ES 333 Microprocessors and Embedded Systems Lab
Assignment 5

22110114 - Kaveri Visavadiya
22110132 - Mahi Agarwal

Q1) Basic Instruction Set Operations

```
JMP START
START: NOP

; a) using MVI, LXI, MOV, and LDA/STA —----------------------------------------------------------
MVI A, 74H                  ; A = 74H
LXI B, 0xABCD               ; BC = 0xABCD
MOV A, B                    ; A = 0xAB
STA 1000H                   ; mem(1000H) = 0xAB
0x1000          AB
XRA A                       ; clear A
LDA 1000H                   ; recovers A, A = mem(1000H) = 0xAB

; b) using INR, DCR, INX, and DCX —----------------------------------------------------------------
INR B                       ; B = 0xAC
DCR C                       ; C = 0xCC
INX B                       ; BC = 0xACCD
LXI H, 1234H                ; HL = 1234H
DCX H                       ; HL = 1233H

; c) using LDAX, STAX for indirect addressing —-------------------------------------------------
STAX B                      ; since BC = 0xACCD, mem(0xACCD) = 0xAB
0xACCD          AB
XRA A ; clear A
LDAX B                      ; recovers A, A = mem(ACCD) = 0xAB

; d) using XCHG —-------------------------------------------------------------------------------
XCHG                        ; HL = 0x0000, DE = 0x1233

HLT
```

Q2) Stack and Branch Operations

```
JMP START
START: NOP

; a) initialized SP, stored on stack using PUSH and recovered using POP—----------------------------

LXI SP, 2050H                ; stack pointer will point to 2050H

MVI B, 25H                   ;  b = 25H
PUSH B              ; sp = sp-2 = 204EH and mem[204f]= 25H(b) and mem[204e]= 00H(c)
MVI D, 40H                   ; d = 40H
PUSH D              ; sp = sp-2 = 204CH and mem[204d]= 40H(d) and mem[204c]= 00H(e)
MVI H, 15H                   ; h = 15H
PUSH H              ; sp = sp-2 = 204AH and mem[204b]= 15H(h) and mem[204a]= 00H(l)

POP H                        ; sp = sp+2 = 204CH and h = mem[204b] and l = mem[204a]
POP D                        ; sp = sp+2 = 204EH and h = mem[204d] and l = mem[204c]
POP B                        ; sp = sp+2 = 2050H and h = mem[204f] and l = mem[204e]
0x204A      00
0x204B      15
0x204C      00
0x204D      40
0x204E      00
0x204F      25

; b) used JMP, JNZ, CALL and RET instructions—-----------------------------------------------------

CALL AddBnD ; push next instr. address in stack (0x16 seen in machine code) and call AddBnD
0x204E      16
0x204F      00

STA 2060H                    ; mem[2060] = 7A
0x2060      7A

HLT

; Subroutines—-------------------------------------------------------------------------------------
AddBnD:
```

```
 MOV A, B ; a = b = 25h
 ADD D ; a = a+d = 25h+40h = 65h
 JNZ AddH ; if z flag = 1 jump to AddH

AddH:
 ADD H ; a = a+h = 65h+15h = 7Ah
 RET ; go to instruction address stored in sp
```

After running the program

A/PSW	0x7A02
BC	0x2500
DE	0x4000
HL	0x1500
SP	0x2050
PC	0x001A

All flags are 0.

Q3) a) Write an 8085 program that: If A=X_1X_2H, then store X_1X_1H in location 1000H and X_2X_2H in location 1001H using only 8085 logical instructions (AND, OR, XRA, CMA). Data transfer instructions should only be used when storing the final outputs.

```
JMP START
START: NOP

MVI B, 0xF8

; B = has initial val, A = working reg, C = temp reg
MOV A, B          ; A = X1 X2
ANI 0xF0          ; duplicate upper nibble, A = X1 00
MOV C, A          ; C = X1 00
RAR
RAR
RAR
RAR               ; A = 00 X1
ORA C             ; A = X1 X1
STA 1000H         ; mem(1000H) = X1 X1

MOV A, B          ; A = X1 X2
ANI 0x0F          ; duplicate lower nibble, A = 00 X2
MOV C, A          ; C = 00 X2
RAL
RAL
RAL
RAL               ; A = X2 00
ORA C             ; A = X2 X2
STA 1001H         ; mem(1001H) = X2 X2

HLT
```

After running the program

0x1000	FF
0x1001	88

b) Write a single 8085 program to demonstrate the following special instructions:

```
;—----------------------------------using RLC and RRC-------------------------------------------------
JMP START
```

```
START: NOP

MVI A, 80H          ; A = 80H
RLC                 ; A = 01H, C flag = 1
RRC                 ; A = 80H (same as original), C flag = 1

HLT


;—---------------------------------------using DAA instruction-----------------------------------------
; example 1 (adding 06H)--------------------------
JMP START
START: NOP

MVI A, 38H
MVI B, 45H
ADD B               ; A = 0x7D
DAA                 ; A = 83H
; since lower nibble > 9, 06H is added to A, and 83 is the BCD value
; S = 1, Z = 0, AC = 1, P = 0, C = 0
HLT

; example 2 (adding nothing)----------------------
JMP START
START: NOP

MVI A, 38H
MVI B, 41H
ADD B               ; A = 0x79
DAA                 ; A = 79H
; since lower and upper nibble <= 9, nothing is added and A = 79 itself is the BCD value
; S = 0, Z = 0, AC = 0, P = 0, C = 0
HLT

; example 3 (adding 60H)-------------------------
JMP START
START: NOP

MVI A, 83H
MVI B, 54H
```

```
ADD B                   ; A = 0xD7 and S = 1, Z = 0, AC = 0, P = 1, C = 0
DAA                     ; C = 1, A = 37H
; since upper nibble > 9, 60H is added to A and 137 is the BCD value
; S = 0, Z = 0, AC = 0, P = 0, C = 1
HLT

; example 4 (adding 66H)------------------------
JMP START
START: NOP

MVI A, 88H
MVI B, 44H
ADD B                   ; A = 0xCC and S = 1, Z = 0, AC = 0, P = 1, C = 0
DAA                     ; C = 1, A = 32H
; since lower and upper nibble > 9, 66H is added to A and 132 is the BCD value
; S = 0, Z = 0, AC = 1, P = 0, C = 1
HLT


;----------------------------------using CMP instruction—-----------------------------------------------
JMP START
START: NOP

MVI A, 8H               ; A = 8H
MVI B, 5H               ; B = 5H
CMP B                   ; since A > B, Z = 0 and C = 0

MVI A, 8H               ; A = 8H
MVI B, 8H               ; B = 8H
CMP B                   ; since A = B, Z = 1 and C = 0

MVI A, 5H               ; A = 5H
MVI B, 8H               ; B = 8H
CMP B                   ; since A < B, Z = 0 and C = 1

HLT


;----------------------------------using JC, JP and JPE instructions—-----------------------------------
JMP START
```

```
START: NOP

MVI A, 8H          ; A = 8H
MVI B, 5H          ; B = 5H
CMP B              ; S = 0, Z = 0, AC = 1, P = 1, C = 0
JP T1              ; will jump if S flag = 0

T1:
 MVI A, 5H         ; A = 5H
 MVI B, 8H         ; B = 8H
 CMP B             ; S = 1, Z = 0, AC = 0, P = 0, C = 1
 JC T2             ; will jump if C flag = 1

T2:
 MVI A, 5H         ; A = 5H
 MVI B, 5H         ; B = 5H
 CMP B             ; S = 0, Z = 1, AC = 1, P = 1, C = 0
 JPE T3            ; will jump if P flag = 1 (even parity)

T3:
 HLT
```

Q4) Array Processing Algorithm

- Assembly code:

Array of 8 bytes:

0x3000	12
0x3001	05
0x3002	18
0x3003	0A
0x3004	10
0x3005	15
0x3006	08
0x3007	0C
0x3008	76

```
JMP START

START: NOP
   LXI H, 3000H          ; HL points to start of array at 3000H
   MVI B, 08H            ; B = counter = 8 (number of array elements)

   MOV A, M              ; A = first element from [3000H]
   MOV C, A              ; C will hold the maximum value (init = first element)
   MOV D, A              ; D will hold the minimum value (init = first element)
   MOV E, A              ; E will hold the sum (init = first element)
   INX H                 ; Move pointer to the second element
   DCR B                  ; Decrement counter (now 7 elements remain)

LOOP1:
    MOV A, M             ; A = current element from array
    CMP C                ; Compare A with C (max so far)
    ;JNC
    JC SKIP_MAX          ; If A < C, do not update max
    MOV C, A             ; Otherwise, update max: C = A
SKIP_MAX:
    MOV A, M             ; Reload current element (A unchanged by CMP)
    CMP D                ; Compare A with D (min so far)
    JNC SKIP_MIN         ; If A >= D, skip updating min
    MOV D, A             ; Else update min: D = A
SKIP_MIN:
```

```
    MOV A, E            ; A = current sum
    ADD M               ; A = sum + current element
    MOV E, A            ; Update sum in E

INX H                   ; Point to the next element in the array
DCR B                   ; Decrement counter
JNZ LOOP1               ; Repeat loop until counter is 0

;   C = largest value,
;   D = smallest value,
;   E = sum of all 8 elements.
MOV A, C
STA 3100H               ; Store largest value at 3100H
MOV A, D
STA 3101H               ; Store smallest value at 3101H

MOV A, E                ; A = total sum (assumed to fit in 8 bits)
MVI D, 00H              ; D will serve as the quotient (average)

DIV_LOOP:
   CPI 08H              ; Compare A with 8
   JC DIV_DONE          ; If A < 8, division is done
   SBI 08H              ; Subtract 8 from A (A = A - 8)
   INR D                ; Increment quotient (average counter)
   JMP DIV_LOOP
DIV_DONE:
   MOV A, D
   STA 3102H            ; Store average at 3102H

   ; Count how many elements in the array are above average.
   ; First, load the computed average from 3102H.
   LDA 3102H
   MOV B, A             ; B = average value for later comparison

   ; Reinitialize pointer to start of array.
   LXI H, 3000H
   MVI C, 08H           ; C = counter = 8 (for the array)
   MVI D, 00H           ; D will count how many elements are above average

COUNT_LOOP:
```

```
    MOV A, M            ; A = current array element
    CMP B               ; Compare element with average (in B)
    JZ SKIP_COUNT       ; If equal, do not count
    JC SKIP_COUNT       ; If below average (carry set), do not count
    INR D               ; Otherwise, element > average so increment count
SKIP_COUNT:
    INX H               ; Move pointer to next element
    DCR C               ; Decrement counter
    JNZ COUNT_LOOP

    MOV A, D
    STA 3103H           ; Store count of elements above average at 3103H

HLT
```

- Output:

```
0x3100      18
0x3101      05
0x3102      0E
0x3103      04
```

After running the program

A/PSW	0x0457
BC	0x0E00
DE	0x0472
HL	0x3008
SP	0xFFFF
PC	0x005B

- Pseudocode:

1. Initialisation

- set pointer HL to address 3000H (start of the array of 8 bytes)

- set a counter (B) to 8 (number of array elements)

2. Set initial vals

- read the first array element at 0x3000 into A
- Initialize the maximum (C), minimum (D), and running sum (E) to that first element (12H).
- Increment HL to point to the second element and decrement the counter (B = 7).

3. Loop over rest of the 7 elements

- For each element in the array:
    1. Load the current element into A
    2. Compare with the current maximum (C). If A > C, C ← A.
    3. Compare with the current minimum (D). If A > D, D ← A.
    4. Add the current element to the running sum (E += A).
    5. Increment HL (to point to the next array element), HL ++.
    6. Decrement the counter B.

4. Store the maximum value (in C) into memory location 3100H and store the minimum value (in D) into memory location 3101H.

5. Calculate mean:

- E holds the total of all 8 elements.
- Initialize a quotient counter (D) to 0.
- While E >= 8,
    1. E -= 8
    2. D++
- When E < 8, the quotient D represents the average (rounded down).
- Store D at memory location 3102H.

6. Count elements above average

- Load the average from memory (3102H) into a register (B) for comparison.
- Initialize the array pointer HL to 3000H.
- Set a counter (C) to 8 (for iterating over the array) and set a counter (D) to 0 to count elements above average.
- For each element in the array
    1. Read the current element into A.
    2. If A <= B, skip it.
    3. If A > B, increment above average counter (D++)
    4. Increment HL to point to the next element.

5. Decrement the loop counter (C-=1).
6. Store the count (in D) at memory location 3103H.

7. End program

Q5) 16-bit Binary to BCD Conversion

- Assembly code:

```
        ;-----------------------------------------
        ; Input : 4000H (low byte), 4001H (high byte)
        ; Output: 4100H (thousands), 4101H (hundreds),
        ;             4102H (tens), 4103H (units)
        ; Temporary Counters:
        ;   5000H: thousands counter
        ;   5001H: hundreds counter
        ;   5002H: tens counter
        ;-----------------------------------------

JMP START

START:
        LXI H, 270FH
        SHLD 4000H
        LHLD 4000H                          ; HL = 0x270F

; to compute thousands digit, divisor (DE reg) = 1000 or 0x03E8
        MVI   D, 03H
        MVI   E, 0xE8
THOUSANDS_LOOP:
        CALL  COMPARE_HL_DE                 ; Compare HL with divisor in DE
        JC    STORE_THOUSANDS               ; If HL < divisor then exit loop
        CALL SUB_HL_DE                      ; HL = HL - DE
        ; Increment thousands counter stored at 5000H:
        LDA   5000H
        INR   A
        STA   5000H
        JMP   THOUSANDS_LOOP
STORE_THOUSANDS:
        LDA   5000H                         ; Load thousands digit from memory
        STA   4100H                         ; Store thousands digit

; to compute hundreds digit, divisor (DE reg) = 100 or 0x0064
        MVI   D, 00H
        MVI   E, 64H
```

```
HUNDREDS_LOOP:
        CALL  COMPARE_HL_DE
        JC    STORE_HUNDREDS          ; If HL < 100, exit loop
        CALL  SUB_HL_DE               ; HL = HL - DE
        ; Increment hundreds counter stored at 5001H:
        LDA   5001H
        INR   A
        STA   5001H
        JMP   HUNDREDS_LOOP
STORE_HUNDREDS:
        LDA   5001H                   ; Load hundreds digit from memory
        STA   4101H                   ; Store hundreds digit

; to compute tens digit, divisor (DE reg) = 10 or 0x000A
        MVI   D, 00H
        MVI   E, 0AH
TENS_LOOP:
        CALL  COMPARE_HL_DE
        JC    STORE_TENS              ; If HL < 10, exit loop
        CALL  SUB_HL_DE               ; HL = HL - DE
        ; Increment tens counter stored at 5002H:
        LDA   5002H
        INR   A
        STA   5002H
        JMP   TENS_LOOP
STORE_TENS:
        LDA   5002H                   ; Load tens digit from memory
        STA   4102H                   ; Store tens digit

; remainder in L is the units digit (0-9) since HL < 10
        MOV   A, L
        STA   4103H                   ; Store units digit

        HLT

;=================================================================
; Subroutines
;=================================================================

; COMPARE_HL_DE:  if HL < DE, C = 1, else if HL >= DE, C = 0
```

```
COMPARE_HL_DE:
        MOV   A, H
        CMP   D                         ; compare high bytes
        JNZ   COMP_DONE
        MOV   A, L
        CMP   E                         ; else compare low bytes if H = D
COMP_DONE:
        RET

; SUB_HL_DE: HL = HL - DE.
SUB_HL_DE:
        ; Subtract low bytes:
        MOV   A, L
        SUB   E                         ; A = L - E
        MOV   L, A

        MOV   A, H
        SBB   D                         ; A = H - D - Cy
        MOV   H, A
        RET
```

- Output:

If 0x4001 = 27, 0x4000 = 0F (0x270F = 9999 in BCD),

```
0x4100      09
0x4101      09
0x4102      09
0x4103      09
```

If 0x4001 = 10, 0x4000 = 00 (0x1000 = 4096 in BCD),

```
0x4100      04
0x4101      00
0x4102      09
0x4103      06
```

- Pseudocode:

1. Initialize output memory and temp regs:

- mem[4100H] = 0 ; thousands digit
- mem[4101H] = 0 ; hundreds digit
- mem[4102H] = 0 ; tens digit
- mem[4103H] = 0 ; units digit
- mem[5000H] = 0 ; thousands counter
- mem[5001H] = 0 ; hundreds counter
- mem[5002H] = 0 ; tens counter

2. Load the input number:
   - HL ← (mem[4001H] : mem[4000H])

3. Compute thousands digit:
   - Set divisor DE ← 1000 (0x03E8)
   - while (HL >= DE):
      a. HL ← HL - DE
      b. Increment the thousands counter (mem[5000H])
   - Store thousands counter to mem[4100H]

4. Compute hundreds digit:
   - Set divisor DE ← 100   (0x0064)
   - while (HL >= DE):
      a. HL ← HL - DE
      b. Increment the hundreds counter (mem[5001H])
   - Store hundreds counter to mem[4101H]

5. Compute tens digit:
   - Set divisor DE ← 10   (0x000A)
   - while (HL >= DE):
      a. HL ← HL - DE
      b. Increment the tens counter (mem[5002H])
   - Store tens counter to mem[4102H]

6. Compute units digit:
   - At this point, HL is less than 10.
   - Store HL into mem[4103H] as the units digit.

7. End program

Q6) GCD and LCM Calculator

- Assembly code:

```
;-----------------------------------------------------------------
; Data locations:
;  5000H: First 8-bit number (original1)
;  5001H: Second 8-bit number (original2)
;  5002H: Output GCD (8-bit)
;  5003H: Output LCM low byte (16-bit result)
;  5004H: Output LCM high byte (16-bit result)
;-----------------------------------------------------------------
START:      LXI H, 5000H                ; HL points to 5000H
            MOV A, M                    ; A <- [5000H]
            MOV B, A                    ; Save original1 in B

            INX H                       ; Now H points to 5001H
            MOV A, M                    ; A <- [5001H]
            MOV C, A                    ; Save original2 in C

            CMP B;
            JC C_IS_SMALLER;
            MOV E, B;
            MOV D, C;
            JMP NEXT;

; E WILL CONTAIN THE SMALLER NUMBER ALWAYS.
C_IS_SMALLER:
   MOV D, B;
   MOV E, C;

;----- Check for zero edge case: if either input is 0 -----
NEXT:
            MOV A, E
            CPI 00H
            JZ ZERO_CASE                ; if original1 == 0, jump


GCD_LOOP:   MOV A, D
            CMP E
```

```
        JZ GCD_DONE                 ; if D == E, then GCD found

        MOV A, D
        CALL MOD;
        MOV D, E;
        MOV E, A
        JMP GCD_LOOP


GCD_DONE:                           ; Now D = E = GCD.
        MOV A, D
        STA 5002H                   ; Store GCD at 5002H

;----- LCM Calculation -----
; Compute LCM = (original1 / GCD) * original2
; First, perform division: quotient = B / GCD
        MOV A, B                    ; A <- original1 (dividend)
        MVI D, 00H                  ; D will accumulate the quotient; clear D
        LDA 5002H                   ; A <- GCD (divisor)
        MOV E, A                    ; E <- GCD
        MOV A, B                    ; Restore dividend in A

DIV_LOOP:   CMP E                   ; Compare dividend with GCD
        JC  DIV_DONE                ; If dividend < GCD, division done
        SUB E                       ; A = A - GCD
        INR D                       ; Increment quotient (in D)
        JMP DIV_LOOP

DIV_DONE:                           ; Quotient is now in D.
        ; Next, compute LCM = (quotient) * (original2)
        MOV B, D                    ; Copy quotient into B (multiplier)
        LXI H, 0000H                ; Clear HL; HL will accumulate the 16-bit product

MULT_LOOP:  MOV A, B
        CPI 00H
        JZ  MULT_DONE               ; If multiplier zero, done multiplying
        ; Add original2 (in C) to the 16-bit product in HL.
        ; First add low bytes:
        MOV A, L
        ADD C
```

```
        MOV L, A
        ; Now add any carry to high byte:
        MOV A, H

        ACI 00H
        MOV H, A
        DCR B
        JMP MULT_LOOP

MULT_DONE:                      ; HL now holds the 16-bit LCM value.
        ; Store the low byte and high byte of LCM.
        MOV A, L
        STA 5003H               ; LCM low byte at 5003H
        MOV A, H
        STA 5004H               ; LCM high byte at 5004H
        JMP END

;----- ZERO CASE handling -----
; If one input is zero then set GCD = (nonzero value or 0 if both are 0)
; and LCM = 0.
ZERO_CASE:
        MOV A, D
        STA 5002H               ; GCD = bigger number
        MVI A, 00H
        STA 5003H               ; LCM low byte = 0
        STA 5004H               ; LCM high byte = 0
        JMP END;

MOD:
  CMP E;
  JC RETURN;
  JZ RETURN;
  SUB E;
  JMP MOD;

RETURN:
  RET;

END:    HLT
```

- Output:

After running the program

0x5000	10
0x5001	18
0x5002	08
0x5003	30

- Explanation:

Firstly we are storing the two numbers in B and C registers, then checking for which number is smaller and storing a copy of it again in E register, the bigger number's copy is stored in D register.
Then checking if the smaller one (E) is zero, if it is, the other number (D) is our GCD and LCM is 0.
If none is 0, we are using euclid's algorithm of repeated subtraction by subtracting smaller number from bigger number repetitively (MOD) and then storing previous smaller number in D (D = E) and remainder in E (because remainder would be smaller) repeating till we get both equal. And then the number in both would be gcd.
For lcm multiplying both numbers and dividing by gcd.

Q7) 3×3 Matrix Multiplication

First 3x3 matrix

0x6100	01
0x6101	02
0x6102	03
0x6103	04
0x6104	05
0x6105	06
0x6106	07
0x6107	08
0x6108	09

Second 3x3 matrix

0x6000	01
0x6001	02
0x6002	03
0x6003	04
0x6004	05
0x6005	06
0x6006	07
0x6007	08
0x6008	09

- Assembly code:

```
JMP START

MATRIX_A EQU 6000H;
MATRIX_B EQU 6100H;
RESULT EQU 6200H;
TEMP EQU 6010H;

START: NOP

MVI B, 0                ; row
MVI C, 0                ; col

ROW:
  MOV A, B              ;  checking if all rows done.
```

```
  CPI 3H;
  JNZ COL;
  HLT;

COL:
  MVI D, 0H                    ; will serve as index for FIND_ELEMENT(K)
  CALL FIND_ELEMENT    ; will store result in temp
  LXI H, RESULT                ;   getting address to store final value
  MOV A, L;
  ADD B;
  ADD B;
  ADD B;
  ADD C;
  MOV L, A;
  LDA TEMP;
  MOV M, A;
  MVI A, 00H;
  STA TEMP;

  ;going for next col
  INR C;
  MOV A, C;
  CPI 3H;
  JNZ COL                      ; if col == 3, reset it and go to next row
  MVI C, 0H;
  INR B;
  JMP ROW;

FIND_ELEMENT:
  ; A[R][K] * B[K][C]
  ; = MATRIX_A+3R+K * MATRIX_B+3K+C

  MOV A, D;
  CPI 3H;
  JNZ ELE_ROW;
  RET;

ELE_ROW:               ;
  LXI H, MATRIX_A;
  MOV A, B;
```

```
 ADD B;
 ADD B;
 ADD D;
 MOV L, A;
 MOV A, M;
 MOV E, A;

 LXI H, MATRIX_B;
 MOV A, C;
 ADD D;
 ADD D;
 ADD D;
 MOV L, A;
 MOV A, M;

 CALL MULT                    ; multiply content of e and a, result stored in e

 LDA TEMP;
 ADD L;
 STA TEMP;
 INR D;
 JMP FIND_ELEMENT;

MULT:
   MVI L, 00H                 ; Clear product in L
   MOV H, A
MULT_LOOP:
   MOV A, H;
   CPI 00H
   JZ MULT_DONE               ; If multiplier (H) is 0, done.
   MOV A, L
   ADD E                      ; A = product + multiplicand(E).
   MOV L, A                   ; Update product.
   DCR H                      ; Decrement multiplier.
   JMP MULT_LOOP
MULT_DONE:
   RET

HLT
```

- Output:

0x6200	1E
0x6201	24
0x6202	2A
0x6203	42
0x6204	51
0x6205	60
0x6206	66
0x6207	7E
0x6208	96

After running the program

A/PSW	0x0346
BC	0x0300
DE	0x0309
HL	0x6208
SP	0xFFFF
PC	0x000F

- Explanation:

B and C registers store row and col for which we are calculating currently. Row is the outer loop which checks whether we have done all the rows 0, 1, 2, if not go to col 0, 1, 2 in the same order. It calls the FIND_ELEMENT subroutine which calculates the particular element and stores it in temp.

Then to find the address to store we are taking the base address for the result matrix and then adding 3r (r=row) and 1c (c=col) to it. Then storing temp at this address.

If col becomes 3 when increasing, reset it to 0 and go to the next row.

FIND_ELEMENT calls ELE_ROW 3 times, which multiplies corresponding element according to value of k (D register) and then adds to the temp in memory. It uses mult internally to multiply to elements.