Section-A1

1. (a) Define the following types of memories:
    - (i) **PROM (Programmable Read-Only Memory):** PROM is a type of ROM that can be **programmed once** by the user or manufacturer. Once programmed, the data stored in a PROM is permanent and **cannot be erased or modified**, making it a non-volatile memory.
    - (ii) **EPROM (Erasable Programmable Read-Only Memory):** EPROM is a type of ROM that can be **reprogrammed multiple times**. The data stored in an EPROM can be erased by exposing the chip to **ultraviolet (UV) light**, after which new data can be written to it.

(b) Differentiate between isolated and memory mapped I/O.

- **Isolated I/O:**
    - Uses **separate address spaces** for memory and input/output (I/O) devices.
    - Requires **specific I/O instructions** (like IN and OUT) to transfer data between the CPU and I/O devices.
    - The address lines used for I/O are distinct from those for memory, preventing address conflicts.
- **Memory-Mapped I/O:**
    - Uses the **same address space** for both memory and I/O devices.
    - I/O devices are treated as if they are memory locations, allowing the CPU to use **standard memory access instructions** (like LOAD and STORE) for I/O operations.
    - The CPU uses the same address lines and control signals for both memory and I/O.

(c) List any three characteristics of a GPU.

- **Highly Parallel Architecture:** GPUs are designed with a massive number of specialized processing cores that enable them to perform a multitude of computations simultaneously, making them ideal for parallelizable tasks.
- **High Memory Bandwidth:** They possess very high memory bandwidth, allowing for rapid transfer of large amounts of data, which is crucial for tasks like graphics rendering and scientific simulations.
- **Specialized for Graphics and Parallel Computing:** While initially developed for rendering computer graphics, their architecture makes them exceptionally efficient for general-purpose parallel computing (GPGPU), widely used in fields like artificial intelligence and scientific research.

(d) Show the 8-bit representation of -14 in

- **Binary representation of +14 (8-bit):** 00001110_2
- (i) **Signed magnitude representation:**
  - The leftmost bit is the sign bit (1 for negative). The remaining 7 bits represent the magnitude.
  - For -14: 10001110_2
- (ii) **Signed-1’s complement representation:**
  - First, find the binary of +14: 00001110_2.
  - Then, invert all the bits: 11110001_2.
- (iii) **Signed-2’s complement representation:**
  - First, find the 1's complement of -14: 11110001_2.
  - Then, add 1 to the 1's complement: $11110001_2 + 1_2 = 11110010_2$.

(e) Describe the functions of the following registers:

- (i) **AR (Address Register):** The Address Register holds the **memory address** of the data or instruction that the CPU needs to access (read from or write to). It acts as the interface between the CPU and the memory address bus.
- (ii) **AC (Accumulator):** The Accumulator is a general-purpose register that temporarily stores **intermediate results of arithmetic and logical operations**. It is often an implicit operand or destination register for many CPU instructions.
- (iii) **DR (Data Register):** The Data Register temporarily holds **data being transferred** between the CPU and memory or I/O devices. It buffers data read from memory before it's processed by the CPU, and data written to memory before it's stored.

(f) Given the Boolean function $F = A'B + ABC'$. Derive the algebraic expression for F'. Also, show that F.F' = 0.

- **Derive F':**
  - Given $F = A'B + ABC'$
  - Using De Morgan's Theorem $(X + Y)' = X'Y'$:
    - $F' = (A'B + ABC')'$
    - $F' = (A'B)' \\cdot (ABC')'$
  - Using De Morgan's Theorem $(XY)' = X' + Y'$:
    - $F' = ((A')' + B') \\cdot (A' + B' + (C')')$
    - $F' = (A + B') \\cdot (A' + B' + C)$
- **Show that F.F' = 0:**
  - Substitute F and F':
    - $F \\cdot F' = (A'B + ABC') \\cdot ((A + B')(A' + B' + C))$

- First, expand the second part of F':
  - $(A + B')(A' + B' + C) = A(A' + B' + C) + B'(A' + B' + C)$
  - $= AA' + AB' + AC + A'B' + B'B' + B'C$
  - Since $AA' = 0$ and $B'B' = B'$:
  - $= 0 + AB' + AC + A'B' + B' + B'C$
  - $= B' + AB' + A'B' + AC + B'C$
  - Factor out B': $B'(1 + A + A' + C) + AC$
  - Since $(1 + X) = 1$: $B'(1) + AC = B' + AC$
- Now substitute this back into $F \\cdot F'$:
  - $F \\cdot F' = (A'B + ABC') \\cdot (B' + AC)$
  - Expand the expression:
    - $F \\cdot F' = A'B(B' + AC) + ABC'(B' + AC)$
    - $F \\cdot F' = A'BB' + A'BAC + ABC'B' + ABC'AC$
  - Since $BB' = 0$ and $C'C = 0$:
    - $F \\cdot F' = A'(0) + A'BAC + A(0)C' + AABC'C$
    - $F \\cdot F' = 0 + A'ABC + 0 + AABC'(0)$
    - Since $A'A = 0$:
    - $F \\cdot F' = 0 + 0 \\cdot BC + 0 + A B \\cdot 0$
    - $F \\cdot F' = 0$
- Therefore, $F \\cdot F' = 0$ is shown.

(g) Write the micro-operations for the following instructions:

- (i) **CMA (Complement Accumulator):**

- $AC \\leftarrow \\overline{AC}$ (The contents of the Accumulator are complemented bit by bit).

- (ii) **SNA (Skip Next instruction if Accumulator is Negative):**
  - If $AC\_15 = 1$ then $PC \\leftarrow PC + 1$ (Assuming the 15th bit (MSB) of AC is the sign bit, and 1 indicates a negative number).

(h) Give the logic diagram and truth table of a 2-to-4 line decoder using NAND gates only.

- **Truth Table for 2-to-4 Line Decoder (Inputs A1, A0; Outputs D0, D1, D2, D3):**

A1	A0	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- **Logic Diagram using NAND gates:**
  - A 2-to-4 line decoder generates a high output for one of its four output lines based on the 2-bit binary input.
  - The outputs in SOP form are:
    - $D\_0 = A1'A0'$
    - $D\_1 = A1'A0$
    - $D\_2 = A1A0'$
    - $D\_3 = A1A0$
  - To implement using only NAND gates:

- First, generate the complements of the inputs:
    - $A1' = \\text{NAND}(A1, A1)$
    - $A0' = \\text{NAND}(A0, A0)$
- Then, use 2-input NAND gates for each output, applying De Morgan's theorem $(XY)' = X' + Y'$, which effectively means $(X'Y')'$ for XY. For direct products $X \\cdot Y$, you can use a NAND gate followed by an inverter. However, for decoders where one output is HIGH, it's common to use the following structure:
    - $D\_0 = (A1' \\cdot A0')'$ (This is a NAND gate with inputs $A1'$ and $A0'$)
    - $D\_1 = (A1' \\cdot A0)'$ (This is a NAND gate with inputs $A1'$ and $A0$)
    - $D\_2 = (A1 \\cdot A0')'$ (This is a NAND gate with inputs $A1$ and $A0'$)
    - $D\_3 = (A1 \\cdot A0)'$ (This is a NAND gate with inputs $A1$ and $A0$)

(i) Represent the decimal numbers 184 and 576 in binary code and BCD form.

- **Decimal 184:**
    - **Binary Code:** To convert 184 to binary, repeatedly divide by 2 and collect the remainders:
        - $184 \\div 2 = 92$ R 0
        - $92 \\div 2 = 46$ R 0
        - $46 \\div 2 = 23$ R 0
        - $23 \\div 2 = 11$ R 1

        - $11 \\div 2 = 5$ R 1
        - $5 \\div 2 = 2$ R 1
        - $2 \\div 2 = 1$ R 0
        - $1 \\div 2 = 0$ R 1
        - Reading remainders from bottom up: 10111000_2
    - **BCD (Binary Coded Decimal) Form:** Each decimal digit is represented by its 4-bit binary equivalent.
        - $1 \\rightarrow 0001$
        - $8 \\rightarrow 1000$
        - $4 \\rightarrow 0100$
        - BCD for 184: $0001 \\ 1000 \\ 0100$

- **Decimal 576:**
    - **Binary Code:**
        - $576 \\div 2 = 288$ R 0
        - $288 \\div 2 = 144$ R 0
        - $144 \\div 2 = 72$ R 0
        - $72 \\div 2 = 36$ R 0
        - $36 \\div 2 = 18$ R 0
        - $18 \\div 2 = 9$ R 0
        - $9 \\div 2 = 4$ R 1
        - $4 \\div 2 = 2$ R 0
        - $2 \\div 2 = 1$ R 0
        - $1 \\div 2 = 0$ R 1

    - Reading remainders from bottom up: 1001000000_2
  - **BCD Form:**
    - $5 \\rightarrow 0101$
    - $7 \\rightarrow 0111$
    - $6 \\rightarrow 0110$
    - BCD for 576: $0101 \\ 0111 \\ 0110$

Section-B2

2. (a) Determine the number of $1024 \\times 16$ memory chips required to achieve the following memory capacities:

- (i) **$8192 \\text{ K} \\times 16$**
  - Total words needed: $8192 \\text{ K} = 8192 \\times 1024$ words.
  - Words per chip: 1024 words.
  - Number of words chips needed = $(8192 \\times 1024) / 1024 = 8192$ chips.
  - Bits per word needed: 16 bits.
  - Bits per word per chip: 16 bits.
  - Number of bits chips needed = $16/16 = 1$ chip.
  - Total chips required = $8192 \\times 1 = \\mathbf{8192 \\text{ chips}}$.
- (ii) **$4096 \\times 32$**
  - Total words needed: 4096 words.
  - Words per chip: 1024 words.

- Number of words chips (horizontal expansion) = $4096/1024 = 4$ chips.
- Bits per word needed: 32 bits.
- Bits per word per chip: 16 bits.
- Number of bits chips (vertical expansion) = $32/16 = 2$ chips.
- Total chips required = $4 \\times 2 = \\mathbf{8 \\text{ chips}}$.

(b) Explain why each of the following register transfer language statements cannot be directly executed in a basic computer. Also specify the correct sequence of micro-operations required to perform these operations:

- (i) **IR ← M[PC]**
  - **Reason for not direct execution:** In a basic computer, the Program Counter (PC) typically holds the address of the next instruction. To fetch an instruction from memory (M), its address must first be placed into the Memory Address Register (AR), which is the only register connected to the memory's address lines. The Instruction Register (IR) then receives data from the Memory Data Register (DR), which is connected to the memory's data lines. A direct transfer from memory using PC as an address directly into IR is not architecturally supported without intermediate steps.
  - **Correct sequence of micro-operations:**
    - $AR \\leftarrow PC$ (The address from PC is transferred to the Address Register).
    - $IR \\leftarrow M[AR]$ (The instruction at the memory location specified by AR is fetched and transferred to the Instruction Register).
- (ii) **AC ← AC + TR**

- **Reason for not direct execution:** Most basic computer architectures have an Arithmetic Logic Unit (ALU) that performs operations. The ALU usually has specific input registers, commonly the Accumulator (AC) and the Data Register (DR). A general-purpose temporary register (TR) might not be directly connected to an ALU input or capable of participating directly in an arithmetic operation with the AC without an intermediate transfer.
- **Correct sequence of micro-operations:**
  - $DR \\leftarrow TR$ (The content of the temporary register TR is transferred to the Data Register, making it available as an ALU operand).
  - $AC \\leftarrow AC + DR$ (The ALU performs the addition of the content of AC and DR, storing the result back in AC).

(c) Design a combinational circuit with three binary inputs a, b and c and three binary outputs x, y and z. When the binary input has an even number of 1's, then the output is one more than the input. When the binary input has an odd number of 1's, then the output is one less than the input. The output remains the same if the input is zero.

- **Truth Table:**
  - The input (a,b,c) represents a binary number N_in. The output (x,y,z) represents N_out.
  - Special condition: if $N\_in = 000\_2$, then $N\_out = 000\_2$.
  - Otherwise:
    - If count of 1s in N_in is even, $N\_out = N\_in + 1$.
    - If count of 1s in N_in is odd, $N\_out = N\_in - 1$.

a	b	c	Count of 1s	Even/Odd	N_in (Dec)	Rule	N_out (Dec)	x	y	z
0	0	0	0	Even	0	Special	0	0	0	0
0	0	1	1	Odd	1	$N\_in - 1$	0	0	0	0
0	1	0	1	Odd	2	$N\_in - 1$	1	0	0	1
0	1	1	2	Even	3	$N\_in + 1$	4	1	0	0
1	0	0	1	Odd	4	$N\_in - 1$	3	0	1	1
1	0	1	2	Even	5	$N\_in + 1$	6	1	1	0
1	1	0	2	Even	6	$N\_in + 1$	7	1	1	1
1	1	1	3	Odd	7	$N\_in - 1$	6	1	1	0

- **Boolean Expressions (from Truth Table, simplified using K-Maps or algebra):**
  - **For x (Output MSB):**
    - $x = a'bc + ab'c + abc' + abc$
    - This simplifies to: $x = (a'bc + abc) + (ab'c + abc')$
    - $x = bc(a' + a) + a(b'c + bc')$
    - $x = bc + a(b'c + bc')$
    - $x = bc + ab'c + abc'$

- **For y (Output Middle Bit):**
  - $y = ab'c' + ab'c + abc' + abc$ (from minterms 4, 5, 6, 7)
  - $y = a(b'c' + b'c + bc' + bc)$ (incorrect terms from table, $y = 1$ for $ab'c'$, $ab'c$, abc', abc)
  - From the truth table: $y = 1$ for $N\_in = 4,5,6,7$.
  - $y = ab'c' + ab'c + abc' + abc$
  - This simplifies to $y = a$.
  - Let's recheck the truth table:
    - For $N\_in = 4(100)$, $N\_out = 3(011)$, so $y = 1$.
    - For $N\_in = 5(101)$, $N\_out = 6(110)$, so $y = 1$.
    - For $N\_in = 6(110)$, $N\_out = 7(111)$, so $y = 1$.
    - For $N\_in = 7(111)$, $N\_out = 6(110)$, so $y = 1$.
    - For $N\_in = 0,1,2,3$: $y = 0$.
  - So, $y = a$. This is remarkably simple.
- **For z (Output LSB):**
  - $z = a'bc' + ab'c' + abc + abc'$ (from minterms 2, 4, 6, 7)
  - $z = (a'bc' + abc') + (ab'c' + abc)$
  - $z = bc'(a' + a) + a(b'c' + bc)$
  - $z = bc' + a(b'c' + bc)$
  - $z = bc' + ab'c' + abc$

- **Simplified Boolean Expressions:**
  - $x = bc + ab'c + abc'$
  - $y = a$

- $z = bc' + ab'c' + abc$

3. (a) Draw the block diagram of a 4-to-1 line multiplexer and explain its operation by means of a function table.

- **Block Diagram of a 4-to-1 Multiplexer:**
  - A 4-to-1 multiplexer has:
    - Four data input lines: I_0, I_1, I_2, I_3
    - Two select lines: S_1, S_0 (these control which input is selected)
    - One output line: Y
    - Often, an Enable (E) input (or Strobe) which, when low, disables the MUX output.
- **Function Table (Truth Table):**
  - This table shows how the select lines determine which input data is routed to the output.

Enable (E)	S1	S0	Output Y
0	X	X	0 (or Z)
1	0	0	I_0
1	0	1	I_1
1	1	0	I_2
1	1	1	I_3

  - (X denotes a "Don't Care" condition)
- **Explanation of Operation:**

- A 4-to-1 line multiplexer (MUX) acts as a digital switch that selects one of its multiple input data lines and forwards it to a single output line.
- The selection is controlled by a set of binary select lines. For a 4-to-1 MUX, two select lines (S_1 and S_0) are needed to represent the four possible input choices ($2^2 = 4$).
- When the **Enable (E) input is low (0)**, the multiplexer is disabled, and its output (Y) is typically forced to a logic low (0) state or a high-impedance (Z) state, regardless of the other inputs.
- When the **Enable (E) input is high (1)**, the multiplexer is active and operates normally:
  - If the select lines S_1S_0 are 00_2, the data present on input I_0 is routed to the output Y.
  - If S_1S_0 are 01_2, data from I_1 is routed to Y.
  - If S_1S_0 are 10_2, data from I_2 is routed to Y.
  - If S_1S_0 are 11_2, data from I_3 is routed to Y.
- In essence, the multiplexer allows sharing a single output channel among multiple data sources based on the control signals.

(b) Define the Interrupt Cycle. Illustrate the process with a flowchart depicting the sequence of operations involved in the interrupt cycle.

- **Definition of Interrupt Cycle:**
  - The Interrupt Cycle is a hardware-initiated process that allows external or internal devices to temporarily suspend the normal execution of a program by the CPU.
  - Its primary function is to enable the CPU to respond to asynchronous events (interrupts) by saving the current state of

the running program (such as the Program Counter) and then transferring control to a special program called an Interrupt Service Routine (ISR).

- This cycle ensures that urgent tasks or events, like I/O completion or error conditions, are handled promptly without continuous CPU polling, thereby improving system efficiency and responsiveness.

- **Flowchart Depicting the Sequence of Operations in the Interrupt Cycle:**
  - [A flowchart would typically be drawn here. Since I cannot create diagrams, I will describe the sequence in a structured format.]
  - **Begin Fetch Cycle (or End of Execute Cycle):**
    - Is there an **Interrupt Request (INT)** active?
    - Are **Interrupts Enabled (IEN)**?
  - **IF (INT = 1 AND IEN = 1) THEN (Interrupt Occurs):**
    - **Disable Further Interrupts:** Set **IEN** ← **0** (to prevent nested interrupts during critical saving).
    - **Save Program Counter (PC):**
      - A fixed memory location (e.g., address 0 or a dedicated stack) is prepared to store the current PC value.
      - Often, the Program Counter's value is transferred to a temporary register: $TR \\leftarrow PC$.
      - Then, this value is stored in memory: $M[0] \\leftarrow TR$ (or $M[AR] \\leftarrow PC$ if AR holds the save address).

        - **Load Interrupt Service Routine (ISR) Address into PC:**
            - The PC is loaded with the starting address of the Interrupt Service Routine (e.g., $PC \\leftarrow 1$ or another pre-defined ISR entry point). This directs the CPU to execute the ISR next.
        - **Return to Fetch Cycle:** The CPU begins fetching the first instruction of the ISR.
    - **ELSE (No Interrupt or Interrupts Disabled):**
        - Proceed with the normal Fetch-Decode-Execute cycle of the current program.
    - **(Later, at the end of the ISR):**
        - An instruction like **Return From Interrupt (RTI)** will:
            - Restore the saved PC from memory: $PC \\leftarrow M[0]$.
            - Re-enable interrupts: $IEN \\leftarrow 1$.
            - Resume execution of the interrupted program.

(c) Given the Boolean function $F = xy'z + x'y'z + w'xy + wx'y + wxy$

- (i) **List the truth table of the given function.**
    - The function has four variables: w, x, y, z. Let's determine the minterms (where F = 1).
    - $xy'z$: This term means $x = 1, y = 0, z = 1$. w can be 0 or 1.
        - $0101\_2 = m\_5$
        - $1101\_2 = m\_13$
    - $x'y'z$: This term means $x = 0, y = 0, z = 1$. w can be 0 or 1.
        - $0001\_2 = m\_1$

- $1001\_2 = m\_9$

- $w'xy$: This term means $w = 0, x = 1, y = 1$. z can be 0 or 1.
  - $0110\_2 = m\_6$
  - $0111\_2 = m\_7$
- $wx'y$: This term means $w = 1, x = 0, y = 1$. z can be 0 or 1.
  - $1010\_2 = m\_10$
  - $1011\_2 = m\_11$
- wxy: This term means $w = 1, x = 1, y = 1$. z can be 0 or 1.
  - $1110\_2 = m\_14$
  - $1111\_2 = m\_15$
- So, $F(w, x, y, z) = \\Sigma(1, 5, 6, 7, 9, 10, 11, 13, 14, 15)$.

w	x	y	z	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1

w	x	y	z	F
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

- (ii) **Draw the logic diagram using the original Boolean expression.**
    - The expression is $F = xy'z + x'y'z + w'xy + wx'y + wxy$.
    - This is a Sum-of-Products (SOP) form.
    - **Diagram Description:**
        - **Inputs:** w, x, y, z, and their complements (w', x', y', z'). You would need inverters for w', x', y'.
        - **AND Gates (one for each product term):**
            - AND Gate 1: Inputs x, y', z. Output: $xy'z$
            - AND Gate 2: Inputs x', y', z. Output: $x'y'z$
            - AND Gate 3: Inputs w', x, y. Output: $w'xy$
            - AND Gate 4: Inputs w, x', y. Output: $wx'y$
            - AND Gate 5: Inputs w, x, y. Output: wxy
        - **OR Gate:**
            - A single 5-input OR gate whose inputs are the outputs from the five AND gates.

- The output of this OR gate is F.

- (iii) **Simplify the function using Boolean algebra.**
  - $F = xy'z + x'y'z + w'xy + wx'y + wxy$
  - **Step 1: Group and simplify terms.**
    - Group 1: $(xy'z + x'y'z)$
      - Factor out $y'z$: $y'z(x + x')$
      - Since $x + x' = 1$: $y'z(1) = y'z$
    - Group 2: $(w'xy + wx'y + wxy)$
      - Combine $w'xy$ and wxy: $xy(w' + w) = xy(1) = xy$
      - So, Group 2 becomes $xy + wx'y$
      - Factor out y: $y(x + wx')$
      - Using the property $A + A'B = A + B$: $y(x + x'w) = y(x + w)$
      - Distribute y: $xy + wy$
  - **Step 2: Combine the simplified groups.**
    - $F = y'z + xy + wy$
  - **Simplified Boolean Expression:** $F = y'z + xy + wy$

4. (a) What mechanism can be used to detect overflow condition while performing arithmetic computations on binary numbers? Explain the same with the help of an example.

- **Mechanism to Detect Overflow:**
  - Overflow occurs in signed binary arithmetic when the result of an operation exceeds the representable range for the given

number of bits. For example, in an 8-bit signed 2's complement system, numbers from -128 to +127 can be represented.

- The most common mechanism to detect overflow during **signed addition** is to examine the **carry-in (C_in)** and **carry-out (C_out)** of the **most significant bit (MSB)** position.
- **Overflow occurs if, and only if, the carry-in to the MSB is different from the carry-out from the MSB.** That is, $C\_{in\_MSB} \\oplus C\_{out\_MSB} = 1$.

- **Explanation with Example (8-bit Signed 2's Complement):**
  - **Example 1: Positive Overflow (+70) + (+80)**
    - $+70\_10 = 01000110\_2$
    - $+80\_10 = 01010000\_2$
    - **Addition:**

      01000110 (+70)
      + 01010000 (+80)

      10010110 (Result)

    - **Detection:**
      - Looking at the MSB (leftmost bit, position 7):
        - Carry-in to MSB (from bit 6): In adding $0 + 1$ (bit 6) and previous carry, a carry of 1 is generated and passed to MSB. $C\_in\_MSB = 1$.
        - Carry-out from MSB (from bit 7): Adding $0 + 0$ (bit 7) and carry-in 1 (from bit 6), results in $1 + 0 + 0 = 1$ with no carry-out. $C\_out\_MSB = 0$.

- Since $C\_in\_MSB = 1$ and $C\_out\_MSB = 0$, they are different ($1 \\neq 0$). Therefore, **overflow has occurred**.
- The result 10010110_2 actually represents -106_10 in 2's complement, which is incorrect as the sum of two positive numbers should be positive.

- **Example 2: Negative Overflow (-70) + (-80)**
  - **Conversion to 2's complement (from earlier question):**
    - $-70\_10 = 10111010\_2$
    - $-80\_10 = 10110000\_2$
  - **Addition:**

    Carries: 11111000 (This is a sample of carries propagated to top)
    10111010 (-70)
    + 10110000 (-80)

    (Carry-out) 1 01101010 (Result, ignoring final carry out of 9th bit)
  - **Detection:**
    - Looking at the MSB (position 7):
      - Carry-in to MSB (from bit 6): The sum of bits at position 6 ($0 + 0$) plus carry from position 5 (1) results in 1, no carry to MSB. Wait, previous calculation for carries was:
        - Bit 0: $0 + 0 = 0$, Carry=0
        - Bit 1: $1 + 0 = 1$, Carry=0

        - Bit 2: $0 + 0 = 0$, Carry=0
        - Bit 3: $1 + 1 = 0$, Carry=1 (to bit 4)
        - Bit 4: $1 + 0 + 1 = 0$, Carry=1 (to bit 5)
        - Bit 5: $1 + 1 + 1 = 1$, Carry=1 (to bit 6)
        - Bit 6: $0 + 1 + 1 = 0$, Carry=1 (to bit 7, MSB) $\\rightarrow C\_{in\_MSB} = 1$.
    - Carry-out from MSB (from bit 7): Adding $1 + 1$ (bit 7) and carry-in 1 (from bit 6), results in $1 + 1 + 1 = 11\_2$ (sum=1, carry-out=1). $\\rightarrow C\_{out\_MSB} = 1$.

- Since $C\_in\_MSB = 1$ and $C\_out\_MSB = 1$, they are the same ($1 = 1$). According to the rule, **no overflow has occurred**.
- However, the sum of two negative numbers (MSB=1, MSB=1) resulted in a positive number (MSB=0, $01101010\_2 = +106\_10$). This indicates an overflow. This example demonstrates how important it is to be precise about C_in_MSB and C_out_MSB.
- **Re-evaluating Example 2's carries with more care:**
    - $-70 = 10111010\_2$
    - $-80 = 10110000\_2$
    - *Carries for each position (right to left):*
      Position: 7 6 5 4 3 2 1 0
      Number 1: 1 0 1 1 1 0 1 0
      Number 2: 1 0 1 1 0 0 0 0

      Sum: 0 1 1 0 1 0 1 0

Propagated Carries (from right to left, to next position):
C_0 (out from pos 0): 0 ($0+0=0$)
C_1 (out from pos 1): 0 ($1+0=1$)
C_2 (out from pos 2): 0 ($0+0=0$)
C_3 (out from pos 3): 1 ($1+0=1$) -> Input to pos 4 (bit 3) is 1.
C_4 (out from pos 4): 1 ($1+1+1=11$, so sum 1, carry 1) -> Input to pos 5 (bit 4) is 1.
C_5 (out from pos 5): 1 ($1+1+1=11$, so sum 1, carry 1) -> Input to pos 6 (bit 5) is 1.
C_6 (out from pos 6): 1 ($0+0+1=1$, so sum 1, carry 0) -> Input to pos 7 (MSB) is 0.
C_7 (out from pos 7, final carry): 1 ($1+1+0 = 10$, so sum 0, carry 1) -> Carry-out from MSB is 1.

- C_in_MSB (carry into bit 7) = 0 (from bit 6 calculation: $0 + 0$ plus incoming $C\_5 = 1$ gave sum 1, no carry to bit 7).
- C_out_MSB (carry out from bit 7) = 1 (from bit 7 calculation: $1 + 1$ plus incoming $C\_6 = 0$ gave sum 0, carry 1).
- Since $C\_in\_MSB = 0$ and $C\_out\_MSB = 1$, they are different ($0 \\neq 1$). Therefore, **overflow has occurred**. The rule holds.

(b) Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

- **Space-Time Diagram for a 6-segment Pipeline (8 Tasks):**
  - **Segments (S):** S1, S2, S3, S4, S5, S6 (representing different stages of an instruction, e.g., Fetch, Decode, Execute, etc.)
  - **Tasks (T):** T1, T2, T3, T4, T5, T6, T7, T8

- Each row represents a segment, and each column represents a clock cycle.
- 'Ti' indicates that Task 'i' is occupying that segment during that clock cycle.

Time (Clock Cycle)	S1 (Fetch)	S2 (Decode)	S3 (Operand Fetch)	S4 (Execute)	S5 (Write Back)	S6 (Commit)
1	T1					
2	T2	T1				
3	T3	T2	T1			
4	T4	T3	T2	T1		
5	T5	T4	T3	T2	T1	
6	T6	T5	T4	T3	T2	T1
7	T7	T6	T5	T4	T3	T2
8	T8	T7	T6	T5	T4	T3
9		T8	T7	T6	T5	T4
10			T8	T7	T6	T5
11				T8	T7	T6
12					T8	T7
13						T8

- **Explanation:**
  - The diagram illustrates the concurrent execution of multiple tasks within the pipeline.
  - In the first clock cycle, Task T1 enters Segment S1.

- In the second cycle, T2 enters S1 while T1 moves to S2. This parallel movement continues.
- By the 6th clock cycle, the pipeline is fully loaded, with all six segments occupied by different tasks (T1 in S6, T2 in S5, ..., T6 in S1).
- From the 6th clock cycle onwards, one task completes its execution in each subsequent clock cycle. For instance, T1 finishes at cycle 6, T2 at cycle 7, and so on.
- The total time taken to process 8 tasks in a 6-segment pipeline is $k + n - 1$ clock cycles, where k is the number of segments (6) and n is the number of tasks (8).
- Total cycles = $6 + 8 - 1 = 13$ clock cycles. This matches the diagram, with T8 completing at the end of the 13th cycle.

(c) The content of the AC in the basic computer is A937 (all numbers are in hexadecimal) and the initial value of E is 1. Determine the contents of AC, E, PC, AR and IR in hexadecimal after the execution of the CMA instruction. The initial value of PC is hexadecimal 021 and hexadecimal code of CMA is 7200.

- **Initial State:**
  - AC (Accumulator) = A937 (Hex)
  - E (Extended Bit/Carry) = 1 (Binary)
  - PC (Program Counter) = 021 (Hex)
  - IR (Instruction Register) = (unknown, will hold the fetched instruction)
  - AR (Address Register) = (unknown, used during memory access)
- **Instruction to Execute:** CMA (Complement Accumulator)

- Hex code for CMA = 7200

- **Execution Process:**

a. **Fetch Cycle:**

- The content of **PC (021)** is transferred to the **AR**: $AR \\leftarrow 021\_{16}$.
- The instruction at memory location 021 is fetched into **IR**. We are given that this instruction is CMA, so: $IR \\leftarrow 7200\_{16}$.
- The **PC** is incremented to point to the next instruction: $PC \\leftarrow 021\_{16} + 1\_{16} = 022\_{16}$.

b. **Execute Cycle (CMA Instruction):**

- The CMA instruction's micro-operation is to complement the content of the Accumulator: $AC \\leftarrow \\overline{AC}$.
- Current AC (Hex): A937
- Convert AC to Binary (16-bit, typical for basic computer AC):
  - A = 1010
  - 9 = 1001
  - 3 = 0011
  - 7 = 0111
  - So, $AC = 1010 \\ 1001 \\ 0011 \\ 0111\_2$
- Complement all bits of AC:
  - $\\overline{AC} = 0101 \\ 0110 \\ 1100 \\ 1000\_2$
- Convert the complemented AC back to Hexadecimal:

- 0101 = 5
- 0110 = 6
- 1100 = C
- 1000 = 8
- New $AC = 56C8\_16$

  - The CMA instruction **does not affect the E register**. So, E remains its initial value.

- **Contents of Registers After Execution:**
  - **AC = 56C8 (Hex)**
  - **E = 1 (Binary)**
  - **PC = 022 (Hex)**
  - **AR = 021 (Hex)**
  - **IR = 7200 (Hex)**

---

5. (a) Assuming the three bit binary code for a register corresponds to the register number and the binary codes for the operations supported by the processor are listed in Table 1. Specify the 14-bit binary control words consisting of four fields SELA, SELB, SELD and OPR that must be applied to implement the following operations:

- **Control Word Format:** SELA (3 bits), SELB (3 bits), SELD (3 bits), OPR (5 bits) = Total 14 bits.
- **Register Encoding (assuming R0=000, R1=001, R2=010, R3=011, etc.):**
  - R1 = 001
  - R2 = 010

- R3 = 011

- **Table 1: Encoding of ALU operations:**
  - Transfer: 00000
  - OR: 01011
  - ADD: 10010
  - Complement: 10100

- (i) $\mathbf{R_1 \leftarrow R_2 + R_3}$
  - SELA (Source A): R2 = 010
  - SELB (Source B): R3 = 011
  - SELD (Destination D): R1 = 001
  - OPR (Operation): ADD = 10010
  - **14-bit Binary Control Word:** $010 \\ 011 \\ 001 \\ 10010\_2$

- (ii) $\mathbf{R_1 \leftarrow R_2 \text{ v } R_3}$ (OR operation)
  - SELA (Source A): R2 = 010
  - SELB (Source B): R3 = 011
  - SELD (Destination D): R1 = 001
  - OPR (Operation): OR = 01011
  - **14-bit Binary Control Word:** $010 \\ 011 \\ 001 \\ 01011\_2$

(b) An instruction is stored at location 100 with its address field at location 101. The address field has the value 500. PC has the value 100. The content of a processor register R_1 is 200. Evaluate the effective address (EA) if the addressing mode of the instruction is (All values are in decimal):

- **Given Information:**
  - Instruction is at memory location 100.

- Address field of the instruction is at memory location 101.
- Value in the address field (let's call it ADDR_VAL) = 500.
- Initial PC (Program Counter) value = 100.
- Content of Register R1 = 200.

- **Assumptions for PC in Relative Addressing:** When calculating relative addressing, the PC value used is typically the address of the *next* instruction to be fetched, i.e., after the current instruction (including its address field) has been fetched. If the instruction is at 100 and its address field is at 101, then the PC would advance to 102. So, $PC\_next = 102$.

- (i) **Direct Addressing:**
  - In direct addressing, the effective address is simply the value contained in the address field of the instruction.
  - $EA = \\text{ADDR\_VAL} = \\mathbf{500}$

- (ii) **Relative Addressing:**
  - In relative addressing, the effective address is calculated by adding the address field value to the current value of the Program Counter.
  - $EA = PC\_{next} + \\text{ADDR\_VAL} = 102 + 500 = \\mathbf{602}$

- (iii) **Indirect Addressing:**
  - In indirect addressing, the address field of the instruction holds the *address* of the effective address (the address where the actual operand's address is stored).
  - $EA = \\text{Memory}[\\text{ADDR\_VAL}] = \\text{Memory}[500]$

    - The effective address is the **content of memory location 500**. (The actual value is not provided, so it is symbolically M[500]).

- (iv) **Register Indirect Addressing:**
    - In register indirect addressing, the effective address is the value stored within a specified CPU register. The instruction would contain the register number (e.g., R1).
    - $EA = \\text{Content of R1} = \\mathbf{200}$
- (v) **Immediate Addressing:**
    - In immediate addressing, the address field of the instruction does not contain an address but *is* the actual operand (data) itself.
    - For this mode, the term "effective address" is often used to refer to the operand itself.
    - The operand value is the address field value.
    - Operand = $\\text{ADDR\_VAL} = \\mathbf{500}$

(c) Perform the arithmetic operations (+70) + (+80) and (–70) + (–80) in binary using signed-2's complement representation for negative numbers. Use eight bits to accommodate each number together with its sign.

- **8-bit Signed 2's Complement Range:** -128 to +127.
- **Binary and 2's Complement Representations (8-bit):**
    - $+70_{10} = 01000110_2$
    - $+80_{10} = 01010000_2$
    - -70_{10}:
        - $+70 = 01000110_2$
        - 1's complement: 10111001_2

        - 2's complement (add 1): 10111010_2
    - -80_{10}:
        - $+80 = 01010000_2$
        - 1's complement: 10101111_2
        - 2's complement (add 1): 10110000_2

- **(i) (+70) + (+80):**
    - $01000110\_2 \\quad (+70)$
    - $+ \\ 01010000\_2 \\quad (+80)$

    - $\\mathbf{10010110\_2}$
    - **Overflow Detection:** Adding two positive numbers (MSB=0 for both) results in a sum with a negative sign bit (MSB=1). This indicates **overflow**. The true sum is +150, which is outside the 8-bit range.

- **(ii) (–70) + (–80):**
    - $10111010\_2 \\quad (-70)$
    - $+ \\ 10110000\_2 \\quad (-80)$

    - $(1)\\mathbf{01101010\_2}$ (The 9th bit (carry-out) is discarded)
    - **Overflow Detection:** Adding two negative numbers (MSB=1 for both) results in a sum with a positive sign bit (MSB=0). This indicates **overflow**. The true sum is -150, which is outside the 8-bit range.

---

6. (a) Perform the following conversions:

- (i) **Convert the hexadecimal number E7B2D.75 to octal**
    - **Step 1: Convert Hexadecimal to Binary:**
        - E = 1110
        - 7 = 0111
        - B = 1011
        - 2 = 0010
        - D = 1101
        - .
        - 7 = 0111
        - 5 = 0101
        - Combined Binary: $1110 \\ 0111 \\ 1011 \\ 0010 \\ 1101.0111 \\ 0101\_2$

    - **Step 2: Group binary digits into sets of three (from the binary point outwards), adding leading/trailing zeros as needed:**
        - **Integer Part:** $111 \\ 001 \\ 111 \\ 011 \\ 001 \\ 011 \\ 010\_2$ (Added one leading '0' to the leftmost group '1', and one trailing '0' to the rightmost group '01')
            - $111\_2 = 7\_8$
            - $001\_2 = 1\_8$
            - $111\_2 = 7\_8$
            - $011\_2 = 3\_8$
            - $001\_2 = 1\_8$

- $011\_2 = 3\_8$
- $010\_2 = 2\_8$
- Integer Octal: 7173132_8

- **Fractional Part:** $011 \\ 101 \\ 010\_2$ (Added one trailing '0' to the rightmost group '01')
  - $011\_2 = 3\_8$
  - $101\_2 = 5\_8$
  - $010\_2 = 2\_8$
  - Fractional Octal: 0.352_8

- **Final Octal Result:** $\\mathbf{7173132.352\_8}$

- (ii) **Convert** $(3431)\_5$ **to decimal.**
  - To convert a number from base N to decimal, use the formula: $d\_k \\times N^k + \\dots + d\_1 \\times N^1 + d\_0 \\times N^0$.
  - $(3431)\_5 = 3 \\times 5^3 + 4 \\times 5^2 + 3 \\times 5^1 + 1 \\times 5^0$
  - $= 3 \\times 125 + 4 \\times 25 + 3 \\times 5 + 1 \\times 1$
  - $= 375 + 100 + 15 + 1$
  - $= \\mathbf{491\_{10}}$

(b) Give characteristics table and excitation table of D flip-flop. What is the disadvantage of D flip-flop?

- **D Flip-Flop:** A D (Data) flip-flop is a fundamental clocked sequential logic circuit that stores a single bit of data. Its output (Q) directly follows the input (D) when a clock pulse or edge is active.
- **Characteristics Table of D Flip-Flop:**

- This table defines the next state ($Q\_t + 1$) of the flip-flop based on its current state (Q_t) and the input (D) when the clock is active.

Q_t	D	$Q\_t + 1$
0	0	0
0	1	1
1	0	0
1	1	1

- From the table, it's clear that the next state simply equals the D input: $Q\_t + 1 = D$.

- **Excitation Table of D Flip-Flop:**
  - This table shows what the D input must be to achieve a desired next state ($Q\_t + 1$) from a given current state (Q_t).

Q_t	$Q\_t + 1$	D
0	0	0
0	1	1
1	0	0
1	1	1

  - From the table, the required input D is directly equal to the desired next state: $D = Q\_t + 1$.

- **Disadvantage of D Flip-Flop:**
  - The primary "disadvantage" or limitation of a basic D flip-flop is its **inability to directly toggle its state (change from 0 to 1 or 1 to 0) without external logic.**

- To make a D flip-flop toggle, you must connect its Q-NOT output (Q') back to its D input ($D = Q'$). This requires additional external gating logic (like an inverter or XOR gate acting as an inverter) if toggling is a desired operation (e.g., in a counter circuit). Other flip-flop types, like the JK flip-flop, have a dedicated toggle mode (J=1, K=1) built-in.

(c) Simplify the following Boolean function F together with don't care conditions d in SOP (sum of products) form and draw the logic diagram for the simplified F.

- $F(x, y, z, w) = \\Sigma(0, 1, 8, 14, 15)$
- $d(x, y, z, w) = \\Sigma (2, 5, 10)$
- **K-Map (using xy for rows and zw for columns):**

$xy \\setminus zw$	00	01	11	10
00	1	1		X
01		X		
11			1	1
10	1			X

- **Grouping the 1s and Don't Cares (X):**
  c. **Quad 1 (Horizontal group in top row with don't care):** Group $m\_0, m\_1, m\_2(X)$. This makes a horizontal group of three, but we look for powers of 2 (2, 4, 8). This is not a direct rectangular group of 4.
    - Let's check for a quad using corners: $(m\_0, m\_2(X), m\_8, m\_10(X))$.
      - $m\_0(0000)$, $m\_2(0010)$, $m\_8(1000)$, $m\_10(1010)$.
      - Common terms: $y=0 \\implies y'$, and $w=0 \\implies w'$.

- So, this group simplifies to $y'w'$. This covers m_0, m_8 and uses don't cares m_2, m_10.

d. **Pair 1 (Horizontal group in third row):** Group m_14, m_15.

- $m\_14(1110)$, $m\_15(1111)$.
- Common terms: $x = 1, y = 1, z = 1$. w varies.
- So, this group simplifies to xyz. This covers m_14, m_15.

e. **Remaining 1s:** m_1 is still uncovered.

- $m\_1(0001)$. Can be grouped with $m\_5(X)$.
- Pair $(m\_1, m\_5(X))$:
  - $m\_1 (0001) \\rightarrow x'y'z'w$
  - $m\_5 (0101) \\rightarrow x'yz'w$
  - Common terms: $x' = 0, z' = 0, w = 1$. (The y bit changes).
  - So, this group simplifies to $x'z'w$. This covers m_1 and uses don't care m_5.

- **Simplified SOP Expression:** $F = y'w' + xyz + x'z'w$
- **Logic Diagram for Simplified F:**
  - **Inputs:** x, y, z, w and their complements (x', y', z', w') generated by inverters.
  - **AND Gates (one for each product term):**
    - AND Gate 1: Inputs y', w'. Output: $y'w'$
    - AND Gate 2: Inputs x, y, z. Output: xyz
    - AND Gate 3: Inputs x', z', w. Output: $x'z'w$
  - **OR Gate:**

- A single 3-input OR gate whose inputs are the outputs from the three AND gates.
- The final output is F.

---

7. (a) What is a binary adder-subtractor? Draw a diagram of a 4-bit binary adder-subtractor and explain its functionality.

- **What is a Binary Adder-Subtractor?**
  - A binary adder-subtractor is a single digital combinational circuit designed to perform both addition and subtraction of binary numbers.
  - It typically uses a set of full adders and XOR gates, along with a mode control input, to switch between addition and subtraction operations.
  - The core principle relies on the fact that subtraction can be achieved by adding the two's complement of the subtrahend to the minuend ($A - B = A + (\\text{2's complement of } B)$).
- **Functionality Explanation:**
  - The circuit usually includes a control input, often labeled **M (Mode)**.
  - **When M = 0 (for Addition):**
    - The XOR gates connected to the 'B' inputs of the full adders effectively pass the 'B' inputs directly (since $B \\oplus 0 = B$).
    - The initial carry-in to the least significant full adder (C_0) is set to 0.
    - The circuit performs standard binary addition: $A + B + 0 = A + B$.

- **When M = 1 (for Subtraction):**
    - The XOR gates connected to the 'B' inputs act as inverters (since $B \\oplus 1 = \\overline{B}$), generating the 1's complement of each bit of B.
    - The initial carry-in to the least significant full adder (C_0) is set to 1.
    - The circuit performs $A + \\overline{B} + 1$, which is equivalent to adding A to the 2's complement of B, thus effectively performing $A - B$.

- **Diagram of a 4-bit Binary Adder-Subtractor:**
    - [A diagram would be here. I will describe it.]
    - The circuit consists of four identical **Full Adder (FA)** modules, one for each bit position (from FA_0 for the LSB to FA_3 for the MSB).
    - Each **FA** has three inputs: two data bits (A_i, B_i) and a carry-in (C_i). It produces two outputs: sum (S_i) and carry-out ($C\_i + 1$).
    - **Input Connections:**
        - The A_i inputs of each FA_i directly receive the i-th bit of number A.
        - For each B_i, there's an **XOR gate**. One input to this XOR gate is B_i, and the other input is the **Mode (M)** control line. The output of this XOR gate is then fed to the B_i input of FA_i.
        - The **Mode (M)** control line is also directly connected to the **carry-in of the first full adder (**FA_0**)**, i.e., $C\_0 = M$.
        - The carry-out of FA_i ($C\_i + 1$) becomes the carry-in for $FA\_i + 1$ (C_i for the next stage).

- **Outputs:**
  - The sum/difference bits are $S\_3S\_2S\_1S\_0$.
  - The final carry-out from FA_3 (C_4) indicates overflow in signed arithmetic.

(b) Explain Direct Memory Access (DMA) in brief. Differentiate between Burst Transfer Mode and Cycle Stealing Mode in DMA.

- **Direct Memory Access (DMA) in Brief:**
  - Direct Memory Access (DMA) is a hardware feature that allows peripheral devices (like hard drives, network cards, or sound cards) to directly read from or write to system memory without involving the Central Processing Unit (CPU).
  - Normally, the CPU handles all data transfers between memory and I/O devices. This can be very inefficient for large data transfers as it keeps the CPU busy with I/O tasks.
  - With DMA, the CPU initiates the transfer by setting up a DMA controller with information such as the source address, destination address, and the amount of data to transfer. Once set up, the DMA controller takes over the bus and manages the transfer independently. This frees the CPU to perform other computational tasks, significantly improving system performance. After the transfer is complete, the DMA controller typically signals the CPU via an interrupt.
- **Differentiation between Burst Transfer Mode and Cycle Stealing Mode in DMA:**
  - **Burst Transfer Mode (or Block Transfer Mode):**
    - **Description:** In this mode, the DMA controller, once granted control of the system bus by the CPU, maintains control for the entire duration of the data transfer. It

transfers a complete block of data (multiple words or bytes) in a continuous "burst."

- **CPU Activity:** The CPU is effectively halted or temporarily idled during the entire block transfer period, as it cannot access the memory bus.
- **Pros:** Highly efficient for transferring large, contiguous blocks of data because the overhead of bus arbitration (requesting and granting the bus) occurs only once for the entire block. This results in very high data transfer rates.
- **Cons:** Can lead to significant CPU latency if the burst size is very large, potentially causing issues for time-sensitive tasks or interactive applications, as the CPU remains locked out.

- **Cycle Stealing Mode:**
  - **Description:** In this mode, the DMA controller requests the system bus from the CPU to transfer only one word or byte of data at a time. After transferring a single data unit, the DMA controller relinquishes control of the bus back to the CPU. It then re-requests the bus for the next data unit, effectively "stealing" one memory cycle at a time from the CPU.
  - **CPU Activity:** The CPU is only paused for a single memory cycle for each data transfer. It continues executing its program in between these brief pauses.
  - **Pros:** Minimizes the impact on CPU performance, allowing the CPU to continue processing most of the time. It is suitable for devices that transfer data in small, intermittent bursts, or where maintaining continuous CPU operation is crucial.

    - **Cons:** Less efficient for large, continuous data transfers due to the increased overhead of repeated bus arbitration (requesting and releasing the bus for each data unit). This can result in lower overall data transfer rates compared to burst mode for large blocks.

(c) A computer uses a memory unit with 256K words of 32 bits each. A binary instruction code stored in one word of memory. The instruction has four parts: a mode field to specify one of the 16 addressing modes, an operation code, a register code part to specify one of 60 registers, and an address part:

- (i) **How many bits are there in the mode field, operation code, register code part, and the address part?**
    - **Total Instruction Length:** The instruction code is stored in one word of memory, and each word is 32 bits. So, the total instruction length is **32 bits**.
    - **Mode Field:**
        - Number of addressing modes = 16.
        - Number of bits required = $\\lceil \\log\_2(16) \\rceil = \\mathbf{4 \\text{ bits}}$.
    - **Register Code Part:**
        - Number of registers = 60.
        - Number of bits required = $\\lceil \\log\_2(60) \\rceil = \\mathbf{6 \\text{ bits}}$ (since $2^5 = 32$ is too small, and $2^6 = 64$ is sufficient).
    - **Address Part:**
        - Memory unit size = 256K words.
        - $256K = 256 \\times 1024 = 2^8 \\times 2^{10} = 2^{18}$ words.

        - To address 2^{18} words, the address part must be **18 bits** long.
    - **Operation Code (Opcode) Part:**
        - Total bits in instruction = 32 bits.
        - Bits used by other fields = Mode bits + Register bits + Address bits = $4 + 6 + 18 = 28$ bits.
        - Bits remaining for Opcode = Total instruction length - Bits used by other fields = $32 - 28 = \\mathbf{4 \\text{ bits}}$.
- **Summary of Bits per Part:**
    - Mode Field: 4 bits
    - Operation Code: 4 bits
    - Register Code Part: 6 bits
    - Address Part: 18 bits

- (ii) **Draw the instruction word format and indicate the number of bits in each part.**
    - [A diagram would be here. I will describe it.]
    - The 32-bit instruction word would be divided into these four fields:
        - **Instruction Word (32 bits long)**
        - | Mode | Opcode | Register | Address |
        - |:----:|:------:|:--------:|:-------:|
        - | 4 bits | 4 bits | 6 bits | 18 bits |
- (iii) **How many bits are there in the data and address inputs of the memory?**
    - **Data Input/Output of Memory:**

- The memory unit is described as having "256K words of **32 bits each**."
- This directly means that each data unit transferred into or out of memory is 32 bits wide.
- Therefore, the data input and output lines (or the data bus width) of the memory are **32 bits**.

- **Address Input of Memory:**
  - The memory unit has a capacity of "256K words."
  - As calculated earlier, $256K = 2^{18}$ words.
  - To uniquely identify any of these 2^{18} words, the memory requires an address input with $\\lceil \\log\_2(2^{18}) \\rceil = \\mathbf{18 \\text{ bits}}$. This is the width of the address bus connected to the memory.