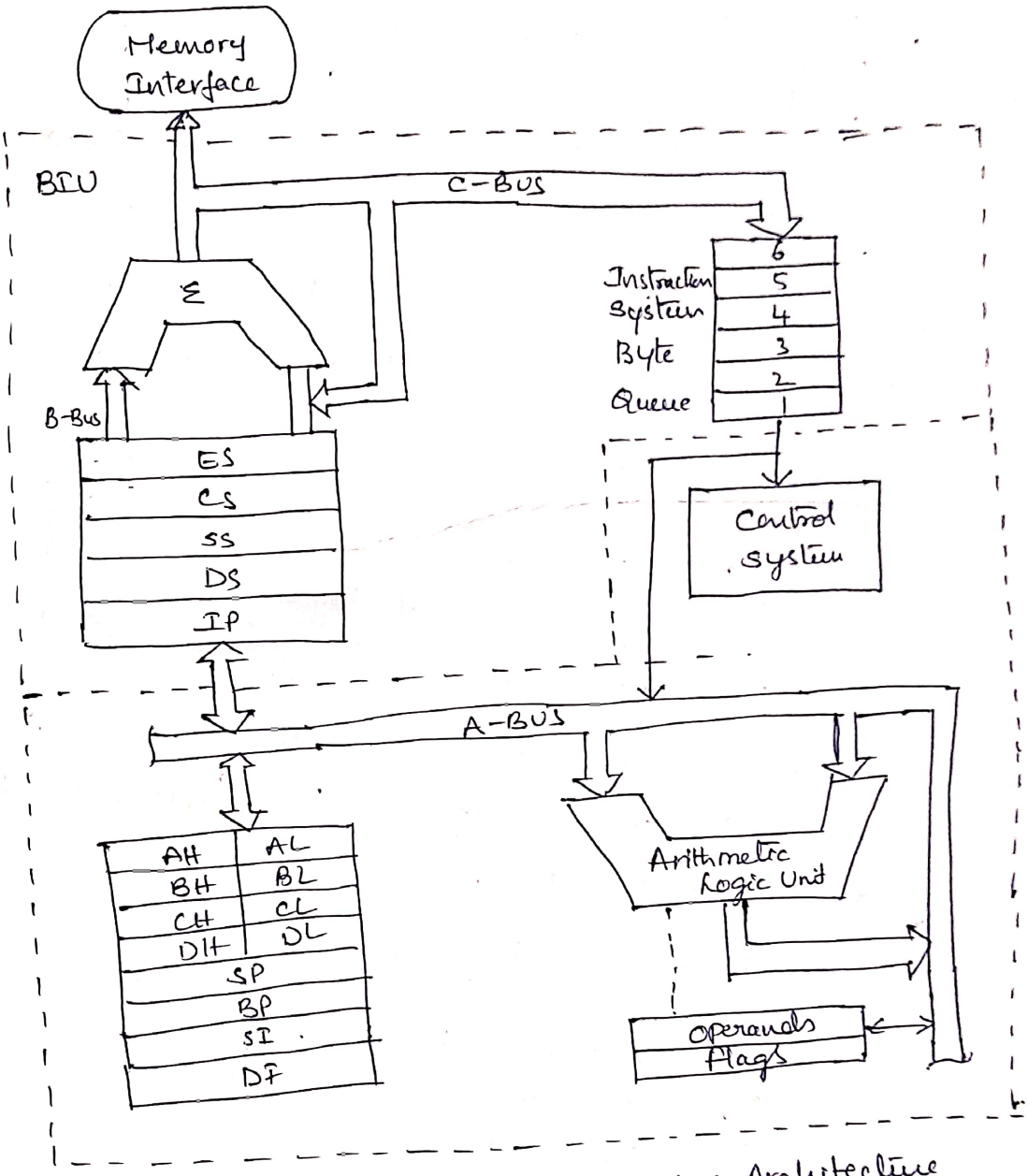


fig: 8086 Microprocessor Architecture

Register Organization

8086 microprocessor has 14, 16-bit registers. Some registers are general purpose registers and some are special purpose registers. four general purpose registers are present in 8086. They are Ax, Bx, Cx and Dx registers. These registers can be used as 8-bit registers or 16 bit registers.

AX	AH	AL	Accumulator
BX	BH	BL	Base Register (Offset register w.r.t DS)
CX	CH	CL	counter register
DX	DH	DL	Data Register

EAX
EBX
ECX
EDX

Even though the above registers are general purpose registers. Every register is having some special purpose.

Segment Registers

CS
DS
ES
SS

Code segment Register
Data segment Register
Extra segment Register
Stack segment Register

CS
SS
DS
ES
FS
GS

All the segment registers are used to store starting address of respective segment. (Upper order 16-bits of 20-bit address).

Pointer and Indexp Registers:-

IP
SP
BP
SI
DI

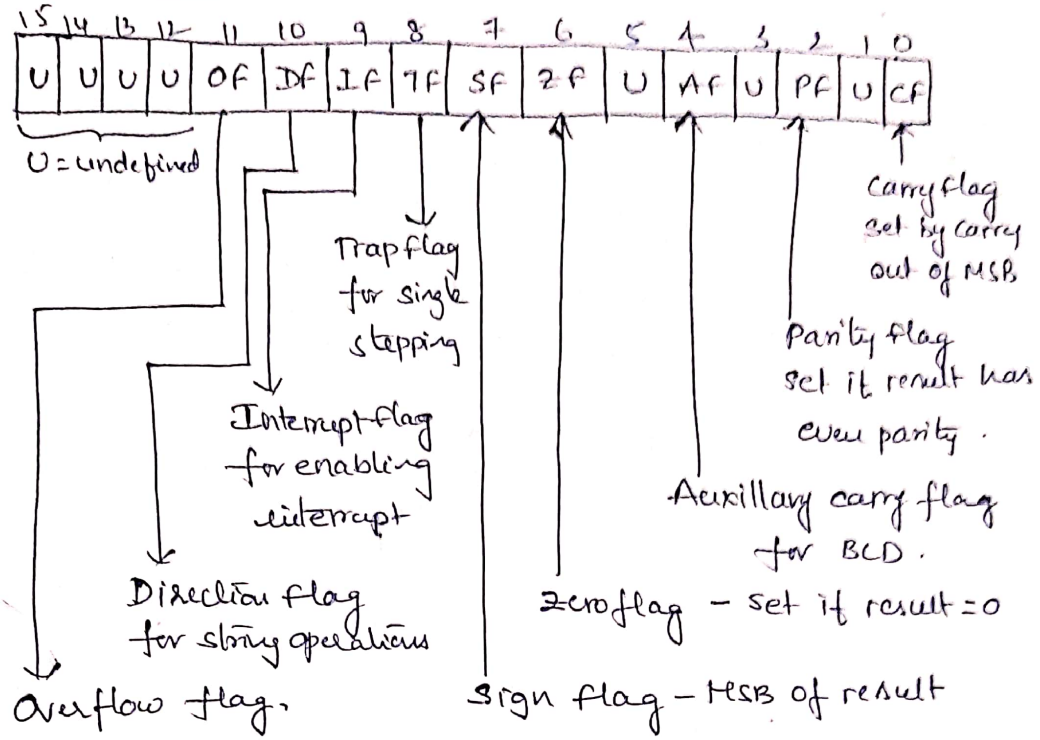
Instruction Pointer Reg.
Stack pointer Reg.
Base pointer Reg.
Source Indexp Reg.
Destination Indexp Reg.

IP:- This register acts as an offset register w.r.t to CS. offset means distance from the base of the segment.

SP & BP :- These two registers acts as offset registers w.r.t to SS.

SI & DI :- These two registers acts as offset regs. w.r.t to Extra segment.

Flag Register :-



Memory Segmentation

The memory in 8086 based system is organized as segmented memory. The CPU 8086 is able to access 1MB of physical memory. The memory is divided into no of logical segment and each segment is having the capacity of 64KB. Segments can be overlapped one upon the other.

The 8086 microprocessor is having 20-bit addrss bus and 16-bit Data bus. With 20-bit addrss bus the CPU can address 2^{20} memory locations means 1MB (1,048,576) locations.

To write an assembly language program for 8086 we require 4 different segments. They are codesegment, Data segment, Stack segment and extra segment.

To address any memory location from any of the segments we need 20-bit address. But the segment registers can store only upper 16-bits of address. So, to calculate the 20-bit physical address of any memory location we take the help of segment register and respective offset register. The physical address calculation is done by physical address generator.

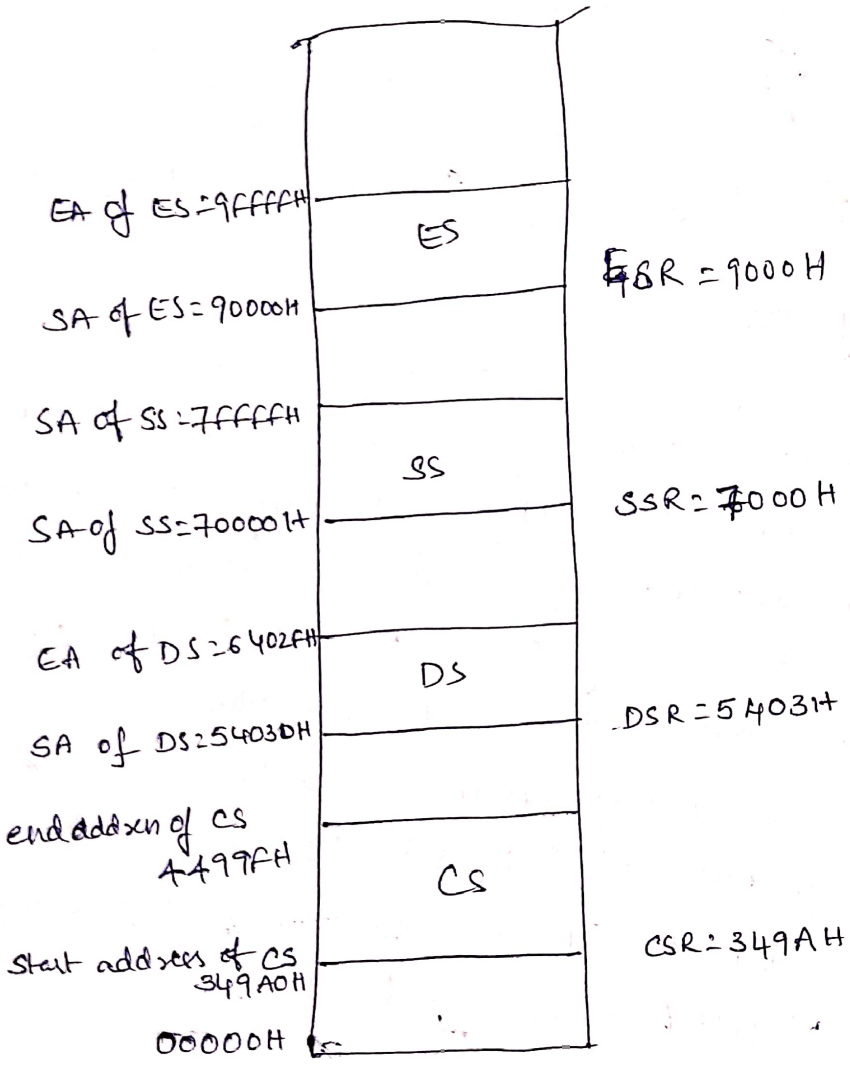


fig: 8086 memory segmentation

Physical Address = Segment Register content * 10H + offset Reg. content

UNIT - II (PART - 2)

II Instruction set architecture of a CPU:

Registers:- In a basic computer the following list of Registers are available.

Register Symbol & Number of bits	Register Name	Function
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

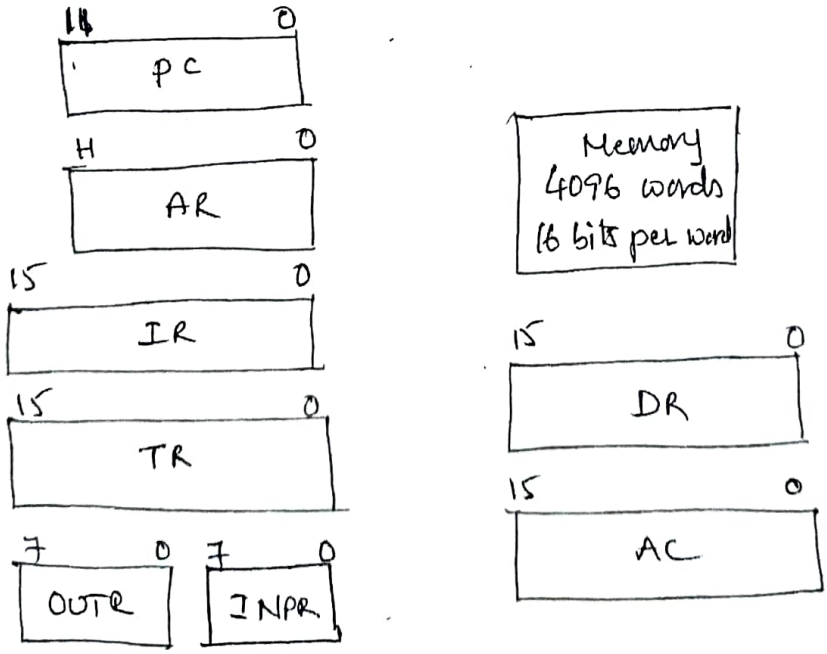


fig: Basic Computer Registers and Memory.

Instruction Cycle :-

A program residing in the memory unit of a computer consists of sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases.

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

1. Fetch an instruction from memory
2. Decode the instruction
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode and execute next instruction.

fetch and Decode :-

T_0 : $AR \leftarrow PC$

T_1 : $IR \leftarrow M[AR]$, $PC \leftarrow PC+1$

T_2 : $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{-}14)$,
$AR \leftarrow IR(0\text{-}11)$, $I \leftarrow IR(15)$

The above three lines shows the RTL (Register Transfer language) for fetch and Decode cycles.

In time period T_0, the address of PC is loaded into AR. (Initially PC contains starting address of program)

Initially sequence counter (SC) also '0'. After each clock pulse, SC is incremented by one so that the timing signals go through a sequence T_0, T_1, T_2 and so on.

In T_1 time period M[AR] is transferred to IR (Instruction Register) and PC is incremented by '1'. In T_0 and T_1 time periods fetch cycle will be completed.

In T_2 time period Decode cycle will be completed.

T_2: $D_0, \ldots, D_7 \leftarrow$ Decode IR(12-14), AR $\leftarrow$ IR(0-11), I $\leftarrow$ IR(15)

The diagram for the fetch phase is shown below.

⇒ Since only AR is connected to the address inputs of memory it is necessary to transfer the address from PC to AR during the clock associated with timing signal 'T0'.

⇒ The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T_1. At the same time PC is incremented by '1' to prepare it for the address of the next instruction in the program.

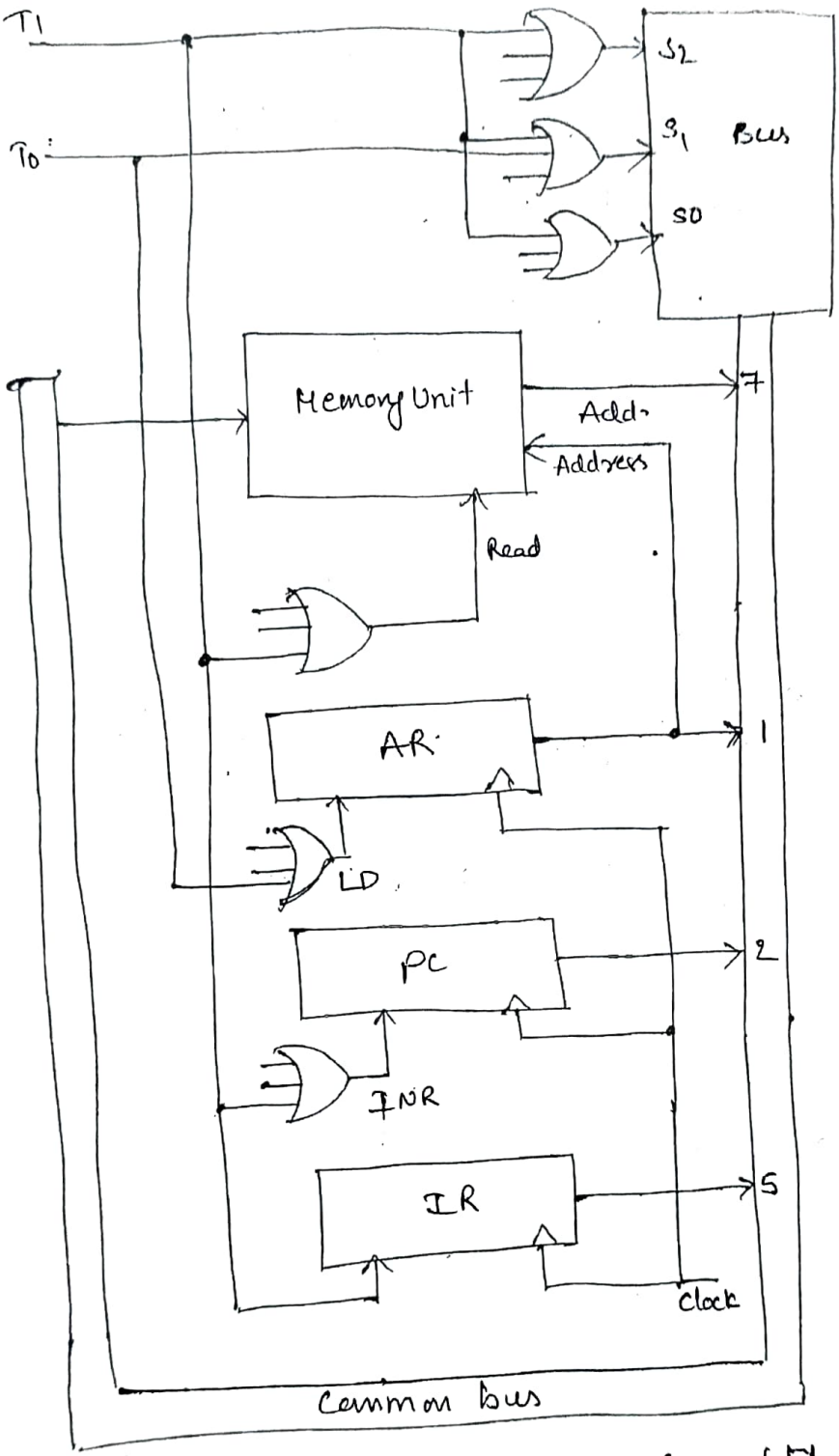


fig : Register transfers for the fetch phase

During T0 : T0 : AR ← PC

(1) Place the content of PC onto the bus by making the bus selection inputs $S_2 S_1 S_0$ equal to 000.

2. Transfer the content of the bus to AR by enabling the LD input of AR.

During T_1

T_1 : $IR \leftarrow M[AR]$, $PC \leftarrow PC+1$

1. Enable the Read input of memory.
2. Place the content of memory on to the bus by making $S_2 S_1 S_0 = 111$
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

Instruction cycle

The fig 2. below shows the instruction cycle.

→ In the instruction cycle the first two time periods 'T_0' and 'T_1' are for fetch cycle.

→ The 'T_2' time period is for decoding. Decoding means the computer understands the meaning of the instruction. It decides whether the instruction is memory-reference instruction, register-reference instruction or I/O instruction. In 'T_2' it self address part of IR is transferred to AR and IR(15) is transferred to I.

After decoding if $D_7 = 1$ it must be either register-reference or I/O instruction

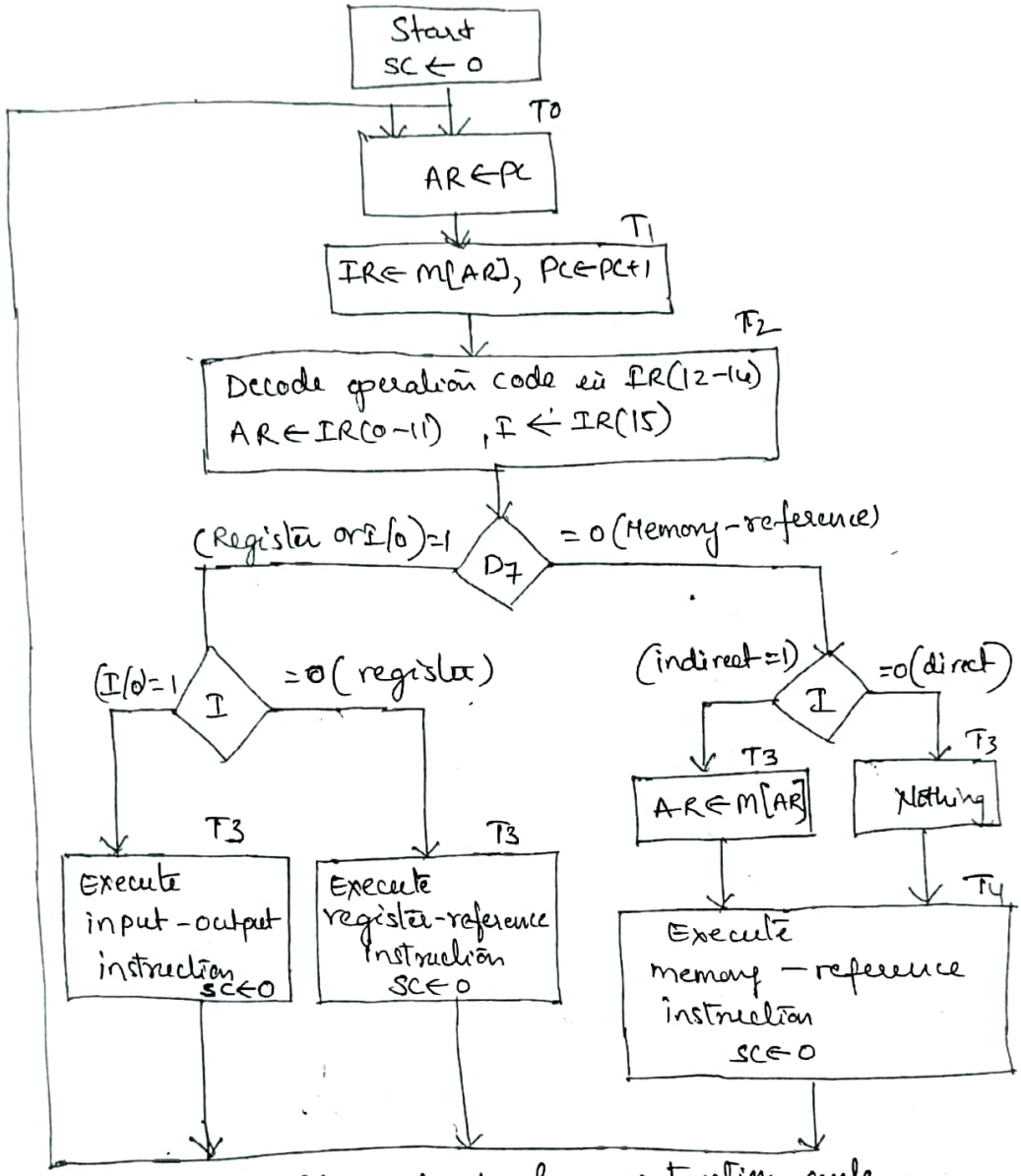


fig: flow chart for instruction cycle.

→ If $D_7 = 1$ and $I = 0$ it is considered as register-reference instruction and in T_3 execution starts.

→ If $D_7 = 1$ and $I = 1$ it is considered as I/O instruction and in T_3 execution starts.

→ If $D_7 = 0$ the instruction is considered as memory-Reference instruction.

→ If $D_7 = 0$ and $I = 0$ it is considered as Direct addressing mode memory reference instruction. Then in T_3 time period nothing will happen.

→ If $D_7 = 0$ and $I = 1$ it is considered as Indirect addressing mode memory reference instruction. Then in T_3 time period. $M[AR]$ is transferred to AR.

$$T_3 : AR \leftarrow M[AR]$$

→ In T_4 time period memory-reference instruction execution cycle starts.

→ In T_3 time period the following operations are performed

The three instructions types are subdivided into four seperate paths. the operations performed in four paths are shown below.

$D_7' I T_3$: $AR \leftarrow M[AR]$

$D_7' I' T_3$: Nothing

$D_7 I' T_3$: Execute a register reference instruction

$D_7 I T_3$: Execute a input output instruction

Some Pre-requisites to study Instruction cycle :-

① Stored Program Organization :-

In this Von-neumann architecture, Stored program organization is used where instructions

are stored in one section of memory and data in another section of memory.

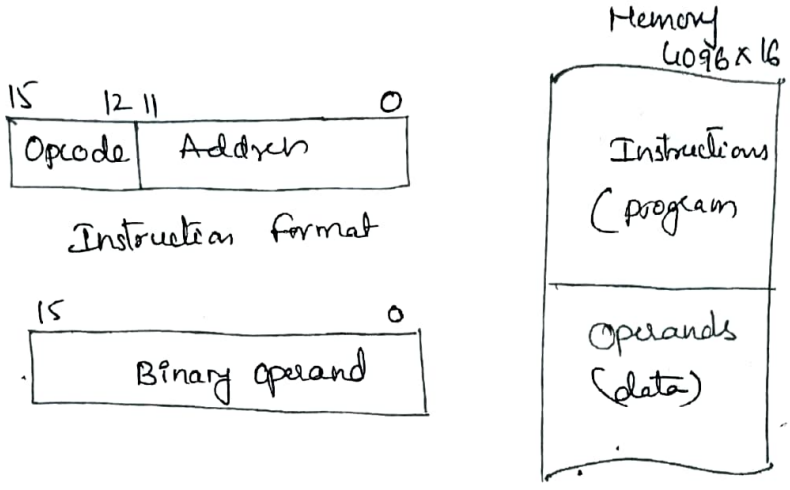


fig: stored program organization

(2) Demonstration of Direct and Indirect addressing Modes :-

Addressing Mode :- the way of specifying the address of operand is known as Addressing mode.

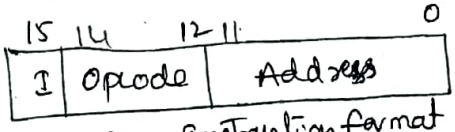


fig: Instruction format

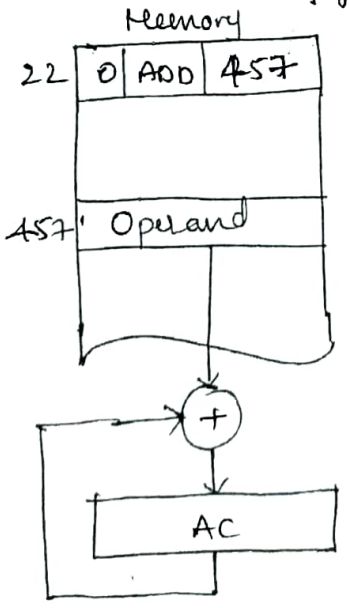


fig: Direct Address

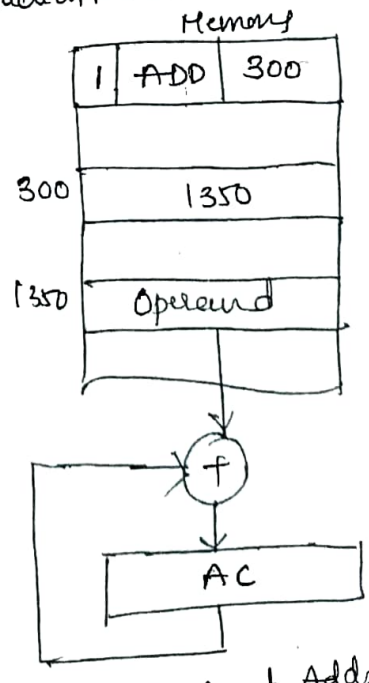
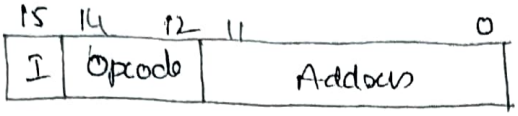


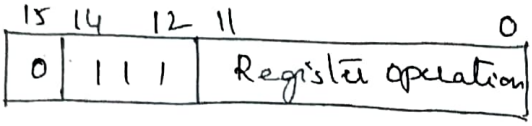
fig: Indirect Address

(B) Computer Instructions



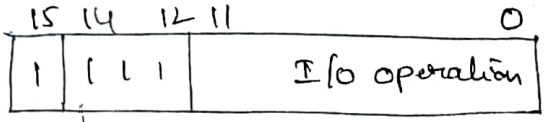
(Opcode = 000 through 110)

(a) Memory-reference instruction



(Opcode = 111, I=0)

(b) Register-reference instruction



(Opcode = 111, I=1)

(c) Input output instruction.

Symbol	Hexadecimal code I=0	I=1	Description
AND	0XXX	8XXX	AND memory word to AC
ADD	1XXX	9XXX	ADD " " " "
LDA	2XXX	AXXX	LOAD " " " "
STA	3XXX	BXXX	STORE content of AC in memory
BUN	4XXX	CXXX	Branch unconditionally
BSA	5XXX	DXXX	Branch and save returnaddress
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 " " AC negative
SZA	7004		" " " " AC zero
SZE	7002		" " " " 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

Fig: Basic Computer Instructions

RTL Interpretation of Instructions:-

→ Register - Reference Instructions

$D_7 I' T_3 = r$ (common to all register reference instructions)

$IR(i) = B_i$ (bit in IR(0-11) that specifies the operation)

Symbol	RTL	Description
	r : $SC \leftarrow 0$	Clear SC
CLA	rB_{11} : $A \leftarrow 0$	Clear AC
CLE	rB_{10} : $E \leftarrow 0$	Clear E
CMA	rB_9 : $AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8 : $E \leftarrow \bar{E}$	Complement E
CIR	rB_7 : $AC \leftarrow shr AC$, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	circular right
CIL	rB_6 : $AC \leftarrow shl AC$, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	circular 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.

The above table represents RTL interpretation of register reference instructions.

Memory-Reference Instructions:-

RTL interpretation of all Memory-reference instructions is shown below.

AND to AC:- D_0T_4 : $DR \leftarrow M[AR]$

D_0T_5 : $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

ADD to AC :-

D_1T_4 : $DR \leftarrow M[AR]$

D_1T_5 : $AC \leftarrow AC + DR$, $E \leftarrow Cout$, $SC \leftarrow 0$

LDA : Load to AC :-

D_2T_4 : $DR \leftarrow M[AR]$

D_2T_5 : $AC \leftarrow DR$, $SC \leftarrow 0$

STA : Store AC :-

D_3T_4 : $M[AR] \leftarrow AC$, $SC \leftarrow 0$

BUN : Branch Unconditionally :-

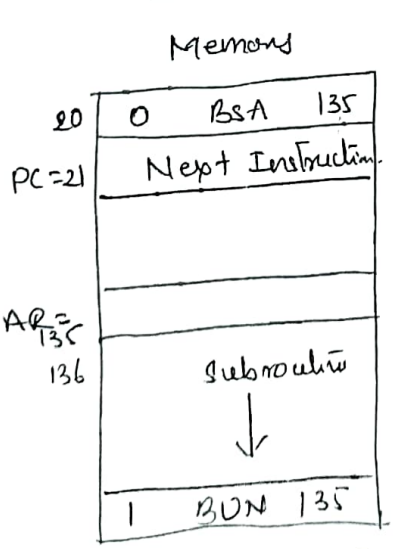
D_4T_4 : $PC \leftarrow AR$, $SC \leftarrow 0$

BSA : Branch and Save Return Address :-

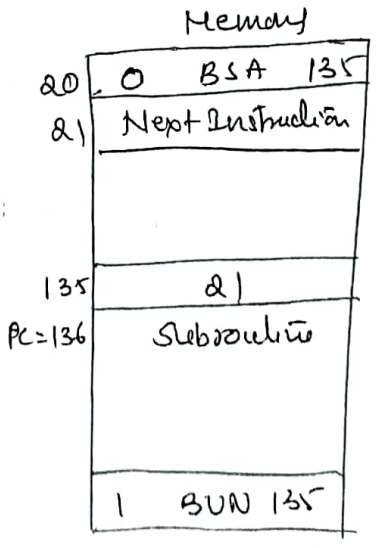
D_5T_4 : $M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$

D_5T_5 : $PC \leftarrow AR$, $SC \leftarrow 0$

Example of BSA instruction execution



(a) Memory, PC and AR at time T_4



(b) Memory and PC after execution

ISZ : Increment and skip if zero

D_6T_4 : $DR \leftarrow M[AR]$

D_6T_5 : $DR \leftarrow DR+1$

D_6T_6 : $M[AR] \leftarrow DR$, if $(DR=0)$ then $(PC \leftarrow PC+1)$, $SC \leftarrow 0$

Input-output Instructions:

The following table specifies the RTL interpretation of I/O instructions:-

$D_7IT_3 = p$ (common to all input-output instr)

$IR(i) = B_i$ [bit in IR [6-11] that specifies the instruction)

Symbol	RTL	Description
	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 if 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.

Addressing Modes :-

The way of specifying the address of the operand is known as addressing mode.

1. Implied Mode :- In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement Accumulator" is an implied mode instruction because the operand in the accumulator register is implied in the definition of the instruction

2. Immediate Mode :- In this mode operand is specified in the instruction itself. That means an immediate mode instruction has an operand field, rather an address field.

3. Register Mode :- In this mode the operands are in registers that reside within the CPU.

4. Register Indirect Mode :- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.

5. Auto increment or Auto decrement mode :- This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is

used to access memory.

6. Direct Address Mode:- In this mode the effective address is equal to the address part of the instruction.

7. Indirect Address Mode:- In this mode the address field of the instruction gives the address where the effective address is stored in memory. (i.e.) address of address of operand.

8. Relative address Mode:- In this mode the Content of the program counter is added to the address part of the instruction in order to obtain the effective address.

9. Indexed Addressing Mode:- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address

10. Base register addressing mode:- In this addressing mode the content of the base register is added to the address part of the instruction to obtain the effective address.

A Numerical example is shown below.

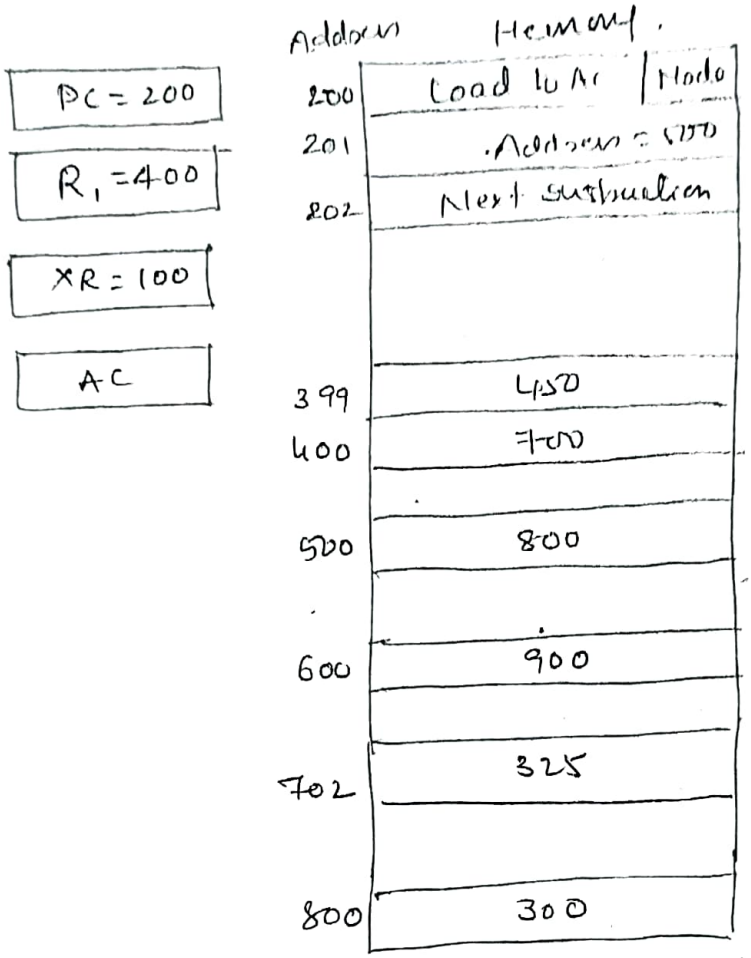


fig: Numerical example for Addressing Modes.

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
Auto increment	400	700
Auto Decrement	399	450

fig: Tabular list of Numerical Example.

Instruction Set:-

Most computer instructions can be classified into three categories:

1. Data Transfer Instructions
2. Data Manipulation Instructions
3. Program Control instructions.

1. Data Transfer Instructions :-

Name	Mnemonic
Load	LD
store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
POP	POP

2. Data Manipulation Instructions

There are three types of Data manipulation instructions,

1. Arithmetic Instructions
2. Logical and Bit manipulation instructions
3. Shift Instructions

Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB

Name	Mnemonic
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with borrow	SUBB
Negate	NEG

Logical and Bit Manipulation Instructions :-

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

Shift Instructions :-

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

Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
skip	SKP
Call	CALL
Return	RET
Compare (by SUB)	CMP
Test (by AND)	TST

Conditional Branch Instructions

Mnemonic	Branch Condition	test condition
BZ	Branch if zero	Z=1
BNZ	Branch if not zero	Z=0
BC	Branch if carry	C=1
BNC	Branch if no carry	C=0
BP	Branch if plus	S=0
BM	Branch if minus	S=1
BV	Branch if overflow	V=1
BNV	Branch if no overflow	V=0

Unit-II (Part-2)

CPU Control unit design: Micro-programmed design approach.

The major functional parts in a digital computer are Central Processing Unit (CPU), Memory, and Input–output. The main functional units of CPU are control unit, arithmetic and logic unit, and registers. The function of the control unit in a digital computer is to initiate sequences of microoperations.

There are two types of control units. One is Hardwired and the other one is called Microprogrammed Control Unit. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be **hardwired Control Unit**. In **Microprogrammed Control Unit** the control unit initiates a series of sequential steps of microoperations. Micro operation is group of control variables (signals),

The control variables at any given time can be represented by a string of 1's and 0's called a **control word (Micro instruction)**. As such, control words can be programmed to perform various operations on the components of the system. A control unit whose binary control variables are stored in memory is called a ***microprogrammed control unit.*** Each word in control memory is called as ***microinstruction.*** The microinstruction specifies **one or more microoperations** for the system. A sequence of microinstructions constitutes a ***microprogram***.

Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM). ROM words are made permanent during the hardware production of the unit.

A memory that is part of a control unit is referred to as a ***control memory.*** The general configuration of a microprogrammed control unit is demonstrated in the block diagram shown below. The control memory is assumed to be a ROM, within which all control information is permanently stored. The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.

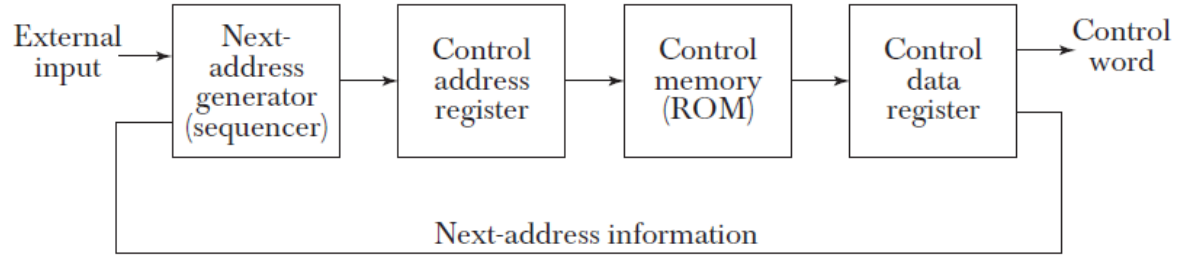


Figure Microprogrammed control organization.

The microinstruction specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason, it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions.

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. Thus, a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

Address Sequencing

The address sequencing (Next address generation) capabilities required in a control memory are:
1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Figure below shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.

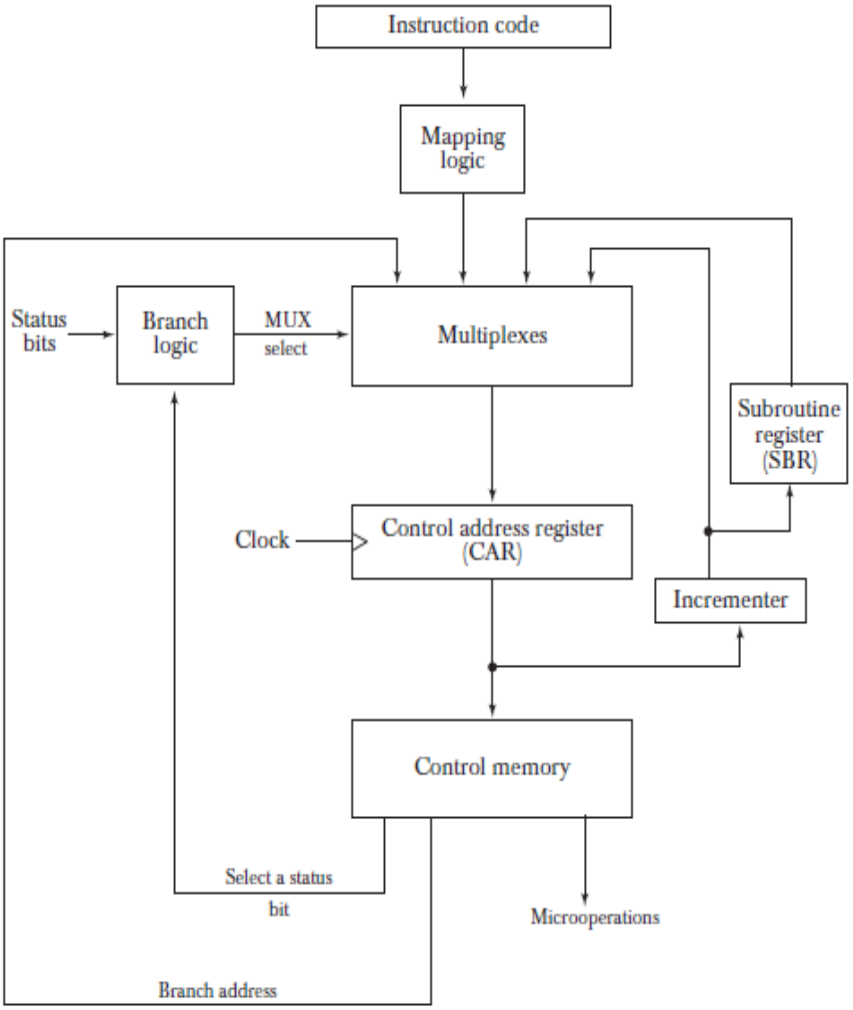
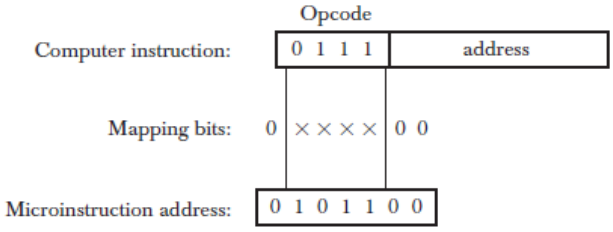


Figure Selection of address for control memory.

Mapping Process: Mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in Fig below. This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.

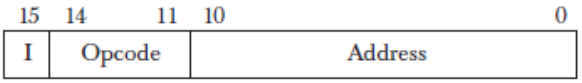
Figure Mapping from instruction code to microinstruction address.



Computer Instruction Format

The computer instruction format is depicted in Fig. below. These instructions are used to explain the Micro programmed Control Unit. It consists of three fields: a 1-bit field for indirect addressing symbolized by *I*, a 4-bit operation code (opcode), and an 11-bit address field. Figure below also shows lists four of the 16 possible memory-reference instructions. The instructions are ADD, BRANCH, STORE and EXCHANGE.

Figure Computer instructions.



(a) Instruction format

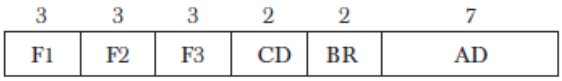
Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

Microinstruction Format

The microinstruction format for the control memory is shown in Fig. below. The 20 bits of the microinstruction are divided into four functional parts. The three fields Fl, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type or branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2^7 words.



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure Microinstruction code format (20 bits).

The CD, BR fields are shown below.

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

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

The microoperation fields F1, F2, F3 are shown in figure below.

TABLE Symbols and Binary Code for Microinstruction Fields

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	$AR \leftarrow DR(0-10)$	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	$DR(0-10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow AC$	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	

The **microprogram sequencer** for a control memory is as shown in figure below. The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register *CAR*.

The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from *CAR* provides the address for the control memory. The content of *CAR* is incremented and applied to one of the multiplexer inputs and to the subroutine register *SBR*. The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of *SBR*, and from an external source that maps the instruction.

The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer. If the bit selected is equal to 1, the *T* (test) variable is equal to 1; otherwise, it is equal to 0. The *T* value together with the two bits from the BR (branch) field go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit.

Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operation.

The input logic circuit has three inputs, *I*0, *I*1, and *T*, and three outputs, *S*0, *S*1, and *L*. Variables *S*0 and *S*1 select one of the source addresses for *CAR*. Variable *L* enables the load input in *SBR*. The binary values of the two selection variables determine the path in the multiplexer.

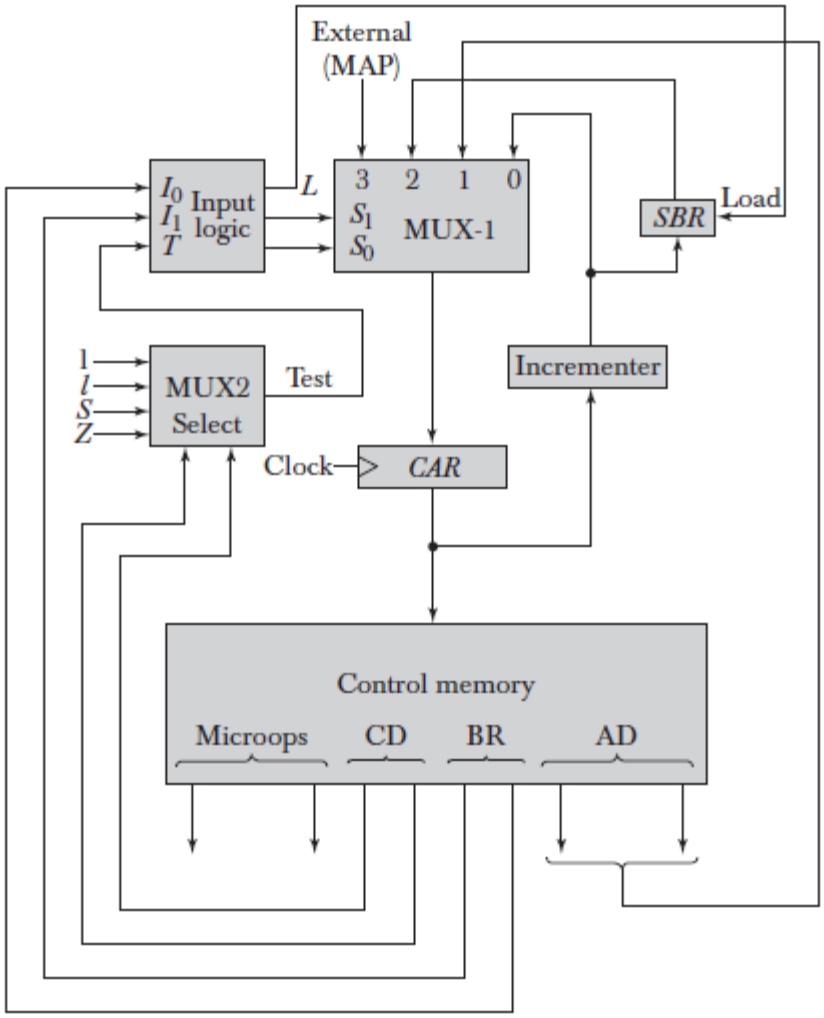


Figure Microprogram sequencer for a control memory.

UNIT – III

Memory system design: Semiconductor memory technologies, memory organization.
Memory organization: Memory interleaving, concept of hierarchical memory organization, Cache memory, cache size vs. block size, mapping functions, Replacement algorithms, write policies.

Semiconductor Memory Technologies:

Semiconductor random-access memories (RAMs) are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns. Semiconductor memory is used in any electronics assembly that uses computer processing technology. The use of semiconductor memory has grown, and the size of these memory cards has increased as the need for larger and larger amounts of storage is needed.

There are two main types or categories that can be used for semiconductor technology.

RAM - Random Access Memory: As the names suggest, the RAM or random access memory is a form of semiconductor memory technology that is used for reading and writing data in any order - in other words as it is required by the processor. It is used for such applications as the computer or processor memory where variables and other stored and are required on a random basis. Data is stored and read many times to and from this type of memory.

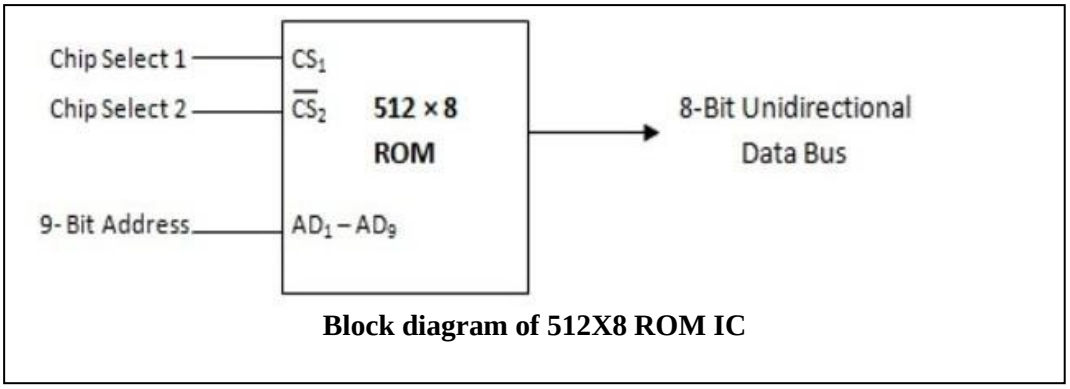
Block Diagram Representing 128 x 8 RAM
(Random Access Memory)

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

Function table

The RAM IC is in operation only when CS1 = 1 and $\overline{CS2}$ = 0. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. CS1 = 1 and $\overline{CS2}$ = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus. When chip select signal lines are not enabled and either RD or WR are not enabled then the chip will be in High-impedance state.

ROM - Read Only Memory: A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently, even when the power is removed - many memory technologies lose the data once the power is removed. As a result, this type of semiconductor memory technology is widely used for storing programs and data that must survive when a computer or processor is powered down. For example, the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM. Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.



Block diagram of 512X8 ROM IC

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1= 1 and $\overline{CS2}$ = 0 for the IC to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus, when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

The different memory types or memory technologies are detailed below:

DRAM: Dynamic RAM is a form of random access memory. DRAM uses a capacitor to store each bit of data, and the level of charge on each capacitor determines whether that bit is a logical 1 or 0.
However, these capacitors do not hold their charge indefinitely, and therefore the data needs to be refreshed periodically. As a result of this dynamic refreshing, it gains its name of being a dynamic RAM. DRAM is the form of semiconductor memory that is often used in equipment including personal computers and workstations where it forms the main RAM for the computer.

SRAM: Static Random Access Memory. This form of semiconductor memory gains its name from the fact that, unlike DRAM, the data does not need to be refreshed dynamically. It is able to support faster read and write times than DRAM (typically 10 ns against 60 ns for DRAM), and in addition its cycle time is much shorter because it does not need to pause between accesses. However, it consumes more power, is less dense and more expensive than DRAM. As a result of this it is normally used for caches, while DRAM is used as the main semiconductor memory technology.

PROM: This stands for Programmable Read Only Memory. It is a semiconductor memory which can only have data written to it once - the data written to it is permanent. These memories are bought in a blank format and they are programmed using a special PROM programmer. Typically, a PROM will consist of an array of fusible links some of which are "blown" during the programming process to provide the required data pattern.

EPROM: This is an Erasable Programmable Read Only Memory. This form of semiconductor memory can be programmed and then erased at a later time. This is normally achieved by exposing the silicon to

ultraviolet light.

EEPROM: This is an Electrically Erasable Programmable Read Only Memory. Data can be written to it and it can be erased using an electrical voltage. This is typically applied to an erase pin on the chip. Like other types of PROM, EEPROM retains the contents of the memory even when the power is turned off.

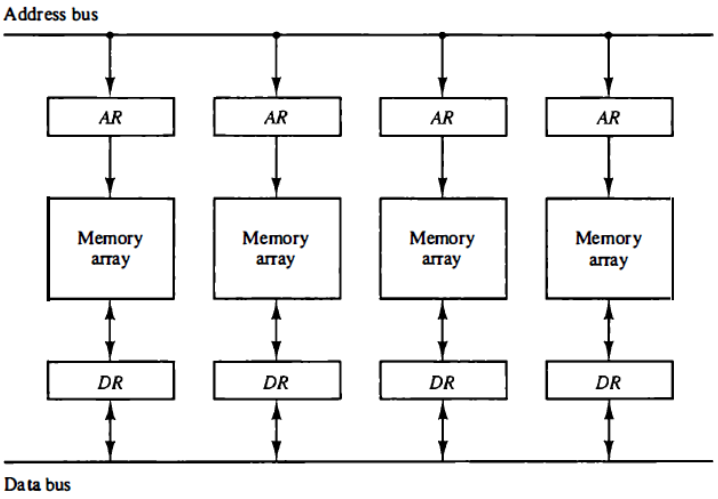
MEMORY ORGANIZATION

Memory Interleaving:

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.

Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure below shows a memory unit with four modules. Each memory array has its own address register AR and data register DR.

Figure 9-13 Multiple module memory organization.



The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

The advantage of a modular memory is that it allows the use of a technique called interleaving. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

Concept of Hierarchical Memory Organization

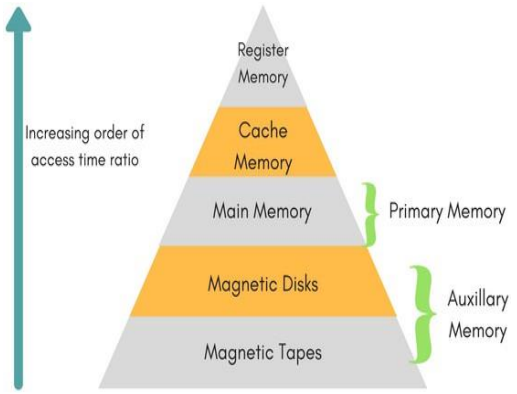
This Memory Hierarchy Design is divided into 2 main types:

External Memory or Secondary Memory

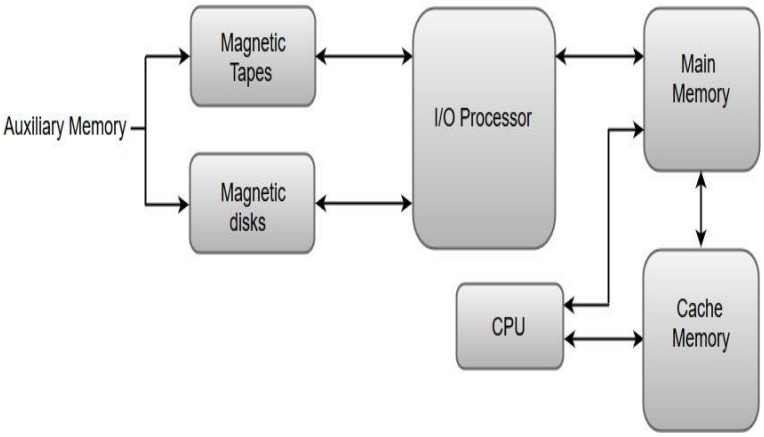
Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.

Internal Memory or Primary Memory

Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.



Memory Hierarchy in a Computer System:



Characteristics of Memory Hierarchy

Capacity:

It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.

Access Time:

It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.

Performance:

Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases.

Cost per bit:

As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Cache Memories:

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.

Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a

few procedures that repeatedly call each other.

The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache‟s *replacement algorithm*.

Cache Hits

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

Cache Misses

A Read operation for a word that is not in the cache constitutes a *Read miss*. It causes the block of words containing the requested word to be copied from the main memory into the cache.

Cache Mapping:

There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.

Direct mapping

The simplest way to determine cache locations in which to store memory blocks is the *direct-mapping* technique. In this technique, block *j* of the main memory maps onto block *j* modulo 128 of the cache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.

For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block.

When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memoryand loaded into the cache.

The direct-mapping technique is easy to implement, but it is not very flexible.

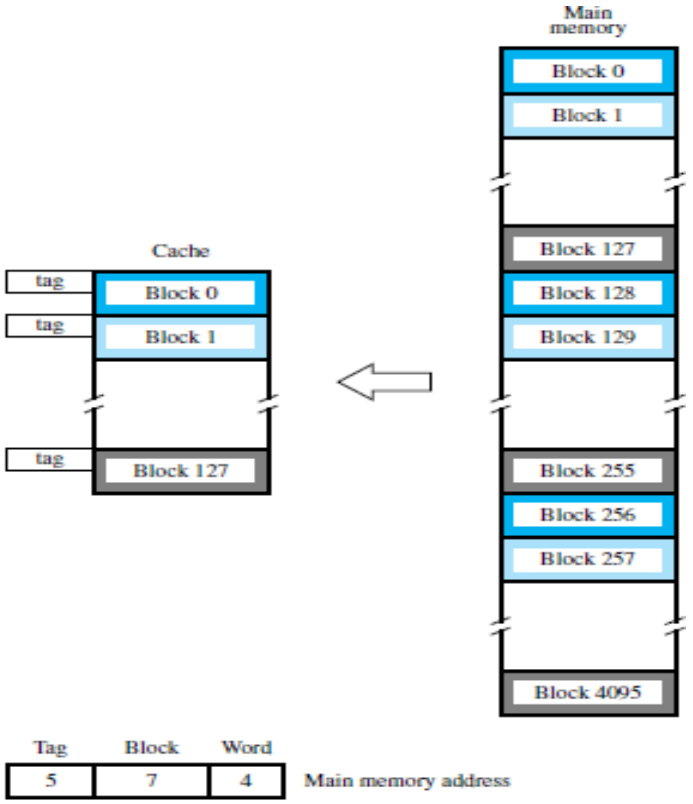


Figure 8.16 Direct-mapped cache.

Associative Mapping

In Associative mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique.

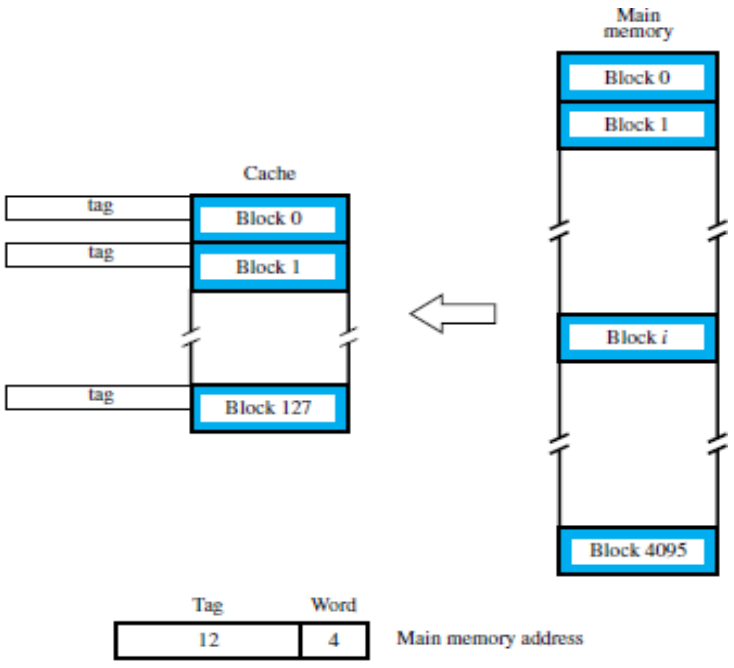


Figure 8.17 Associative-mapped cache.

It gives complete freedom in choosing the cache location in which to place the memory block,

resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced.

To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an *associative search*.

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement.

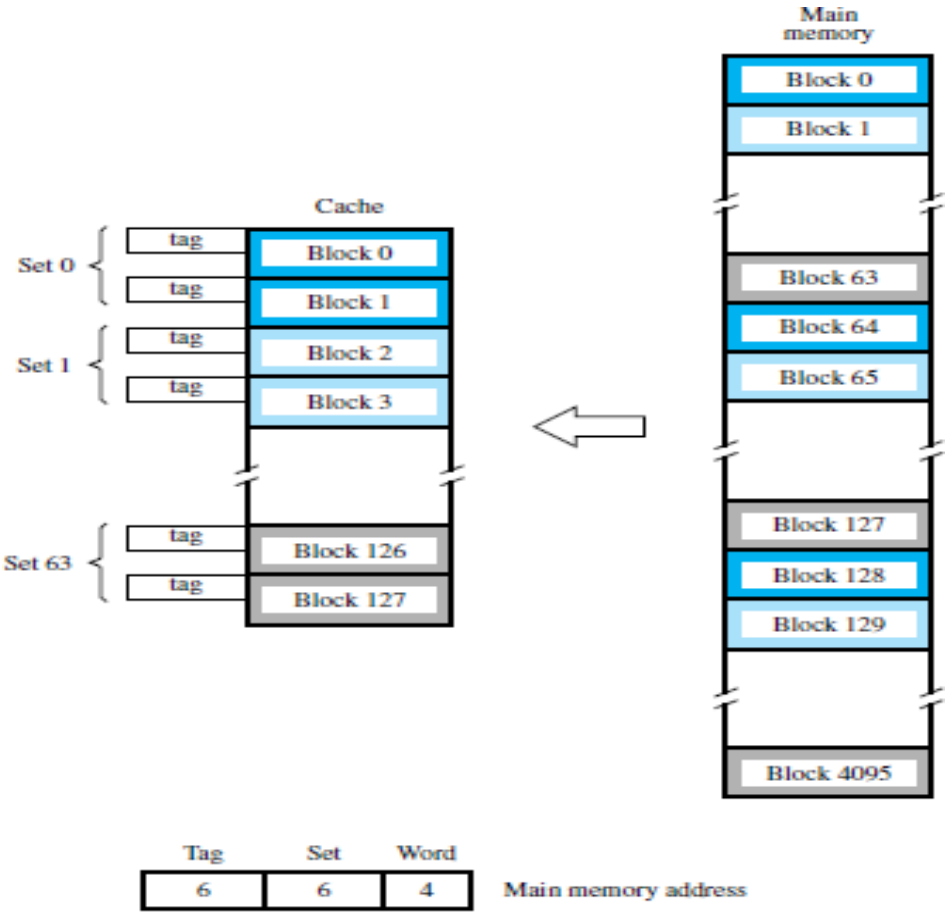


Figure 8.18 Set-associative-mapped cache with two blocks per set.

At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this *set-associative-mapping* technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set.

Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be

accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping.

Replacement Algorithms

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite.

This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced.

The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique is called the *LRU replacement algorithm*.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases.

Write Policies

The write operation is proceeding in 2 ways.

- Write-through protocol
- Write-back protocol

Write-through protocol:

Here the cache location and the main memory locations are updated simultaneously.

Write-back protocol:

- This technique is to update only the cache location and to mark it as with associated flag bit called dirty/modified bit.
- The word in the main memory will be updated later, when the block containing this marked word is to be removed from the cache to make room for a new block.
- To overcome the read miss Load –through / Early restart protocol is used.

Unit-III (Part-2)

Memory Connections to CPU:

The interconnection between memory and processor is established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a *memory address map,* is a pictorial representation of assigned address space for each chip in the system. To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.

The memory address map for this configuration is shown in Table below. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines.

The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose.

The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to $2^9 = 512$ bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

TABLE Memory Address Map for Microprocomputer

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

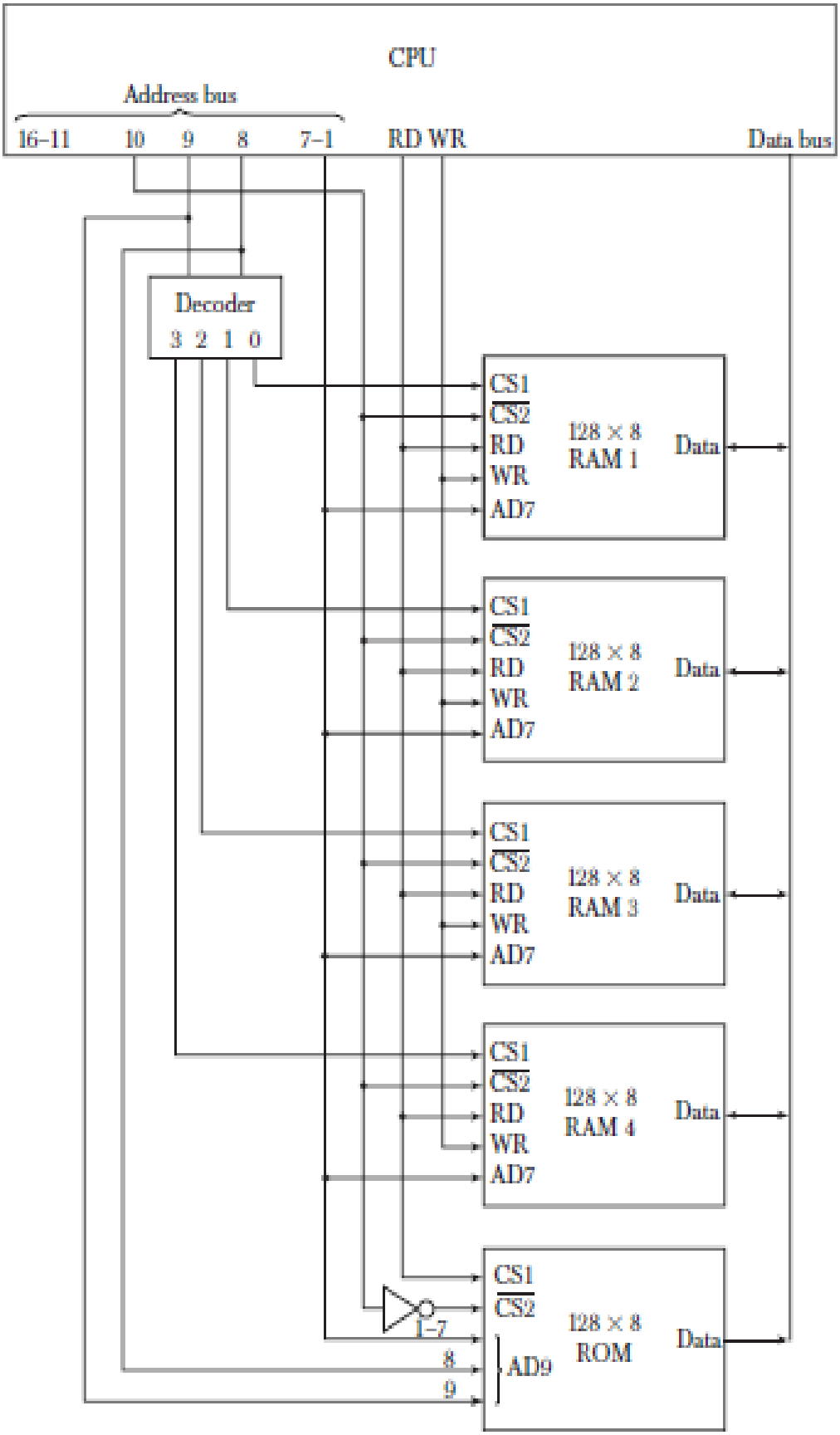


Figure Memory connection to the CPU.

UNIT – IV

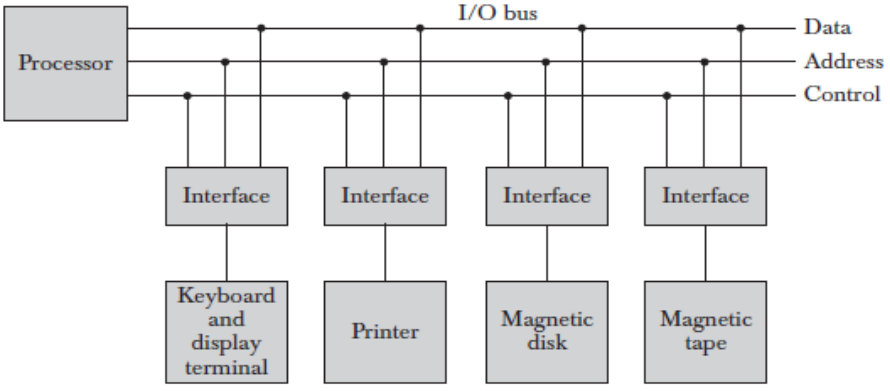
Peripheral devices and their characteristics: Input-output subsystems, I/O device interface, I/O transfers – program controlled, interrupt driven and DMA, privileged and non-privileged instructions, software interrupts and exceptions. Programs and processes – role of interrupts in process state transitions

Input-output subsystems

The Input/output organization of computer depends upon the size of computer and the peripherals connected to it. The I/O Subsystem of the computer provides an efficient mode of communication between the central system and the outside environment**.**

The most common input output devices are: Monitor, Keyboard, Mouse, Printer, Magnetic tapes Input Output Interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of communication link is to resolve the differences that exist between the central computer and each peripheral.

Figure Connection of I/O bus to input–output devices.



-

The Major Differences are: -

- Peripherals are electromechanical and electromagnetic devices and CPU and memory are electronic devices. Therefore, a conversion of signal values may be needed.
- The data transfer rate of peripherals is usually slower than the transfer rate of CPU and consequently, a synchronization mechanism may be needed.
- Data codes and formats in the peripherals differ from the word format in the CPU and memory.
- The operating modes of peripherals are different from each other and must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and Peripherals to supervises and synchronizes all input and output transfers. These components are called Interface Units because they interface between the processor bus and the peripheral devices.

I/O device interface

The I/O Bus consists of data lines, address lines and control lines. The I/O bus from the processor is attached to all peripherals interface. To communicate with a particular device, the processor places a device address on address lines. Each Interface decodes the address and control received from the I/O bus, interprets them for peripherals and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own

controller. For example, the printer controller controls the paper motion, the print timing.

There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.

A ***control command*** is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

A ***status command*** is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

A ***data output command*** causes the interface to respond by transferring data from the bus into one of its registers.

The ***data input command*** is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register.

I/O Versus Memory Bus

To communicate with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address and read/write control lines. There are 3 ways that computer buses can be used to communicate with memory and I/O:

1. Use two Separate buses, one for memory and other for I/O.
2. Use one common bus for both memory and I/O but separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

Example I/O Interface:

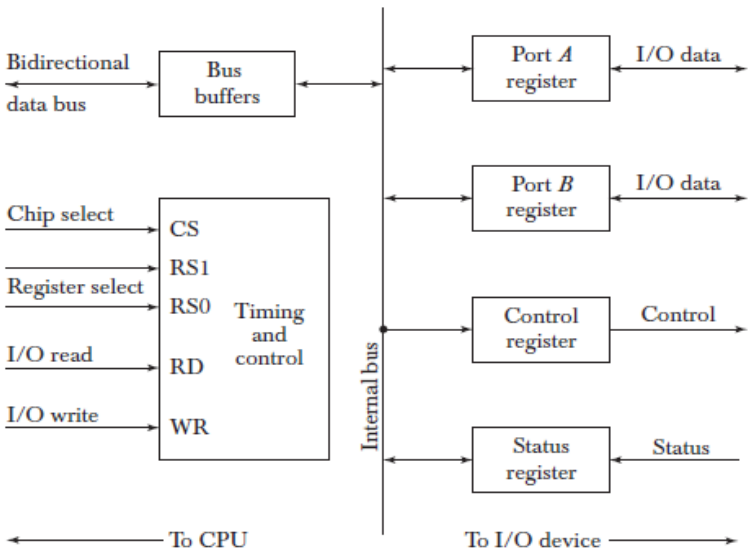


Fig: An Example I/O Interface

CS	RS1	RS0	Register selected
0	×	×	None: data bus in high–impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

An example of an I/O interface unit is shown in the above block diagram. It consists of two data registers called *ports*, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

The I/O data to and from the device can be transferred into either port *A* or port *B*. The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines.

This circuit enables the chip select (*CS*) input when the interface is selected by the address bus. The two register select inputs *RS*1 and *RS*0 are usually connected to the two least significant lines of the address bus. These two inputs select one of the four registers in the interface as specified in the table shown above. The content of the selected register is transferred into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

I/O Transfer (or) Modes of Transfer

Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

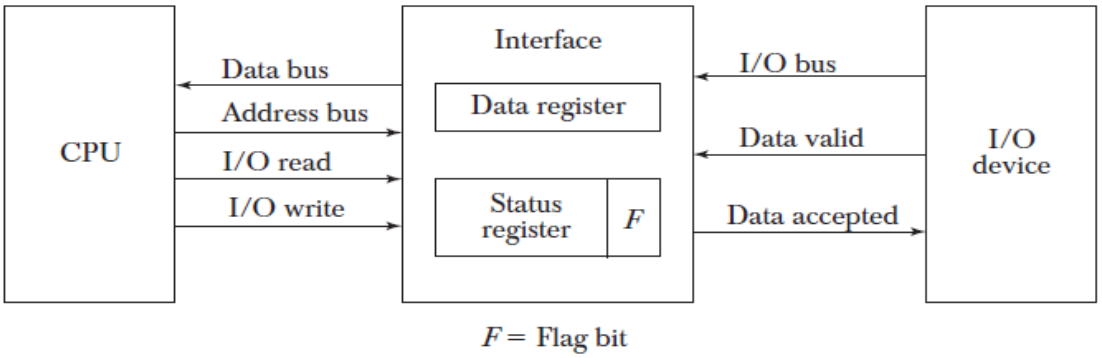
Programmed I/O Mode:

In this mode of data transfer the operations are the results in I/O instructions which is a part of computer program. Each data transfer is initiated by an instruction in the program. Normally the transfer is from a CPU register to peripheral device or vice- versa. Once the data is initiated the CPU starts monitoring the interface to see when next transfer can made. The instructions of the program keep close tabs on everything that takes place in the interface unit and the I/O devices.

The transfer of data requires three instructions:

- Read the status register.
- Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
- Read the data register.

Figure Data transfer from I/O device to CPU.



F = Flag bit

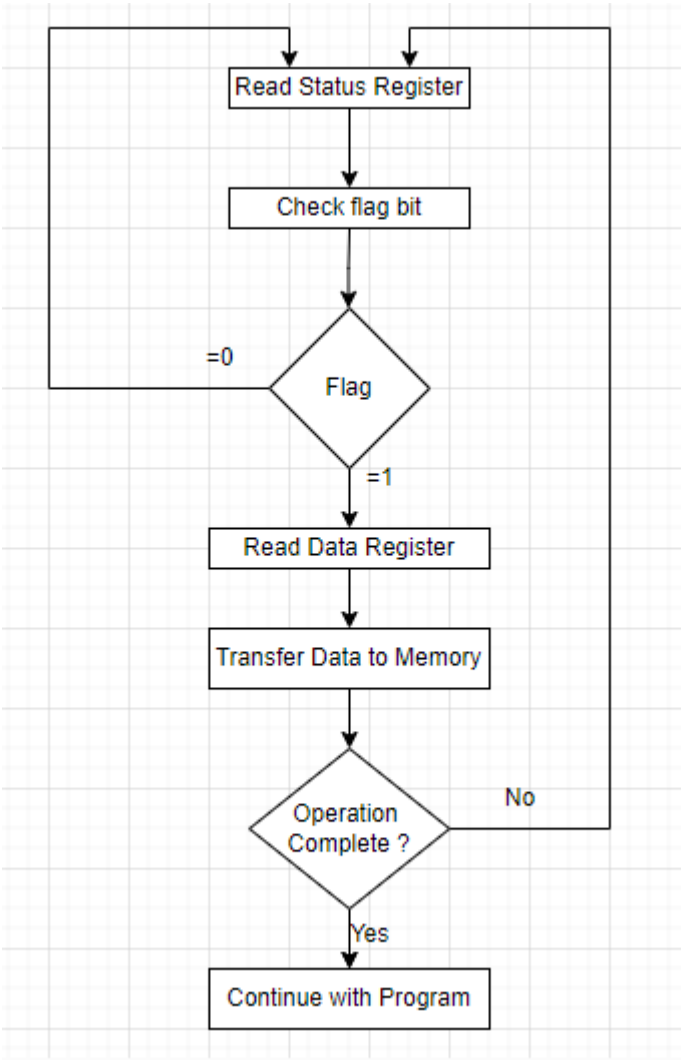


Figure: Flowchart for CPU program to input data

In this technique CPU is responsible for transferring data from the memory to output and storing data in memory for executing of Programmed I/O as shown in Fig.

Drawback of the Programmed I/O:

The main drawback of the Programmed I/O was that the CPU has to monitor the units all the times when the program is executing. Thus, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process.

Interrupt-Initiated I/O:

In this method an interrupt facility an interrupt command is used to inform the device about the start and end of transfer. In the meantime, the CPU executes another program. When the interface determines that the device is ready for data transfer it generates an Interrupt Request and sends it to the computer.

When the CPU receives such a signal, it temporarily stops the execution of the program and branches to a service program to process the I/O transfer and after completing it returns back to task, what it was originally performing.

In this type of IO, computer does not check the flag. It continues to perform its task. Whenever any device wants the attention, it sends the interrupt signal to the CPU. CPU then deviates from what it was doing, stores the return address of main program and branches to the address of the subroutine.

There are two ways of choosing the branch address:

Vectored Interrupt: In vectored interrupt the source that interrupts the CPU provides the branch information. This information is called interrupt vectored.

Non-vectored Interrupt: In non-vectored interrupt, the branch address is assigned to the fixed address in the memory.

In Interrupt initiated I/O there are two techniques called

1. Daisy Chaining Priority
2. Parallel Priority Interrupt Controller.

Diasy Chaining Priority:

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Figure below. The interrupt request line is common to all devices and forms a wired logic connection.

If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts arc pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its *PI* (priority in) input.

The acknowledge signal passes on to the next device through the *PO* (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the *PO* output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

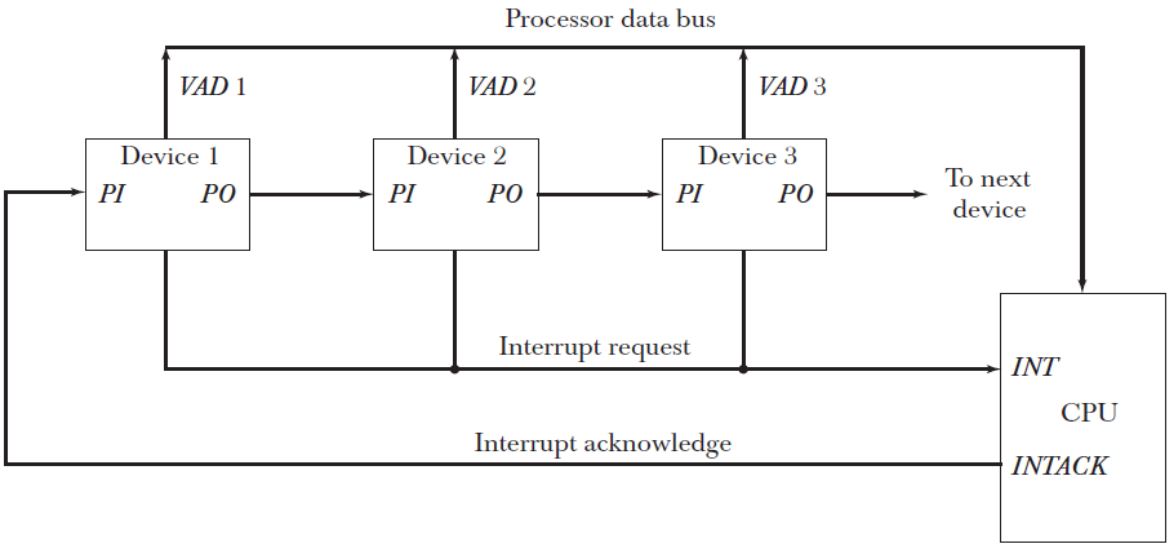


Figure Daisy-chain priority interrupt.

Parallel Priority Interrupt

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

The priority logic for a system of four interrupt sources is shown in Figure below. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

Another output from the encoder sets an interrupt status flip-flop *IST* when an interrupt that is not masked occurs. The interrupt enable flip-flop *IEN* can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of *IST* ANDed with *IEN* 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.

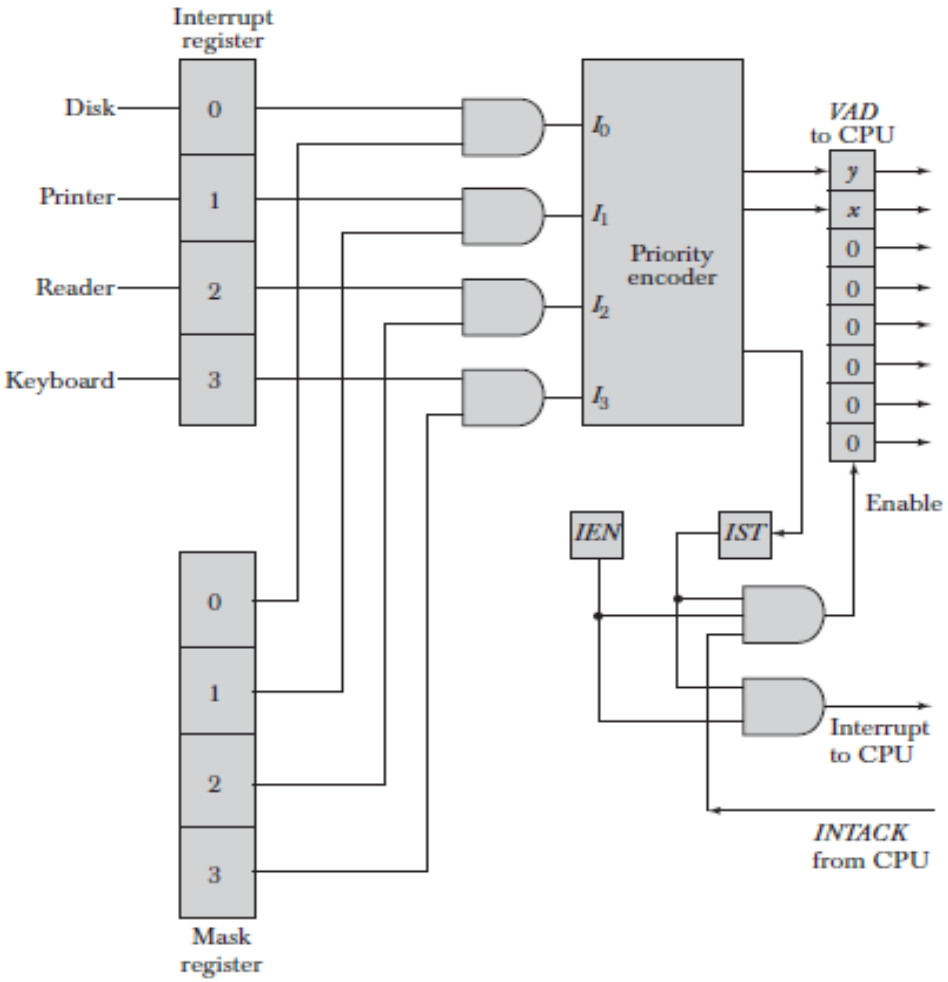


Figure Priority interrupt hardware.

Direct Memory Access (DMA):

In the Direct Memory Access (DMA) the interface transfers the data into and out of the memory unit through the memory bus. The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called Direct Memory Access (DMA). During the DMA transfer, the CPU is idle and has no control of the memory buses. A DMA Controller takes over the buses to manage the transfer directly between the I/O device and memory.

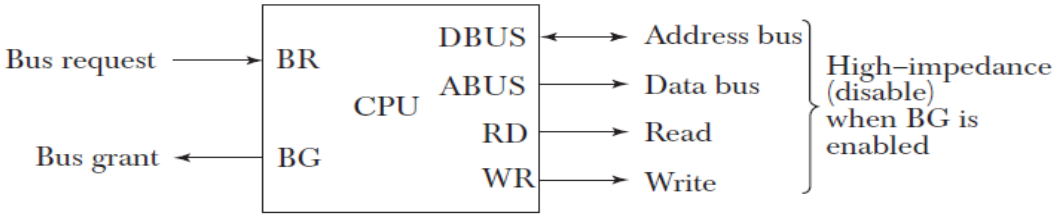
The Bus Request (BR) input is used by the DMA controller to request the CPU. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, data bus and read write lines into a high Impedance state. High Impedance state means that the output is disconnected. The CPU activates the Bus Grant (BG) output to inform the external DMA that the Bus Request (BR) can now take control of the buses to conduct memory transfer without processor. When the DMA terminates the transfer, it disables the Bus Request (BR) line. The CPU disables the Bus Grant (BG), takes control of the buses and return to its normal operation.

The transfer can be made in several ways that are:

- DMA Burst
- Cycle Stealing

DMA Burst: In DMA Burst transfer, a block sequence consisting of a number of memory words is transferred in continuous burst while the DMA controller is master of the memory buses.
Cycle Stealing: Cycle stealing allows the DMA controller to transfer one data word at a time, after which it must returns control of the buses to the CPU.

Figure CPU bus signals for DMA transfer.



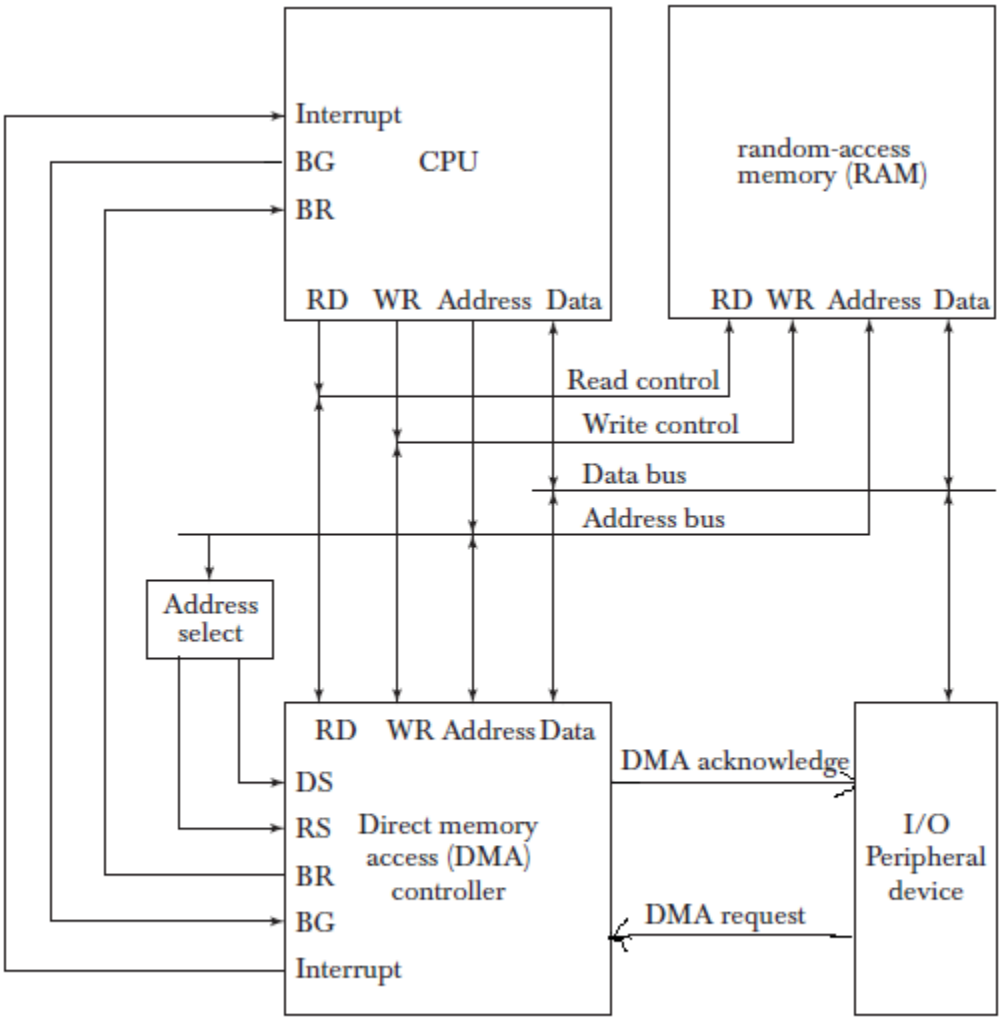
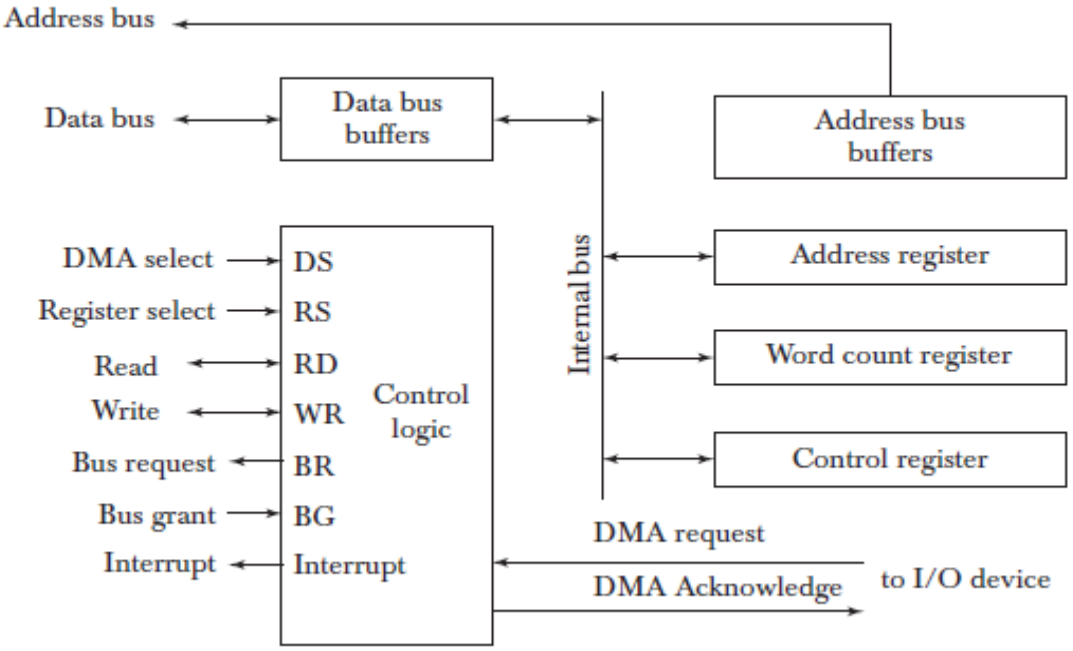


Figure DMA transfer in a computer system.

DMA Controller:
The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. The DMA controller has three registers:
Address Register: Address Register contains an address to specify the desired location in memory.
Word Count Register: WC holds the number of words to be transferred. The register is decremented

by one after each word transfer and internally tested for zero.
Control Register: Control Register specifies the mode of transfer

Figure Block diagram of DMA controller.



The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (Register select) inputs. The RD (read) and WR (write) inputs are bidirectional. When the BG (Bus Grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG =1, the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control.

The CPU initializes the DMA by sending the following information through the data bus:
1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write).
2. The word count, which is the number of words in the memory block.
3. Control to specify the mode of transfer such as read or write.
4. A control to start the DMA transfer.

Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

Privileged and Non-Privileged Instructions

In any Operating System, it is necessary to have a **Dual Mode Operation** to ensure the protection and security of the System from unauthorized users. This Dual Mode separates the User Mode from the System Mode or Kernel Mode.
In an operating system, instructions are divided into two categories: privileged and non-privileged instructions.

Privileged instructions are those that can only be executed by the operating system kernel or a privileged process, such as a device driver. These instructions typically perform operations that require direct access to hardware or

other privileged resources, such as setting up memory mappings or accessing I/O devices. Privileged instructions are executed in kernel mode, which provides unrestricted access to the system resources.

Non-privileged instructions are those that can be executed by any process, including user-level processes. These instructions are typically used for performing computations, accessing user-level resources such as files and memory, and managing process control. Non-privileged instructions are executed in user mode, which provides limited access to system resources and ensures that processes cannot interfere with one another.

Some key differences between privileged and non-privileged instructions:

1. Access to resources: Privileged instructions have direct access to system resources, while non-privileged instructions have limited access.
2. Execution mode: Privileged instructions are executed in kernel mode, while non-privileged instructions are executed in user mode.
3. Execution permissions: Privileged instructions require special permissions to execute, while non-privileged instructions do not.
4. Purpose: Privileged instructions are typically used for performing low-level system operations, while non-privileged instructions are used for general-purpose computing.
5. Risks: Because privileged instructions have access to system resources, they pose a higher risk of causing system crashes or security vulnerabilities if not used carefully. Non-privileged instructions are less risky in this regard.

In summary, privileged instructions are used by the operating system kernel and privileged processes to perform low-level system operations, while non-privileged instructions are used by user-level processes for general-purpose computing. The distinction between privileged and non-privileged instructions is an important mechanism for ensuring the security and stability of an operating system.

Software Interrupts and Exceptions

Interrupt

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an ISR (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt. The following image shows the types of interrupts.

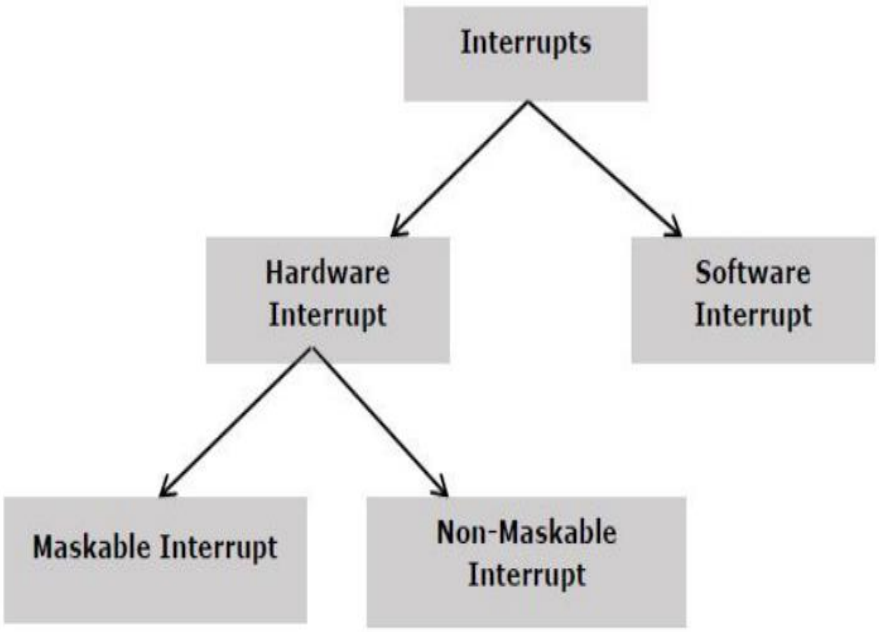


Fig: Types of Interrupts

Hardware Interrupts: Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor. (i.e., INTR)

Software Interrupts: Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to **test the working of various interrupt handlers**. It includes INT Interrupt instruction with type number It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

- TYPE 0 interrupt represents division by zero situation. (INT 0)
- TYPE 1 interrupt represents single-step execution during the debugging of a program. (INT 1)
- TYPE 2 interrupt represents non-maskable NMI interrupt. (INT 2)
- TYPE 3 interrupt represents break-point interrupt. (INT 3)
- TYPE 4 interrupt represents overflow interrupt. (INT 4)

Exception:

Exceptions occur during program execution and are so extraordinary that they cannot be handled by the program itself. If you give the processor the command to divide a number by zero, for instance, it will give a divide-by-zero exception, which will cause the computer to either stop the operation or display an error notice.

Programs and Processes

The difference between Program and Process:

Program	Process
Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
Program is a passive entity as it resides in the secondary memory.	Process is a active entity as it is created during execution and loaded into the main memory.

Program	Process
Program exists at a single place and continues to exist until it is deleted.	Process exists for a limited span of time as it gets terminated after the completion of task.
Program is a static entity.	Process is a dynamic entity.
Program does not have any resource requirement; it only requires memory space for storing the instructions.	Process has a high resource requirement, it needs resources like CPU, memory address, I/O during its lifetime.
Program does not have any control block.	Process has its own control block called Process Control Block.
Program has two logical components: code and data.	In addition to program data, a process also requires additional information required for the management and execution.

UNIT-V
Pipelining & Parallel Processors

Basic Concepts of Pipelining: Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is 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 simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the suboperation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. The pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Figure below. $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations 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

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table below. The first clock pulse transfers $A1$ and $B1$ into $R1$ and $R2$.

Figure Example or pipeline processing.

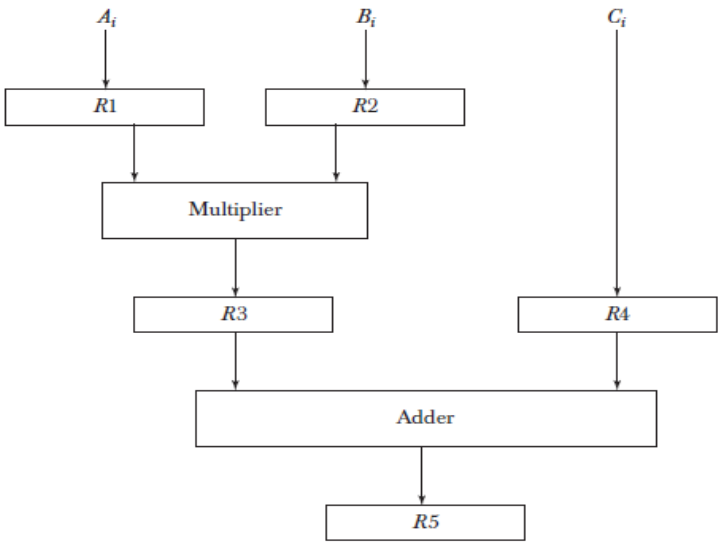


TABLE Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	–	–	–
2	A_2	B_2	$A_1 * B_1$	C_1	–
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	–	–	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	–	–	–	–	$A_7 * B_7 + C_7$

The second clock pulse transfers the product of $R1$ and $R2$ into $R3$ and $C1$ into $R4$. The same clock pulse transfers $A2$ and $B2$ into $R1$ and $R2$. The third clock pulse operates on all three segments simultaneously. It places $A3$ and $B3$ into $R1$ and $R2$, transfers the product of $R1$ and $R2$ into $R3$, transfers C2 into $R4$, and places the sum of $R3$ and $R4$ into $R5$. It takes three clock pulses to fill up the pipe and retrieve the first output from $R5$. From there on, each clock produces a new output and moves the data one step down the pipeline.

The general structure of a four-segment pipeline is illustrated in Figure below. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit *Si* that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers *Ri* that hold the intermediate results between the stages.

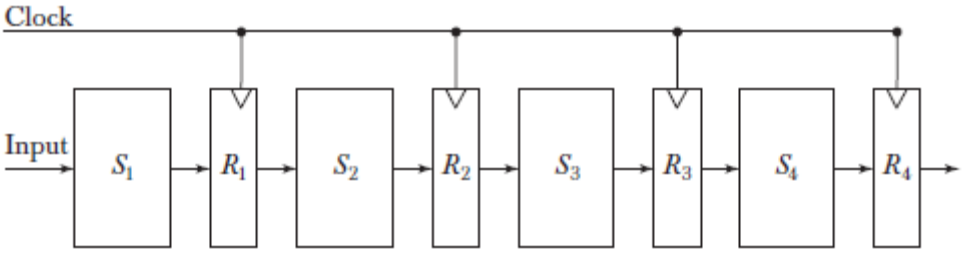


Figure Four-segment pipeline.

The behaviour of a pipeline can be illustrated with a *space-time* diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Figure below. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks $T1$ through $T6$ executed in four segments. Initially, task $T1$ is handled by segment 1. After the first clock, segment 2 is busy with $T1$, while segment 1 is busy with task $T2$. Continuing in this manner, the first task $T1$ is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

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

using a *k*-segment pipeline requires $k + (n - 1)$ clock cycles. For example, the diagram below shows four segments and six tasks. The time required to complete all the operations is $4 + (6 - 1) = 9$ clock cycles, as indicated in the diagram. Next consider a non pipeline unit that performs the same operation and takes a time equal to tn to complete each task. The total time required for n tasks is ntn. The speedup of a pipeline processing over an equivalent nonpipelined processing is defined by the ratio

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

Figure Space-time diagram for pipeline.

	1	2	3	4	5	6	7	8	9	→ Clock cycles
Segment: 1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

As the number of tasks increases, n becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of n. Under this condition, the speedup becomes

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

If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, we will have $tn = ktp$. Including this assumption, the speedup reduces to

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

This shows that the theoretical maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.

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. Consider the following two normalized floating point numbers

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

A and B are two fractions that represent the mantissas and a and b are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Figure below. The registers labelled *R* are placed between the segments to store intermediate results. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

The following numerical example may clarify the suboperations performed in each segment. Consider two the 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

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

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits.

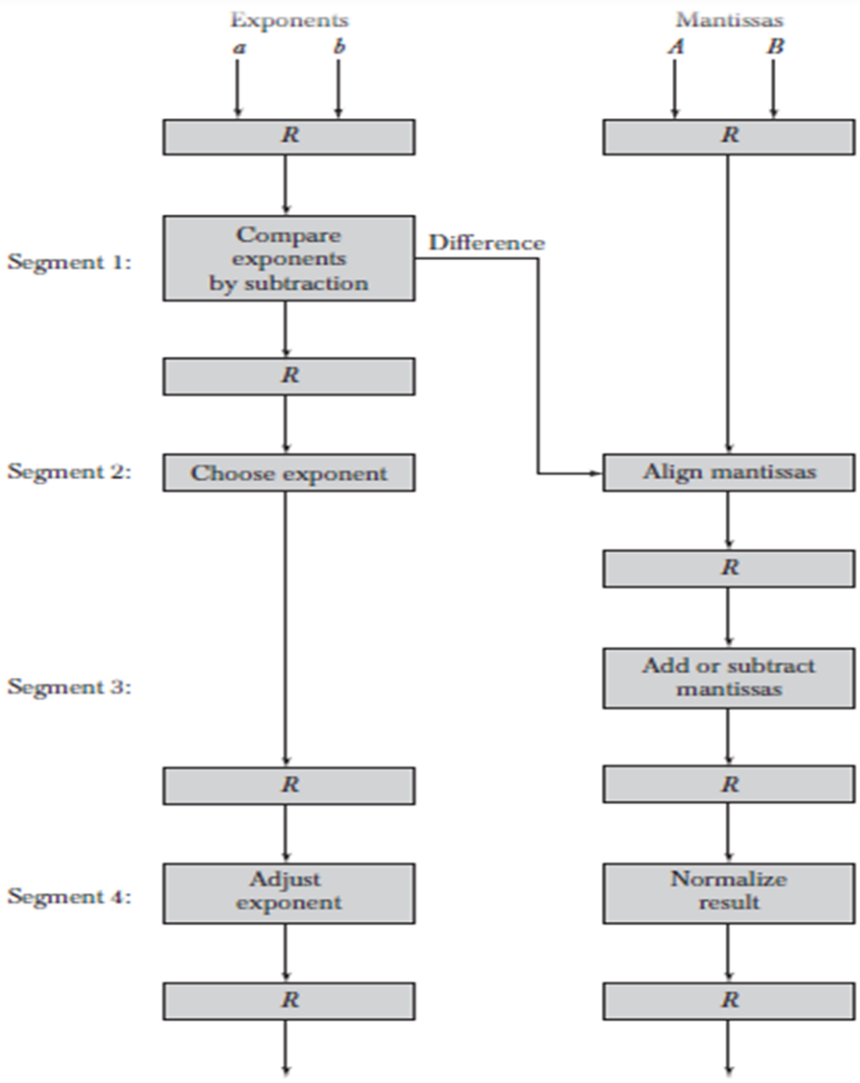


Figure Pipeline for floating-point addition and subtraction.

Instruction Pipeline

Pipeline processing can occur not only 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. One possible problem associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

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

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Example: Four-Segment Instruction Pipeline

Figure below shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO. Thus, up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time. Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

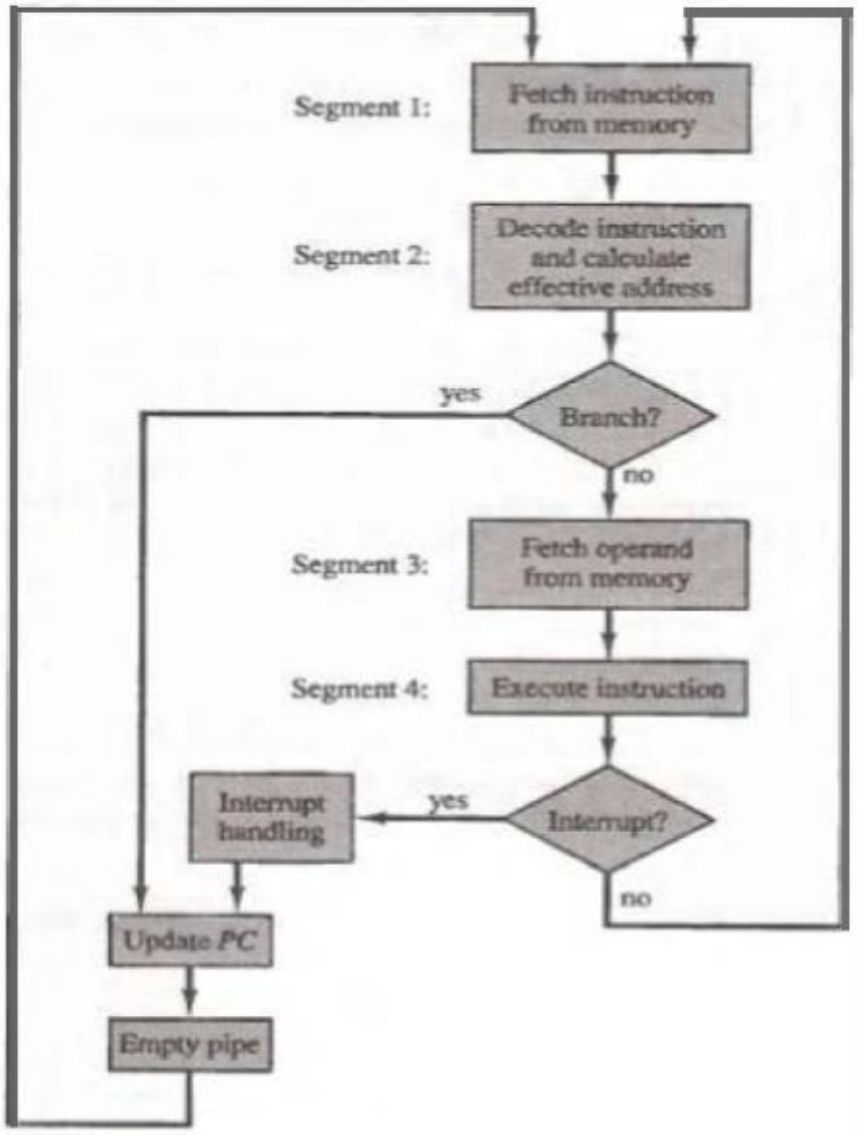


Figure Four-segment CPU pipeline.

Figure below shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. Fl is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in Fl and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI. Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered. Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure Timing of instruction pipeline.

Throughput: The amount of processing that can be accomplished during a given interval of time is called throughput.

[The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time.]

Pipelining hazards

• Pipeline hazards prevent next instruction from executing during designated clock cycle

• There are 3 classes of hazards:

1. **Structural Hazards:**

• Arise from resource conflicts

• HW cannot support all possible combinations of instructions

Avoid structural hazards by duplicating resources – e.g. an ALU to perform an arithmetic operation and an adder to increment PC

2. **Data Hazards:**

• Occur when given instruction depends on data from an instruction ahead of it in pipeline

ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11

In the above code, after **ADD** instruction result will be stored in **R1** register after the execution of the instruction. But the second instruction **SUB** is dependent on **R1** register because it is one of the source operand for it. But in the pipeline structure, by the time SUB instruction starts fetching operand the result of ADD will not be available in R1. So Data Hazard occurs.

3. **Control Hazards:**

• Result from branch, other instructions that change flow of program (i.e. change PC)

Parallel Processors

Introduction to parallel processors:

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of easing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

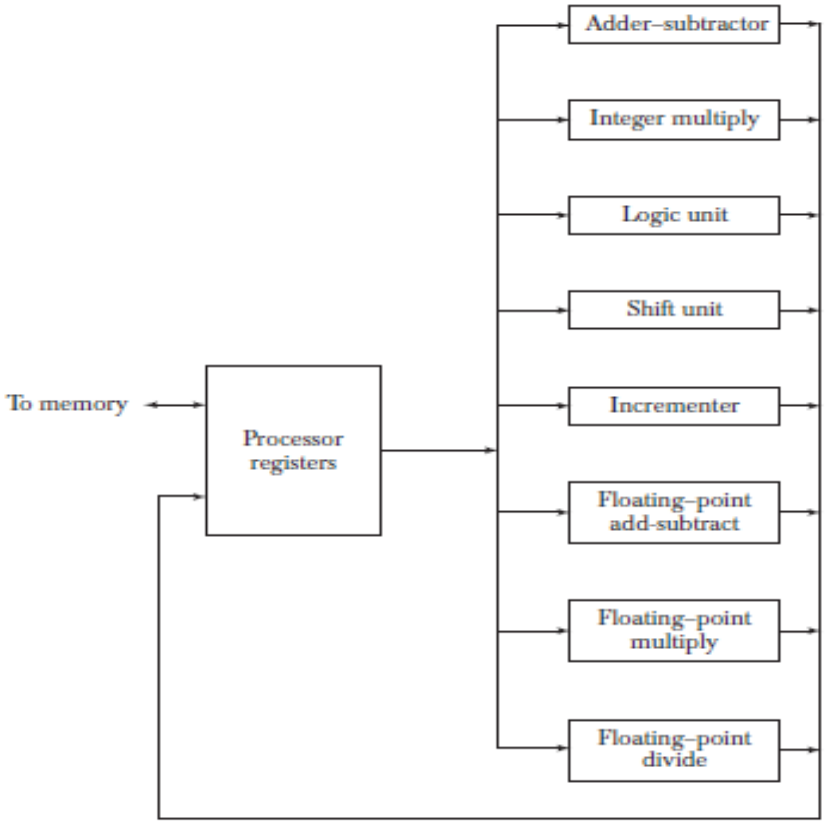
The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For

example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

Figure below shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers.

Figure Processor with multiple functional units.



Parallel Processing can be classified in a variety of way. M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor. The sequence of instructions read from memory constitutes an instruction stream . The operations performed on the data in the processor constitutes a data stream. Parallel processing may occur in the instruction stream, in the data stream, or in both.

Flynn's classification divides computers into four major groups as follows:

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.
SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.
MISD structure is only of theoretical interest since no practical system has been constructed using this organization.
MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category.

Concurrent access to memory and cache coherence:

The primary advantage of cache is its ability to reduce the average access time in **uniprocessors.** When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory.
Write-through policy: In the write-through policy, both cache and main memory are updated with every write operation.
Write-back policy: In the write-back policy, only the cache is updated and the location is marked so that it can be copied later into main memory.
In a **shared memory multiprocessor system**, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory.
To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical.
This requirement imposes a cache coherence problem. **A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address.** Without a proper solution to the cache coherence problem, caching cannot be used in bus- oriented multiprocessors with two or more processors.

Conditions for Incoherence

Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data. Read-only data can safely be replicated without cache coherence enforcement mechanisms.
To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. below. Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2, and P3. As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory. If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache.

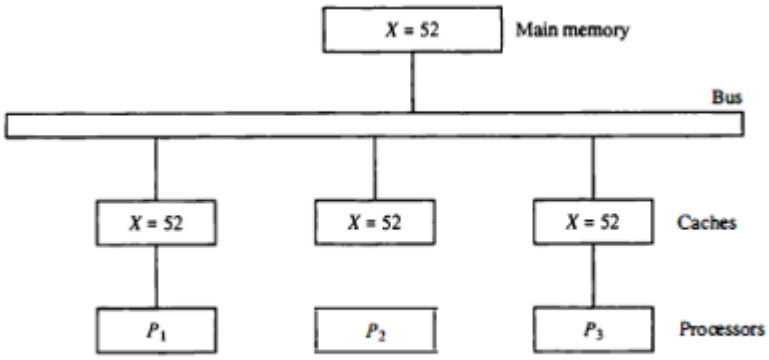
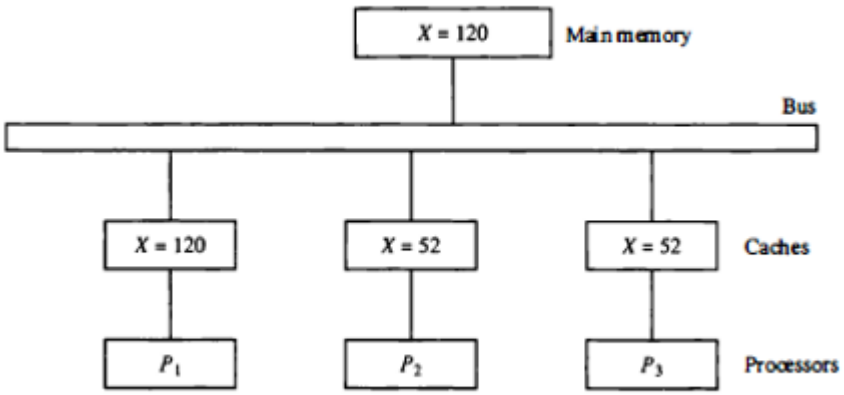


Figure Cache configuration after a load on X.

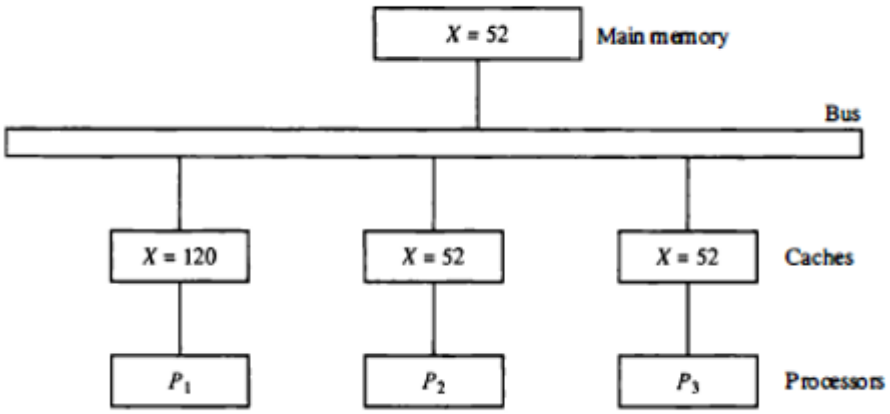
A store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy. A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value which is shown in figure below.

Cache configuration after a store to X by processor P_1.



(a) With write-through cache policy

In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.



(b) With write-back cache policy

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy.

The important terms related to the data or information stored in the cache as well as in the main memory are as follows:

- ***Modified*** - The modified term signifies that the data stored in the cache and main memory are different. This means the data in the cache has been modified, and the changes need to be reflected in the main memory.
- ***Exclusive*** - The exclusive term signifies that the data is clean, i.e., the cache and the main memory hold identical data.
- ***Shared*** - Shared refers to the fact that the cache value contains the most current data copy, which is then shared across the whole cache as well as main memory.
- ***Owned*** - The owned term indicates that the block is currently held by the cache and that it has acquired ownership of it, i.e., complete privileges to that specific block.
- ***Invalid*** - When a cache block is marked as invalid, it means that it needs to be fetched from another cache or main memory.

Below is a list of the different **Cache Coherence Protocols** used in multiprocessor systems:

- MSI protocol (Modified, Shared, Invalid)
- MOSI protocol (Modified, Owned, Shared, Invalid)
- MESI protocol (Modified, Exclusive, Shared, Invalid)
- MOESI protocol (Modified, Owned, Exclusive, Shared, Invalid)

There exist three varieties of coherency mechanisms, which are listed below:

1. ***Directory Based*** - A directory-based system keeps the coherence amongst caches by storing shared data in a single directory. In order to load an entry from primary memory into its cache, the processor must request permission through the directory, which serves as a filter. The directory either upgrades or devalues the other caches that contain that record when a record is modified.
2. ***Snooping*** - Individual caches watch address lines during the snooping process to look for accesses to memory locations that they have cached. A write invalidate protocol is what it is known as. When a write activity is seen to a memory address for which a cache maintains a copy, the cache controller invalidates its own copy of the snooped memory location.
3. ***Snarfing*** - A cache controller uses this approach to try and update its own copy of a memory location when a second master alters a place in the main memory by keeping an eye on both

the address and the contents. The cache controller updates its own copy of the underlying memory location with the new data when a write action is detected to a place of which a cache holds a copy.