CHAPTER 1 (6 Marks)
INTRODUCTION

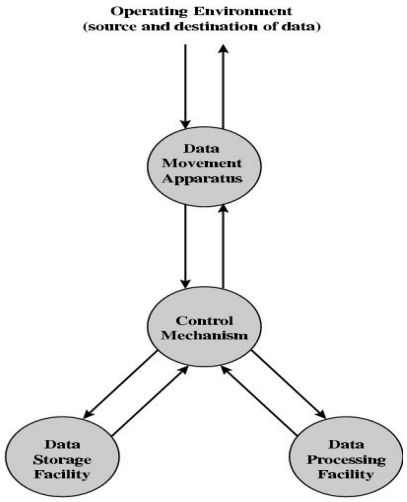
1.1 Computer Organization and Architecture

Computer Organization	Computer Architecture
Describes how hardware components of a computer system work together.	Deals with the design and specification of computer systems and components
How the computer works from an engineering perspective	How to design and optimize computer systems to achieve specific goals
How the computer do it	What the computer does
Deals with low level design issues (Logic and Circuits)	Deals with high level design issues (system and computer)
Structural Relationship	Functional Behaviour
It comprises of physical units like Circuit Design, Peripherals ALU, CPU and Memory	It comprises logical functions such as Instruction sets, Addressing Modes, Data Types and registers
For designing a computer organization is decided after it's architecture	For designing a computer it's architecture is fixed first

1.2 Structure and Function

- Structure is the way in which components relate to each other
- Function is the operation of individual components as part of the structure

Q. functional view and four types of operations performed in computer



All computer functions are-:

- **Data processing:** This refers to the ability of a computer to perform calculations, manipulate information, and execute instructions based on input data.

- **Data storage:** Computers have various types of memory that allow them to store data either temporarily (in RAM) or permanently (on a hard drive or other storage device). This data can be accessed and manipulated as needed by the computer's processor.

- **Data movement:** This involves the ability of a computer to transfer data between itself and other devices or networks. This can include sending and receiving data over a network, copying files between storage devices, or even communicating with other types of hardware devices such as printers or scanners.

- **Control:** In order to perform these functions effectively, a computer needs to be able to manage and control its own operations. This can involve coordinating data processing tasks, managing storage resources, and controlling the flow of data into and out of the system. Control also involves managing the

interaction between the user and the computer, such as through a user interface or command line.

The four main structural components of a computer system are:

- **Central Processing Unit (CPU):** The CPU is the "brain" of the computer and is responsible for executing instructions and performing calculations. It contains one or more processing cores, which can execute multiple instructions simultaneously through pipelining and other techniques.

- **Main Memory:** Main memory, also known as random-access memory (RAM), is used to temporarily store data and instructions that the CPU is actively using. It is volatile, meaning that its contents are lost when the power is turned off.

- **Input/Output (I/O):** Input/output refers to the methods by which a computer interacts with the external world. Input devices such as keyboards, mice, and touchscreens allow users to enter data into the computer, while output devices such as monitors, printers, and speakers display or communicate the results of computations.

- **System Interconnections:** System interconnections refer to the ways in which the various components of the system are connected and communicate with each other. This includes buses, switches, and other interconnect technologies that allow data to be transferred between components.

Q. What is performance balance? why required?

Performance balance, also known as workload balancing, is the process of distributing computing tasks and resources in a system to achieve optimal performance. It involves managing the allocation of processing power, memory, storage, and network

bandwidth to ensure that all parts of the system are functioning efficiently and effectively.

Performance balance is required because modern computer systems are complex and consist of multiple interconnected components. These components, such as processors, memory, and storage devices, have different capabilities and limitations. In addition, different software applications have different demands on system resources. If these resources are not properly balanced, certain components of the system may become overloaded, leading to performance degradation and system failures.

Q. Explain design goals and performance metrics?

Design goals are the primary objectives that a computer system architect or designer aims to achieve when creating a system. Performance metrics are measurements used to evaluate the effectiveness and efficiency of the system in meeting those design goals.

Some common design goals for computer systems include:

- **Performance:** This refers to how fast the system can perform tasks and how much work it can handle in a given amount of time.

- **Reliability:** This refers to the system's ability to perform consistently and predictably over time, without experiencing crashes or other errors.

- **Availability:** This refers to the system's ability to remain operational and accessible to users over time, even in the face of hardware or software failures.

- **Security:** This refers to the system's ability to protect against unauthorized access or use, as well as its ability to maintain data privacy and integrity.

Some common performance metrics used to evaluate computer systems include:

- **Throughput:** This refers to the amount of work that a system can perform in a given amount of time.
- **Latency:** This refers to the amount of time it takes for a system to respond to a request or perform a task.
- **Response time:** This refers to the amount of time it takes for a user to receive a response to a request.
- **Scalability:** This refers to the system's ability to handle increasing workloads without suffering a significant decrease in performance.

Q. how can we maintain performance balance between processor and memory

Maintaining performance balance between processor and memory is crucial for achieving optimal system performance. The processor and memory are two of the most critical components in a computer system, and their performance is closely linked. If the processor is too fast for the memory, the processor will be forced to wait for data to be fetched from memory, resulting in performance degradation. Similarly, if the memory is too slow for the processor, the processor will spend too much time waiting for data to be transferred, leading to decreased performance.

There are several techniques that can be used to maintain performance balance between processor and memory:

- **Cache optimization:** Caching is a technique used to improve performance by storing frequently used data in a faster, smaller memory called a cache. By optimizing the cache, designers can

ensure that the processor has access to the data it needs quickly, without overloading the memory.

- **Memory bandwidth optimization:** Memory bandwidth refers to the amount of data that can be transferred from memory to the processor in a given amount of time. By optimizing memory bandwidth, designers can ensure that the processor can access the data it needs quickly and efficiently.

- **Processor speed throttling:** If the processor is too fast for the memory, it may be necessary to slow down the processor to maintain performance balance. This can be achieved through techniques such as clock throttling or dynamic voltage scaling, which reduce the processor speed when it is not needed.

- **Memory optimization:** By optimizing the memory, designers can ensure that it is able to keep up with the demands of the processor. This can involve techniques such as increasing memory capacity, improving memory access times, or using specialized memory technologies such as DDR (double data rate) or HBM (high-bandwidth memory).

Q. computer functions with different cycles

- **Fetch:** The processor retrieves the instruction from memory by sending the address of the instruction to the memory controller. The memory controller retrieves the instruction and sends it back to the processor.

- **Decode:** The processor decodes the instruction by determining what operation to perform based on the opcode and operands specified in the instruction. The operands may be registers or memory locations.

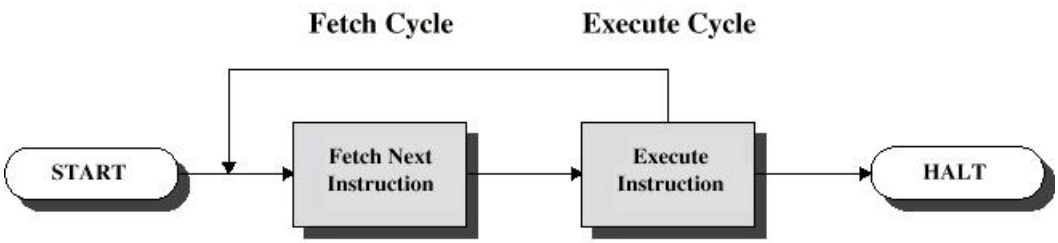
- **Execute:** The processor performs the operation specified by the instruction. This may involve arithmetic or logical operations,

data transfers between registers and memory, or branching to a different instruction.

- **Store:** The processor stores the result of the operation back in memory or a register. This may involve writing the result to a memory location or updating a register with the new value.

Computer Function (Youtube)

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory.



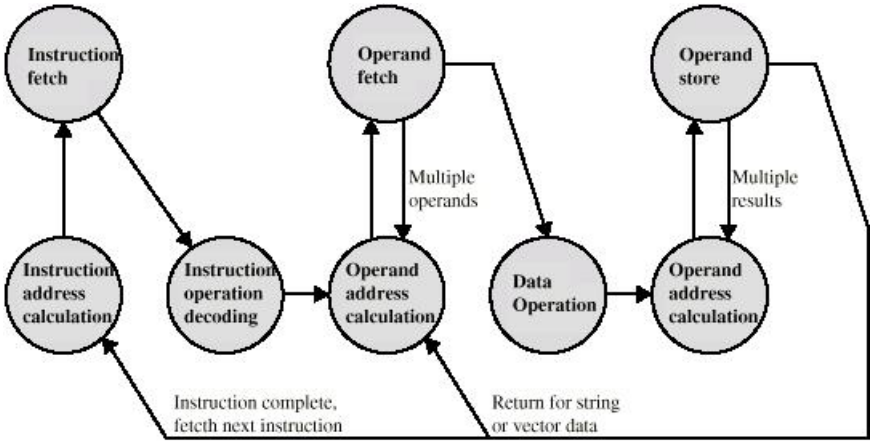


Fig: Instruction cycle state diagram

- **Instruction address calculation (IAC):** In this state, the CPU calculates the memory address of the next instruction to be executed.

- **Instruction fetch (IF):** In this state, the CPU retrieves the instruction from memory address calculated in the previous step. The instruction is then stored in the instruction register (IR).

- **Instruction operation decoding (IOD):** In this state, the CPU decodes the instruction stored in the IR to determine what operation needs to be performed.

- **Operand address calculation (OAC):** In this state, the CPU calculates the memory address of any operands required by the instruction.

- **Operand fetch (OF):** In this state, the CPU retrieves any operands required by the instruction from memory. The operands are then stored in temporary registers or memory locations for use in the next step.

- **Data operation (DO):** In this state, the CPU performs the actual operation specified by the instruction. This may involve performing arithmetic or logical operations on the operands retrieved in the previous step.

- **Operand address calculation (OAC):** In this state, the CPU calculates the memory address where the result of the operation should be stored.

- **Operand store (OS):** In this state, the CPU stores the result of the operation back into memory or registers, depending on the nature of the instruction. This completes the execution of the instruction

Interrupts

An interrupt is a signal sent to the processor by a device or program, indicating that an event has occurred that requires the processor's attention.

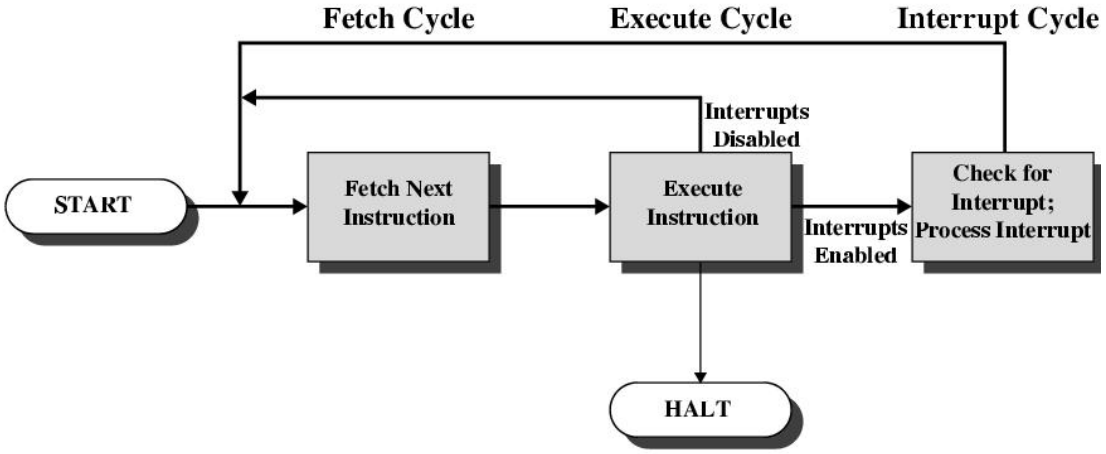


Fig: Instruction Cycle with Interrupts

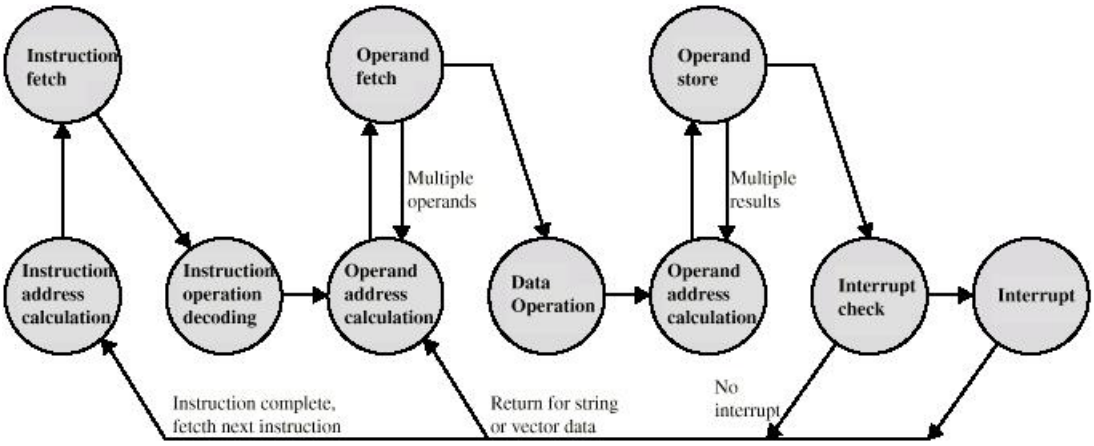


Fig: Instruction cycle state diagram, with interrupts

- **Interrupt check (IC):** In this state, the CPU checks for any pending interrupts. If an interrupt is pending, the CPU saves the current state of the system and moves to the interrupt service routine (ISR).

- **Interrupt :** In this state, the CPU executes the code necessary to handle the interrupt. Once the ISR is complete, the CPU returns to the point in the program where the interrupt occurred..

Interconnection structure

Q . explain the interconnection structures of computer. Also the different operations over them?

The interconnection structure of a computer system is responsible for connecting different components such as the CPU, memory, and I/O devices to facilitate the transfer of data and control signals among them.

- **Memory Connection:** In the case of memory, the interconnection structure receives and sends data, addresses and control signals. It also deals with timing signals to ensure that data is transferred at the right time. The memory is typically connected to the CPU and other devices through a bus system, which is a shared communication channel that enables multiple devices to communicate with each other.

- **I/o connection :** Input/output devices are typically connected to the interconnection structure through I/O ports, which act as interfaces between the devices and the computer system. These ports may be specialized for specific types of devices, such as USB ports for connecting USB devices, or they may be more general-purpose, such as serial or parallel ports.

- **CPU Connection :** The CPU is also connected to the interconnection structure through a bus system, which allows it to communicate with other devices in the system. The bus system typically consists of multiple buses, each dedicated to a specific type of communication, such as data transfer, address transfer, or control signal transfer.

The CPU interacts with memory and I/O devices in different ways, depending on the operation being performed. Some examples of operations include:

- **Instruction Fetch**
- **Data Read**
- **Data Write**
- **I/O Read**
- **I/O Write**

Bus Interconnection

Q. Explain Bus interconnection?

In computer organization and architecture, bus interconnection refers to the way in which different components of a computer system are connected to each other using a set of shared communication lines called a bus.

The bus is used to transfer data, instructions, and control signals between different components such as the CPU, memory, and I/O devices. The bus interconnects these components by providing a common communication pathway for the transfer of information.

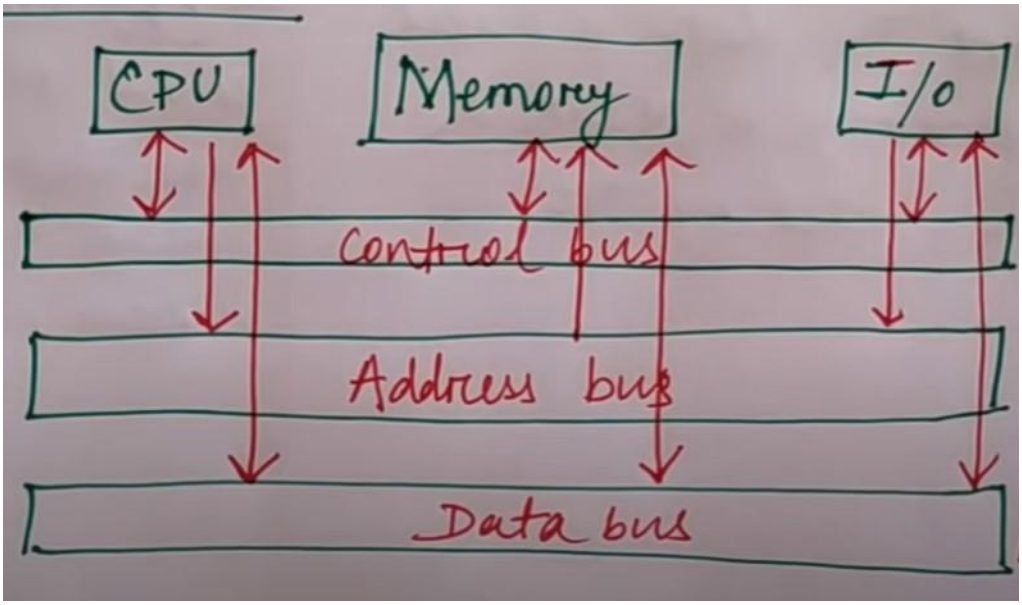
There are different types of buses used in computer systems, such as the address bus, data bus, and control bus.

- **Address Bus:** This type of bus is used to transfer memory addresses between the components of the computer system, such as the CPU and the memory. The address bus is a unidirectional bus, meaning that data can only be transferred in one direction - from the CPU to the memory. The number of

wires in the address bus determines the maximum amount of memory that can be accessed by the CPU.

- **Data Bus:** This type of bus is used to transfer data between the components of the computer system, such as the CPU and the memory. The data bus is a bidirectional bus, meaning that data can be transferred in both directions - from the CPU to the memory and from the memory to the CPU. The number of wires in the data bus determines the maximum amount of data that can be transferred between the CPU and the memory.

- **Control Bus:** This type of bus is used to transfer control signals between the components of the computer system, such as the CPU and the memory. The control bus is a bidirectional bus, meaning that control signals can be transferred in both directions - from the CPU to the memory and from the memory to the CPU. Control signals can include signals to indicate whether the data being transferred is a read or a write operation, signals to indicate the timing of the data transfer, and signals to indicate errors in the data transfer.



Q . limitations of single bus system to connected different devices. what does the width of data and address bus represent in a system? why is bus hierarchy required?

Here are the limitations of a single bus system:

- Single bus systems have limited bandwidth, and as more devices are connected, the available bandwidth per device decreases. This leads to slower data transfer rates and reduced system performance.

- When two or more devices try to access the bus simultaneously, leading to delays and increased system complexity.

- Single bus systems have limited scalability, and as the number of devices connected to the bus increases, the system becomes more complex and difficult to manage.

- As the number of devices connected to the bus increases. This can result in data corruption, errors, and system instability.

- Security risks are higher in single bus systems, as all devices have access to the same bus, making it more difficult to implement security measures to protect the system from external threats.

The width of the data bus determines the amount of data that can be transferred between the components of the system in a single cycle. For example, a 32-bit data bus can transfer 32 bits (or 4 bytes) of data at a time. The wider the data bus, the faster the data can be transferred between components.

The width of the address bus determines the maximum amount of memory that can be addressed by the system. For example, a 32-bit address bus can address up to 4GB of memory. The wider the address bus, the larger the amount of memory that can be addressed by the system.

Bus hierarchy is required in computer systems to improve system performance and scalability. In a complex computer system with multiple components, a single bus can become a bottleneck, leading to slow data transfer rates and reduced system

---

performance. Bus hierarchy helps to address this issue by dividing the system into multiple buses at different levels.

Q. Explain different elements of bus design.

The design of a bus system in a computer architecture involves several elements that must be considered to ensure efficient and reliable communication between different components. Here are some of the key elements of bus design:

- **Bus Topology:** The bus topology refers to the physical arrangement of the bus, including the wiring, connectors, and other components. The choice of topology depends on factors such as the number of devices to be connected, the distance between devices, and the bandwidth required.

- **Bus Protocol:** The bus protocol refers to the rules and procedures that govern how devices communicate over the bus. The protocol specifies the format of data transfer, the timing of signals, error handling, and other aspects of communication. Common bus protocols include PCI, USB, and Ethernet.

- **Bus Width:** The bus width refers to the number of data lines on the bus, which determines how much data can be transferred at a time. A wider bus allows for faster data transfer, but also requires more power and more complex hardware.

- **Bus Speed:** The bus speed refers to the frequency at which data is transferred over the bus, measured in megahertz (MHz) or gigahertz (GHz). A higher bus speed allows for faster data transfer, but also requires more power and can cause signal integrity issues.

- **Bus Arbitration:** Bus arbitration refers to the process of resolving conflicts when multiple devices try to access the bus simultaneously.

Q. Explain different types of bus arbitration and compare them

Bus arbitration refers to the process of resolving conflicts when multiple devices try to access the bus simultaneously

- **Centralized Arbitration:** Centralized arbitration involves a central arbiter that controls access to the bus. In this type of arbitration, all devices must request permission from the arbiter before accessing the bus. The arbiter decides which device has priority and grants access to the bus accordingly. The advantage of centralized arbitration is that it is simple and easy to implement. However, it can also be a bottleneck, as all requests must go through a single point of control.

- **Distributed Arbitration:** Distributed arbitration allows each device to make its own decision about when to access the bus. In this type of arbitration, each device listens for the bus to become available and then attempts to access it. If multiple devices attempt to access the bus simultaneously, a collision occurs, and the devices must wait for a random amount of time before trying again. The advantage of distributed arbitration is that it can be more efficient than centralized arbitration, as devices can access the bus more quickly. However, it can also be more complex to implement and may require additional hardware.

- **Priority Arbitration:** Priority arbitration assigns a priority level to each device on the bus. The device with the highest priority is granted access to the bus first, followed by the device with the next-highest priority, and so on. If two devices have the same priority level, a collision occurs, and the devices must wait for a random amount of time before trying again. The advantage of priority arbitration is that it can be very efficient, as high-priority devices are granted access to the bus quickly. However, it can also be complex to implement and may require additional hardware.

- **Round-Robin Arbitration:** Round-robin arbitration allocates bus access to each device in turn. Each device is granted access to the bus for a fixed amount of time, after which the next device in line is granted access. The advantage of round-robin arbitration is that it is fair, as each device is given equal access to the bus. However, it can also be inefficient, as some devices may not require access to the bus as frequently as others.

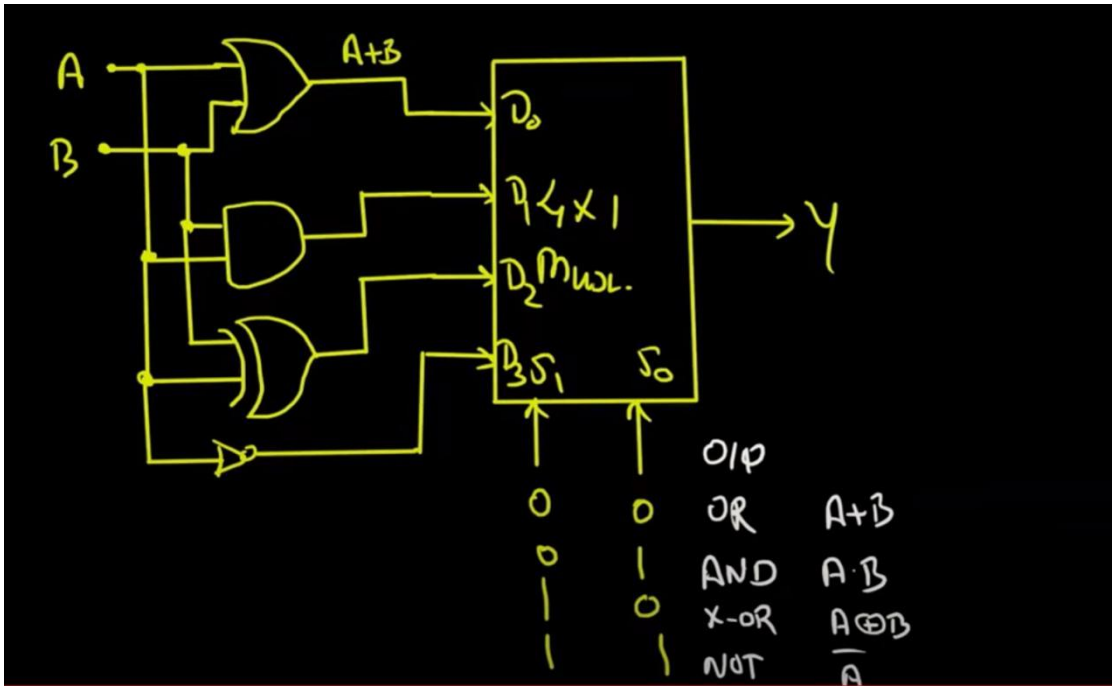
Q. What is PCI?

PCI (Peripheral Component Interconnect) is a type of computer bus used for connecting peripheral devices to a computer's motherboard. The PCI bus is a standard interface that provides high-speed data transfer between devices, allowing them to communicate with each other and with the CPU.

CHAPTER 2 (18 Marks)

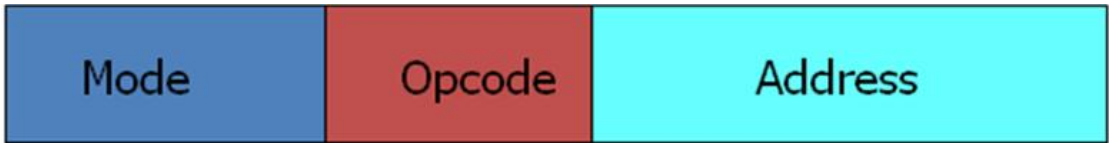
Central Processing Unit

Q. Design a 2-Bit ALU that can perform , AND, OR, X-OR and NOT operations.



Instruction formats (Youtube)

Instruction formats are a way of representing the various parts of a machine instruction in a computer's memory. They define the structure of an instruction and specify how the various fields of the instruction are encoded.



- An Operation code field specifies the operation to be performed such as add, subtract etc.

- An Address field specifies a memory address or a processor register, where operand is stored.

- A Mode field that specifies the way the operand or the effective address of the operand is determined.

Computers may have instructions of several different lengths containing varying number of addresses. Following are the types of instructions.

1. Three address Instruction

Instruction with three operands reference is called three address instruction. Computers with three address instruction format can use each address field to specify either a processor register or a memory operand. It is also called general register organization.

Three-address instruction format

To evaluate X = (A + B) * (C + D)

```
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
```

It is assumed that processor has two registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

- **ADVANTAGE:** It results in short programs when evaluating arithmetic expressions.
- **DISADVANTAGE:** The instructions requires too many bits to specify 3 addresses.

2. Two address Instruction

Two-address instruction format

Instruction with two operands reference is called two address instruction. They are the most common in commercial computers. Each address field may specify either a processor register or a memory operand.

To evaluate X = (A + B) * (C + D)

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

3. One address Instruction

Instruction with one operand reference is called one address instruction and use an implied accumulator (AC) register for all data manipulation. All operations is done between accumulator register and memory operand. It is called single accumulator organization.

One-address instruction format

To evaluate X = (A + B) * (C + D)

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

4. Zero address Instruction

Instruction without address field/operand is known as zero address instruction. Also called stack based organization.

Zero-address instruction format

To evaluate X = (A + B) * (C + D). convert first to Post fix = AB+CD+* (link for conversion)

```
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
```

Question 2
Question 3
Question 4
Question 5

Addressing Modes (Youtube)

Addressing modes refer to the different techniques used by computers to interpret or modify the address field of an instruction before accessing the operand.

These techniques are used to provide programming flexibility to users, reduce the number of bits needed for addressing, and enable the use of pointers, counters, indexing, and other functions. There are various modes of addressing, including those that do not require an address field at all.

In computer architecture, the **effective address** is the final memory address used to access data in memory. It is calculated by combining the base address with an offset, displacement, or index value based on the addressing mode used. The effective address is determined by the addressing mode of the instruction, and it is the actual address of the operand used by the CPU to retrieve or store data during program execution. It is an important concept as it determines the location of data in memory.

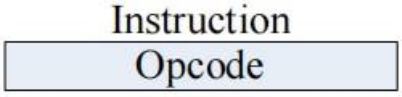
1. Implied addressing mode

- Implied addressing mode is a simple addressing mode used in computer programming where the instruction does not explicitly specify the operand's memory address.

- The operand's address is instead implied by the instruction opcode itself or the context in which the instruction is executed.

Eg:

- CLC - clears the carry flag in the processor's status register
- INX - increments the value of the X register by one



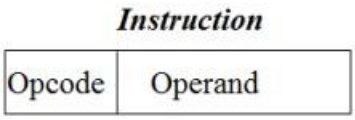
- **Advantage:** no memory reference.
- **Disadvantage:** limited operand

2. Immediate addressing mode

- The operand value is directly included in the instruction itself, typically as a constant.
- The immediate addressing mode is used for operations that require constant values, such as adding a fixed value to a register or setting a flag.
- Immediate addressing mode is often used for arithmetic and logical operations
- No computation required to calculate EA

E.g. ADD 5

- Add 5 to contents of accumulator
- 5 is operand

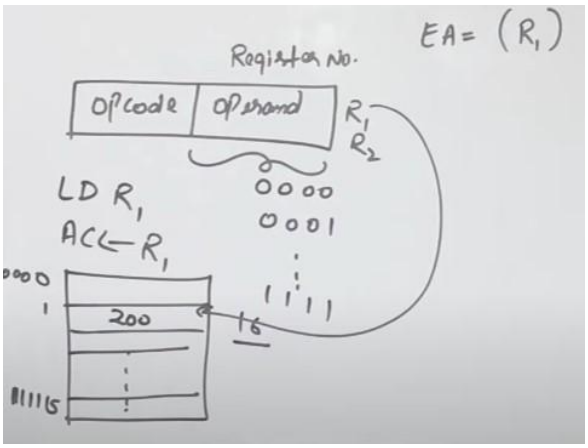


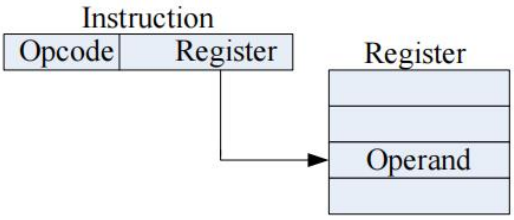
Eg : MVI B, 50H (B <- 50H)

- **Advantage:** no memory reference.
- **Disadvantage:** limited operand

3. Register direct addressing mode

- In direct register addressing mode, the operand is stored directly in the register specified in the instruction.
- Register No is written in Instruction.





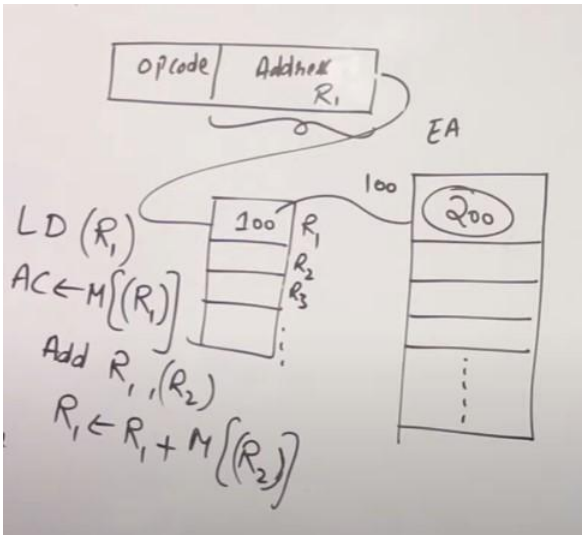
For example

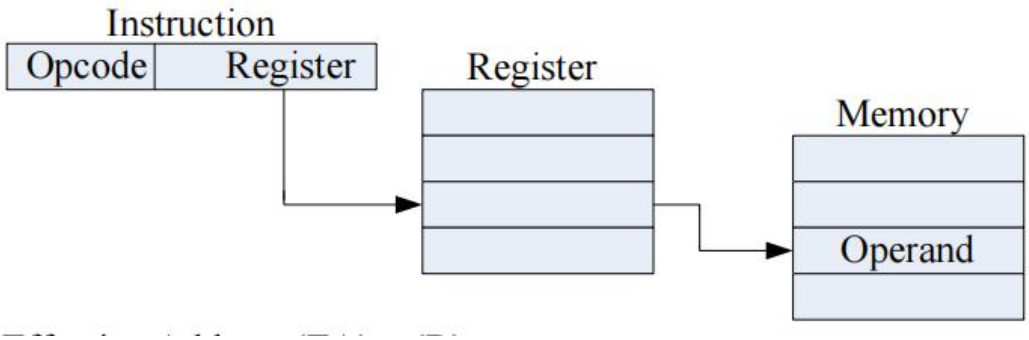
- MOV R1, #10 - moves the constant value 10 into register R1
- ADD R1, R2 - adds the contents of register R2 to the contents of register R1 and stores the result in R1
- SUB R1, #5 - subtracts the constant value 5 from the contents of register R1 and stores the result in R1

Advantage: no memory reference.
Disadvantage: limited address space

4. Register Indirect addressing mode

- In indirect register addressing mode, the memory address of the operand is stored in the register specified in the instruction.
- Register indirect addressing mode is useful for accessing data in memory without knowing the actual memory address of the operand.





EA = (R1)

- LDR R1, [R2] - loads the contents of the memory address stored in register R2 into register R1
- STR R1, [R2] - stores the contents of register R1 at the memory address stored in register R2
- ADD R1, [R2] - adds the contents of the memory address stored in register R2 to the contents of register R1 and stores the result in R1

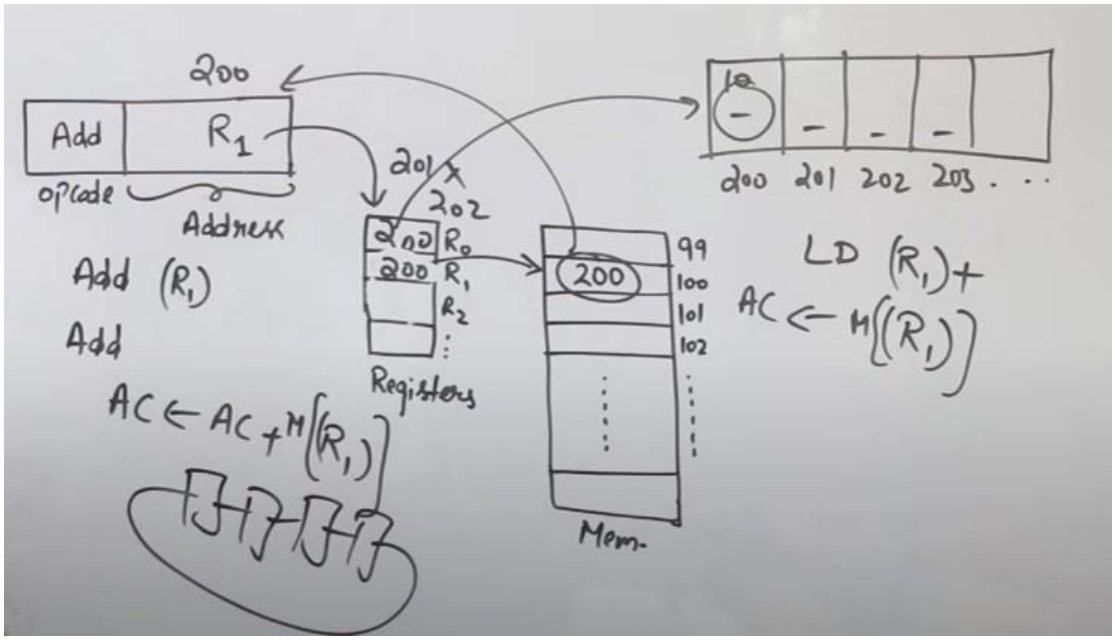
Advantage: Large address space.
Disadvantage: Extra memory reference

5. Auto increment or decrement addressing mode

- This is similar to register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

- When the address stored in the registers refers to a table of data in memory, it is necessary to increment or decrement the registers after every access to the table.

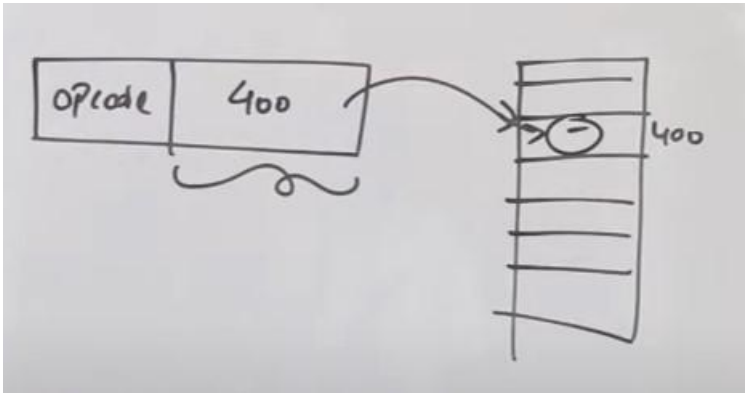
- This can be achieved by using the increment or decrement instruction. In some computers it is automatically accessed.

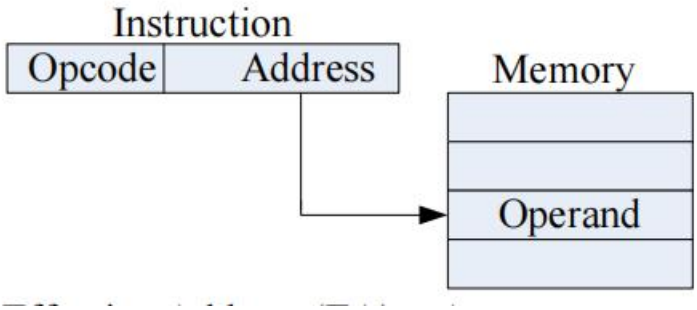


6. Direct Addressing Mode

- Actual address is given in the instruction

- Use to access variables





Effective Address (EA) = A

For example :

LDA 4000H - loads the contents of memory location 4000H into the accumulator.

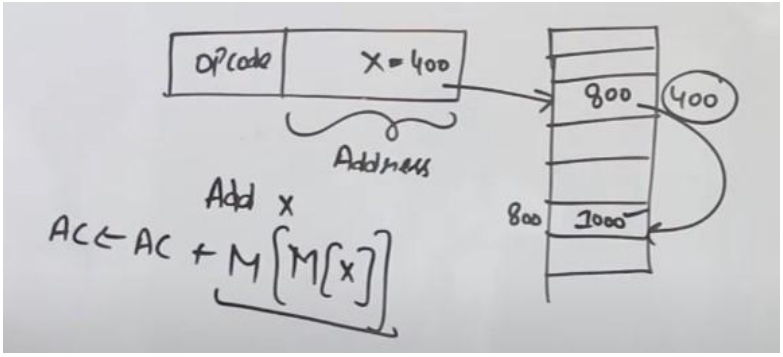
ADD 4001H - adds the contents of memory location 4001H to the accumulator

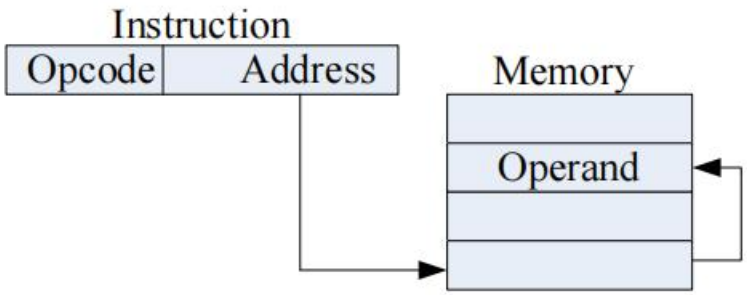
Advantage: Simple.
Disadvantage: limited address field

7. Indirect Addressing Mode

- The address of the memory location to be accessed is not specified directly, but rather indirectly through a register or memory location that holds the address.

- Used to implement pointers and passing parameters.





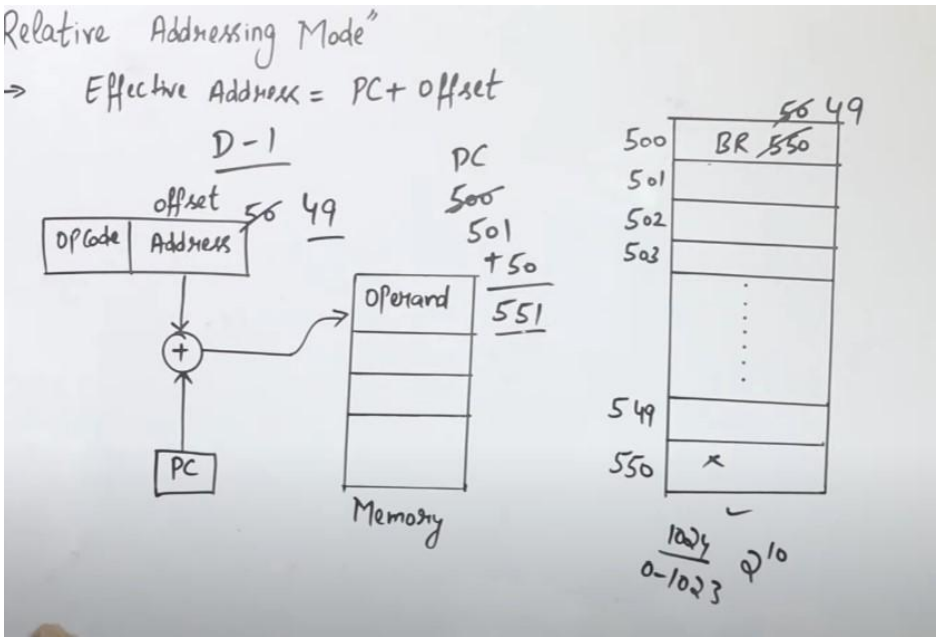
EA = (X)

Eg:
MOV AX, [BX] - This instruction moves the contents of the memory location whose address is stored in the BX register into the AX register. Here, BX acts as a pointer to the memory location that contains the data.

Advantage: Flexibility.
Disadvantage: Complexity

8. Relative Addressing Mode

- Used for program control instructions like branch and jump

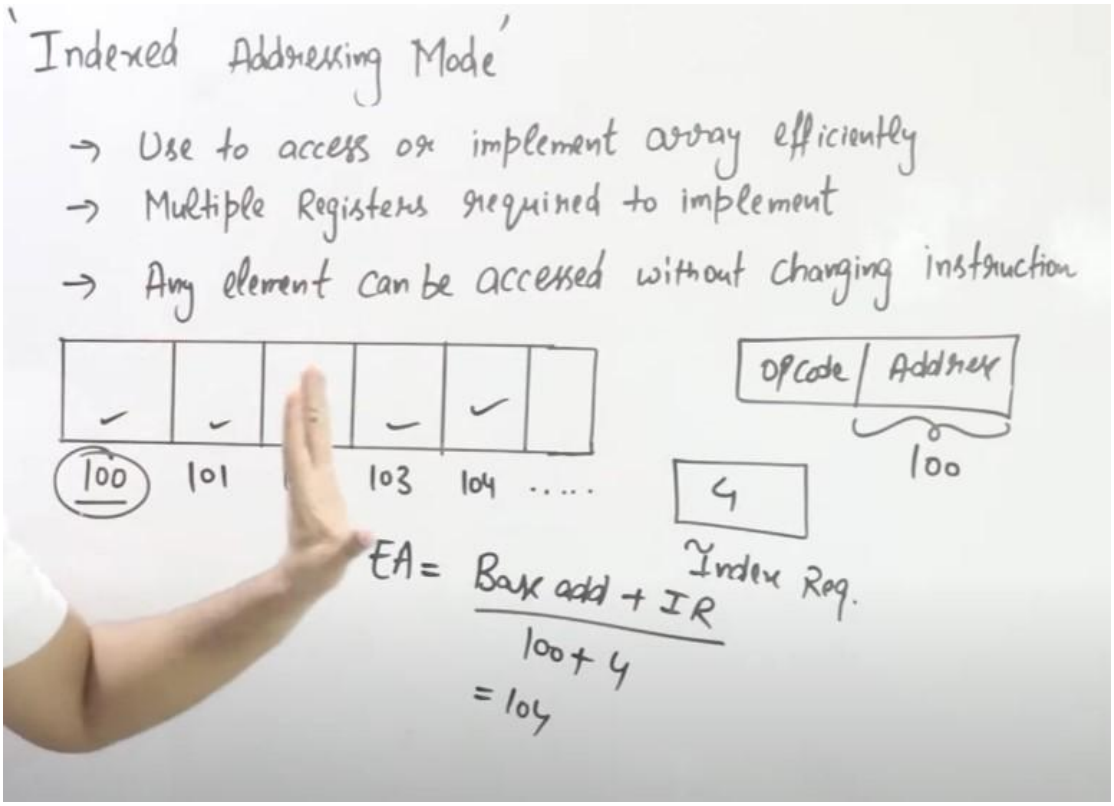


We give offset in address and when we give 50 in it the PC is 500 and the data should be fetched from 550 because and program shoud branch to BR 550 but PC will increase to 501(since PC always gives address of next instruction) and if we give 50 off set then after 50 displacement PC - 551 which is not 550 which we need so offset is give 1 less that is 49 to reach 550 and fetch data.

The major **advantage** of relative addressing mode is that it allows programs to access memory locations without having to know their absolute addresses, making the programs more portable and easier to write.

The major **disadvantage** of relative addressing mode is that it is limited to accessing memory locations that are located at a fixed distance from the current instruction, so it cannot be used to access memory locations that are farther away.

9. Indexed Addressing Mode



- Here fix the base address of the array say 100
- Here EA=BA+IR
- If we want to access 4^{th} element of array then set Index register to 4 the EA = 100+4 = 104 for 6^{th} element set IR= 6 and so on.

The major **advantage** of indexed addressing mode is that it allows for more flexible memory access than direct or immediate addressing modes, as it can be used to access memory locations that are located at a variable distance from the base address stored in the register.

The major **disadvantage** of indexed addressing mode is that it can be slower than direct addressing mode due to the additional time required to compute the address by adding the offset or index value to the base address stored in the register.

Q. Explain different types of data manipulation instructions with example.

Data manipulation instructions are used in computer architecture and programming to perform various operations on data stored in memory or registers. Here are some different types of data manipulation instructions along with examples:

1.Arithmetic Instructions: Arithmetic instructions are used to perform arithmetic operations on data. Examples include:

- ADD AX, BX ; Add the contents of the BX register to the AX register
- SUB CX, DX ; Subtract the contents of the DX register from the CX register

- MUL BX ; Multiply the contents of the AX register by the BX register

2.Logical Instructions: Logical instructions are used to perform logical operations on data, such as AND, OR, and NOT. Examples include:

- AND AX, BX ; Perform a bitwise AND operation on the contents of the AX and BX registers

- OR CX, DX ; Perform a bitwise OR operation on the contents of the CX and DX registers

- NOT BX ; Perform a bitwise NOT operation on the contents of the BX register

3.Shift and Rotate Instructions: Shift and rotate instructions are used to shift or rotate the bits of data to the left or right. Examples include:

- SHL AX, 2 ; Shift the bits in the AX register two positions to the left

- SHR BX, 3 ; Shift the bits in the BX register three positions to the right

- RCL CX, 1 ; Rotate the bits in the CX register one position to the left through the carry flag

4.Data Transfer Instructions: Data transfer instructions are used to transfer data between memory and registers, or between registers. Examples include:

- MOV AX, BX ; Move the contents of the BX register to the AX register

- MOV [SI], AX ; Move the contents of the AX register to the memory location specified by the SI register

- XCHG BX, CX ; Exchange the contents of the BX and CX registers

---

Q. Explain the component of CPU.

The CPU, or Central Processing Unit, is the "brain" of a computer system, responsible for executing instructions and performing calculations. It consists of several components, including:

- **Control Unit (CU):** The control unit is responsible for managing the operation of the CPU, fetching instructions from memory, decoding them, and executing them. It controls the flow of data within the CPU and between the CPU and other components of the computer system.

- **Arithmetic Logic Unit (ALU):** The ALU is responsible for performing arithmetic and logical operations on data. It can perform operations such as addition, subtraction, multiplication, division, and comparison.

- **Registers:** Registers are small, high-speed storage locations located within the CPU that hold data and instructions being processed by the CPU. There are several types of registers, including:

  - **Program Counter (PC):** The program counter holds the address of the next instruction to be executed.

  - **Instruction Register (IR):** The instruction register holds the current instruction being executed.

  - **Accumulator (ACC):** The accumulator holds data being processed by the ALU.

- **General-Purpose Registers:** General-purpose registers can be used to hold data or addresses.

- **Cache:** Cache is a small, high-speed memory located within the CPU or on the CPU chip that stores frequently accessed data and instructions. It is used to improve the performance of the CPU by reducing the time it takes to access data from memory.

- **Bus Interface Unit (BIU):** The bus interface unit is responsible for communicating with other components of the computer system through the system bus. It fetches data and instructions from memory and writes data to memory.

Q. Write down the need for addressing modes.

- Addressing modes allow for efficient and flexible memory access by specifying how to access data and instructions stored in memory.

- Addressing modes are used to support program control flow, such as branching and jumping to different locations in memory.

- Addressing modes provide flexibility and versatility in performing data manipulation operations.

- Addressing modes help to optimize code by allowing for efficient memory access and minimizing the number of instructions required to perform operations.

- Addressing modes can be designed to work within the limitations of the hardware, such as the number of registers, memory size, or processor speed.

RISC Architecture	CISC Architecture
RISC stands for Reduced Instruction Set Computing.	CISC stands for Complex Instruction Set Computing.
RISC processors have a small and simple set of instructions.	CISC processors have a large and complex set of instructions.
RISC architecture emphasizes simpler instructions that can be executed quickly.	CISC architecture emphasizes complex instructions that can perform multiple operations in one instruction.
RISC processors typically have a large number of registers for storing data.	CISC processors typically have a smaller number of registers, and some operations are performed directly on memory.
RISC processors rely heavily on compilers to optimize code.	CISC processors rely on hardware to optimize code.
RISC architecture is more suited for applications that require high performance and efficient use of memory, such as embedded systems and mobile devices.	CISC architecture is more suited for applications that require complex operations and large amounts of memory, such as database systems and scientific computing.
Heavily Pipelined	Not/less Pipelined

CHAPTER 3 (10 MARKS)

CONTROL UNIT

Control Memory	Main Memory
Also known as "control store" or "microcode memory"	Also known as "RAM" or "primary memory"
Stores microcode, which provides the control signals for the processor to execute instructions	Stores data and instructions that are currently being used by the processor
Non-volatile, meaning that its contents are retained even when power is turned off	Volatile, meaning that its contents are lost when power is turned off
Small in size compared to main memory	Much larger in size compared to control memory
Access time is very fast	Access time is relatively slower compared to control memory
Typically used in embedded systems, where the microcode is fixed and doesn't change	Used in all types of computers, where both data and instructions need to be stored and retrieved

Control memory, also known as control store or microcode memory, is a type of computer memory that stores microcode/microprogram. Microcode is a low-level code that provides the control signals for the processor to execute instructions.

Hardwired Control Unit	Micro-Programmed Control Unit
Uses fixed logic gates and combinational circuits to generate control signals	Uses microcode to generate control signals
Designed using a hardware description language (HDL) and implemented directly in hardware	Implemented using a control store, which contains microcode
Faster execution since it does not require memory access to generate control signals	Slower execution compared to hardwired control unit since it requires memory access to fetch microcode
Difficult to modify or update since changes require physical changes to the hardware	Easily modifiable since changes can be made to the microcode without changing the hardware
Used in simple processors with a limited instruction set	Used in more complex processors with a larger instruction set
Lower cost since it requires less memory and has simpler implementation	Higher cost since it requires more memory to store microcode and has a more complex implementation

Microprogrammed control unit:

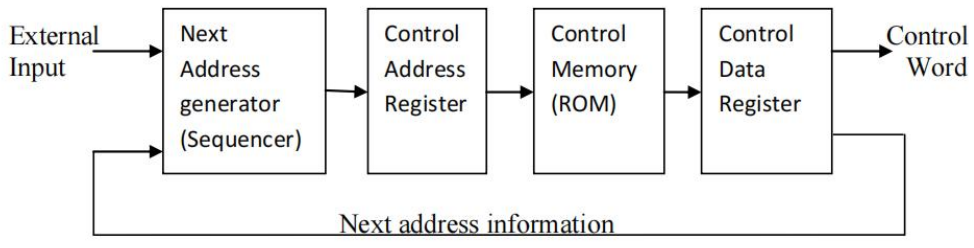


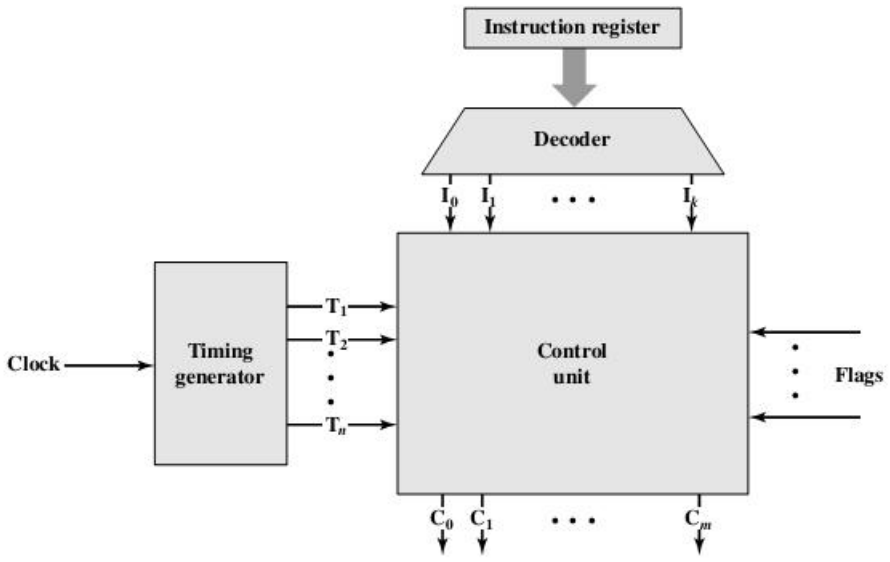
Fig 3-1: Microprogrammed Control Organization

A micro-programmed control unit is a type of control unit that uses a microprogram to control the operations of a computer. Here's how it works:

When an instruction is fetched from memory, the micro-programmed control unit uses the sequencer to generate a sequence of microinstructions that define the operations to be performed by the computer. The CAR holds the address of the next microinstruction to be executed, which is fetched from the control memory. The CDR holds the data required for the microinstruction being executed, which is used by the computer to perform the operation.

The control word is used to specify the microinstruction to be executed, including the address of the next microinstruction and any other control signals required for the operation. The micro-programmed control unit then executes the microinstruction, updating the CAR with the address of the next microinstruction to be executed, and repeating the process until the instruction is completed.

Hardwired Control Unit



A hardwired control unit is a component of a computer's central processing unit (CPU) responsible for managing the execution of instructions. It typically contains the following components:

- **Timing generator:** This component generates timing signals that synchronize the operations of the CPU's different components, such as the control unit, arithmetic logic unit (ALU), and memory unit.

- **Control unit (CU):** The CU is responsible for fetching instructions from memory, decoding them, and then executing them. It uses the timing signals generated by the timing generator to coordinate its operations with those of other components.

- **Decoder:** The decoder is responsible for decoding the instructions fetched by the CU. It translates the instruction into a series of control signals that are sent to other components of the CPU, such as the ALU and memory unit.

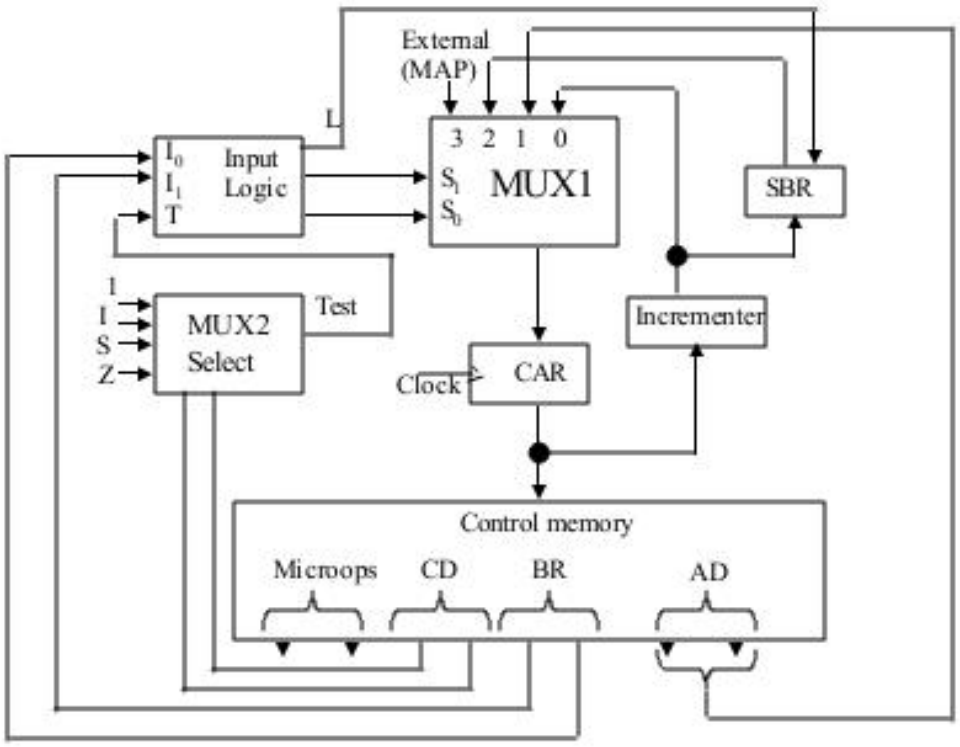
- **Instruction register (IR):** The IR is a small memory unit that temporarily stores the instruction being executed by the CPU. It is used by the CU to decode the instruction.

- **Flags:** Flags are special registers that contain the status of the CPU. They are used by the CPU to indicate whether certain conditions have been met, such as whether an arithmetic operation resulted in a zero value or whether an instruction caused an overflow.

In summary, a hardwired control unit uses a timing generator to coordinate the operations of the CPU's different components, a CU to fetch, decode, and execute instructions, a decoder to translate instructions into control signals, an IR to temporarily store instructions, and flags to indicate the status of the CPU.

Microprogram Sequencer

Q. Describe how address of control memory is selected.
Q. How address of micro instruction is generated by next address generator in control unit?
Explain with suitable diagram.



A microprogram sequencer is a crucial component of a microprogrammed control unit responsible for selecting the next address to be read from the control memory so that a microinstruction can be fetched and executed.

It consists of several parts, including a control memory, a subroutine register (SBR), two multiplexers, and an input logic circuit.

The control memory stores microinstructions that are executed by the control unit. The microprogram sequencer selects the address of the next microinstruction to be executed by the control unit. The

subroutine register (SBR) holds the return address for subroutine calls and returns.

The two multiplexers in the microprogram sequencer select an address from four different sources. The first multiplexer selects an address from one of the four sources and routes it to the Current Address Register (CAR). The second multiplexer tests the value of the selected status bit and applies the result to an input logic circuit.

The output from the CAR provides the address for the control memory. The contents of the CAR are incremented and applied to one of the multiplexer's inputs and to the SBR. Although the block diagram in the text shows a single SBR, a typical microprogram sequencer will have a register stack about four to eight levels deep to support push, pop, and stack pointer operations for subroutine calls and returns.

The CD (Condition) field of the microinstruction selects one of the status bits in the second multiplexer. The Test variable, which can be either 1 or 0, together with the two bits from the Branch (BR) field, goes to an input logic circuit to determine the type of operation to be executed.

Overall, the microprogram sequencer operates by selecting the address of the next microinstruction to be executed by the control unit. It uses two multiplexers to select an address from four different sources and test the value of the selected status bit. The contents of the CAR are incremented and applied to one of the multiplexer's inputs and to the SBR. The CD field of the microinstruction selects one of the status bits in the second multiplexer, and the Test variable, together with the two bits from the BR field, goes to an input logic circuit to determine the type of operation to be executed.

Q. What factors cause micro-programmed control unit to be selected over hardwired control unit

- Micro-programmed control units are more flexible than hardwired control units.
- Micro-programmed control units can be easier to design and implement.
- Micro-programmed control units are easier to test and debug than hardwired control units.
- Micro-programmed control units can be developed more quickly and at a lower cost than hardwired control units because they require less design and development effort.

CHAPTER 4 (10 MARKS)

Pipeline and Vector Processing

Q. What is pipeline? How performance of computer is increased using pipelining? Explain with example.

Pipelining is a technique for improving the performance of computer processors by dividing a complex task into smaller sub-tasks and processing them concurrently. In a pipelined system, the processing of a task is divided into a series of stages, and each stage performs a specific operation on the task.

For example, consider a simple pipeline for processing a list of numbers:

- In the first stage, the numbers are read from memory and loaded into a register.
- In the second stage, the numbers are sorted.
- In the third stage, the sorted numbers are written back to memory.

In a non-pipelined system, each of these stages would be performed one after the other, which would take a long time to complete. However, in a pipelined system, each stage can be performed concurrently on a different set of data.

For example, while the first stage is reading the numbers from memory, the second stage can be sorting the previously read numbers, and the third stage can be writing the already sorted numbers back to memory. This results in a significant improvement in the processing speed and efficiency of the system.

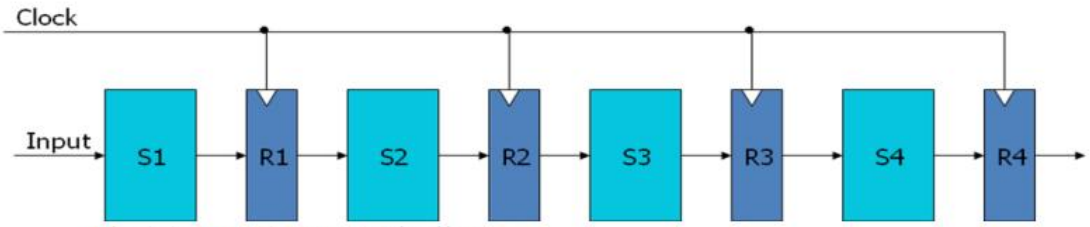


Fig 4-2: Four Segment Pipeline

The general structure of a four-segment pipeline is illustrated in Fig. 4-2. We define a task as the total operation performed going through all the segments in the pipeline.

Here IR (Instruction Registers i.e R1,R2..) are used to store the outputs of each stage

For 4-Stage Pipelining:

- **Instruction Fetch:** In this stage, the processor fetches the instruction from memory.

- **Instruction Decode:** In this stage, the processor decodes the instruction and determines the operations to be performed.

- **Execute:** In this stage, the processor performs the required operations on the data, such as arithmetic or logical operations.

- **Write-Back:** In this stage, the results of the execute stage are written back to memory or the processor's registers.

The behavior of a pipeline can be illustrated with a space-time diagram. It shows the segment utilization as a function of time.

For a 5-stage pipelining architecture, the five stages are typically:

- **Instruction Fetch (IF):** The processor fetches the instruction from memory.

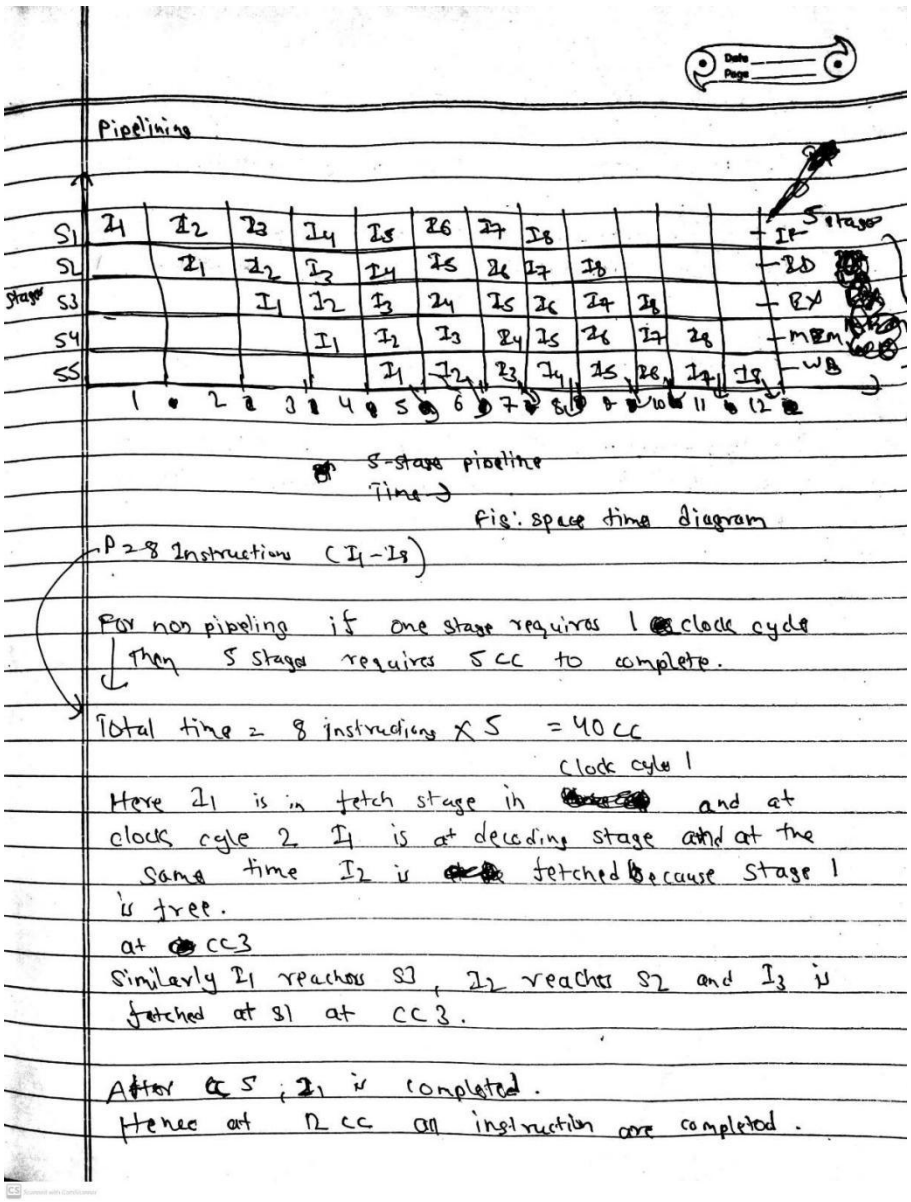
- **Instruction Decode (ID):** The processor decodes the instruction and determines the operations to be performed.

- **Execute (EX):** The processor performs the required operations on the data, such as arithmetic or logical operations.

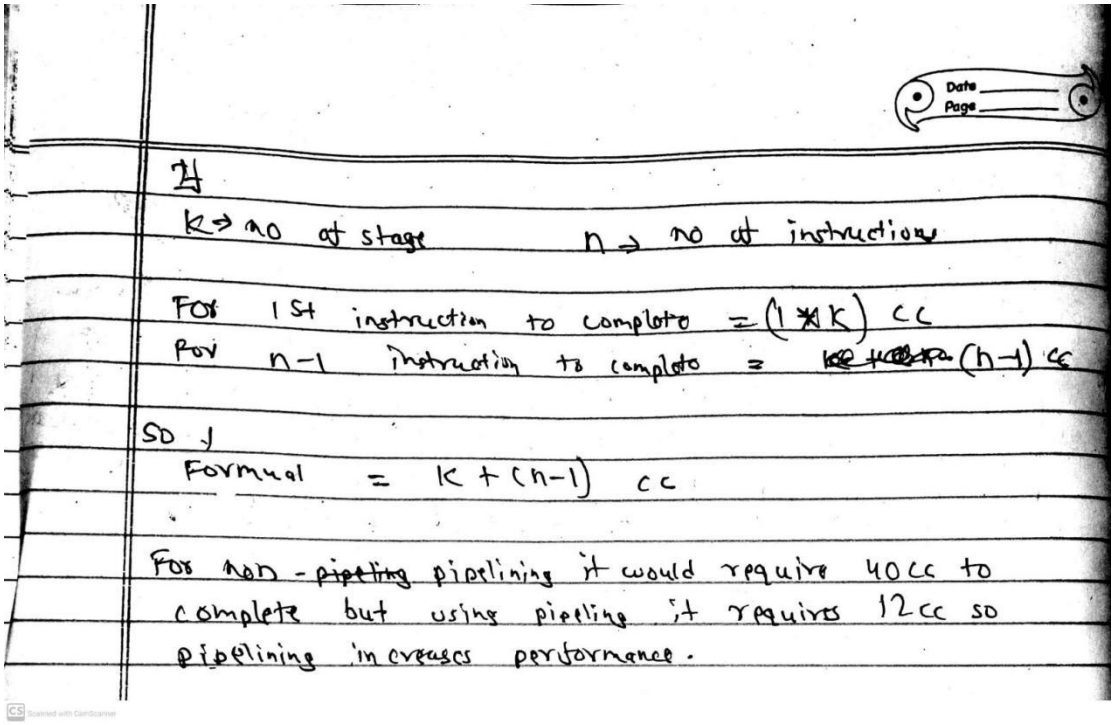
- **Memory Access (MEM):** The processor accesses memory, if required by the instruction.

- **Write-Back (WB):** The results of the execute stage are written back to memory or the processor's registers.

(6 stage pipelining ma write back pachi commit huncha)



Youtube link



Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
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	

Fig: 4-stage pipelining with 6 instructions

Limitations of Pipelining

- Not so easy process.
- Advance technology so cannot possible to implement in all computers.
- If instructions are not designed properly then may have adverse effect.
- If the depth of pipelining increases, no guarantee to increase the performance i.e. may hazardous.

- An unbalanced length of pipe stages reduces speedup.

Arithmetic Pipeline (youtube link)

An arithmetic pipeline is a type of processor pipeline that is specifically designed to perform arithmetic and mathematical operations on data. It is a common technique used in processors for handling mathematical operations more efficiently.

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. An arithmetic pipeline usually requires two or more operands to enter the pipeline at same time and the memory can be partitioned into a number of modules connected to a common memory address and data buses.

For example:

Floating point adder:

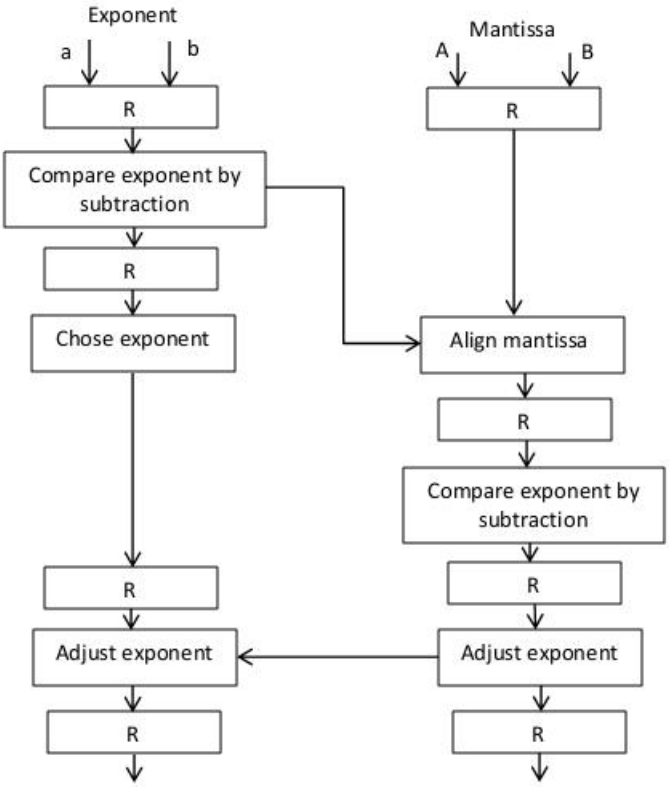
Inputs:

X=A*10a

Y=B*10b where, A & B: Mantissa and a and b: Exponents

The floating point addition and subtraction can be performed in four segments, the sub operations 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



Let us take: X=09505*10^3 and Y=0.8100*10^2

1. Compare exponents by subtraction
3-2=+1 (here + says that you should adjust 2nd exponent part by 1 place if -1 then adjust 1st exponent for 1 place)
Larger exponent is choose as the exponents of the result

2. Align mantissas
X=0.9505*10^3
0.08100*10^3

3. Add mantissas
Z=1.0315*10^3

4. Normalize result
Z=0.10315*10^4

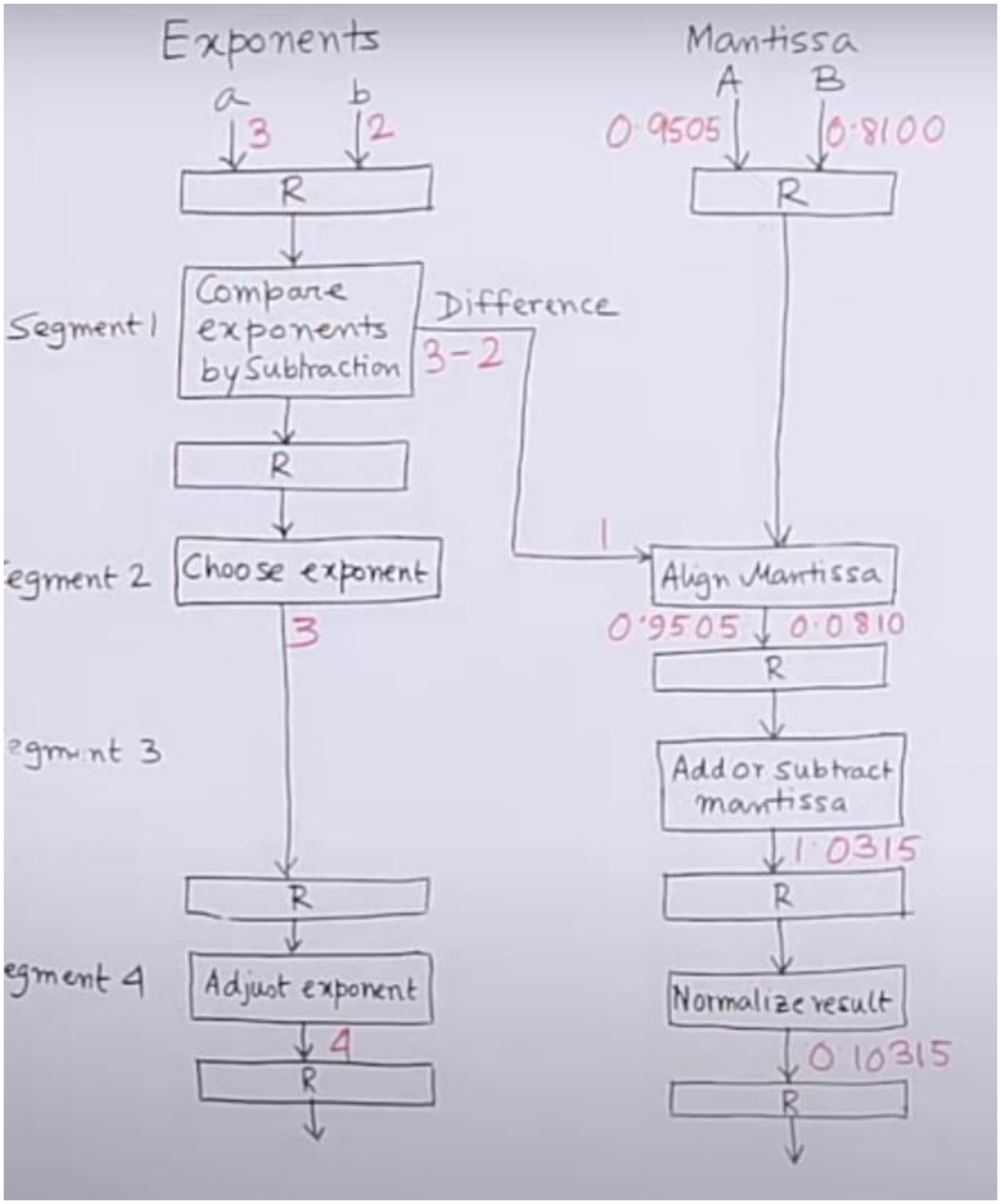


Fig : 4 stage arithmetic pipelining

Instruction Pipelining (Youtube Link)

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

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

- Fetch the instruction from memory.
- Decode the instruction.
- Calculate the effective address.
- Fetch the operands from memory.
- Execute the instruction.
- Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate.

- Different segments may take different times to operate on the incoming information.
- Some segments are skipped for certain operations.
- Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

Example: Four-Segment Instruction Pipeline

Assume that:

- The decoding of the instruction can be combined with the calculation of the effective address into one segment.
- The instruction execution and storing of the result can be combined into one segment.

Fig 4-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline.

- Thus up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

An instruction in the sequence may be causes a branch out of normal sequence.

- In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted.

- Similarly, an interrupt request will cause the pipeline to empty and start again from a new address value.

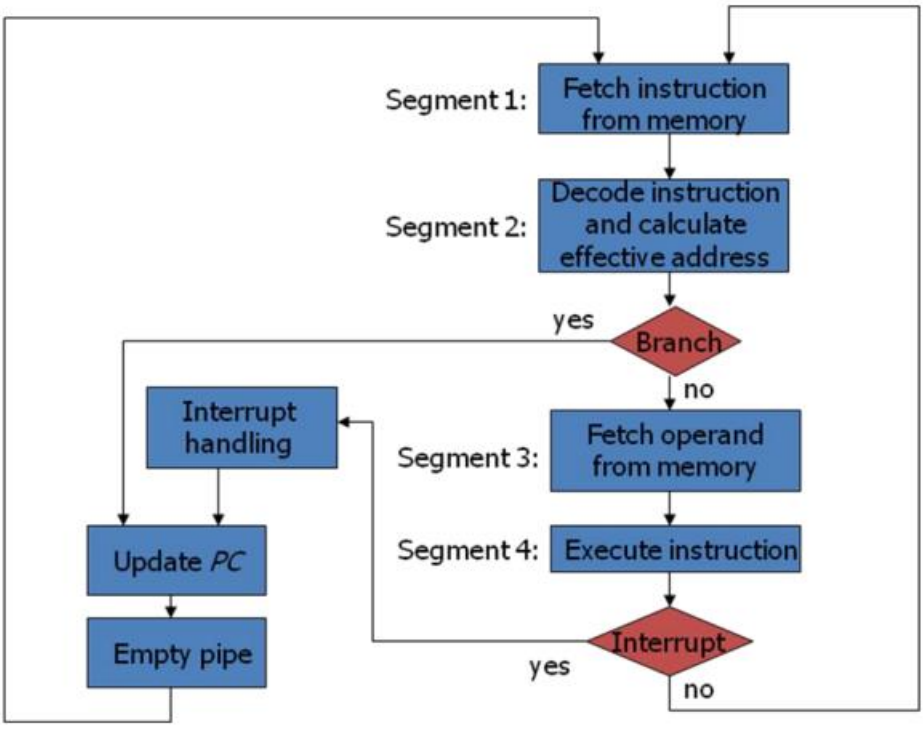


Fig 4-7: Four-segment CPU pipeline

- Fig. 9-8 shows the operation of the instruction pipeline.

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

Fig 4-8: Timing of Instruction Pipeline

  - FI: the segment that fetches an instruction
  - DA: the segment that decodes the instruction and calculate the effective address
  - FO: the segment that fetches the operand
  - EX: the segment that executes the instruction

Q. Discuss in detail about data dependency problem that arises in pipelining along with its solution.

Pipelining is a technique used in computer architecture to increase the efficiency of processors by breaking down the execution of instructions into smaller, overlapping stages. The concept of pipelining is to divide a task into smaller sub-tasks and execute them concurrently to increase the overall throughput of the system. However, one of the main challenges in pipelining is the data dependency problem.

Data dependency refers to the relationship between two instructions in a program, where one instruction requires the result of another instruction to complete its execution. In pipelining, data dependency can cause stalls in the pipeline, resulting in a decrease in overall performance.

There are three types of data dependencies:

- **RAW (Read After Write) dependency:** Occurs when a later instruction reads data from a register or memory location that has been written by an earlier instruction, before the earlier instruction has completed its execution.

- **WAR (Write After Read) dependency:** Occurs when a later instruction writes data to a register or memory location before an earlier instruction has finished reading from the same register or memory location.

- **WAW (Write After Write) dependency:** Occurs when two instructions write to the same register or memory location, and the order of execution is not preserved.

To solve the data dependency problem in pipelining, there are several techniques used:

- **Forwarding:** Also known as data forwarding or bypassing. This technique allows the results of an instruction to be forwarded to another instruction that needs the data before the result is written to a register. This technique eliminates the need for a stall in the pipeline.

- **Stalling:** Also known as bubble or NOP insertion. This technique inserts a "no-operation" instruction (NOP) into the pipeline to allow an earlier instruction to complete before a later instruction can proceed. This technique, however, decreases the overall performance of the system.

- **Compiler Techniques:** Compiler techniques can be used to reorder instructions to eliminate data dependencies. The compiler can also use register renaming to create new registers that are not shared between instructions.

- **Hardware Techniques:** Hardware techniques can be used to increase the number of registers or add additional functional units to the processor to reduce data dependencies.

Q. Describe Flynn's classification

Flynn's classification is a taxonomy used in computer architecture to categorize the structure and behavior of computer systems based on the number of instruction streams and data streams that can be processed simultaneously.

There are four classifications in Flynn's taxonomy:

- **Single Instruction Single Data (SISD):** This is the simplest and most common classification of computer architecture. In an SISD system, there is only one instruction stream and one data stream. The processor executes one instruction at a time and operates on one data item at a time.

- **Single Instruction Multiple Data (SIMD):** In a SIMD system, there is a single instruction stream that is broadcast to multiple processing units, each of which operates on different data items. This architecture is often used in parallel processing for scientific computing or multimedia applications.

- **Multiple Instruction Single Data (MISD):** In a MISD system, there are multiple instruction streams that operate on a single data stream. This architecture is less common and is mainly used in fault-tolerant systems or in highly specialized applications such as medical imaging.

- **Multiple Instruction Multiple Data (MIMD):** In a MIMD system, there are multiple instruction streams and multiple data streams. Each processing unit operates independently and can execute different instructions on different data items concurrently. This architecture is used in parallel processing for a wide range of applications, including scientific computing, data analytics, and artificial intelligence.

Q. Discuss about parallel processing? How parallel processing can be achieved in pipelining, explain it with time-space diagram for four segments pipeline having six tasks.

Parallel processing is a technique in computer architecture where multiple processors or processing units work together to perform a task. This technique can significantly improve the speed and performance of computer systems by dividing a large task into smaller sub-tasks that can be executed concurrently.

Parallel processing can be achieved in several ways, such as pipelining, multi-core processors, multi-processor systems, and distributed computing.

Pipelining is a form of parallel processing that involves breaking down the execution of a task into smaller, overlapping stages. Each stage of the pipeline operates on a different subset of data or instructions, and the outputs of each stage are passed to the next stage as soon as they are available. This overlapping of stages can result in significant performance improvements.

(4 stage pipelining with 6 task is explained above in the note pg 4)

Hazards (Youtube link)

Hazards are the situations, which prevent the next instruction in the instruction stream from executing during the desired clock cycle. They reduce the performance gained from the ideal speedup by pipelining.

Data hazards

Data hazards are a type of hazard that can occur in pipelining when an instruction depends on the result of a previous instruction that has not yet completed execution. In other words, the data that

the second instruction needs is not yet available because the first instruction has not yet finished writing it to a register or memory.

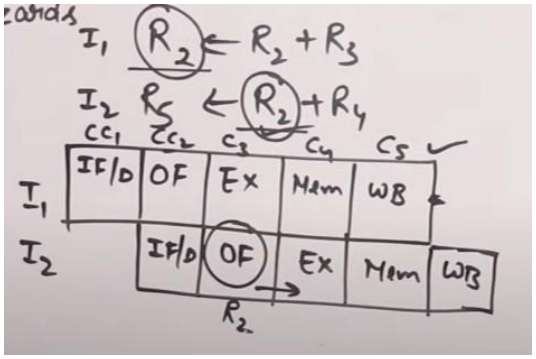
This can cause the pipeline to stall or result in incorrect output. For example, if the second instruction needs the result of an addition operation performed by the first instruction, and the pipeline does not handle this dependency properly, the second instruction might start executing before the result is available. As a result, the second instruction might use the old value in the register, leading to incorrect output.

To avoid data hazards:

- **Forwarding/Bypassing:** In this technique, the data is passed directly from the output of the instruction that produces it to the input of the instruction that needs it. This allows the second instruction to use the updated data without waiting for the data to be written to the register or memory.

- **Register Renaming:** This technique involves renaming registers to eliminate data dependencies. By assigning a new name to a register, instructions can be reordered and scheduled without causing data hazards.

- **Compiler Techniques:** By analyzing the code at compile time, the compiler can rearrange instructions or insert "dummy" instructions to break data dependencies and avoid data hazards.

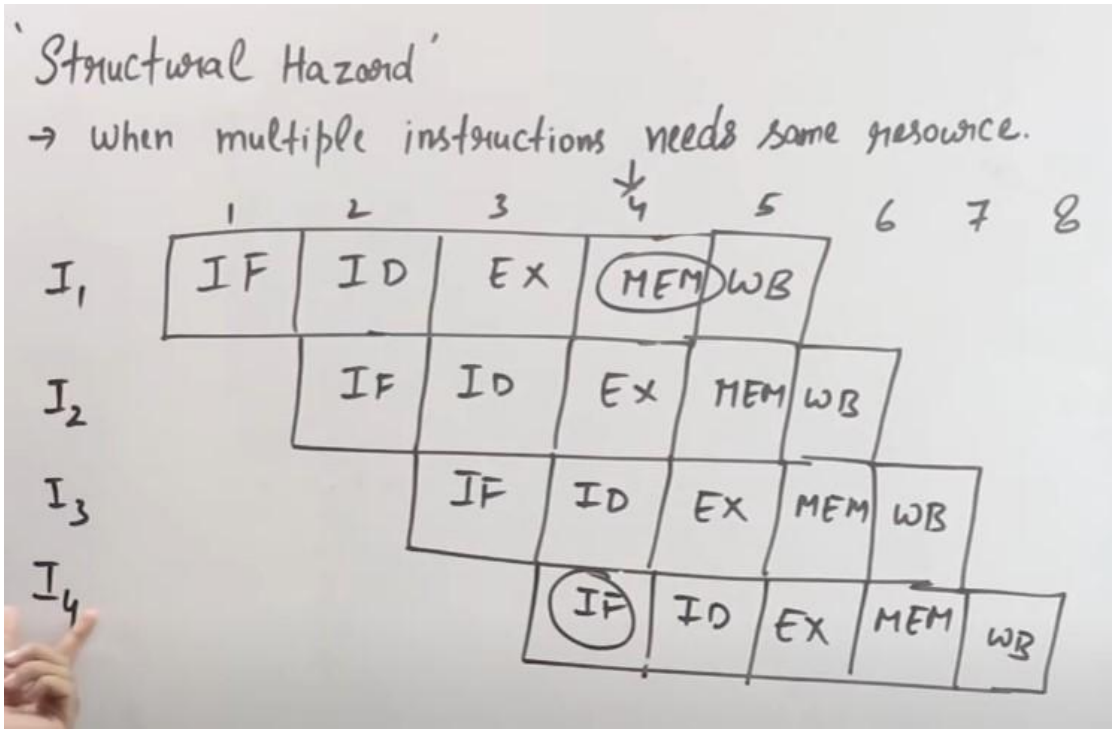


Here OF for I1 is done in cc2 and we are trying to use OF value in cc3 for I2. The OF value for I1 is not stored in regsiter as it is not completed execution. It will complete exection afte WB. Hence we are not using updated data which causes data harzard.

Structural Hazards

Structural hazard occur when multiple instructions needs same resources. These resources may include processor registers, functional units, or memory modules.

For example, consider a CPU that has a single adder unit and two instructions that require the use of the adder unit. If the two instructions are scheduled to execute at the same time, a structural hazard occurs because the adder unit can only be used by one instruction at a time. In this case, one of the instructions will have to wait for the adder unit to become available, which can result in a delay in the execution of the program.



Here at cc4 for I4 we are accessing memory for IF and at the same time for I1 we are accessing the memory so it creates structural hazard because we are accessing the same memory.

There are several solutions to structural hazards in computer architecture. Some of the common techniques used to overcome structural hazards include:

- **Resource Duplication:** This technique involves duplicating hardware resources such as functional units or registers to eliminate the structural hazard. For example, if a CPU has only one adder unit and multiple instructions require the use of the adder unit, the CPU can be designed with multiple adder units to allow the instructions to execute concurrently.

- **Resource Partitioning:** Resource partitioning involves dividing the available hardware resources into smaller units and assigning each unit to a specific set of instructions. This technique allows multiple instructions to use the same resource without interfering with one another. For example, a CPU can be partitioned into multiple processing units, each with its own set of registers and functional units.

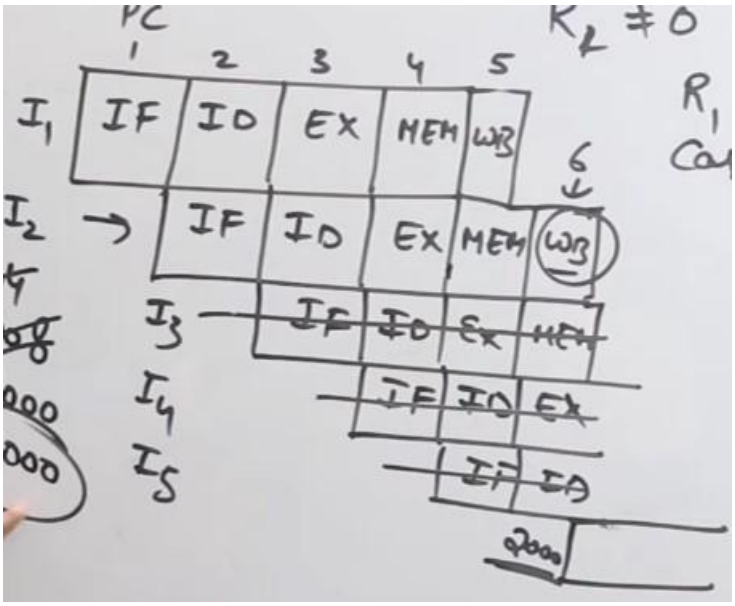
- **Delayed Execution:** Delayed execution involves delaying the execution of an instruction until the required hardware resources become available. This technique can be used when the structural hazard cannot be eliminated by duplicating or partitioning the resources.

- **Compiler Optimization:** Compiler optimization involves optimizing the code generated by the compiler to reduce the number of instructions that require the same hardware resources. This technique can be used to minimize the occurrence of structural hazards in the code.

Control Hazards

All instruction who change the program counter leads to control hazards.

Control hazards arise when instructions are executed out of order, and a conditional branch instruction is encountered. If the branch condition is not yet known, the processor has to wait for the condition to be resolved before it can determine the next instruction to be executed. This delay in determining the next instruction can lead to performance degradation and reduce the overall efficiency of the CPU.



Here we find we have to go to 2000 after ID of I2 and after WB of I2 we go to 2000 address due to brancing instruction so we have to flush I3,I4,I5 instructions which hampers performance.

There are several techniques used to mitigate control hazards, including:

- **Branch Prediction:** Branch prediction is a technique that predicts the outcome of a conditional branch instruction before it is executed, based on the behavior of previous executions of the same instruction. The predicted outcome is used to speculatively execute the instructions following the branch instruction. If the prediction is correct, the speculatively executed instructions are retained, and if the prediction is incorrect, they are discarded.

- **Delayed Branch:** A delayed branch involves adding one or more instructions between the branch instruction and the target instruction. This technique allows the processor to execute instructions following the branch instruction while waiting for the branch condition to be resolved.

- **Branch Target Buffer:** A branch target buffer is a cache that stores the target address of recently executed branch instructions. When a branch instruction is encountered, the branch target buffer is searched for the target address. If the address is found, the processor can directly jump to the target address, reducing the delay in determining the next instruction.

- **Speculative Execution:** Speculative execution is a technique that allows the processor to execute instructions ahead of the current program flow, based on a predicted outcome of a conditional branch instruction. The instructions are executed speculatively, and if the predicted outcome is incorrect, the speculatively executed instructions are discarded.

Q. What is vector processing?

Vector processing is a type of computer architecture that allows a single instruction to operate on multiple data elements simultaneously. In other words, it enables a processor to perform the same operation on multiple pieces of data at the same time, by grouping the data into vectors or arrays.

Alternate Method of Unsigned Binary Multiplication

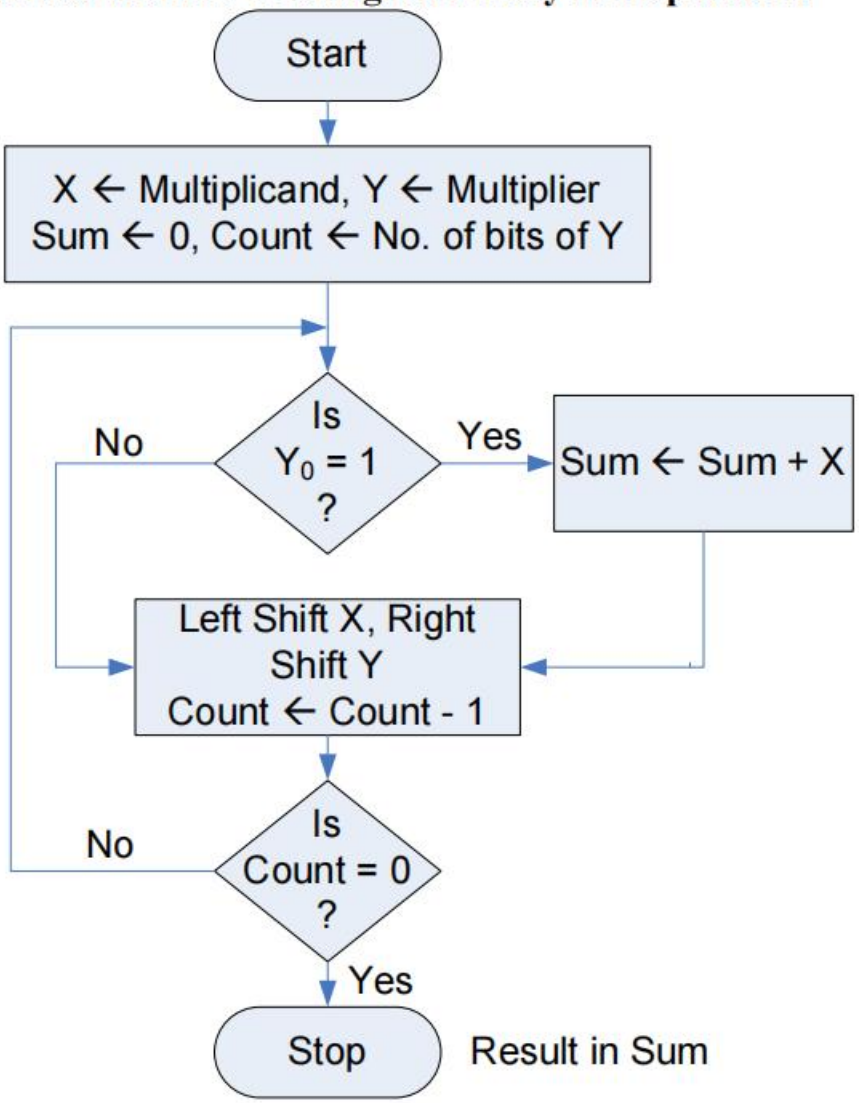


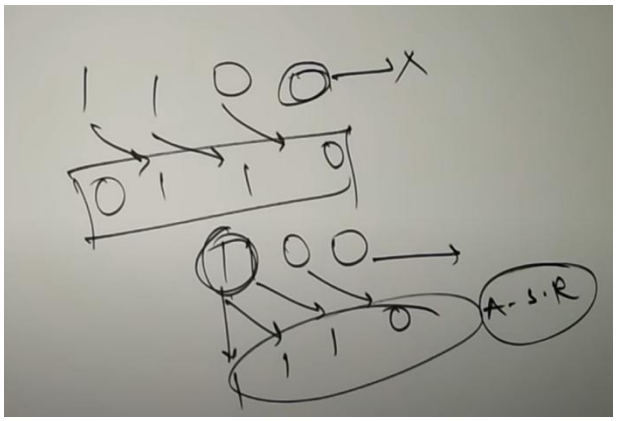
Fig: Unsigned Binary Multiplication Alternate method

Example: Multiply 7 X 6

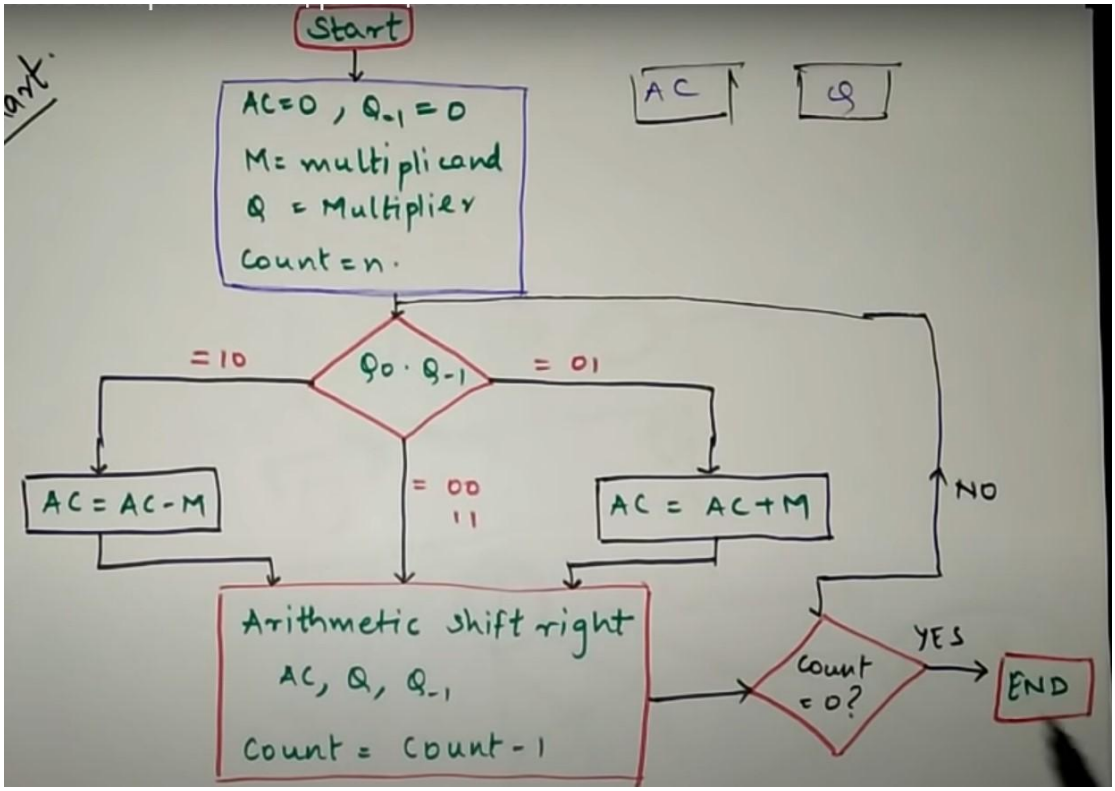
Sum	X	Y	Count	Remarks
000000	000111	110	3	Initialization
000000	001110	011	2	Left shift X, Right Shift Y
001110	011100	001	1	Sum ← Sum + X, Left shift X, Right Shift Y
101010	111000	000	0	Sum ← Sum + X, Left shift X, Right Shift Y

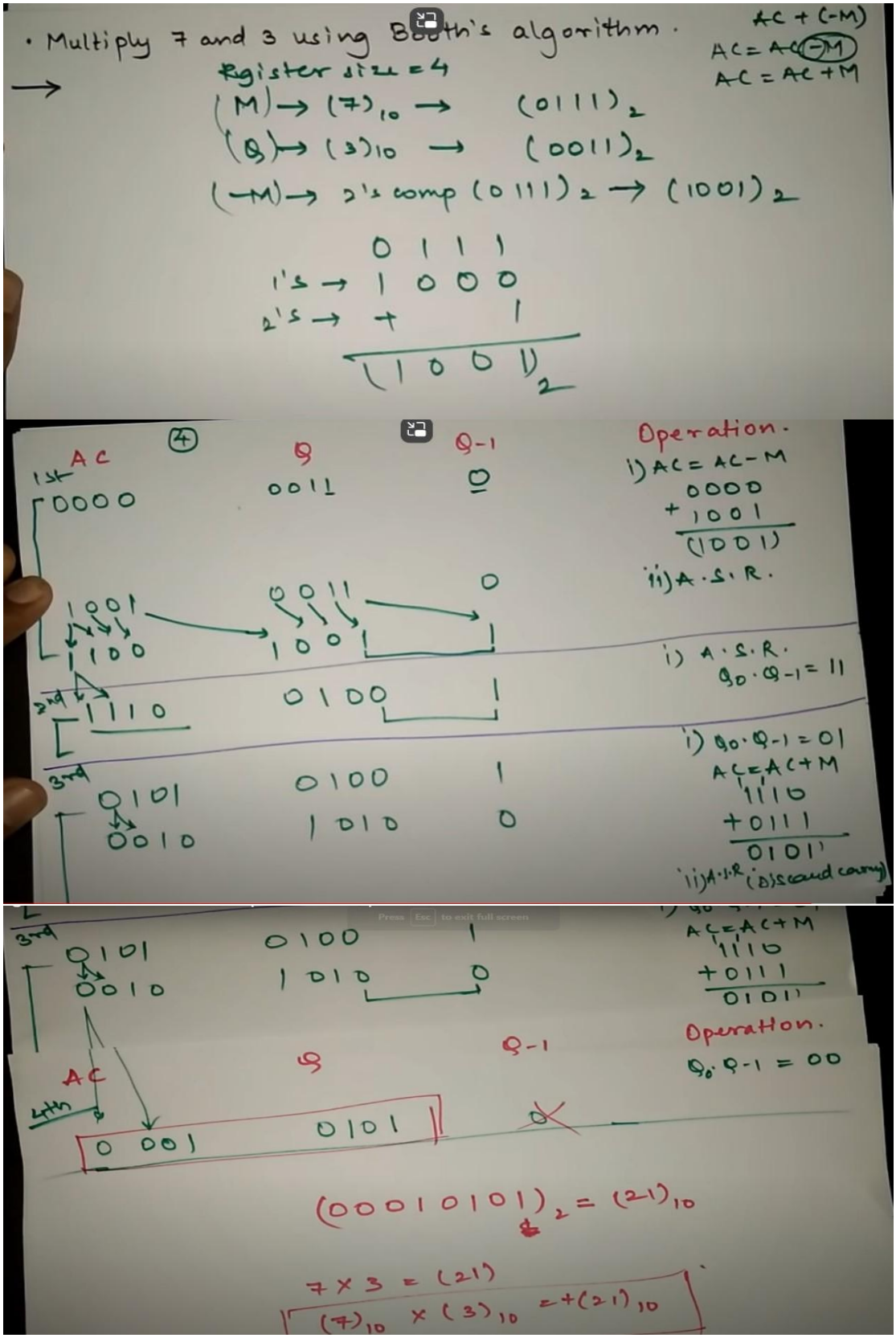
Result = 101010 = $2^5 + 2^3 + 2^1 = 42$

Signed Multiplication (Booth Algorithm) – 2's Complement Multiplication



Normal vs arithmetic shift right





(4 bits register - 4 cycles how many bits that many cycles.)

orithm with Solved Example in Hindi | part 2| COA Lectures

BOOTH'S ALGORITHM

PART 02 - (+ve) × (-ve) / (-ve) × (+ve)

+ (Number 1) × − (Number 2)

= − (Product).

• Multiply -7 and +3 using Booth's algorithm.
register bits = 5

→

00111
11000
+ 1
11001

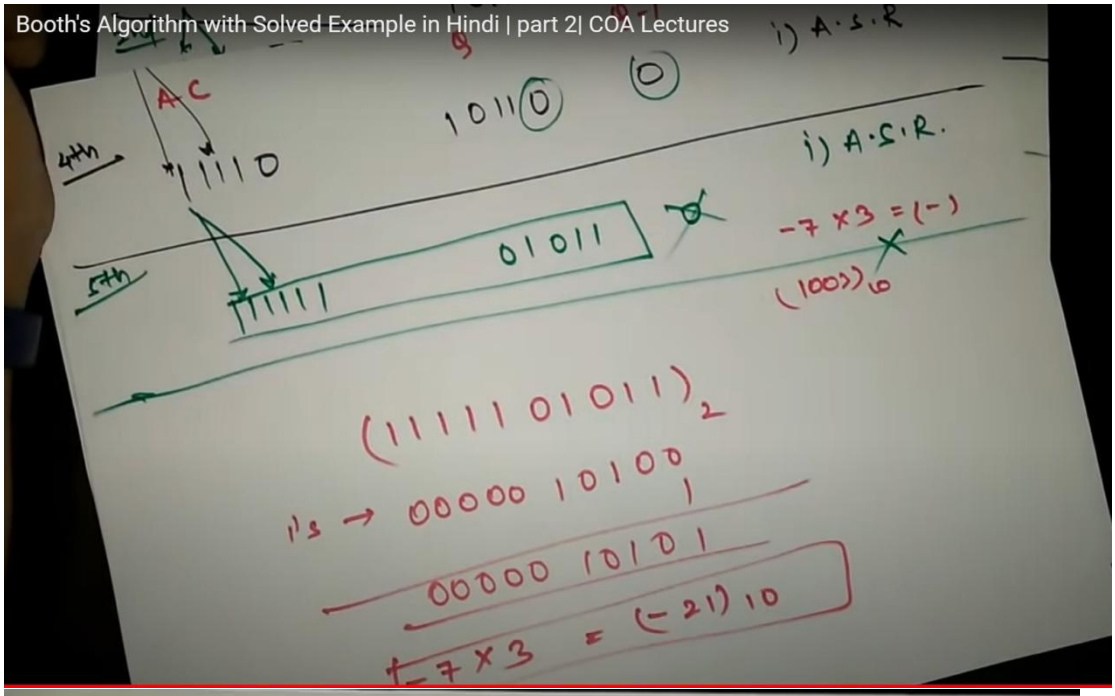
M → $(-7)_{10}$ → $(11001)_2$

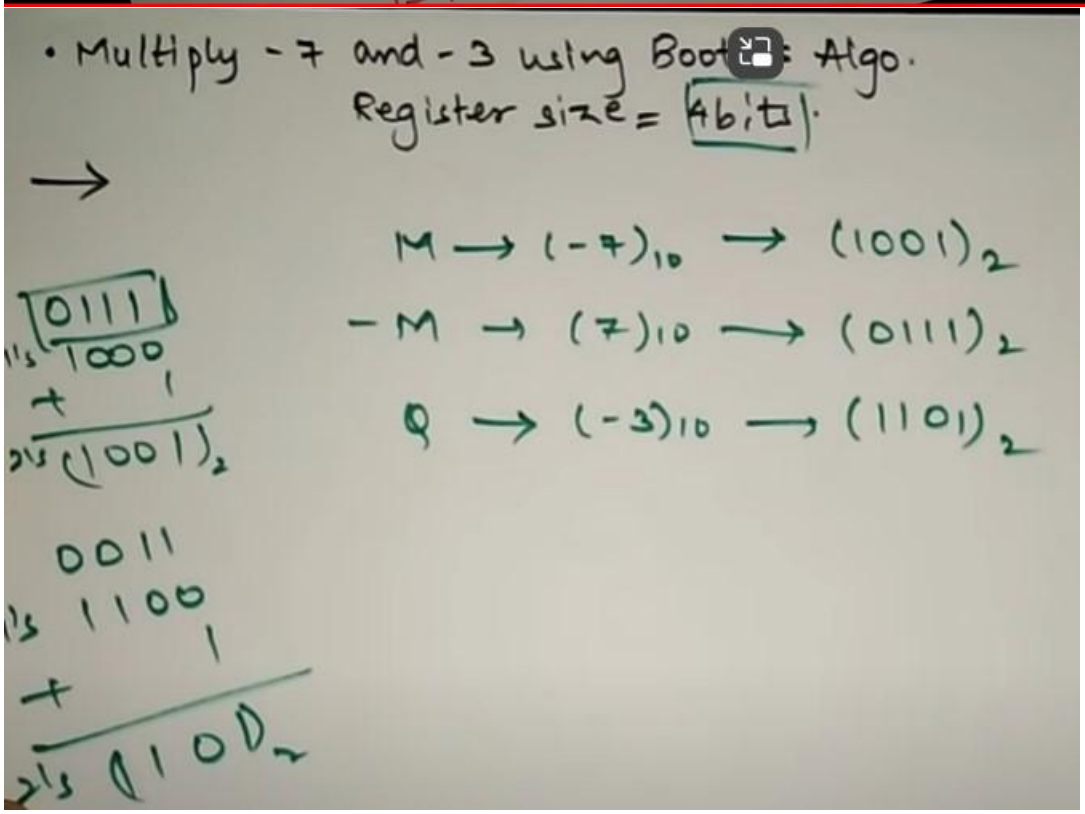
Q → $(3)_{10}$ → $(00011)_2$

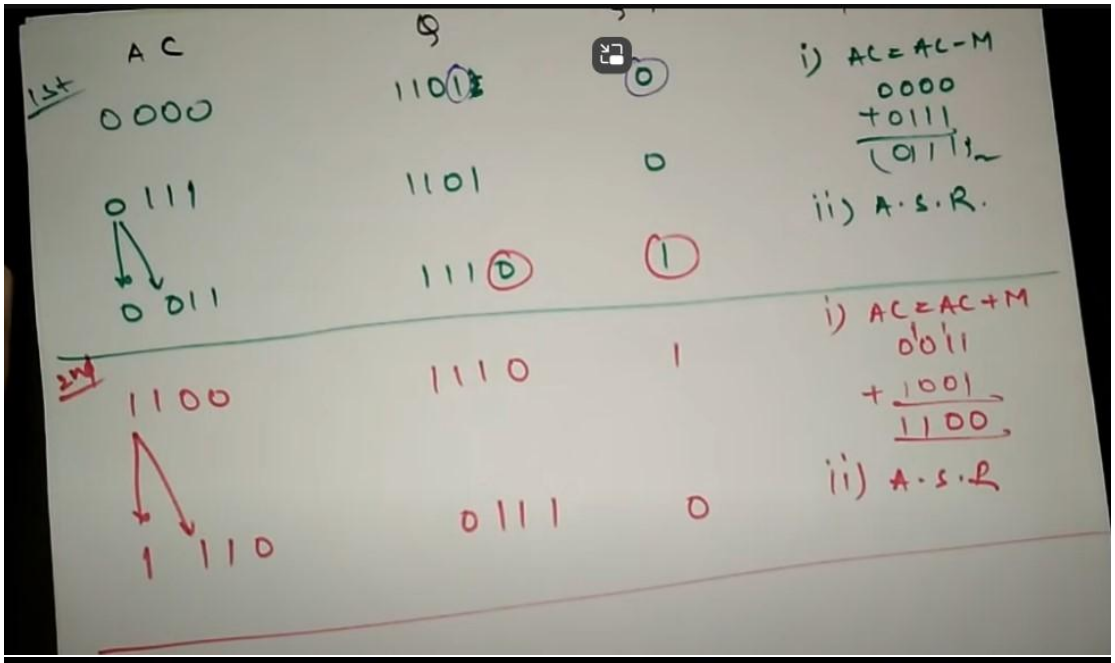
(−M) → $(7)_{10}$ → $(00111)_2$

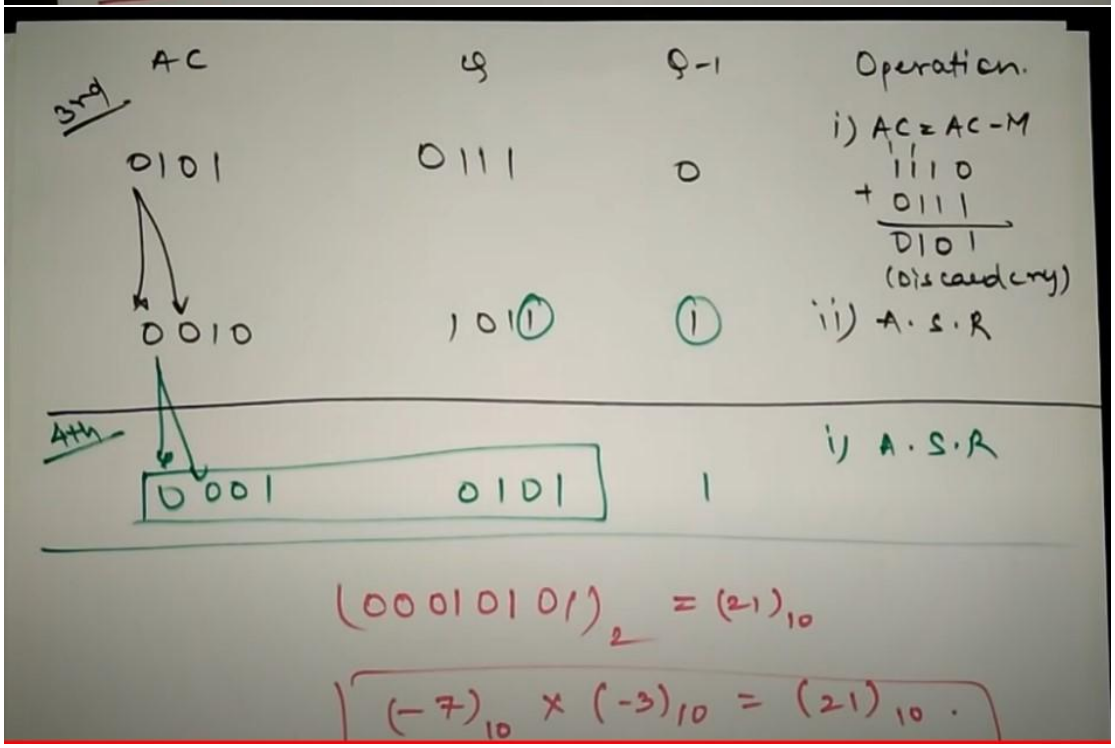
(5)

	AC	Q	Q_{-1}	Operation
1st	00000	0001	1	0 i) AC = AC − M = 00000 + 00111 = 00111
	00111	00011	0	ii) A.S.R
	00011	10001	1	
2nd	00001	11000	1	i) A.S.R.
3rd	11010	11000	1	i) AC = AC + M = 00001 + 11001 = 11010
	11101	01100	0	ii) A.S.R.


Booth's Algorithm with Solved Example in Hindi | part 2| COA Lectures
i) A.S.R
AC
1011 0
4th
1110
i) A.S.R.
01011
5th
11111
-7 x 3 = (-)
(100)10
(111110101 1)2
1's → 00000 10100
1
00000 10101
-7 x 3 = (-21)10


• Multiply -7 and -3 using Booth Algo.
Register size = 4bit.
M → (-7)10 → (1001)2
-M → (7)10 → (0111)2
Q → (-3)10 → (1101)2
0111
1's 1000
+ 1
1001
0011
1's 1100
+ 1



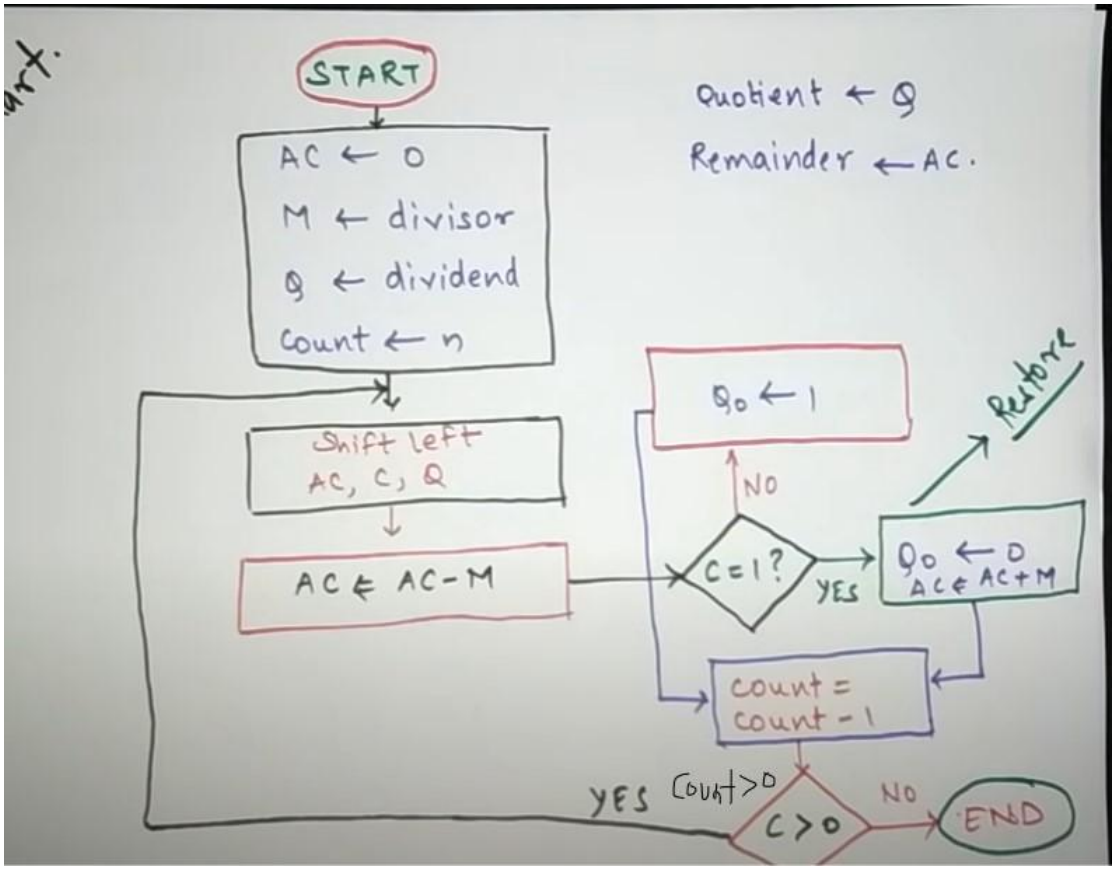


$(00010101)_2 = (21)_{10}$

$(-7)_{10} \times (-3)_{10} = (21)_{10}$.

RESTORING DIVISION
(PART-01)

- Restoring division algorithm uses:
  - Left shift
  - Subtraction
  - carry control bit (using NOT gate).



erform division of the following numbers using restoring division.

Dividend = 17 $\quad 17 \div 3$ $\quad AC = AC - M$ / $AC + M$

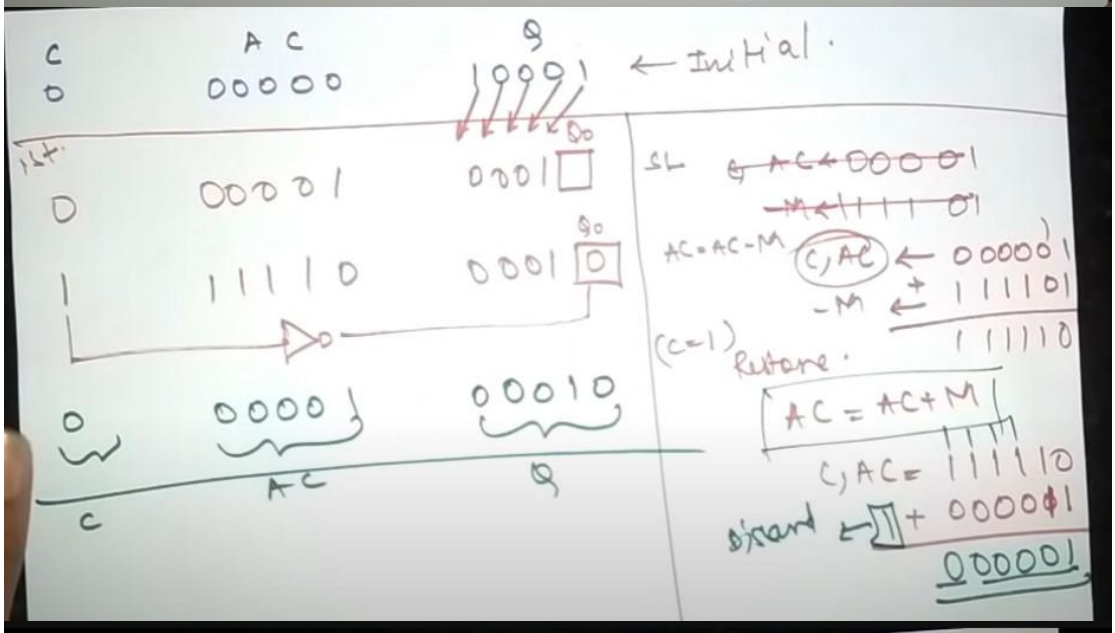
Divisor = 03.

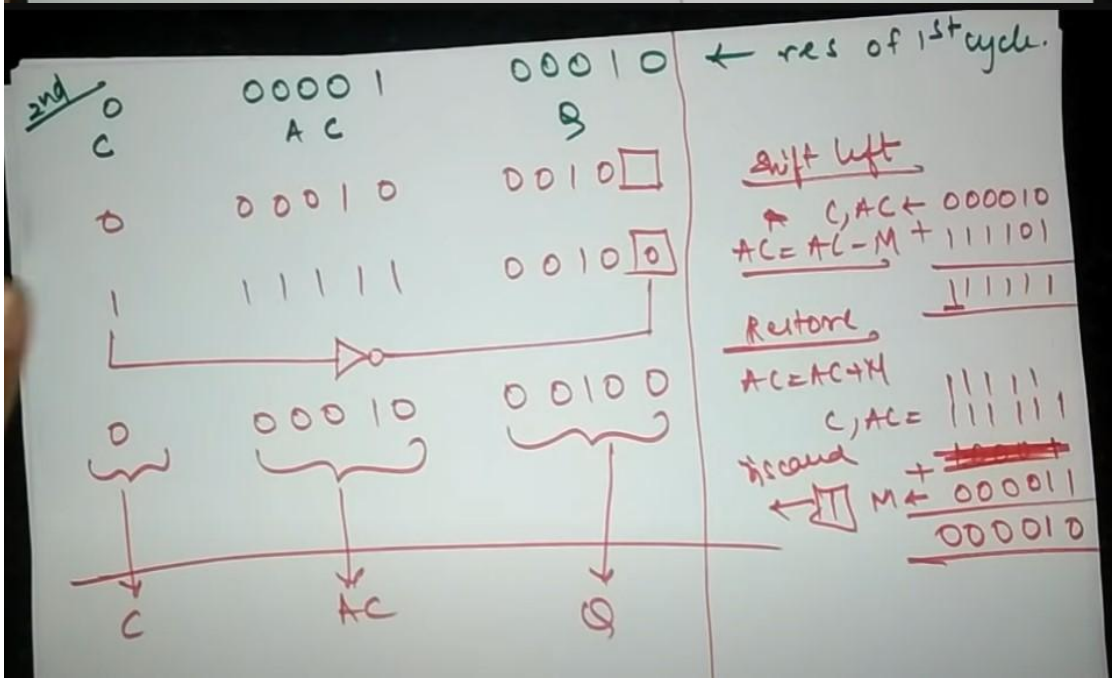
$Q \leftarrow (10001)_2$

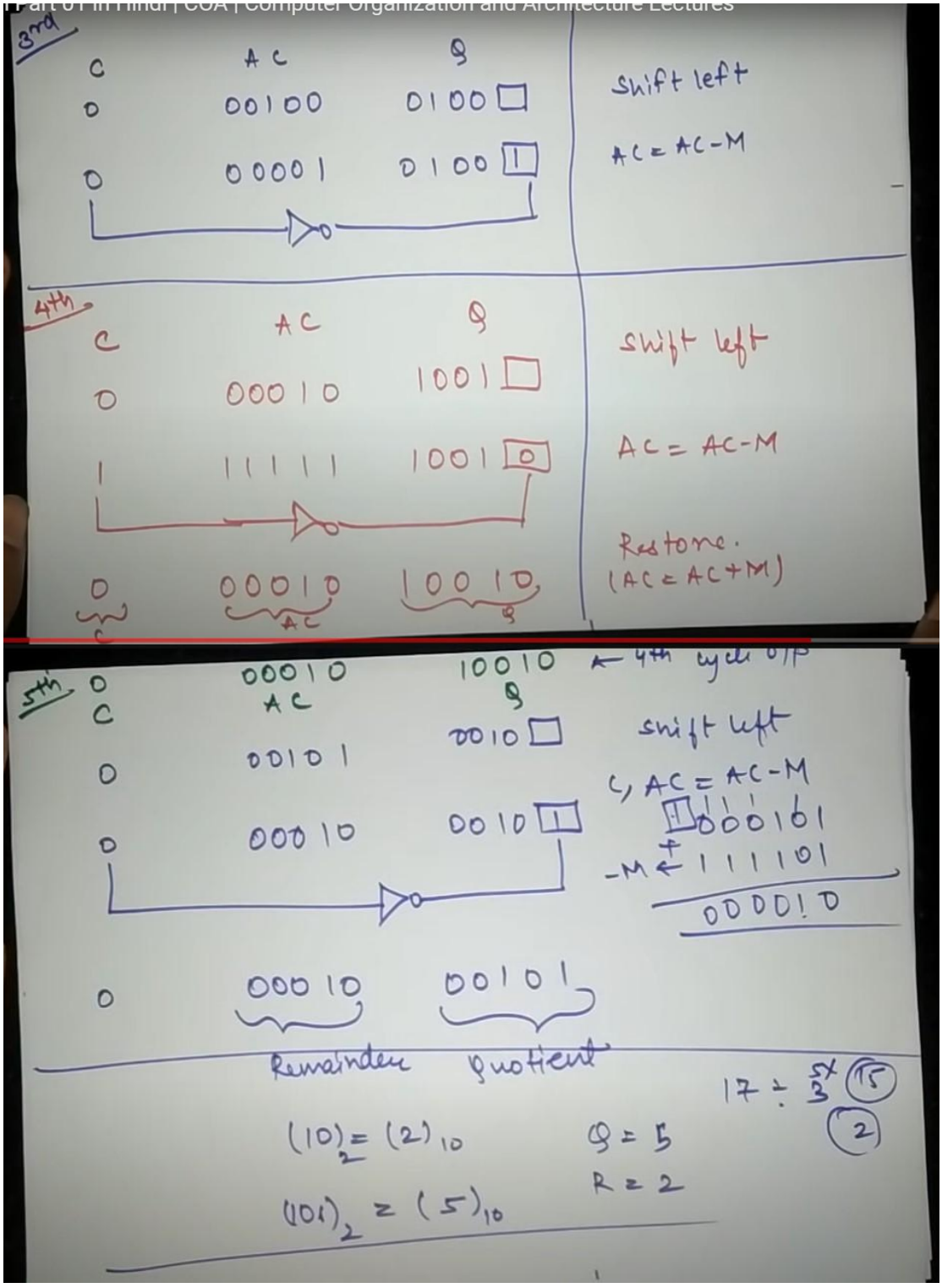
$M \leftarrow (000011)_2$ $\quad |-M = 111100 + 1 = 111101$

$n = 5$ (5 cycles).

(n+1) bits for handling borrow







Remainder = CAC o combo not AC only

2. Perform division of the following :

$(1100)_2 \div (11)_2$

→ Q ← 1100

M ← 00011

to handle borrow

4 cycles.

-M = 11100 ← 1's

\+ 1

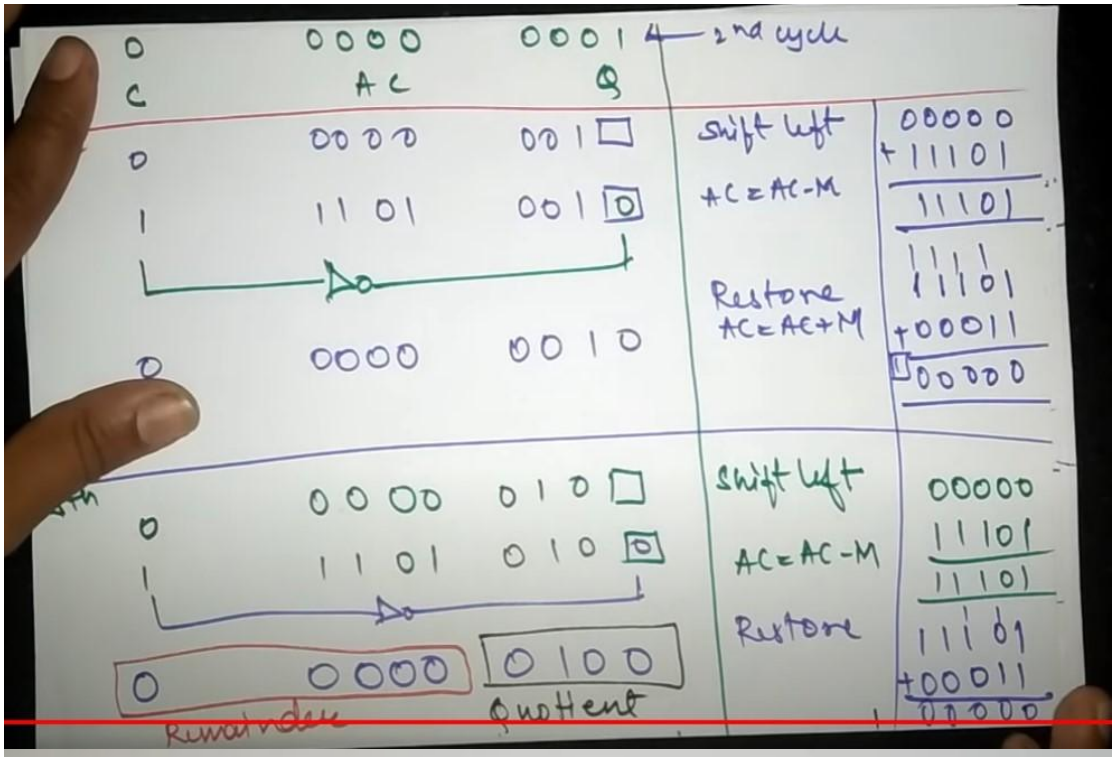
(11101) ← 2's

C	AC	Q	
0	0000	1100	Initial
0	0001	100□	Shift left AC=AC-M
1	1110	1000	Restore.
0	0001	1000	
2nd 0	0011	000□	Shift left AC=AC-M
0	0000	0001	

00001 + 11101 = 11110

11110 + 00011 = 00001

00011 + 11101 = 00000



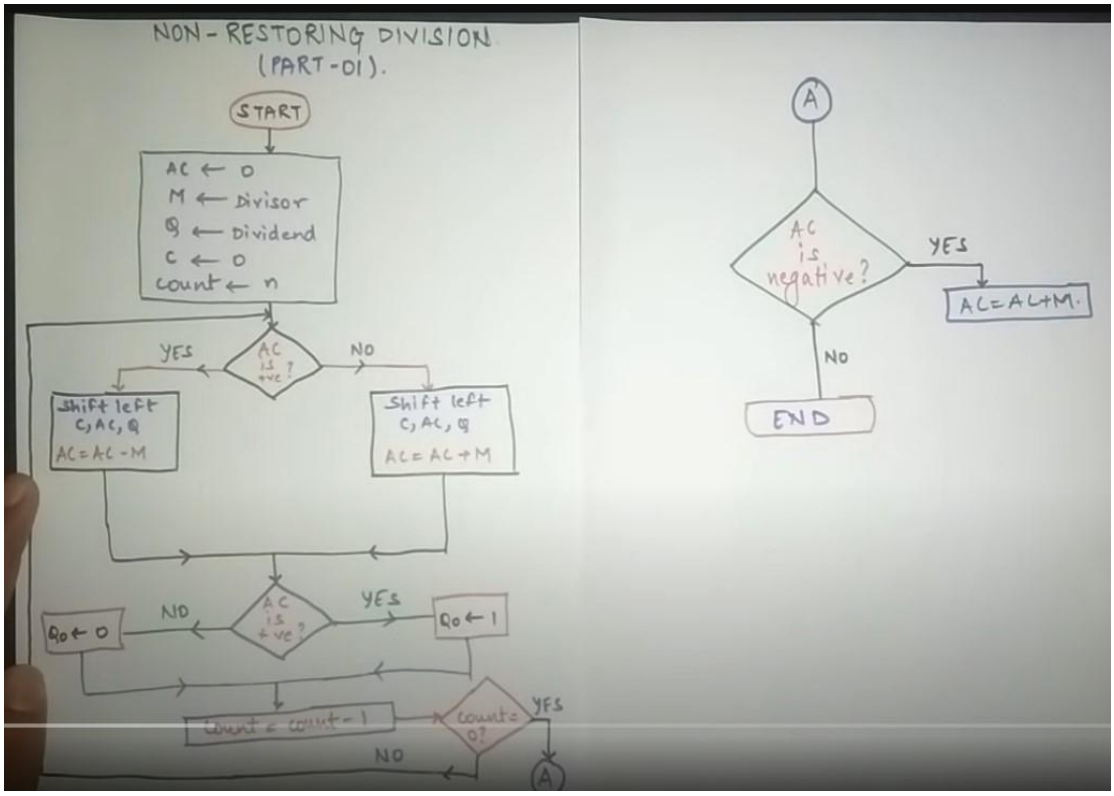
$R \leftarrow (00000)_2 = (0)_{10}$

$Q \leftarrow (0100)_2 = (4)_{10}$

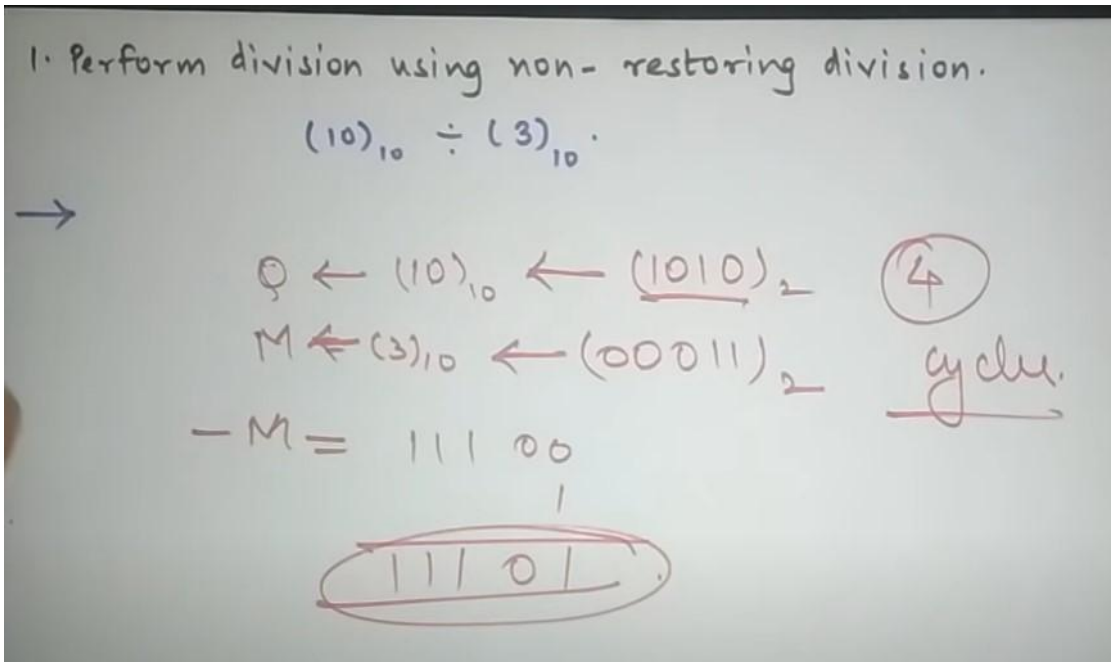
$1100 \div 11$

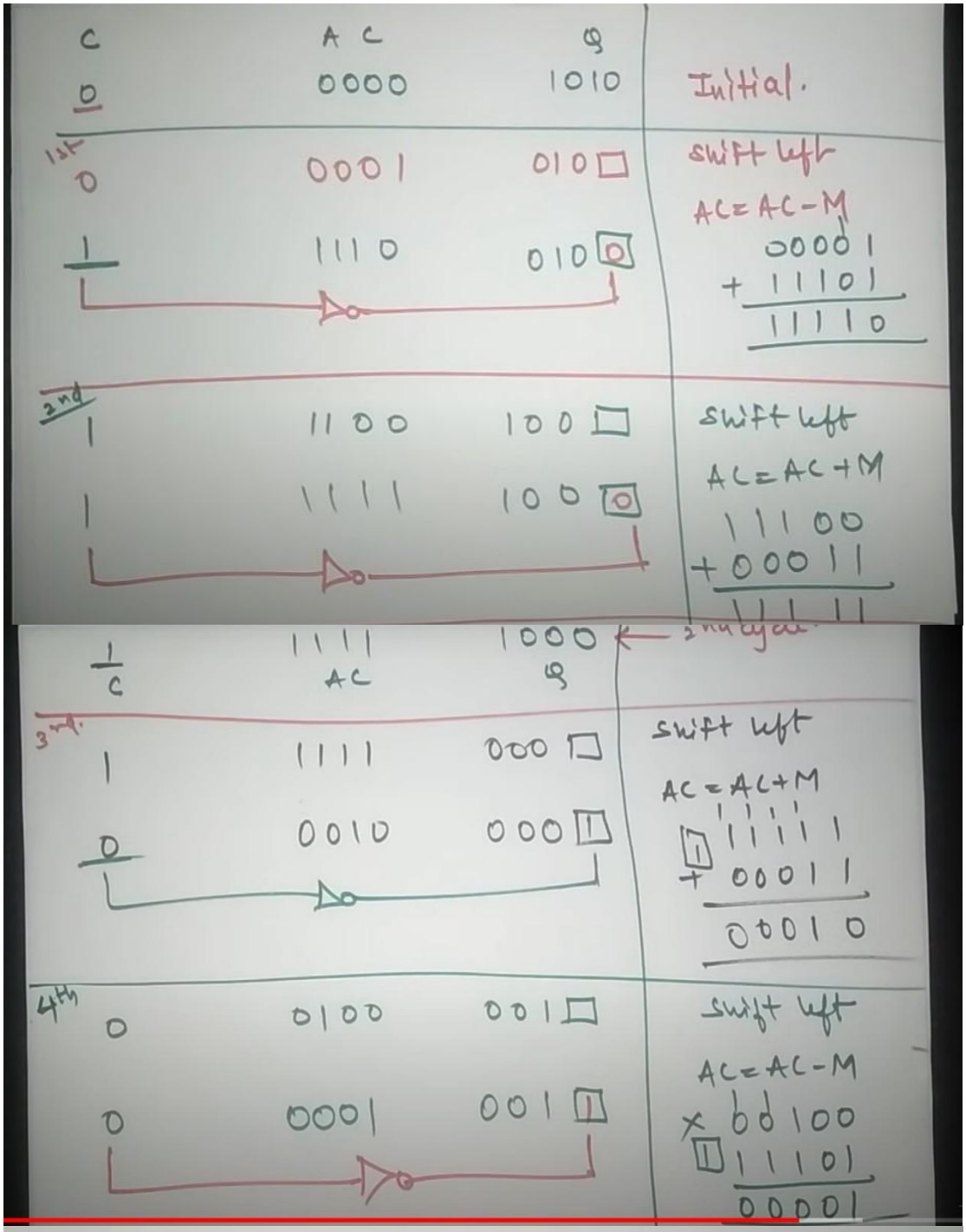
$(12)_{10} \div (3)_{10} = (4)_{10}$

$R \leftarrow (0)_{10}$



Positive c ko vaue 0 negative c ko value 1





$R = (00001)_2 = (1)_{10}$

$Q = (0011)_2 = (3)_{10}$

$(10) \div (3)$

Quotient $= (3)_{10}$
Remainder $= (1)_{10}$

until the most significant bit is nonzero.

Example:

Let us take: $X=09504*10^3$ and $Y=0.8200*10^2$

5. Compare exponents by subtraction

   3-2=1

   Larger exponent is choose as the exponents of the result

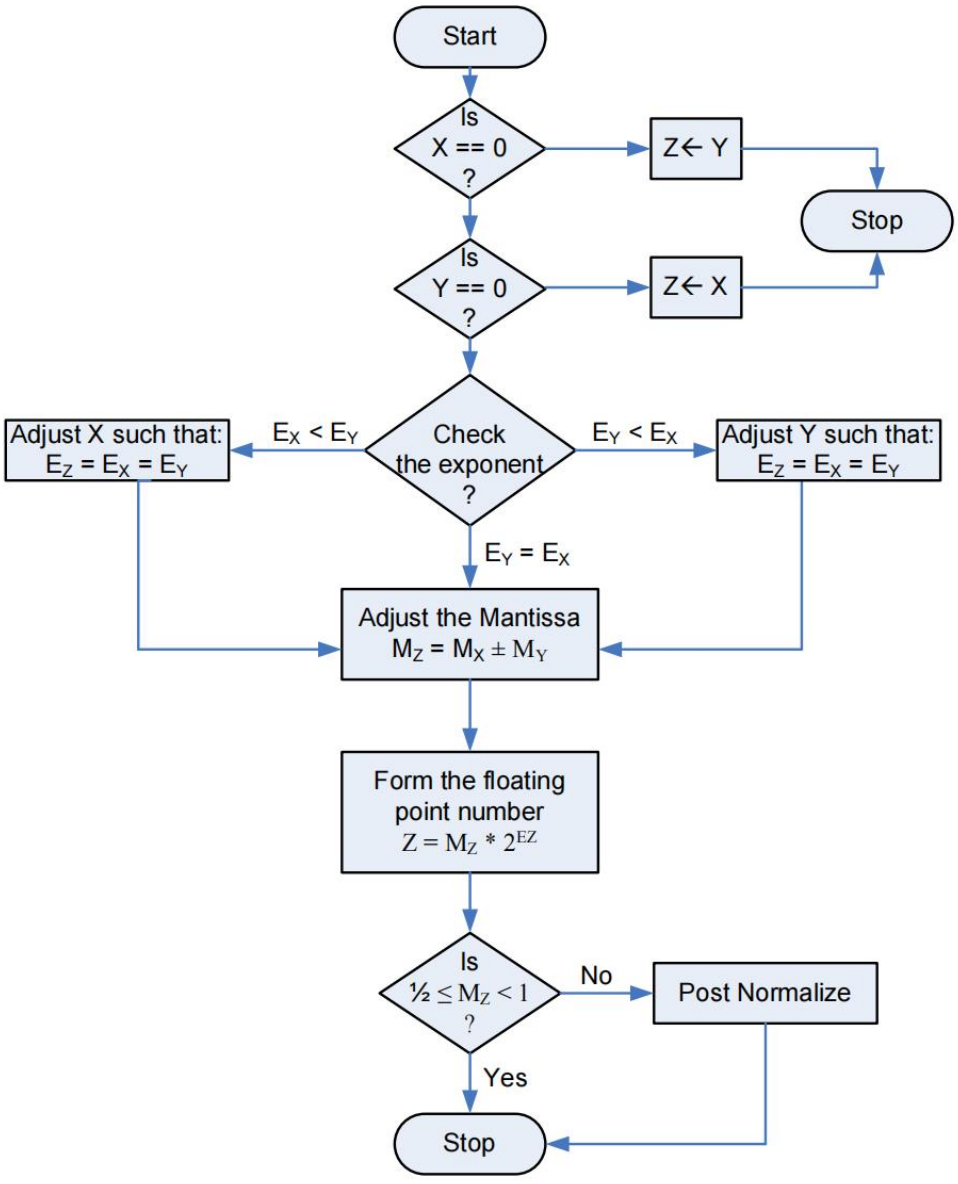
6. Align mantissas

   $X=0.9504*10^3$

   $0.08200*10^3$

   Add mantissas

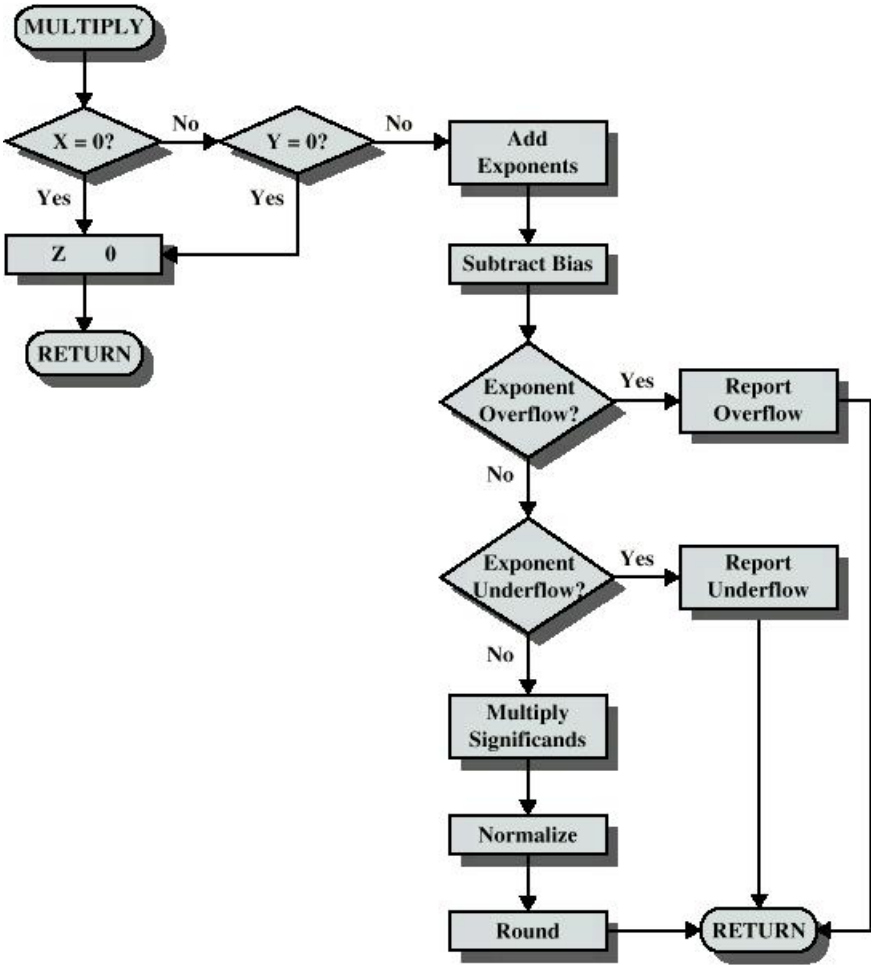
   $Z=1.0324*10^3$



Floating Point Multiplication

The multiplication can be subdivided into 4 parts.

1. Check for zeroes.
2. Add the exponents.
3. Multiply mantissa.
4. Normalize the product.



Example:

$X = 0.101 * 2^{8}$

$Y = 0.1001 * 2^{-3}$

As we know, $Z = X * Y = (Mx * My) * 2^{(EX+EY)}$

$Z = (0.101 * 0.1001) * 2^{(8-3)}$

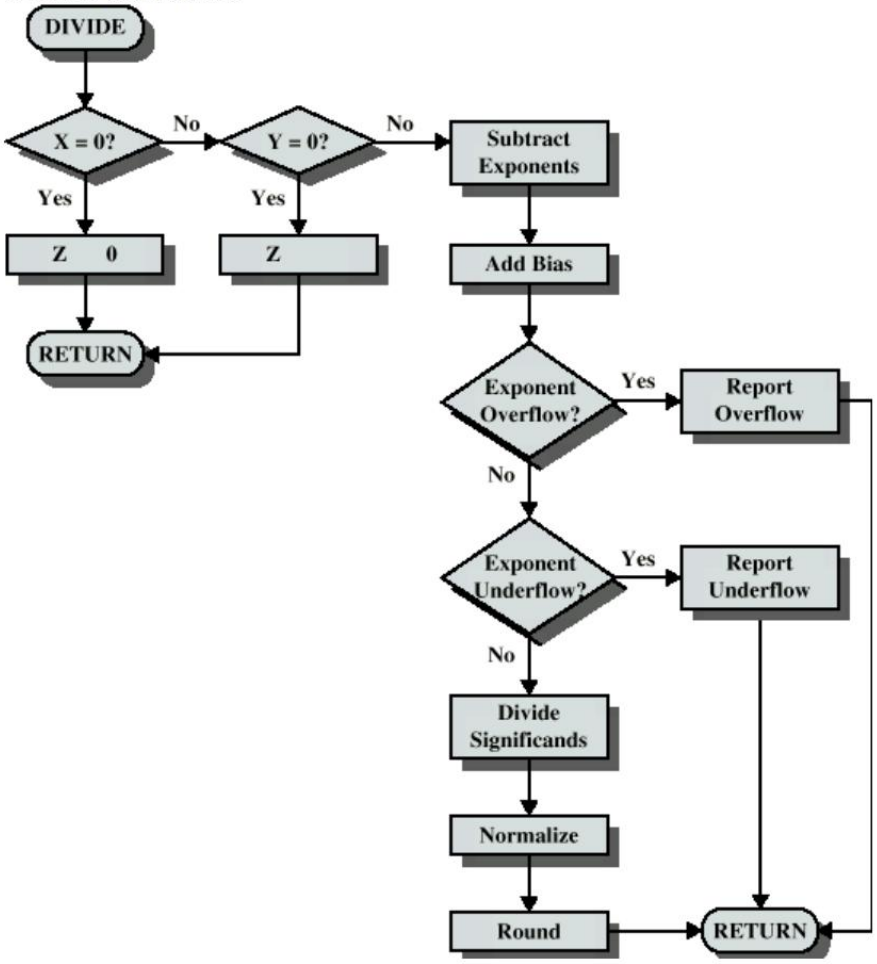
$= 0.0101101 * 2^{5}$

$= 0.101101 * 2^{5}$ (Normalized result)

Floating Point Division

The division algorithm can be subdivided into 5 parts

1. Check for zeroes.
2. Initial registers and evaluates the sign.
3. Align the dividend.
4. Subtract the exponent.
5. Divide the mantissa.



Example:

$X = 0.101 * 2^{8}$ and $Y = 0.1001 * 2^{-3}$

As we know, $Z = X / Y = (Mx / My) / 2^{(Ex-Ey)}$

$Mx / My = (0.101/0.1001) = 1.00011$

$Ex - Ey = 8+3 = 11$

Now, $Z = 1.00011 * 2^{11}$ (Normalized result)

CHAPTER 6 (8 Marks)

MEMORY

Memory is a critical component of a microcomputer system, which stores binary instructions and data. It is where the computer holds current programs and data in use. The main memory is the memory unit that communicates directly with the CPU, while auxiliary or secondary memory devices provide backup storage.

Characteristics of memory:

- **Location:** The location of memory refers to where it is physically located in a computer system. There are two main types of memory locations: internal and external.
  - **Internal memory:** This is memory that is located inside the computer system. Examples include RAM (Random Access Memory) and Cache memory.
  - **External memory**: This is memory that is located outside the computer system. Examples include hard drives, CDs, and flash drives.
- **Capacity:** The capacity of memory refers to the amount of data it can hold. Different types of memory have different capacities.
  - **RAM:** The capacity of RAM is measured in gigabytes (GB) and typically ranges from 2 GB to 32 GB or more.
  - **Hard Drives**: The capacity of hard drives is measured in terabytes (TB) and typically ranges from 500 GB to 4 TB or more.
- **Unit of transfer:** The unit of transfer refers to the amount of data that can be read from or written to memory at one time.

- **Cache memory:** The unit of transfer for cache memory is typically one cache line, which is usually 64 bytes.

- **RAM:** The unit of transfer for RAM is usually one word, which is typically 8 bytes.

➢ **Access method:** The access method refers to how data is retrieved from memory. There are two main access methods: random access and sequential access.

- **Random access:** This allows data to be accessed in any order. RAM is an example of random access memory.

- **Sequential access:** This requires data to be accessed in a specific order. Magnetic tape is an example of sequential access memory.

➢ **Performance:** The performance of memory refers to how quickly data can be read from or written to it.

- **Cache memory**: This is the fastest type of memory, with access times typically measured in nanoseconds (ns).

- **RAM:** This is slower than cache memory, with access times typically measured in microseconds (μs).

➢ **Physical characteristics:**

- **Volatile memory** is a type of memory that loses its stored information when the power is turned off. RAM (Random Access Memory) is a typical example of volatile memory.

- **Non-volatile memory**, on the other hand, retains its stored information even when the power is turned off. Examples of non-volatile memory include ROM (Read-Only Memory), PROM (Programmable Read-Only Memory), EPROM

(Erasable Programmable Read-Only Memory), and EEPROM (Electrically Erasable Programmable Read-Only Memory).

- **Non-erasable memory** is a type of memory that cannot be modified or altered once it is programmed. ROM is an example of non-erasable memory because its contents cannot be changed after it has been manufactured

➢ **Organisation:**

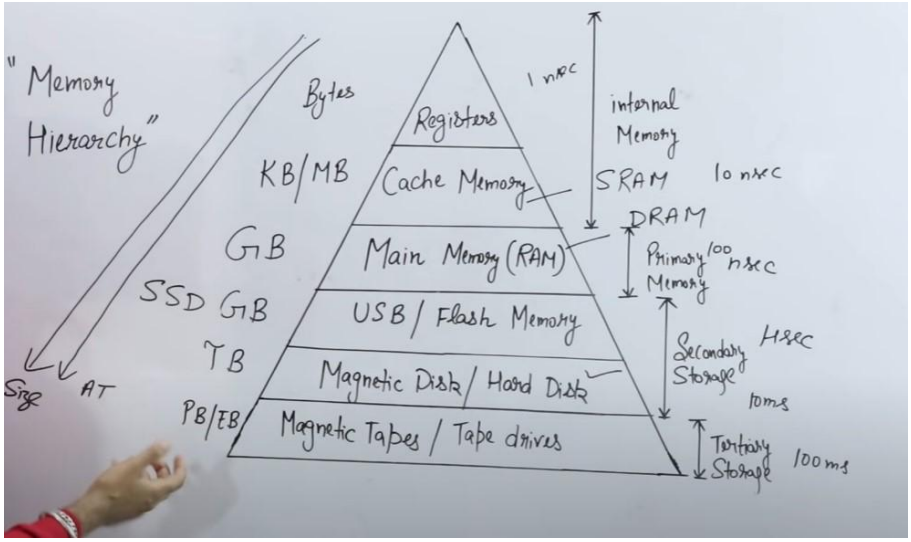
- Organization refers to the physical arrangement of bits into words or blocks.

- The choice of organization depends on factors such as performance requirements, type of data being stored, and cost constraints.

- Common memory organizations include byte-addressable, word-addressable, and bit-addressable.

- Interleaving can be used to divide memory into multiple banks for simultaneous access, improving memory bandwidth and reducing access latency.

Memory Hierarchy



1. **Registers:** These are the fastest and smallest storage locations in a computer system, used to hold data that is currently being processed by the CPU.

2. **L1 Cache:** This is a small amount of memory that is integrated into the CPU chip and used to store data that is frequently accessed by the CPU.

3. **L2 Cache:** This is a larger amount of memory that is located on the CPU chip or on a separate chip and used to store data that is frequently accessed but not as frequently as data stored in L1 cache.

4. **Main memory (RAM):** This is the primary system memory that holds data and instructions that the CPU accesses frequently. It is slower than cache memory but faster than secondary storage.

5. **Disk Cache:** This is a portion of RAM that is used to temporarily store data that is being read from or written to a hard disk drive, in order to improve performance.

6. **Hard Disk Drive**: This is a non-volatile storage device that stores data on spinning magnetic disks. It provides large storage capacity at a relatively low cost, but is slower than main memory.

7. **Optical Storage:** This includes CD-ROMs, DVDs, and Blu-ray discs. These are non-volatile storage devices that store data using lasers to read and write information on a reflective surface.

8. **Tape Storage:** This is a non-volatile storage medium that uses magnetic tape to store data. It is used for long-term storage and archiving, but is very slow and has limited random access capabilities.

Memory hierarchy refers to the way that computer systems organize and manage different types of memory, from the smallest and fastest registers to the largest and slowest secondary storage devices.

The main reason for having a memory hierarchy is to balance the conflicting requirements of speed, cost, and capacity. Different types of memory have different trade-offs between these factors. Registers, for example, are very fast but very small, while hard disks are slower but have much larger capacity.

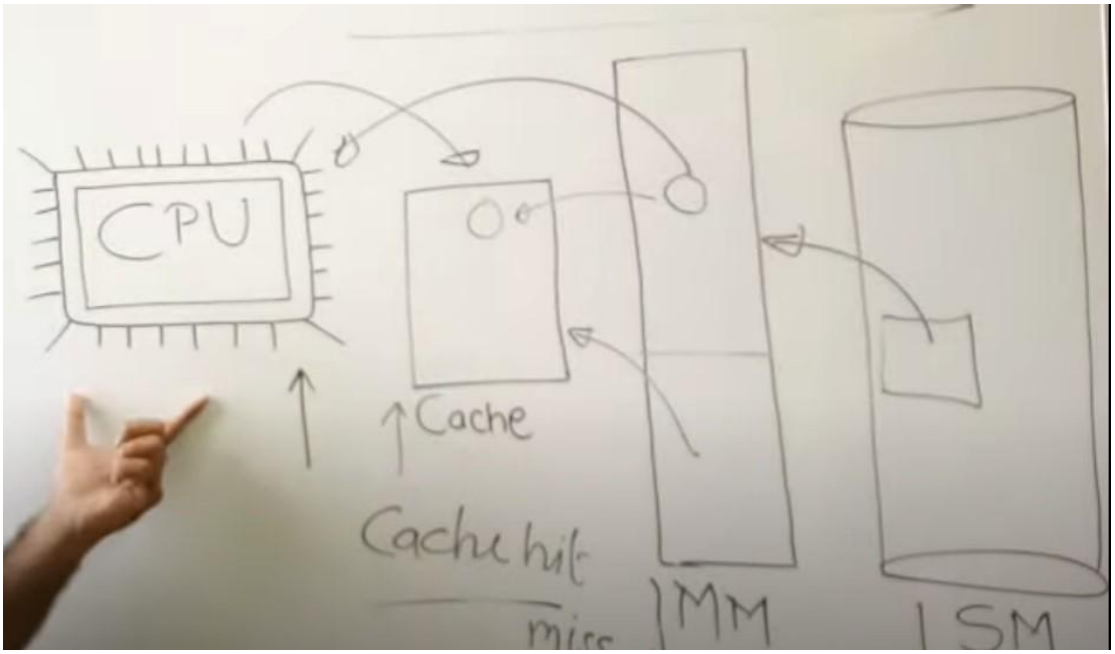
By organizing memory into a hierarchy, computer systems can take advantage of the strengths of each type of memory while minimizing the impact of its weaknesses. The highest levels of the memory hierarchy (registers and cache memory) are used for storing data that needs to be accessed frequently and quickly, while the lower levels (main memory and secondary storage) are used for storing data that is accessed less frequently or can tolerate longer access times.

The use of a memory hierarchy is an important aspect of computer architecture, as it enables systems to achieve high performance while still being cost-effective and scalable.

SRAM	DRAM
Retains data without power (non-volatile)	Needs power to retain data (volatile)
Faster access speed	Slower access speed
Lower storage density	Higher storage density
More expensive	Less expensive
Used for cache memory, CPU registers, and small buffers	Used for main memory

SRAM	DRAM
Typically used in low-power, battery-operated devices and applications where high speed is critical	Typically used in desktops, laptops, and servers
Requires less power to operate	Requires more power to operate

Cache Organization/Operation



Caches are a type of high-speed memory that are used to improve the performance of computer systems by temporarily storing frequently accessed data or instructions. The basic idea behind caching is to store a copy of data that is likely to be needed again in the near future in a faster, more accessible location.

When a CPU requests data or instructions from memory, the cache first checks if the requested data is already stored in the cache. If it

is, the cache returns the data directly to the CPU without accessing the slower main memory. This is known as a cache hit. If the requested data is not in the cache, then the cache must fetch it from main memory and store it in the cache for future requests. This is known as a cache miss.

Caches are designed to be small enough to fit on the same chip as the CPU, but large enough to hold a significant amount of frequently accessed data. The cache is organized into blocks or lines, each of which contains a fixed amount of data. When a block of data is loaded into the cache, it is assigned a tag that identifies its location in main memory. When the CPU requests data, the cache checks the tags of each block to determine if the requested data is already in the cache. If it is, the cache returns the data directly to the CPU. If it is not, the cache must load the required block from main memory into the cache, and then return the data to the CPU.

Overall, caching is an effective way to reduce the average memory access time and improve system performance, as long as the cache is well-designed and properly managed.

Elements of Cache Design

- **Cache size:** The cache must be large enough to store a significant amount of frequently accessed data, but not so large that it becomes too expensive or takes up too much space on the chip.

- **Mapping function:** The mapping function determines how cache lines are mapped to specific locations in the main memory. There are several different mapping functions, including direct mapping, set-associative mapping, and fully associative mapping.

- **Replacement policy/Algorithms:** When the cache is full and a new block of data needs to be loaded, the replacement policy

determines which block should be evicted from the cache to make room for the new block. Common replacement policies include LRU (Least Recently Used), LFU (Least Frequently Used), and random replacement.

- **Write policy:**

The write policy is a key element in the design of a cache and determines how write operations to the cache are handled. There are two main types of write policies: write-through and write-back.

In a **write-through policy**, the cache and the main memory are always kept in sync, meaning that every write operation to the cache is immediately written to the main memory as well. This ensures that the data in the cache is always up-to-date, but can be slower since every write operation must update both the cache and main memory.

In a **write-back policy**, the cache only writes data back to the main memory when the cache block is evicted from the cache. When a write operation occurs, the data is written only to the cache, which can be faster than write-through policy because the write operation does not need to update main memory immediately. Instead, the cache tracks which data has been modified and writes back only the modified data to the main memory when the cache block is evicted. However, this can lead to inconsistencies if the modified data is not written back to the main memory before eviction, and the main memory could contain outdated data.

- **Number of caches:**

The number of caches in a computer system can vary depending on the architecture and design of the system, but a typical system might have two or three levels of cache.

The first level cache, also known as the L1 cache, is usually the smallest and fastest cache, and is located on the same chip as the

CPU. The L1 cache is designed to provide fast access to frequently accessed data and instructions.

The second level cache, or L2 cache, is larger than the L1 cache and is usually located on a separate chip, but still close to the CPU. The L2 cache is designed to hold more data and provide faster access to data that is not found in the L1 cache.

Some systems may also have a third level cache, or L3 cache, which is even larger than the L2 cache and may be shared between multiple cores or processors in a system. The L3 cache is designed to provide additional storage for frequently accessed data and to reduce the number of requests to main memory.

The number and size of caches in a system can have a significant impact on overall performance, and cache design is an important aspect of computer architecture.

- **Line Size:**

Cache design involves several key elements, and line size is one of them. Line size refers to the number of bytes that are stored in a single cache line. When data is accessed from main memory, it is loaded into the cache in units of the line size.

Cache memory principles

It works on the principle of exploiting the locality of reference in computer programs, which refers to the tendency of programs to access the same data and instructions repeatedly over a short period of time. The key principles of cache memory are as follows:

- **Temporal Locality:** This principle refers to the tendency of programs to access the same data and instructions repeatedly over a short period of time. Cache memory exploits temporal locality by storing recently accessed data and instructions in a

small, fast memory that can be accessed more quickly than the main memory.

- **Spatial Locality:** This principle refers to the tendency of programs to access data and instructions that are close to each other in memory. Cache memory exploits spatial locality by storing contiguous blocks of memory in cache lines, so that when one memory location is accessed, the entire cache line is loaded into the cache.

Cache mapping techniques

They are used to determine how data is stored and retrieved in a cache memory. The goal of cache mapping is to optimize the use of the cache memory and reduce the number of cache misses, which occur when data is not found in the cache and must be retrieved from the main memory.

There are three main types of cache mapping techniques: direct mapping, set-associative mapping, and fully-associative mapping.

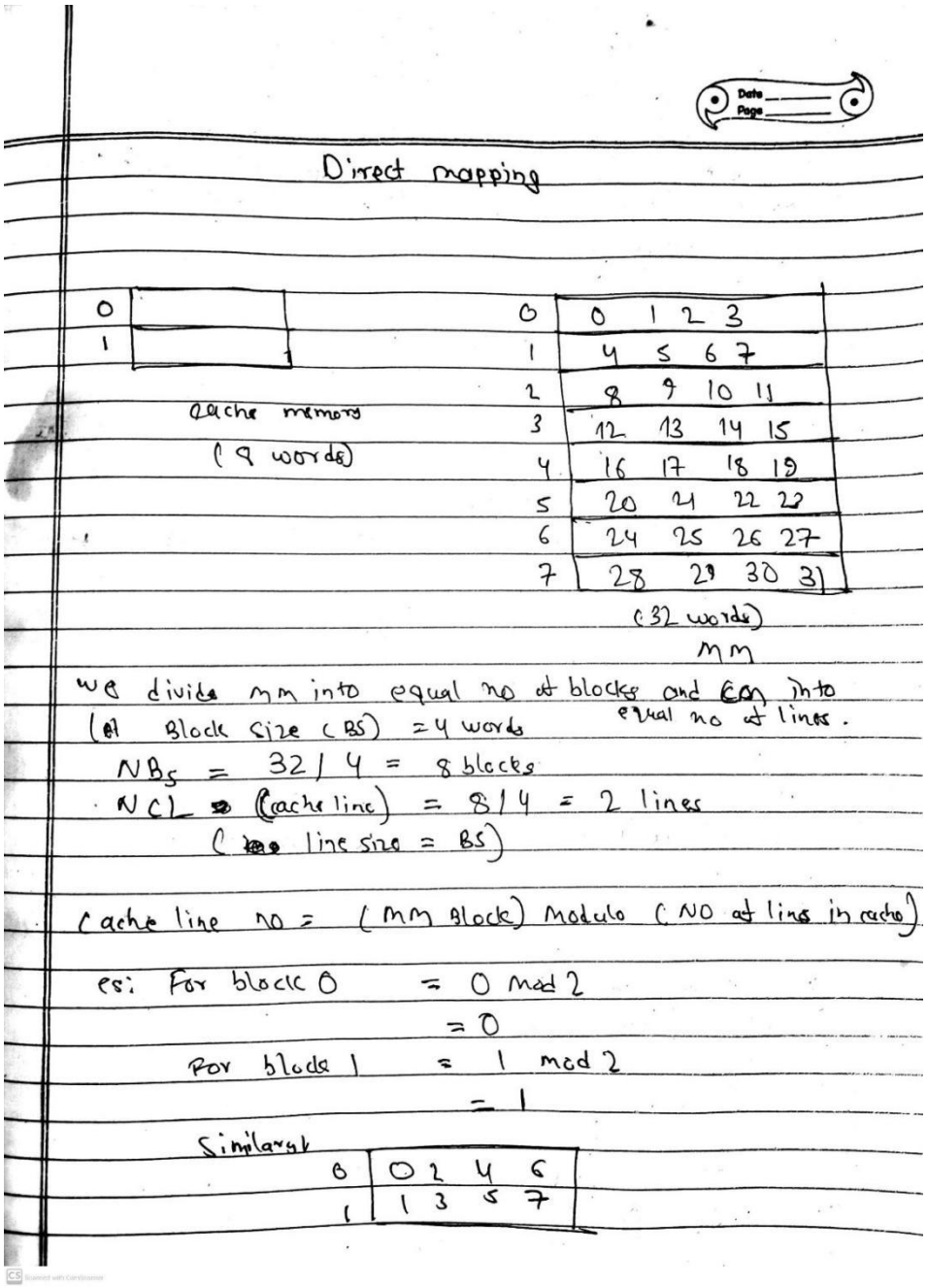
- **Direct mapping:** In this technique, each memory block is mapped to a specific cache line. The mapping is based on the memory address of the block, with each block being stored in a particular line based on the index/line no field . Direct mapping is simple and fast, but it can suffer from cache thrashing when multiple memory blocks map to the same cache line.

- **Set-associative mapping:** In this technique, each memory block can be mapped to a set of cache lines. A set is a group of cache lines that are accessed together. The mapping is based on both the index/line no field and a subset of the tag field. Set-associative mapping can reduce cache thrashing by allowing for more flexibility in block placement.

- **Fully-associative mapping:** In this technique, each memory block can be stored in any cache line. The mapping is based on

the entire tag field. Fully-associative mapping is the most flexible and can provide the highest hit rate, but it is also the most complex and expensive.

Direct mapping (Youtube Link)



Date
Page

line of

we have to pick a block and put into n cache memory. For every block only one line is allotted in cache memory.

0	0 2 4 6
1	1 3 5 7

0 block goes to cache line 0, 1 block goes to cache line 1 and so on.

Now we have physical address divided into three parts.

5 bits

(5-3) →	2bits	1 bit	2bits
	Tag	line no	Block offset

MM = 32 words = 2^5 (physical address = 5 bits)
BS = 4 words = 2^2 (Block offset = 2 bits)

we have two line in cache so 2^1 (1 bits line no)

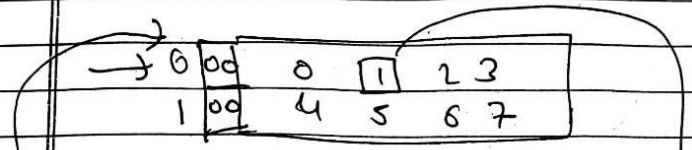
* Tag and line no combine indicate block no (3bits)
lets convert block no to binary

0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

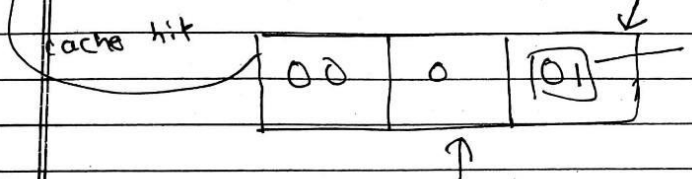
Last bit represents line no.

CPU asks for a particular word if it's in cache then cache hit or else cache miss.

Let us shift block 0 and 1 into cache.

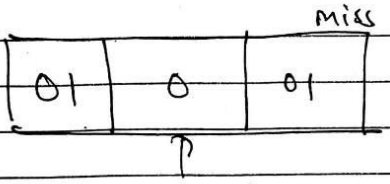


Suppose we have following physical address



At first we see line no and move to line no 0 of cache and we match the tag field with the cache line if it matches then cache hit otherwise miss.

In the above we get [1] word.



Here tag field of physical address and cache line doesn't match so cache miss. If cache miss then we have to approach MM then shift the block into cache the we have to give particular word to CPU. If hit direct we can give the word from cache to CPU.

Q. Suppose main memory has 64 blocks and cache memory has 8 blocks when 10 blocks of main memory are used, show how mapping is performed in direct mapping technique.

In direct mapping technique, each block of main memory is mapped to a specific cache line. The cache is divided into a fixed number of cache lines, and each line has a fixed size.

In this example, the main memory has 64 blocks, and the cache memory has 8 blocks. To map a block from main memory to the cache memory, we use the following formula:

Cache line = Main memory block % Number of cache lines

where '%' denotes the modulo operator.

In this case, the number of cache lines is 8, so we have:

Cache line = Main memory block % 8

Let's assume that the first 10 blocks of main memory are being used, and we want to map them to the cache memory. The mapping is shown in the table below:

Cache line = 10 % 8

Main memory block	Cache line
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	0
9	1

As we can see, each block of main memory is mapped to a specific cache line based on its position in main memory. The first 8 blocks are mapped to cache lines 0 to 7, and the next 2 blocks (blocks 8 and 9) are mapped to cache lines 0 and 1, respectively. If a block is accessed that is already present in the cache memory, it is accessed directly from the cache. If a block is not present in the cache memory, it is fetched from main memory and stored in the cache, replacing the block that was mapped to the same cache line.

(Explain like above diagram banayera ani.)

Merits:

- Direct cache mapping is a simple and efficient cache mapping technique.

- It provides a fast access to the cache lines since each memory block has a fixed location in the cache.

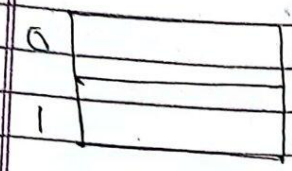
Demerits:

- Direct mapping suffers from the problem of cache thrashing, where multiple memory blocks map to the same cache line, leading to frequent cache misses and poor performance.

- Direct mapping cannot handle multiple accesses to different memory blocks that map to the same cache line.

Associate mapping (Youtube Link)

Associative Mapping



0	0 1 2 3
1	4 5 6 7
2	8 9 10 11
3	12 13 14 15
4	16 17 18 19
5	20 21 22 23
6	24 25 26 27
7	28 29 30 31

(32 words).
(2^5)

Here any block of MM can map to any line in cache memory.

Blocksize = line size = 4 words
$= 2^2$
(2 bit block offset)

Physical address
5 bit

3 bits	2bits
Tag	Block offset

Here line no is not imporatant because every block can map with every line.

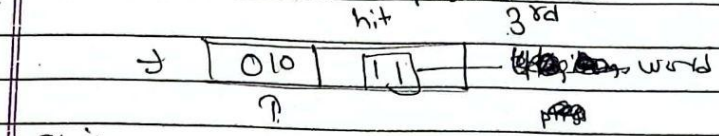
Let initial cache memory status be

	Tag	0	1	2	3
2	0 010	8	9	10	11
4	1 100	16	17	18	19

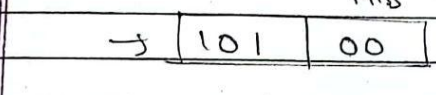
BN

Here tag is used to identify mapping of block number Here BN 2 and 4 are mapped to cache line.

Now let us consider physical addrs



It is cache hit because the tag matches the tag at 0 cache line. So required block is in cache. Here 3rd word 11 is given to the CPU.



Here tag doesn't match with tag of any cache line so it is cache miss. So we have to approach to the MM for the particular word. It the replaces the cache line with it and CPU gets the word it requires.

Advantages:

- High flexibility
- Low conflict misses

Disadvantages:

- Higher hardware complexity and cache access latency
- Higher power consumption

Set- Associate mapping

Q. Describe how set associative mapping combines the feature of direct and associated mapping technique.
(explained below with figure)

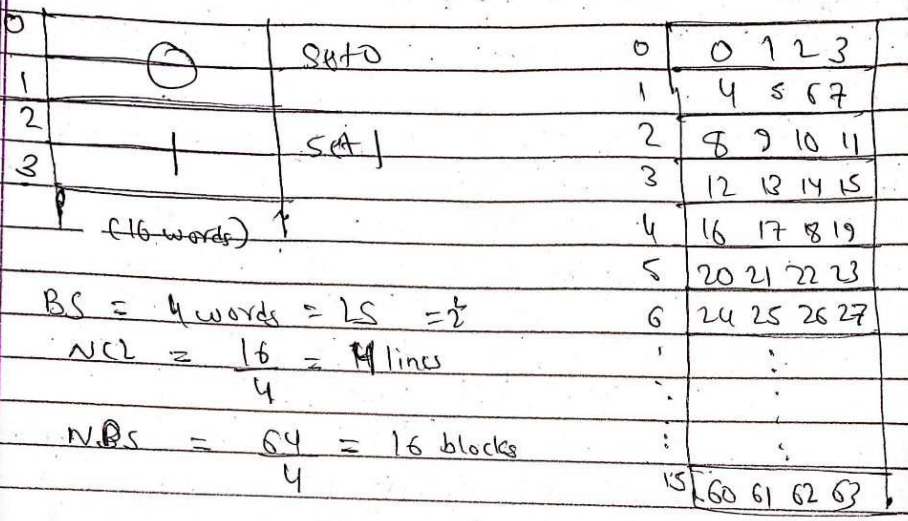
Advantages

- Better balance between hardware complexity and flexibility
- Lower conflict misses than direct mapping

Disadvantages

- Higher conflict misses than associative mapping
- More complex hardware than direct mapping

Set Associative Mapping



BS = 4 words = 2S = 2^2

NCL = $\frac{16}{4}$ = 4 lines

NBS = $\frac{64}{4}$ = 16 blocks

We dive cache line into sets,
We divide into 2-way set associative because a set contains 2 lines so it is called 2-way set associative. If it contained 3 lines it would be 3-way set associative.

* Here both the caractoristic of direct and associative mapping are used.

Here

0 mod 2 = 0 (0 block should only be mapped to Set 0)
1 mod 2 = 1 (1 block show be mapped only to set 1)
2 mod 2 = 0 ↓ Similar
3 mod 2 = 1
4 mod 2 = 0

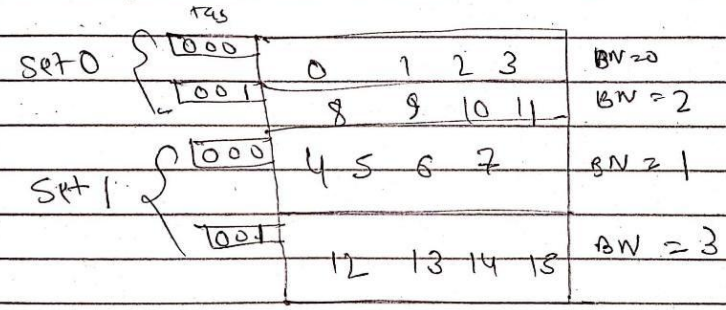
We select set using direct mapping characteristics and after we select set we use associative mapping characteristics.

Physical address — 2^1 sets

8 bits

3 bits	1 bit	2 bits
Tag	Set no	Block offset

Block no



Here 4 bit of BN are

0 - 0000 - set no 2 - 0010

1 - 0001 - set no 3 - 0011

Here block 0 is loaded in 0 line of set 0. Block 1 is loaded in 2 line of set 1 and so on.

hit

001	0	10

word

00 0th
01 1st
10 2nd
11 3rd

go to set 0 and compare tag. it matches so cache hit. 10 is given to CPU.

010	11	00

→ in set 1 tag doesn't match so cache miss.

Mapping Type	Advantages	Disadvantages	Use Case
Direct Mapping	- Simple hardware implementation - Lower cache access latency	- Limited flexibility due to one-to-one mapping - Higher conflict misses	Small cache size or when data has a regular access pattern
Associative Mapping	- High flexibility - Low conflict misses	- Higher hardware complexity and cache access latency - Higher power consumption	Large cache size or when data has an irregular access pattern
Set-Associative Mapping	- Better balance between hardware complexity and flexibility - Lower conflict misses than direct mapping	- Higher conflict misses than associative mapping -More complex hardware than direct mapping	Medium-sized cache

Cache replacement algorithms

They used to determine which cache line to evict when a cache is full and a new line needs to be brought in.

- To reduces miss
- To increase hit

Least Recently Used (LRU): The LRU algorithm evicts the cache line that was least recently used. This algorithm works by keeping track of the order in which cache lines are accessed and selecting the line that was accessed furthest in the past.

First-In, First-Out (FIFO): The FIFO algorithm evicts the cache line that was first brought into the cache. This algorithm works by keeping track of the order in which cache lines are brought into the cache and selecting the line that was brought in first.

Random: The random replacement algorithm evicts a randomly selected cache line. This algorithm is simple to implement and requires minimal hardware overhead, but it does not take into account the access patterns of the data.

Least Frequently Used (LFU): The LFU algorithm evicts the cache line that was accessed the least number of times. This algorithm works by keeping track of the number of times each cache line is accessed and selecting the line with the lowest count.

Most Recently Used (MRU): The MRU algorithm evicts the cache line that was most recently used. This algorithm works by keeping track of the order in which cache lines are accessed and selecting the line that was accessed most recently.

Q. Give reasons why replacement algorithm is not required in direct mapping technique.

In a cache memory system that uses direct mapping, each main memory block can only be mapped to one specific cache memory block. Therefore, there is no need for a replacement algorithm to determine which cache block to evict when a new main memory block needs to be brought into the cache. Here are some reasons why replacement algorithm is not required in direct mapping technique:

One-to-one mapping: In direct mapping, each main memory block can only be mapped to one specific cache memory block. Therefore, when a new block needs to be loaded into the cache, it

will only occupy one specific cache block, and no other block will be affected.

Fixed cache size: In a direct mapping cache, the number of cache blocks is fixed. This means that the cache cannot hold more blocks than its capacity. Therefore, when a new block is loaded into the cache, it must replace an existing block. The block to be replaced is predetermined by the direct mapping scheme.

Simple design: Direct mapping is the simplest cache mapping technique. It requires no complex algorithms to determine which block to evict from the cache. This simplicity makes it easy to implement and efficient in terms of both hardware and software.

Fast access time: Direct mapping provides fast access time because the location of each main memory block is predetermined by its mapping to a specific cache block. This eliminates the need for complex algorithms to determine the location of a block in the cache, resulting in faster access times.

Q. Why replacement algorithm is necessary in associative mapping? Justify.

In an associative mapping cache, each main memory block can be mapped to any cache memory block. Therefore, a replacement algorithm is necessary to determine which cache block to evict when a new main memory block needs to be brought into the cache. Here are some reasons why a replacement algorithm is necessary in associative mapping:

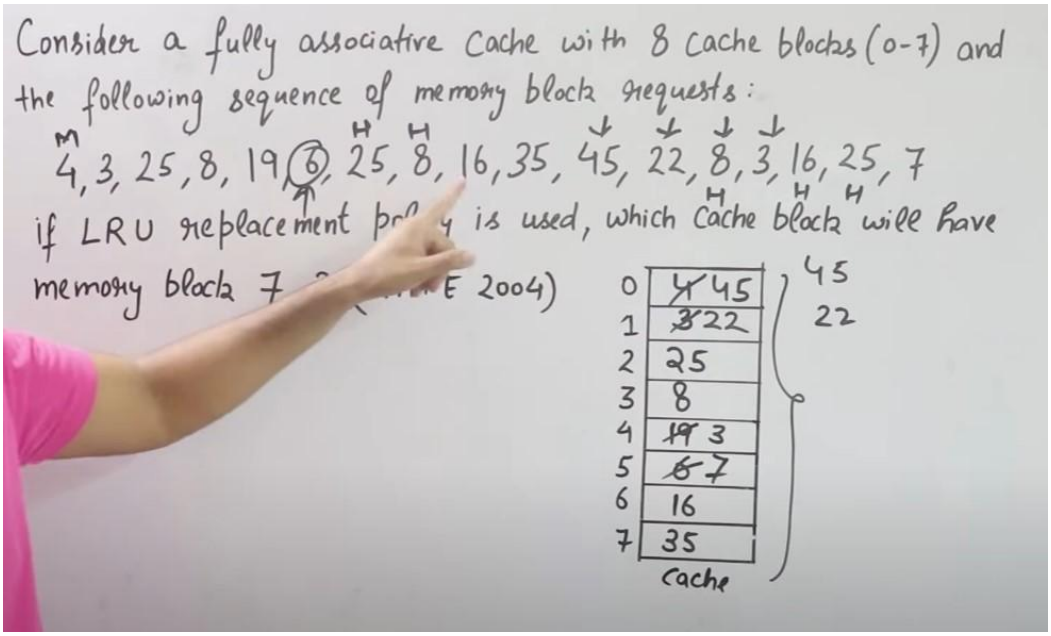
Multiple blocks mapping: In an associative mapping cache, multiple main memory blocks can be mapped to the same cache block. This means that when a new block needs to be loaded into the cache, there may not be an empty cache block available. The replacement algorithm is used to evict an existing block from the cache to make room for the new block.

Variable cache size: In an associative mapping cache, the number of cache blocks is not fixed. The cache can hold as many blocks as there is available space. Therefore, the replacement algorithm is necessary to decide which block to evict when a new block needs to be loaded into the cache.

Complex design: Associative mapping is a more complex cache mapping technique compared to direct mapping. It requires a more complex hardware design and software algorithms to determine the location of a block in the cache. The replacement algorithm is a critical part of this algorithm and determines which block to evict from the cache when there is a cache miss.

Longer access time: Associative mapping provides a longer access time compared to direct mapping because the location of each main memory block is not predetermined. Therefore, a more complex algorithm is required to determine the location of a block in the cache. This algorithm includes the replacement algorithm, which adds to the overall access time.

Q. Explain Least Recently Used (LRU) replacement algorithm in case of hit and miss with suitable example. (YOUTUBE)



Suppose we have a cache memory with four blocks and the following access sequence:

Block 1 is accessed.
Block 2 is accessed.
Block 3 is accessed.
Block 4 is accessed.
Block 1 is accessed again.
Block 2 is accessed again.
Block 5 is accessed (miss).

Here's how LRU would work in this example:

Block 1 is accessed and stored in the first cache block.
Cache: 1, -, -, -

Block 2 is accessed and stored in the second cache block.
Cache: 1, 2, -, -

Block 3 is accessed and stored in the third cache block.
Cache: 1, 2, 3, -

Block 4 is accessed and stored in the fourth cache block.
Cache: 1, 2, 3, 4

Block 1 is accessed again. Since Block 1 is already in the cache, there is a cache hit and the cache remains unchanged.
Cache: 1, 2, 3, 4

Block 2 is accessed again. Since Block 2 is already in the cache, there is a cache hit and the cache remains unchanged.
Cache: 1, 2, 3, 4

Block 5 is accessed and is not in the cache (miss). The LRU algorithm determines that Block 1 has not been accessed for the longest time and replaces it with Block 5.
Cache: 5, 2, 3, 4

The FIFO (First-In-First-Out) cache replacement algorithm is a simple and commonly used approach in computer memory management. In this algorithm, the cache keeps track of the order in which items are stored, and when a new item is added to the cache, it is placed at the end of the list. When the cache reaches its maximum capacity and a new item needs to be added, the oldest item in the cache (i.e., the item that was added first) is removed.

Let's consider an example of a cache that can hold three items, and the following sequence of requests for data:

Read item A - Miss
Read item B - Miss
Read item C - Miss
Read item A again - Hit
Read item D - Miss
Initially, the cache is empty:

Cache
When item A is requested, the cache misses and it is added to the cache:

Cache
A

When item B is requested, the cache misses again and it is added to the end of the list:

Cache
A
B

When item C is requested, the cache misses again and it is added to the end of the list:

Cache
A

B
C

When item A is requested again, it is already in the cache, so it is a hit:

Cache
A
B
C

Finally, when item D is requested, the cache misses and the oldest item in the cache (item A) is removed, and item D is added to the end of the list:

Cache
B
C
D

This is how the FIFO cache replacement algorithm works. It can be a simple and effective approach for managing cache memory in a variety of computer systems.

Chapter 7 (10 marks)

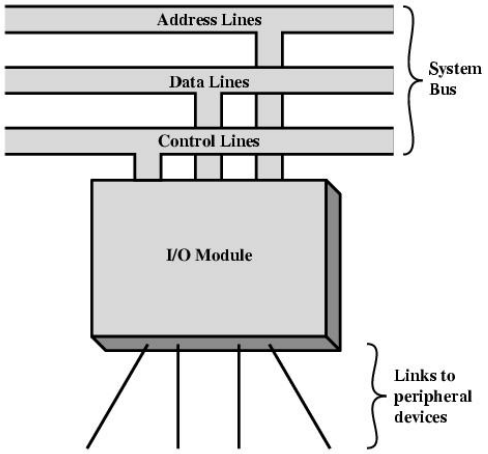
I/O organization

External devices

Peripherals, also known as peripheral devices, are external devices that are connected to a computer system to enhance its functionality and provide additional features. These devices are not the core components of the computer but serve specific purposes and interact with the computer system to input or output data.

I/O Module

An I/O (Input/Output) module, also known as an I/O controller or I/O interface, is a device or component in a computer system that facilitates communication between the CPU (Central Processing Unit) and external devices. It manages the exchange of data and control signals between the computer and various input and output devices, such as keyboards, mice, monitors, printers, storage devices, and network interfaces.



I/O Module Functions

The I/O module is a special hardware component interface between the CPU and peripherals to supervise and synchronize all I/O transformation

The detailed functions of I/O modules are;

Control & Timing:

I/O module includes control and timing to coordinate the flow of traffic between internal resources and external devices. The control of the transfer of data from external devices to processor consists following steps:

- The processor interrogates the I/O module to check status of the attached device.
- The I/O module returns the device status.
- If the device is operational and ready to transmit, the processor requests the transfer of data by means of a command to I/O module.
- The I/O module obtains the unit of data from the external device.
- The data are transferred from the I/O module to the processor.

Processor Communication:

I/O module communicates with the processor which involves:

- **Command decoding:** I/O module accepts commands from the processor.
- **Data:** Data are exchanged between the processor and I/O module over the bus.
- **Status reporting:** Peripherals are too slow and it is important to know the status of I/O module.
- **Address recognition:** I/O module must recognize one unique address for each peripheral it controls.

Device Communication:

In the context of an I/O module, device communication refers to the exchange of commands, status information, and data between the module and the connected external devices.

Commands are instructions sent from the processor to the I/O module, specifying the desired actions to be performed by the device. Status information is provided by the I/O module to the processor, indicating the

current state or condition of the device. Data refers to the actual information being transferred between the external device and the processor.

Data buffering:

Data buffering is essential when the I/O device operates at a different speed than the memory. If the device is faster, there is a risk of losing data. To prevent this, an I/O module buffers the data. This intermediate step allows the memory to catch up with the device's speed and ensures that no data is lost. Buffering means temporarily storing the data in a dedicated area called a buffer or cache. Conversely, if the device is slower, the I/O module buffers the incoming data from the slower device and holds it in the buffer until the memory is ready to receive it.

Error detection:

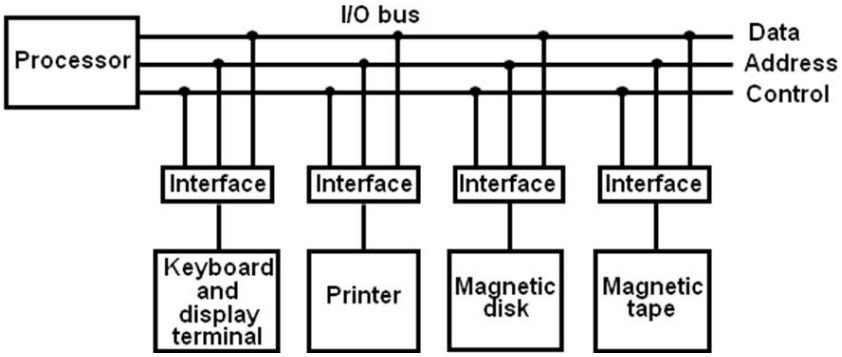
Error detection is another critical function of the I/O module. It is responsible for detecting mechanical and electrical malfunctions reported by the device, such as paper jams or bad ink tracks. The module also identifies unintentional changes to the bit pattern and transmission errors. Techniques like parity bits, checksums, or CRC calculations are used to verify data integrity and detect errors during transfer.

An I/O interface

An I/O (Input/Output) interface, also known as an I/O controller or I/O module, is a hardware component that allows a computer system to communicate with external devices and exchange data. It serves as a bridge between the computer's internal components and the outside world.

I/O Bus and Interface Modules

The I/O bus consists of data lines, address lines and control lines.



- peripheral devices are connected to the CPU through a bus that contains the data bus, address bus, and control bus. However, the processor is not directly connected to these devices and requires an interface.

- The interface acts as a translator and converter between the CPU and peripheral devices.

- CPUs are significantly faster than peripheral devices such as keyboards, printers, and magnetic disks. To synchronize their speeds, an interface is necessary to match the data transfer rates.

- different devices generate signals of varying nature, such as electrical, electro-mechanical, or electromagnetic. The interface assists in converting these signals to ensure compatibility.

- CPUs have specific data word sizes (e.g., 32-bit or 64-bit), whereas peripheral devices like keyboards, printers, and magnetic disks may have different formats and word sizes. The interface helps align the data formats and word sizes between the CPU and the peripherals.

- peripheral devices have different functions and require different logics or programs for data transfer. Instead of creating separate logics for each device, an interface provides a standardized way to handle the data transfer process.

- using interfaces helps avoid unnecessary complexity in handling different devices and allows for easier system upgrades or changes without altering the underlying logic.

I/O versus Memory Bus

Computer buses can be used to communicate with memory and I/O in three ways:

- Use two separate buses, one for memory and other for I/O. In this method, all data, address and control lines would be separate for memory and I/O.

- Use one common bus for both memory and I/O but have separate control lines. There is a separate read and write lines; I/O read and I/O write for I/O and memory read and memory write for memory.

- Use a common bus for memory and I/O with common control line. This I/O configuration is called memory mapped.

Isolated I/O versus Memory Mapped I/O

Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions

Memory-mapped I/O

- A single set of read/write control lines (no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space which reduces memory address range available
- No specific input or output instruction so the same memory reference instructions can be used for I/O transfers
- Considerable flexibility in handling I/O operations

Q. explain three reasons behind the requirement of i/o interfaces.?

Device Connectivity: I/O interfaces enable computers to connect and communicate with a wide range of external devices. These devices can include peripherals such as keyboards, mice, printers, scanners, and storage devices like hard drives and USB flash drives.

Data Transfer and Communication: I/O interfaces play a crucial role in facilitating data transfer and communication between the computer and external systems. For example, network interface cards (NICs) provide the necessary interface for connecting a computer to a local area network (LAN) or the internet.

Standardization and Compatibility: I/O interfaces help establish standards and ensure compatibility between different devices and systems. By providing a standardized interface, I/O interfaces enable devices from different

manufacturers to connect and communicate with each other seamlessly. For instance, USB (Universal Serial Bus) has become a widely adopted standard for connecting various peripherals to computers, regardless of the specific manufacturer or device type. Standardized I/O interfaces reduce complexity, promote interoperability, and simplify the integration of different hardware components and devices within a computer system.

Modes of transfer

Data Transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use CPU as an intermediate path, others transfer the data directly to and from the memory unit.

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

- Programmed I/O
- Interrupt Driven I/O
- Direct Memory Access (DMA)

Programmed I/O

refers to a method of data transfer between a computer's CPU (Central Processing Unit) and an I/O (Input/Output) device. Here are the key points to understand:

Programmed I/O is initiated by specific instructions in the computer program. In this method, the CPU continuously monitors the interface between the CPU and the I/O device.

There are three types of instructions used: input, store, and output.

- **Input instruction:** Transfers data from the I/O device to the CPU.
- **Store instruction:** Transfers data from the CPU to memory.
- **Output instruction:** Transfers data from the CPU to the I/O device.

Programmed I/O is typically used in slow-speed computers and may not be efficient if the CPU and I/O device have different speeds.

Here's how data transfer occurs from an I/O device to the CPU:

The I/O device puts the data on the I/O bus and activates the data valid signal.

The interface receives the data in the data register, sets the status register's F bit (indicating data availability), and activates the data accepted signal.

The I/O device then disables the data valid line.

The CPU continuously monitors the interface by checking the F bit of the status register.

- If the F bit is set (1), the CPU reads the data from the data register and sets the F bit to zero.
- If the F bit is reset (0), the CPU continues monitoring the interface.

The interface disables the data accepted signal, and the system returns to an initial state, waiting for the next item of data to be placed on the data bus.

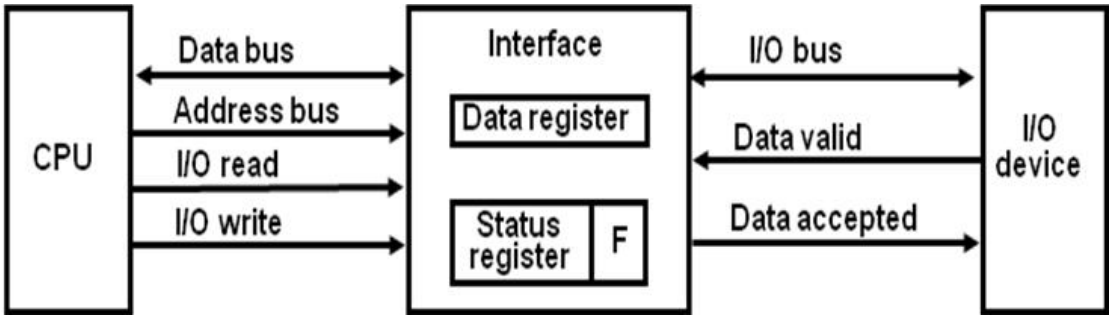
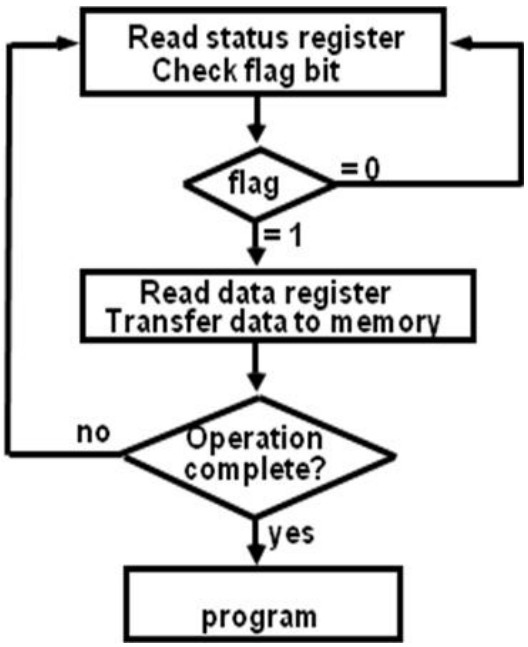


Fig. Data transfer from I/o to cpu



Characteristics:

- Continuous CPU involvement
- CPU slowed down to I/O speed
- Simple
- Least hardware

Applications of programmed I/O method

- Useful in small low speed computers
- Used in systems that is dedicated to monitor a device continuously.
- Used in the data register.
- Used to check the status of the flag bit and branch

Interrupt-driven I/O

is a method that allows more efficient communication between the CPU and I/O devices. Here are the key points to understand:

1. Polling, which involves repeatedly checking the status of an I/O device, consumes valuable CPU time.

2. Interrupt-driven I/O avoids constant polling by establishing communication only when data needs to be transferred.

3. In this method, the I/O interface, instead of the CPU, monitors the I/O device.

4. When the interface determines that the I/O device is ready for data transfer, it generates an Interrupt Request (IRQ) to the CPU.

5. Upon receiving an interrupt, the CPU temporarily pauses its current task, switches to the interrupt service routine (ISR) to process the data transfer, and then returns to the original task.

6. The problem with programmed I/O is that the processor has to wait for the I/O module to be ready, resulting in degraded system performance.

7. With interrupt-driven I/O, the CPU can issue an I/O command to the module and continue with other tasks.

8. The I/O module interrupts the CPU when it is ready to exchange data, requesting service.

9. The interrupt can be initiated either by software or hardware.

10. Here's how interrupt-driven I/O operates:

    a) The CPU sends a read command to the I/O module.

    b) While the CPU continues other work, the I/O module retrieves data from the peripheral device.

    c) The I/O module interrupts the CPU to indicate that data is available.

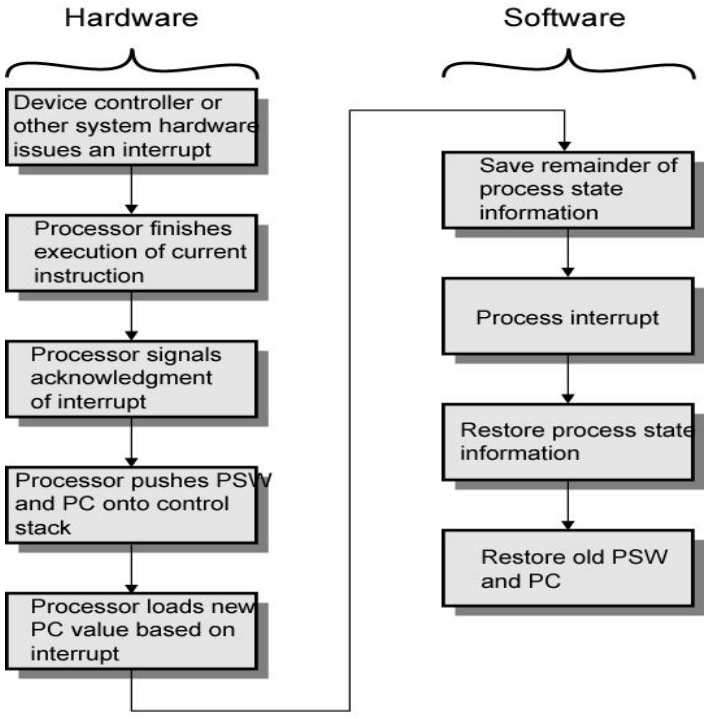
    d) The CPU responds to the interrupt and requests the data.

    e) The I/O module transfers the data to the CPU.

From the CPU's perspective, interrupt processing involves the following steps:

- The CPU issues a read command.
- The CPU continues with other tasks.
- At the end of each instruction cycle, the CPU checks for interrupts.
- If an interrupt occurs:

    - The CPU saves its current state (registers).
    - It processes the interrupt by executing the interrupt service routine.
    - The CPU fetches and stores the transferred data.

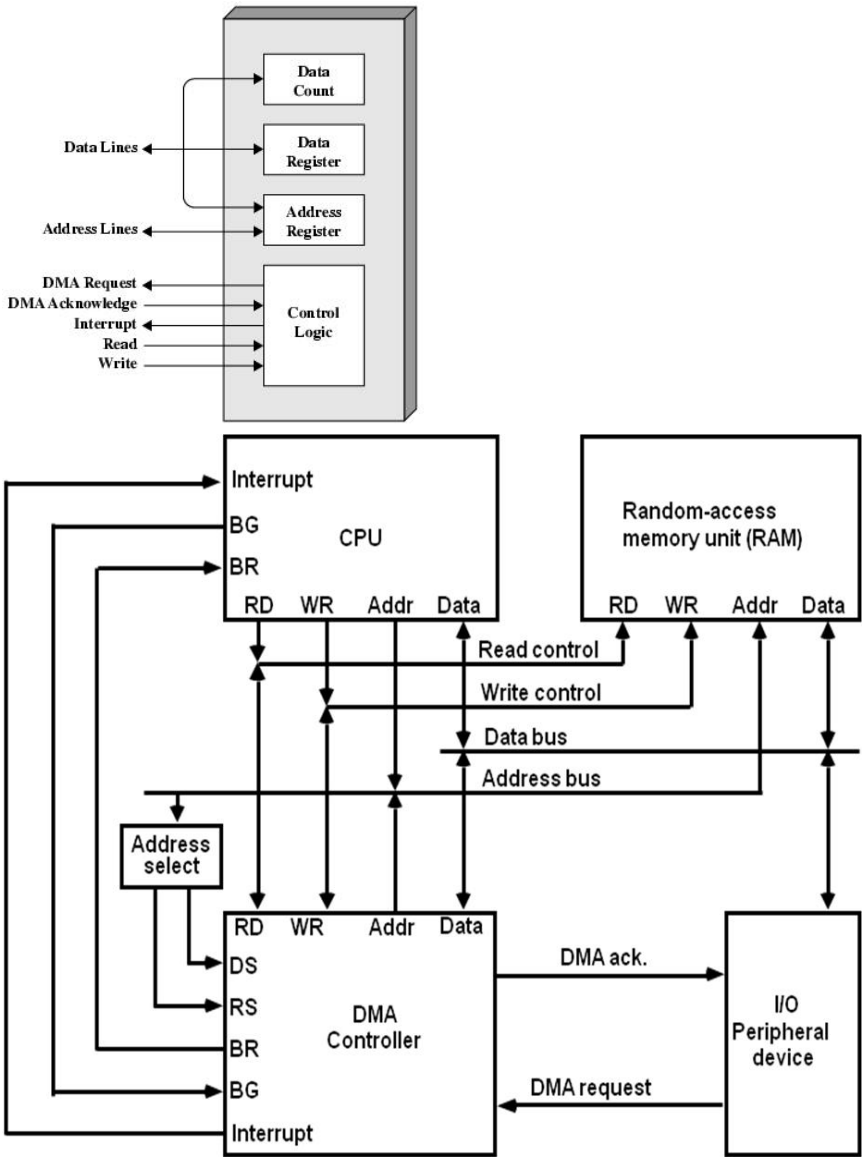


Priority Interrupt

- Priority interrupts are a concept in computer systems where different interrupts are assigned different levels of priority. This allows the system to determine which interrupt should be serviced first when multiple interrupts occur simultaneously.

Direct Memory Access (DMA)

is a method used in computer systems to coordinate data transfers between an I/O device and the core processing unit or memory. It is a faster synchronization mechanism compared to interrupts, offering improved latency and throughput.



DMA is a method used in computer systems to facilitate data transfers between an I/O device and memory without involving the CPU extensively.

DMA operates through a dedicated DMA controller that coordinates the data transfer process.

Here's a simplified overview of the DMA process:

- The CPU initiates the DMA transfer by providing the DMA controller with the necessary information, such as the starting memory address, the device address, and the amount of data to be transferred.

- The CPU can then proceed to perform other tasks while the DMA controller handles the actual data transfer.

- The DMA controller interfaces with the I/O device and directly accesses the system memory via the memory bus.

- The DMA controller takes control of the memory bus and transfers the data between the I/O device and memory without the CPU's active involvement.

- The CPU's involvement during DMA transfer is minimal, and it may be temporarily suspended or have limited access to the memory bus during the transfer.

- Once the DMA transfer is complete, the DMA controller may generate an interrupt to notify the CPU.

Key points to note about DMA:

- DMA is beneficial for large data transfers or frequent I/O operations that would otherwise consume a significant amount of CPU resources and time.

- DMA reduces CPU involvement in data transfer, resulting in improved system performance, reduced latency, and increased throughput.

- DMA operations can be categorized into cycle stealing and burst mode. Cycle stealing involves taking control of the memory bus for a single data transfer, while burst mode allows exclusive access for a block of data transfers.

- DMA transfers can be initiated for both reading data from an I/O device to memory (input) and writing data from memory to an I/O device (output).
- DMA controllers are typically integrated into the computer system's hardware and can be programmed by the CPU to perform specific data transfer tasks.

I/O processor

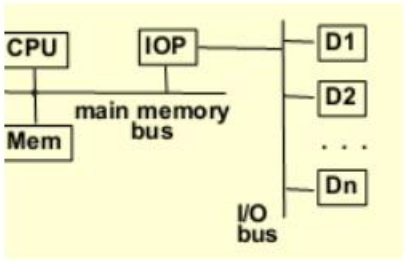
also known as an IOP, is a specialized processor in a computer system that handles input/output (I/O) operations. Here are the key points to understand about I/O processors:

- In some computer systems, the interface logic and direct memory access (DMA) requirements are combined into a single unit called an I/O processor (IOP).
- The IOP is responsible for managing communication between the CPU and I/O devices, handling multiple peripherals through its DMA and interrupt capabilities.
- In a system with an IOP, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP.
- The IOP is a specialized processor that not only performs data transfers between I/O devices and memory but can also execute instructions specific to I/O operations.
- The IOP interfaces with the system and devices, serving as a mediator between the CPU and the I/O subsystem.
- The IOP is capable of executing a set of I/O instructions, which allow it to perform operations such as loading and storing data into memory and controlling I/O devices.

- When an I/O transfer occurs, the sequence of events involves moving or operating the results of an I/O operation into the main memory. This process is typically facilitated by a program for the IOP, which resides in the main memory.

- In the hierarchy of the computer system, the CPU acts as the master processor, while the IOP functions as a slave processor that handles I/O operations on behalf of the CPU.

In summary, an I/O processor (IOP) is a specialized processor with direct memory access capability that manages communication between the CPU and I/O devices. It handles data transfers, executes I/O instructions, interfaces with the system and devices, and operates as a slave processor in the computer system hierarchy.



Where, D1, D2,..., Dn are peripheral devices

Fig.7.10 Block diagram of a computer with I/O processor

CPU-IOP communication

Here's an explanation of the CPU-IOP communication process based on the example provided:

- The CPU sends an instruction to test the IOP path
- The IOP responds by inserting a status word in memory
- The status word indicates the condition of the IOP and I/O device
- The CPU refers to the status word in memory
- If everything is fine, the CPU sends an instruction to start I/O transfer
- The CPU continues with another program while IOP is busy
- IOP sends an interrupt request to the CPU when I/O transfer is complete

- CPU issues an instruction to read the status from the IOP
- IOP places the contents of its status report into a specified memory location
- Status word indicates completion or error

This CPU-IOP communication process allows the CPU and IOP to coordinate and exchange information, enabling efficient input and output operations in a computer system.

Data Communication Processor

Data Communication Processor is a type of I/O (Input-Output) processor that handles the distribution and collection of data from remote terminals connected through communication lines, such as telephone lines.

- Handles distribution and collection of data from remote terminals.
- Connects remote terminals through communication lines.
- Acts as an I/O processor.
- Communicates with terminals through a single pair of wires.

Transmission Types:

- **Synchronous Transmission:** Continuous and synchronized data transfer.
- **Asynchronous Transmission:** Data sent in separate characters with start and stop bits.

Transmission Error Detection:

- **Parity:** Extra bit added to each character for error detection.
- **Checksum:** Sum of data bits sent as extra information for error detection.
- **Cyclic Redundancy Check (CRC):** Mathematical algorithm-based checksum for error detection.
- **Longitudinal Redundancy Check (LRC):** Extra block of bits added to the end of transmission for error detection.

Transmission Modes:

- **Simplex:** One-way data transmission (e.g., TV broadcasting).

- **Half Duplex:** Two-way transmission, but not simultaneously (e.g., walkie-talkie).
- **Full Duplex:** Two-way simultaneous transmission (e.g., telephone).

Data Link and Protocol:

- **Data link:** Communication lines, modems, and equipment used for information transmission.
- **Protocol:** Set of rules and conventions for orderly information transfer.

1. Why priority interrupt is needed for data transmission between COU and I/O device. Explain the types of priority interrupt in detail?

Priority interrupts are needed in this context for several reasons:

Time-sensitive operations: Certain I/O devices may require immediate attention from the CPU due to time-critical operations. By assigning a higher priority to such interrupts, the CPU can quickly service these devices, minimizing the risk of data loss or system failure.

Device dependencies: Some I/O devices may have dependencies on others. For instance, a disk controller may need to communicate with a network interface to retrieve data from a remote server. In such cases, assigning priorities ensures that the required devices receive attention in the correct order, enabling efficient data transfers and preventing bottlenecks.

System efficiency: Prioritizing interrupts helps optimize system performance. By servicing high-priority interrupts first, the CPU can respond promptly to critical events, reducing latency and improving overall system responsiveness. This ensures that time-critical tasks receive the necessary resources in a timely manner.

Types of Priority Interrupts:

Fixed Priority Interrupts: In this scheme, each interrupt source or device is assigned a fixed priority level during system configuration. The priority levels remain constant and determine the order in which interrupts are serviced. Higher priority interrupts preempt lower priority interrupts, allowing critical operations to be handled first.

Software-Programmable Interrupts: These interrupts provide flexibility by allowing the CPU or the software to dynamically assign priorities to interrupt sources. The software can modify the priority levels based on the system's requirements or the nature of the tasks. This adaptability enables efficient utilization of system resources and better responsiveness to changing conditions.

Daisy Chain Interrupts: In a daisy chain configuration, the interrupt signals from multiple devices are linked in a chain. Each device is assigned a priority level, and the chain follows that order. When an interrupt occurs, the CPU services the interrupt from the highest priority device in the chain. Daisy chain interrupts are commonly used in systems with a large number of devices or complex interrupt requirements.
Nested Interrupts: Nested interrupts allow interrupts to be interrupted by higher priority interrupts. If a lower-priority interrupt is being serviced and a higher-priority interrupt occurs, the CPU temporarily suspends the lower-priority interrupt, saves its context, and handles the higher-priority interrupt. Once the higher-priority interrupt is completed, the CPU resumes the execution of the lower-priority interrupt from where it left off. This nesting capability is useful when dealing with a mix of interrupt priorities in complex systems.

2. how is dma techniques is different from programmed input-output?

DMA (Direct Memory Access) and programmed I/O (Input/Output) are two different techniques used for data transfer between I/O devices and memory in a computer system. Let's explore how they differ:

Programmed I/O:

In programmed I/O, the CPU is responsible for managing the entire data transfer process between the I/O device and memory. When an I/O operation is required, the CPU initiates the transfer by executing specific I/O instructions.

The CPU performs the following steps:

- It issues a command to the I/O device, specifying the operation to be performed (e.g., read or write).

- It waits for the device to complete the operation, often by repeatedly polling a status register to check if the device is ready.

- Once the device completes the operation, the CPU transfers the data between the I/O device and memory using load and store instructions.

- Programmed I/O requires continuous CPU involvement throughout the data transfer process, which can result in inefficient CPU utilization and increased overhead. The CPU spends a significant amount of time waiting for I/O operations to complete, limiting its availability for other tasks.

DMA (Direct Memory Access):

DMA is a technique that offloads the data transfer task from the CPU to a dedicated DMA controller, also known as a DMA engine or DMA controller. The DMA controller acts as an intermediary between the I/O device and memory, allowing direct data transfer without CPU intervention. The steps involved in a DMA transfer are as follows:

- The CPU sets up the DMA controller by providing it with the necessary information, such as the source and destination addresses in memory and the transfer length.

- The CPU initiates the transfer by issuing a DMA command to the DMA controller.

- The DMA controller takes control of the system bus and transfers data directly between the I/O device and memory without CPU involvement.

- Once the transfer is complete, the DMA controller releases the bus and signals the CPU.

DMA offers several advantages over programmed I/O:

- **Reduced CPU overhead**: Since the CPU is not involved in the actual data transfer, it is free to perform other tasks concurrently. This improves overall system performance and efficiency.

- **Faster data transfer:** DMA transfers data directly between the I/O device and memory, bypassing the CPU. This eliminates the need for multiple load and store instructions, resulting in faster data transfer rates.

- **Improved data streaming:** DMA is particularly beneficial for devices that require continuous streaming of data, such as sound cards or video capture cards. The DMA controller can continuously fetch data from the

I/O device and store it directly in memory without CPU intervention, enabling smooth and uninterrupted data streaming.

- **Increased system responsiveness:** With DMA, the CPU can offload time-consuming data transfer tasks to the DMA controller, allowing it to focus on critical tasks and respond more quickly to interrupts and other events.

It's important to note that both programmed I/O and DMA have their use cases. Programmed I/O is often suitable for low-speed or sporadic I/O operations, where the CPU's involvement does not significantly impact overall system performance. DMA, on the other hand, is more beneficial for high-speed data transfers or devices that require continuous and efficient data streaming.

3. differentiate between isolated and memory mapped input output.

Isolated and memory-mapped input/output (I/O) are two different approaches used in computer systems to interact with peripheral devices. Here's a brief differentiation between the two:

Isolated I/O:

- In isolated I/O, the CPU communicates with peripheral devices using dedicated input and output instructions.
- Isolated I/O uses a separate address space distinct from the memory address space
- Special instructions are provided for I/O operations, such as IN and OUT instructions in x86 assembly language.
- Typically, a separate I/O bus or a set of dedicated lines are used for isolated I/O.
- The CPU has direct control over the timing and data transfer to and from the peripherals.
- Peripherals are assigned specific I/O addresses, and the CPU accesses them by specifying the appropriate I/O address in the instructions.
- Isolated I/O may involve slower access speeds compared to memory-mapped I/O due to the need for dedicated instructions and bus cycles.

Memory-mapped I/O:

- In memory-mapped I/O, peripheral devices are assigned addresses within the same memory address space as the main memory.
- Memory-mapped I/O uses the same address space as the memory, treating peripheral devices as if they were memory locations.
- Regular load and store instructions are used to interact with peripheral devices, treating them as if they were accessing or storing data in memory.
- Memory-mapped I/O shares the same bus as the memory bus.
- The CPU indirectly controls the peripherals by reading from and writing to their memory-mapped addresses.
- Peripherals are assigned specific memory addresses, and the CPU accesses them through normal load and store instructions, just like accessing memory.
- Memory-mapped I/O can offer faster access speeds compared to isolated I/O since it leverages the existing memory bus infrastructure.

In summary, isolated I/O involves dedicated instructions and separate I/O address spaces for accessing peripheral devices, while memory-mapped I/O treats peripheral devices as if they were memory locations, utilizing the same address space and regular load and store instructions. Memory-mapped I/O can offer faster access speeds but requires careful management to prevent conflicts between memory and I/O addresses.

Q. why data communication processor is required in an I10 organization?

In an I/O organization, a Data Communication Processor (DCP) is required to handle the specific requirements of data communication tasks. Here are some reasons for its necessity:

- **Protocol handling**: DCPs are designed to handle communication protocols required for data transfer between devices. They can perform tasks like packetization, error checking, flow control, and protocol-specific processing.
- **Offloading CPU:** DCPs can offload the CPU from handling low-level data communication tasks, allowing it to focus on other critical operations.

- **Speed and efficiency:** DCPs are optimized for data communication tasks, enabling faster and more efficient handling of data transfers compared to general-purpose CPUs.

- **Scalability:** DCPs are often designed to support multiple I/O channels and high-speed data transfer rates, making them suitable for handling large-scale data communication requirements.

4. how does dma have request over the cpu when both request a memory transfer?

When both the Direct Memory Access (DMA) controller and the CPU request a memory transfer, there is a priority mechanism in place to determine which device gets access to the memory bus. The DMA controller typically has higher priority over the CPU for memory transfers.

By giving priority to the DMA controller, the system ensures efficient and timely data transfers between I/O devices and memory without significantly impacting the CPU's performance. The DMA controller can handle large data transfers in the background, freeing up the CPU to perform other tasks during the transfer.

5. In memory-mapped I/O, the memory address space is reduced?

to allocate a portion of it for mapping I/O devices. This is done due to address space constraints and to prevent conflicts between memory and I/O addresses. By mapping I/O devices to specific memory addresses, it simplifies access and allows the CPU to use the same instructions for accessing both memory and I/O. Memory-mapped I/O offers advantages such as simplified programming, enhanced performance, and efficient utilization of the memory bus. However, careful management is required to ensure proper allocation of memory addresses for both general-purpose memory and I/O devices.

6. compare programmed i/o,interrupt driven i/o and direct memory access(dma)?

Certainly! Here's a comparison of Programmed I/O, Interrupt-driven I/O, and Direct Memory Access (DMA):

Programmed I/O:

- In Programmed I/O, the CPU actively manages the entire I/O process.
- The CPU initiates I/O operations by sending specific instructions to the I/O device.
- The CPU continuously polls or checks the status of the I/O device to determine if the operation is complete.
- It is a simple and straightforward method but can be inefficient as the CPU is fully involved in I/O, resulting in a waste of processing time.

Interrupt-driven I/O:

- In Interrupt-driven I/O, the CPU initiates an I/O operation and then continues with other tasks.
- The CPU is interrupted by the I/O device when the operation is finished or requires attention.
- The CPU acknowledges the interrupt and handles the necessary I/O tasks in response.
- This approach reduces CPU involvement in I/O operations, allowing it to perform other tasks while waiting for I/O completion.
- It is more efficient than Programmed I/O as it minimizes CPU idle time and allows for multitasking.

Direct Memory Access (DMA):

- DMA allows data to be transferred directly between an I/O device and memory without continuous CPU involvement.
- A DMA controller takes over the data transfer process.
- The CPU sets up the DMA controller with necessary transfer parameters and initiates the transfer.
- The DMA controller accesses memory independently of the CPU, transferring data between the I/O device and memory.
- DMA significantly reduces CPU overhead and increases data transfer rates, making it the most efficient I/O method for large data transfers.

Comparison:

- Programmed I/O requires continuous CPU involvement, leading to inefficient use of processing time.
- Interrupt-driven I/O reduces CPU involvement by allowing the CPU to perform other tasks while waiting for I/O completion.
- DMA further reduces CPU overhead by enabling direct data transfer between I/O devices and memory without CPU intervention.

- Programmed I/O is simplest but least efficient, while DMA is the most efficient but more complex to set up.
- Interrupt-driven I/O provides a balance between simplicity and efficiency, suitable for many I/O scenarios.

The choice of which method to use depends on factors such as the nature of the I/O operation, data transfer size, and the need for CPU multitasking.

9. Why input-output processor is needed in an input-output organization? How does a computer know which device issued the interrupt; if multiple devices, how does the selection take place?

An Input-Output Processor (IOP) is a component in a computer system that is responsible for managing input and output operations. Its primary purpose is to facilitate communication between the central processing unit (CPU) and the input/output (I/O) devices connected to the system.

There are several reasons why an IOP is needed in an input-output organization:

- **Efficiency:** I/O operations involve transferring data between the CPU and external devices, such as keyboards, printers, disks, and network interfaces. The IOP offloads these tasks from the CPU, allowing it to focus on computation and processing tasks, which improves overall system efficiency.

- **Device Independence:** The IOP provides a standardized interface for different types of I/O devices. It abstracts the complexities of interacting with various devices, allowing the CPU to communicate with the IOP using a consistent set of commands and protocols. This simplifies the development of software and makes it easier to add or upgrade devices without major modifications to the CPU.

- **Concurrency:** The IOP can handle I/O operations concurrently with the CPU's computation tasks. While the CPU is executing instructions, the IOP can manage data transfers to and from the I/O devices in parallel, minimizing the waiting time for I/O operations and improving overall system performance.

When an I/O device needs attention or requires service, it generates an interrupt to notify the computer system. The interrupt signals that an event has occurred, such as data ready for input or output, or an error condition.

The computer system needs to determine which device issued the interrupt to handle it appropriately.

To identify the device that issued the interrupt, several methods can be used:

- **Polling:** The CPU sequentially checks each device's status or interrupt request line to identify the source of the interrupt. This method involves regularly querying each device, which can be time-consuming and inefficient.

- **Interrupt Request (IRQ) lines:** Each device is assigned a specific interrupt line. When a device requires attention, it asserts its corresponding IRQ line, signaling the CPU to handle the interrupt. The CPU can then determine the interrupting device by examining the state of the IRQ lines.

- **Interrupt priority:** Devices can be assigned different priority levels. When multiple interrupts occur simultaneously, the CPU can use the priority level associated with each device to determine the order in which to handle the interrupts. The device with the highest priority gets serviced first.

The method used to select and handle interrupts depends on the computer system's design and the specific interrupt handling mechanisms implemented in the hardware and operating system.

10. Mention the three possible configurations of DMA and compare them.

Direct Memory Access (DMA) is a technique used in computer systems to transfer data between peripheral devices and memory without involving the CPU. DMA can be configured in three different ways:

- **Single Bus Master DMA:** In this configuration, only one device on the system bus can act as the DMA controller or bus master. The DMA controller takes control of the bus and transfers data between the peripheral device and memory directly, without CPU intervention. While the DMA transfer is in progress, the CPU is idle and cannot access the bus. This configuration is simple and cost-effective but can result in bus contention and reduced CPU efficiency.

- **Multiple Bus Master DMA:** This configuration allows multiple devices to act as DMA controllers or bus masters. Each bus master is assigned a specific priority level, and the highest priority master gains control of the bus when it needs to transfer data. The bus arbitration mechanism is used

to determine the order in which the bus masters access the bus. This configuration provides better flexibility and reduces bus contention compared to single bus master DMA. However, it requires more complex hardware and coordination mechanisms to manage multiple bus masters effectively.

When comparing these DMA configurations, the key factors to consider are:

- **Bus Contention:** Single bus master DMA is more prone to bus contention because only one device can access the bus at a time. Multiple bus master DMA provide better bus utilization and reduce contention by allowing multiple devices to act as bus masters.

- **Flexibility and Scalability:** Multiple bus master DMA offer more flexibility and scalability compared to single bus master DMA. They allow for the integration of additional devices as bus masters without major modifications to the system architecture.

- **Complexity and Cost**: As the number of bus masters increases, the hardware complexity and cost of the DMA system also increase. Single bus master DMA is the simplest and most cost-effective configuration, while multiple bus master DMA configurations require additional hardware and coordination mechanisms, making them more complex and costly to implement.

The choice of DMA configuration depends on the specific requirements of the computer system, including the number and type of peripherals, the need for bus efficiency, and the available resources and budget.

CHAPTER 8 (4 Marks)
MULTIPROCESSORS

Multiprocessors, also known as parallel computers, are computer systems that have multiple processors or CPUs working together to execute programs or tasks simultaneously. These processors can be connected to each other through a shared memory or interconnect network, allowing them to communicate and coordinate their actions.

Multiprocessing refers to the use of multiple processors or cores in a computer system to perform multiple tasks or processes simultaneously. This allows for improved performance, as different processes can be executed on different processors or cores, reducing the overall processing time.

Characteristics of Multiprocessors

Q. Write the characteristics of multiprocessors

- Multiprocessors have multiple CPUs that work together to perform tasks.
- Multiprocessors use shared memory, where all processors can access the same physical memory location.
- Multiprocessors typically have a single operating system that manages all the processors and coordinates their actions.
- Multiprocessors can perform tasks in parallel, which leads to faster processing times.
- Multiprocessors can be symmetric, where all CPUs are equal, or asymmetric, where some CPUs have different roles or capabilities.

- Multiprocessors require a high-bandwidth interconnect to allow for efficient communication between the CPUs.

- Multiprocessors can be scalable, where additional CPUs can be added to the system as needed to increase performance.

- Multiprocessors can provide fault tolerance, where a failing CPU can be replaced by a backup CPU to ensure continued system operation.

- Multiprocessors can support multiple users or applications simultaneously, allowing for better resource utilization and efficiency.

- Multiprocessors are used in a variety of applications, including scientific computing, database management, and real-time systems.

Q. How can multiprocessor be classified according to their memory organization?

Tightly Coupled Multiprocessor	Loosely Coupled Multiprocessor
Interconnected processors sharing memory and resources	Independent processors with their own memory and resources
High-speed and complex interconnection network	Simple and slower interconnection network
Suited for high-performance and low-latency applications	Suited for easily parallelizable applications
Programming model: shared memory	Programming model: message passing
Scalability is limited due to high cost and complexity of interconnection	Scalability is higher due to ease of adding more processors as independent units
A failure of one processor can impact the entire system.	A failure of one processor does not necessarily impact the entire system.

Q.Describe how the multiprocessor systems increase the performance level and reliability.

Multiprocessor systems can increase the performance level and reliability of a computer system in several ways:

- **Parallel Processing:** Multiprocessor systems allow multiple processors to work on a single task simultaneously, using parallel processing techniques. This can greatly increase the speed of processing, as each processor can work on different parts of the task at the same time.

- **Load Balancing:** In a multiprocessor system, the workload can be balanced between processors, ensuring that no one processor is overburdened. This can improve the overall performance of the system and prevent bottlenecks.

- **Scalability:** Multiprocessor systems can be scaled up easily by adding more processors to the system. This can increase the processing power of the system as workload increases.

- **Fault Tolerance:** Multiprocessor systems can be designed with redundancy and fault-tolerant features to improve system reliability. For example, if one processor fails, another processor can take over the workload to prevent system failure.

- **Improved Resource Utilization:** Multiprocessor systems can utilize resources such as memory and input/output devices more efficiently. Multiple processors can access the same memory simultaneously, reducing the need for data transfer between processors.

- **Improved Throughput:** Multiprocessor systems can increase the throughput of a system by allowing multiple tasks to be executed simultaneously. This can be particularly beneficial in real-time data processing environments.

Interconnection structures (Youtube)

Interconnection structures of multiprocessors refer to the way the processors are connected to each other and to the memory and I/O devices in the system.

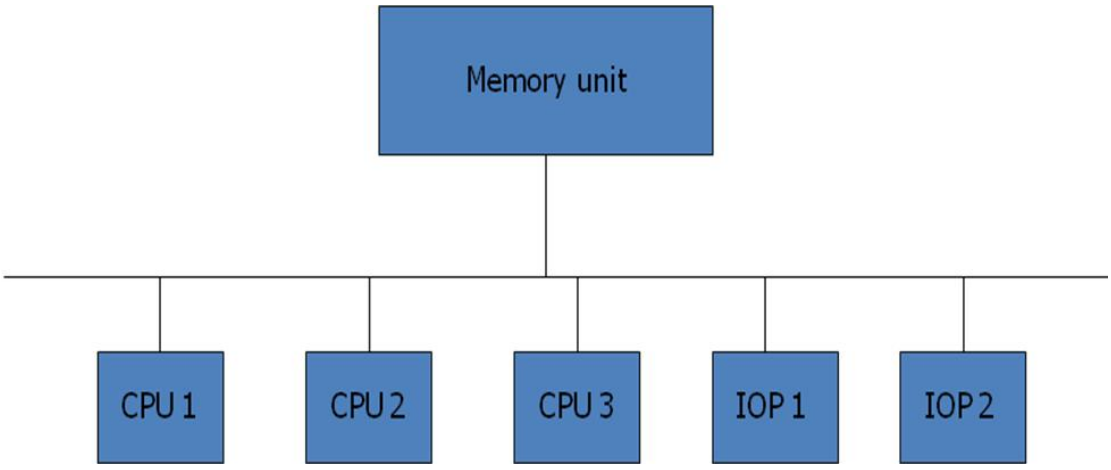
There are several interconnection structures used in multiprocessor systems, including:

1. Timed Shared Common Bus

- A time-shared common bus is a type of computer bus that is shared by multiple devices or components in a computer system.
- Each device or component that is connected to the bus is given a specific time slot during which it can use the bus to transfer data.
- The time slots are usually allocated by a bus controller or arbiter, which determines the order in which devices can access the bus.
- When a device wants to use the bus, it requests permission from the bus controller or arbiter.
- If the bus is currently being used by another device, the requesting device must wait until its allocated time slot before it can access the bus.
- Once a device has been granted access to the bus, it can transfer data to or from other devices that are also connected to the bus.
- After the device has finished using the bus, it releases control of the bus so that other devices can access it during their allocated time slots.

- The time-shared common bus is a cost-effective solution for connecting multiple devices or components in a computer system, as it eliminates the need for dedicated communication channels between each device.

- However, the time-shared nature of the bus means that the overall performance of the system may be limited by the speed at which the bus can transfer data between devices.



2. Multiport Memory

- Multiport memory is a type of memory that has multiple ports, allowing multiple processors to access the memory simultaneously.

- Each port of the multiport memory has its own independent set of address and data lines, allowing it to operate independently of the other ports.

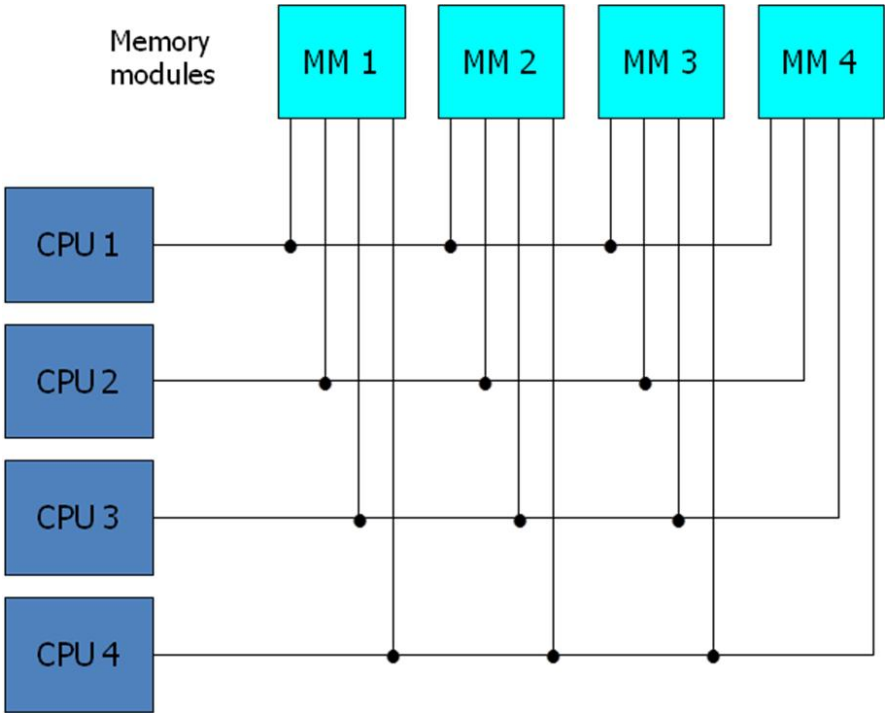
- When a processor wants to access the memory, it sends a memory request to the appropriate port of the multiport memory.

- The multiport memory then performs the memory access and returns the requested data to the requesting processor.

- Since each port operates independently, multiple processors can access the memory simultaneously without interfering with each other.

- Multiport memory can help to improve the performance of a multiprocessor system by reducing contention for the memory bus and allowing multiple processors to access the memory simultaneously.

- However, multiport memory is typically more complex and expensive than traditional single-port memory, and its use may be limited by factors such as cost, power consumption, and availability.



3. Crossbar Switch

- The crossbar switch is composed of a set of input ports, a set of output ports, and a matrix of switches that connect the inputs to the outputs.

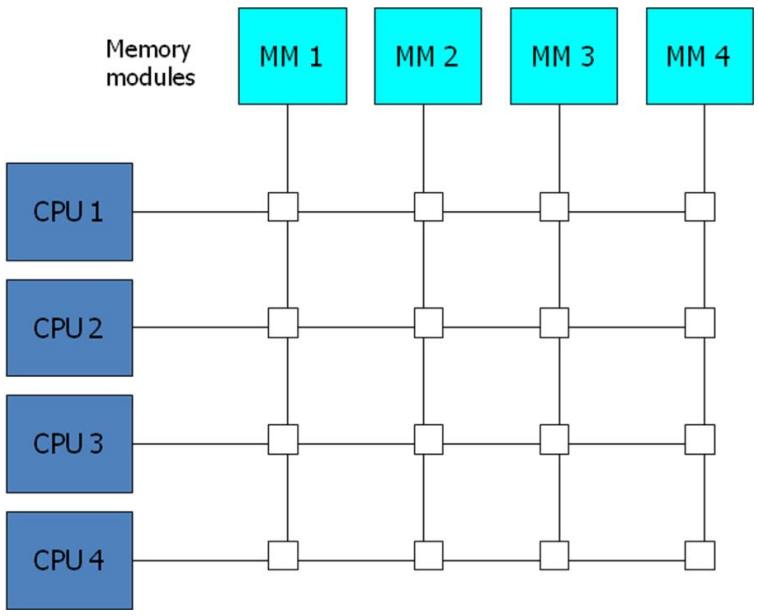
- Each processor is connected to an input port, and each memory or I/O device is connected to an output port.

- When a processor wants to read from or write to memory or I/O, it sends a request to the crossbar switch.

- The crossbar switch then routes the request to the appropriate output port based on the memory address or I/O device address.

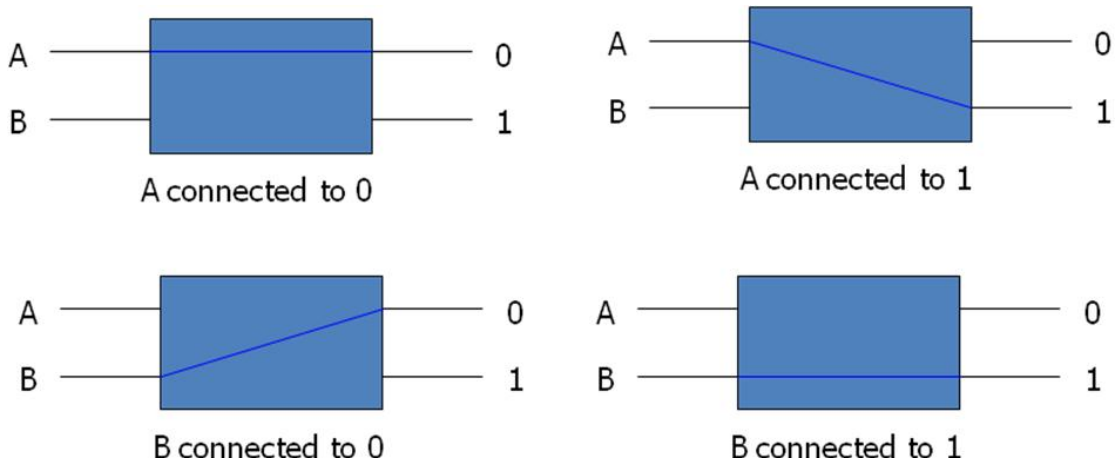
- If multiple processors request access to the same memory or I/O device at the same time, the crossbar switch can prioritize the requests based on the request type or the priority of the processors.

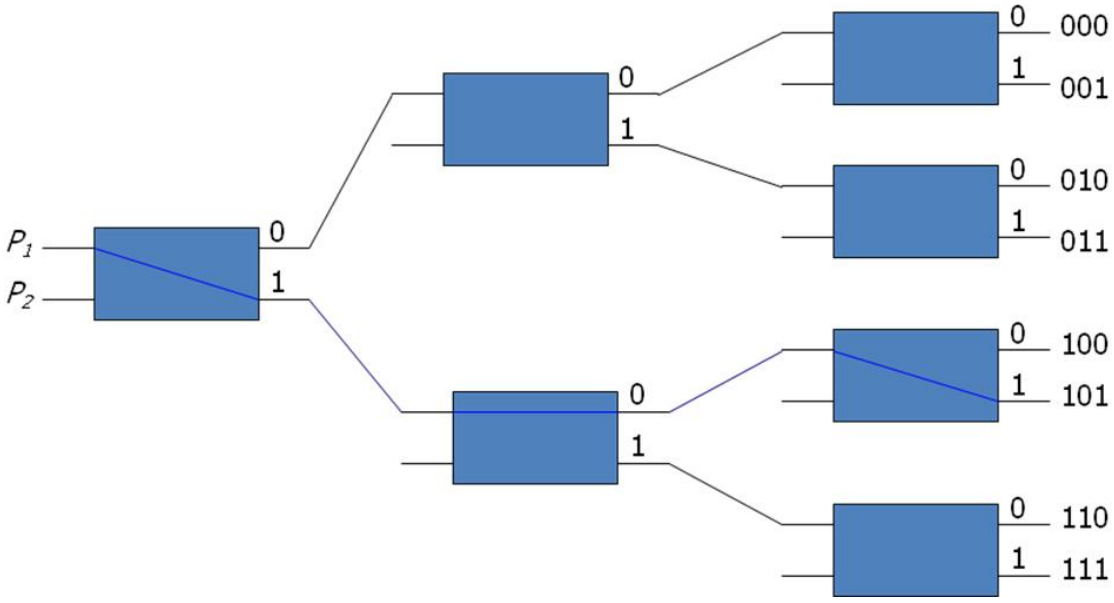
- The crossbar switch provides a high-speed, low-latency, and scalable interconnection network that can handle multiple requests concurrently, making it ideal for multiprocessor systems.



4. Multistage Switching Network

- The multistage switching network is composed of multiple stages, each consisting of a set of switches and connecting links.
- Each processor is connected to an input stage, and each memory or I/O device is connected to an output stage.
- The request from the processor is divided into packets and sent through the switching network, with each packet being directed through a specific path determined by the switches in each stage.
- Each packet is buffered at each stage before being forwarded to the next stage, to prevent congestion and ensure reliable delivery.
- The multistage switching network can handle multiple requests concurrently, making it ideal for high-performance computing applications.
- The multistage switching network provides a fault-tolerant and scalable interconnection network that can be customized to meet the specific needs of the multiprocessor system.

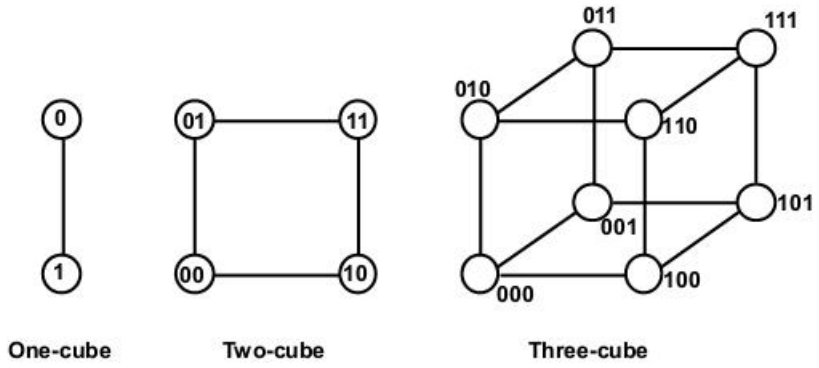




5. Hypercube System

- A hypercube topology consists of nodes that are connected to each other in a way that forms a geometric hypercube shape.
- Each processor is connected to a node in the hypercube, and each memory or I/O device is also connected to a node.
- Requests from processors to memory or I/O devices are sent through the hypercube to the appropriate destination node.
- The hypercube system uses routing algorithms to determine the optimal path for each request to follow through the hypercube.
- The routing algorithm is based on the destination address of the request and the current location of the processor.
- The hypercube system provides a scalable and fault-tolerant interconnection network that can handle multiple requests concurrently.

- The hypercube system can be extended to higher dimensions to increase the number of nodes and processors that can be connected, although this increases the complexity and cost of the system.



Q. Explain inter-processor synchronization with example

Inter-processor synchronization is the process of coordinating the execution of multiple processors in a multiprocessor system to ensure that they operate correctly and efficiently. Here's an example to explain inter-processor synchronization:

Suppose we have two processors, A and B, that are executing a program that uses a shared variable X. The program increments the value of X and prints its value to the console.

If both processors increment X at the same time, the output of the program will be unpredictable, as the value of X will depend on the order in which the increments are executed. To avoid this, we need to synchronize the processors using a locking mechanism.

One common locking mechanism is a mutex (short for mutual exclusion), which allows only one processor to access the shared variable at a time. Here's how inter-processor synchronization using a mutex would work:

- Processor A requests a lock on the mutex before accessing the shared variable X.

- If the mutex is currently unlocked, Processor A acquires the lock and executes its increment operation on X.

- Processor B attempts to acquire the lock on the mutex, but is blocked because the lock is already held by Processor A.

- Processor A releases the lock on the mutex after it has finished accessing X.

- Processor B acquires the lock on the mutex and executes its increment operation on X.

- Processor B releases the lock on the mutex after it has finished accessing X.

By using a mutex to coordinate access to the shared variable, we ensure that the increments of X are executed in a mutually exclusive manner, and that the output of the program is predictable and consistent.