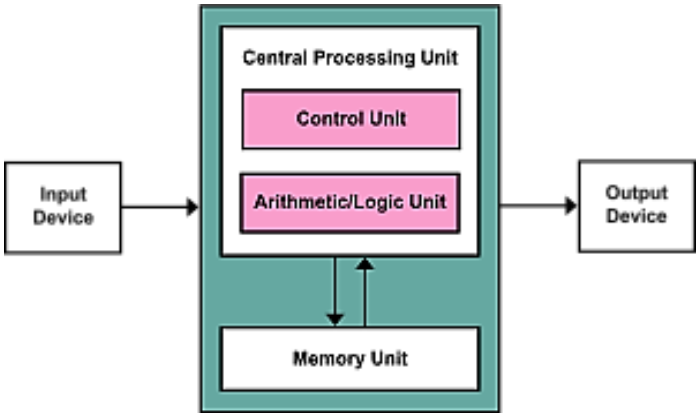
COMPUTER ARCHITECTURE

Introduction

Q-01:What do you mean by Computer Architecture & Organization? Show the structure of Von Neumann Machine.[mec] OR, Describe Von-Neumann Architecture with appropriate diagram. Or, Define Computer Architecture? Draw a block diagram of the basic components of a computer system. [Pre_3:1_2021]

💻 Computer architecture refers to the organization and design of computer systems, including hardware and software components, that enables the execution of instructions and the processing of data.

💻 Computer Architecture & Organization refers to the design and functional organization of a computer system. It encompasses various aspects like the structure of the computer, how different components interact, and the overall organization of data and instructions.



Central Processing Unit (CPU): The brain of the computer, responsible for fetching, decoding, and executing instructions.

Input/Output (I/O) Devices: Allow the computer to communicate with the outside world, like keyboard, mouse, monitor, etc.

Main Memory: Stores program instructions and data that the CPU needs to access.

Arithmetic Logic Unit (ALU): Performs mathematical and logical operations on data.

Control Unit: Decodes instructions and manages the flow of data between the components.

The instruction cycle in a Von Neumann Machine is as follows:

Fetch: The CPU fetches an instruction from memory.

Decode: The CPU decodes the instruction to determine what operation needs to be performed.

Execute: The ALU or other components execute the instruction based on the decoded information.

Store: The results of the operation are stored in memory or a register.

Repeat: The cycle starts again with fetching the next instruction.

This fetch-decode-execute cycle repeats continuously until the program is finished or halted.

Q-02:Apply Fixed Point Arithmetic Restoring Division Algorithm for unsigned integers where Dividend = 14 and Divisor = 4. [NIT_C_01]

Dividend (A) = 14, Divisor (M) = 4.

Step	Action	Quotient (Q)	Remainder (A)
0	Initial State	0000	1110
1	Shift A, Q left	0000	1100
2	A = A - M	0001	0000
3	Shift A, Q left	00010	0000
4	Shift A, Q left	000100	0000
5	Shift A, Q left	0001000	0000

Result: Quotient (Q) = 0001000, Remainder (A) = 0000.

Q-03:Explain the Fixed Point Arithmetic Algorithm (Booth's Multiplication Algorithm) with proper example. [Assume Multiplicand = 6 and Multiplier = 4.] [NIT_C_01]

Given: Multiplicand (M) = 6, Multiplier (Q) = 4.

Step	Action	Product (P)	Multiplier (Q)
0	Initial State	0000 0000	0000 0100
1	Q[0] = 0	0000 0000	0000 0010
2	Q[0] = 1	0000 0110	0000 0001
3	Right Shift	0000 1100	1000 0000
4	Right Shift	0000 0110	1100 0000

Result: Product (P) = 0000 0110 (18).

Q-04:What is Sign Magnitude? Represent 9.725 in 32-bit format using IEEE 754 standard. [NIT_C_01]

Sign Magnitude is a way of representing signed numbers where the most significant bit represents the sign of the number (0 for positive, 1 for negative). The remaining bits represent the magnitude or absolute value of the number.

To represent 9.725 in 32-bit format using IEEE 754 standard:

1. Convert 9.725 into binary: 1001.111101
2. Determine the sign bit based on the sign of the number (positive in this case): Sign bit (S): 0
3. Normalize the binary fractional part: 1.00111101 x 2^3
4. Convert the normalized binary into scientific notation: 1.00111101 x 2^3
5. Determine the biased exponent (8-bit): Biased Exponent (E): 3 + bias (127) = 130 (in binary: 10000010)
6. Combine the sign, exponent, and mantissa bits: 0 10000010 00111101000000000000000

Therefore, 9.725 in IEEE 754 32-bit format is represented as: 0 10000010 00111101000000000000000 VS

Sign Magnitude: The sign magnitude representation uses the leftmost bit as the sign bit (0 for positive, 1 for negative) and the remaining bits to represent the magnitude. For 9.725, in 32-bit format:

- Sign bit: 0 (positive)
- Magnitude: 01000001000111100111110100011110

IEEE 754 Representation:

- Sign bit: 0 (positive)
- Exponent: 10000010 (biased exponent)
- Fraction: 00111100111110100011110

Combined: 01000001000111100111110100011110

Q-05:Perform floating point addition of 10.01 * 104 and 500 in 32-bit format using IEEE 754 standard. [NIT_C_01]

Representation of 10.01 * 10^4:

- Sign bit: 0 (positive)
- Exponent: 10000010 (biased exponent)
- Fraction: 00101000010011110011010

Representation of 500:

- Sign bit: 0 (positive)
- Exponent: 10000101 (biased exponent)
- Fraction: 11111010000000000000000

Performing addition:

1. Align exponents by shifting the fraction of the smaller exponent to the right.
2. Add the fractions.

Result:

- Sign bit: 0 (positive)
- Exponent: 10000101 (biased exponent)
- Fraction: 00101000010011110011010 (sum of the aligned fractions)

Final result: 1.00100001001111001101 0 * 10^5

Q-06:Draw the DFA diagram following the conditions: [NIT_C_01]

i) Accepting strings ending with '0110' over input alphabets ∑ = {0, 1}

ii) Accepting odd binary numbers strings over input alphabets ∑ = {0, 1}

Q-07:Differentiate between DFA and NFA. [NIT_C_01]

Answer:

DFA	NFA
DFA stands for Deterministic Finite Automata.	NFA stands for Nondeterministic Finite Automata.
DFA cannot use Empty String transition.	NFA can use Empty String transition.
DFA can be understood as one machine.	NFA can be understood as multiple little machines computing at the same time.
DFA is more difficult to construct.	NFA is easier to construct.
All DFA are NFA.	Not all NFA are DFA.
DFA requires more space.	NFA requires less space then DFA.
Backtracking is allowed in DFA.	Backtracking is not always possible in NFA.
Conversion of Regular expression to DFA is difficult.	Conversion of Regular expression to NFA is simpler compared to DFA.

Q-08:Determine the different addressing modes for the following expression: (A*B)+(C*D) [NIT_C_01]

Answer:

Q-09: What is pipeline processing? Describe about how the instructions are executed in a pipeline processor. [NIT_C_02]

Answer:

Pipeline processing is a method of executing instructions in a computer processor where the execution of instructions is overlapped to achieve better overall performance. In a pipeline processor, the execution of instructions is divided into a sequence of distinct stages, and multiple instructions can be processed simultaneously in different stages of the pipeline.

The typical stages in a pipeline processor include instruction fetch, instruction decode, execute, memory access, and write back. Each stage performs a specific operation on the instruction or data that it receives.

Here's a general overview of how instructions are executed in a pipeline processor:

1. Instruction Fetch (IF): The processor fetches the next instruction from memory.

2. Instruction Decode (ID): The fetched instruction is decoded to determine its type and operands.

3. Execute (EX): The decoded instruction is executed or performs the necessary computations.

4. Memory Access (MEM): If the instruction requires accessing memory (e.g., load or store), data is read from or written to memory.

5. Write Back (WB): The results of the executed instruction are written back to registers or memory.

Each stage in the pipeline works concurrently with the other stages. As soon as an instruction moves to the next stage, the previous stage can start fetching, decoding, executing, or accessing memory for the subsequent instruction. This overlapping of stages allows for multiple instructions to be processed simultaneously, resulting in increased throughput and better performance.

However, pipeline processing can also introduce some challenges like pipeline hazards, which can occur when one instruction depends on the result of a previous instruction that has not yet been completed. These hazards can impact the proper execution order of instructions and may require additional mechanisms like forwarding or stalling to resolve the dependencies.

Overall, pipeline processing is a technique used to improve the efficiency and performance of processors by breaking down instruction execution into multiple stages and processing multiple instructions at the same time. 🚀

Q-10: Describe about different data dependencies occurred at pipeline processor.[NIT_C_02]

Answer:

In a pipeline processor, data dependencies refer to situations where the execution of instructions depends on the availability of certain data from preceding instructions. Some of the common data dependencies in a pipeline processor are:

1. Read-after-Write (RAW) Dependency:

- Also known as true dependency or data dependency.
- Occurs when an instruction depends on the result of a previous instruction.
- RAW dependencies can lead to pipeline stalls or bubbles as the pipeline needs to wait for the previous instruction to complete before proceeding.

2. Write-after-Read (WAR) Dependency:

- Also known as anti-dependency.
- Occurs when a subsequent instruction tries to read a value from a register before a previous instruction finishes writing to it.
- The subsequent instruction may receive incorrect data if it proceeds before the write operation is completed.
- To resolve WAR dependencies, the pipeline may need to introduce data forwarding or use renaming techniques.

3. Write-after-Write (WAW) Dependency:

- Also known as output dependency.
- Occurs when two or more instructions try to write to the same register or memory location.
- The order of write operations needs to be enforced to ensure consistent results.
- The pipeline needs to ensure that the write operations are serialized to avoid any data corruption.

4. Control Dependency:

- Also known as control flow dependency.
- Occurs when the execution and outcome of an instruction depend on a previous branch instruction.
- The pipeline may need to speculate or predict the outcome of the branch before it is resolved to ensure the correct execution path is followed

These dependencies can create hazards in the pipeline, leading to pipeline stalls or incorrect results if not properly handled.

Q-11: Write short note on: [NIT_C_02]

a) Pipeline processor Throughput and

b) Latency

Answer:

a) Pipeline Processor Throughput: Pipeline processor throughput refers to the rate at which instructions are completed per unit of time in a pipeline architecture. It indicates the efficiency of the processor in processing instructions quickly. Higher throughput is desirable as it means more instructions can be executed within a given time period, leading to improved performance and faster computation.

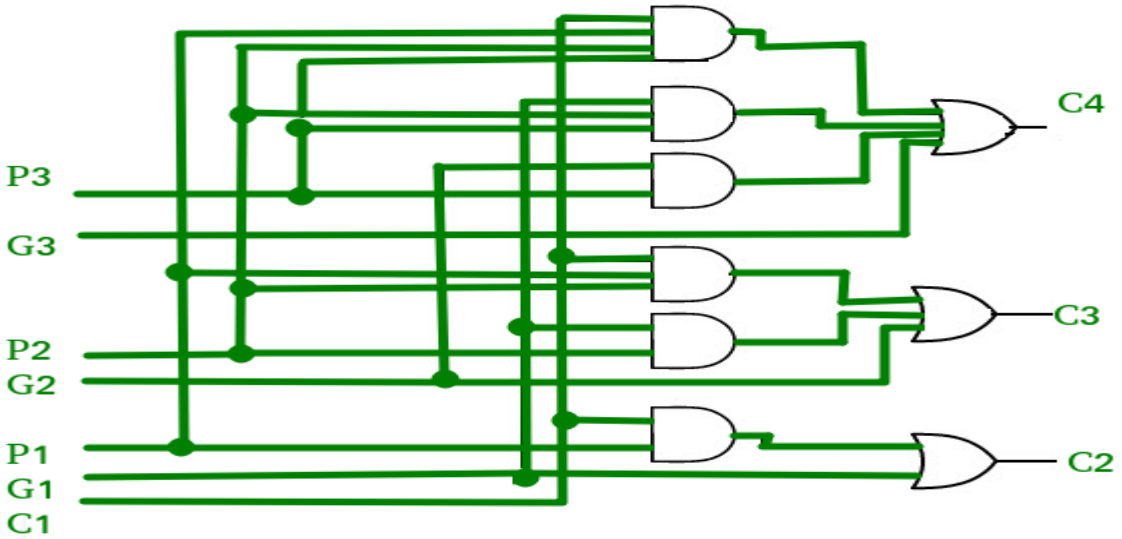
b) Latency: Latency in the context of pipeline processors refers to the time it takes for an instruction to complete its execution. It measures the delay experienced by an instruction from entering the pipeline to producing the final output. Lower latency is desirable as it implies faster execution of instructions and reduced waiting time. However, reducing latency may come with increased pipeline complexity and potential trade-offs with throughput.

Both pipeline processor throughput and latency are important metrics for evaluating the performance and efficiency of pipeline architectures. Higher throughput and lower latency are generally desired to achieve faster and more efficient execution of instructions. 😊 □

Q-12:What is carry propagation delay? Design the carry look ahead generator circuit. *[MEC_C_01]*

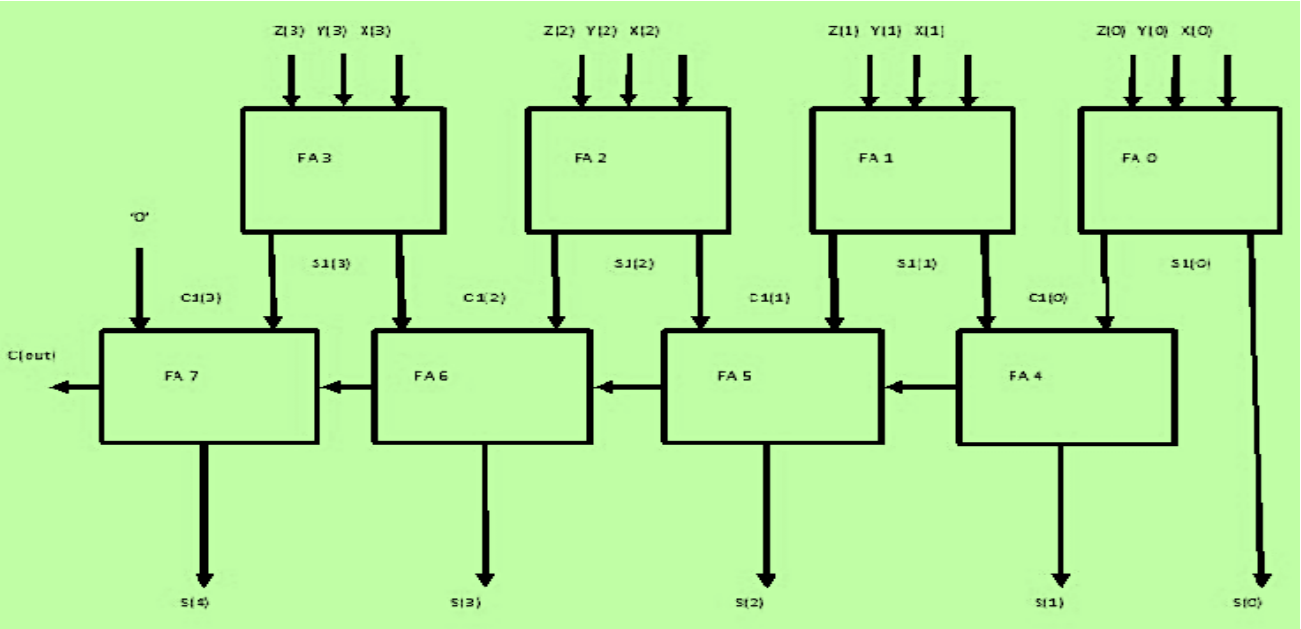
Carry propagation delay refers to the time taken for the carry signal to propagate through a series of adders in a digital circuit. It determines the maximum operating frequency of the circuit and affects the overall performance of the adder.

Here's a high-level representation of the Carry Look-ahead generator circuit:



Q-13:What is the limitation of carry save adder? Implement carry delayed adder. [MEC_C_01]

Answer:



Carry-save adders (CSAs) offer speed advantages in multi-operand addition but exhibit limitations in standalone arithmetic and sign determination. Carry-delayed adders (CDAs) provide direct sum and overflow detection but suffer from increased delay.

Here are the specific limitations of CSAs:

1. **Indirect Sum and Overflow Detection:** CSAs only output "carry-save pairs" (C, S) representing sum and carry bits without the final sum value. Post-processing using ripple-carry or carry-lookahead adders is required, introducing complexity.
2. **Sign Determination Challenge:** Sign determination based on carry-save pairs can be difficult, especially with mixed-sign operands or two's complement representation.
3. **Limited Applicability in Simple Arithmetic:** CSAs excel in multi-operand scenarios (e.g., multiplication) but their complexity outstrips ripple-carry or carry-lookahead adders for basic two-operand addition.
4. **Potential Power Consumption Increase:** While the reduced carry propagation in CSAs generally lowers power, the final sum generation step can negate this benefit, especially in low-power scenarios.

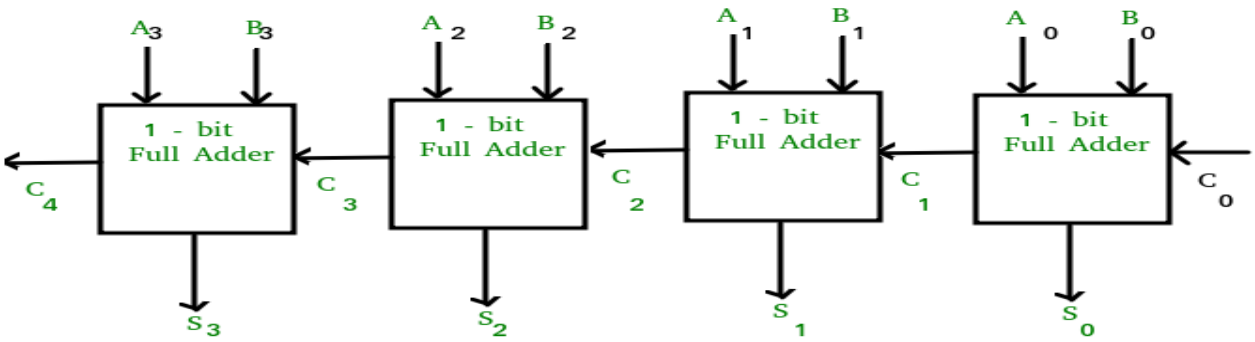
A CDA resembles a ripple-carry adder with delayed carry-out bits. This enables independent sum and carry calculations without waiting for the previous stage's carry-in.

Here's how a CDA works:

Structure: The first full adder receives operands and a zero carry-in. Subsequent adders receive the delayed carry-out from the previous stage. The final sum is obtained after carry propagation through all stages.

Operation: The first full adder receives operands and a zero carry-in. Subsequent adders receive the delayed carry-out from the previous stage. The final sum is obtained after carry propagation through all stages.

Limitations: CDAs offer direct sum and overflow detection but inherit the ripple-carry adder's delay limitations, especially for longer operands. More complex carry anticipation or lookahead techniques are often preferred for speed-critical applications.

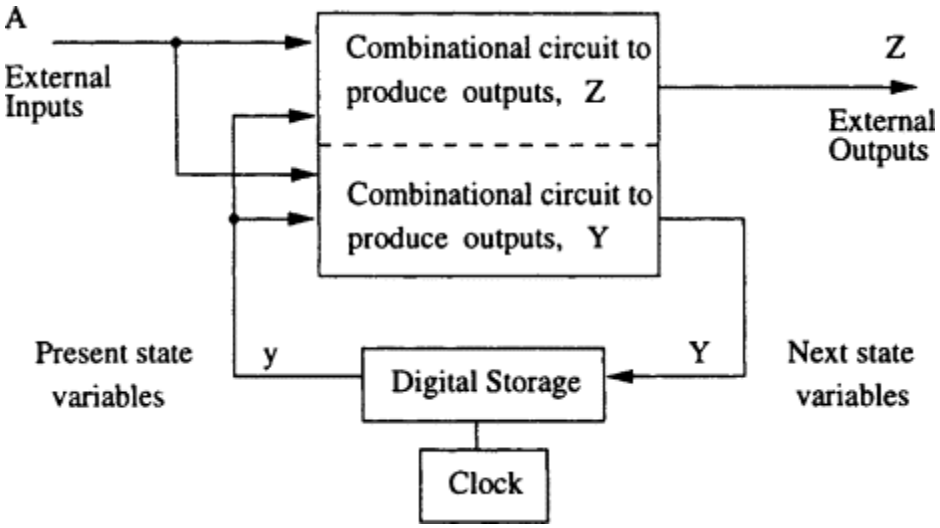


Q-14:Explain divider circuit with hardware implementation. [MEC_C_01]

Q-15:What is Finite State machine? Show the circuit representation of a Synchronous Sequential Machine (FSM). [Pre_2:2_2020] [MEC_C_02]

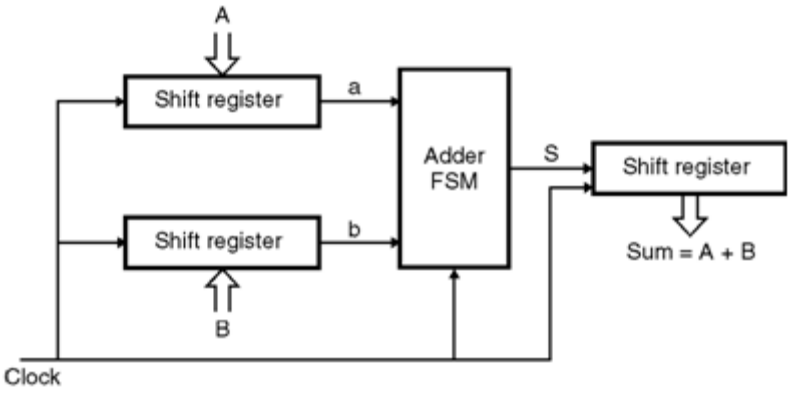
Answer:

A Finite State Machine (FSM) is an abstract machine model used to perform computations and control a system based on a finite number of states. It consists of a specific set of states, input symbols, transitions, and output symbols.

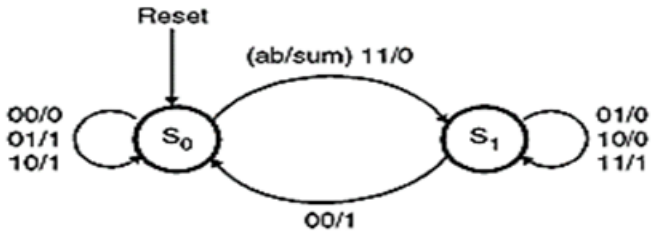


Q-16:Show the Transition table, Transition diagram and block diagram of Serial Binary Adder based on Finite State Machine. [Pre_2:2_2020] [MEC_C_02]

Answer:



The assignment gives the following next state and output equations
P = ab + aQ + bQ Sum = a ï*f*… b ï*f*… Q The serial adder is a simple, circuit that can be used to add numbersof any length. The length of the adder is limited only by the size of the shift register.



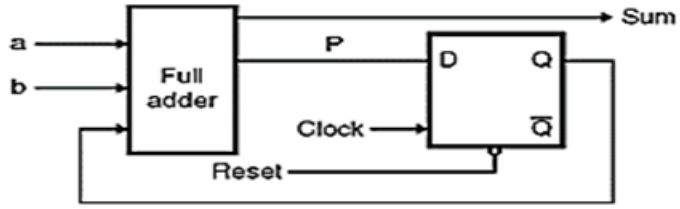
S_0 presents carry in = 0
S_1 presents carry in = 1

State table :

Present state	Next state				Output sum			
	00	01	10	11	00	01	10	11
S_0	S_0	S_0	S_0	S_1	0	1	1	0
S_1	S_0	S_1	S_1	S_1	1	0	0	1

State assignment table :

Present state	Next state (P)				Output			
(Q)	00	01	10	11	00	01	10	11
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1



Q-17:Compare RISC with CISC. [Pre_2:2_2020] [MEC_C_02]

Answer:

🚀 RISC (Reduced Instruction Set Computer):

- Emphasizes simplicity and efficiency.

- Instruction set consists of a limited number of simple and atomic instructions.

- Instructions are usually executed in a single clock cycle, making them faster.

- RISC processors rely on optimizing compilers to perform complex operations through sequences of simple instructions.

- Uses a load/store architecture where data must be explicitly loaded from memory into registers for processing.

🐌 CISC (Complex Instruction Set Computer):

- Emphasizes instruction set completeness and versatility.

- Instruction set consists of a large number of complex instructions that can perform multi-step operations.

- Instructions can take variable clock cycles to execute, with some instructions requiring multiple cycles.

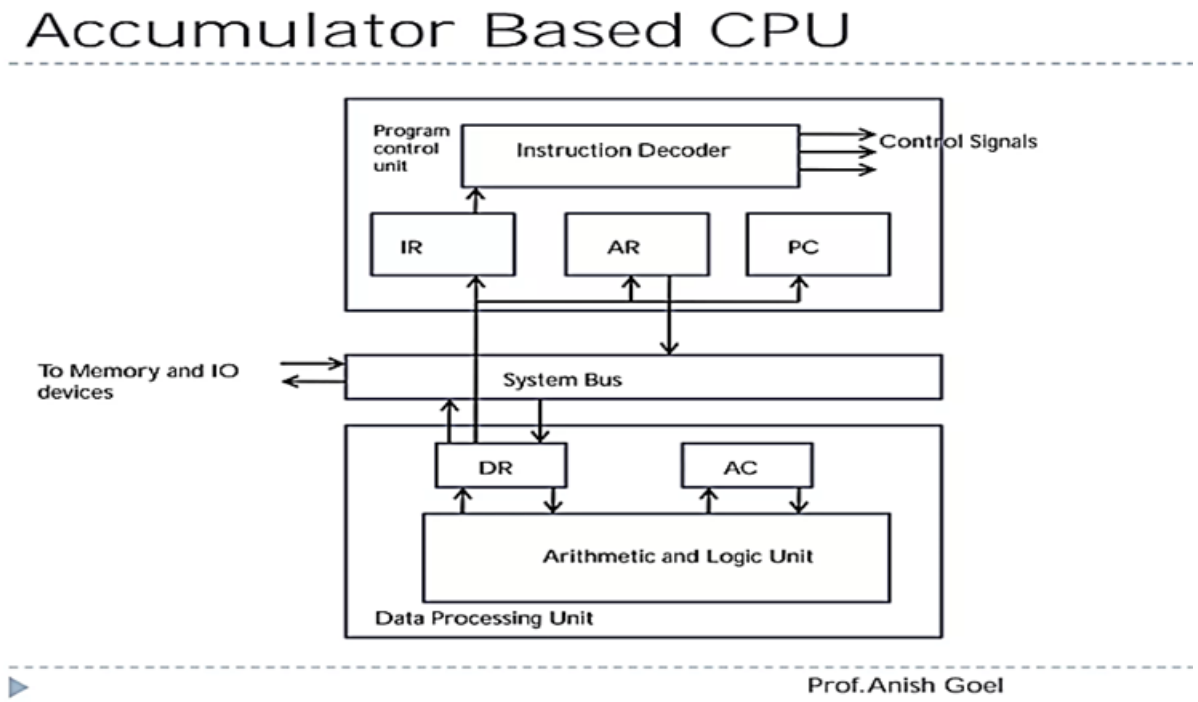
- CISC processors aim to provide a comprehensive set of instructions to simplify programming, allowing high-level operations to be expressed in a single instruction.

- Allows for memory-to-memory operations, reducing the need for explicit data movement between registers and memory.

- Flexible instruction formats allow for variable-length instructions, enabling more compact code.

- Suitable for general-purpose computing and applications that require a mix of complex operations.

Q-18: Show the block diagram accumulator-based CPU Organization. [Pre_2:2_2020] [MEC_C_02]



Q-19:Explain types of Instruction format with example. Write the code to evaluate C-A+B using each instruction format. [Pre_2:2_2020] [MEC_C_02]

Answer:

Three-Address Instruction Format: This format allows operations to be performed on three operands. The result is stored in a separate destination operand. Example: ADD D, A, B (which adds the values of A and B and stores the result in D).

Two-Address Instruction Format: This format allows operations to be performed on two operands. The result is typically stored in one of the operands. Example: MUL A, B (which multiplies the values of A and B and stores the result in A).

One-Address Instruction Format: This format allows operations to be performed on a single operand. The result is typically stored in the same operand. Example: INC A (which increments the value of A by 1).

Zero-Address Instruction Format: This format does not require any operands. The operation is performed on operands implicitly specified by the computer architecture. Example: POP (which pops the top value from the stack).

Three-Address Instruction Format:

```
SUB D, C, A    // Subtract A from C and store the result in D
ADD D, D, B    // Add B to D and store the final result in D
```

Two-Address Instruction Format:

```
SUB C, A      // Subtract A from C and store the result in C
ADD C, B      // Add B to C and store the final result in C
```

One-Address Instruction Format:

```
DEC A         // Decrement the value of A by 1
ADD A, B      // Add B to A and store the final result in A
```

Note: These examples assume that variables A, B, C, and D are predefined and hold the values you want to evaluate.

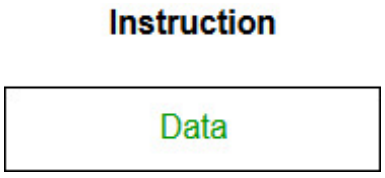
Q-20:What do you mean by addressing mode? Describe any four types of addressing mode. [Pre_2:2_2020] [MEC_C_02]

Answer:

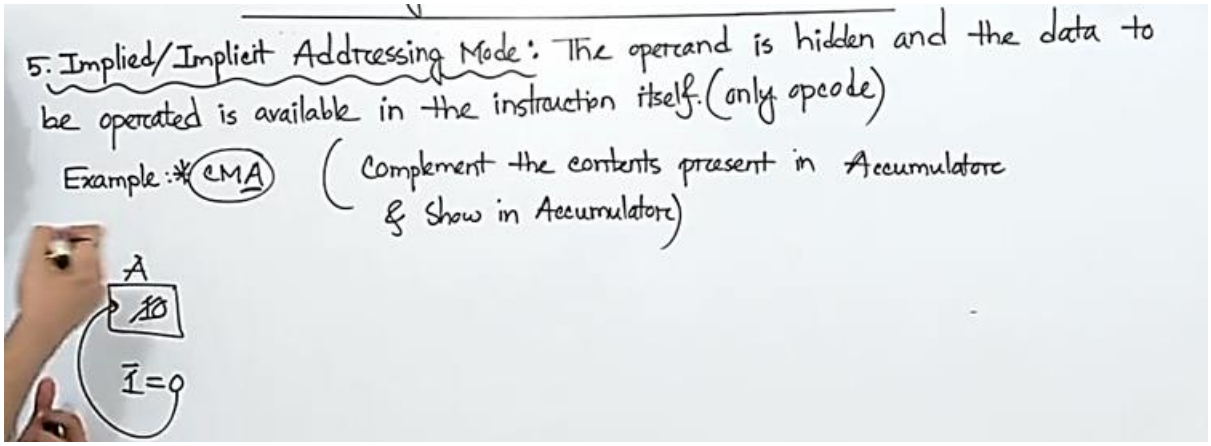
Addressing Modes– The term addressing modes refers to the way in which the operand of an instruction is specified. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

Addressing modes used by 8086 microprocessors are discussed below:

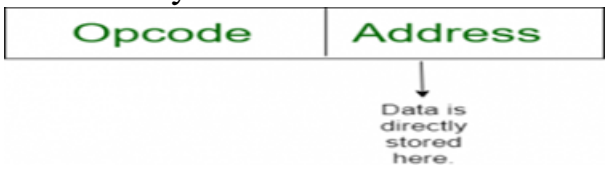
- **Implied mode:** In implied addressing the operand is specified in the instruction itself. In this mode the data is 8 bits or 16 bits long and data is the part of instruction. Zero address instruction are designed with implied addressing mode.



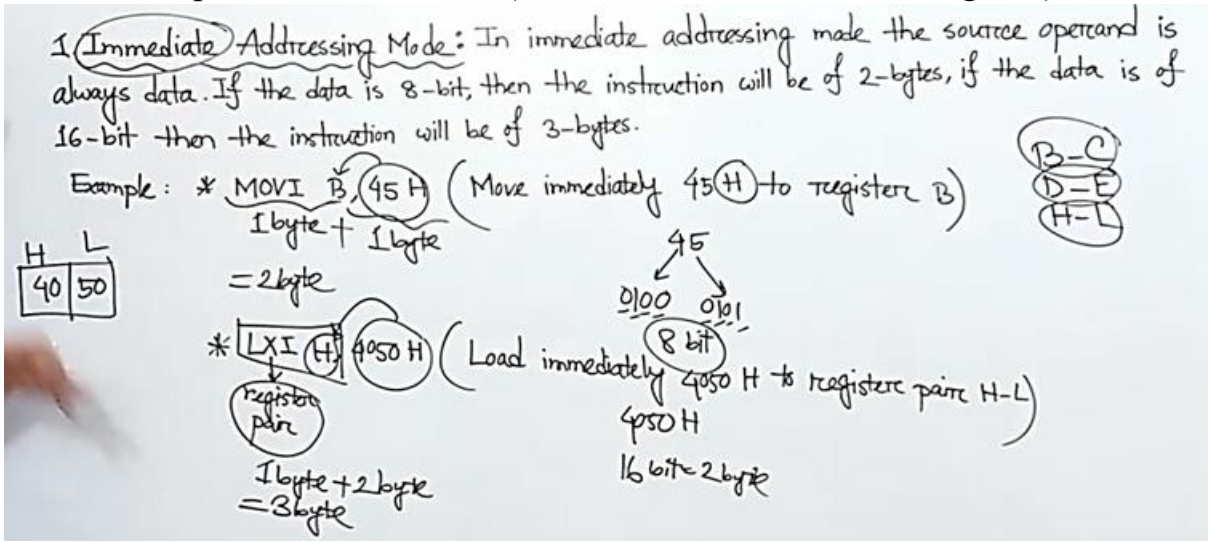
Example: CLC (used to reset Carry flag to 0)



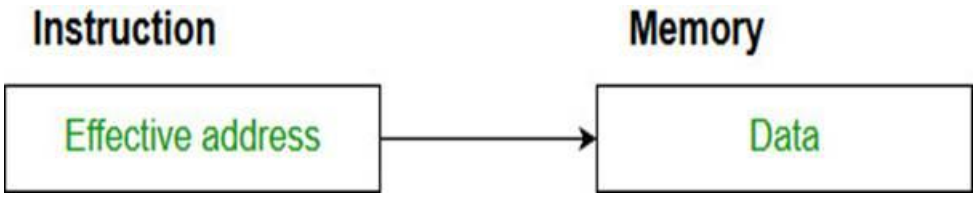
- **Immediate addressing mode (symbol #):** In this mode data is present in address field of instruction. Designed like one address instruction format. **Note:** Limitation in the immediate mode is that the range of constants are restricted by size of address field.



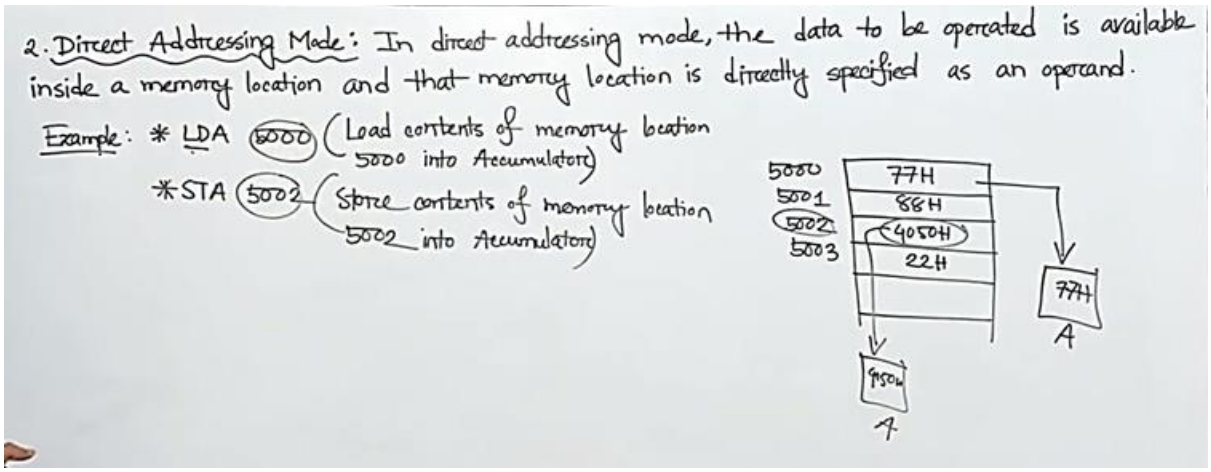
Example: MOV AL, 35H (move the data 35H into AL register)



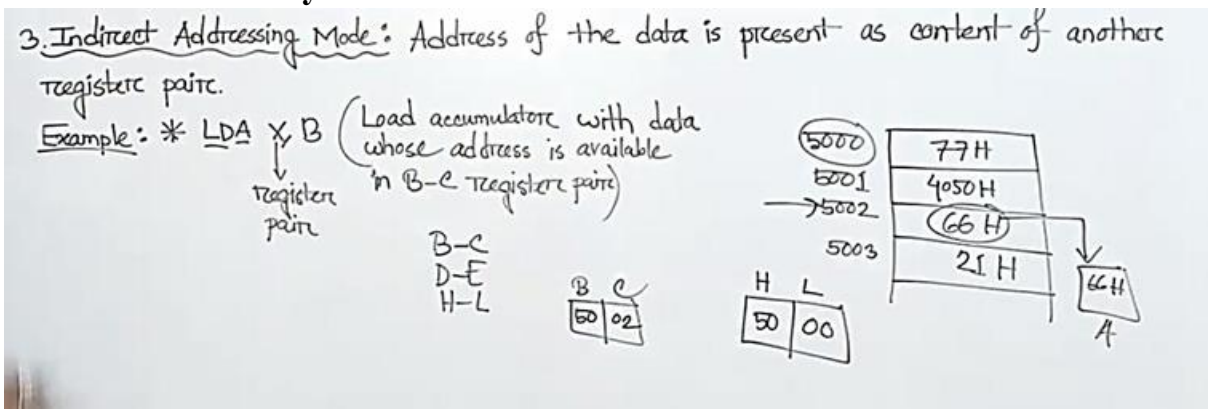
- ***Direct addressing/ Absolute addressing Mode (symbol [ ]):*** The operand's offset is given in the instruction as an 8 bit or 16 bit displacement element. In this addressing mode the 16 bit effective address of the data is the part of the instruction. *Here only one memory reference operation is required to access the data.*



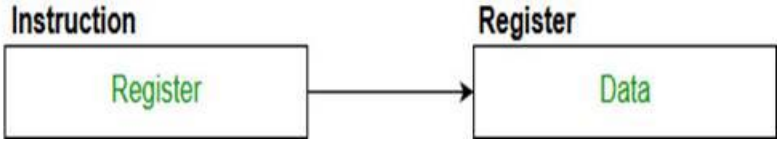
Example: ADD AL,[0301] //add the contents of offset address 0301 to AL



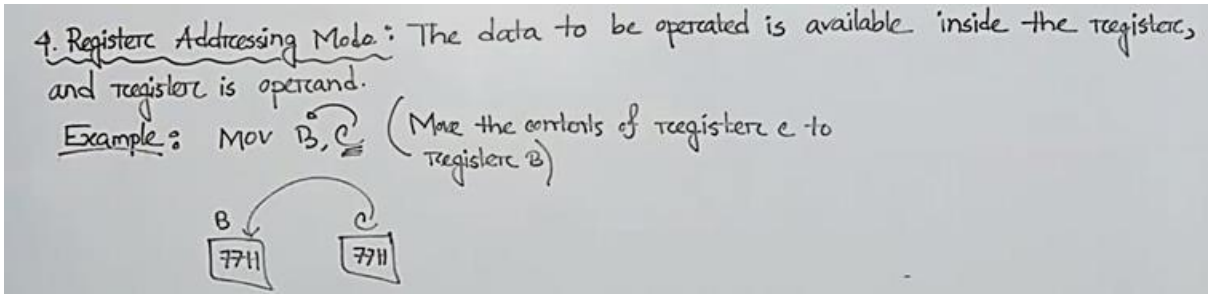
- ***Indirect addressing Mode (symbol @ or ()):***In this mode address field of instruction contains the address of effective address.Here two references are required.
  Based on the availability of Effective address, Indirect mode is of two kind:
  - **Register Indirect**
  - **Memory Indirect**



Register mode: In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction. Here one register reference is required to access the data.



Example: MOV AX, CX (move the contents of CX register to AX register)



- **Register Indirect mode**: In this addressing the operand's offset is placed in any one of the registers BX, BP, SI, DI as specified in the instruction. The effective address of the data is in the base register or an index register that is specified by the instruction.

Q-21: Draw the expanded structure of an IAS computer. [Pre_3:1_2020]

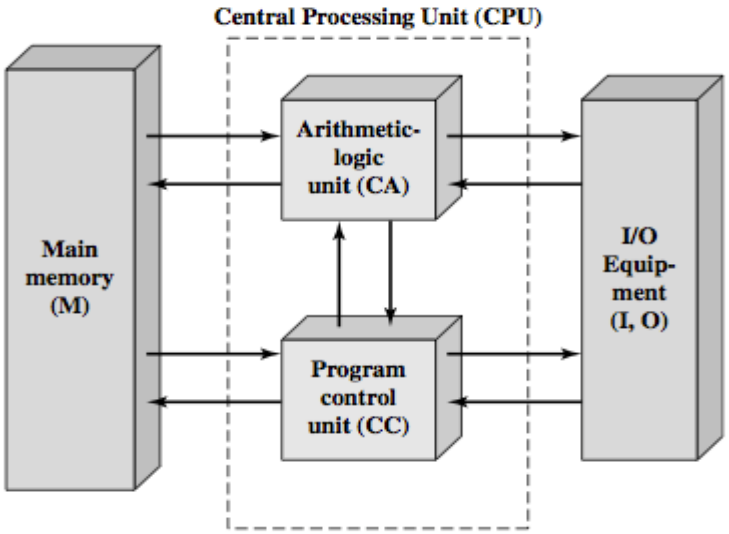


Figure 2.1 Structure of the IAS Computer

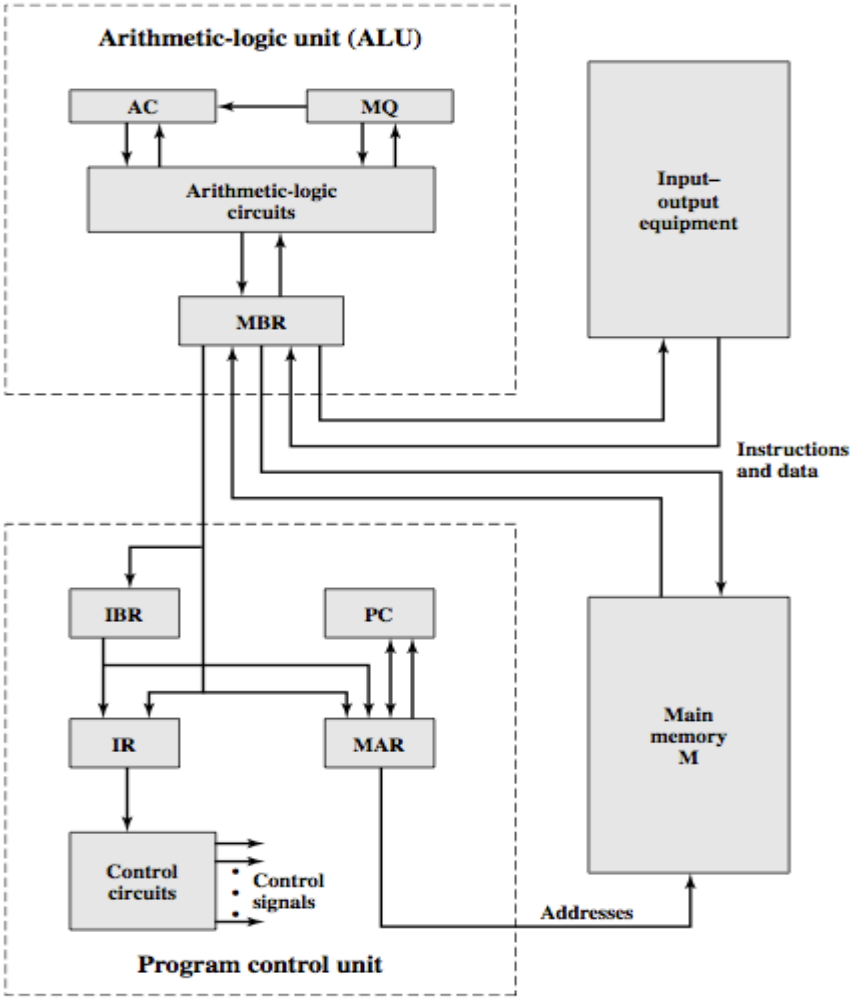


Figure 2.3 Expanded Structure of IAS Computer

Q-22: Assume that an LSI IC at semiconductor memory stores 1024 bits. Assume a main memory unit of 1024 ICs. How many bytes do these ICs store? [Pre_3:1_2020]

Answer:

To calculate the total number of bytes stored by the ICs, we need to consider the bit-to-byte conversion.

In general, 8 bits make up 1 byte. Therefore, to convert bits to bytes, we divide the number of bits by 8.

Given that each LSI IC stores 1024 bits, we can convert this to bytes as follows:

Number of bytes per IC = 1024 bits / 8 bits per byte = 128 bytes per IC

Now, we have a main memory unit comprising 1024 ICs. To determine the total number of bytes stored by these ICs, we can multiply the number of bytes per IC by the number of ICs:

Total number of bytes stored = Number of ICs * Number of bytes per IC

= 1024 ICs * 128 bytes per IC

= 131,072 bytes

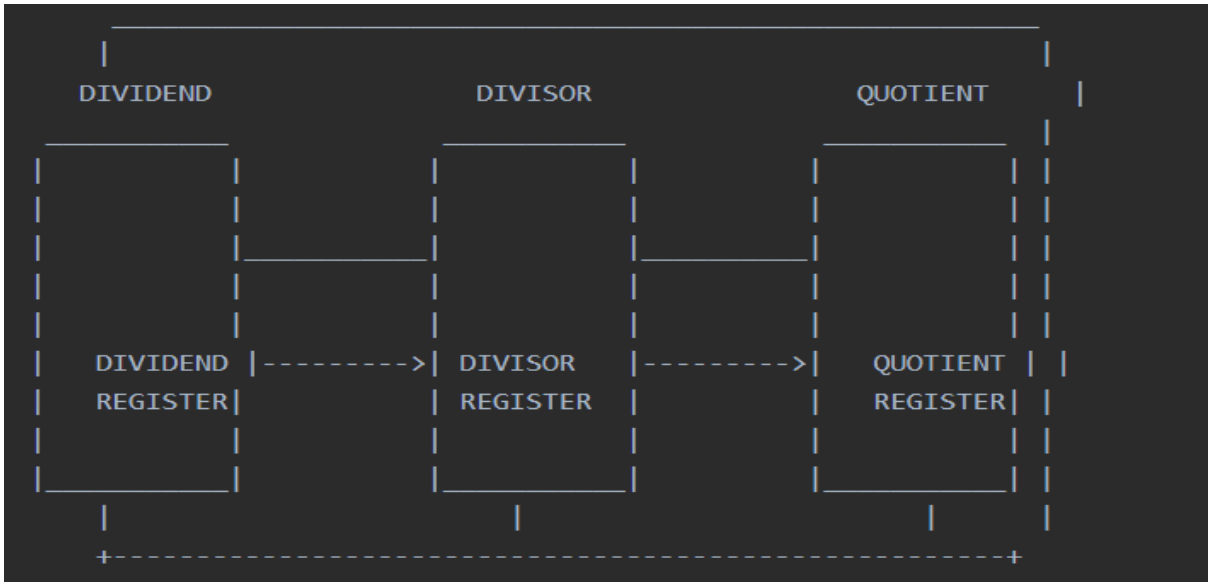
Therefore, the ICs in the main memory unit store a total of 131,072 bytes.

Computer Arithmetic

Q-01: Explain divider circuit with dividend register block diagram. [MEC_C_01]

A divider circuit is used to perform division operations in digital arithmetic. It takes in a dividend (the number being divided) and a divisor (the number by which the dividend is divided) and produces the quotient (the result of the division) and the remainder (the left-over part).

The dividend register is a key component of the divider circuit. It stores the dividend value during the division process. Here is a block diagram representation of the divider circuit with a dividend register:



In this block diagram, the DIVIDEND REGISTER stores the dividend value. The DIVISOR REGISTER holds the divisor value. The QUOTIENT REGISTER stores the quotient value generated during the division process.

During each iteration of the division algorithm, the divider circuit performs the following steps:

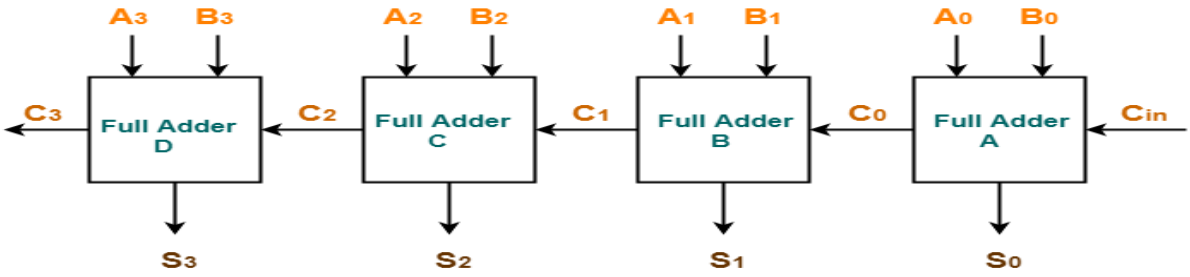
1. Adjust the dividend register: The dividend is shifted left, and the most significant bit is updated based on the remainder from the previous iteration.
2. Compare and subtract: The current dividend value is compared with the divisor. If the dividend is larger or equal, subtraction is performed and the quotient register is updated.
3. Update the remainder: The remainder from the subtraction is stored in the dividend register.
4. Repeat the above steps until the division process is completed.

To design a 4-bit Carry Look-ahead Adder, we need to combine four full adders using the carry look-ahead logic. The carry look-ahead logic reduces the carry propagation delay by generating the carry signals across all stages simultaneously. Here are the appropriate equations for the 4-bit Carry Look-ahead Adder:

```
G0 = A0 AND B0
P0 = A0 XOR B0
C0 = G0

G1 = A1 AND B1
P1 = A1 XOR B1
C1 = (G1 AND C0) OR P1

G2 = A2 AND B2
P2 = A2 XOR B2
C2 = (G2 AND C1) OR (P2 AND C0)

G3 = A3 AND B3
P3 = A3 XOR B3
C3 = (G3 AND C2) OR (P3 AND C1)
```

In these equations, A0 to A3 and B0 to B3 represent the binary inputs for sum bits, G0 to G3 are the individual generate signals, P0 to P3 are the individual propagate signals, and C0 to C3 are the carry-out signals for each stage of the 4-bit adder. The carry signals are generated based on the input bits and the carry signals from previous stages, allowing for parallel computation of the carry signals instead of depending on the carry propagation from one stage to another.

Q-02:Design a 4-bit Carry Look-ahead Adder with appropriate equations. [Pre_2:2_2020]



4-bit Ripple Carry Adder

Consider two 4-bit binary numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are to be added

From here, we have-

$C_1 = C_0 (A_0 \oplus B_0) + A_0B_0$

$C_2 = C_1 (A_1 \oplus B_1) + A_1B_1$

$C_3 = C_2 (A_2 \oplus B_2) + A_2B_2$

$C_4 = C_3 (A_3 \oplus B_3) + A_3B_3$

For simplicity, Let-

- $G_i = A_iB_i$ where G is called carry generator
- $P_i = A_i \oplus B_i$ where P is called carry propagator

Then, re-writing the above equations, we have-

$C_1 = C_0P_0 + G_0$ …….. (1)

$C_2 = C_1P_1 + G_1$ …….. (2)

$C_3 = C_2P_2 + G_2$ …….. (3)

$C_4 = C_3P_3 + G_3$ …….. (4)

- Clearly, C1, C2 and C3 are intermediate carry bits.
- So, let's remove C_1, C_2 and C_3 from RHS of every equation.
- Substituting (1) in (2), we get C_2 in terms of C_0.
- Then, substituting (2) in (3), we get C_3 in terms of C_0 and so on.

Finally, we have the following equations-

- $C_1 = C_0P_0 + G_0$
- $C_2 = C_0P_0P_1 + G_0P_1 + G_1$
- $C_3 = C_0P_0P_1P_2 + G_0P_1P_2 + G_1P_2 + G_2$
- $C_4 = C_0P_0P_1P_2P3 + G_0P_1P_2P_3 + G_1P_2P_3 + G_2P_3 + G_3$

These equations are important to remember.

Q-03:Design the arithmetic unit and show the arithmetic unit function table. [Pre_2:2_2020]

Answer:

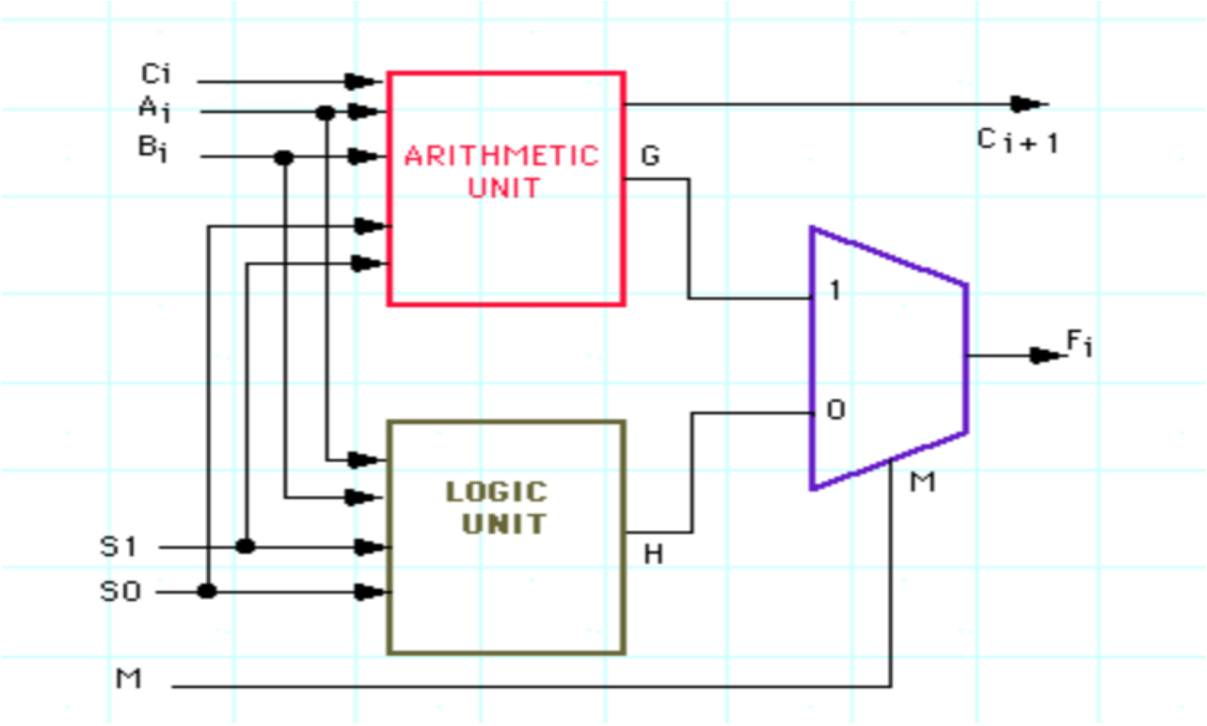
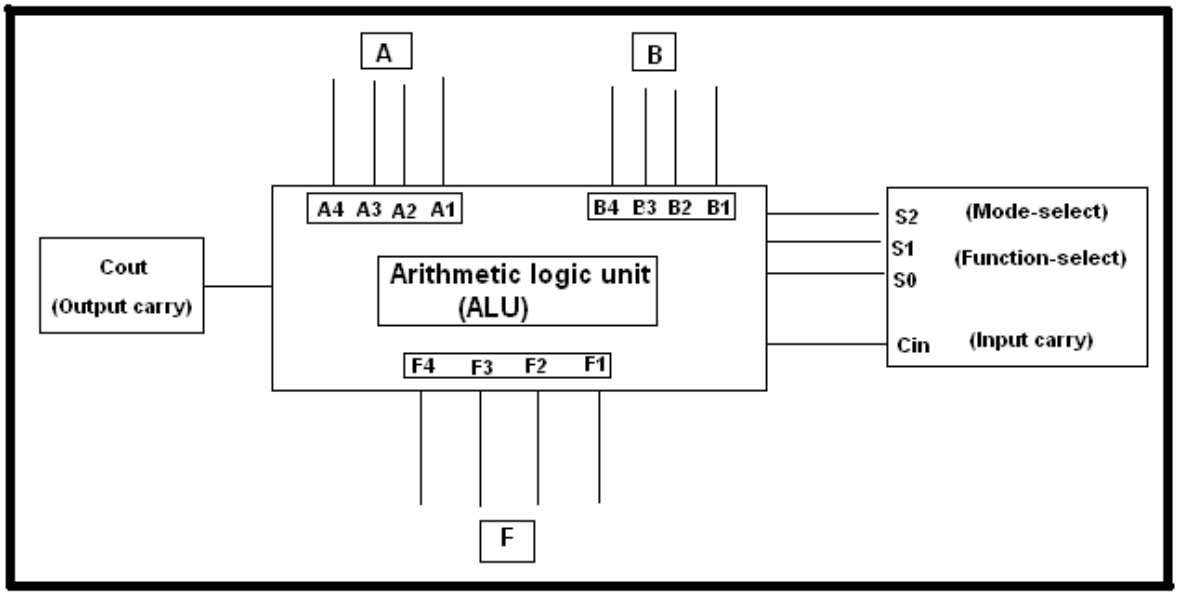


Table:

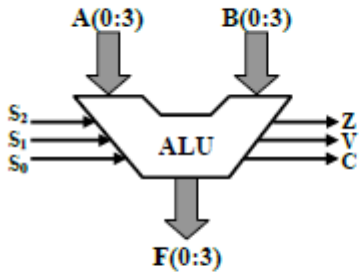
Control Inputs			Output	Results
C_0	C_1	C_2		
0	0	0	B	Transfer B
0	0	1	B+1	Increment B
0	1	0	A + B	Addition
0	1	1	A + B + 1	Addition with carry
1	0	0	A + B	1's complement subtraction
1	0	1	A + B + 1	2's complement subtraction
1	1	0	B-1	Decrement B
1	1	1	B	Transfer B

Q-04:Design 4-bit ALU and explain its function. [Pre_2:2_2020]

Answer:



1. Design a 4-bit Arithmetic Logic Unit (ALU) according to the following specification. Follow the design shown during the lecture. Notice this table is different, though.



Z, C and V are status flags
Z = 1 if F = 0
V = Overflow
C = Carry or Borrow

S_2	S_1	S_0	Function (F)
0	0	0	A + B
0	0	1	A – B
0	1	0	A – 1
0	1	1	A + 1
1	0	0	A AND B
1	0	1	A OR B
1	1	0	NOT A
1	1	1	A XOR B

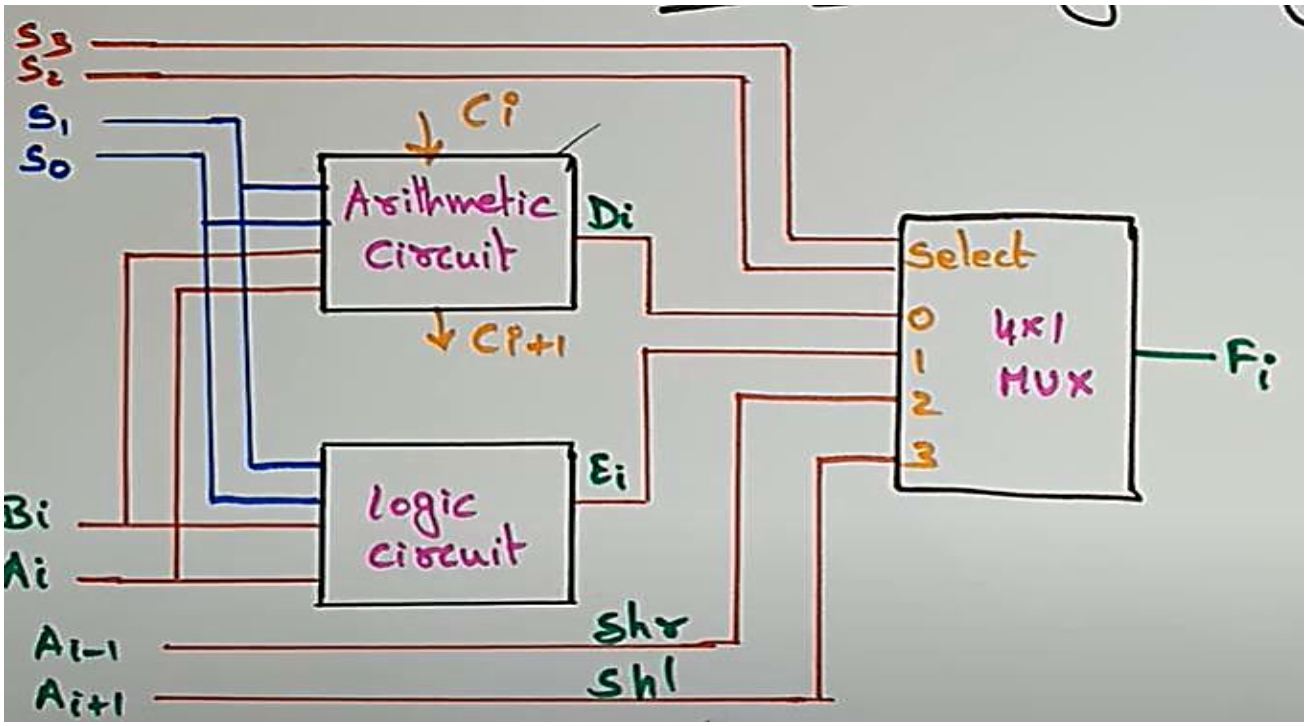
Figure 1 Design specifications.

Let's explain the function of each component:

- A0 and B0 are the least significant bits of the two 4-bit input operands.
- A1 and B1 are the second least significant bits of the two 4-bit input operands.
- ADD/SUB control signal determines whether the operation is addition or subtraction.
- AND performs a bitwise AND operation on A and B.
- OR performs a bitwise OR operation on A and B.
- O0 and O1 are the outputs of the AND and OR operations, respectively.
- OUT0 and OUT1 are the outputs of the ALU.
- Carry In is an additional input for handling carry and borrow operations

Q-05: Define bus transfer and microoperation. Design a 4-bit ALU that performs addition, subtraction, logical AND, logical OR, and logical SHIFT operation. [Pre_3:1_2020]

S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	F= A	Transfer A
0	0	0	0	1	F= A+1	Increment A
0	0	0	1	0	F= A+B	Addition
0	0	0	1	1	F= A+B+1	Addition with carry
0	0	1	0	0	F= A+B'	Sub with borrow
0	0	1	0	1	F= A+B'+1	Subtraction
0	0	1	1	0	F= A-1	Decrement A
0	0	1	1	1	F= A	Transfer A
0	1	0	0	-	F= A^B	AND
0	1	0	1	-	F= A˅B	OR
0	1	1	0	-	F= A ⊕ B	XOR
0	1	1	1	-	F= A'	Complement A
1	0	-	-	-	F= Shift right A	Shift Right A into F
1	1	-	-	-	F= Shift Left A	Shift Left A into F



Q-06: State Amdahl's Law. Write down the CPU performance equation. [Pre_3:1_2020]

Answer:

📚 State Amdahl's Law: Amdahl's Law is a formula that describes the potential speedup of a computing task when only part of the task can be parallelized. The law is named after computer architect Gene Amdahl.

The formula is as follows:

$$\text{Speedup} = \frac{1}{F + \frac{1-F}{P}}$$

where:

- **F** is the fraction of the task that is sequential
- **P** is the fraction of the task that can be parallelized
- **Speedup** is the theoretical improvement in performance.
- **CPU Performance** is the overall performance improvement,
- **Original CPU Performance** is the performance of the original, non-parallelized task.

CPU performance equation

The CPU performance equation, often derived from Amdahl's Law, expresses the overall speedup of a system based on the execution time of the original (non-parallel) task and the speedup achieved by the parallelized portion. The equation is as follows:

CPU Performance = Speedup × Original CPU Performance

Q-07:To achieve a speed-up of 4 on a program that originally took 80 ns to execute, what must the execution time of the program be reduced to? [Pre_3:1_2020]

Answer:

Amdahl's Law provides a formula to calculate the speedup of a program given the fraction of the program that can be parallelized. The formula is:

$$\text{Speedup} = \frac{1}{F + \frac{1-F}{P}}$$

Now, To achieve a speedup of 4 and Assuming P is 1 (indicating the entire program can be parallelized). the equation simplifies to:,

- $4 = \frac{1}{F}$
- $F = \frac{1}{4}$

This means that $\frac{1}{4}$ of the program is sequential and cannot be parallelized.

Now, to find the reduced execution time (T'), we can use the formula:

- $T' = F * T$
- $T' = \frac{1}{4} * 80 = 20$ ns

To achieve a speedup of 4, the execution time of the program must be reduced to 20 s.

Fixed Point Arithmetic Restoring Division Algorithm for Unsigned Integer

Q-01:Represent the decimal number -1.75 in IEEE 754 floating point format. [Pre_2:2_2020]

Answer:

- Sign bit: 1 (negative)
- Exponent: 01111111 (biased exponent for zero)
- Fraction: 11000000000000000000000

Combined: 10111111110000000000000000000000

Q-02:Multiply +23 by -9 using Booth's Multiplication algorithm. [Pre_2:2_2020]

Answer:

Step	Action	Product (P)	Multiplier (Q)
0	Initial State	0000 0000	0000 1001
1	Q[0] = 1	1110 1111	1000 0100
2	Q[1] = 0	1110 1111	1100 0010
3	Right Shift	1111 0111	1110 0001
4	Right Shift	1111 1101	1111 0000
5	Right Shift	1111 1110	1111 1000
6	Right Shift	1111 1111	1111 1100
7	Right Shift	1111 1111	1111 1110

Result: Product (P) = 1111 1111 (2's complement of -207), Multiplier (Q) = 1111 1110 (2's complement of -2).

Q-03:Apply Restoring Division algorithm for dividing the following two numbers: Dividend, Q=8 and Divisor, M=3[Pre_2:2_2020]

Dividend (A) = 8, Quotient (Q) = 8, Divisor (M) = 3.

Step	Action	Quotient (Q)	Remainder (A)
0	Initial State	1000	0010
1	A = A - M	1001	0001
2	A < 0	1010	0001
3	A = A + M	1001	0001
4	A < 0	1010	0010

Result: Quotient (Q) = 1010, Remainder (A) = 0010.

Q-04:What is the necessity of floating point number representation? Write the normalization rules of floating point number. [Pre_2:2_2020]

Necessity: Floating point representation is essential for handling a wide range of numbers, both very small and very large, with a consistent level of precision. It allows efficient representation of real numbers and facilitates arithmetic operations on them.

Normalization Rules:

1. The leftmost digit of the fraction (mantissa) must be 1.
2. Adjust the exponent accordingly for the position of the binary point.
3. The exponent is represented in a biased form.

Q-05:Write short notes on IEEE standard floating point formats. [Pre_3:1_2021]

IEEE 754 Floating Point Formats:

- Single Precision (32 bits): 1 bit for sign, 8 bits for exponent (biased), and 23 bits for fraction (mantissa).
- Double Precision (64 bits): 1 bit for sign, 11 bits for exponent (biased), and 52 bits for fraction (mantissa).

Format: $(-1)^{sign} \times 1.fraction \times 2^{exponent-bias}$$(-1)^{sign} \times 1.fraction \times 2^{exponent-bias}$

Bias: For single precision, bias is 127; for double precision, bias is 1023.

Q-06:What is Booth's algorithm? Explain Booth's multiplication algorithm with example.

[Pre_3:1_2021]

Booth's algorithm is a multiplication algorithm that allows for the efficient multiplication of two binary numbers. It was developed by Andrew Donald Booth in 1951 and is particularly useful for signed binary numbers (numbers with a sign bit, indicating positive or negative).

Here's an explanation of Booth's multiplication algorithm:

Booth's Multiplication Algorithm:

1. **Initialize:**
    - Consider two binary numbers, multiplicand (M) and multiplier (Q), of n-bits each.
    - Initialize an $+1n+1$-bit product (P) to zero.
    - Also, initialize a variable called "accumulator" to zero.
2. **Repeat for *n* cycles:** a. **Check the last two bits of (Q):**
    - If the last two bits are 1010, perform $P=P+M$.
    - If the last two bits are 0101, perform $P=P-M$.

b. **Shift Right:**

- Perform an arithmetic right shift on P and Q.

3. **Repeat until Q becomes zero:**
    - Continue steps 2a and 2b until Q becomes zero.
4. **Final Product:**
    - The final product is stored in the n+1-bit register P.

Example:

Let's multiply =5M=5 and 3Q=3 using 4-bit Booth's algorithm.

1. **Initialize:**
    - M=0101
    - Q=0011
    - P=0000
2. **Iterations:**
    - Iteration 1: Q=1001, P=1111
    - Iteration 2: Q=1100, P=1101
    - Iteration 3: Q=1110, P=1110
    - Iteration 4: Q=1111, P=1111
3. **Result:**
    - P=1111, which is 5×3=M×Q=15.

Finite state machine (DFA & NFA)

CPU Organization and Instruction

Q-01:Define L1 and L2 Cache Memory with examples. [Pre_3:1_2021]

L1 Cache:

L1 cache, also known as Level 1 cache, is a type of cache memory that is located closest to the processor or CPU. It is designed to provide fast access to frequently accessed instructions and data. L1 cache is generally split into two separate caches: the instruction cache (L1i) and the data cache (L1d).

L1 Cache:

- **Smallest and fastest:** Think of it as your CPU's immediate workspace
- **Size:** Measured in kilobytes (KB), typically ranging from 16KB to 32KB per core (modern CPUs can have multiple cores).
- **Content:** Stores most frequently used instructions and data, like recently used calculations or frequently accessed program loops.
- **Example**: If you're editing a document, the words you're currently typing and formatting instructions might be stored in L1 cache for quick access.

Example of L1 Cache: consider a hypothetical CPU with a dual-core processor, where each core has its own L1 cache. Each core might have an L1 cache.

L2 Cache:

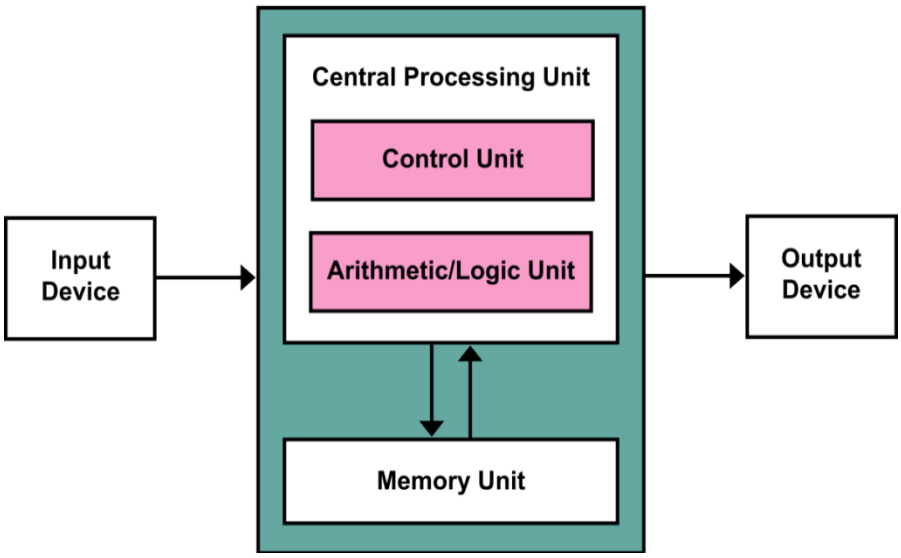
L2 cache, also known as Level 2 cache, is the second level of cache memory in a computer system. It is located between the L1 cache and the main memory. L2 cache is larger in size compared to L1 cache and can hold more instructions and data.

L2 Cache:

- **Larger and slightly slower:** Acts as a secondary layer, still way faster than main memory but slightly slower than L1. Often shared among multiple CPU cores.
- **Size:** Measured in megabytes (MB), typically ranging from 256KB to several megabytes.
- **Content**: Stores data overflow from L1 or frequently used data not immediately needed in L1. Think of it as an overflow bin for your immediate workspace.
- **Example:** When switching tabs in a web browser, recently visited pages might be moved from L1 to L2, ready for quick retrieval if you switch back.

Example of L2 Cache: Continuing with the hypothetical CPU example, the dual-core processor might have a shared L2 cache.

Q-02:Draw the functional block diagram of a Control Unit and describe its operation. [Pre_3:1_2021]



➢ ALU

- This unit executes all arithmetic and logical operations as specified by instruction set; and
- produces output.
- The results of addition, subtraction, and logical operations (AND, OR, XOR) are stored in the
- registers or in memory unit or sent to output unit.

➢ Control Unit

- Controls the operations of different instructions.
- Provides necessary timing and control signals to all the operations in the MP and peripherals
- including memory.

➢ Memory

- Stores binary information such as instruction and data, and provide these information to MP
- when required.
- computing operations.

➢ System bus

- The system bus is a communication path between MP and peripherals.
- It is used to carry data, address and control signals.

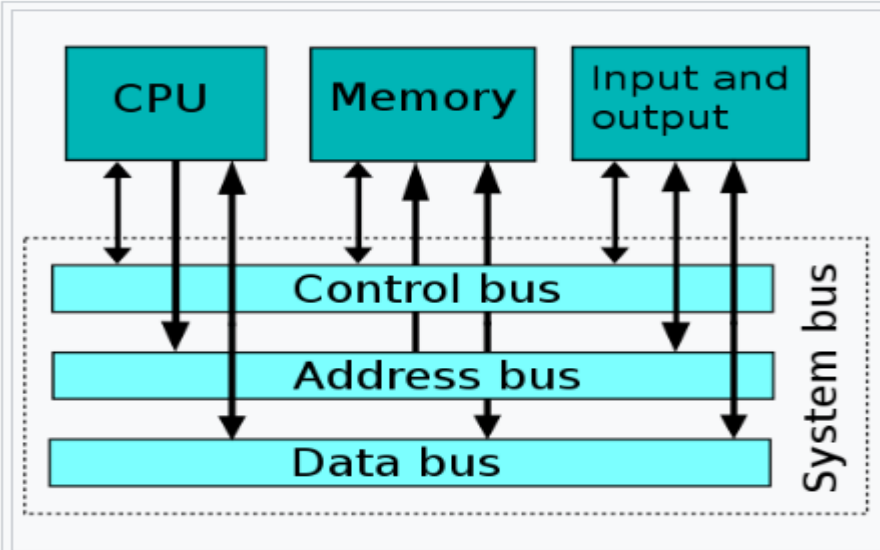
➢ I/O bus

- Input unit is used to input instruction or data to the MP externally.
- Output unit is used to carry out the information from the MP unit.

Q-03:Describe Bus Interfacing with a Processor. [Pre_3:1_2021]

Answer:Bus interfacing with a processor refers to the communication and interaction between the processor and the system bus.

In a typical computer system architecture, the processor is the central processing unit (CPU) responsible for executing instructions and performing calculations.



Bus interfacing involves the following components and processes:

Data Bus: The data bus is a bi-directional pathway through which data is transferred between the processor and other devices. It carries the actual data being read from or written to memory or peripheral devices.

Address Bus: The address bus is a unidirectional pathway used by the processor to transmit memory addresses. It specifies the location in memory where data is to be read from or written to.

Control Bus: The control bus carries control signals that coordinate and control the activities between the processor and other components. It includes signals such as read/write controls, memory enable signals, and interrupt signals.

Bus Arbitration: Bus arbitration is the process by which multiple devices contend for control of the bus when they need to access it simultaneously. Various arbitration techniques, such as priority-based or round-robin, are used to determine which device gets access to the bus at a given time.

Synchronization: Bus interfacing requires synchronization between the processor and other devices. Timing signals and protocols ensure that data is transferred accurately and reliably between components. Clock signals define the timing of bus operations, ensuring that data is sampled and transferred at the correct time.

Bus Protocols: Bus protocols define the rules and standards for communication between the processor and other devices. These protocols specify how data is formatted, how addresses are interpreted, and how control signals are encoded.

Instructions and Addressing Modes

Q-01:Write a program using zero-, two-, and three-address instruction to evaluate the arithmetic statement: X=(ABC)/(D-EF) [Pre_2:2_2020]

- **Write a program using Zero-,two-and three- address instruction to evaluate the aritmatic statement:X=(A+B*C)/(D-E*F)**
- **Describe the expression (A-B) *(C-D) in terms of 4 instruction format (zero, one, two and three address instruction).**

Answer:

Zero Address Instruction

Expression: X = (A+B)*(C+D)
Postfixed : X = AB+CD+*
TOP means top of stack
M[X] is any memory location

PUSH	A	TOP = A
PUSH	B	TOP = B
ADD		TOP = A+B
PUSH	C	TOP = C
PUSH	D	TOP = D
ADD		TOP = C+D
MUL		TOP = (C+D)*(A+B)
POP	X	M[X] = TOP

One Address Instruction

Expression: X = (A+B)*(C+D)
AC is accumulator

M[] is any memory location
M[T] is temporary location

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

Two Address Instruction

Expression: X = (A+B)(C+D)*
R1, R2 are registers
M[] is any memory location

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*

Three Address Instruction

Expression: X = (A+B)(C+D)*
R1, R2 are registers
M[] is any memory location

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*

Q-02:Define fetch and execution cycle. Using three, two, and one address instruction, write a program to evaluate the arithmetic statement X = (A+B)*(C+D). [Pre_3:1_2021]

The fetch and execution cycle, also known as the fetch-decode-execute cycle, is a fundamental process in the operation of a computer's central processing unit (CPU). It describes the sequence of steps performed by the CPU to fetch instructions from memory, decode them, and then execute them.

1. Fetch: In this step, the CPU fetches the instruction from the main memory or instruction cache. The program counter (PC) holds the memory address of the next instruction to be fetched. The CPU retrieves the instruction located at that address and increments the program counter to point to the next instruction.

2. Decode: Once the instruction is fetched, the CPU decodes it to understand its meaning and determine the necessary operations to be performed. The instruction is typically divided into different fields, such as opcode (operation code) and operands.

3. Execute: In this step, the CPU carries out the operation specified by the decoded instruction. The execution phase involves accessing data from registers or memory, performing arithmetic or logical operations, and updating the result back to registers or memory.

After the execution phase, the cycle typically repeats by fetching the next instruction based on the updated program counter and proceeds with decoding and executing that instruction. This cycle continues until the program or process being executed is completed.

Q-03:Assuming that all registers initially contain 0. What is the value of R1 and R3 after the following instruction sequence is executed? [Pre_2:2_2020]

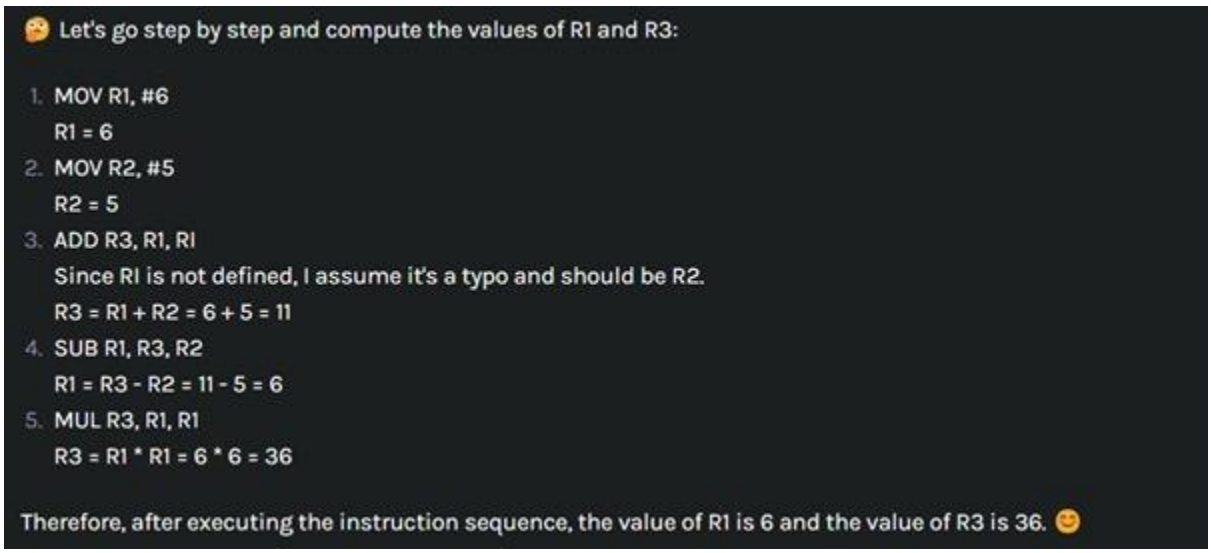
MOV R1, #6

MOV R2, #5

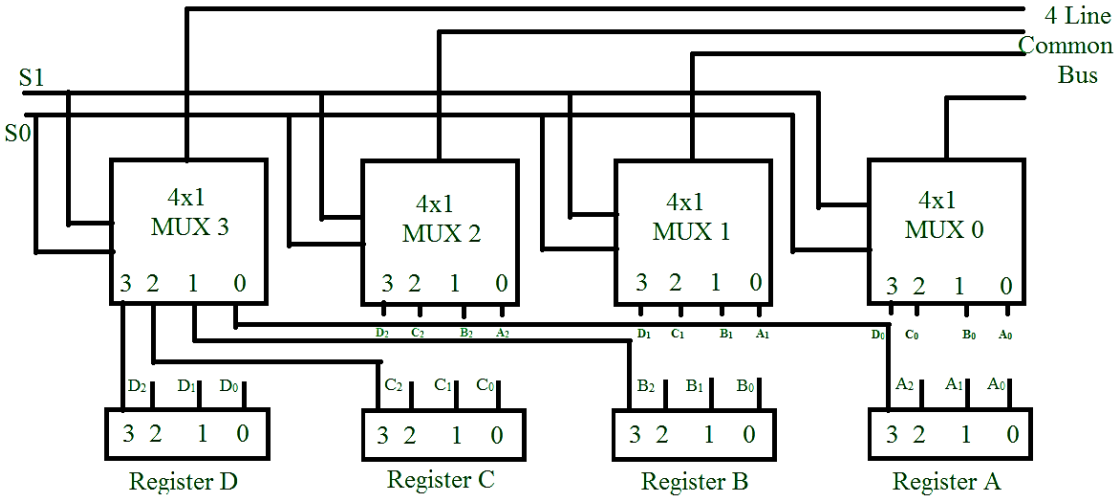
ADD R3, R1, R1

SUB R1, R3, R2

MUL R3, R1, RI



Q-04:Describe the construction of a common bus using multiplexers with a figure. [Pre_3:1_2020]



Let's discuss the common bus system with multiplexers.

The construction of this bus system for 4 registers is shown above. The bus consists of 4×1 multiplexers with 4 inputs and 1 output and 4 registers with bits numbered 0 to 3. There are 2 select inputs S0 and S1 which are connected to the select inputs of the multiplexers.

The output 1 of register A is connected to input 0 of MUX 1 and similarly other connections are made as shown in the diagram. The data transferred to the bus depends upon the select lines. A table for the various combinations of select lines is shown below.

Select Lines combination S1S0	**Register Selected**
00	Register A
01	Register B
10	Register C
11	Register D

As we can see that when S1S0=00, register A is selected because on 00 the 0 data inputs of all the multiplexers are applied to the common bus.

Q-05:What types of operands are typical in machine instruction sets? [Pre_3:1_2020]

Answer:

Machine instruction sets typically have several types of operands that allow the CPU to perform operations on data. Here are some common operand types:

Registers: Registers are small, high-speed memory locations within the CPU. They provide fast access to data and are used for temporary storage during instruction execution. Instructions can involve reading data from registers, writing data to registers, or performing calculations using register contents.

Immediate Values: Immediate values are constant values included directly in the instruction itself. They are used for providing operands or parameters to instructions. For example, an instruction might add a constant value to a register or use a constant as a comparison operand.

Memory Addresses: Memory addresses are used to access data stored in the main memory. They can be specified as part of the instruction or calculated using registers and offsets. Instructions that involve accessing memory locations often include operands representing the source address, destination address, or both.

Pointers: Pointers are memory addresses that point to specific locations in memory. They are commonly used for accessing data structures, arrays, or dynamically allocated memory. Instructions involving pointers often include operands that specify the source or destination of pointer-based operations.

Accumulators: Some older CPU architectures have dedicated accumulators, which are special registers used for arithmetic and logical operations. Instructions in these architectures often operate directly on the accumulator without explicitly specifying operands.

Condition Codes: Condition codes are special registers that hold status flags indicating the results of previous arithmetic or logical operations, such as zero, negative, or carry flags. They are used in conditional instructions to control program flow based on specific conditions.

These operand types are just general examples, and different CPU architectures may have variations or additional operand types depending on their design and intended use.

Q-06:Describe the common addressing modes used for a processor to access data. [Pre_3:1_2020]

Answer:

Here are some common addressing modes used for data access:

1. Immediate Addressing: In immediate addressing, the operand is a constant value or immediate data embedded within the instruction itself. It allows the processor to perform operations directly with the immediate value without needing to fetch it from memory or a register.

2. Register Addressing: Register addressing mode involves using the value stored in a register as the operand for an operation. The instruction specifies the register by its identifier or number, and the processor directly accesses the contents of that register for the operation.

3. Direct Addressing: Direct addressing mode directly specifies the memory address where the operand is located. The memory address is typically included within or referenced by the instruction itself. The processor fetches the data from or stores it to the specified memory location.

4. Indirect Addressing: Indirect addressing involves using a memory address stored in a register or memory location as a pointer to access the operand's actual memory address. The instruction contains the register or memory location address, and the processor retrieves the memory address from the specified location before accessing the operand.

5. Indexed Addressing: Indexed addressing mode adds an offset or index value to a base memory address to calculate the operand's actual memory address. The instruction specifies the base address and the index value, and the processor performs the necessary calculations to access the data.

6. Relative Addressing: Relative addressing mode uses a memory address relative to the current program counter (PC) value. It is often employed in branching or jumping instructions, where the instruction sets the offset from the current PC value to determine the target address.

7. Stack-based Addressing: Stack-based addressing mode involves using the top element of a stack as an operand. The stack pointer keeps track of the stack's top, and the processor directly accesses the topmost element for operations.

These are some of the commonly used addressing modes, and different processor architectures may have additional modes or variations based on their design and intended purpose.

Pipeline Processing and Data Control and Instruction Pipelining

Q-01: What is pipelining? Explain Performance of Pipeline Processor. [Pre_2:2_2020]

Answer:

Pipelining is a technique used in computer architecture to improve the performance of processors by allowing them to execute multiple instructions concurrently. It divides the execution of instructions into a series of stages, with each stage handling a specific task. Each stage operates on a different instruction, creating an assembly line-like structure where each instruction enters the pipeline and progresses through the stages until it completes.

The performance of a pipeline processor can be evaluated based on the following factors:

1. Throughput: Pipeline processors aim to achieve higher throughput, which refers to the rate at which instructions are completed per unit of time. By allowing multiple instructions to be in different stages of execution simultaneously, pipelines enable parallel processing and enhance overall system performance.

2. Latency: Although pipeline processors improve throughput, they may introduce an increase in latency. Latency refers to the time taken for a single instruction to complete its execution. Pipelining introduces pipeline overhead and processing delays due to the introduction of stages, which can increase individual instruction latency. Balancing the number of stages and the complexity of pipeline operations is crucial to minimize latency.

3. Instruction-Level Parallelism (ILP): Pipelines exploit ILP, which is the ability to execute multiple instructions in parallel. By dividing instruction execution into stages, the pipeline allows overlapping of operations, minimizing idle processor cycles and maximizing the utilization of functional units within the processor.

4. Hazards: Pipeline hazards are situations that arise when one instruction depends on the result of a previous instruction that has not completed yet. Hazards can cause delays or incorrect results. Types of hazards include data hazards, control hazards, and structural hazards. Techniques such as forwarding, branch prediction, and instruction scheduling are employed to mitigate these hazards and improve pipeline performance.

5. Pipeline Efficiency: The efficiency of a pipeline processor is determined by the extent to which the pipeline stages are utilized. Pipeline stalls and bubbles occur when instructions cannot progress due to dependencies or conflicts, resulting in reduced efficiency. Techniques such as out-of-order execution and speculative execution are employed to minimize stalls and increase pipeline efficiency.

Overall, pipelining enhances the performance of processors by enabling parallel execution of instructions, exploiting ILP, and increasing throughput. However, careful design and consideration of factors such as latency, hazards, and efficiency are required to achieve optimal performance gains. 🚀

Q-02: What do you mean by pipelining? Design a floating-point adder pipelining. [Pre_3:1_2021]

Answer:

Pipelining, in the context of computer architecture, refers to a technique that divides the execution of instructions into a series of stages, allowing multiple instructions to be processed concurrently. Each stage performs a specific operation, and instructions progress through the stages in a pipeline fashion, similar to an assembly line, to achieve increased throughput and performance.

Here's a simplified example of a three-stage floating-point adder pipeline: 💉➕🚀

Stage 1: Fetch Operand

In this stage, the operands for the addition operation are fetched from the memory or register file.

Stage 2: Add Operation

The fetched operands are added together in this stage using the floating-point addition operation.

Stage 3: Write Result

The result of the addition operation is written back to the destination register or memory.

To ensure correct pipelining and avoid hazards, additional steps may be required, such as register renaming, operand forwarding, and hazard detection.

Here's an example illustrating the pipelining of three floating-point addition instructions:

Instruction 1:

1. Fetch Operand (Stage 1)
2. Add Operation (Stage 2)
3. Write Result (Stage 3)

Instruction 2:

1. Fetch Operand (Stage 1, while Instruction 1 is in Stage 2)
2. Add Operation (Stage 2, while Instruction 1 is in Stage 3)
3. Write Result (Stage 3, while Instruction 1 is completing)

Instruction 3:

1. Fetch Operand (Stage 1, while Instruction 2 is in Stage 2)
2. Add Operation (Stage 2, while Instruction 2 is in Stage 3)
3. Write Result (Stage 3, while Instruction 2 is completing)

By overlapping the execution of instructions in different stages, the pipeline allows multiple instructions to be in progress simultaneously, leading to improved performance and throughput.

Q-03: Define pipeline hazards? What are the different types of pipeline hazards? Explain each of them. [Pre_3:1_2021]

Answer:

Pipeline hazards are situations that occur in pipelined computer architectures where the execution of instructions is hindered or delayed due to conflicts or dependencies between instructions. These hazards can impact the performance and efficiency of pipelined processors. There are three main types of pipeline hazards:

1. Structural Hazards:

Structural hazards occur when there is contention for system resources such as registers, functional units, or memory. If multiple instructions require the same resource simultaneously, a structural hazard arises. It leads to resource conflicts and requires operations to be serialized, causing pipeline stalls. For example, if an instruction requires a specific functional unit that is already occupied by another instruction, a structural hazard occurs.

2. Data Hazards:

Data hazards occur when there are dependencies between instructions that can lead to incorrect results if instructions are executed out of order. There are three subtypes of data hazards:

a) Read-after-Write (RAW) Hazard:

An RAW hazard, also known as a true dependency or data dependency, occurs when an instruction depends on the result of a previous instruction that is still in progress in the pipeline.

b) Write-after-Read (WAR) Hazard:

A WAR hazard, also known as an anti-dependency, occurs when a later instruction writes to a location that an earlier instruction reads.

c) Write-after-Write (WAW) Hazard:

A WAW hazard, also known as an output dependency, occurs when multiple instructions try to write to the same destination register.

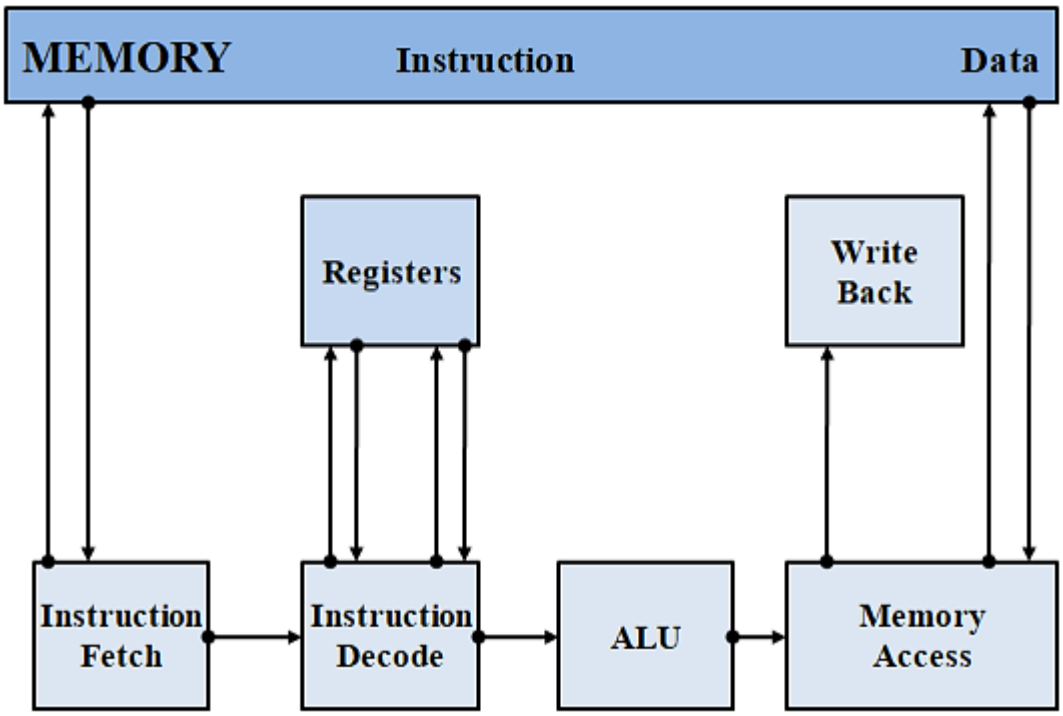
3. Control Hazards:

Control hazards occur when there is a change in the control flow of instructions, such as branches, jumps, or conditional instructions. Control hazards arise due to the possibility of branches being taken or not taken, which makes it difficult to predict the next instruction to be fetched accurately. Fetching and executing the wrong branch path can cause wasted work and pipeline bubbles. Control hazards can be mitigated using branch prediction techniques, such as branch target buffers and branch prediction tables.

To handle pipeline hazards and minimize their impact on pipeline performance, techniques such as forwarding, operand bypassing, branch prediction, and out-of-order execution are employed in modern pipelined processors. These techniques aim to reduce stalls, maintain correct program execution order, and maximize pipeline utilization. 🚧🤔💥

Q-04: Draw a Pipelined Data path for LOAD word and describe its stages. [Pre_3:1_2020]

Answer:



Description of Stages:

1. Stage 1 (Instruction Fetch): In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.

2. Stage 2 (Instruction Decode): In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.

3. Stage 3 (Instruction Execute): In this stage, ALU operations are performed.

4. Stage 4 (Memory Access): In this stage, memory operands are read and written from/to the memory that is present in the instruction.

5. Stage 5 (Write Back): In this stage, computed/fetched value is written back to the register present in the instructions.

Q-05: What is an ALU? Draw a block diagram for a 4-bit ALU. [Pre_3:1_2020]

ALU (Arithmetic Logic Unit): An Arithmetic Logic Unit (ALU) is a digital circuit within a computer's central processing unit (CPU) that performs arithmetic and logic operations on data.

Design a 4 function ALU: to design 4-bit ALU, there we will need 2-bit logic unit and 2-bit arithmetic unit

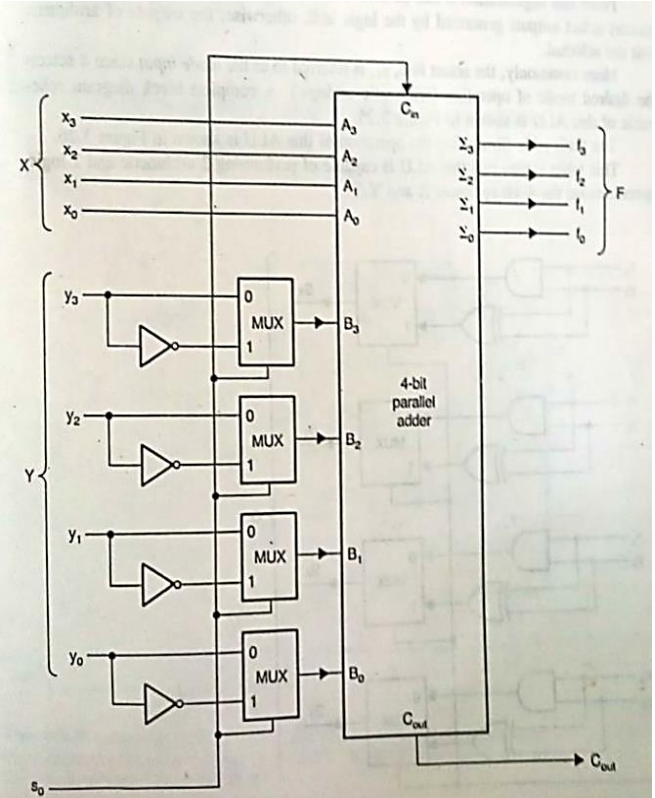


Figure 3.22 Organization of an Arithmetic Unit

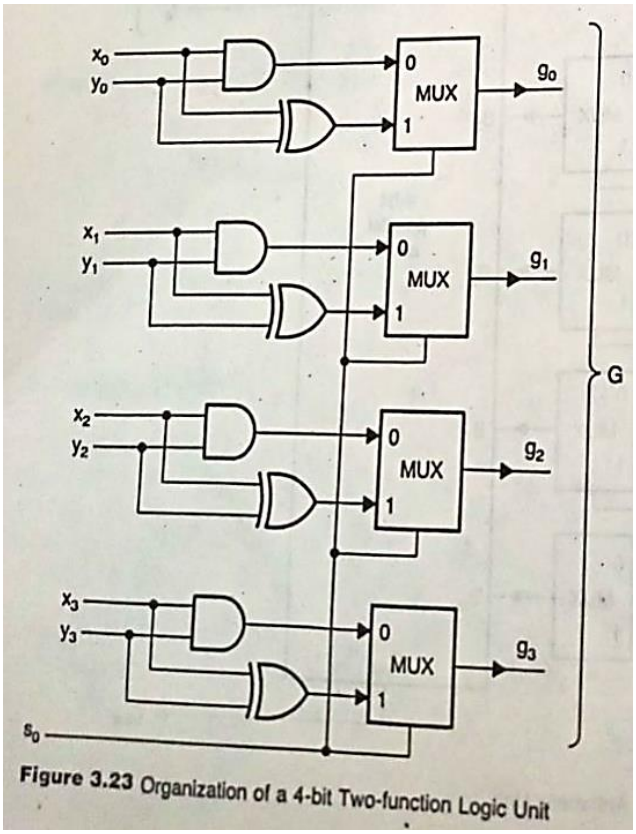


Figure 3.23 Organization of a 4-bit Two-function Logic Unit

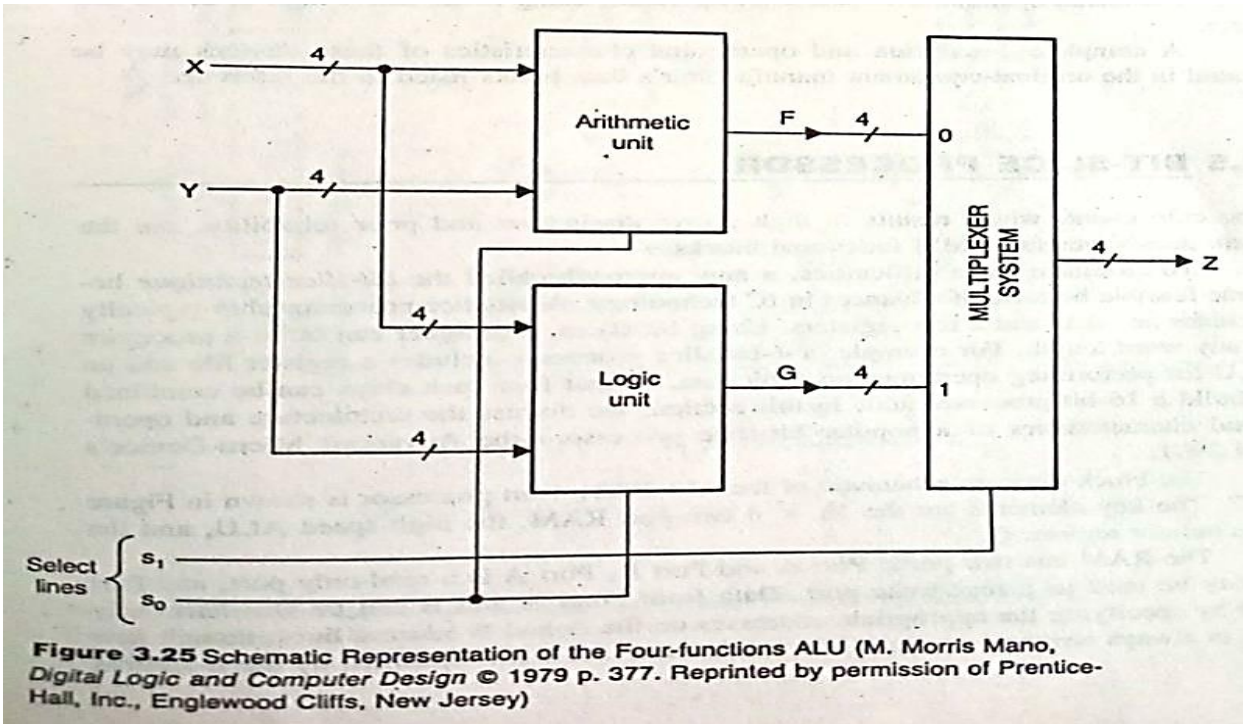


Figure 3.25 Schematic Representation of the Four-functions ALU (M. Morris Mano, *Digital Logic and Computer Design* © 1979 p. 377. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, New Jersey)

Q-06: What is pipelining? List out the ways to characterize pipelining. [Pre_3:1_2020]

Answer:

Pipelining is a technique used in computer architecture to improve the efficiency of instruction execution. It involves breaking down the execution of instructions into smaller sequential stages, allowing multiple instructions to overlap in their execution.

Ways to characterize pipelining include:

1. Instruction Pipeline: In this type of pipelining, the execution of an instruction is divided into distinct stages, with each stage handling a specific operation. The instruction pipeline typically includes stages such as instruction fetch, instruction decode, execute, memory access, and writeback.

2. Data Pipeline: Data pipelining focuses on the flow of data through different stages of the processing unit.

3. Instruction-Level Pipelining: Also known as ILP, this type of pipelining exploits the parallelism that exists among multiple instructions.

4. Structural Pipelining: Structural pipelining involves splitting the processor hardware into separate stages, each responsible for a particular task or operation.

5. Control Pipelining: Control pipelining focuses on dividing the control signals and control flow decisions into stages.

6. Superscalar Pipelining: Superscalar pipelining refers to the concept of pipelining that incorporates multiple functional units within the processor.

7. Instruction Dependency Handling: Pipelining also involves techniques for handling dependencies between instructions. Common methods include data forwarding on pipeline stalls and performance. 🚀 ⚙️ 💡

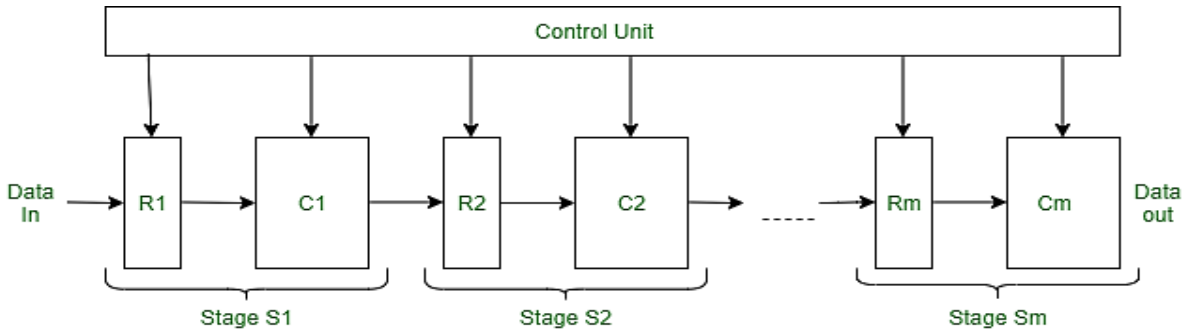


Figure - Structure of a Pipeline Processor

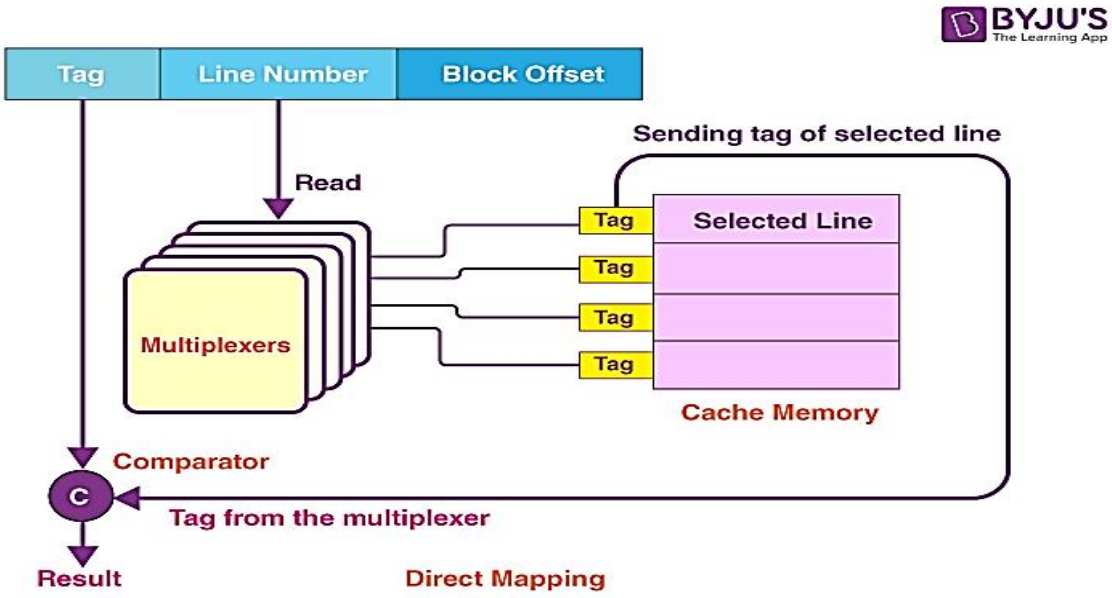
Memory Organization in Computer Architecture:

Q-01: Explain the Direct cache mapping Technique with example. [Pre_2:2_2020]

Answer:

Direct Cache Mapping Technique

In a nutshell, direct cache mapping assigns each memory block in the main memory to a specific, predetermined line in the cache. This means each block has only one possible location in the cache. It's the simplest cache mapping technique, but it can lead to conflicts when multiple blocks compete for the same cache line.



Here's how it works:

1. Address Division: The memory address is divided into three fields:
    - Tag: Identifies the block in main memory.
    - Line Index: Specifies the cache line where the block can reside.
    - Block Offset: Points to the specific word within the block.
2. Cache Access:
    - When the CPU requests data, the line index is used to directly access the corresponding cache line.
    - The tag field of the memory address is compared with the tag stored in the cache line.
    - If the tags match (cache hit), the data is retrieved from the cache.

Q-02:What is the limitation of Direct Cache mapping? How you can solve it describe proper example. [Pre_2:2_2020]

Answer:

Limitations of Direct Cache Mapping:

Cache Conflicts:

- When multiple memory blocks map to the same cache line, they compete for the same space, leading to frequent evictions and replacements. This is called a "conflict miss" and can significantly reduce performance, especially with workloads that exhibit unpredictable memory access patterns.

Thrashing:

- In extreme cases, excessive conflicts can result in "cache thrashing," where blocks are constantly being evicted and reloaded, causing the cache to become ineffective and potentially slowing down the system.

Uneven Access Distribution:

- If memory blocks aren't evenly accessed across the address space, some cache lines may become heavily loaded while others remain underutilized, leading to suboptimal cache utilization and performance.

Solving the Limitations:

Set-Associative Cache Mapping:

- This technique divides the cache into sets, where each set contains multiple cache lines.
- A memory block can map to any line within its designated set, reducing conflicts and improving cache hit rates.

Fully Associative Cache Mapping:

- In this approach, a block can be placed in any cache line, providing maximum flexibility and potentially the highest hit rates.
- However, it requires complex hardware implementation and can have longer access times due to the need to search all cache lines.

Additional Techniques:

- Cache Replacement Policies: The choice of replacement policy (e.g., LRU, FIFO, Random) can also impact cache performance in direct-mapped caches.
- Cache Prefetching: Anticipating future memory accesses and proactively loading blocks into the cache can help reduce misses.
- Cache Blocking: Rearranging data structures to improve cache locality can also enhance performance.

Q-03: How many 256 x 4 RAM chips are needed to provide a memory capacity of 2048 bytes? [Pre_2:2_2020]Show the corresponding interconnection diagram. [Pre_2:2_2020]

Answer:

To calculate the number of 256 x 4 RAM chips needed to provide a memory capacity of 2048 bytes, we can use the following steps:

1. Determine the number of bits required to represent the memory capacity:

Number of bits = Memory capacity in bytes x 8

Number of bits = 2048 bytes x 8

Number of bits = 16384 bits

2. Determine the capacity of a single 256 x 4 RAM chip:

Capacity of a single RAM chip = Number of addressable locations x Number of bits stored in each location

Capacity of a single RAM chip = 256 x 4 bits

3. Calculate the number of RAM chips required:

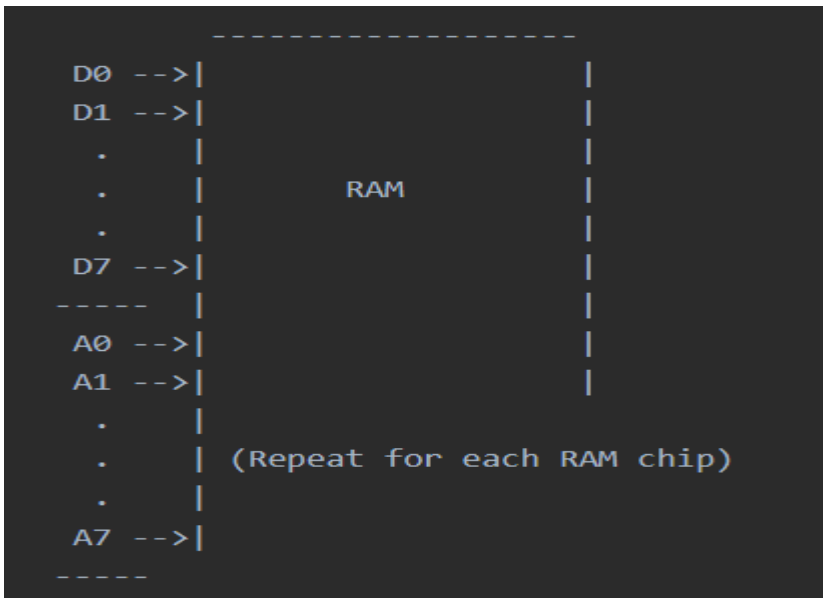
Number of RAM chips = Number of bits required / Capacity of a single RAM chip

Number of RAM chips = 16384 bits / (256 x 4 bits)

Number of RAM chips = 16 RAM chips

Therefore, you would need 16 256 x 4 RAM chips to provide a memory capacity of 2048 bytes.

Now, let's look at the corresponding interconnection diagram:



In the diagram, each RAM chip is represented as a rectangle. The input data lines (D0-D7) and address lines (A0-A7) are connected to each RAM chip. 😊 🔢 💡

Q-04:Explain Interrupt and Interrupt Handling Mechanism with block diagram. [Pre_2:2_2020]

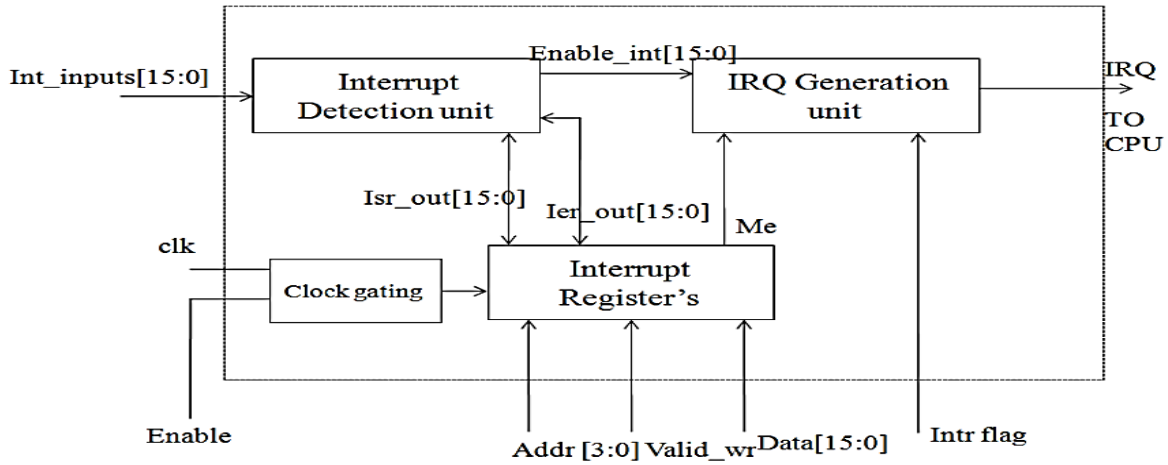
Answer:
Interrupts and Interrupt Handling Mechanism

Interrupts are signals that alert the CPU to pause its current execution and attend to an urgent event or task. They are crucial for efficient system operation, handling time-sensitive events, and enabling multitasking.

Interrupt Handling Mechanism outlines the steps the CPU undertakes to manage interrupts:

1. **Interrupt Request (IRQ):**
   - An external device or internal system component signals an interrupt by setting an IRQ line to a high voltage level.
2. **Interrupt Detection:**
   - The CPU's Interrupt Controller (IC) continuously monitors IRQ lines.
3. **Interrupt Acknowledgement:**
   - The CPU acknowledges the interrupt by sending a signal to the IC.
4. **Interrupt Identification:**
   - The IC determines the interrupt's source (device or event).
5. **Interrupt Service Routine (ISR) Execution:**
   - The CPU jumps to a specific memory address where the ISR for the identified interrupt resides.
6. **Interrupt Return:**
   - Once the ISR completes, the CPU restores the previously saved state and resumes its interrupted task.

Block Diagram:



Q-05:Draw the basic connection between processor and memory as well as write down their operational details. [Pre_3:1_2021]

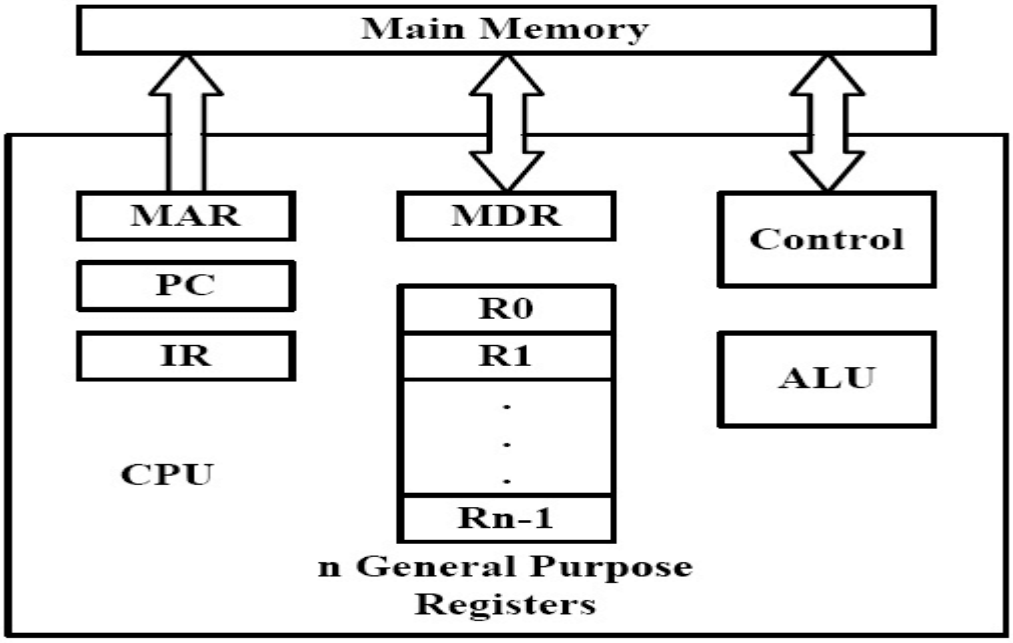
Connections and Operational Details of Processor and Memory

The processor and memory form the core of a computer system, constantly interacting to fetch instructions, store data, and perform computations. Here's a breakdown of their basic connections and operational details:

Connection:

The processor and memory are physically connected through a set of buses:

- Address Bus: Carries the memory address of the data or instruction the processor wants to access.
- Data Bus: Transfers the actual data between the processor and memory.
- Control Bus: Sends control signals like read/write commands and synchronization signals.



There are two operational details to consider:

1. Read Operation: When the processor wants to read data from a specific memory location, it places the address of that location on the address bus. The memory receives the address and retrieves the data stored at that location. The data is then sent back to the processor, typically through a data bus.
2. Write Operation: When the processor wants to write data to a specific memory location, it places the address of that location on the address bus along with the data to be written. The memory receives the address and data and stores the data at the specified location.

Q-06:Describe the types of operations used in Instruction. [Pre_3:1_2021]

Answer:

Here are some common types:

1. Arithmetic Operations: These instructions perform mathematical calculations such as addition, subtraction, multiplication, and division on numerical values. They typically operate on registers within the processor.

2. Logical Operations: These instructions manipulate binary data based on logical operations such as AND, OR, NOT, and XOR. They are used for tasks like bit manipulation and boolean logic operations.

3. Data Transfer Operations: These instructions move data between registers, memory, and other input/output devices. They enable the processor to read data from memory, write data to memory, or transfer data between different registers.

4. Control Transfer Operations: These instructions control the flow of execution within a program. They include instructions for branching (jumping) to different parts of the program based on conditions, looping (repeating) a block of code, and calling and returning from subroutines.

5. Input/Output Operations: These instructions enable the processor to communicate with external devices such as keyboards, displays, and storage devices. They facilitate the exchange of data between the processor and the outside world.

6. System Control Operations: These instructions manage system-level operations, such as enabling or disabling interrupts, changing processor modes, and handling exceptions or system calls.

Q-07:Define locality of reference. [Pre_3:1_2021]

Locality of reference refers to the observation that programs tend to access a relatively small portion of their address space or memory at any given time. It is a principle that highlights the pattern in which data and instructions are accessed or referenced by a program.

There are two main types of locality of reference:

1. Temporal Locality: This refers to the tendency of a program to access the same data or instructions repeatedly within a short period of time. It suggests that if a certain memory location has been accessed recently, it is likely to be accessed again in the near future. Caching mechanisms take advantage of temporal locality by storing recently accessed data or instructions in a faster and closer memory hierarchy, reducing the need to access the slower main memory.

2. Spatial Locality: This refers to the tendency of a program to access data or instructions that are near each other in memory. It implies that if a certain memory location has been accessed, it is likely that nearby locations will also be accessed. Spatial locality is exploited by memory systems through techniques such as block or page-based transfers, where a single read or write operation retrieves or stores multiple contiguous memory locations.

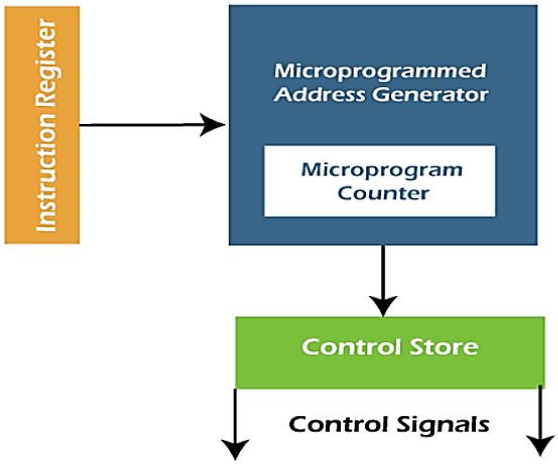
Q-08:What is the function of the control unit? Using necessary diagrams, explain the function of a microprogrammed control unit. [Pre_3:1_2021]

Answer:

The control unit is an essential component of a computer's central processing unit (CPU). Its main function is to interpret and execute instructions by coordinating activities within the CPU and other hardware components.

Microprogrammed Control Unit (MCU):

A microprogrammed control unit implements the control unit's logic using microprograms stored in a special memory called control memory. This provides flexibility and ease of modification.



Functioning of an MCU:

Fetch Microinstruction: The current µPC address is used to fetch a microinstruction from control memory.

Decode Microinstruction: The µIR stores and decodes the fetched microinstruction.

Execute Microoperations: The decoded microinstruction generates control signals to execute the corresponding microoperations in the CPU's datapath.

Update µPC: The address sequencing logic determines the next µPC address based on the current µIR and status signals.

Repeat: This cycle continues until the entire instruction is executed.

Q-09:Differentiate between vectored interrupt and non-vectored interrupt. [Pre_3:1_2021]

Answer:

Feature	Vectored Interrupts	Non-Vectored Interrupts
ISR Address	Provided directly by device	Determined by polling
Response Time	Faster	Slower
Efficiency	More efficient for many sources	Less efficient for many sources
Implementation	More complex hardware	Simpler hardware
Suitability	Systems with multiple interrupt sources	Systems with few interrupt sources

Q-10:Discuss in brief how the data transfer takes place between CPU and a peripheral. [Pre_3:1_2021]

Answer:

Here's a brief explanation of how the process takes place:

1. Identifying the Peripheral: The CPU needs to identify the specific peripheral device it wants to communicate with. Each peripheral is assigned a unique address or identifier that the CPU uses to establish a connection.

2. Preparing Data: The CPU prepares the data that needs to be transferred to or from the peripheral. This could involve loading data into specific registers or memory locations.

3. Issuing I/O Instructions: The CPU executes I/O instructions to initiate the data transfer. These instructions are specific to the peripheral and are provided by the peripheral's driver or software interface.

4. Communication via Data Bus: The CPU and peripheral communicate through a data bus, which is a group of parallel conductors used to transmit data.

5. Timing and Control Signals: Alongside the data bus, additional control signals are used to synchronize and manage the transfer.

6. Data Transfer: During the data transfer phase, the CPU either reads data from the peripheral or writes data to it. This happens as the data bits are transmitted over the data bus in parallel or serial form, depending on the interface and protocol used by the peripheral.

7. Interrupts and Status Checking: After the data transfer, the CPU may check for any status or error signals provided by the peripheral to ensure the successful completion of the operation. Interrupt signals can also be used, allowing the peripheral to interrupt the CPU when specific conditions are met.

Q-11:What do you mean by logical and physical address space? Show the memory hierarchy of a computer system. [Pre_3:1_2021]

Answer:

Logical Address Space: The logical address space refers to the addresses that an application program uses. It represents the view of the memory that a process has.

These addresses offer:

- Simplicity: Programs don't need to worry about the physical layout of memory or its limitations.
- Flexibility: Logical addresses can be manipulated easily, facilitating features like virtual memory and dynamic memory allocation.
- Protection: Different programs can have their own distinct address spaces, preventing them from interfering with each other's memory.

Physical Address Space: The physical address space refers to the actual addresses of the physical memory hardware. It represents the physical locations where data is stored in the RAM modules or other storage devices.

Physical addresses offer:

- Direct Access: Physical addresses allow the CPU to directly access specific memory locations for faster data transfers.
- Hardware Efficiency: Memory chips and controllers operate efficiently using hardware-aligned addresses.
- Real-World Limitations: Physical addresses are constrained by the actual size and capabilities of the available memory hardware.

Memory Hierarchy: The memory hierarchy in a computer system refers to the arrangement of different storage levels based on their speed, capacity, and cost. It consists of multiple levels, each with varying characteristics.

- Registers: Registers are the fastest memory units and are located directly in the CPU. They are used to store small amounts of data that need to be accessed quickly by the CPU.
- Cache Memory: It stores frequently accessed instructions and data, reducing the memory access time and improving overall system performance.
- Main Memory (RAM): Main memory, also known as random-access memory (RAM), is the primary storage where data and instructions are stored during program execution.
- Secondary Storage: Secondary storage devices, such as hard disk drives (HDDs) and solid-state drives (SSDs), have larger storage capacities than RAM but slower access speeds
- Tertiary Storage: They are commonly used for backup and archiving purposes.

Q-12:Mention some reasons for using virtual memory. Briefly explain different types of cache mapping. [Pre_3:1_2021]

Answer:

Reasons for Using Virtual Memory:

- Increased Program Size: Allows programs to exceed the physical memory size by storing unused portions on secondary storage and swapping them in/out as needed.
- Multiple Processes: Facilitates multitasking by providing each process with its own virtual address space, preventing memory interference and enabling memory protection.
- Efficient Memory Utilization: Reduces fragmentation by dynamically allocating memory to processes based on their current needs, maximizing available memory usage.
- Simplified Programming: Abstracts the physical memory layout from programmers, allowing them to focus on program logic without worrying about physical address limitations.
- Security: Helps isolate processes from each other, preventing malicious code from accessing other programs' memory and enhancing system security.

Different Types of Cache Mapping:

- Direct Mapping: Simplest method, each memory block maps to a single cache line. Efficient but prone to conflicts and thrashing when multiple blocks compete for the same line.
- Set-Associative Mapping: Divides cache into sets, each containing multiple lines. A memory block can map to any line within its designated set, reducing conflicts and improving hit rates compared to direct mapping.
- Fully-Associative Mapping: Most flexible approach, a block can map to any cache line. Offers highest hit rates but requires complex hardware and longer access times due to searching all lines.
- Victim Caching: Variant of set-associative mapping where an evicted block can be temporarily stored in a separate victim cache instead of being sent back to main memory. Reduces main memory access for potential reloads.

Q-13:Define Machine Cycle and Instruction Cycle. [Pre_3:1_2021]

Answer:

Machine Cycle: A machine cycle, also known as an instruction cycle or an execution cycle, refers to the basic operational steps performed by a computer's central processing unit (CPU) to execute a single instruction. It consists of a series of sub-cycles or phases that are repeated for each instruction being executed. The typical phases of a machine cycle include:

1. Fetch: During the fetch phase, the CPU fetches the next instruction from memory to be executed. The program counter (PC) is used to determine the memory address of the instruction.
2. Decode: In the decode phase, the fetched instruction is decoded by the CPU's instruction decoder. The decoder identifies the operation to be performed and the operands involved.
3. Execute: In the execute phase, the CPU carries out the operation specified by the decoded instruction. This may involve calculations, data manipulation, or memory access.
4. Store: Finally, in the store phase, the result of the executed instruction is stored in memory or a register, depending on the specific instruction and its requirements.

Instruction Cycle: The instruction cycle, also known as the fetch-decode-execute cycle, is a specific type of machine cycle that represents the sequence of steps required to execute a single instruction. It includes the fetch, decode, and execute phases mentioned earlier. The instruction cycle is repeated for each instruction in a program.

It includes additional steps like:

1. Fetching operands: Retrieving data needed for the instruction from registers or memory.
2. Storing results: Writing back the results of the operation to registers or memory.

Q-14:Explain the function of a DMA controller. [Pre_3:1_2021]

Answer:

DMA (Direct Memory Access) controller is a specialized hardware component that enables direct data transfer between external devices (e.g., hard drives, sound cards, network interfaces) and main memory, without extensive CPU involvement. It streamlines data movement and frees up the CPU for other tasks, leading to performance improvements.

Key Functions:

1. Initiating Data Transfers:
    - Receives requests for data transfers from devices or software.
2. Managing Data Flow:
    - Sets up data transfer parameters (source, destination, size, etc.).
3. Overseeing Transfer Progress:
    - Monitors transfer completion and status.
4. Signaling Completion:
    - Interrupts the CPU or triggers other events when a transfer finishes.

Q-15:Write down the Advantages and Disadvantages of DMA Data Transfer. [Pre_3:1_2021]

Answer:

Advantages:

- **Improved CPU Utilization:** Frees up CPU cycles for other tasks, enhancing overall system performance.
- **Faster Data Transfers:** Often more efficient than CPU-driven transfers, especially for large data blocks.
- **Reduced System Overhead:** Offloads data movement tasks from the CPU, reducing system overhead and potential bottlenecks.
- **Support for Concurrent Operations:** Enables the CPU to perform computations or handle other tasks while data transfers occur in the background.

Disadvantages:

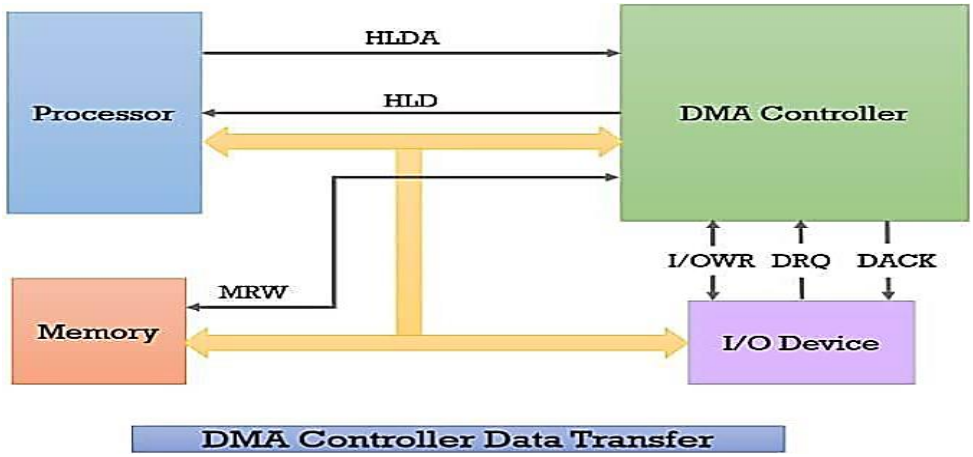
- **Hardware Complexity:** Requires additional hardware (DMA controller), increasing system cost and complexity.
- **Potential for Conflicts:** Improper configuration or sharing of resources can lead to conflicts between DMA transfers and other system operations.
- **Limited Visibility**: CPU may have reduced visibility into DMA transfers, potentially impacting debugging and troubleshooting.

Q-17:What is DMA? Draw the block diagram that shows how a DMA controller operates in a microcomputer system? . [Pre_3:1_2020]

Answer:

DMA (Direct Memory Access) is a technology that allows external devices to transfer data directly to and from main memory without requiring continuous CPU intervention. This frees up the CPU for other tasks, leading to improved system performance and efficiency.

Block Diagram Showing DMA Controller Operation:



Key Components and Process:

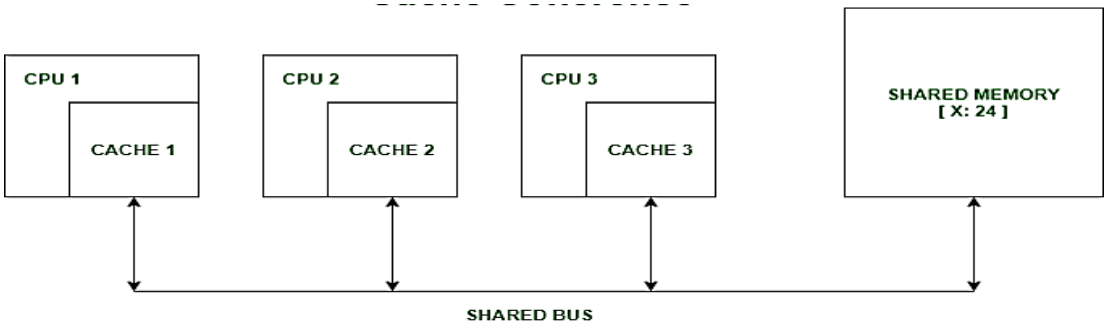
1. DMA Controller: Receives requests from devices or software.
2. CPU: Can perform other tasks while DMA transfers occur in the background.
3. I/O Devices: Sources or destinations of data transfers (e.g., hard drives, network cards, sound cards).
4. Memory: Stores data being transferred during DMA operations.
5. Bus: Shared communication pathway for data transfer between devices, memory, and the DMA controller.

Typical Steps for a DMA Transfer:

1. Request: Device or software signals a data transfer request to the DMA controller.
2. Setup: DMA controller receives the request and sets up transfer parameters (source, destination, size, etc.).
3. Bus Arbitration: DMA controller requests access to the bus from the CPU.
4. Transfer: Data is transferred directly between the device and memory, controlled by the DMA controller.
5. Completion: DMA controller signals transfer completion to the CPU or triggers other events.
6. CPU Intervention: CPU may intermittently check transfer status or handle other tasks during DMA operations.

Q-18:Briefly describe the Cache coherence problem. [Pre_3:1_2020]

Cache coherence problem refers to the challenge of maintaining consistency and synchronization of data stored in multiple caches in a multiprocessor system. When multiple processors or cores have their own local caches, they can each have a copy of the same data item in their respective caches. The cache coherence problem arises when one processor modifies its copy of the data, while other processors still hold their own copies, which may lead to inconsistencies.



Suppose there are three processors, each having cache. Suppose the following scenario:-

- Processor 1 read X : obtains 24 from the memory and caches it.
- Processor 2 read X : obtains 24 from memory and caches it.
- Again, processor 1 writes as X : 64, Its locally cached copy is updated. Now, processor 3 reads X, what value should it get?
- Memory and processor 2 thinks it is 24 and processor 1 thinks it is 64.

Q-20:What is an Interrupt? Describe Bus Interfacing with a processor. [Pre_3:1_2020]

Answer:

Interrupt: An interrupt is a signal that temporarily suspends the normal execution of a program and redirects the control to a specific interrupt service routine (ISR).

Bus interfacing with a processor refers to the connection and communication between the processor and other devices or components using a shared bus. The processor communicates with other devices using control, address, and data signals over the bus. When an interrupt occurs, the device generating the interrupt sends a signal to the processor through the bus, indicating the need for attention. The processor receives this interrupt signal and pauses its current execution to handle the interrupt request. It then performs the necessary actions based on the type of interrupt received.

Q-21:Describe Flynn's Taxonomy. [Pre_3:1_2020]

Answer:

Flynn's Taxonomy is a classification system used to categorize computer architectures based on the number of instruction streams and data streams that can be processed concurrently. It was proposed by Michael J. Flynn in 1966.

Flynn's Taxonomy identifies four categories:

1. **Single Instruction, Single Data (SISD):** This is the traditional uniprocessor model, where a single instruction stream operates on a single data stream at any given time. It represents a serial or sequential processing model.
2. **Single Instruction, Multiple Data (SIMD**): In this category, a single instruction stream controls multiple processing units (or a single processing unit with multiple execution units) operating on different data elements simultaneously. It is commonly used in parallel computing for operations that can process large arrays or vectors of data in parallel.
3. **Multiple Instruction, Single Data (MISD**): This category is relatively rare and less commonly implemented. It involves multiple instruction streams executing on the same data stream. It can be seen as a theoretical model for fault tolerance or highly specialized applications.
4. **Multiple Instruction, Multiple Data (MIMD):** This category represents multiprocessor systems or distributed computing environments where multiple instruction streams can execute on multiple data streams concurrently. MIMD architectures are commonly used for multiprocessing, cluster computing, or distributed systems.

Q-22:Distinguish between Multiprocessor and Multicomputer. [Pre_3:1_2020]

Answer:

Multiprocessor	Multicomputer
It is a system with multiple processors that enables programs to be processed at the same time.	It is a collection of processors linked by a communication network that collaborate to solve a computation task.
It is easy to program.	It is complex to program.
It supports parallel computing.	It supports distributed computing.
It is easy and less expensive to develop.	It is complex and expensive to develop.
It is a type of dynamic network.	It is a type of static network.

SIMPLE VISUALIZATION

MD RAIHAN ALI_[63]
CSE_02
Faculty of Engineering & Technology
University of Dhaka (NITER)