MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

Autonomous Institution – UGC, Govt. of India

Department of CSE
(Artificial Intelligence and Machine Learning)

B. TECH (R-24 Regulation)

(II YEAR – I SEM)

2025-26

COMPUTER ORGANIZATION AND ARCHITECTURE

(R24A0561)

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 Computer Science and Engineering
(Artificial Intelligence and Machine Learning)

Vision

To be a premier center 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 instil in students a desire to conduct research in order to tackle challenging technical problems for industry by sustaining the ethical values.
- ❖ 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 AND TECHNOLOGY

II year B.Tech. CSE-I Sem **L/T/P/C**
3/1/0/4

(R24A0561) COMPUTER ORGANIZATION AND ARCHITECTURE

COURSE OBJECTIVES:

Course Objectives

● The purpose of the course is to introduce principles of computer organization and the basic architectural concepts.

● It begins with basic organization, design, and programming of a simple digital computer and introduces simple register transfer language to specify various computer operations.

● Topics include computer arithmetic, instruction set design, microprogrammed control unit, pipelining and vector processing, memory organization and I/O systems, and multiprocessors

UNIT - I

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.
Register Transfer Language and Micro operations: Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit.
Basic Computer Organization and Design: Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt.

UNIT - II

Microprogrammed Control: Control memory, Address sequencing, micro program example, design of control unit.
Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.

UNIT - III

Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation.
Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

UNIT – IV

Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt Direct memory Access.
Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

UNIT - V

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics.
Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor.
Multi Processors: Characteristics of Multiprocessors, Interconnection Structures, Inter processor arbitration, Inter processor communication and synchronization, Cache Coherence.

TEXT BOOK:

1. Computer System Architecture – M. Morris Mano, Third Edition, Pearson/PHI.

REFERENCE BOOKS:

1. Computer Organization – Carl Hamacher, Zvonks Vranesic, SafeaZaky, V th Edition, McGraw Hill.
2. Computer Organization and Architecture – William Stallings Sixth Edition, Pearson/PHI.
3. Structured Computer Organization – Andrew S. Tanenbaum, 4 th Edition, PHI/Pearson.

COURSE OUTCOMES:

● Understand the basics of instruction sets and their impact on processor design.
● Demonstrate an understanding of the design of the functional units of a digital computer system.
● Evaluate cost performance and design trade-offs in designing and constructing a computer processor including memory.
● Design a pipeline for consistent execution of instructions with minimum hazards. Recognize and manipulate representations of numbers stored in digital computers

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY
CSE (Artificial Intelligence and Machine Learning)

INDEX

****

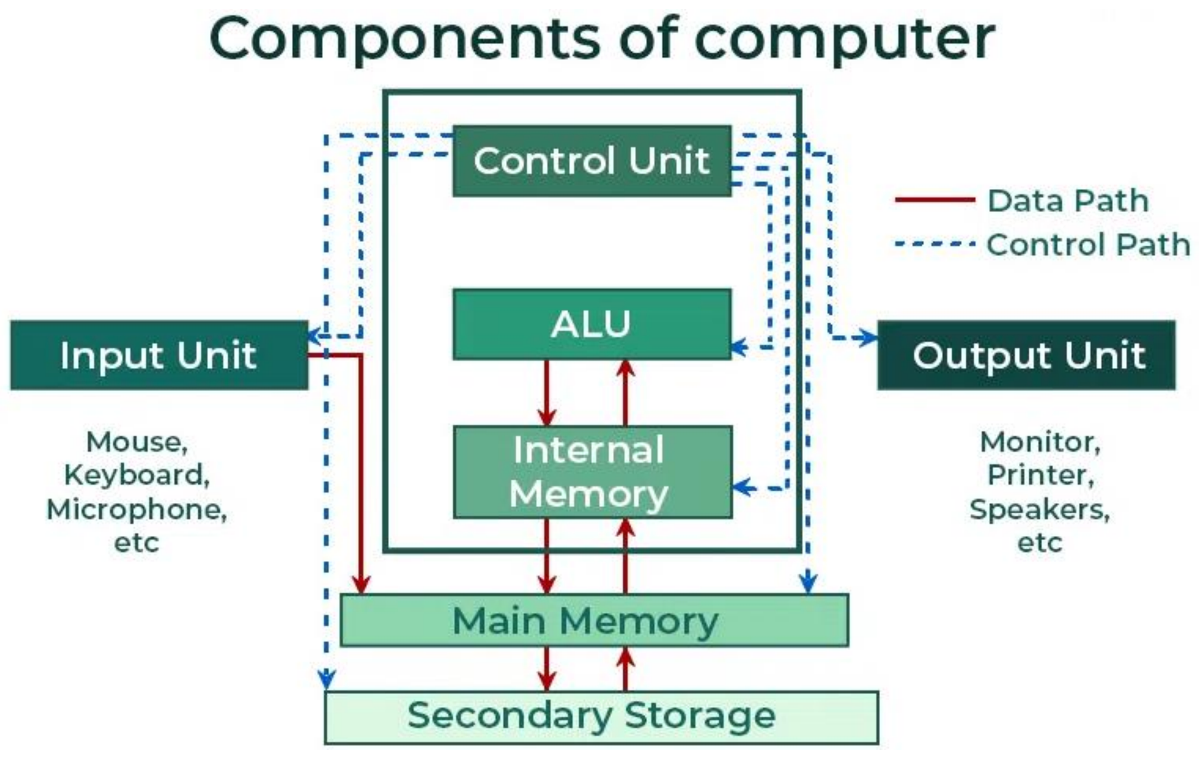
UNIT - I

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.

Register Transfer Language and Micro operations: Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit.

Basic Computer Organization and Design: Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt.

The **functional components of a digital computer** include the **Input Unit**, which takes in data; the **CPU**, which processes data with its **Control Unit (CU)**, **Arithmetic Logic Unit (ALU)**, and **Registers**; the **Memory Unit**, which stores data temporarily (RAM) or permanently (HDD/SSD); the **Output Unit**, which displays results; and the **Bus System**, which connects and transfers data between components. These parts work together to execute tasks and provide results.



The **functional components of a computer** are the key parts that work together to process and manage data. These include the **Input Unit** for receiving data, the **CPU** for processing it, the **Memory Unit** for storing information, the **Output Unit** for displaying results, and the **Bus System** that connects all parts. These components help the computer perform tasks efficiently.

1. Input Unit

- **Purpose:** Captures data and instructions from users or external sources.
- **Function:** Converts user input into binary signals that the computer can process.
- **Common Devices (2025):**
    - Keyboard, Mouse, Touchscreens
    - Scanners, Sensors, Stylus pens
    - Voice Assistants (e.g., Siri, Alexa)
    - Biometric devices (face/fingerprint recognition)
    - Iot-based inputs from smart devices

2. Central Processing Unit (CPU) – The Brain of the Computer

The **CPU** executes instructions and controls all internal operations. In 2025, CPUs will often have multiple **cores** and **threads** to handle parallel processing efficiently.

Components of CPU:

a. Arithmetic Logic Unit (ALU)

- Performs arithmetic operations (add, subtract, multiply, divide).
- Handles logical operations (comparison, decision-making).
- Supports AI/ML tasks using built-in vector/matrix operations (in modern CPUs).

b. Control Unit (CU)

- Directs the operations of all computer parts.
- Decodes instructions and coordinates data flow.
- Sends control signals to memory and I/O devices.

c. Registers

- **High-speed memory locations** within the CPU.
- Temporarily store instructions, addresses, and intermediate data.
- Examples: Accumulator, Instruction Register, Program Counter, Address Register.
- **Modern CPUs** include 64-bit or even 128-bit registers for faster processing.

3. Memory / Storage Unit

The **memory unit** holds data and instructions before, during, and after processing.

a. Primary Memory (Main Memory):

- **RAM (Random Access Memory):** Temporarily stores data during execution.
    - Types in 2025: DDR5, LPDDR5X, and emerging **MRAM**.
- **ROM (Read-Only Memory):** Stores boot-up instructions and firmware.
- **Cache Memory:** Ultra-fast memory between CPU and RAM (L1, L2, L3 levels).

b. Secondary Storage:

- Used for long-term data storage.
- **Examples**: SSDs (NVMe drives), HDDs, flash drives, and cloud storage.
- **Modern Trend:** Use of **Cloud Integration** and **hybrid storage models**.

4. Output Unit

- **Purpose:** Converts processed data (binary) into a form users can understand.
- **Examples:**
    - Visual: Monitors (LED, OLED, 4K/8K displays)
    - Print: Printers (Inkjet, Laser, 3D Printers)
    - Audio: Speakers, Headphones
    - Haptic: Vibration feedback devices

Computer Organization and Architecture is used to design computer systems.

- **Computer architecture** is about designing a computer system to balance performance, efficiency, cost, and reliability. It describes how a system is built from its components. This can be a high-level overview or a detailed explanation, including the instruction set architecture, micro architecture, logic design, and implementation.

- **Computer Organization** is about how the components of a computer system, like the CPU, memory, and input/output devices, are connected and work together to execute programs. It focuses on the operational aspects and how hardware components are implemented to support the architecture.

Register Transfer Language (RTL):

Register Transfer Language (RTL) is a low-level language that is used to describe the functioning of a digital circuit and, more specifically, the transfer of information between registers. It provides how data moves from one register to the other and how data is processed within the digital system. Through RTL, there is a capability of creating abstraction levels where high-level design descriptions can be created and easily linked to low-level hardware implementation in designing, simulating, as well as synthesizing digital circuits.

Register Transfer Operations
The operation performed on the data stored in the registers are referred to as register transfer operations.
There are different types of register transfer operations:
1. Simple Transfer - R2 <- R1
The content of R1 are copied into R2 without affecting the content of R1. It is an unconditional type of transfer operation.
2. Conditional Transfer
It indicates that if P=1, then the content of R1 is transferred to R2. It is a unidirectional operation.

3. Simultaneous Operations -
If 2 or more operations are to occur simultaneously then they are separated with comma **(,)**. If the control function P=1, then load the content of R1 into R2 and at the same clock load the content of R2 into R1.

Advantages of Register Transfer Language (RTL)

- Enables efficient hardware design.
- This makes it possible to simulate some activities and perhaps detect some errors at an early date.
- Implements conceptual description up to the gate-level hardware.
- It helps to reuse the design components.
- It gives a clear guide on how to do timing analysis on a given design.

In basic computer organization, an instruction is a binary code that tells the computer what operation to perform and where to find the data. It's a fundamental unit of a program, specifying the actions the CPU must execute. The instruction cycle, also known as the fetch-decode-execute cycle, is the process by which the CPU handles each instruction.

Elaboration:

- **Instruction Structure:**

A basic instruction typically consists of an opcode (operation code) and an address field. The opcode specifies the operation to be performed (e.g., add, subtract, load, store). The address field indicates the location of the data or the next instruction.

- **Instruction Cycle:**

The instruction cycle involves several steps:

1. **Fetch:** The CPU retrieves the instruction from memory using the program counter.
2. **Decode:** The CPU interprets the opcode and determines the necessary actions.
3. **Execute:** The CPU performs the specified operation, which may involve reading or writing data from memory, using the ALU, and updating registers.

- **Instruction Types:**
- **Memory-reference instructions:** These instructions operate on data stored in memory, including addressing modes like direct and indirect addressing.
- **Register-reference instructions:** These instructions operate on data stored in processor registers, such as the accumulator.
- **Input/output instructions:** These instructions control the transfer of data between the computer and external devices.

Instruction Set Architecture (ISA):

The ISA defines the set of instructions that a CPU can execute. It essentially acts as an interface between the hardware and software, specifying what the processor can do and how it does it.

Stored Program Concept:

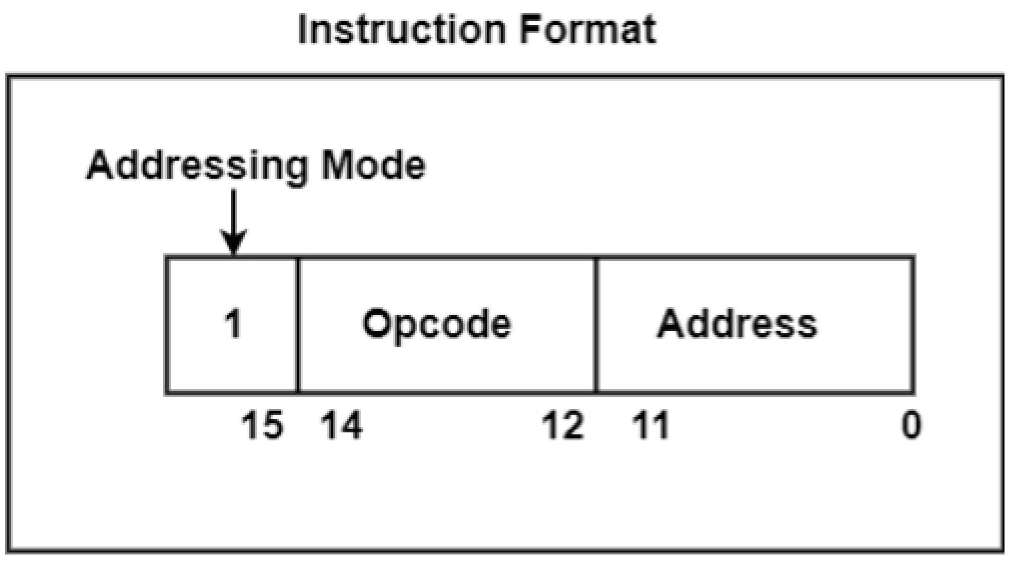
The computer stores both instructions and data in memory, allowing for flexible program execution and re-usability.

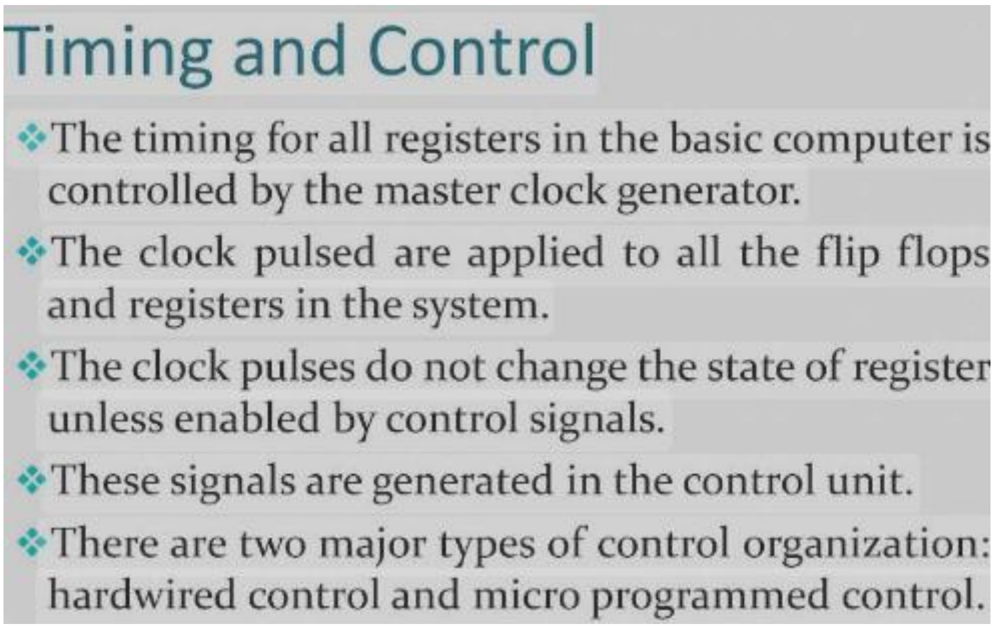
Basics of Computer Organization and Design -

In basic computer organization, an instruction is a binary code that tells the computer what operation to perform and where to find the data. It's a fundamental unit of a program, specifying the actions the CPU must execute. The instruction cycle, also known as the fetch-decode-execute cycle, is the process by which the CPU l

Computer Organization is about how the components of a computer system, like the CPU, memory, and input/output devices, are connected and work together to execute programs. It focuses on the operational aspects and how hardware components are implemented to support the architecture.

A **computer instruction** is a binary code that determines the micro-operations in a sequence for a computer. They are saved in the memory along with the information. Each computer has its specific group of instructions.
They can be categorized into two elements as Operation codes (Opcodes) and Address. Opcodes specify the operation for specific instructions. An address determines the registers or the areas that can be used for that operation. Operands are definite elements of computer instruction that show what information is to be operated on.
It consists of 12 bits of memory that are required to define the address as the memory includes 4096 words. The 15th bit of the instruction determines the addressing mode (where direct addressing corresponds to 0, indirect addressing corresponds to 1). Therefore, the instruction format includes 12 bits of address and 1 bit for the addressing mode, 3 bits are left for Opcodes.
The following block diagram shows the instruction format for a basic computer.





Timing and Control

- The timing for all registers in the basic computer is controlled by the master clock generator.
- The clock pulsed are applied to all the flip flops and registers in the system.
- The clock pulses do not change the state of register unless enabled by control signals.
- These signals are generated in the control unit.
- There are two major types of control organization: hardwired control and micro programmed control.

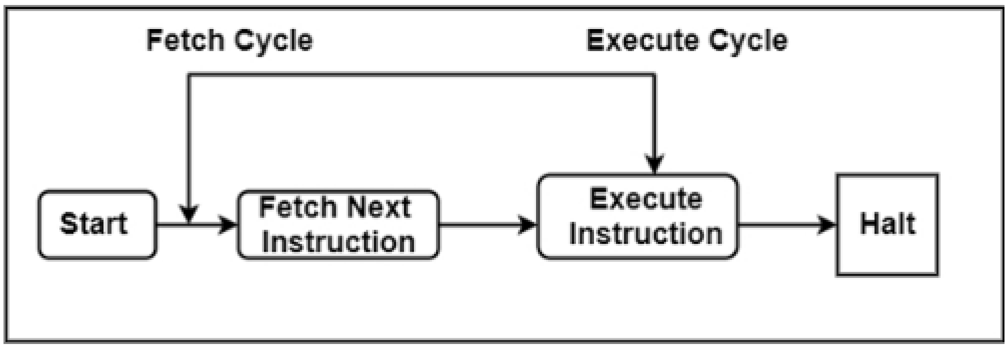
A program consisting of the **memory unit of the computer** includes a series of instructions. The program is implemented on the computer by going through a cycle for each instruction.

In the basic computer, each instruction cycle includes the following procedures −

- It can fetch instruction from memory.
- It is used to decode the instruction.
- It can read the effective address from memory if the instruction has an indirect address.
- It can execute the instruction.

After the following four procedures are done, the control switches back to the first step and repeats the similar process for the next instruction. Therefore, the cycle continues until a **Halt** condition is met. The figure shows the phases contained in the instruction cycle.

Instruction Cycle



Fetch Cycle

The address instruction to be implemented is held at the program counter. The processor fetches the instruction from the memory that is pointed by the PC.

Next, the PC is incremented to display the address of the next instruction. This instruction is loaded onto the instruction register. The processor reads the instruction and executes the important procedures.

Execute Cycle

The data transfer for implementation takes place in two methods are as follows −

- **Processor-memory** − The data sent from the processor to memory or from memory to processor.
- **Processor-Input/Output** − The data can be transferred to or from a peripheral device by the transfer between a processor and an I/O device.

In the execute cycle, the processor implements the important operations on the information, and consistently the control calls for the modification in the sequence of data implementation. These two methods associate and complete the execute cycle.

Memory Reference Instructions:

Memory Reference Instructions are instructions that involve accessing **main memory** for either fetching operands or storing results, typically using an **address field** in the instruction.

There are seven memory reference instructions which are as follows &

AND
The AND instruction implements the AND logic operation on the bit collection from the register and the memory word that is determined by the effective address. The result of this operation is moved back to the register.

ADD
The ADD instruction adds the content of the memory word that is denoted by the effective address to the value of the register.

LDA
The LDA instruction shares the memory word denoted by the effective address to the register.

STA
STA saves the content of the register into the memory word that is defined by the effective address. The output is next used to the common bus and the data input is linked to the bus. It needed only one micro-operation.

BUN
The Branch Unconditionally (BUN) instruction can send the instruction that is determined by the effective address. They understand that the address of the next instruction to be performed is held by the PC and it should be incremented by one to receive the address of the next instruction in the sequence. If the control needs to implement multiple instructions that are not next in the sequence, it can execute the BUN instruction.

BSA
BSA stands for Branch and Save return Address. These instructions can branch a part of the program (known as subroutine or procedure). When this instruction is performed, BSA will store the address of the next instruction from the PC into a memory location that is determined by the effective address.

ISZ

The Increment if Zero (ISZ) instruction increments the word determined by effective address. If the incremented cost is zero, thus PC is incremented by 1. A negative value is saved in the memory word through the programmer. It can influence the zero value after getting incremented repeatedly. Thus, the PC is incremented and the next instruction is skipped.

UNIT - II

Micro programmed Control: Control memory, Address sequencing, micro **program example, design of control unit.**

Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.

A control memory is a part of the control unit. Any computer that involves micro programmed control consists of two memories. They are the main memory and the control memory. Programs are usually stored in the main memory by the users. Whenever the programs change, the data is also modified in the main memory. They consist of machine instructions and data.

The control memory consists of micro programs that are fixed and cannot be modified frequently. They contain microinstructions that specify the internal control signals required to execute register micro-operations.

The machine instructions generate a chain of microinstructions in the control memory. Their function is to generate micro-operations that can fetch instructions from the main memory, compute the effective address, execute the operation, and return control to fetch phase and continue the cycle.

Here, the control is presumed to be a Read-Only Memory (ROM), where all the control information is stored permanently. ROM provides the address of the microinstruction. The other register, that is, the control data register stores the microinstruction that is read from the memory. It consists of a control word that holds one or more micro-operations for the data processor.

The next address must be computed once this operation is completed. It is computed in the next address generator. Then, it is sent to the control address register to be read. The next address generator is also known as the microprogram sequencer. Based on the inputs to a sequencer, it determines the address of the next microinstruction. The microinstructions can be specified in several ways.

The main functions of a microprogram sequencer are as follows −

- It can increment the control register by one.
- It can load the address from the control memory to the control address register.
- It can transfer an external address or load an initial address to begin the start operation.

The data register is also known as the pipeline register. It allows two operations to be performed at a time. It allows performing the micro-operation specified by the control word and also the generation of the next microinstruction.

A dual-phase clock is required to be applied to the address register and the data register. It is possible to apply a single-phase clock to the address register and work without the control data register.

The main advantage of using a microprogrammed control is that, if the hardware configuration is established once, no further changes can be done. However, if a different control sequence is to be implemented, a new set of microinstructions for the system must be developed.

The address sequencing can be done in 4 ways.

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.

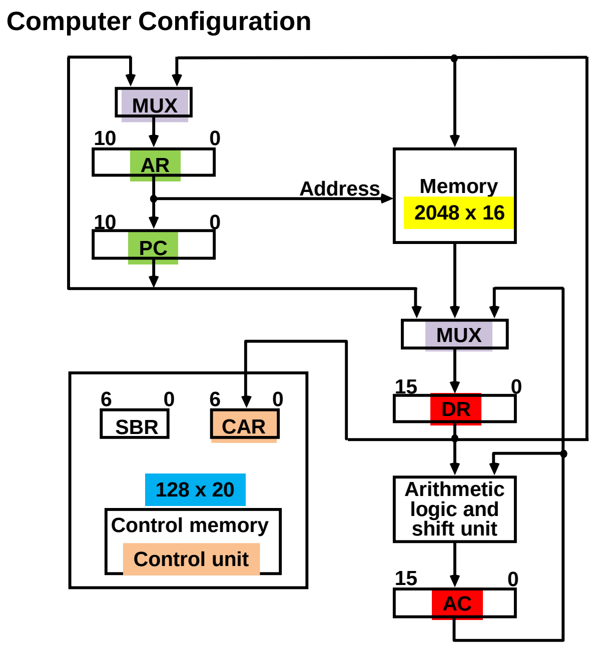
13. Explain the role of microprogram Example ?

A micro program is a collection of microinstructions that tells a microprocessor how to perform operations. Micro programs are stored in control memories and can be used to customize and enhance the functionality of a CPU.

- In micro programming we have 5 sub topics
- 1) Computer Configuration
- 2) Micro Instruction format
- 3) Symbolic Micro instruction
- 4) Symbolic Micro Program

5) Binary Micro program

MICROPROGRAM EXAMPLE



4

This contains two memories Units

1. Main Memory

2. Control Memory

Main Memory: This is used to store instructions and data. The capacity of Main Memory is 2048 x 16.

It contains 2048 words , each word size is of 16 bits each.

11 bits are required to identify Address of each location.

Control Memory: used to store Micro program which is nothing but sequence of instructions .

The capacity of Main Memory is 128 x 20. each word is 20 bits and 07 bits are required to identify address of each location

5

- Control Memory is associated with CAR and SBR.
- CAR: Control Address Register : It provides the address of the micro instructions which are present in the control memory. The size of CAR is 7 bits.
- SBR : Sub Routine Register : when ever a subroutine called in the program the control transfers from main program to sub routine program , once those instructions are completed the control come back to the address which is specified by SBR . SBR contains return address.

- Here we have two multiplexers

- MUX -1 have two inputs from PC and DR and one output which goes to AR.
- MUX -2 have three inputs from PC , DR and main memory one output which goes to DR.
- Among these 4 registers two registers are associated for address(AR & PC), two registers are associated for data(DR & AC)

- AR-Address Register: It provides the address of the instructions that are stored in main memory. 11 bits. It is receiving information from Pc as well as DR through MUX-1

- PC: Program Counter always points the next instruction to be fetched. 11 bits. It is receiving information only from AR.

8

- DR: Data Register contains the data. 16 bits. It is receiving information from AC, DR and main memory through MUX-2
- AC: Accumulator : It contains the operands and the result. 16 bits. It is receiving information only from ALU.
- ALU is receiving information only from DR.

- Among these 4 registers two registers are associated for address(AR & PC), two registers are associated for data(DR & AC)

9

Main Memory Operations:

- Read Operation: If we want to perform read information from particular location which is present in main memory . We required address that provided by AR go to that location from that location read the data and send to the DR.
- Write Operation: DR provides data . That data to be write in memory location specified by the AR.
- Micro programmed unit provides control signals

10

MACHINE INSTRUCTION FORMAT

Microinstruction Format

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

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

MICROINSTRUCTION FIELD DESCRIPTIONS - F1,F2,F3

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

DRTAR **READ, INCPC**

PCTAR

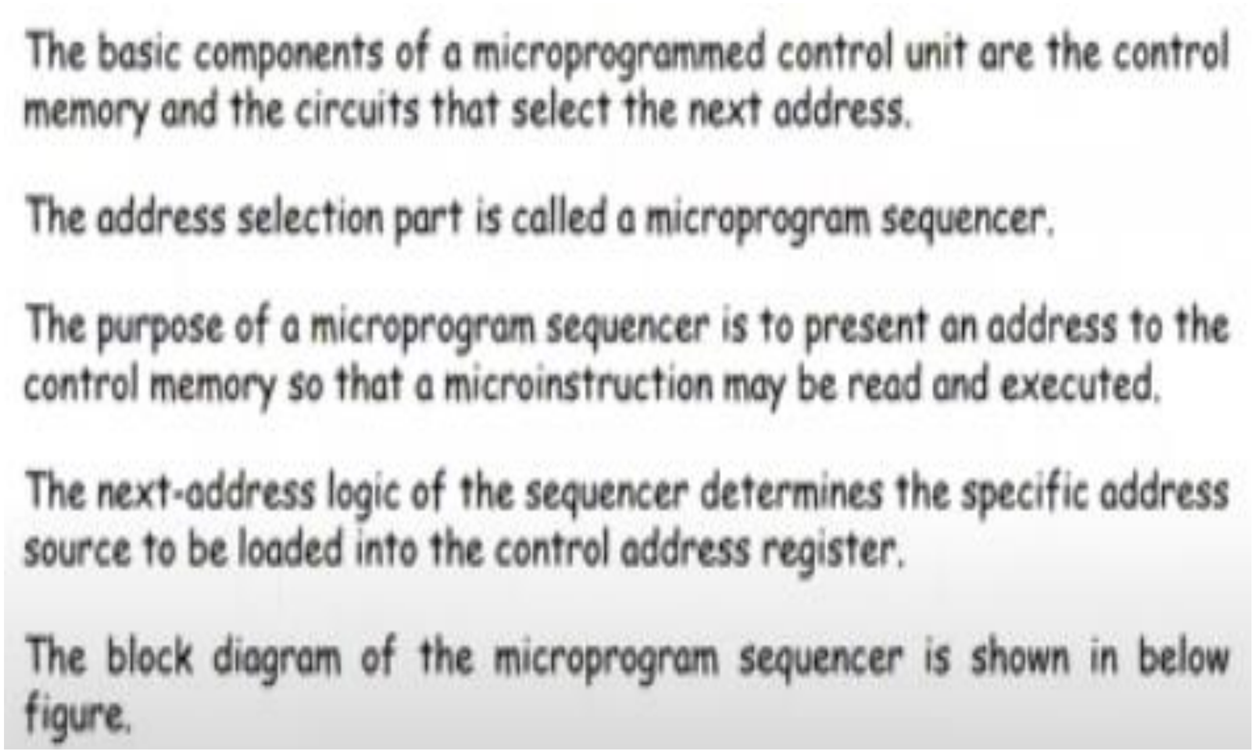
F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC’	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

MICROINSTRUCTION FIELD DESCRIPTIONS - CD, BR

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 ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

Microprogram sequencer in generating next address for control memory?



The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.

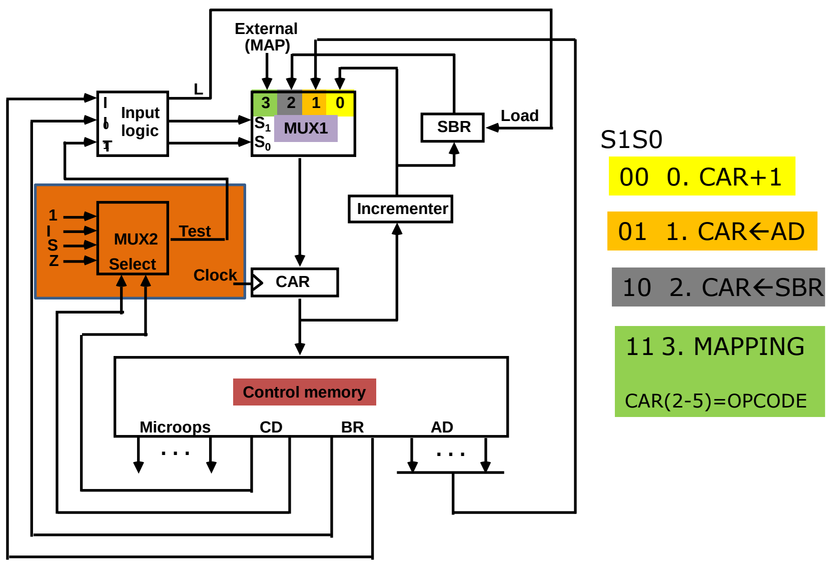
The address selection part is called a microprogram sequencer.

The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.

The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.

The block diagram of the microprogram sequencer is shown in below figure.

MICROPROGRAM PROGRAMMED CONTROL



3

MICRO PROGRAM EXAMPLE

- **Microprogrammed Control Unit :**
  Microprogrammed Control Unit produces control signals by using micro-instructions.
- **Micro program :**
- A program is a set of instructions. An instruction requires a set of micro-operations.
- Micro-operations are performed using control signals.Here, these control signals are generated using micro-instructions.This means every instruction requires a set of micro-instructions
- A set of micro-instructions are called micro-program.
- Microprograms for all instructions are stored in a small memory called control memory.
  The control memory is present inside the processor.

4

- Consider an instruction that is fetched from the main memory into the instruction Register (IR). The processor uses its unique opcode to identify the address of the first micro-instruction. That address is loaded into CMAR (Control Memory Address Register). This address is decoded to decide the corresponding memory instruction from the control Memory.
- Micro-instructions will only have a control field. The control field Indicates the control signals to be generated. Most micro-instructions will not have an address field. Usually PC will simply get incremented after every micro-instruction.

5

- This is as long as the micro-program is executing sequentially. If there is a branch micro-instruction only then there will be an address filed.
- If the branch is unconditional, the branch address will be directly loaded into CMAR. For conditional branches, the branch condition will check the appropriate flag.
- This is done using a MUX which has all flag inputs. If the condition is true, then the mux will inform CMAR to load the branch address.
- If the condition is false CMAR will simply get incremented. The control memory is usually implemented using flash ROM as it is non-volatile.

6

MICROINSTRUCTION FIELD DESCRIPTIONS - CD, BR

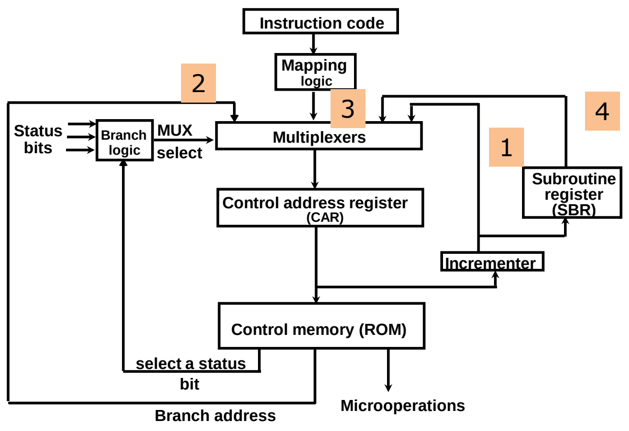
CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

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

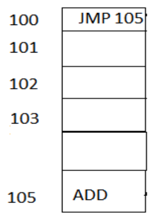
0XXXX00
0001000

7

ADDRESS SEQUENCING

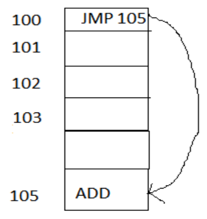


- **Incrementing of the control address register** An initial address is loaded into the control address register when power is turned on in the computer (CAR=100)
- This address is usually the address of the first
- microinstruction that activates the instruction fetch routine.
- At the end of the fetch routine, the content of the CAR is incremented to fetch the next instruction in sequence.
- Incrementor increments the content of CAR by 1(CAR=101) . This is loaded in to CAR through the multiplexer.



Unconditional branch or conditional branch, depending on status bit conditions.

- Without checking the any condition we are transferring the program control from one location to another location. That branch address is specified within the instruction.
- Example: Branch Address is 105 specified in the instruction.



conditional branch, depending on status bit conditions

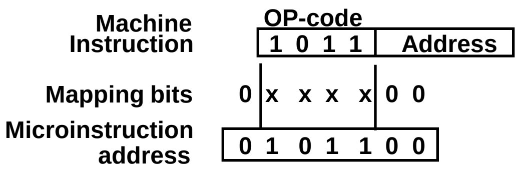
- In this operation we need to check the condition (Zero flag, Sign Flag, Carry Flag)of Branch Logic (BL)
- If BL= 1 :we need to go to that particular location
- If BL= 0: simply move forward and increment the CAR.

A mapping process from the bits of the instruction to an address for control memory.

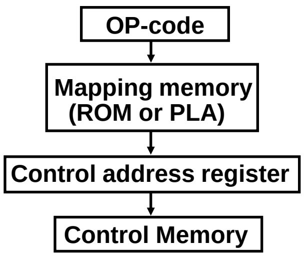
- Conversion from 4 bit opcode into 7 bit address in control memory where the data is located is referred to as a **mapping process**.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

MAPPING OF INSTRUCTIONS TO MICROROUTINES

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its execution microprogram



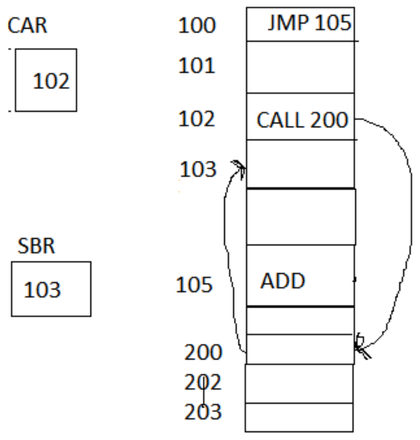
Mapping function implemented by ROM or PLA



10

A facility for Subroutine Call and Return.

Subroutine call is a simple program or functions. Which performs particular task. After completing that task the control return back to calling function. This return address is loaded in the SBR(Sub Routine Register)



11

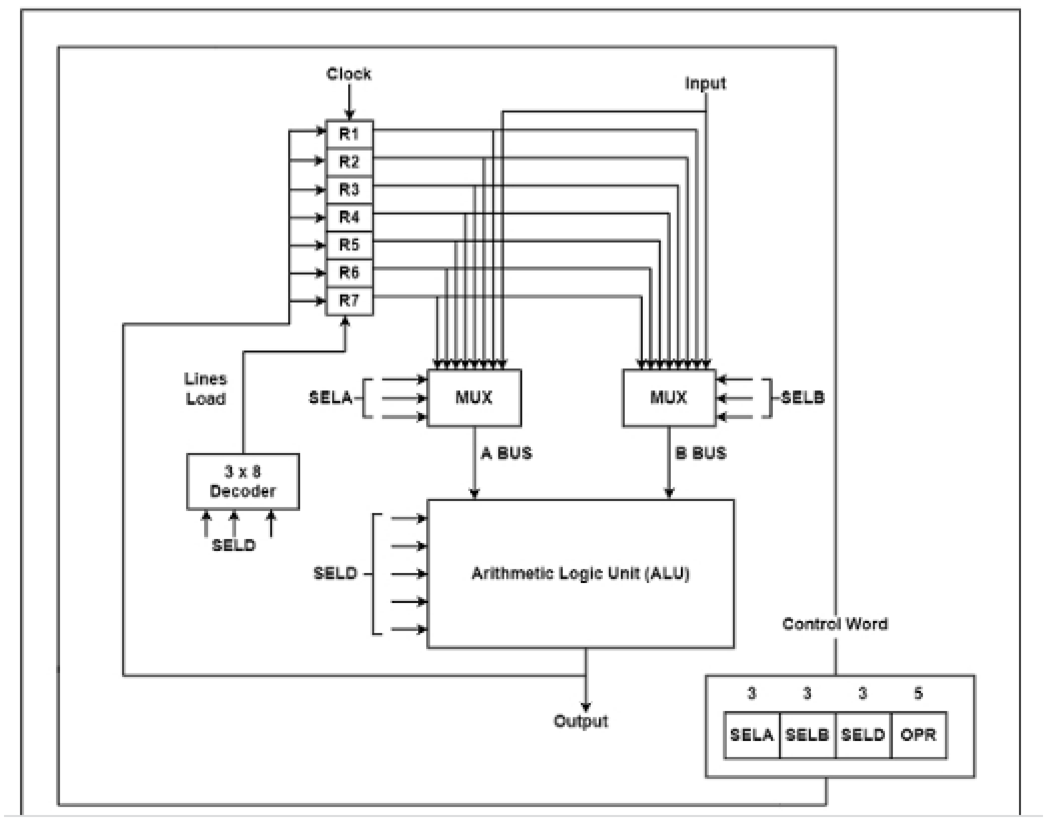
Here multiplexer (4 x 1) getting from 4 inputs and produces one output. This output which is the address of the operand is loaded in CAR. Here Car operated by Clock .

Registers implement two important functions in the CPU operation are as follows −

- It can support a temporary storage location for data. This supports the directly implementing programs to have fast access to the data if required.
- It can save the status of the CPU and data about the directly implementing program

If a CPU includes some registers, therefore a common bus can link these registers. A general organization of seven CPU registers is displayed in the figure.

General Organization of Registers



The CPU bus system is managed by the control unit. The control unit explicit the data flow through the **ALU** by choosing the function of the ALU and components of the system.

Consider R1 ← R2 + R3, the following are the functions implemented within the CPU −

MUX A Selector (SELA) − It can place R2 into bus A.

MUX B Selector (SELB) − It can place R3 into bus B.

ALU Operation Selector (OPR) − It can select the arithmetic addition (ADD).

Decoder Destination Selector (SELD) − It can transfers the result into R1.

The multiplexers of 3-state gates are performed with the buses. The state of 14 binary selection inputs determines the control word. The 14-bit control word defines a micro-operation.

Addressing modes with numerical example.

Types of Addressing Modes

1. Implied Mode
2.Immediate Mode
3.RegisterAddressing Mode
4. Register Indirect Mode
5. Direct Addressing Mode
6. Indirect Addressing Mode
7. Relative Addressing Mode
8. Indexed Addressing Mode
9. Base Register Addressing Mode
10. Auto Increment Addressing Mode
11. Auto Decrement Addressing Mode

1. Implied Mode: Address of the operands are specified implicitly in the definition of the instruction. No need to specify address in the instruction
EX:

- CMA (Complement Accumulator)
- CLA – Clear Accumulator,
- INC – Increment Accumulator

Zero address instructions in stack organization are Implied Mode instruction

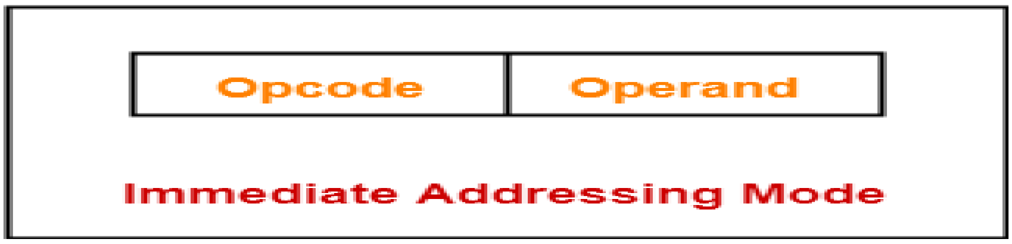
Ex: PUSH A, PUSH B

2. Immediate Mode : The operand itself is specified explicitly in the definition of the instruction. The operand field contains operand rather than address field. Fast to acquire an operand.

EX:

- MOV A #10   # indicates that 10 is immediate value
- ADD 23,45

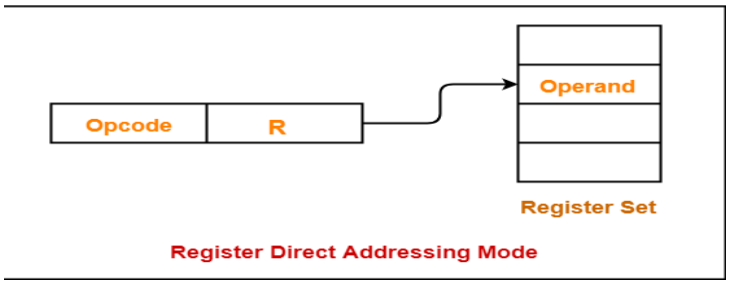
This addressing is used for initializing registers to a constant value



3.Register Addressing Mode : In this mode the operand is specified in the register.The name of the register is specified In the instruction
Advatages:

- -- Shorter address than the memory address
- -- Saving address field in the instruction
- -- Faster to acquire an operand than the memory addressing

EX:
MOV A,B
ADD R1,R2   R1=23 R2=45



4. Register Indirect Addressing Mode : Instruction specifies a register which contains the memory address of the operand

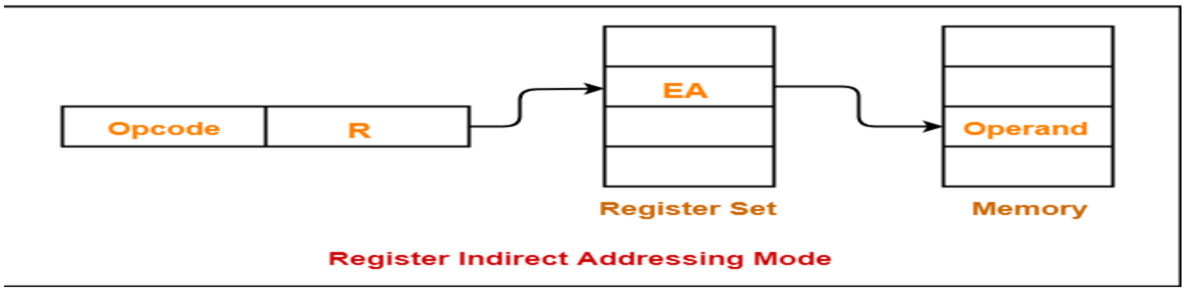
- Ex:

  MOV A, [R0]

R0 contains the address of the operand in memory
go to that address in memory to fetch the data , that data is moved to A.

- When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 Automatically

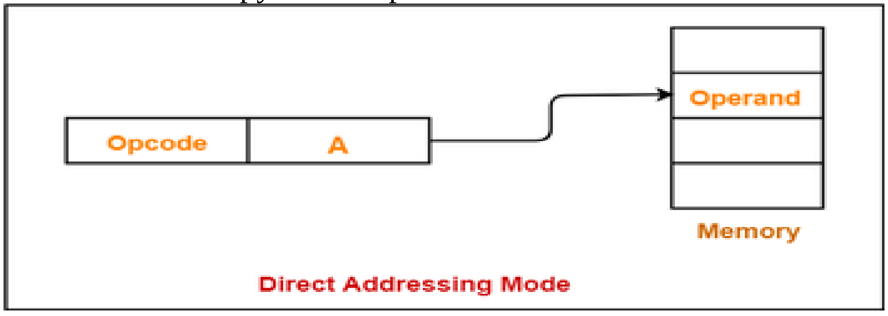
R0 – does nor contain data
R0 contains the address of the data where it is located



5. Direct Address Mode : Instruction specifies the memory address which can be used directly to the physical memory
EX:

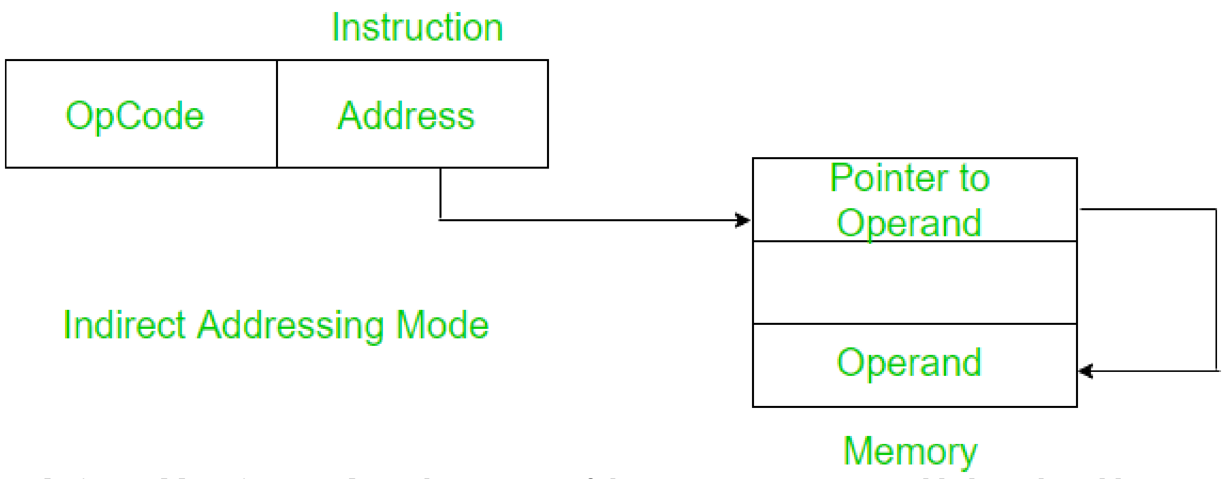
MOV A , 2000

This instruction copy the data present in the location 2000 and move to register A.



6. **Indirect Addressing Mode:** The address field of an instruction specifies the address of a memory location that contains the address of the operand.

EX:
MOV A, [2000]
let the content of 2000 is 3000
3000 is the address of the operand
In address 3000 we can get the operand



7.Relative Addressing Modes: The content of the program counter is added to the address part of the instruction in order to get the effective address of the operand.

Effective Address= Address of the operand

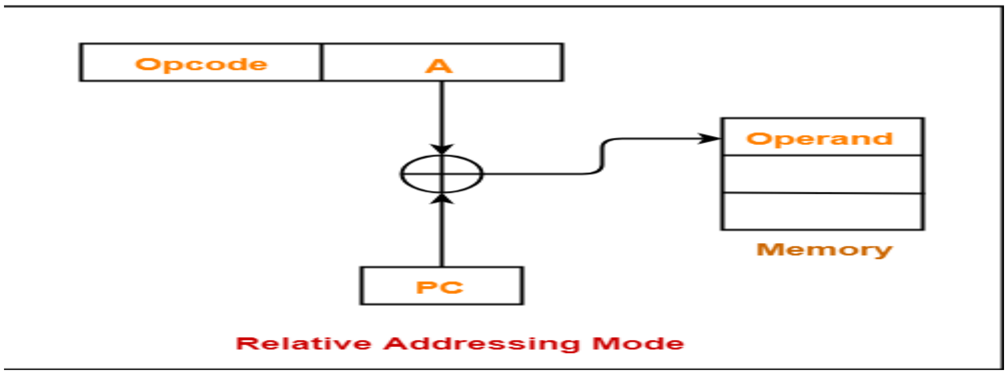
EA= PC+ Address present in the address part of the instruction.

Let PC contains 825

The address part of the Instruction contains = 24

The instruction at location 825 is read from the memory during fetch phase and program counter is incremented by 1 so present value in PC=826 then

The address of the operand = 826+24= 850

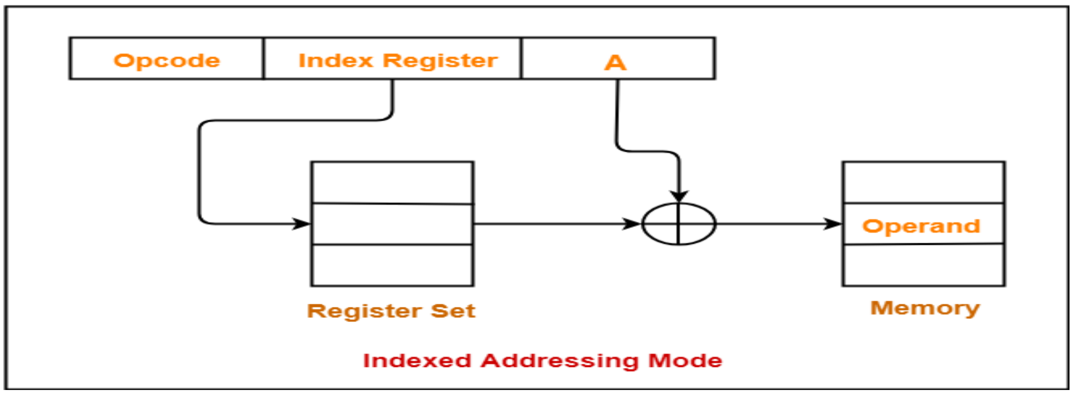


8.Indexed Addressing Mode : In this addressing mode the content of the index register is added to the address part of the instruction to obtain the address of the operand or Effective Address.

Index Register(IR) Is a special CPU register which contains the index value.

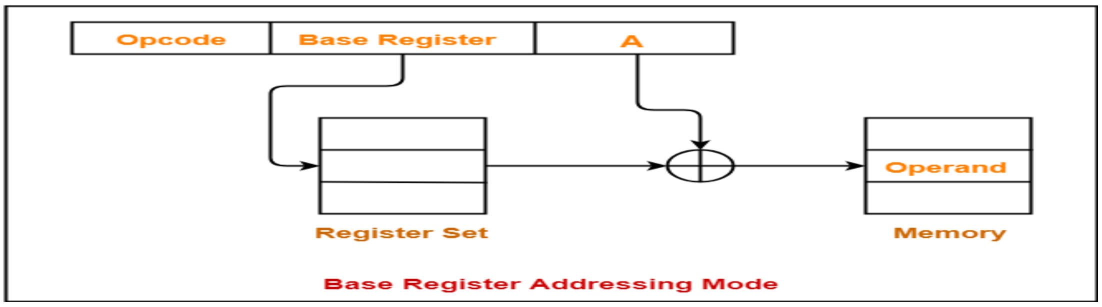
Effective Address=

Content of the Index Register + Address part of the instruction.



9. 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 address of the operand or Effective Address.

Effective Address = content of the Base Register + Address part of the instruction

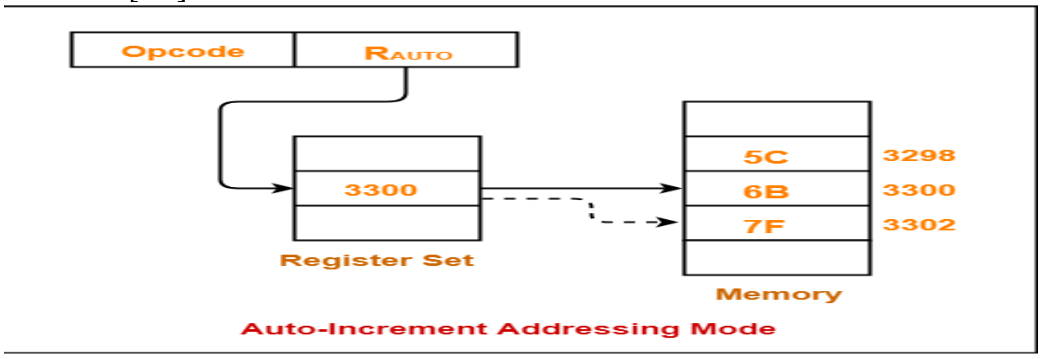


10. Auto Increment Addressing Mode

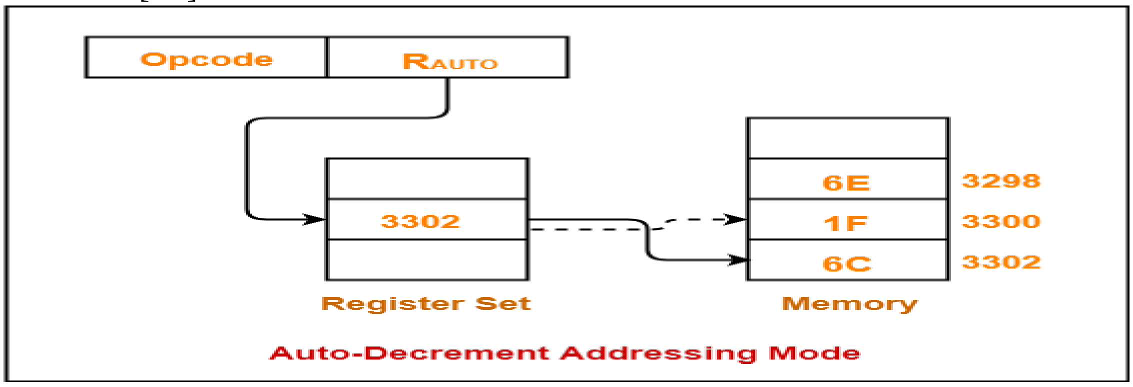
In this addressing mode the Effective Address operand is the content of the register specified in the instruction. After accessing the operand the content of this register are incremented to the address of the next location.

EX:

MOV R1 [R0]+

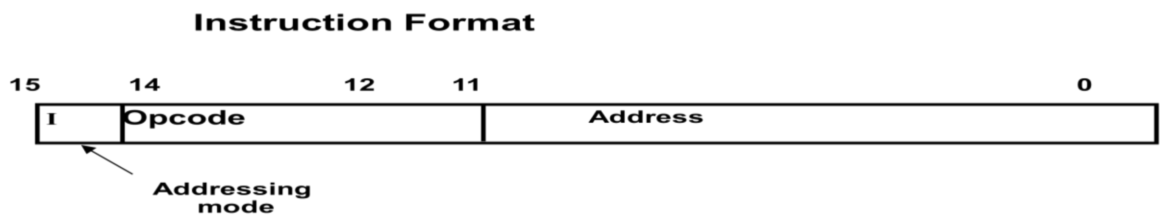


11. Auto Decrement Addressing Mode : In this addressing mode the Effective Address operand is the content of the register specified in the instruction. After accessing the operand the content of this register are decremented to the address of the next location.
EX:
MOV R1 -[R0]



Instruction formats with suitable example

INSTRUCTION FORMAT
A Program is a Set of instructions that instruction are stored in the memory in particular format, that format is called instruction format.
This instruction format has 32 bit , 64 bit and 16 bit formats. Here we are using 16 bit format The bits of the instruction format divided into fields.



- **Mode field** - specifies the way the operand or the effective address is determined
  I = 0 -→ Direct Addressing Mode
  I = 1 --→ Indirect Addressing Mode
- **OP-code field** - specifies the operation to be performed Examples of opcodes
  ADD: Adds two values
  SUB: Subtracts two values
  DIV: Divides two values

- **Address field** - designates memory address(es) or a processor register(s)

  The number of address fields in the instruction format depends on the internal organization of CPU
  - **Three-address Instructions**
  - **Two-address Instructions**
  - **One-address Instructions**
  - **Zero-address Instructions**

  Example = (A + B) * (C + D)

THREE ADDRESS INSTRUCTIONS

- *Advantages:*
- - Results in short programs
- **Disadvantages:**
- - Instruction becomes long(many bits) to specify three addresses
- Program to evaluate X = (A + B) * (C + D)

OPCODE	ADDRESS FIELD1	ADD F2	ADD F 3

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

1

TWO ADDRESS INSTRUCTIONS

- Program to evaluate X = (A + B) * (C + D) :

OPCODE	ADDRESS FIELD1	ADD F2

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

- - Computers with two-address instructions are most common

1

ONE-ADDRESS INSTRUCTIONS

- ***One-Address Instructions***
- **- Use an implied AC register for all data manipulation**
- **- Program to evaluate X = (A + B) * (C + D) :**

OPCODE	ADDRESS FIELD1

$X = (A + B) * (C + D)$ is

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

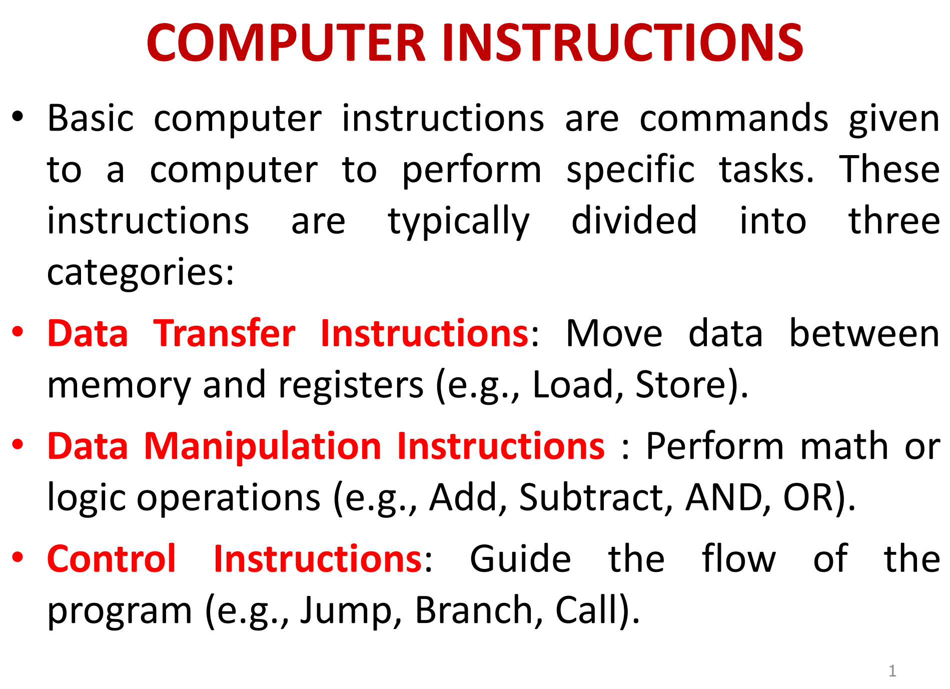
Zero-Address Instructions

- **- Can be found in a stack-organized computer**
- **- Program to evaluate X = (A + B) * (C + D) :**

OPCODE

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

Data transfer, data manipulation and program control instructions.

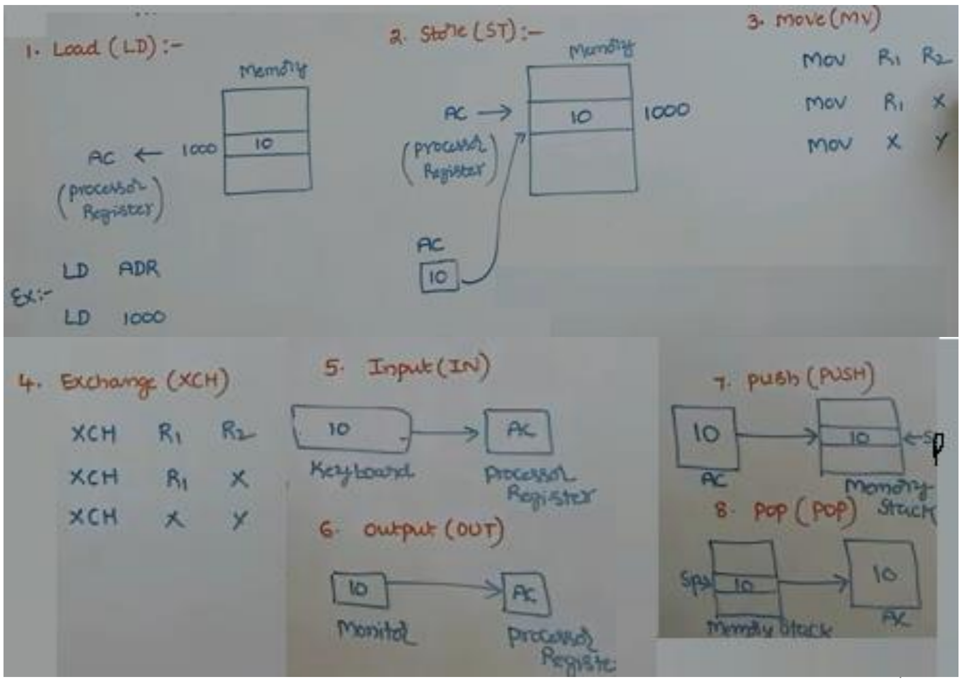


COMPUTER INSTRUCTIONS

- Basic computer instructions are commands given to a computer to perform specific tasks. These instructions are typically divided into three categories:
- **Data Transfer Instructions**: Move data between memory and registers (e.g., Load, Store).
- **Data Manipulation Instructions** : Perform math or logic operations (e.g., Add, Subtract, AND, OR).
- **Control Instructions**: Guide the flow of the program (e.g., Jump, Branch, Call).

1

NAME	MNEMONICS	DESCRIPTION
Load:	**LD**	Copies data from memory to a register
Store:	**ST**	Transfers data from a register to memory.
Move	**MOV**	Transfers data from one register to another.
Exchange	**XCH**	Exchange the data from one location to another
Input	**IN**	Provide data to a computer program or system. I
Output	**OUT**	Moves data from a memory address to an I/O port
Push	**PUSH**	Saves data to a stack in memory
Pop	**POP**	Retrieves data from a stack, or restores the status of an instruction

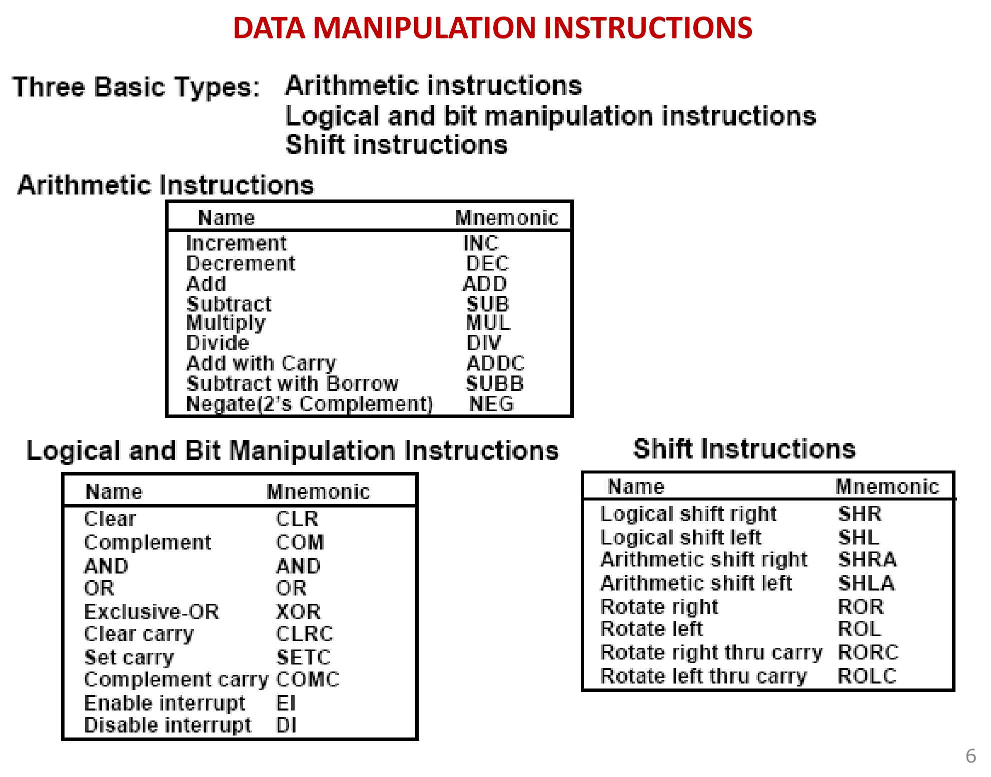


DATA MANIPULATION INSTRUCTIONS

These instructions modify data to execute program

They are broadly categorized into three types:

- Arithmetic instructions
- Logical and bit manipulation instructions
- Shift instructions



DATA MANIPULATION INSTRUCTIONS

Three Basic Types: Arithmetic instructions
Logical and bit manipulation instructions
Shift instructions

Arithmetic Instructions

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

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 thru carry	RORC
Rotate left thru carry	ROLC

6

Program Control Instructions are the machine code instructions which are used to control the flow of execution of instructions in the processor domain. These are important in instilling on the processor how to execute a certain task, access different programs and control the decision making on the basis of some conditions. They are commonly used in assembly language and generated by high level language which is compiled into machine code form to enable the processor act in the desired manner.

Types of Program Control Instructions

1. Compare Instruction

Compare instruction is specifically provided, which is similar to a subtract instruction except the result is not stored anywhere, but flags are set according to the result.
Example: *CMP R1, R2 ;*

2. Unconditional Branch Instruction

It causes an unconditional change of execution sequence to a new location.
Example: *JUMP L2 Mov R3, R1 goto L2*

3. Conditional Branch Instruction

A conditional branch instruction is used to examine the values stored in the condition code register to determine whether the specific condition exists and to branch if it does.
Example: *Assembly Code :*

BE R1, R2, L1 Compiler allocates R1 for x and R2 for y

High Level Code: if (x==y) goto L1;

4. Subroutines

A subroutine is a program fragment that lives in user space, performs a well-defined task. It is invoked by another user program and returns control to the calling program when finished.
Example: *CALL and RET*

5. Halting Instructions

- **NOP Instruction -** NOP is no operation. It cause no change in the processor state other than an advancement of the program counter. It can be used to synchronize timing.

- **HALT -** It brings the processor to an orderly halt, remaining in an idle state until restarted by interrupt, trace, reset or external action.

6. Interrupt Instructions

Interrupt is a mechanism by which an I/O or an instruction can suspend the normal execution of processor and get itself serviced.

- **RESET -** It reset the processor. This may include any or all setting registers to an initial value or setting program counter to standard starting location.
- **TRAP -** It is non-maskable edge and level triggered interrupt. TRAP has the highest priority and vectored interrupt.
- **INTR -** It is level triggered and maskable interrupt. It has the lowest priority. It can be disabled by resetting the processor.

Advantages of Program Control Instructions

- **Efficient Control Flow:** Program Control Instructions provide the processor with the means to decide the order of instructions. This means that branching can be efficiently made which is important for complicated

UNIT - III

Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation.

Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations

Data Representation:

- Data Representation refers to the form in which data is stored, processed, and transmitted.
- In computer organization, data refers to the symbols that are used to represent events, people, things and ideas.

Data can be anything like a number, a name, notes in a musical composition, or the color in a photograph. Data representation can be referred to as the form in which we stored the data, processed it and transmitted it. In order to store the data in digital format, we can use any device like computers, smartp hones, and iPads. Electronic circuitry is used to handle the stored data.

The Number Systems used in computers are

- Binary number system
- Octal number system
- Decimal number system
- Hexadecimal number system
- Binary Coded decimal number system(BCD)

Binary number system

- It has only two digits '0' and '1' so its base is 2. Each digit is called a bit.
- A group of four bits (1101) is called a nibble a
- Group of eight bits (11001010) is called a byte. The position of each digit in a binary number represents a specific power of the base (2) of the number system.

Decimal Number	Binary Number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011

Octal number system

- It has eight digits (0, 1, 2, 3, 4, 5, 6, 7) so its base is 8. Each digit in an octal number represents a specific power of its base (8). The three binary digits can be represented with a single octal digit**.**

Decimal Number	**Octal Number**
0	**0**
1	**1**
2	**2**
3	**3**
4	**4**
5	**5**
6	**6**
7	**7**
8	**10**
9	**11**
10	**12**
11	**13**
12	**14**

Decimal number system

- This number system has ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) so its base is 10.
- In this number system, the maximum value of a digit is 9 and the minimum value of a digit is 0.
- The position of each digit in decimal number represents a specific power of the base (10) of the number system. This number system is widely used in our day to day life.
- It can represent any numeric value.

Hexadecimal number system:

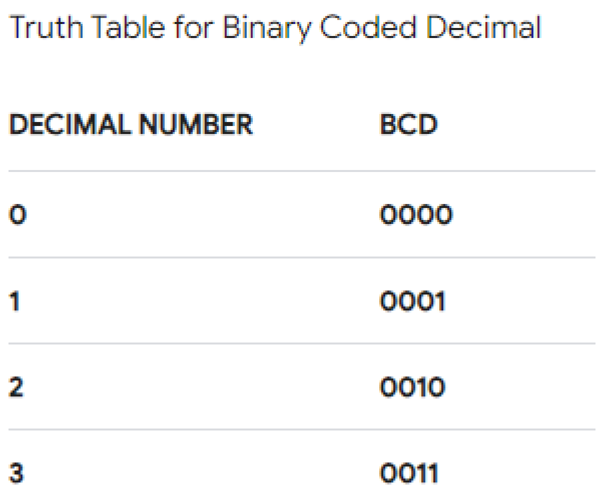
- This number system has 16 digits that ranges from 0 to 9 and A to F. So, its base is 16.
- The A to F alphabets represent 10 to 15 decimal numbers.
- The position of each digit in a hexadecimal number represents a specific power of base (16) of the number system.
- It is also known as alphanumeric number system as it uses both numeric digits and alphabets

Decimal Number	**Hexa Decimal Number**
0	**0**
1	**1**
2	**2**
3	**3**
4	**4**
5	**5**
6	**6**
7	**7**

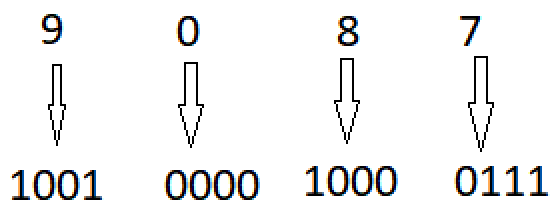
8	**8**
9	**9**
10	**A**
11	**B**
12	**C**

Binary Coded Decimal Number.:

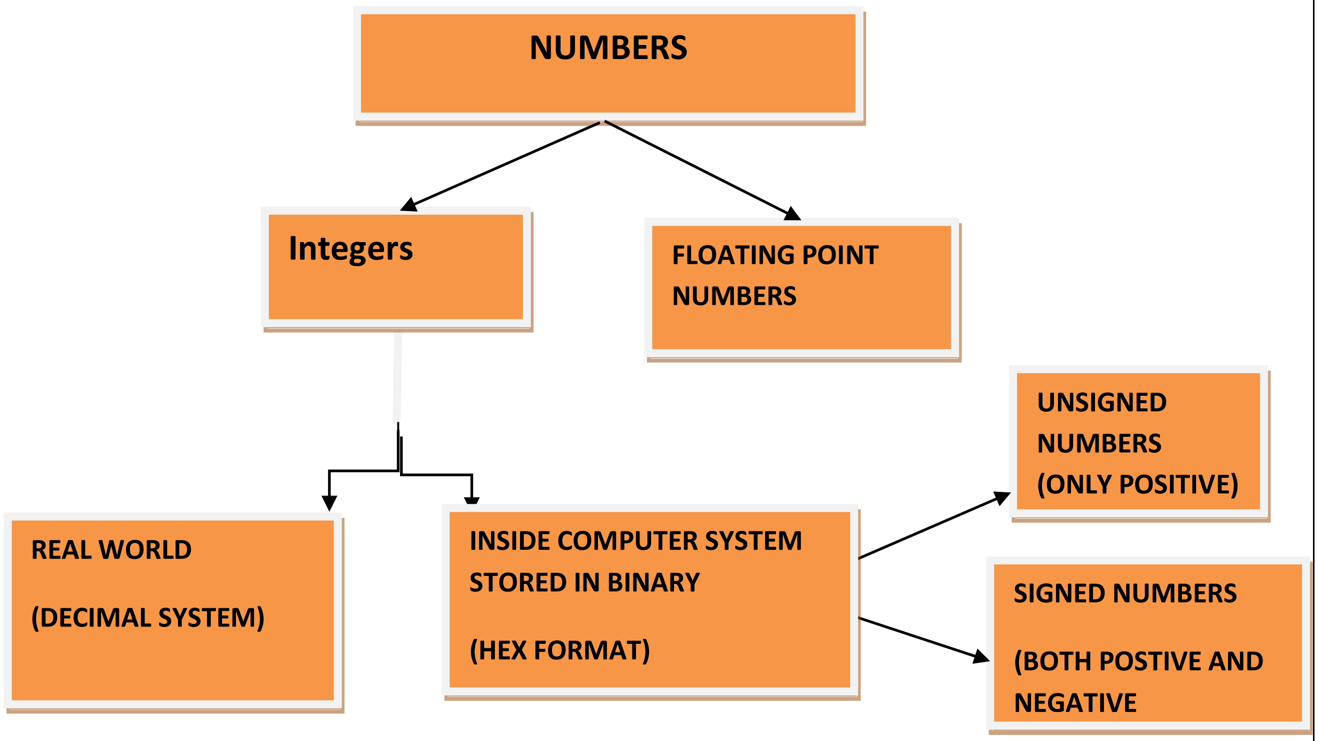
BCD stands for Binary Coded Decimal Number. In BCD code, each digit of the decimal number is represented as its equivalent binary number. So, the LSB and MSB of the decimal numbers are represented as its binary numbers.


Truth Table for Binary Coded Decimal

DECIMAL NUMBER	BCD
0	0000
1	0001
2	0010
3	0011



NUMBER REPRESENTATION:


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

UNSIGNED INTEGERS

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

SIGNED INTEGERS

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

Three systems are used forrepresenting such numbers:

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

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

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

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

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

Fig: Binary signed number Representations

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

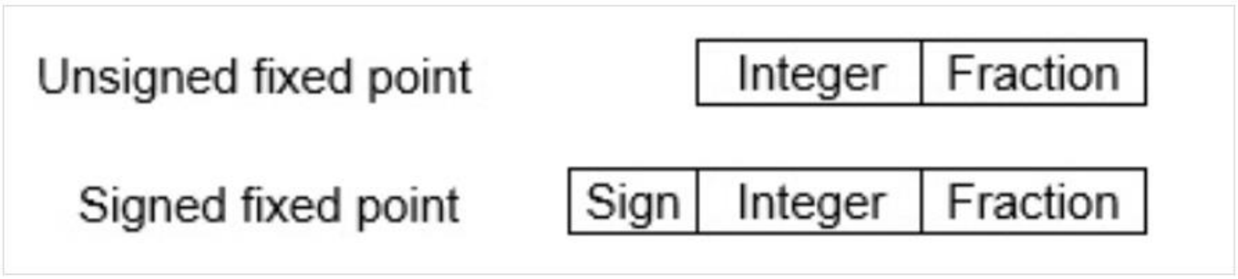
3 Bit Integer	
$2^3 = 8$ therefore 8 combinations	
Unsigned	Signed
0 … 7	-4 … -1 0 1 … 3

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

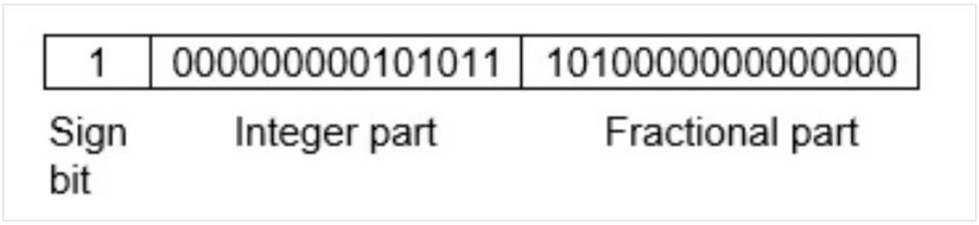
Fixed and Floating point Representations:
There are two major approaches to store real numbers (i.e., numbers withfractional component) in modern computing. These are

(i) Fixed Point Notation and
(ii) Floating Point Notation.

Fixed Point Notation: In **fixed point notation**, there are a fixed number of digits after the decimal point, whereas **floating point number** allows for avarying number of digits after the decimal point.
This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.



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



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

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

Floating Point Representation:

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

NORMALIZATION OF A FLOATING POINTNUMBER:

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

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

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

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

A normalized form of a number is:

-1^s x1.MX2^E

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

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

Advantages of Normalization.

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

SHORT REAL FORMAT / SINGLE PRECISION FORMAT / IEEE 754: 32 BIT FORMAT:

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

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

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

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

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

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

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

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

Overflow:

For a value, 1.0 the normalized form will be

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

Herethe True Exponent is 0.

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

This is because the 8-bit biased exponent cannot hold a value more than 255.Hence, all cases where the TE = 128 or more, the ***BE will be represented as 1111 1111.This indicates as exception (error) called OVERFLOW. The number is called NaN (Not a Number).***It is identified by Exponent being all 1s (1111 1111).Here, the Mantissa can be anything!The **Extreme case of NaN is Infinity**.It is also an OVERFLOW and hence the Exponent will be 1111 1111.To differentiate Infinity from NaN, the Mantissa in infinity is 0000 0000.Hence **Infinity is identified as Exponent all 1s and Mantissa all 0s.**

Suppose the number is 0.1.It will be normalized as

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

The true exponent here is -1.

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

Underflow: All cases where the TE = -127 or less, the BE will be represented as 0000 0000.This indicates as exception (error) called UNDERFLOW.
The number is called De-Normal Number.It is identified by Exponent being all 0s (0000 0000).Here, the Mantissa can be anything.The **Extreme case of De-Normal Number is Zero.**
It is also an UNDERFLOW and hence the Exponent will be 0000 0000.To differentiate Zero from De-Normal Number, the Mantissa in Zero is 0000 0000.Hence **Zero is identified as Exponent all 0s and Mantissa all 0s**.This means Zero is represented as all 0s.

<u>Example:</u>Convert 2A3BH into Short Real format.

Soln: Converting the number into binary we get:
0010 1010 0011 1011
Normalizing the number we get:
$(-1)^0$**x** 1.**0101000111011** x 2^{13}
Here S = 0; M = 0101000111011; True Exponent = 13.
Bias value for Short Real format is 127:
Biased Exponent (BE) = True Exponent + Bias
= 13 + 127
= 140.
Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (1000 1100)

Representing in the required format we get:

0	**10001100**	**010100011101100…**

S Biased Exp Mantissa
(1) (8) (23)

Computer Arithmetic

Integer Addition:

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

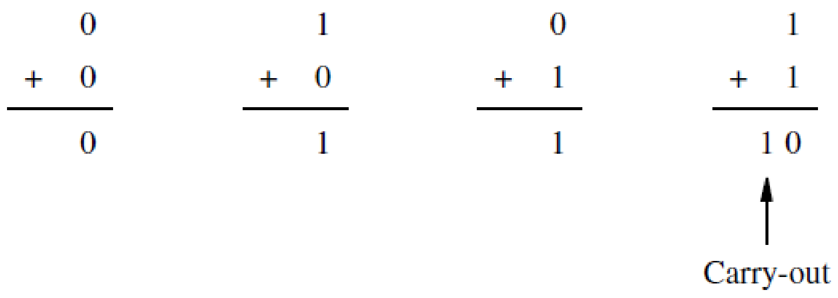


Fig: Addition of 1-bit Numbers

Addition and Subtraction of Signed Integers:

To add two numbers, add their n-bit representations, ignoring the carry-out bit fromthe most significant bit (MSB) position. The sum will be the algebraically correct value in2's-complement representation if the actual result is in the range$-(2^{n-1})$ through$+2^{n-1} - 1$.

To subtract two numbers X and Y, that is, to perform X − Y , form the 2's-complement of Y , then add it to X using the add rule. Again, the result will be the algebraically correct value in 2’s-complement representation if the actual result is in the range $-(2^{n-1})$ through$+2^{n-1}$.

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

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

```
     0  1  1  1
  +  1  1  0  1
  -------------
  1  0  1  0  0
  ↑
  Carry-out
```

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

Few more examples:

```
(a)       0010     (+2)           (b)       0100     (+4)
        + 0011     (+3)                   + 1010     (−6)
        ------     ----                   ------     ----
          0101     (+5)                     1110     (−2)

(c)       1011     (−5)           (d)       0111     (+7)
        + 1110     (−2)                   + 1101     (−3)
        ------     ----                   ------     ----
          1001     (−7)                     0100     (+4)

(e)       1101     (−3)                     1101
        − 1001     (−7)       ⇒           + 0111
        ------     ----                   ------     ----
                                            0100     (+4)
```

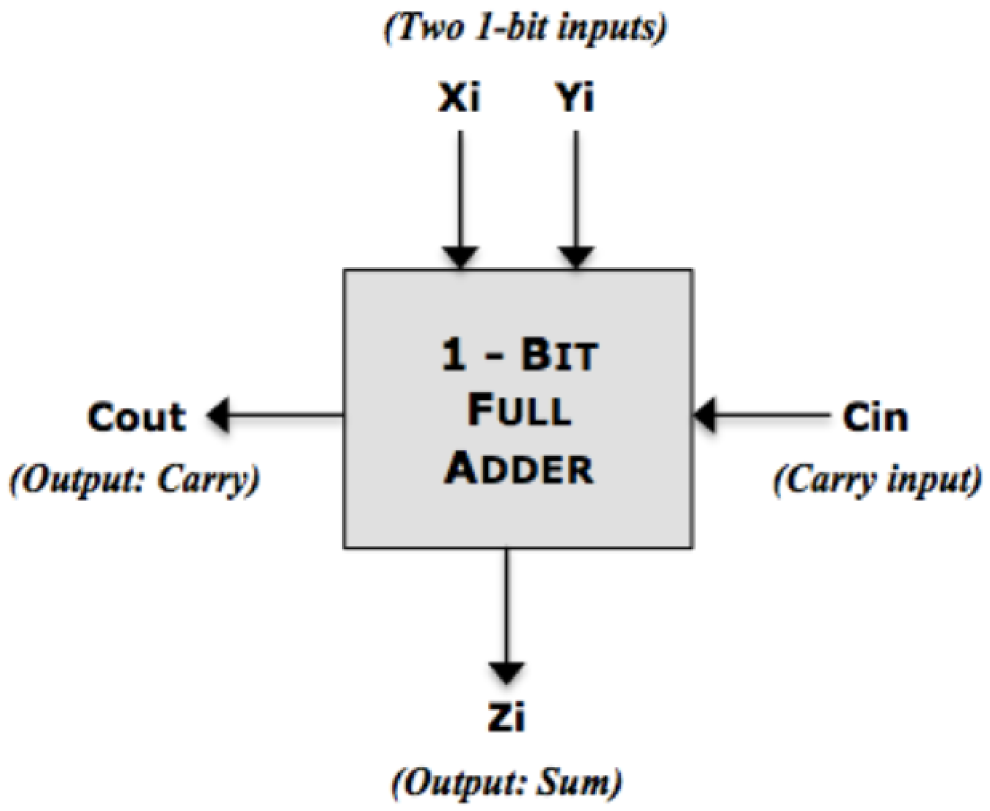
Sign Extension: We often need to represent a value given in a certain number of bits by using a larger number of bits. For a positive number, this is achieved by adding 0s to the left. For a negative number in 2’s-complement representation, the leftmost bit, which indicates the sign of the number, is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

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

Introduction to adder circuits:

ONE BIT ADDITION: FULL ADDER

1) It is a 1-bit adder circuit.
2) It adds two 1-bit inputs Xi & Yi, along with a Carry Input Cin.
3) It produces a sum Zi and a Carry output Cout.
4) As it considers a carry input, it can be used in combination to add large numbers.
5) Hence it is called a Full Adder.



Inputs bits: Xi and Yi.
Input Carry: Cin

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

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

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

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

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

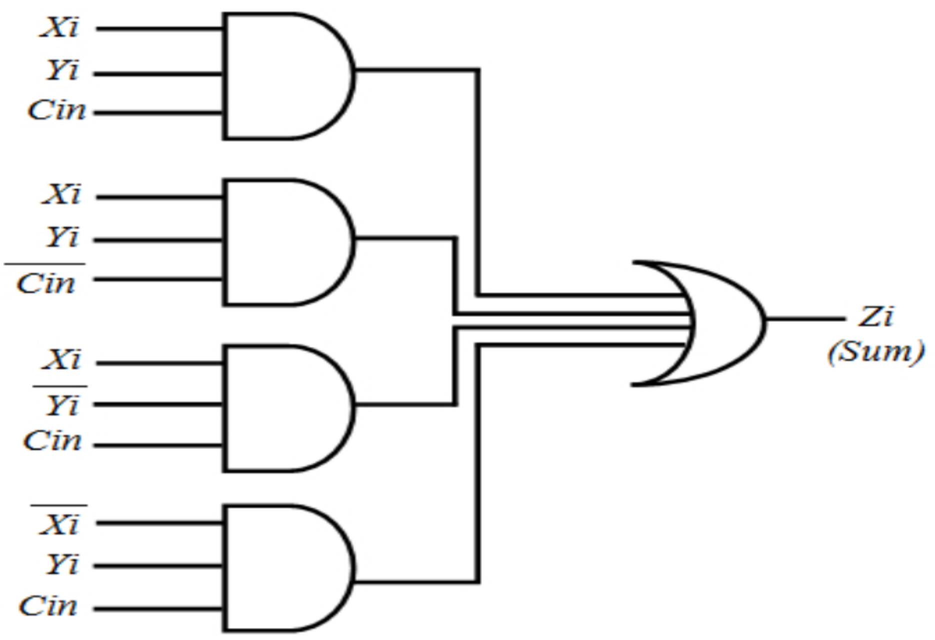


Fig: Circuit for Sum

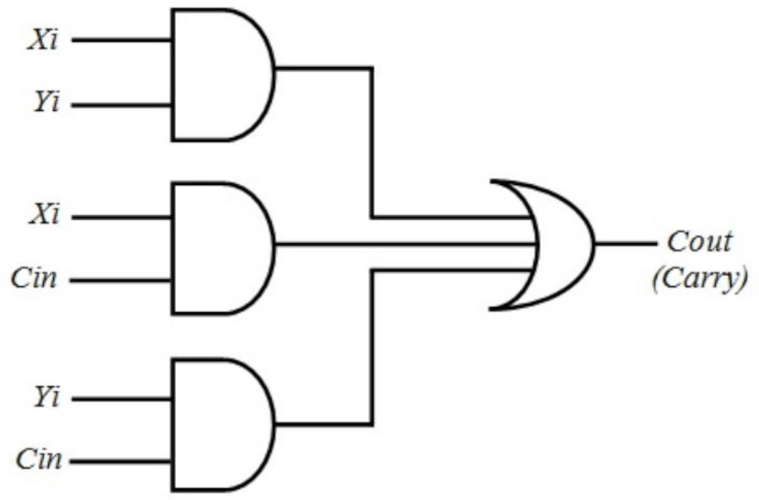


Fig: Circuit for carry

RIPPLE CARRY ADDER (For Multiple bit addition):

1)A Full Adder can add two “1-bit” numbers with a Carry input.
2) It produces a “1-bit” Sum and a Carry output.
3) Combining many of these Full Adders, we can add multiple bits.
4) One such method is called Serial Adder.
5) Here, bits are added one-by-one from Least significant bit(LSB) onwards.
6) The carries are connected in a chain through the full adders. The Carry of each stage is propagated (Rippled) into the next stage.
7) Hence, these adders are also called Ripple Carry Adders.
Advantage: They are very easy to construct.
Drawback: As addition happens bit-by-bit, they are slow.
8) Number of cycles needed for the addition is equal to the number of bits to be added.
Inputs:

Assume X and Y are two “4-bit” numbers to be added, along with a Carry input CIN.
X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB)
Y = Y0 Y1 Y2 Y3 (Y0 is the MSB … Y3 is the LSB)
CIN = Carry Input
Outputs:
Assume Z to be a “4-bit” output, and COUT to be the output Carry
Z = Z0 Z1 Z2 Z3 (Z0 is the MSB … Z3 is the LSB)(Here Z represents the sum)
COUT = Carry Output

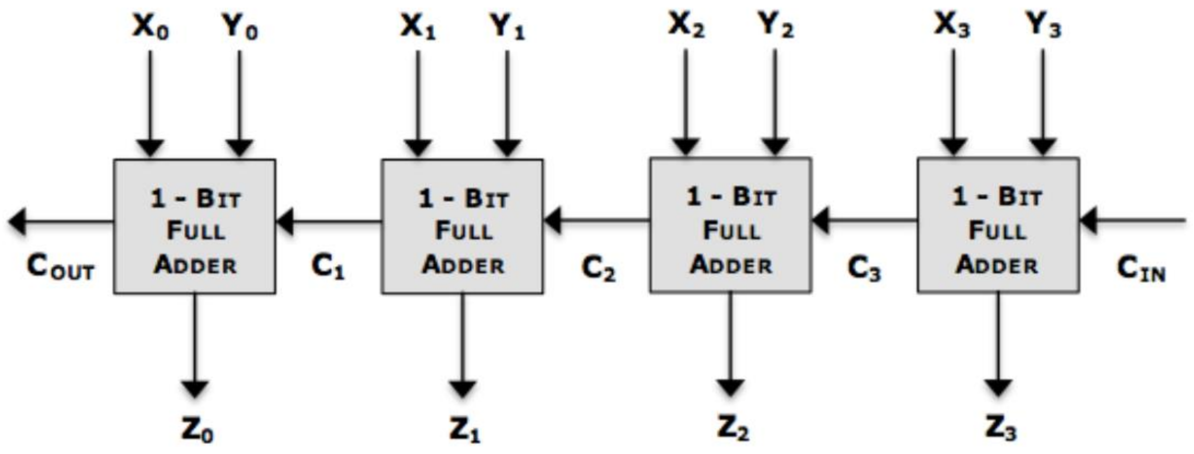


Fig:4-bit Ripple Carry Adder

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

1) This is also called as parallel adder. It is used to add multiple bits simultaneously.
2) While adding multiple bits, the main issue is that of the intermediate carries.
3) In Serial Adders, we therefore added the bits one-by-one.
4) This allowed the carry at any stage to propagate to the next stage.
5) But this also made the process very slow.
6) If we “PREDICT” the intermediate carries, then all bits can be added simultaneously.
7) This is done by the Carry Look Ahead Generator Circuit.
8) Once all carries are determined beforehand, then all bits can be added simultaneously.
Advantage: This makes the addition process extremely fast.
Drawback: Circuit is complex.
Inputs:
Assume X and Y are two “4-bit” numbers to be added, along with a Carry input CIN.
X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB); Y = Y0 Y1 Y2 Y3 & CIN = Carry Input
Outputs:
Assume Z to be a “4-bit” output, and C_{OUT} to be the output Carry
Z = Z0 Z1 Z2 Z3 & C_{OUT} = Carry Output

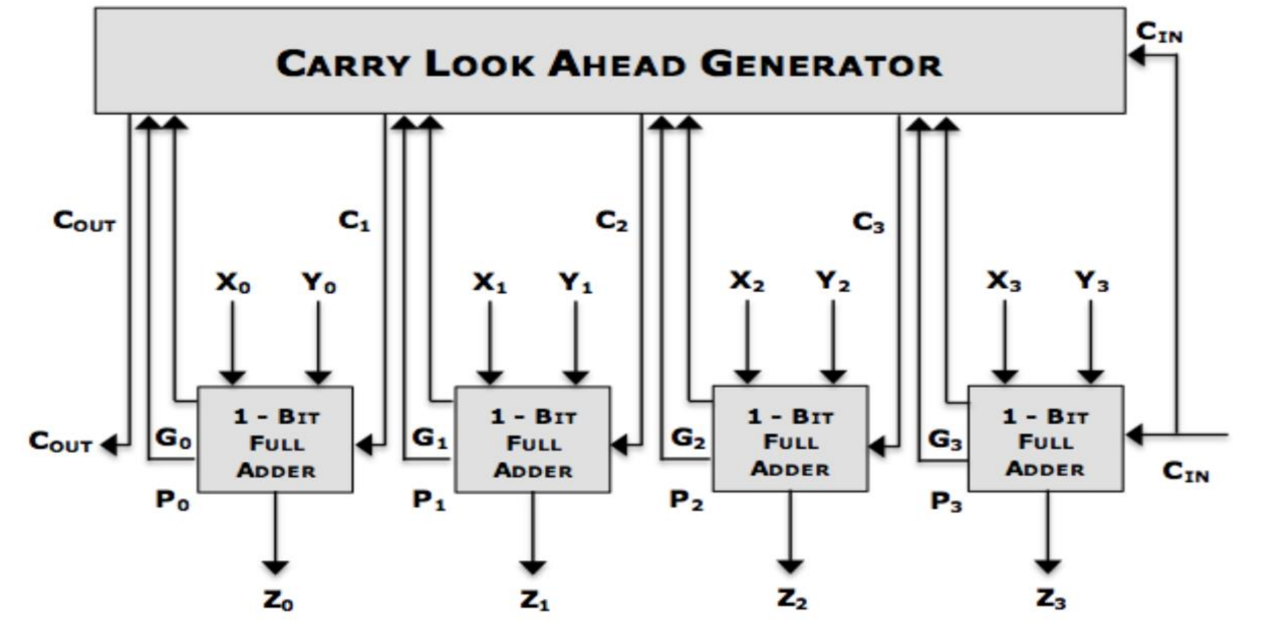


Fig: Circuit for Carry Look ahead Adder

We can "Predict" (Look Ahead) all the intermediate carries in the followingmanner: The carry at any stage can be calculated as:

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

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

Here $G_i = X_i.Y_i$ ... (Generate)
And $P_i = X_i+Y_i$ ... (Propagate)

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

C3 = G3 + P3CIN (I)
C2 = G2 + P2C3
Substituting the value of C3, we get:
C2 = G2 + P2G3 + P2P3CIN (II)
C1 = G1 + P1C2
Substituting the value of C2, we get:
C1 = G1 + P1G2 + P1P2G3 + P1P2P3CIN (III)
C0 = G0 + P0C1
Substituting the value of C1, we get:
C0 = G0 + P0G1 + P0P1G2 + P0P1P2G3 + P0P1P2P3CIN (IV)

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

Cycle 1: g_1, p_1, g_2, p_2, g_3, p_3, g_0, $p_0$are given to the carry look ahead generator.

Cycle 2: Input carries are given to the adders by the carry generator.
Cycle 3: Results are produced.
Total number of cycles required :3

Multiplication:

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

<u>Algorithm:</u>
The **number of steps** required is equal to the **number of bits in the multiplier**.
1) At each step, **examine** the current **multiplier bit** starting from the **LSB**.
2) If the current **multiplier bit is "1"**, then the **Partial-Product** is the **Multiplicand** itself.
3) If the current **multiplier bit is "0"**, then the **Partial-Product** is the **Zero**.
4) At each step, **ADD the Partial-Product to the Accumulator**.
5) Now **Right-Shift the Result** produced so far (**A & Q combined**).
Repeat steps 1 to 5 for **all bits** of the multiplier.
The **final answer** will be in **A & Q** combined.

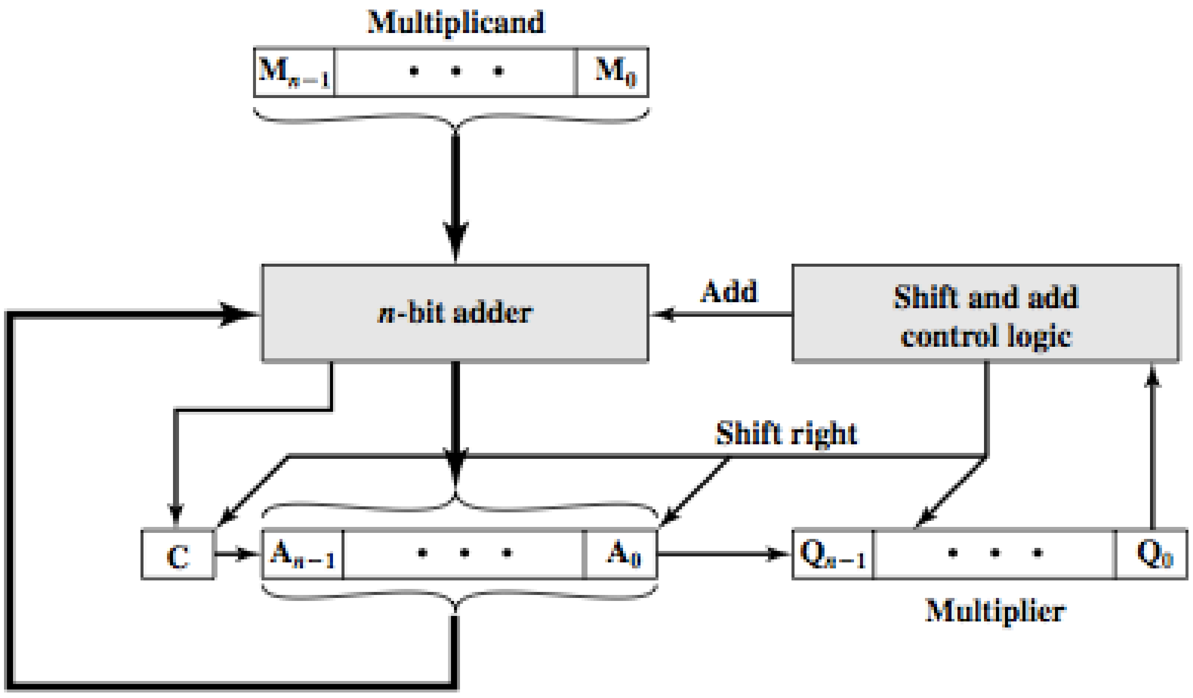


Fig: Shift and Add Multiplication

Example: Let us consider <u>7X6</u>

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

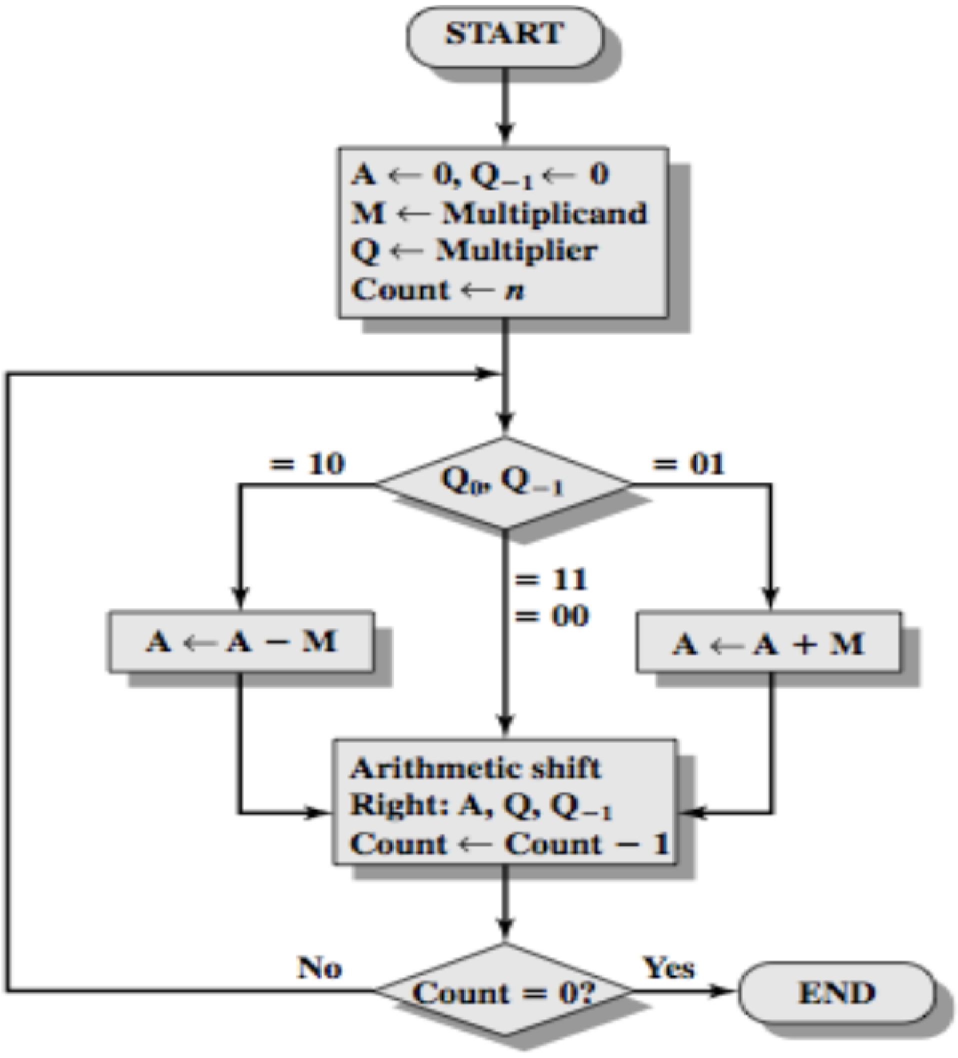
Step	C Carry	A Accumulator	Q Multiplier	M Multiplicand	Explanation
	0	0000	0110	0111	*Initial Value*
1	**0** **0**	0000 0000	0110 0011		*Current Multiplier bit is "0" so ADD "0" to Accumulator and Right-Shift*
2	0 0	0111 0011	0011 1001		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
3	0 0	1010 0101	1001 0100		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
4	0 0	0101 ***0010***	0100 ***1010***		*Current Multiplier bit is "0" so ADD "0" to Accumulator and Right-Shift*

2) Booth Multiplier(For signed Multiplication):

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

Algorithm:
The **number of steps** required is equal to the **number of bits in the multiplier**.
At the beginning, consider an **imaginary "0" beyond LSB of Multiplier**
1) At each step, **examine two adjacent Multiplier bits** from **Right to Left**.
2) If the transition is from **"0 to 1"** then **Subtract M** from **A** and **Right-Shift** (A & Q) combined.
3) If the transition is from **"1 to 0"** then **ADD M** to **A** and **Right-Shift**.
4) If the transition is from **"0 to 0"** then **simply Right-Shift**.
5) If the transition is from **"1 to 1"** then **simply Right-Shift**.
Repeat steps 1 to 5 for **all bits** of the multiplier.
The **final answer** will be in **A & Q** combined.

<u>**Flowchart for Booth's Algorithm:**</u>



Example: -9x10=-90
Multiplicand (M): **-9 = 10111**   **9 = 01001**. (Two's Complement Form)
Multiplier (Q): **10 = 01010**.   **-10 = 10110** (Two's Complement Form)

step	**A Accumulator**	**Q Multiplier**	**Q(-1)**	**M Multiplicand**
Initial	**00000**	**01010**	**0**	**10111**
1) (0 ç 0) No Add or Sub Right-Shift	**00000** **00000**	**01010** **00101**	**0** **0**	
2) (1 ç 0) Perform **(A - M)** Right-Shift	**01001** **00100**	**00101** **10010**	**0** **1**	
3) (0 ç 1) Perform **(A + M)** Right-Shift	**11011** **11101**	**10010** **11001**	**1** **0**	
4) (1 ç 0) Perform **(A - M)** Right-Shift	**00110** **00011**	**11001** **01100**	**0** **1**	
5) (0 ç 1) Perform **(A + M)** Right-Shift	**11010** **11101**	**01100** **00110**	**1** **0**	

Restoring and Non-Restoring Division:

Non Restoring Division:

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

Algorithm:
The number of steps required is equal to the number of bits in the Dividend.
1) At each step, left shift the dividend by 1 position.
2) Subtract the divisor from A (perform A - M).
3) If the result is positive then the step is said to be “Successful”. In this case quotient bit will be “1” and Restoration is NOT Required. The Next Step will also be Subtraction.
4) If the result is negative then the step is said to be “Unsuccessful”. In this case quotient bit will be “0”. Here Restoration is NOT Performed. Instead, the next step will be ADDITION in place of subtraction.
As restoration is not performed, the method is called Non-Restoring Division.
Repeat steps 1 to 4 for all bits of the Dividend.

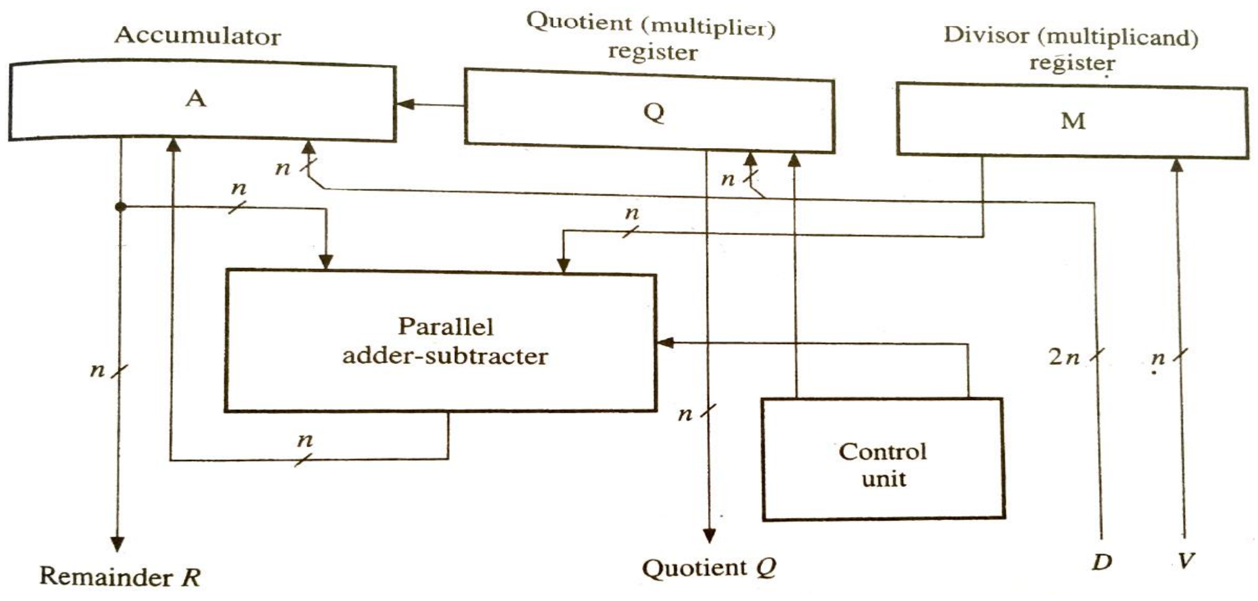
Example: (7) / (5)
Dividend (Q) = 7
Divisor (M) = 5
Accumulator (A) = 0
7 = 0111 **5** = 0101
-7 = 1001**-5** = 1011

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

RESTORING DIVISION (For unsigned Numbers)

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

Algorithm:
The number of steps required is equal to the number of bits in the Dividend.
1) At each step, left shift the dividend by 1 position.
2) Subtract the divisor from A (perform A - M).
3) If the result is positive then the step is said to be “Successful”.In this case quotient bit will be “1” and Restoration is NOT Required.
4) If the result is negative then the step is said to be “Unsuccessful”.In this case quotient bit will be “0”.Here Restoration is performed by adding back the divisor.
Hence the method is called Restoring Division.Repeat steps 1 to 4 for all bits of the Dividend.


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

Example: (6) / (4)
Dividend (Q) = 6
Divisor (M) = 4
Accumulator (A) = 0
6 = 0110 4 = 0100
-6 = 1010 -4 = 1100

	Accumulator A(0)	**Dividend Q(6)**	**Divisor M(4)**
Initial Values	0000	0110	0100
Step 1: Left shift A-M Unsuccessful(-ve) Restoration:	0000 + 1100 **1100** 0000	110_ 1100	
Step 2: Left shift A-M Unsuccessful(-ve) Restoration:	0001 +1100 1101 0001	100_ 1000	
Step 3: Left shift A-M Unsuccessful(-ve) Restoration:	0011 +1100 1111 0011	000_ 0000	
Step 3: Left shift A-M Successful(+ve) No Restoration	0110 +1100 0010	000_ 0001	
	Remainder(2)	Quotient(1)	

RESTORING DIVISION FOR SIGNED NUMBERS:

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

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

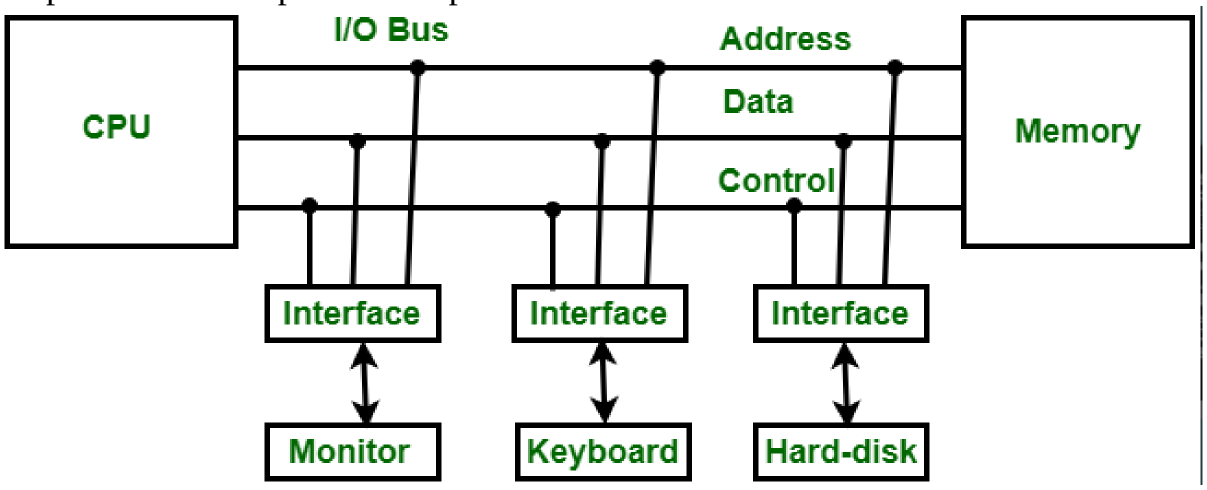
Example: (-19) / (7)
19 = 010011 7 = 000111
-19 = 101101 -7 = 111001

	Accumulator A(Sign Extension)	**Dividend Q(-19)**	**Divisor M(7)**
Initial Values	111111	101101	000111
Step 1: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111111 + 000111 000110 111111	01101_ 011010	
Step 2: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111110 + 000111 000101 111110	11010_ 110100	
Step 3: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111101 + 000111 000100 111101	10100_ 101000	
Step 4: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111011 + 000111 000010 111011	01000_ 010000	
Step 5: Left-shift Sign(A,M) Different: A+M Sign still same: Successful Restoration not required	110110 + 000111 111101 111101	10000_ 100001	
Step 6: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111011 + 000111 000010 111011	00001_ 000010	
	Remainder(-5)	Quotient(2)	

UNIT – IV

Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt Direct memory Access.
Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

Input-Output Interface is used as a method which helps in transferring of information between the internal storage devices i.e. memory and the external peripheral device. A peripheral device is that which provide input and output for the computer, it is also called Input-Output devices. For Example: A keyboard and mouse provide Input to the computer are called input devices while a monitor and printer that provide output to the computer are called output devices. Just like the external hard-drives, there is also availability of some peripheral devices which are able to provide both input and output.



Input-Output Interface

In micro-computer base system, the only purpose of peripheral devices is just to provide **special communication links** for the interfacing them with the CPU. To resolve the differences between peripheral devices and CPU, there is a special need for communication links.
The major differences are as follows:
1. The nature of peripheral devices is electromagnetic and electro-mechanical. The nature of the CPU is electronic. There is a lot of difference in the mode of operation of both peripheral devices and CPU.
2. There is also a synchronization mechanism because the data transfer rate of peripheral devices are slow than CPU.
3. In peripheral devices, data code and formats are differ from the format in the CPU and memory.

4. The operating mode of peripheral devices are different and each may be controlled so as not to disturb the operation of other peripheral devices connected to CPU.

There is a special need of the additional hardware to resolve the differences between CPU and peripheral devices to supervise and synchronize all input and output devices.

Functions of Input-Output Interface:

1. It is used to synchronize the operating speed of CPU with respect to input-output devices.
2. It selects the input-output device which is appropriate for the interpretation of the input-output signal.
3. It is capable of providing signals like control and timing signals.
4. In this data buffering can be possible through data bus.
5. There are various error detectors.
6. It converts serial data into parallel data and vice-versa.
7. It also convert digital data into analog signal and vice-versa.

Asynchronous data transfer:

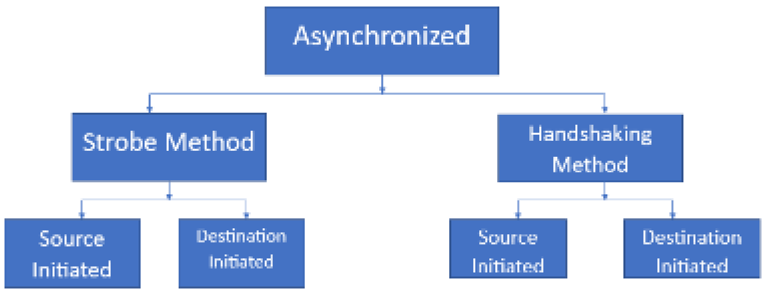
Asynchronous data transfer enable computers to send and receive data without having to wait for a real-time response. With this technique data is conveyed in discrete units known as packets that may be handled separately. This article will explain what asynchronous data transfer is, its primary terminologies, advantages and disadvantages, and some frequently asked questions.

Terminologies used in Asynchronous Data Transfer

- **Sender**: The machine or gadget that transfers the data.
- **Receiver**: A device or computer that receives data.
- **Packet**: A discrete unit of transmitted and received data.
- **Buffer**: A short-term location for storing incoming or departing data.

Classification of Asynchronous Data Transfer

- **Strobe Control Method**
- **Handshaking Method**



Classification of Asynchronous Data Transfer

Strobe Control Method For Data Transfer

Strobe control is a method used in asynchronous data transfer that synchronizes data flow between two devices. Bits are transmitted one at a time, independently of one another, and without the aid of a clock signal in asynchronous communication. To properly receive the data, the receiving equipment needs to be able to synchronize with the transmitting device.

Strobe control involves sending data along with a different signal known as the strobe signal. The strobe signal alerts the receiving device that the data is valid and ready to be read. The receiving device waits for the strobe signal before reading the data to ensure sure it is synchronized with its clock.

The strobe signal is usually generated by the transmitting device and is sent either before or after the data. If the strobe signal is sent before the data, it is called a leading strobe. If it is sent after the data, it is called a trailing strobe.

Handshaking Method for Data Transfer

The strobe method has a limitation in that the initiating source unit cannot confirm whether the destination unit has received the data placed on the bus. Similarly, the destination unit cannot verify if the source has placed data on the bus. This issue is resolved by the handshaking method which introduces a second control signal line to confirm the transfer between units.

In this method, one control line follows the data flow from the source to the destination and allow the source to inform the destination whether valid data is on the bus. The other control line runs in the opposite direction from the destination to the source and enable the destination to notify the source about its ability to accept data. The control sequence depends on which unit initiates the transfer as the process varies based on whether the source or destination is initiating the exchange.

Priority interrupts:

1. Priority interrupts allow for the efficient handling of high-priority tasks that require immediate attention. This is especially important in real-time systems where certain tasks must be completed within strict time constraints.

2. They are more efficient than software polling as the processor does not waste time constantly checking for events that have not occurred.

3. Priority interrupts are also more deterministic, as the response time to an event can be accurately predicted based on its priority level.

Disadvantages:

1. One potential disadvantage of priority interrupts is the possibility of lower priority tasks being starved of resources if high-priority tasks are continuously

interrupting the processor.

2. If not implemented properly, priority interrupts can lead to priority inversion, where a low-priority task holds a resource required by a higher-priority task, causing a delay in the high-priority task's execution.

Memory Hierarchy Design

1. Registers

Registers are small, high-speed memory units located in the CPU. They are used to store the most frequently used data and instructions. Registers have the fastest access time and the smallest storage capacity, typically ranging from 16 to 64 bits.

2. Cache Memory

Cache memory is a small, fast memory unit located close to the CPU. It stores frequently used data and instructions that have been recently accessed from the main memory. Cache memory is designed to minimize the time it takes to access data by providing the CPU with quick access to frequently used data.

3. Main Memory

Main memory, also known as RAM (Random Access Memory), is the primary memory of a computer system. It has a larger storage capacity than cache memory, but it is slower. Main memory is used to store data and instructions that are currently in use by the CPU.

Types of Main Memory

- **Static RAM:** Static RAM stores the binary information in flip flops and information remains valid until power is supplied. Static RAM has a faster access time and is used in implementing cache memory.
- **Dynamic RAM:** It stores the binary information as a charge on the capacitor. It requires refreshing circuitry to maintain the charge on the capacitors after a few milliseconds. It contains more memory cells per unit area as compared to SRAM.

read more about - Different Types of RAM (Random Access Memory)

4. Secondary Storage

Secondary storage, such as hard disk drives (HDD) and solid-state drives (SSD) , is a non-volatile memory unit that has a larger storage capacity than main memory. It is used to store data and instructions that are not currently in use by the CPU. Secondary storage has the slowest access time and is typically the least expensive type of memory in the memory hierarchy.

5. Magnetic Disk

Magnetic Disks are simply circular plates that are fabricated with either a metal or a plastic or a magnetized material. The Magnetic disks work at a high speed inside the computer and these are frequently used.

6. Magnetic Tape

Magnetic Tape is simply a magnetic recording device that is covered with a plastic film. Magnetic Tape is generally used for the backup of data. In the case of a magnetic tape, the access time for a computer is a little slower and therefore, it requires some amount of time for accessing the strip.

Main Memory :

The main memory is the fundamental storage unit in a computer system. It is associatively large and quick memory and saves programs and information during computer operations. The technology that makes the main memory work is based on semiconductor integrated circuits.

RAM is the main memory. Integrated circuit Random Access Memory (RAM) chips are applicable in two possible operating modes are as follows ?

- **Static** ? It consists of internal flip-flops, which store the binary information. The stored data remains solid considering power is provided to the unit. The static RAM is simple to use and has smaller read and write cycles.

- **Dynamic** ? It saves the binary data in the structure of electric charges that are used to capacitors. The capacitors are made available inside the chip by Metal Oxide Semiconductor (MOS) transistors. The stored value on the capacitors contributes to discharge with time and thus, the capacitors should be regularly recharged through stimulating the dynamic memory.

The Secondary storage media can be fixed or removable. Fixed Storage media is an internal storage medium like a hard disk that is fixed inside the computer. A storage medium that is portable and can be taken outside the computer is termed removable storage media.

Secondary memory is a type of computer memory that is used for long-term storage of data and programs. It is also known as auxiliary memory or external memory, and is distinct from primary memory, which is used for short-term storage of data and instructions that are currently being processed by the CPU.

Secondary memory devices are typically larger and slower than primary memory, but offer a much larger storage capacity. This makes them ideal for storing large files such as documents, images, videos, and other multimedia content.

Some examples of secondary memory devices include hard disk drives (HDDs), solid-state drives (SSDs), magnetic tapes, optical discs such as CDs and DVDs, and flash memory such as USB drives and memory cards. Each of these devices uses different technologies to store data, but they all share the common feature of being non-volatile, meaning that they can store data even when the computer is turned off.

Secondary memory devices are accessed by the CPU via input/output (I/O) operations, which involve transferring data between the device and primary memory. The speed of these operations is affected by factors such as the type of device, the size of the file being accessed, and the type of connection between the device and the computer.

ASSOCIATIVE MEMORY:

Associative memory is also known as content addressable memory (CAM) or associative storage or associative array. It is a special type of memory that is optimized for performing searches through data, as opposed to providing a simple direct access to the data based on the address.
It can store the set of patterns as memories when the associative memory is being presented with a key pattern, it responds by producing one of the stored pattern which closely resembles or relates to the key pattern.
It can be viewed as **data correlation** here. input data is correlated with that of stored data in the CAM.
It forms of two type:

1. auto associative memory network : An auto-associative memory network, also known as a recurrent neural network, is a type of associative memory that is used to recall a pattern from partial or degraded inputs. In an auto-associative network, the output of the network is fed back into the input, allowing the network to learn and remember the patterns it has been trained on. This type of memory network is commonly used in applications such as speech and image recognition, where the input data may be incomplete or noisy.
2. hetero associative memory network : A hetero-associative memory network is a type of associative memory that is used to associate one set of patterns with another. In a hetero-associative network, the input pattern is associated with a different output pattern, allowing the network to learn and remember the associations between the two sets of patterns. This type of memory network is commonly used in applications such as data compression and data retrieval.

Cache memory is a small, high-speed storage area in a computer. The cache is a smaller and faster memory that stores copies of the data from frequently used main memory locations. There are various independent caches in a CPU, which store instructions and data.

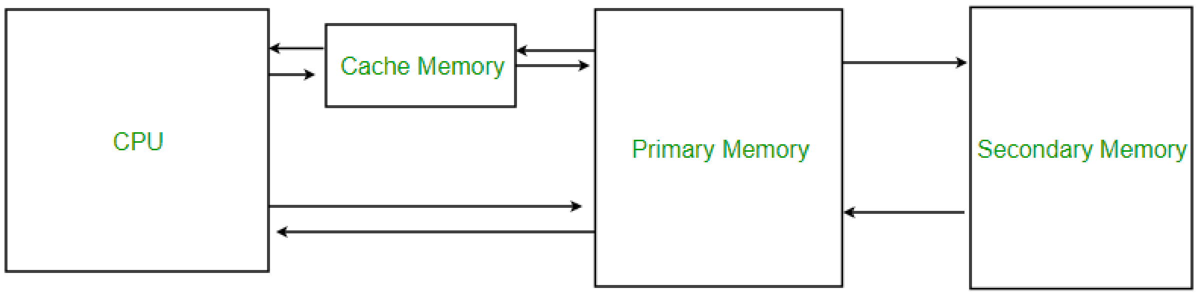
- The most important use of cache memory is that it is used to reduce the average time to access data from the main memory.
- The concept of cache works because there exists locality of reference (the same items or nearby items are more likely to be accessed next) in processes.

By storing this information closer to the CPU, cache memory helps speed up the overall processing time. Cache memory is much faster than the main memory

(RAM). When the CPU needs data, it first checks the cache. If the data is there, the CPU can access it quickly. If not, it must fetch the data from the slower main memory.

Characteristics of Cache Memory

- Extremely fast memory type that acts as a buffer between RAM and the CPU.
- Holds frequently requested data and instructions, ensuring that they are immediately available to the CPU when needed.
- Costlier than main memory or disk memory but more economical than CPU registers.
- Used to speed up processing and synchronize with the high-speed CPU.



UNIT - V

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics.

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline,

Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor.

Multi Processors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, Cache Coherence

---

CISC Characteristics, RISC Characteristics:

RISC is the way to make hardware simpler whereas CISC is the single instruction that handles multiple work. In this article, we are going to discuss RISC and CISC in detail as well as the Difference between RISC and CISC, Let's proceed with RISC first.

Reduced Instruction Set Architecture (RISC)

The main idea behind this is to simplify hardware by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, a store command will store the data.

Characteristics of RISC

- Simpler instruction, hence simple instruction decoding.
- Instruction comes undersize of one word.
- Instruction takes a single clock cycle to get executed.
- More general-purpose registers.
- Simple Addressing Modes.
- Fewer Data types.
- A pipeline can be achieved.

Advantages of RISC

- **Simpler instructions:** RISC processors use a smaller set of simple instructions, which makes them easier to decode and execute quickly. This results in faster processing times.
- **Faster execution:** Because RISC processors have a simpler instruction set, they can execute instructions faster than CISC processors.
- **Lower power consumption:** RISC processors consume less power than CISC processors, making them ideal for portable devices.

Disadvantages of RISC

- **More instructions required:** RISC processors require more instructions to perform complex tasks than CISC processors.

- **Increased memory usage:** RISC processors require more memory to store the additional instructions needed to perform complex tasks.
- **Higher cost:** Developing and manufacturing RISC processors can be more expensive than CISC processors.

Complex Instruction Set Architecture (CISC)

The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it's complex.

Characteristics of CISC

- Complex instruction, hence complex instruction decoding.
- Instructions are larger than one-word size.
- Instruction may take more than a single clock cycle to get executed.
- Less number of general-purpose registers as operations get performed in memory itself.
- Complex Addressing Modes.
- More Data types.

Advantages of CISC

- **Reduced code size:** CISC processors use complex instructions that can perform multiple operations, reducing the amount of code needed to perform a task.
- **More memory efficient:** Because CISC instructions are more complex, they require fewer instructions to perform complex tasks, which can result in more memory-efficient code.
- **Widely used:** CISC processors have been in use for a longer time than RISC processors, so they have a larger user base and more available software.

Disadvantages of CISC

- **Slower execution:** CISC processors take longer to execute instructions because they have more complex instructions and need more time to decode them.
- **More complex design:** CISC processors have more complex instruction sets, which makes them more difficult to design and manufacture.
- **Higher power consumption:** CISC processors consume more power than RISC processors because of their more complex instruction sets.

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 dictatedbythewaythetaskispartitioned.Theresultobtainedfromthecomputationineachsegment 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 thecombinational circuitperformsthesuboperationintheparticularsegment.Theoutputofthecombinationalcircuit in a given segment is applied to the input register of the next segment. The pipeline organization willbedemonstratedbymeansofasimpleexample.Supposethatwewanttoperformthecombined 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 tworegistersandacombinationalcircuitasshowninFigurebelow.*R*1through*R*5areregistersthat 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:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	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 *A*1 and *B*1 into *R*1 and *R*2.

Figure Example or pipeline processing.

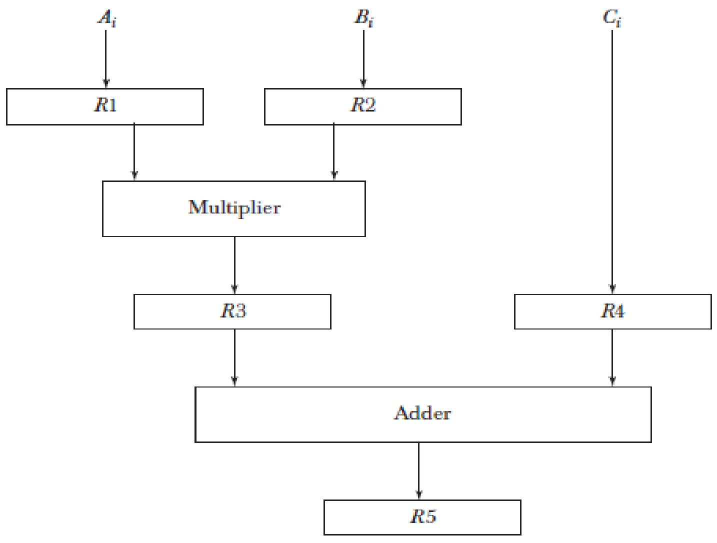


TABLE Content of Registers in Pipeline Example

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	–	–	–	–	$A_7 * B_7 + C_7$

ThesecondclockpulsetransferstheproductofR1andR2intoR3andC1intoR4.Thesameclock pulse transfers *A*2 and *B*2 into *R*1 and *R*2. The third clock pulse operates on all three segments simultaneously. It places *A*3 and *B*3 into *R*1 and *R*2, transfers the product of *R*1 and *R*2 into *R*3, transfersC2 into *R*4, and places the sum of *R*3 and *R*4 into *R*5. It takes three clock pulses to fill up thepipeandretrievethefirstoutputfrom*R*5.Fromthe each clock produce same 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 sub operation over the data stream flowing through the pipe. The segments are separated by registers *Ri* that hold the intermediate results between the stages.

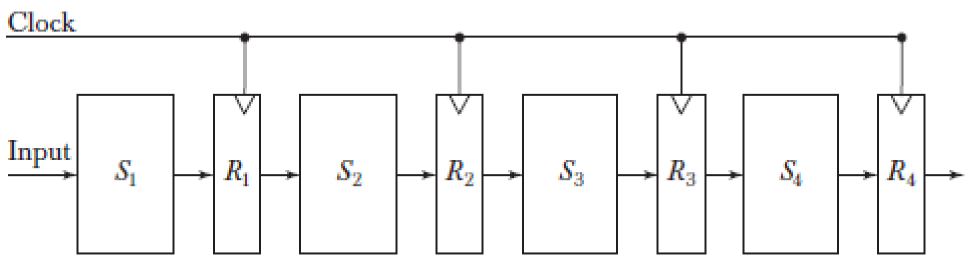


Figure Four-segment pipeline.

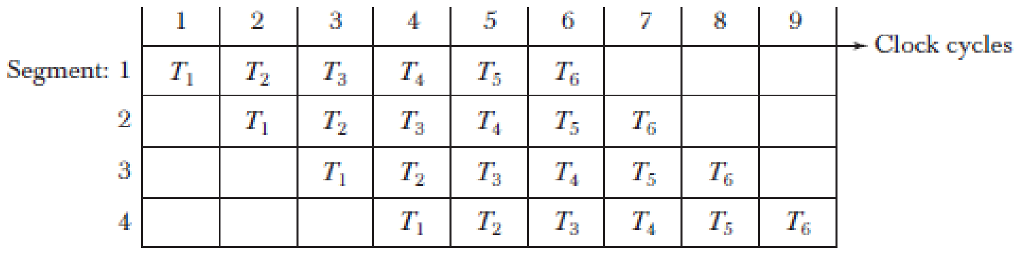
The behavior of a pipeline can be illustrated with a *space-time* diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Figure below. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks *T*1through *T*6 executed in four segments. Initially, task *T*1 is handled by segment 1. After the first clock, segment 2 is busy with *T*1, while segment 1 is busy with task *T*2. Continuing in this manner, the first task *T*1 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 *T*1 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 tim eequalto(*n*-1)*tp*. Therefore,to complete *n*t asks

Using*ak*-segmentpipelinerequires*k*+(*n*-1)clockcycles.Forexample,thediagrambelowshowsfoursegmentsandsixtasks.Thetimerequiredtocompletealltheoperationsis4+(6-1)=9clock 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*.Thespeedupofapipelineprocessingoveranequivalentnonpipelinedprocessingisdefinedby 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
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

Clock cycles

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

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

Ifweassumethatthetimeittakestoprocessataskisthesameinthepipelineandnon-pipeline circuits, we will have *tn*= *ktp*. Including this assumption, the speedup reduces to

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

Thisshowsthatthetheoreticalmaximumspeedupthatapipelinecanprovide is *k*,where *k*i sthe 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 registerslabelled*R*areplacedbetweenthesegmentstostoreintermediateresults.Thesuboperations 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 sub operations perform 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 inthefirstsegmenttoobtain3-2=1.Thelarger exponent3is chosen as the exponent to f the result. Then exist 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 mantissa sunder 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 results it has a fraction with a non zero first digit. Thisisdonebyshiftingthemantissaoncetotherightandincrementingtheexponentbyoneto obtain the normalized sum.

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

Thecomparatorshifter,adder-subtractor,incrementer,anddecrementerinthefloating-point pipeline are implemented with combinational circuits.

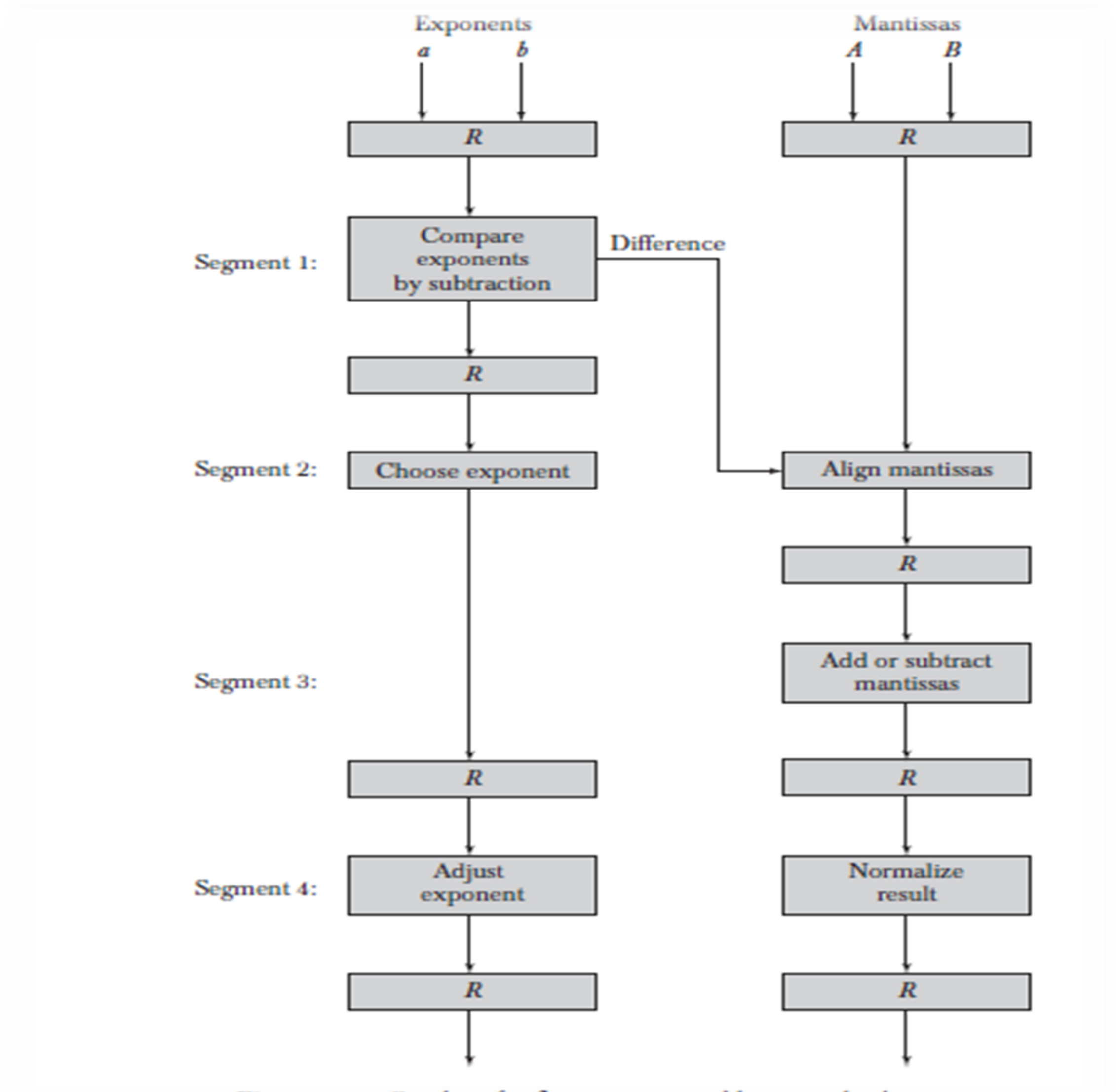


Figure Pipeline for floating-point addition and subtraction.

InstructionPipeline

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 aninstructionmaycauseabranchoutofsequence.Inthatcasethepipelinemustbeemptiedandall 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 fetch from memoryinsegment3.Theeffectiveaddressmaybecalculatedin 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 sub operations 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 operationsinthelasttwosegmentsarecompletedandallinformationstoredintheinstructionbuffer isdeleted.Thepipelinethenrestartsfromthenewaddressstoredintheprogramcounter.Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

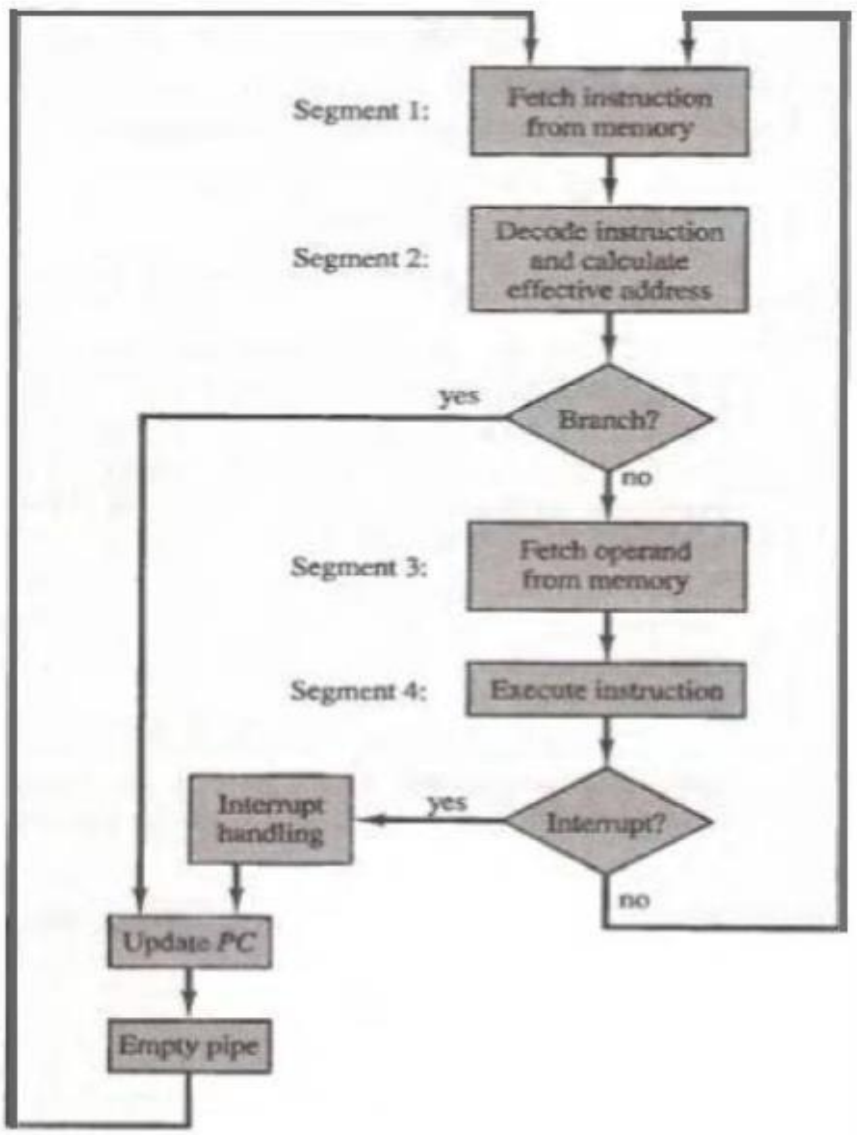
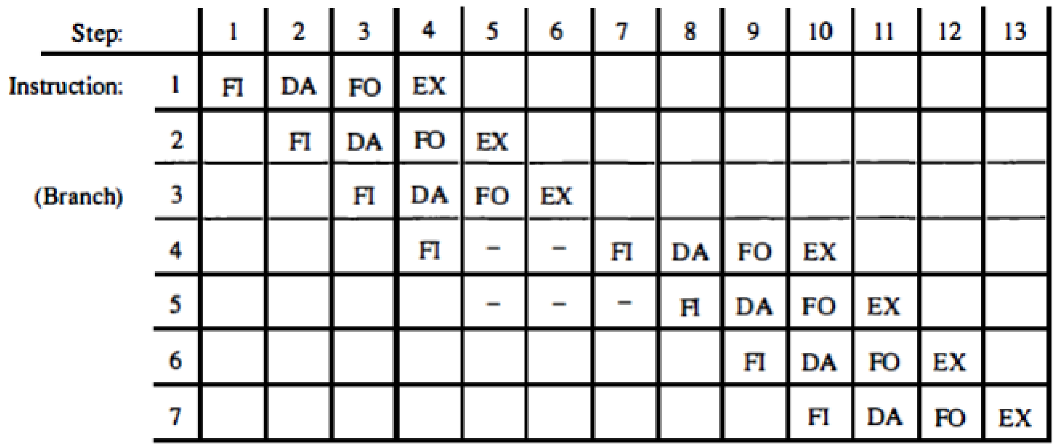


Figure Four-segment CPU pipeline.

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. Flisth 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.

ItisassumedthattheprocessorhasseparateinstructionanddatamemoriessothattheoperationinFla ndFOcanproceedatthesametime.Intheabsenceofabranchinstruction,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 no wthat 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, anew instruction is fetched In step7. Ifthebranch 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 theEX 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.

[Thepurposeofparallelprocessingistospeedupthecomputerprocessingcapabilityandincreaseits throughput,thatis,theamountofprocessingthatcanbeaccomplishedduringagivenintervalof time.]

Parallel Processors

Introduction to parallel processors:

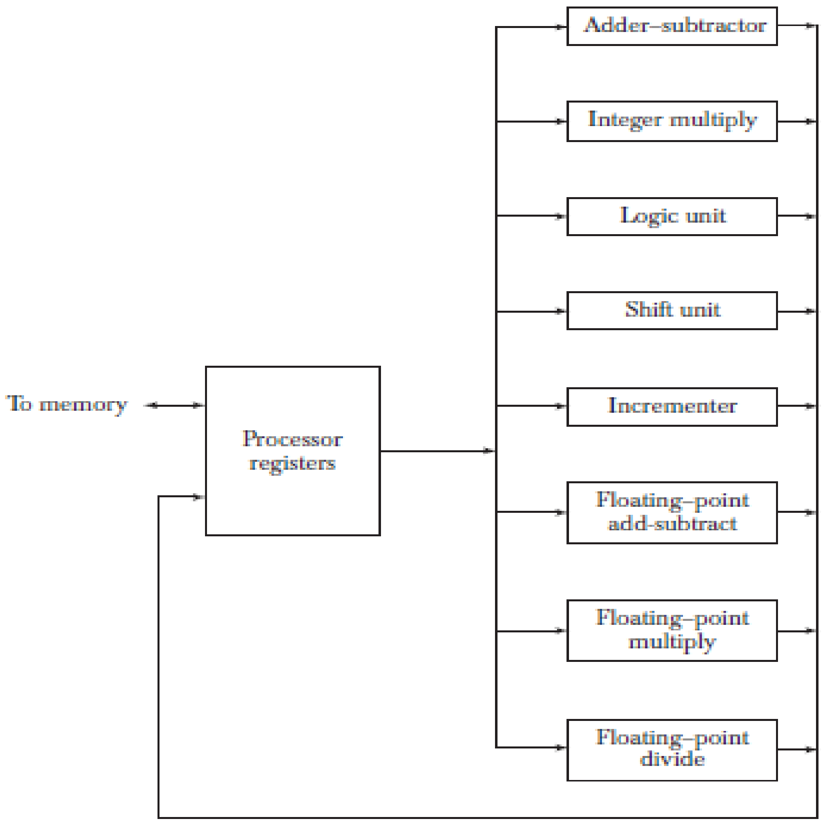
Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of easing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing 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 givenintervaloftime.Theamountofhardwareincreaseswithparallelprocessingandwith 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 processingisestablishedbydistributingthedataamongthemultiplefunctionalunits.For

Example the arithmetic logic and shift operations can be separated in to 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:

Parallel Processing can be classified in a variety of way. 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 constitute a data stream. Parallel processing may occur in the instruction stream, in the data stream, or in both.

<u>Flynn's classification divides computers in to four major groups as follows:</u>

- 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 maynot 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 organizationthatincludesmanyprocessingunitsunderthesupervisionof 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 **uni processors.** 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 compellingreasonforhavingseparatecachesforeachprocessoristoreducetheaverageaccess time in each processor.The same information may reside in a number of copies in some caches and main memory.
Toensuretheabilityofthesystemtoexecutememoryoperationscorrectly,the multiple copies must be kept identical.
This requirement imposes a cache coherence problem. **A memory scheme is coherent if the valuereturnedonaloadinstructionisalwaysthevaluegivenbythelateststoreinstruction 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.

<u>Conditions for Incoherence</u>

Cachecoherenceproblemsexistinmultiprocessorswithprivatecachesbecauseoftheneedto share writable data. Read-only data can safely be replicated without cache coherence enforcement mechanisms.
To illustrate the problem,consider the three-processorconfigurationwithprivatecachesshown in Fig. below. Some time during the operation an element X from main memory is loaded into thethreeprocessors,P1,P2,andP3.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.Theloadon X to the three processors results in consistent copie sin the caches and main memory. If one of theprocessorsperformsastoretoX,thecopiesofXinthecachesbecomeinconsistent.Aload

bytheotherprocessorswillnotreturnthelatestvalue.Dependingonthememoryupdatepolicy used in the cache, the main memory may also be inconsistent with respect to the cache.

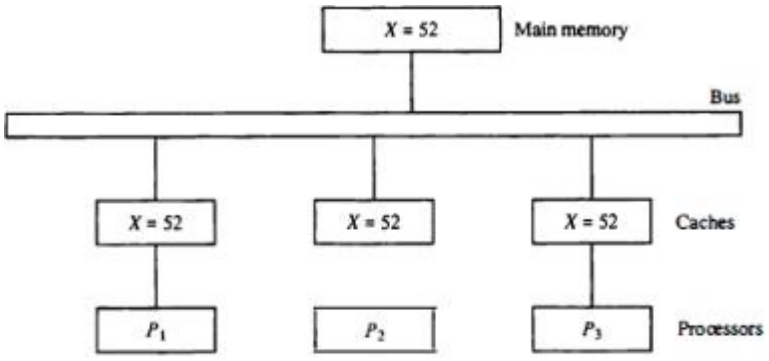
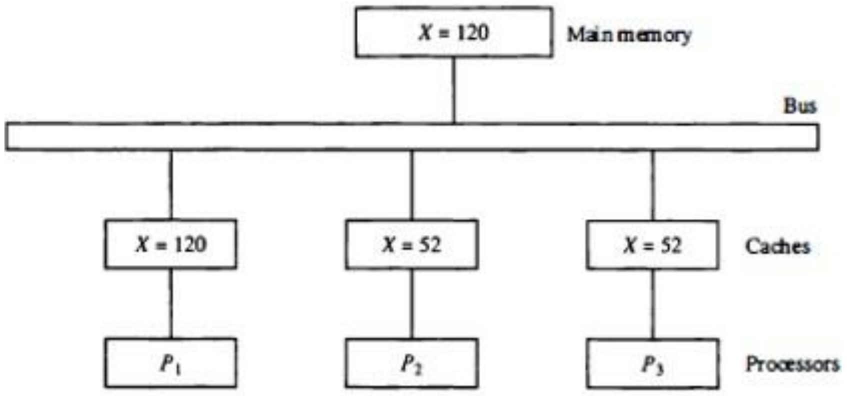


Figure Cache configuration after a load on X.

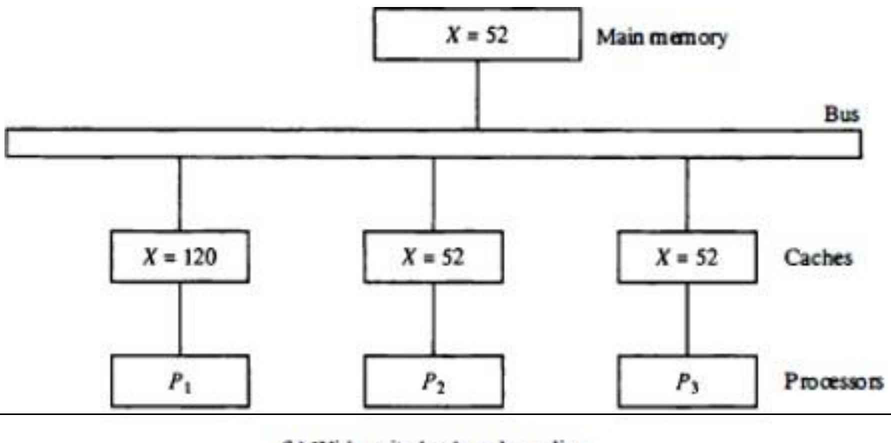
A store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy. A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value which is shown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

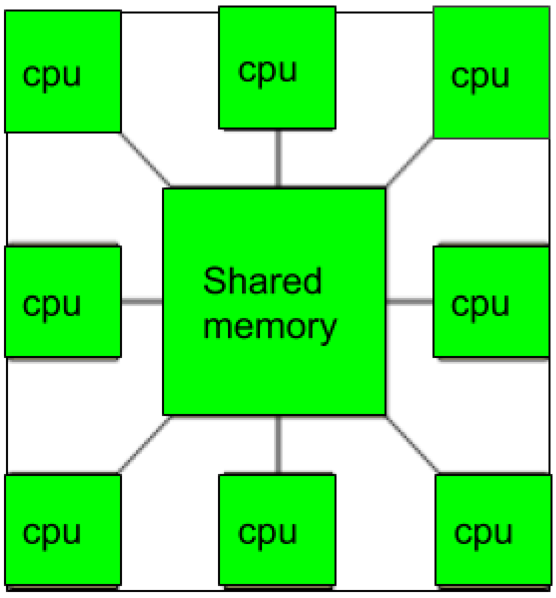


(b) With write-back cache policy

Another configuration that may cause consistency problems is a direct memory access (DMA) activityinconjunctionwithanIOPconnectedtothesystembus.Inthecaseofinput,the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations maybe read before the yareupdated from the cache when using a write-back policy.

Multiprocessor:

A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost the system's execution speed, with other objectives being fault tolerance and application matching. There are two types of multiprocessors, one is called shared memory multiprocessor and another is distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs shares the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.



In multiprocessor systems, interconnection structures define how processors, memory, and I/O devices connect and communicate. Inter-processor arbitration is the process of managing access to shared resources like buses when multiple processors need them simultaneously. Inter-processor communication refers to the methods and mechanisms by which these processors exchange data and synchronize their operations.

Interconnection Structures:

These structures dictate the physical and logical connections between components. Common types include:

- **Shared Memory:** Processors share a single memory space, allowing for fast data exchange.
- **Distributed Memory:** Each processor has its own private memory, requiring explicit communication for data sharing.
- **Bus-based:** A common bus connects all components, requiring arbitration for access.
- **Crossbar Switch:** Provides dedicated connections between any two components, allowing for concurrent communication.
  Inter-processor Arbitration:

When multiple processors need to access a shared resource, like a bus, a mechanism is needed to resolve conflicts. This is where inter-processor arbitration comes in.

- Daisy Chain Arbitration:

  A serial approach where a grant signal is passed from one processor to the next, giving priority to the first in line.
- Centralized/Parallel Arbitration:

  Dedicated lines for requesting and granting access, allowing for faster resolution.
- Dynamic Arbitration:
  The priority of processors can change dynamically based on certain conditions.

Inter-processor Communication:

This involves the methods and protocols by which processors exchange data and synchronize their actions.

- Shared Memory:

  Processors read and write to a shared memory location, allowing for fast communication. However, it can lead to synchronization issues.
- Message Passing:

Processors send and receive messages through designated channels, providing a more structured and controlled communication method.

- Semaphores and Mutexes:
  Synchronization primitives used to coordinate access to shared resources and prevent race conditions

Cache coherence :

Cache coherence is a mechanism that ensures data consistency across multiple caches in a multiprocessing or multicore system. When multiple processors or cores share data, each might have its own cache. If one processor modifies a piece of shared data, cache coherence protocols ensure that all other caches holding that data are updated or invalidated to maintain data consistency. This prevents situations where different processors have conflicting versions of the same data, which could lead to incorrect program execution.

*****