# Digital Techniques and Microprocessors (313305)

Model Question Paper — Set A · Model Answers

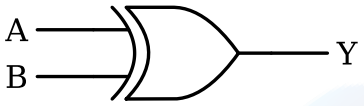
## Q.1

**a) Bit, nibble, byte, word.** 2 M

1. **Bit:** a single binary digit, 0 or 1 — the smallest unit of digital data.
2. **Nibble:** a group of 4 bits, e.g. 1011.
3. **Byte:** a group of 8 bits (2 nibbles), e.g. 1011 0110.
4. **Word:** a group of bits handled by the processor as one unit; for 8086 a word = 16 bits (2 bytes), e.g. 1011 0110 0101 1100.

*(½ M each)*

**b) EX-OR and NOR gate.** 2 M




*EX-OR gate: Y = A ⊕ B*

| A | B | Y = A ⊕ B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

*NOR gate: Y = (A + B)'*

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

*(Symbol ½ M + truth table ½ M for each gate)*

**c) Features of 8086.** 2 M

1. 16-bit microprocessor: 16-bit ALU, 16-bit registers and 16-bit data bus.
2. 20-bit address bus, so it can address $2^{20}$ = 1 MB memory; up to 64 K I/O ports.
3. Multiplexed address/data bus (AD0–AD15) — fewer pins (40-pin DIP, single +5 V supply).
4. 6-byte instruction queue; BIU and EU work in parallel (pipelining) → faster execution.
5. Memory segmentation (CS, DS, SS, ES — 64 KB segments).
6. Works in minimum mode or maximum mode; clock 5 / 8 / 10 MHz; 256 types of interrupts.

*(Any four, ½ M each)*

**d) ALE and DT/$\overline{R}$ pins.** 2 M

**ALE (Address Latch Enable, pin 25, output):** a positive pulse given by 8086 in T1 of every bus cycle to show that AD0–AD15 / A16–A19 carry a valid address. It is used as strobe for the address latches (8282) to separate the address from the multiplexed bus.

**DT/$\overline{R}$ (Data Transmit / Receive, pin 27, output):** decides the direction of data flow through the data transceivers (8286). DT/$\overline{R}$ = 1 → data is transmitted from CPU (write); DT/$\overline{R}$ = 0 → data is received by CPU (read). Both are minimum-mode signals.

*(1 M each)*

**e) DB and ASSUME directives.** 2 M

**DB (Define Byte):** reserves one or more bytes of memory and optionally initialises them.
Example: `NUM DB 25H ARR DB 10H, 20H, 30H MSG DB 'HELLO$'`

**ASSUME:** tells the assembler which logical segment is to be used with which segment register (it does not load the register).
Example: `ASSUME CS:CODE, DS:DATA`

*(1 M each)*

**f) Two arithmetic instructions.** 2 M

**i) ADD destination, source** — adds source to destination; result in destination; all status flags affected.
Example: AL = 35H, BL = 12H → `ADD AL, BL` → AL = 47H.

**ii) MUL source** — unsigned multiplication. 8-bit: AX ← AL × source; 16-bit: DX:AX ← AX × source.
Example: AL = 09H, BL = 06H → `MUL BL` → AX = 0036H.

(Other examples: SUB, INC, DEC, ADC, DIV, NEG.)

*(1 M each)*

**g) Minterm and maxterm.** 2 M

**Minterm:** a product (AND) term that contains every variable of the function once, either true or complemented. It is 1 for exactly one input combination. Example (3 variables): $\overline{A}BC = m_3$ (011).

**Maxterm:** a sum (OR) term that contains every variable once, true or complemented. It is 0 for exactly one input combination. Example: $(A + \overline{B} + C) = M_2$ (010).

*(1 M each)*

## Q.2

**a) Conversions.** 4 M

**i) $(237)_{10}$ → binary**

**Integer part (237):** divide repeatedly by 2, write remainders from bottom to top.

| Division | Quotient | Remainder |
|---|---|---|
| 237 ÷ 2 | 118 | 1 |
| 118 ÷ 2 | 59 | 0 |
| 59 ÷ 2 | 29 | 1 |
| 29 ÷ 2 | 14 | 1 |
| 14 ÷ 2 | 7 | 0 |
| 7 ÷ 2 | 3 | 1 |
| 3 ÷ 2 | 1 | 1 |
| 1 ÷ 2 | 0 | 1 |

**$\therefore (237)_{10} = (11101101)_2$**

**$(237)_{10}$ → hexadecimal**

**Integer part (237):** divide repeatedly by 16, write remainders from bottom to top.

| Division | Quotient | Remainder |
|---|---|---|
| 237 ÷ 16 | 14 | D |
| 14 ÷ 16 | 0 | E |

**$\therefore (237)_{10} = (ED)_{16}$**

Check: group 11101101 in 4 bits → 1110 1101 = ED ✓

**ii) $(3AC)_{16}$ → octal** (through binary)

**Hex → Binary:** replace each hex digit by its 4-bit binary equivalent.

| Hex digit | 3 | A | C |
|---|---|---|---|
| Binary | 0011 | 1010 | 1100 |

$(3AC)_{16} = (1110101100)_2$

**Binary → Octal:** make groups of 3 bits from the binary point (add 0s at the ends if needed).

| 3-bit group | 001 | 110 | 101 | 100 |
|---|---|---|---|---|
| Octal | 1 | 6 | 5 | 4 |

**$\therefore (3AC)_{16} = (1110101100)_2 = (1654)_8$**

*(2 M each)*

**b) De Morgan's theorems.** 4 M

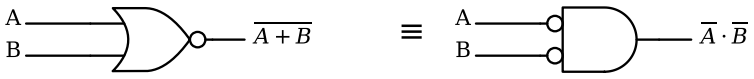
**Theorem 1:** The complement of a sum is equal to the product of the complements: $\overline{A + B} = \overline{A} \cdot \overline{B}$

**Theorem 2:** The complement of a product is equal to the sum of the complements: $\overline{A \cdot B} = \overline{A} + \overline{B}$

**Proof of theorem 1 (truth table):**

| A | B | A + B | $\overline{A + B}$ (LHS) | $\overline{A}$ | $\overline{B}$ | $\overline{A} \cdot \overline{B}$ (RHS) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

LHS and RHS columns are identical for all input combinations, hence theorem 1 is proved. NOR gate ≡ AND gate with inverted (bubbled) inputs:




**Proof of theorem 2 (truth table):**

| A | B | A · B | $\overline{A \cdot B}$ (LHS) | $\overline{A}$ | $\overline{B}$ | $\overline{A} + \overline{B}$ (RHS) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

LHS and RHS columns are identical, hence theorem 2 is proved. NAND gate ≡ OR gate with bubbled inputs:

*(Statements 1 M, each proof 1½ M)*

**c) Memory segmentation in 8086.** 4 M

- 8086 has a 20-bit address bus (1 MB memory) but its registers are only 16 bits. So the 1 MB memory is divided into logical blocks called **segments**, each of maximum **64 KB**.
- Four segments can be active at a time: **Code segment** (CS — program), **Data segment** (DS — data / variables), **Stack segment** (SS — stack) and **Extra segment** (ES — extra data, string destination).
- A segment register holds the upper 16 bits of the segment's starting address (segment base). A segment always starts at an address divisible by 16 (paragraph boundary).
- A memory location is given as **segment : offset**; physical address = segment × 10H + offset (e.g. CS:IP, DS:SI, SS:SP, ES:DI). Segments may be separate, overlapping or even the same.

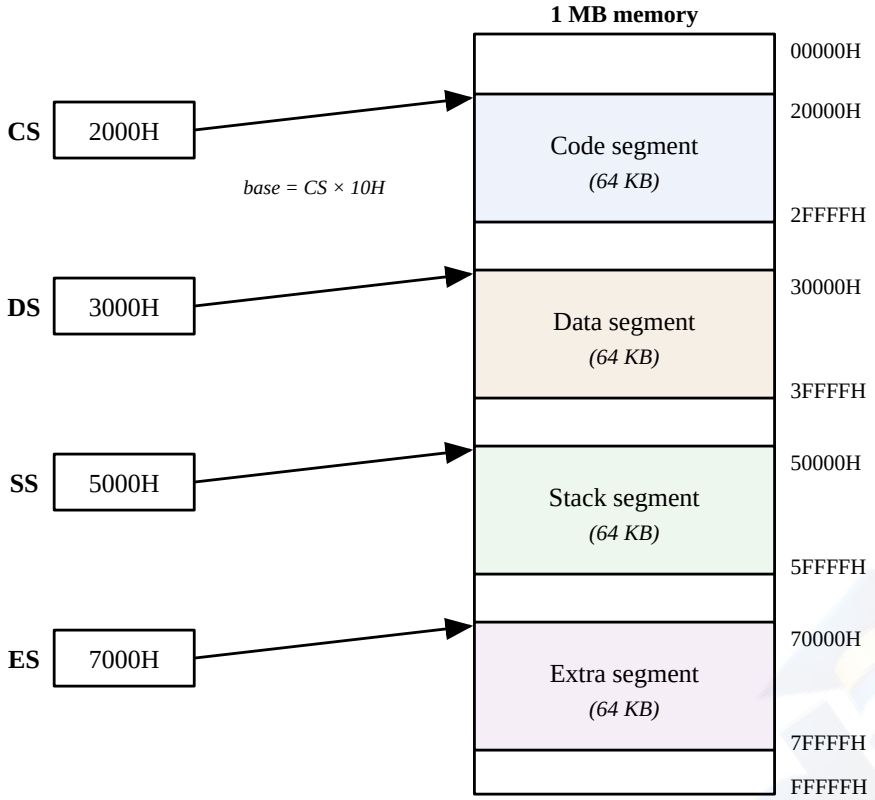



*Fig.: Memory segmentation in 8086 (1 MB memory, four 64 KB segments)*

**Advantages:**

1. 1 MB memory can be addressed using 16-bit registers.
2. Code, data and stack are kept separate — easy to manage and protect.
3. Programs are relocatable: only the segment register value has to be changed.
4. Allows multiple programs / modules to share memory easily.

*(Explanation 2 M, diagram 1 M, advantages 1 M)*

**d) ALP — addition of two 16-bit numbers.** 4 M

```
DATA SEGMENT
    NUM1  DW 8A57H
    NUM2  DW 9C3EH
    SUM   DW ?
    CARRY DB 00H
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA      ; initialise data segment
       MOV DS, AX
       MOV AX, NUM1      ; AX = first number
       ADD AX, NUM2      ; AX = AX + second number
       MOV SUM, AX       ; store sum
       JNC SKIP          ; no carry -> skip
       INC CARRY         ; carry = 1
SKIP:  MOV AH, 4CH       ; terminate program
       INT 21H
CODE ENDS
END START
```

**Result (checked by running the program on an 8086 simulator):**
NUM1 = 8A57H, NUM2 = 9C3EH → **SUM = 2695H**, **CARRY = 01H** (8A57H + 9C3EH = 12695H)

*(Data segment 1 M, logic 2 M, result storing 1 M)*

## Q.3

**a) Subtraction using 2's complement.** 4 M

**i) $(46)_{10} - (29)_{10}$**

$(46)_{10} - (29)_{10} = (46)_{10} + (2\text{'s complement of } (29)_{10})$. Use 8-bit numbers.

$(46)_{10}$ = 00101110  $(29)_{10}$ = 00011101

1's complement of 00011101 = 11100010
2's complement = 11100010 + 1 = **11100011**

```
  00101110   (46)
+ 11100011   (2's complement of 29)
------------
100010001
```

A final carry is generated → discard it. The result is positive: 00010001 = **+17**

**ii) $(25)_{10} - (58)_{10}$**

$(25)_{10} - (58)_{10} = (25)_{10} + (2\text{'s complement of } (58)_{10})$. Use 8-bit numbers.

$(25)_{10}$ = 00011001  $(58)_{10}$ = 00111010

1's complement of 00111010 = 11000101
2's complement = 11000101 + 1 = **11000110**

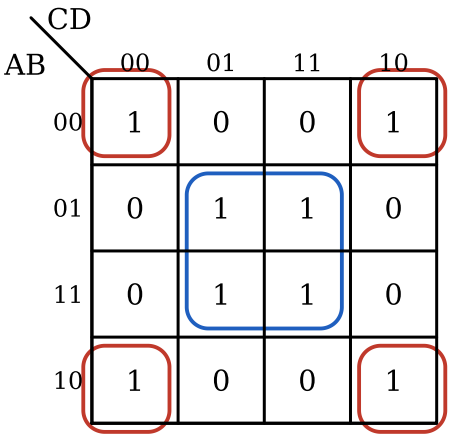
```
  00011001   (25)
+ 11000110   (2's complement of 58)
------------
 11011111
```

No final carry → the result is **negative** and is in 2's complement form.
Take 2's complement of 11011111: 1's complement = 00100000, + 1 = 00100001 = 33

**∴ $(25)_{10} - (58)_{10} = -33$** (check: 25 − 58 = -33)

*(2 M each)*

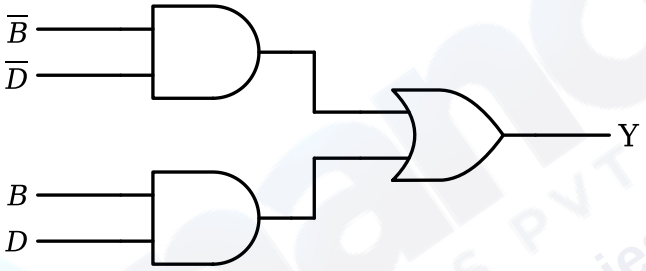
**b) K-map minimisation.** 4 M




*K-map for Y (grouping 1s)*

| Group | Cells (minterms) | Term |
|---|---|---|
| Group 1 | 0, 2, 8, 10 | $\overline{BD}$ |
| Group 2 | 5, 7, 13, 15 | BD |

**Minimal SOP: Y = $\overline{BD}$ + BD**




*Logic diagram (AND–OR)*

Note: Y = $\overline{BD}$ + BD = B ⊙ D (EX-NOR of B and D).

*(K-map 1 M, grouping 1 M, expression 1 M, logic diagram 1 M)*

**c) Addressing modes of 8086.** 4 M

The way in which the operand of an instruction is specified is called the addressing mode.

1. **Immediate addressing:** the 8/16-bit data is given in the instruction itself.
   e.g. `MOV AX, 2500H` ; AX ← 2500H
2. **Register addressing:** both operands are registers.
   e.g. `MOV BX, CX` ; BX ← CX
3. **Direct addressing:** the 16-bit offset (effective address) of the operand is given directly in the instruction.
   e.g. `MOV AL, [3000H]` ; AL ← [DS × 10H + 3000H]
4. **Register indirect addressing:** the offset of the operand is in BX, SI or DI.
   e.g. `MOV AL, [BX]` ; AL ← [DS:BX]
5. **Based addressing:** EA = BX or BP + displacement. e.g. `MOV AX, [BX + 05H]`
6. **Indexed addressing:** EA = SI or DI + displacement. e.g. `MOV AL, [SI + 10H]`

7. **Based indexed addressing:** EA = base + index (+ displacement). e.g. `MOV AX, [BX + SI + 08H]`

*(Any four modes with example, 1 M each)*

**d) String instructions.** 4 M

String instructions use DS:SI as source and ES:DI as destination. After each operation SI/DI are incremented (DF = 0, set by CLD) or decremented (DF = 1, set by STD) by 1 for byte and 2 for word. They can be repeated using REP / REPE / REPNE prefix with CX as counter.

1. **MOVSB** (move string byte): [ES:DI] ← [DS:SI]; SI ← SI ± 1, DI ← DI ± 1. No flags affected. e.g. `MOV CX, 10 / CLD / REP MOVSB` copies 10 bytes from source string to destination string.
2. **CMPSB** (compare string byte): computes [DS:SI] − [ES:DI], only flags are changed; SI, DI updated.
   e.g. `REPE CMPSB` compares two strings till a mismatch or CX = 0 (ZF = 1 at the end → strings equal).
3. **SCASB** (scan string byte): computes AL − [ES:DI], sets flags; DI ← DI ± 1.
   e.g. `MOV AL, 'A' / REPNE SCASB` searches the character 'A' in a string.
4. **LODSB** (load string byte): AL ← [DS:SI]; SI ← SI ± 1. e.g. `LODSB` reads the next character of a string into AL.

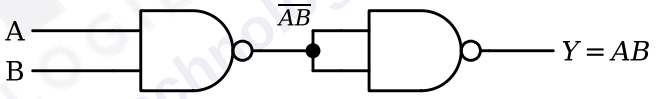
*(1 M each)*

## Q.4

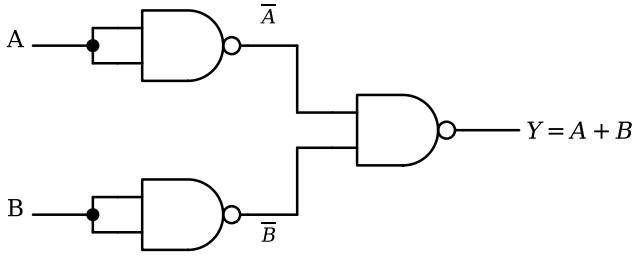
**a) Basic gates using NAND gates only.** 4 M

NAND is a universal gate, so any gate can be built using NAND gates alone.

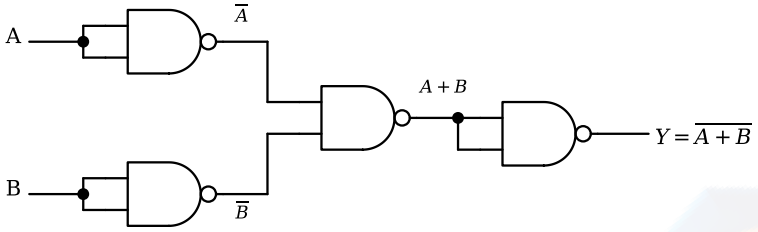
*i) NOT: inputs tied together*




*ii) AND: NAND followed by NAND inverter*




*iii) OR: inverted inputs to NAND*




*iv) NOR: OR followed by NAND inverter*

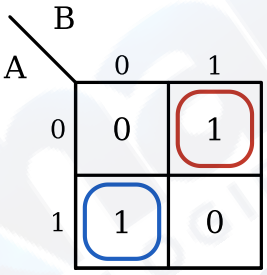
OR: $\overline{\overline{A} \cdot \overline{B}}$ = A + B (De Morgan)  NOR: invert the OR output.

*(1 M each)*

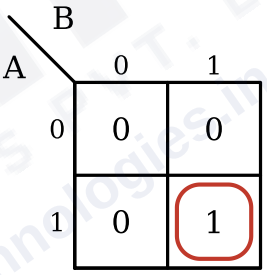
**b) Half adder design.** 4 M

A half adder adds two 1-bit inputs A and B and gives Sum (S) and Carry (C).

| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

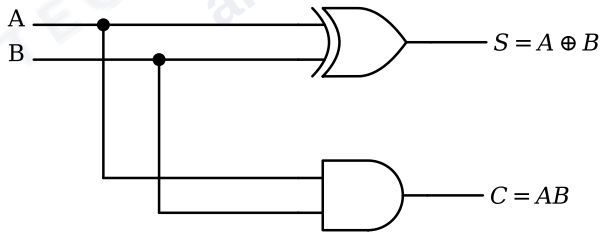



*S = A'B + AB'*




*C = AB*

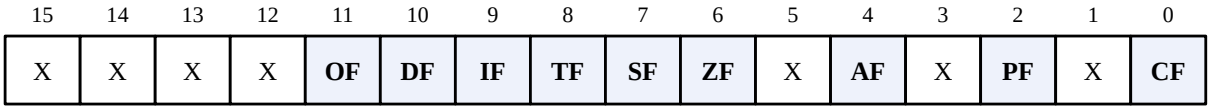
**S = $\overline{A}$B + A$\overline{B}$ = A ⊕ B  C = AB**




*Half adder logic diagram*

*(Truth table 1 M, K-maps 1 M, equations & diagram 2 M)*

**c) Flag register of 8086.** 4 M




**Status flags (6): CF, PF, AF, ZF, SF, OF**

**Control flags (3): TF, IF, DF**

*Fig.: Flag register of 8086 (X = undefined / reserved)*

8086 has a 16-bit flag register with 9 active flags.

| Flag | Name | Meaning (set = 1 when …) |
|---|---|---|
| CF | Carry | carry out of MSB in addition / borrow in subtraction |
| PF | Parity | lower byte of result has even number of 1s |
| AF | Auxiliary carry | carry from bit 3 to bit 4 (used in BCD/DAA) |
| ZF | Zero | result is zero |
| SF | Sign | MSB of result is 1 (negative) |
| OF | Overflow | signed result is out of range |
| TF | Trap | single-step mode for debugging (control) |
| IF | Interrupt enable | INTR (maskable interrupt) is enabled (control) |
| DF | Direction | string instructions auto-decrement SI/DI (control); DF = 0 → increment |

*(Diagram 2 M, explanation 2 M)*

**d) Physical address calculation.** 4 M

Physical address (PA) = Segment register × 10H + Offset (segment value shifted left by 4 bits, i.e. a 0 is appended in hex).

**i) CS = 3420H, IP = 1A5CH**

```
  CS × 10H =  34200H
+ IP       =  01A5CH
-----------------------
  PA       =  35C5CH
```

**ii) DS = 7B00H, BX = 0F36H**

```
  DS × 10H =  7B000H
+ BX       =  00F36H
-----------------------
  PA       =  7BF36H
```

**iii) SS = 5000H, SP = FFF0H**

```
  SS × 10H =  50000H
+ SP       =  0FFF0H
```

```
-----------------------
  PA       =  5FFF0H
```

**iv) ES = 2A3FH, DI = 4C21H**

```
  ES × 10H =  2A3F0H
+ DI       =  04C21H
-----------------------
  PA       =  2F011H
```

*(1 M each)*

**e) ALP — largest number in an array.** 4 M

```
DATA SEGMENT
    ARR   DB 25H, 7AH, 13H, 0C4H, 56H, 9BH, 3EH, 81H, 0A7H, 64H
    LARGE DB ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
       MOV DS, AX
       MOV SI, OFFSET ARR   ; SI points to array
       MOV CX, 9            ; 9 comparisons
       MOV AL, [SI]         ; first number = largest so far
UP:    INC SI
       CMP AL, [SI]         ; compare with next number
       JNC NEXT             ; AL >= [SI] -> keep AL
       MOV AL, [SI]         ; else new largest
NEXT:  LOOP UP
       MOV LARGE, AL        ; store result
       MOV AH, 4CH
       INT 21H
CODE ENDS
END START
```

**Result (checked by running the program on an 8086 simulator):**
Array: 25H, 7AH, 13H, C4H, 56H, 9BH, 3EH, 81H, A7H, 64H → **LARGE = C4H**

*(Initialisation 1 M, comparison loop 2 M, storing result 1 M)*

## Q.5

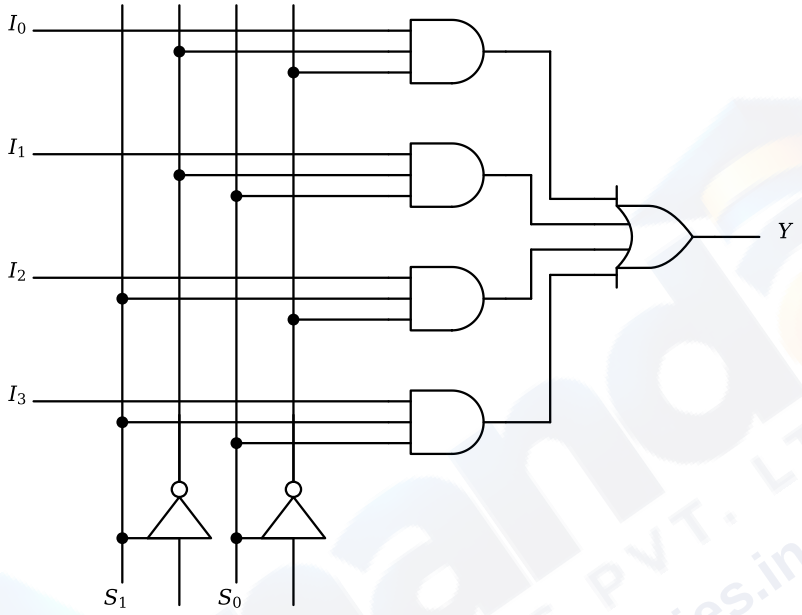
**a) 4 : 1 MUX and 16 : 1 MUX using 4 : 1 MUX.** 6 M

**i) 4 : 1 multiplexer:** selects one of 4 data inputs ($I_0$–$I_3$) and connects it to the single output Y

depending on the select inputs $S_1S_0$.

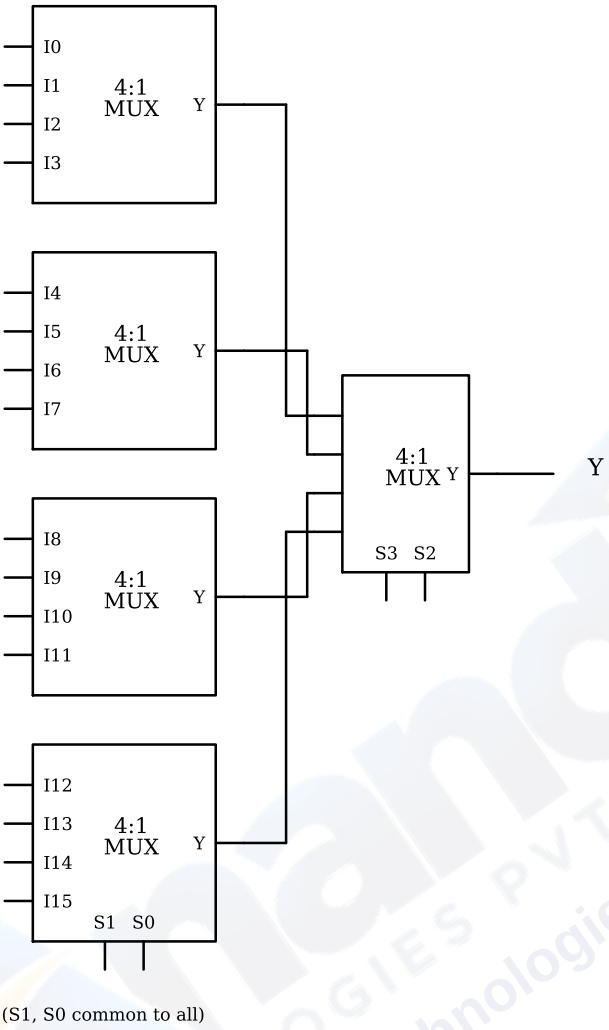
| $S_1$ | $S_0$ | Output Y |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |

$Y = \overline{S_1}\,\overline{S_0}I_0 + \overline{S_1}S_0I_1 + S_1\overline{S_0}I_2 + S_1S_0I_3$




*Logic diagram of 4 : 1 MUX*

**ii) 16 : 1 MUX using 4 : 1 MUX:** 16 inputs need 4 select lines ($2^4 = 16$). Four 4 : 1 MUX in the first level select one input from each group of four using $S_1S_0$; their outputs go to a fifth 4 : 1 MUX controlled by $S_3S_2$. Total = 5 MUX.




*16 : 1 MUX using five 4 : 1 MUX*

| $S_3$ $S_2$ | MUX selected in 1st level | Y |
|---|---|---|
| 0 0 | MUX-1 | $I_0 - I_3$ (as per $S_1S_0$) |
| 0 1 | MUX-2 | $I_4 - I_7$ |
| 1 0 | MUX-3 | $I_8 - I_{11}$ |
| 1 1 | MUX-4 | $I_{12} - I_{15}$ |

*(i) diagram 2 M + table 1 M; ii) diagram 2 M + explanation 1 M)*

**b) Architecture of 8086; BIU and EU.** 6 M

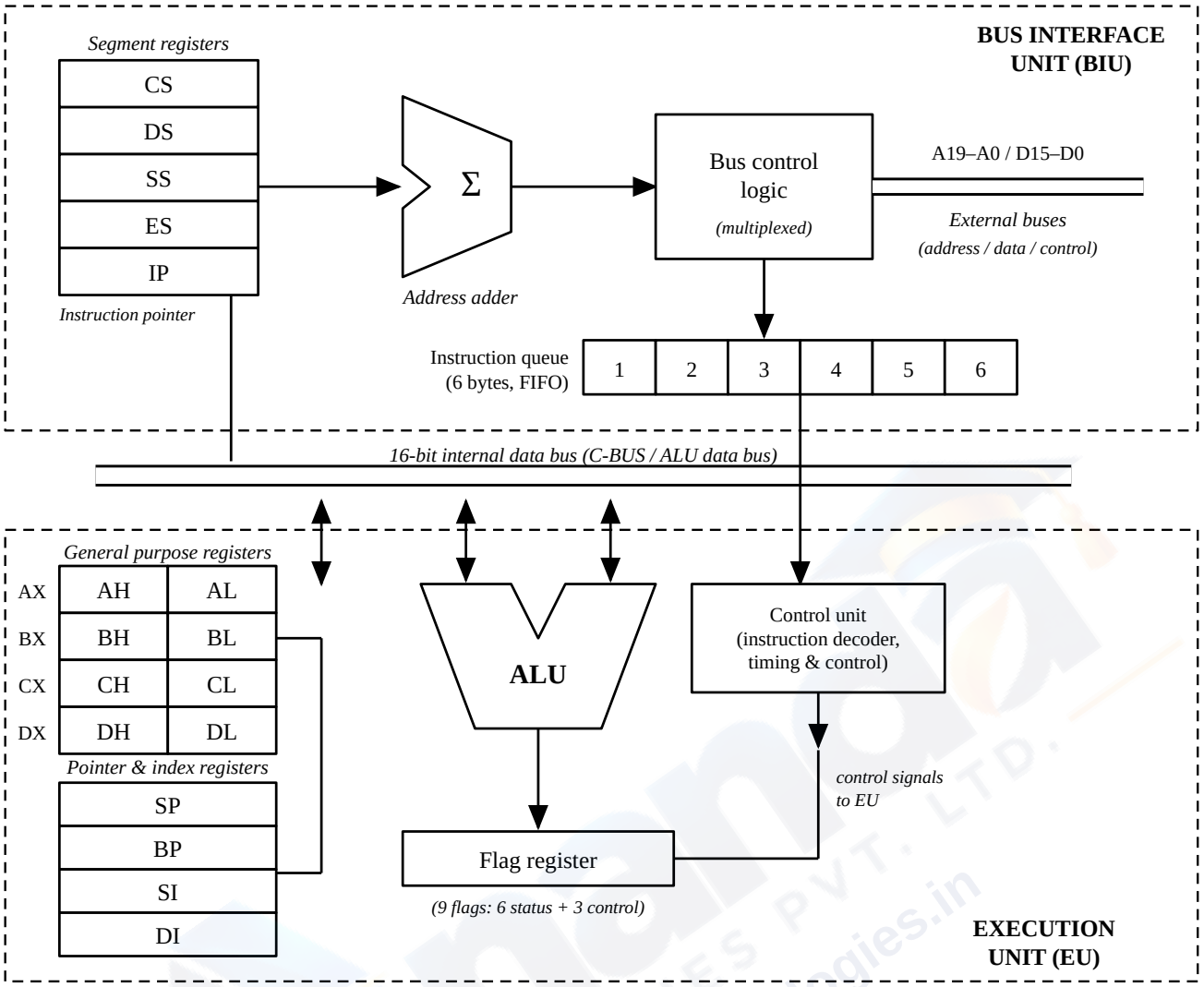



*Fig.: Architecture (functional block diagram) of 8086 microprocessor*

8086 is divided into two independent units which work in parallel (pipelining):

**Bus Interface Unit (BIU) — functions:**

1. Fetches instruction bytes from memory and stores them in the 6-byte instruction queue (pre-fetch) when the bus is free.
2. Calculates the 20-bit physical address using segment registers and the address adder (segment × 10H + offset).
3. Reads / writes data from / to memory and I/O ports for the EU (all bus operations).
4. Contains segment registers CS, DS, SS, ES, instruction pointer IP, address adder, queue and bus control logic.

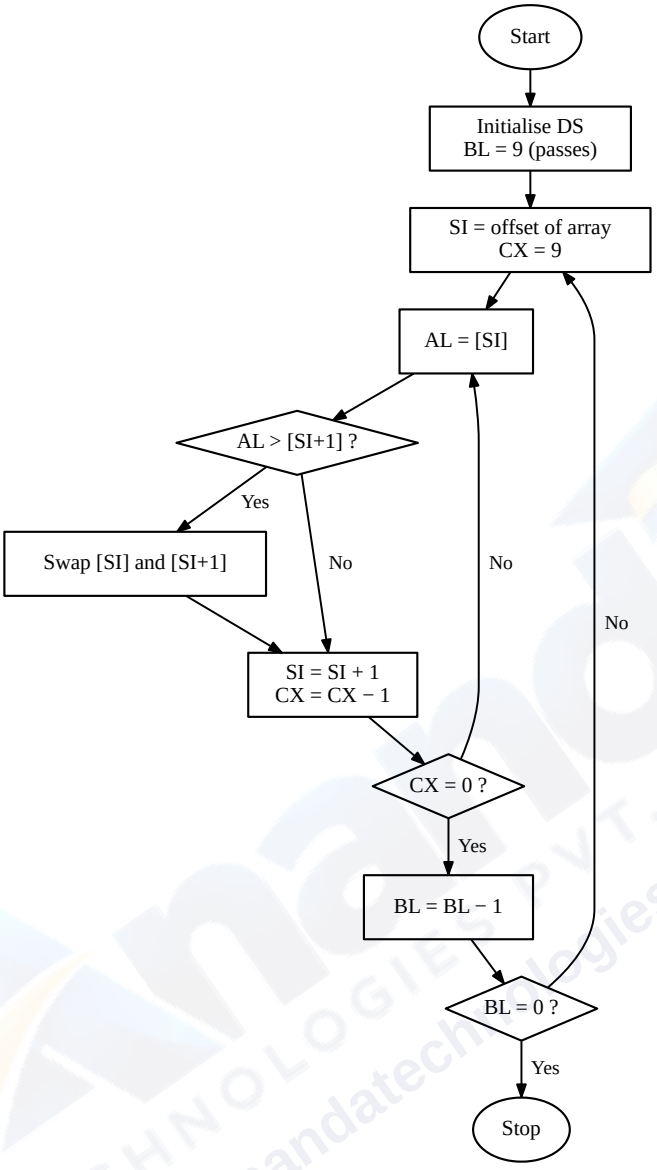
**Execution Unit (EU) — functions:**

1. Takes instruction bytes from the queue, decodes them (control unit / decoder) and executes them.
2. Performs arithmetic and logic operations in the 16-bit ALU and updates the flag register.
3. Contains general purpose registers AX, BX, CX, DX, pointer and index registers SP, BP, SI, DI and the flag register.

4. Gives the 16-bit offset (effective address) to BIU whenever memory or I/O access is needed.

*(Diagram 3 M, BIU 1½ M, EU 1½ M)*

**c) ALP — ascending order with flowchart.** 6 M




*Flowchart — ascending order (bubble sort)*

```
DATA SEGMENT
    ARR DB 45H, 12H, 89H, 0A3H, 06H, 77H, 3CH, 0F1H, 28H, 5DH
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START:  MOV AX, DATA
        MOV DS, AX
        MOV BL, 9            ; number of passes = N - 1
PASS:   MOV SI, OFFSET ARR   ; start of array
        MOV CX, 9            ; comparisons in one pass
COMP:   MOV AL, [SI]
        CMP AL, [SI+1]       ; compare adjacent numbers
        JBE NOSWAP           ; already in order (unsigned)
        XCHG AL, [SI+1]      ; swap the two numbers
        MOV [SI], AL
NOSWAP: INC SI
```

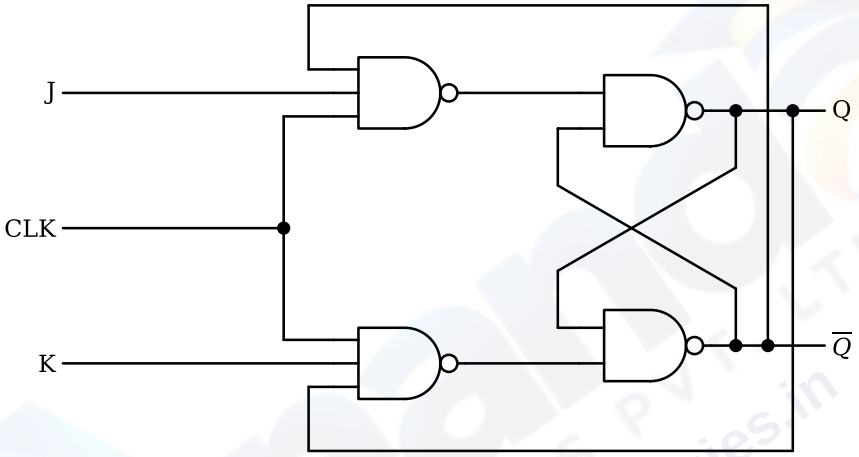
```
        LOOP COMP
        DEC BL
        JNZ PASS
        MOV AH, 4CH
        INT 21H
CODE ENDS
END START
```

**Result (checked by running the program on an 8086 simulator):**
Before: 45H, 12H, 89H, A3H, 06H, 77H, 3CH, F1H, 28H, 5DH
**After: 06H, 12H, 28H, 3CH, 45H, 5DH, 77H, 89H, A3H, F1H**

*(Flowchart 2 M, program 4 M)*

## Q.6

**a) Clocked J-K flip-flop; race-around.** 6 M




*Clocked J-K flip-flop using NAND gates*

| CLK | J | K | $Q_{n+1}$ | State |
|---|---|---|---|---|
| 0 | X | X | $Q_n$ | No change (clock inactive) |
| 1 | 0 | 0 | $Q_n$ | No change |
| 1 | 0 | 1 | 0 | Reset |
| 1 | 1 | 0 | 1 | Set |
| 1 | 1 | 1 | $\overline{Q_n}$ | Toggle |

**Working:** Q and $\overline{Q}$ are fed back to the input NAND gates, so J sets the FF only when it is reset and K resets it only when it is set. J = K = 1 makes the output toggle; this removes the invalid state of the SR flip-flop.

**Race-around condition:** when J = K = 1 and the clock pulse width ($t_p$) is larger than the propagation delay of the flip-flop, the output toggles many times during the same clock pulse. At the end of the pulse the output is uncertain. This is called race-around.

**Avoided by:** (i) keeping $t_p$ < propagation delay (difficult), (ii) using edge-triggered J-K flip-flop, or (iii) using **master–slave J-K flip-flop**, in which the master responds when CLK = 1 and the slave

copies the master when CLK = 0, so the output changes only once per clock.

*(Diagram 2 M, truth table 2 M, race-around 2 M)*

**b) Logical / rotate / shift instructions.** 6 M

Given AX = 5A3CH = 0101 1010 0011 1100 and BX = 0F0FH = 0000 1111 0000 1111 (each instruction works on these values).

| | Instruction | Result | Binary | CF | ZF |
|---|---|---|---|---|---|
| i) | `AND AX, BX` | AX = 0A0CH | 0000 1010 0000 1100 | 0 | 0 |
| ii) | `OR AX, BX` | AX = 5F3FH | 0101 1111 0011 1111 | 0 | 0 |
| iii) | `XOR AX, BX` | AX = 5533H | 0101 0101 0011 0011 | 0 | 0 |
| iv) | `NOT AX` | AX = A5C3H | 1010 0101 1100 0011 | — (no change) | — (no change) |
| v) | `MOV CL, 04H`<br>`ROL AX, CL` | AX = A3C5H | 1010 0011 1100 0101 | 1 | — (no change) |
| vi) | `SHR BX, 1` | BX = 0787H | 0000 0111 1000 0111 | 1 | 0 |

AND, OR, XOR always clear CF (and OF); ZF = 0 because results are non-zero. NOT affects no flags. ROL by 4 moves the upper nibble 5 to the lowest position (CF = last bit rotated = 1); ZF is not affected by rotates. SHR BX, 1 moves the LSB (1) into CF.

*(1 M each)*

**c) ALP — sum of 10 BCD numbers.** 6 M

DAA (decimal adjust after addition) converts the binary sum in AL into a valid packed BCD number after every ADD. A carry from DAA is counted in BL as the hundreds digit.

```
DATA SEGMENT
    ARR   DB 25H, 38H, 47H, 19H, 62H, 81H, 56H, 93H, 74H, 08H
    SUMLO DB ?            ; result stored at end of array
    SUMHI DB ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
       MOV DS, AX
       MOV SI, OFFSET ARR
       MOV CX, 10          ; 10 BCD numbers
       MOV AL, 00H         ; lower 2 BCD digits of sum
       MOV BL, 00H         ; higher BCD digit(s)
UP:    ADD AL, [SI]        ; add next BCD number
       DAA                 ; decimal adjust after addition
       JNC SKIP
       INC BL              ; carry -> next BCD digit
SKIP:  INC SI
       LOOP UP
       MOV SUMLO, AL
       MOV SUMHI, BL
       MOV AH, 4CH
       INT 21H
CODE ENDS
END START
```

**Result (checked by running the program on an 8086 simulator):**
25 + 38 + 47 + 19 + 62 + 81 + 56 + 93 + 74 + 08 = 503 → **SUMHI = 05H, SUMLO = 03H**, i.e. BCD sum = 0503

*(Initialisation 1 M, loop with ADD/DAA 3 M, carry & storing 2 M)*

**Note:** This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in