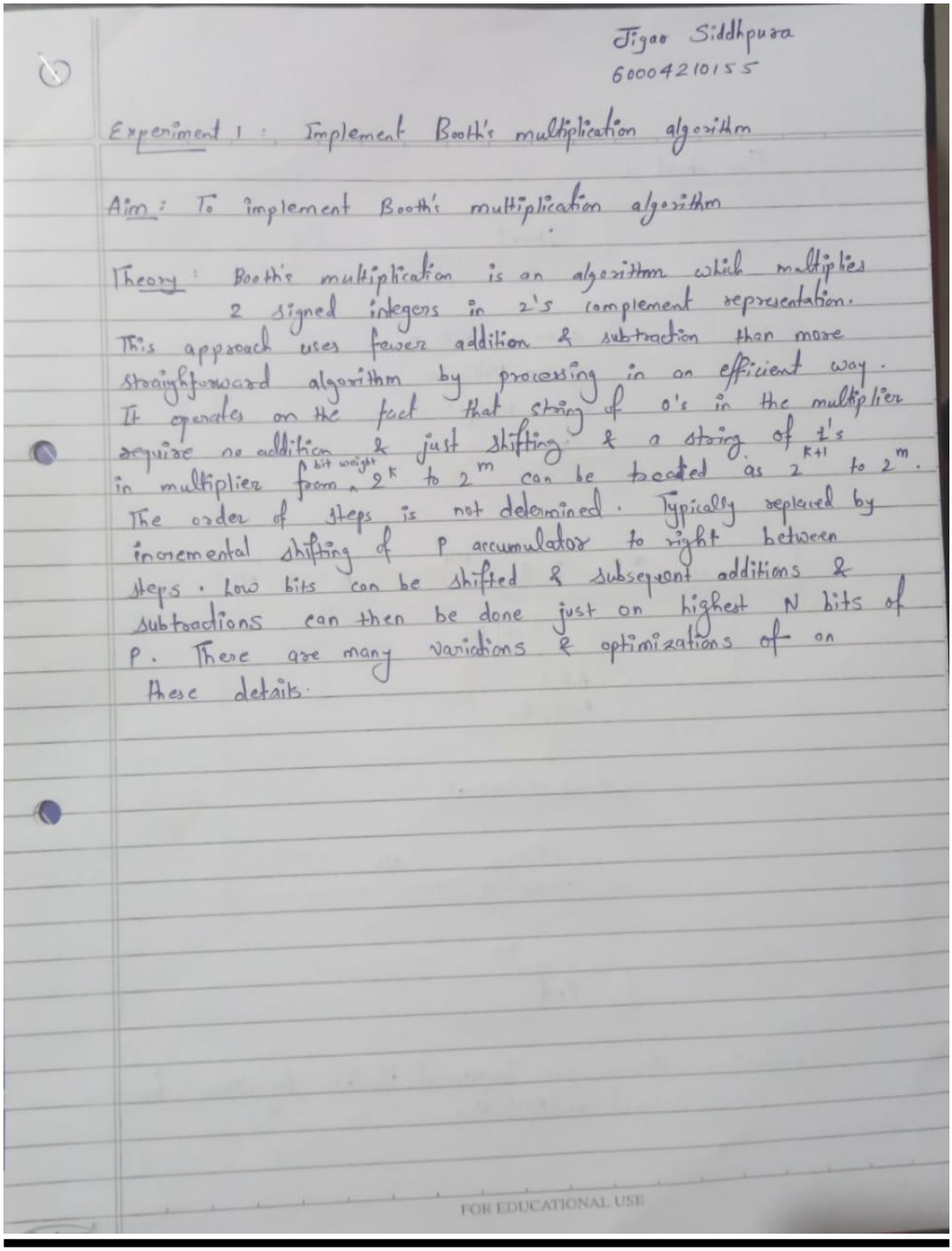
Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 1



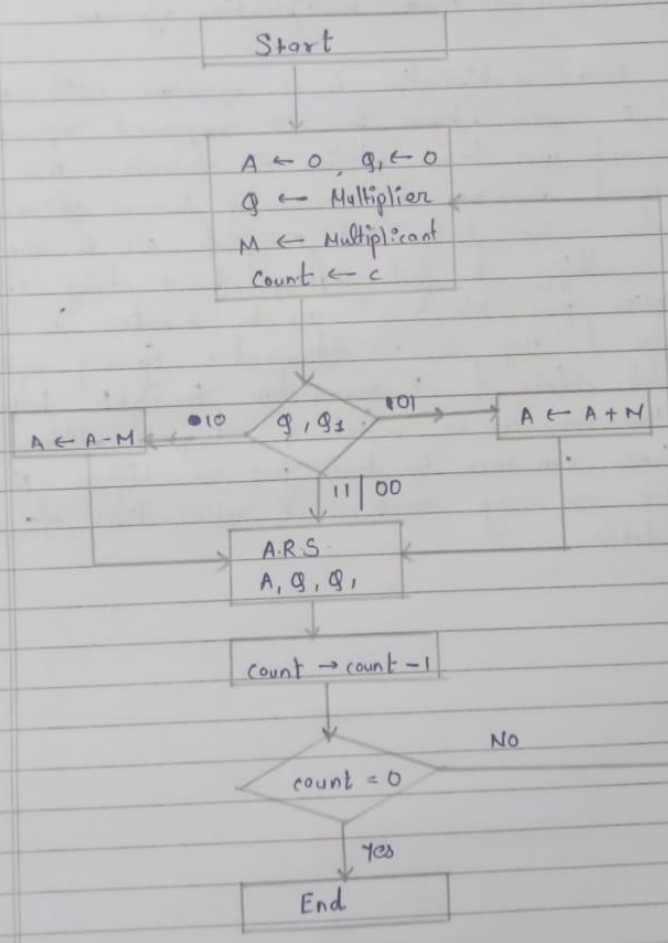
Jigar Siddhpura
60004210155

Experiment 1 : Implement Booth's multiplication algorithm

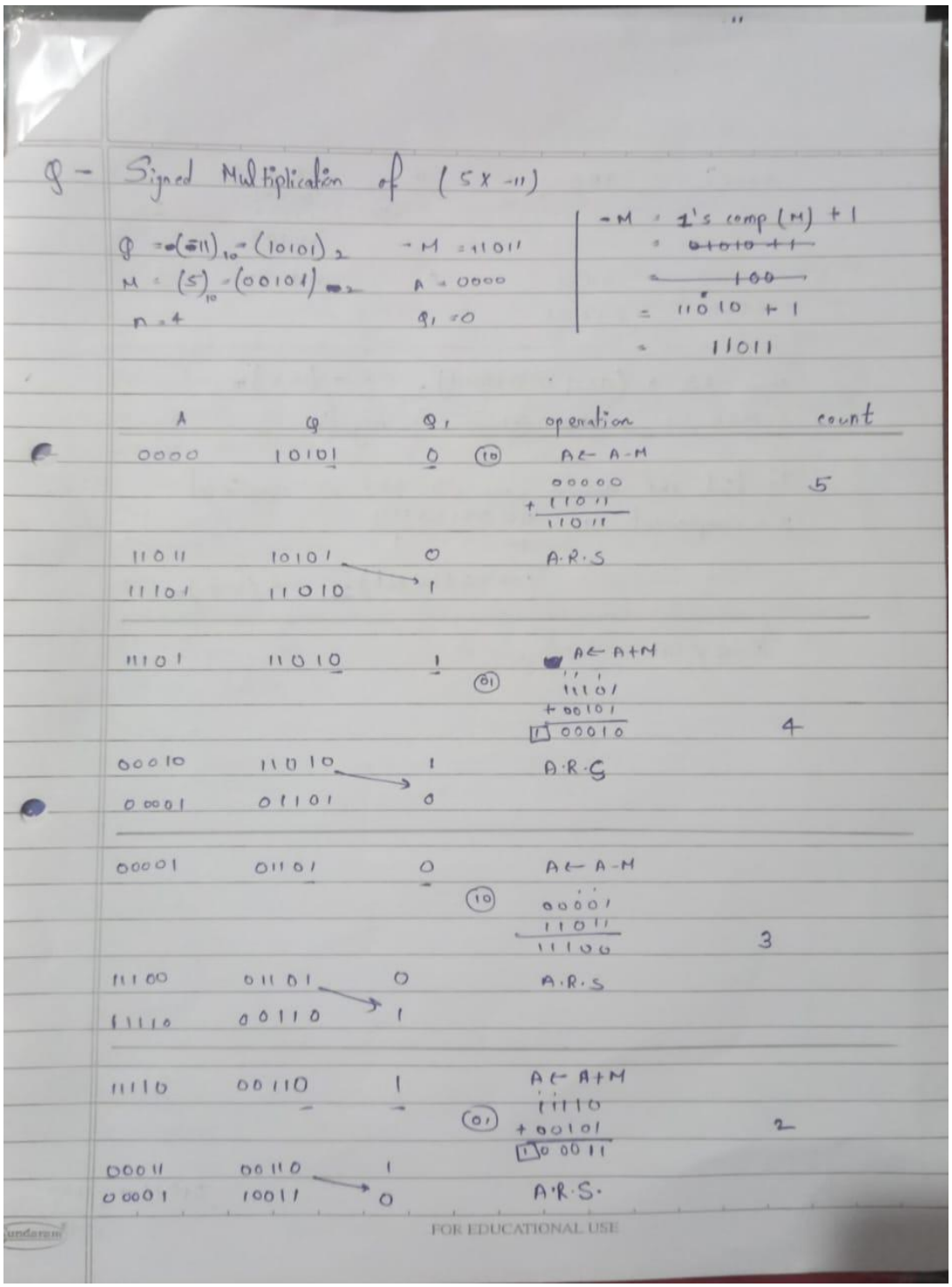
Aim : To implement Booth's multiplication algorithm

Theory : Booth's multiplication is an algorithm which multiplies 2 signed integers in 2's complement representation. This approach uses fewer addition & subtraction than more straightforward algorithm by processing in an efficient way. It operates on the fact that string of 0's in the multiplier require no addition & just shifting & a string of 1's in multiplier from bit weight 2^k to 2^m can be treated as 2^{k+1} to 2^m. The order of steps is not determined. Typically replaced by incremental shifting of P accumulator to right between steps. Low bits can be shifted & subsequent additions & subtractions can then be done just on highest N bits of P. There are many variations & optimizations of on these details.

Flow chart :



Conclusion : Hence we implemeat Booths algorithm for signed multiplication



Q- Signed Multiplication of (5 x -11)

Q = $(-11)_{10} = (10101)_2$ — -M = 11011

M = $(5)_{10} = (00101)_2$ — A = 0000

n = 4 — Q_1 = 0

-M = 1's comp (M) + 1
= ~~01010 + 1~~
= ~~100~~
= 11010 + 1
= 11011

A	Q	Q_1	operation	count
0000	10101	0	(10) A ← A-M: 00000 + 11011 = 11011	5
11011	10101	0 → 1	A.R.S	
11101	11010	1		
11101	11010	1	(01) A ← A+M: 11101 + 00101 = 1 00010	4
00010	11010	1 → 0	A.R.S	
00001	01101	0		
00001	01101	0	(10) A ← A-M: 00001 + 11011 = 11100	3
11100	01101	0 → 1	A.R.S	
11110	00110	1		
11110	00110	1	(01) A ← A+M: 11110 + 00101 = 1 00011	2
00011	00110 → 10011	1 → 0	A.R.S.	

00001	10011	0		A ← A − M
			(10)	00001 11011
				11 100

1

11100	10011	0	A.R.S.
11110	01001		

Now AQ → $(11110\ 01001)_2 = -(55)_{10}$

∵ M.S.B = 1, thus answer is negative

To find the answer, we will take 2's complement

2's complement = 00 00110 110
+ 1

$(00\ 00110\ 111)_2 = (55)_{10}$

∴ 5 × −11 = −55.

Code :

```
def binary_addition(a,b):
    """a,b are binary strings"""
    max_len = max(len(a), len(b))
    a = a.zfill(max_len)
    b = b.zfill(max_len)

    # Initialize the result
    result = ''

    # Initialize the carry
    carry = 0

    # Traverse the string
    for i in range(max_len - 1, -1, -1):
        r = carry
        r += 1 if a[i] == '1' else 0
        r += 1 if b[i] == '1' else 0
        result = ('1' if r % 2 == 1 else '0') + result

        # Compute the carry.
        carry = 0 if r < 2 else 1

    if carry != 0:
        result = '1' + result

    result = result.zfill(max_len)

    return result[-max_len:]

def complement(b):
    # 2's complement of binary number
    b_comp = ""
    for i in b:
        if i=="1": b_comp += "0"
        elif i=="0": b_comp += "1"
    b_comp = binary_addition(b_comp,"1")
    return b_comp

def binary_subtraction(a,b):
    """a,b are binary strings"""
    b_complement = complement(b)
    max_len = len(a)
    return binary_addition(a,b_complement.zfill(max_len))

def ARS(a,q):
    """all are binary strings"""
    q1 = q[-1]
    q = a[-1]+q[:-1]
    a = a[0]+a[:-1]
    return a,q,q1
```

```
def isNegative(bin_number):
    return bin_number[0] == "1"


def booth(multiplier,multiplicant,count,A="0000",Q1="0"):
    binary_multiplier  = bin(multiplier)[2:].zfill(count) if multiplier>0 else
bin(multiplier)[3:].zfill(count)
    binary_multiplicant = bin(multiplicant)[2:].zfill(count) if multiplicant>0
else bin(multiplier)[3:].zfill(count)
    print(f"Multiplier Q = {binary_multiplier}, Multiplicant M =
{binary_multiplicant}, A = {A}, count = {count}, Q1 = 0\n")
    print("Count\tA\tQ\tQ1")
    print("---------------------------------------")
    while(count!=0):
        print(f"{count}\t{A}\t{binary_multiplier}\t{Q1}")
        case = binary_multiplier[-1]+Q1
        print(f"A,Q <- {case}")
        if(case=="01"):
            A = binary_addition(A,binary_multiplicant)
            print(f"{count}\t{A}\t{binary_multiplier}\t{Q1}\tbefore ARS")
        elif(case=="10"):
            A = binary_subtraction(A,binary_multiplicant)
            print(f"{count}\t{A}\t{binary_multiplier}\t{Q1}\tbefore ARS")
        elif(case=="00" or case=="11"):
            pass
        A,binary_multiplier,Q1 = ARS(A,binary_multiplier)
        print(f"{count}\t{A}\t{binary_multiplier}\t{Q1}")
        count -= 1
        print("---------------------------------------")
    bin_ans = A+binary_multiplier
    ans = int(complement(bin_ans),base=2) if isNegative(bin_ans) else
int(bin_ans,base=2)
    print(f"Answer = {ans}")


booth(int(input("Enter Multiplier = ")),int(input("Enter Multiplicant = ")),4)
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\booth.py"
Enter Multiplier = 3
Enter Multiplicant = 7
Multiplier Q = 0011, Multiplicant M = 0111, A = 0000, count = 4, Q1 = 0

Count   A       Q       Q1
---------------------------------------
4       0000    0011    0
A,Q <- 10
4       1001    0011    0       before ARS
4       1100    1001    1
---------------------------------------
3       1100    1001    1
A,Q <- 11
3       1110    0100    1
---------------------------------------
2       1110    0100    1
A,Q <- 01
2       0101    0100    1       before ARS
2       0010    1010    0
---------------------------------------
1       0010    1010    0
A,Q <- 00
1       0001    0101    0
---------------------------------------
Answer = 21
PS D:\SEM 5\POA\EXPERIMENTS>
```

Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 2

Jigar Siddhpura
60004210155

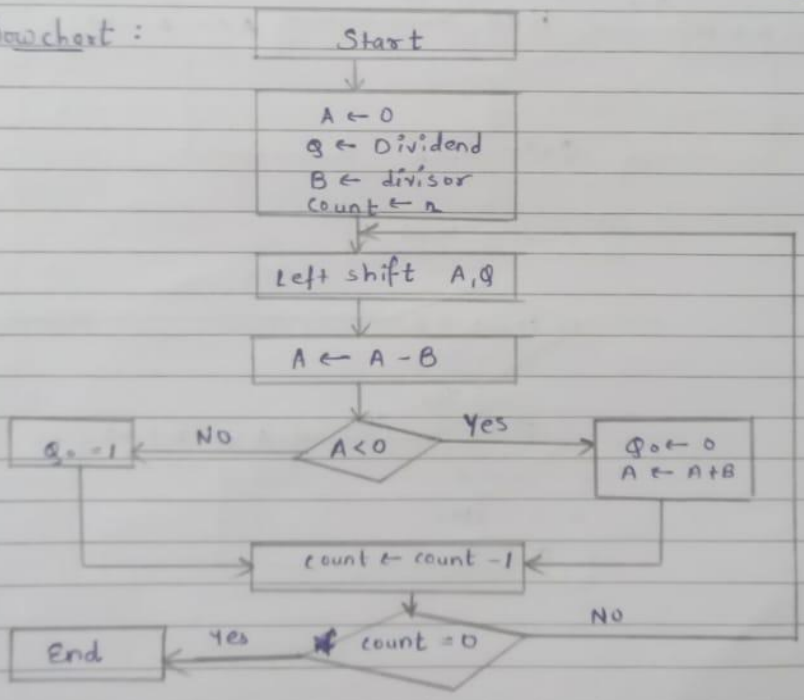
Experiment 2

Aim: To study & implement restoring & non-restoring division algorithm.

Theory:

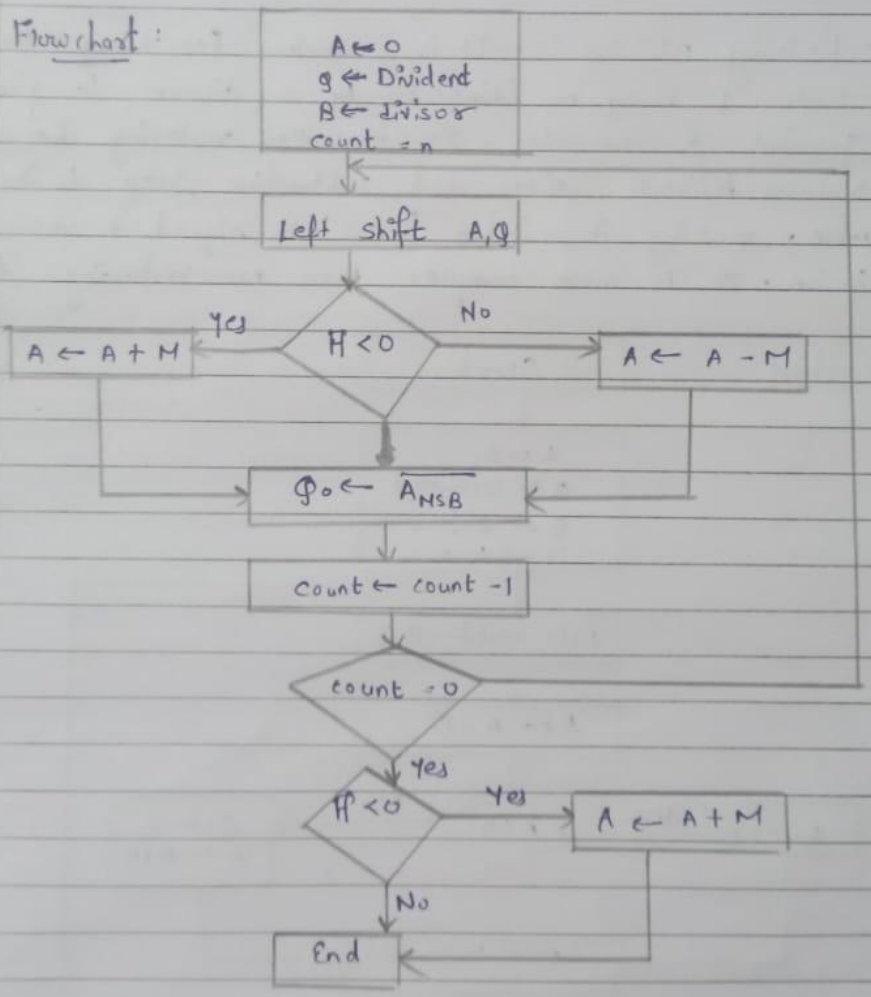
1. Restoring Division :- It is a method for binary division to divide 1 binary no. (dividend) by a divisor to give ~~get~~ quotient & remainder. It is called restoring because it restores 'dividend', after each subtraction step to its original value, making it suitable for both signed & unsigned division. It is more complex than non-restoring division.

Flowchart:



Non-restoring Division :- It is another method for binary division. Here it doesn't restore the orignal value of dividend after each subtraction, making it more suitable for hardware implementations where subtraction can be more complex. Its commonly used in digital arithmetic circuits.

Flowchart :



Q - Restoring division $\left(\frac{14}{Q} \mid \frac{5}{M}\right)$

→ $Q = (14)_{10} = (01110)_2$

$B = (5)_{10} = (00101)_2$

Count = $(n+1)$ bits

= 5 bits = 5

$-B = (11011)_2$

$A = 00000$

00101 → 1's comp. → 11010 + 1 = 11011

A	Q	operation	count
00000	01110	A.L.S.	
00000	1110□	A ← A − B 00000 + 11011 11011	4
11011	1110□		
00000	11100	$Q_0 \leftarrow 0$ A ← A + B 11011 + 00101 1 00000	
00000	11100	A.L.S	
00001	1100□	A ← A − B 00001 + 11011 11100	3
11100	1100□		
00001	11000	$Q_0 = 0$ & A ← A+B 11100 + 00101 1 00001	
00001	11000	A.L.S	
00011	1000_	A ← A − B 00011 + 11011 11110	2
11110	1000_		
00011	1000[0]	$Q_0 \leftarrow 0$ & A ← A+B 11110 + 00101 1 00011	

00011	10000	A·L·S
00111	0000 □	A ← A-B

```
  00111
+ 11011
-------
 100010
```

00010	0000 □	$q_0 = 1$
00010	00001	

00010	00001	A·L·S·
00100	00001 _	A ← A - B

```
  00100
+ 11011
-------
  11111
```

11111	00001 _	$q_0 = 0$ & A ← A + B
00100	00010	

```
  11111
+ 00101
-------
 100100
```

Answer : Remainder : A → $(00100)_2 = (4)_{10}$

Quotient : Q → $(00010)_2 = (2)_{10}$

Answer : Quotient = $(0011)_2 = (3)_{10}$

Remainder = ~~$(0011)_2$~~ $(0000)_2 = (0)_{10}$

Conclusion : Hence we have implement both restoring & non-restoring division to perform binary division.

Restoring Division Code :

```
def binary_addition(a,b):
    """a,b are binary strings"""
    max_len = max(len(a), len(b))
    a = a.zfill(max_len)
    b = b.zfill(max_len)

    # Initialize the result
    result = ''

    # Initialize the carry
    carry = 0

    # Traverse the string
    for i in range(max_len - 1, -1, -1):
        r = carry
        r += 1 if a[i] == '1' else 0
        r += 1 if b[i] == '1' else 0
        result = ('1' if r % 2 == 1 else '0') + result

        # Compute the carry.
        carry = 0 if r < 2 else 1

    if carry != 0:
        result = '1' + result

    result = result.zfill(max_len)

   return result[-max_len:]

def complement(b):
    # 2's complement of binary number
    b_comp = ""
    for i in b:
        if i=="1": b_comp += "0"
        elif i=="0": b_comp += "1"
    b_comp = binary_addition(b_comp,"1")
   return b_comp

def binary_subtraction(a,b):
    """a,b are binary strings"""
    b_complement = complement(b)
    max_len = len(a)
   return binary_addition(a,b_complement.zfill(max_len))




def ALS(a,q):
    """all are binary strings"""
    # x = Q not
    a = a[1:]+q[0]
    q = q[1:]+"_"
   return a,q

def isNegative(bin_number):
```

```
    return bin_number[0] == "1"


def restoring_division(dividend,divisor,count,A="00000",Q1="0"):
    binary_dividend  = bin(dividend)[2:].zfill(count) if dividend>0 else
bin(dividend)[3:].zfill(count)
    binary_divisor = bin(divisor)[2:].zfill(count+1) if divisor>0 else
bin(dividend)[3:].zfill(count+1)
    print(f"Dividend Q = {binary_dividend}, divisor B = {binary_divisor}, A = {A},
count = {count}\n")
    print("Count\tA\tQ\tOperation")
    print("----------------------------------------")

    while(count!=0):
        print(f"{count}\t{A}\t{binary_dividend}\tbefore ALS")
        A,binary_dividend = ALS(A,binary_dividend)
        print(f"{count}\t{A}\t{binary_dividend}\tA <- A-B")
        A = binary_subtraction(A,binary_divisor)
        case = isNegative(A)
        # print(f"A,Q <- {case}")
        if(case):
            binary_dividend = binary_dividend.replace("_","0")
            A = binary_addition(A,binary_divisor)
            print(f"{count}\t{A}\t{binary_dividend}\tQ. = 0 & A<-A+B")
        else:
            # A = binary_subtraction(A,binary_divisor)
            binary_dividend = binary_dividend.replace("_","1")
            print(f"{count}\t{A}\t{binary_dividend}\tQ. = 1")
        print(f"{count}\t{A}\t{binary_dividend}")
        count -= 1
        print("----------------------------------------")

    remainder = int(A,base=2)
    quotient = int(binary_dividend,base=2)

    print(f"Remainder = {remainder}, Quotient = {quotient}")

restoring_division(int(input("Enter Dividend = ")),int(input("Enter Divisor = ")),4)
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\restoringDivision.py"
Enter Dividend = 11
Enter Divisor = 3
Dividend Q = 1011, divisor B = 00011, A = 00000, count = 4

Count   A       Q       Operation
----------------------------------------
4       00000   1011    before ALS
4       00001   011_    A <- A-B
4       00001   0110    Q. = 0 & A<-A+B
4       00001   0110
----------------------------------------
3       00001   0110    before ALS
3       00010   110_    A <- A-B
3       00010   1100    Q. = 0 & A<-A+B
3       00010   1100
----------------------------------------
2       00010   1100    before ALS
2       00101   100_    A <- A-B
2       00010   1001    Q. = 1
2       00010   1001
----------------------------------------
1       00010   1001    before ALS
1       00101   001_    A <- A-B
1       00010   0011    Q. = 1
1       00010   0011
----------------------------------------
Remainder = 2, Quotient = 3
PS D:\SEM 5\POA\EXPERIMENTS>
```

Non-restoring Division Code :

```
def binary_addition(a,b):
    """a,b are binary strings"""
    max_len = max(len(a), len(b))
    a = a.zfill(max_len)
    b = b.zfill(max_len)

    # Initialize the result
    result = ''

    # Initialize the carry
    carry = 0

    # Traverse the string
    for i in range(max_len - 1, -1, -1):
        r = carry
        r += 1 if a[i] == '1' else 0
        r += 1 if b[i] == '1' else 0
        result = ('1' if r % 2 == 1 else '0') + result

        # Compute the carry.
        carry = 0 if r < 2 else 1

    if carry != 0:
        result = '1' + result

    result = result.zfill(max_len)

   return result[-max_len:]

def complement(b):
    # 2's complement of binary number
    b_comp = ""
    for i in b:
        if i=="1": b_comp += "0"
        elif i=="0": b_comp += "1"
    b_comp = binary_addition(b_comp,"1")
   return b_comp

def binary_subtraction(a,b):
    """a,b are binary strings"""
    b_complement = complement(b)
    max_len = len(a)
   return binary_addition(a,b_complement.zfill(max_len))

def ALS(a,q):
    """all are binary strings"""
    # x = Q not
    a = a[1:]+q[0]
    q = q[1:]+"_"
   return a,q

def isNegative(bin_number):
   return bin_number[0] == "1"

def nonRestoringDivision(dividend,divisor,count,A="00000"):
```

```
    binary_dividend  = bin(dividend)[2:].zfill(count) if dividend>0 else
bin(dividend)[3:].zfill(count)
    binary_divisor = bin(divisor)[2:].zfill(count+1) if divisor>0 else
bin(dividend)[3:].zfill(count+1)
    print(f"Dividend Q = {binary_dividend}, divisor B = {binary_divisor}, A = {A},
count = {count}\n")
    print("Count\tA\tQ\tOperation")
    print("----------------------------------------")

    while(count!=0):
        print(f"{count}\t{A}\t{binary_dividend}\tbefore ALS")
        A,binary_dividend = ALS(A,binary_dividend)
        case = isNegative(A)

        if(case):
            print(f"{count}\t{A}\t{binary_dividend}\tA <- A+B")
            A = binary_addition(A,binary_divisor)
        else:
            print(f"{count}\t{A}\t{binary_dividend}\tA <- A-B")
            A = binary_subtraction(A,binary_divisor)

        print(f"{count}\t{A}\t{binary_dividend}\tQ. = ~(Amsb)")
        binary_dividend = binary_dividend.replace("_",str(int(not(int(A[0])))))
        print(f"{count}\t{A}\t{binary_dividend}")

        count -= 1
        print("----------------------------------------")

    if isNegative(A):
        A = binary_addition(A,binary_divisor)

    remainder = int(A,base=2)
    quotient = int(binary_dividend,base=2)

    print(f"Remainder = {remainder}, Quotient = {quotient}")

nonRestoringDivision(int(input("Enter Dividend = ")),int(input("Enter Divisor =
")),4)
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\nonRestoringDivision.py"
Enter Dividend = 15
Enter Divisor = 4
Dividend Q = 1111, divisor B = 00100, A = 00000, count = 4

Count   A       Q       Operation
----------------------------------------
4       00000   1111    before ALS
4       00001   111_    A <- A-B
4       11101   111_    Q. = ~(Amsb)
4       11101   1110
----------------------------------------
3       11101   1110    before ALS
3       11011   110_    A <- A+B
3       11111   110_    Q. = ~(Amsb)
3       11111   1100
----------------------------------------
2       11111   1100    before ALS
2       11111   100_    A <- A+B
2       00011   100_    Q. = ~(Amsb)
2       00011   1001
----------------------------------------
1       00011   1001    before ALS
1       00111   001_    A <- A-B
1       00011   001_    Q. = ~(Amsb)
1       00011   0011
----------------------------------------
Remainder = 3, Quotient = 3
PS D:\SEM 5\POA\EXPERIMENTS>
```

Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 3

Jigar Siddhpura
60004210155

POA Experiment 4 : Page Replacement

Aim: To implement page replacement algorithms : FIFO, optimal & LRU.

Theory: Page replacement algorithms are used in virtual memory systems to decide which page to bring in to replace a page in memory. Some algorithms are:

1. FIFO (First In First Out) : It replaces oldest page in memory when new page is to be loaded. It is a simple algorithm that keeps track of orders in which pages were brought in.

2. Optimal : Here, we replace that page that will not be used for longest time in future. It is impractical to implement it in real systems as it requires further knowledge of page accesn.

3. LRU (Least Recently Used) : Here the page that is not being used recently is replaced. Its based on the priciple that recently used pages are more likely to be used in immediate future.

Example :

No. of frames = 3

Page Reference String = 4, 7, 6, 1, 7, 6, 1, 2, 7, 2

FIFO :

4	7	6	1	7	6	1	2	7	2
		6	6	6	6	6	6	7	7
	7	7	7	7	7	7	2	2	2
4	4	4	1	1	1	1	1	1	1
F	F	F	F	H	H	H	F	F	H

Optimal :

4	7	6	1	7	6	1	2	7	2
		6	6	6	6	6	2	2	2
	7	7	7	7	7	7	7	7	7
4	4	4	1	1	1	1	1	1	1
F	F	F	F	H	H	H	F	H	H

LRU :

4	7	6	1	7	6	1	2	7	2
		6	6	6	6	6	6	7	7
	7	7	7	7	7	7	2	2	2
4	4	4	1	1	1	1	1	1	1
F	F	F	F	H	H	H	F	F	H

Observation :

FIFO : Page fault = 6 , Page hit = 4
∴ Hit ratio = 0.4 , Fault ratio = 0.6

Optimal : Page fault = 5 , Page hit = 5
∴ Hit ratio = Fault ratio = 0.5

LRU : Page fault = 6 , Page hit = 4
∴ Hit ratio = 0.4 , Fault ratio = 0.6

Conclusion : Hence, we implement page replacement algorithms using FIFO, LRU & optimal & Calculated page fault, page hit, hit ratio & fault ratio.

FIFO:

```python
allPageRequests = [4,7,6,1,7,6,1,2,7,2]
numPageFrames = 3
pageFaults = 0
pageFrames = []

for page in allPageRequests:
    if page not in pageFrames:
        if len(pageFrames) < numPageFrames:
            pageFrames.append(page)
        else:
            pageFrames = pageFrames[1:] + [page]
        pageFaults+=1
        print(f"Page input {page} : page frames - {pageFrames}, fault")
    else:
        print(f"Page input {page} : page frames - {pageFrames}, hit")

print(f"\nPage Faults = {pageFaults}, Page Hits = {len(allPageRequests)-pageFaults}")
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\fifo.py"
Page input 4 : page frames - [4], fault
Page input 7 : page frames - [4, 7], fault
Page input 6 : page frames - [4, 7, 6], fault
Page input 1 : page frames - [7, 6, 1], fault
Page input 7 : page frames - [7, 6, 1], hit
Page input 6 : page frames - [7, 6, 1], hit
Page input 1 : page frames - [7, 6, 1], hit
Page input 2 : page frames - [6, 1, 2], fault
Page input 7 : page frames - [1, 2, 7], fault
Page input 2 : page frames - [1, 2, 7], hit

Page Faults = 6, Page Hits = 4
PS D:\SEM 5\POA\EXPERIMENTS>
```

LRU:

```python
allPageRequests = [4,7,6,1,7,6,1,2,7,2]
numPageFrames = 3
pageFaults = 0
pageFrames = []

for page in allPageRequests:
    if page not in pageFrames:
        if len(pageFrames) < numPageFrames:
            pageFrames.append(page)
        else:
            pageFrames = pageFrames[1:] + [page]
        pageFaults+=1
        print(f"Page input {page} : page frames - {pageFrames}, fault")
    else:
        pageFrames = pageFrames[1:] + [page]
        print(f"Page input {page} : page frames - {pageFrames}, hit")

print(f"\nPage Faults = {pageFaults}, Page Hits = {len(allPageRequests)-pageFaults}")
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\lru.py"
Page input 4 : page frames - [4], fault
Page input 7 : page frames - [4, 7], fault
Page input 6 : page frames - [4, 7, 6], fault
Page input 1 : page frames - [7, 6, 1], fault
Page input 7 : page frames - [6, 1, 7], hit
Page input 6 : page frames - [1, 7, 6], hit
Page input 1 : page frames - [7, 6, 1], hit
Page input 2 : page frames - [6, 1, 2], fault
Page input 7 : page frames - [1, 2, 7], fault
Page input 2 : page frames - [2, 7, 2], hit

Page Faults = 6, Page Hits = 4
PS D:\SEM 5\POA\EXPERIMENTS>
```

OPTIMAL :

```python
allPageRequests = [4,7,6,1,7,6,1,2,7,2]
numPageFrames = 3
pageFaults = 0
pageFrames = []

def findPageToReplace(allPageRequests, currentIndex, pageFrames):
    indexToReplace = -1
    futurePages = []

    for page in allPageRequests[-1:currentIndex]:
        if page in pageFrames:
            futurePages.append(page)

    if indexToReplace == -1:
        indexToReplace = 0
    elif len(futurePages) < len(pageFrames):
        unused_pages = [page for page in pageFrames if page not in futurePages]
        indexToReplace = pageFrames.index(unused_pages[0])

    return indexToReplace

for i,page in enumerate(allPageRequests):
    if page not in pageFrames:
        if len(pageFrames) < numPageFrames:
            pageFrames.append(page)
        else:
            indexToReplace = findPageToReplace(allPageRequests, i, pageFrames)
            pageFrames[indexToReplace] = page
        pageFaults+=1
        print(f"Page input {page} : page frames - {pageFrames}, fault")
    else:
        print(f"Page input {page} : page frames - {pageFrames}, hit")

print(f"\nPage Faults = {pageFaults}, Page Hits = {len(allPageRequests)-pageFaults}")
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\optimal.py"
Page input 4 : page frames - [4], fault
Page input 7 : page frames - [4, 7], fault
Page input 6 : page frames - [4, 7, 6], fault
Page input 1 : page frames - [1, 7, 6], fault
Page input 7 : page frames - [1, 7, 6], hit
Page input 6 : page frames - [1, 7, 6], hit
Page input 1 : page frames - [1, 7, 6], hit
Page input 2 : page frames - [2, 7, 6], fault
Page input 7 : page frames - [2, 7, 6], hit
Page input 2 : page frames - [2, 7, 6], hit

Page Faults = 5, Page Hits = 5
PS D:\SEM 5\POA\EXPERIMENTS>
```

Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 4

Jigar Siddhpura
60004210155

POA Experiment 3 : Memory allocation

Aim : To implement memory allocation algorithms : best, worst, first fit.

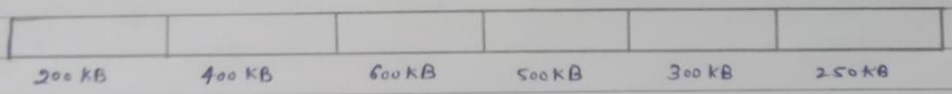
Theory : Memory allocation strategies in computer system refer to method used to assign & ~~may~~ manage memory for processes. Some of the algorithms are

1. Best fit – Here, smallest available block of memory, ~~is used~~, that satisfies the requirement, is used. It may leave small unused gaps.

2. First fit – Here, the first available block, large enough that satisfies the request, is used. It may cause memory fragmentation.

3. Worst fit – Here, largest available block is used. It is less commonly used.

Example : Main memory



Processes : P1 – 357 KB , P2 – 210 KB
P3 – 468 KB , P4 – 491 KB

First fit :

	P_1	P_2	P_3		
200 KB	400 KB	600 KB	500 KB	300 KB	250 KB

Best fit :

	P_1	P_4	P_3		P_2
200 KB	400 KB	600 KB	500 KB	300 KB	250 KB

Worst fit :

		P_1	P_2		
200 KB	400 KB	600 KB	500 KB	300 KB	250 KB

Conclusion : Hence, we have implemented the memory allocation algorithm & allocated the memory for the process with best, first & worst fit strategies.

Best Fit:

```python
bNo = 6
pNo = 4
bSizes = [200,400,600,500,300,250]
pSizes = [357,210,468,491]
flag = [0] * len(bSizes)
allocated = [0] * len(bSizes)
MAX_BLOCK_SIZE = 999999

print(f"Memory Blocks : {bSizes}")
print(f"Processes : {pSizes}\n")

for i,psize in enumerate(pSizes):
    index_placed = -1
    for j,bsize in enumerate(bSizes):
        if((psize <= bsize) & (bsize < MAX_BLOCK_SIZE) & (flag[j] == 0)):
            index_placed = j
            MAX_BLOCK_SIZE = bsize
    if index_placed != -1:
        flag[index_placed] = 1
        allocated[index_placed] = psize
        print(f"Memory allocation for {psize} - {allocated}")
    else:
        print(f"Memory allocation for {psize} - No Space to allocate")
    MAX_BLOCK_SIZE = 999999

print(f"\nFinal Memory Allocation - {allocated}")
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\bestFit.py"
Memory Blocks : [200, 400, 600, 500, 300, 250]
Processes : [357, 210, 468, 491]

Memory allocation for 357 - [0, 357, 0, 0, 0, 0]
Memory allocation for 210 - [0, 357, 0, 0, 0, 210]
Memory allocation for 468 - [0, 357, 0, 468, 0, 210]
Memory allocation for 491 - [0, 357, 491, 468, 0, 210]

Final Memory Allocation - [0, 357, 491, 468, 0, 210]
PS D:\SEM 5\POA\EXPERIMENTS>
```

First Fit:

```python
bNo = 6
pNo = 4
bSizes = [200,400,600,500,300,250]
pSizes = [357,210,468,491]
flag = [0] * len(bSizes)
allocated = [0] * len(bSizes)

print(f"Memory Blocks : {bSizes}")
print(f"Processes : {pSizes}\n")

for i,psize in enumerate(pSizes):
    for j,bsize in enumerate(bSizes):
        if((psize <= bsize) & (flag[j] == 0)):
            flag[j] = 1
            allocated[j] = psize
            print(f"Memory allocation for {psize} - {allocated}")
            break
        elif(j == len(bSizes)-1):
            print(f"Memory allocation for {psize} - No Space to allocate")

print(f"\nFinal Memory Allocation - {allocated}")
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\firstFit.py"
Memory Blocks : [200, 400, 600, 500, 300, 250]
Processes : [357, 210, 468, 491]

Memory allocation for 357 - [0, 357, 0, 0, 0, 0]
Memory allocation for 210 - [0, 357, 210, 0, 0, 0]
Memory allocation for 468 - [0, 357, 210, 468, 0, 0]
Memory allocation for 491 - No Space to allocate

Final Memory Allocation - [0, 357, 210, 468, 0, 0]
PS D:\SEM 5\POA\EXPERIMENTS>
```

Worst Fit :

```
bNo = 6
pNo = 4
bSizes = [200,400,600,500,300,250]
pSizes = [357,210,468,491]
flag = [0] * len(bSizes)
allocated = [0] * len(bSizes)
MIN_BLOCK_SIZE = -10

print(f"Memory Blocks : {bSizes}")
print(f"Processes : {pSizes}\n")

def getMaxMemoryIndex(psize,memory, flag):
    """
    Returns memory index with max capacity given that it is unallocated.

    Parameters:
    - psize (int): integer representing the process size
    - memory (list): List representing memory capacities.
    - flag (list): List representing the allocation status (0 for unallocated, 1 for allocated).

    Returns:
    - int: Index of the unallocated memory with the maximum capacity but less than process size.
    """
    max_capacity = -1
    max_index = None
    for i in range(len(memory)):
        if flag[i] == 0 and memory[i] >= psize and memory[i]>max_capacity:
            max_capacity = memory[i]
            max_index = I
    return max_index

for i,psize in enumerate(pSizes):
    index_placed = -1
    allocated_index = getMaxMemoryIndex(psize,bSizes,flag)
    if allocated_index:
        flag[allocated_index] = 1
        allocated[allocated_index] = psize
        print(f"Memory allocation for {psize} - {allocated}")
    else:
        print(f"Memory allocation for {psize} - No Space to allocate")

print(f"\nFinal Memory Allocation - {allocated}")
```

Output :

```
PS D:\SEM 5\POA\EXPERIMENTS> python -u "d:\SEM 5\POA\EXPERIMENTS\worstFit.py"
Memory Blocks : [200, 400, 600, 500, 300, 250]
Processes : [357, 210, 468, 491]

Memory allocation for 357 - [0, 0, 357, 0, 0, 0]
Memory allocation for 210 - [0, 0, 357, 210, 0, 0]
Memory allocation for 468 - No Space to allocate
Memory allocation for 491 - No Space to allocate

Final Memory Allocation - [0, 0, 357, 210, 0, 0]
PS D:\SEM 5\POA\EXPERIMENTS>
```

Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 5

Jigar Siddhpura
60004210155

Experiment 5: Addition & Subtraction in 8086

Aim: To implement assembly program for 16-bit addition / subtraction using direct, immediate & Register addressing mode.

Theory: 8086 Microprocessor works on instructions in assembly language (a low level language). Here, with the help of 8086, we perform arithmetic operations using various addressing mode.

Addition: First, we used direct addressing mode to add 2 numbers at location [1000h] & [1002h], the result is stored in [1004h]. The carry flag is examined & if set then it's stored in [1006h]. Here instruction – MOV, ADD, INC, JNC, HLT are used.

Secondly we used immediate addressing mode where we add 2 immediate values – 0ffffh & 00099h. The result is stored in DX using register addressing mode. Carry flag is checked & carry count is stored in [1006h] using direct addressing.

Subtraction: Thirdly, we perform subtraction using direct addressing, so 2 no.s from memory location [1000h] & [1002h] are subtracted & result is stored in register DX using register addressing mode. Borrow flag is examined to determine if borrowing occured during subtraction. If so, borrow count

is stored in [1006h].

Lastly, immediate addressing is used for subtraction i.e. immediate values are subtracted & result is stored in register DX. Again, if borrow exists, it's stored in [1006h].

For subtraction, instructions are same as addition, except we have SUB in replacement of ADD.

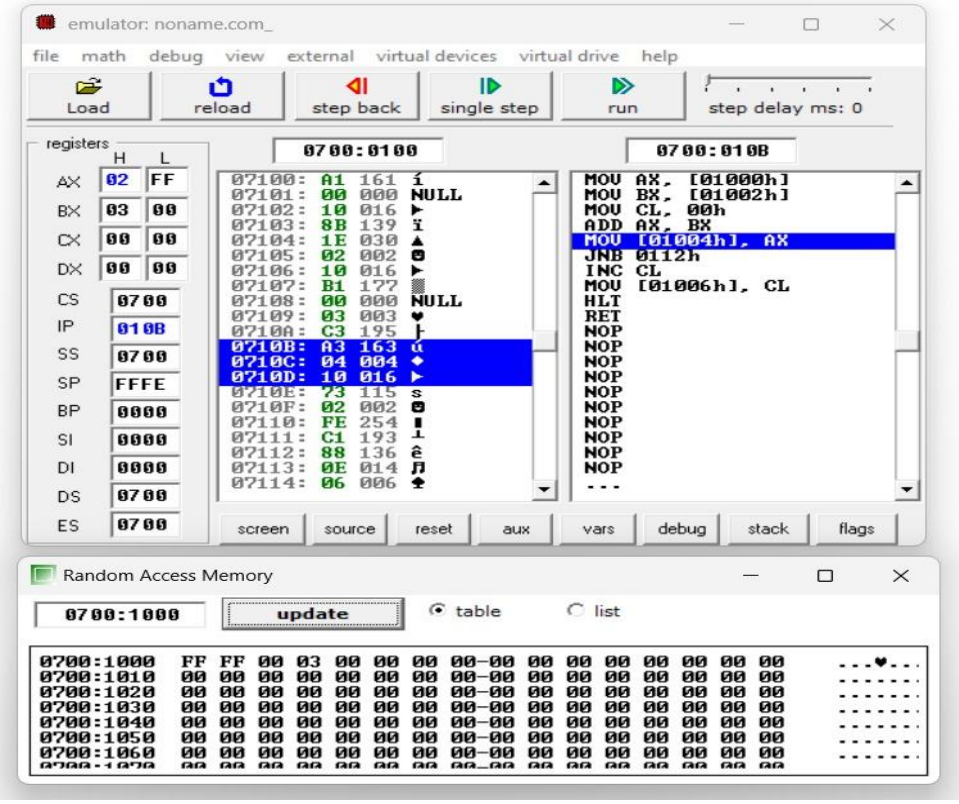
Conclusion: So we used various addressing modes in 8086 to perform arithmetic operations (addition / subtraction), where direct addressing mode simplifies access, while immediate proves valuable for handling constants.

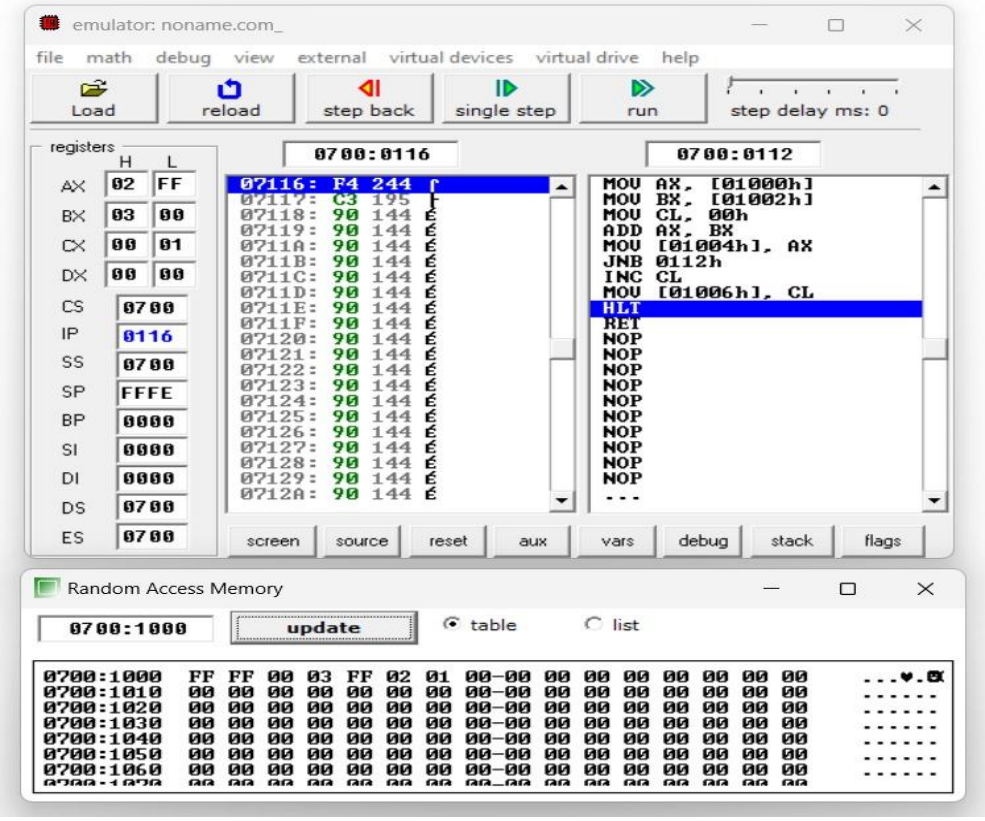
Code :

1. **Addition using Direct Addressing mode** :

```
org 100h

MOV AX, [1000h]
MOV BX, [1002h]
MOV CL,00h
ADD AX,BX
MOV [1004h],AX
JNC carry
INC CL
carry:
MOV [1006h],CL
HLT
ret
```

- Addition of ffffh + 0900h which gives output has 08ff with carry has 01 where sum can be seen at addresses 1005h,1004h and carry at 1006h

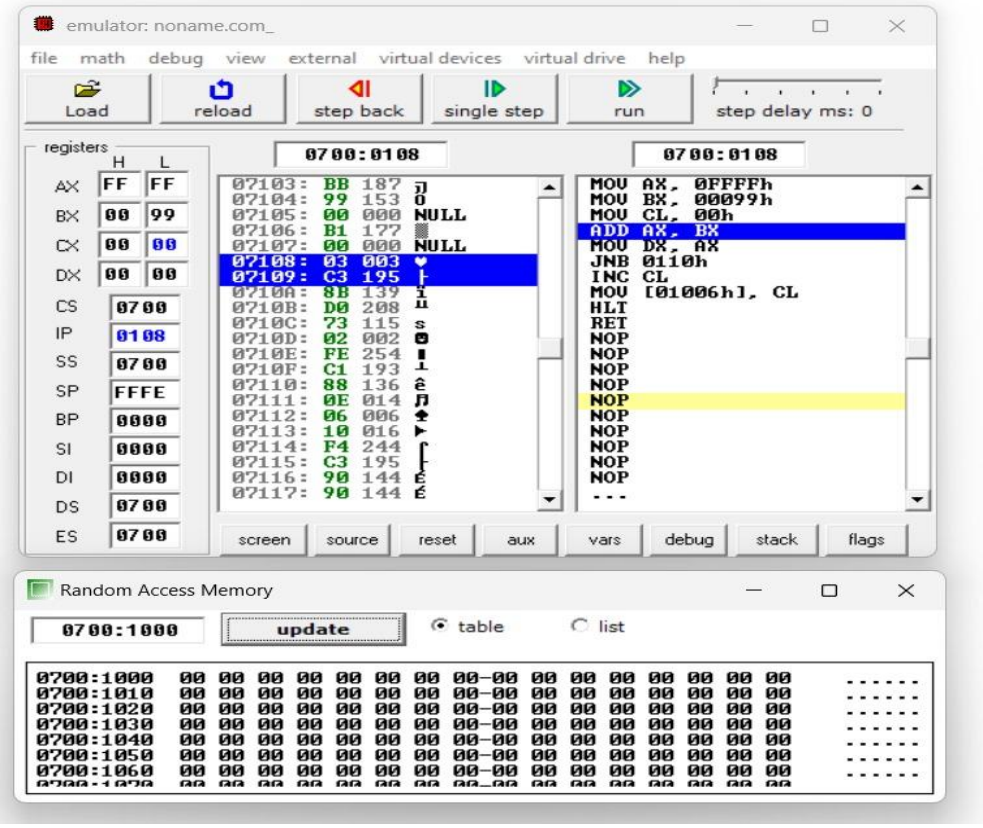


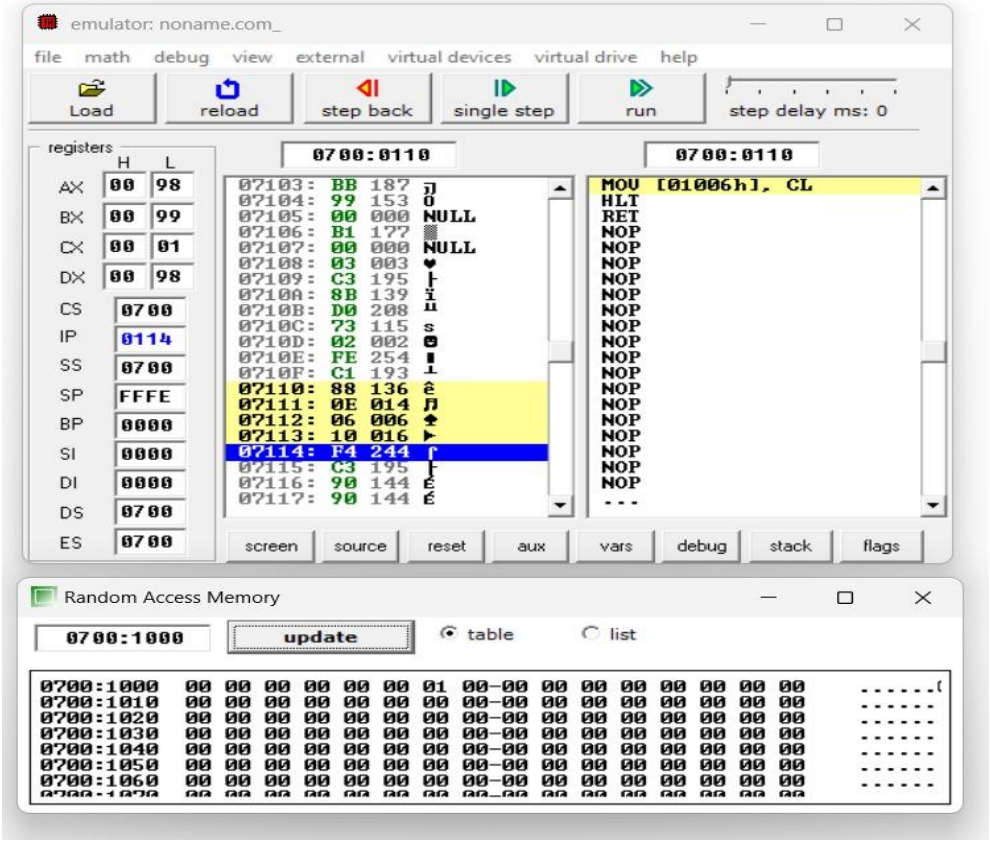


2. **Addition using Immediate Addressing mode** :

```
org 100h

MOV AX, 0ffffh
MOV BX, 00099h
MOV CL,00h
ADD AX,BX
MOV DX,AX
JNC carry
INC CL
carry:
MOV [1006h],CL
HLT
ret
```

- Addition of ffffh + 0099h which gives output has 0098 with carry has 01 where sum can be seen at register DX and carry at 1006h

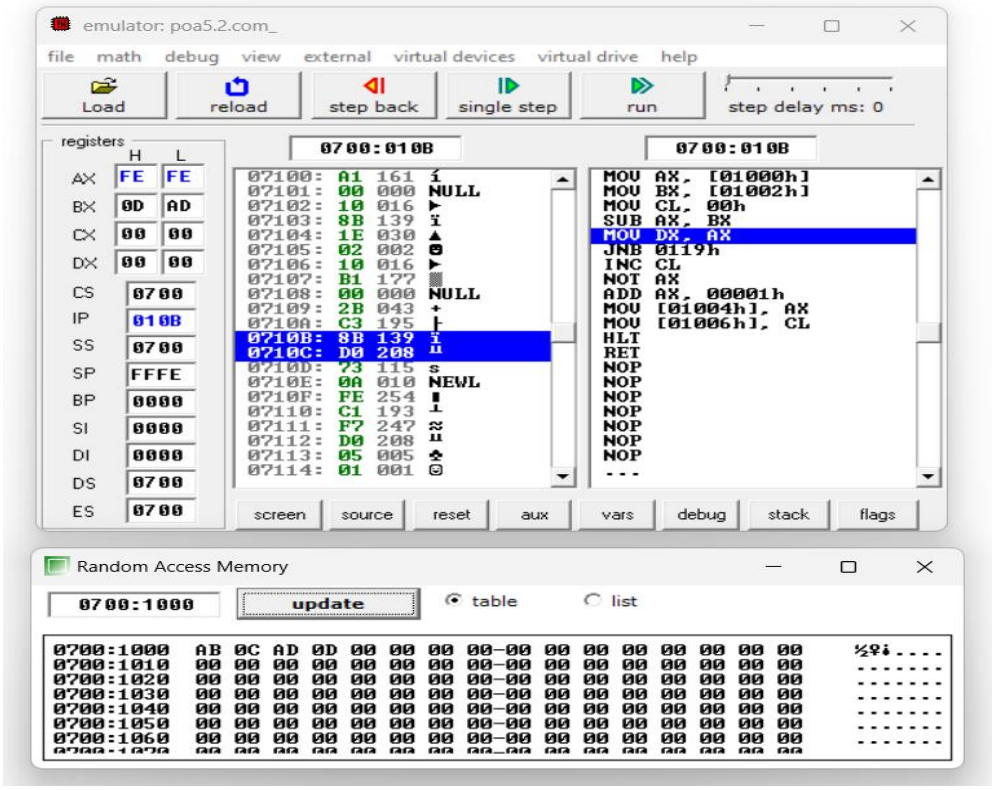


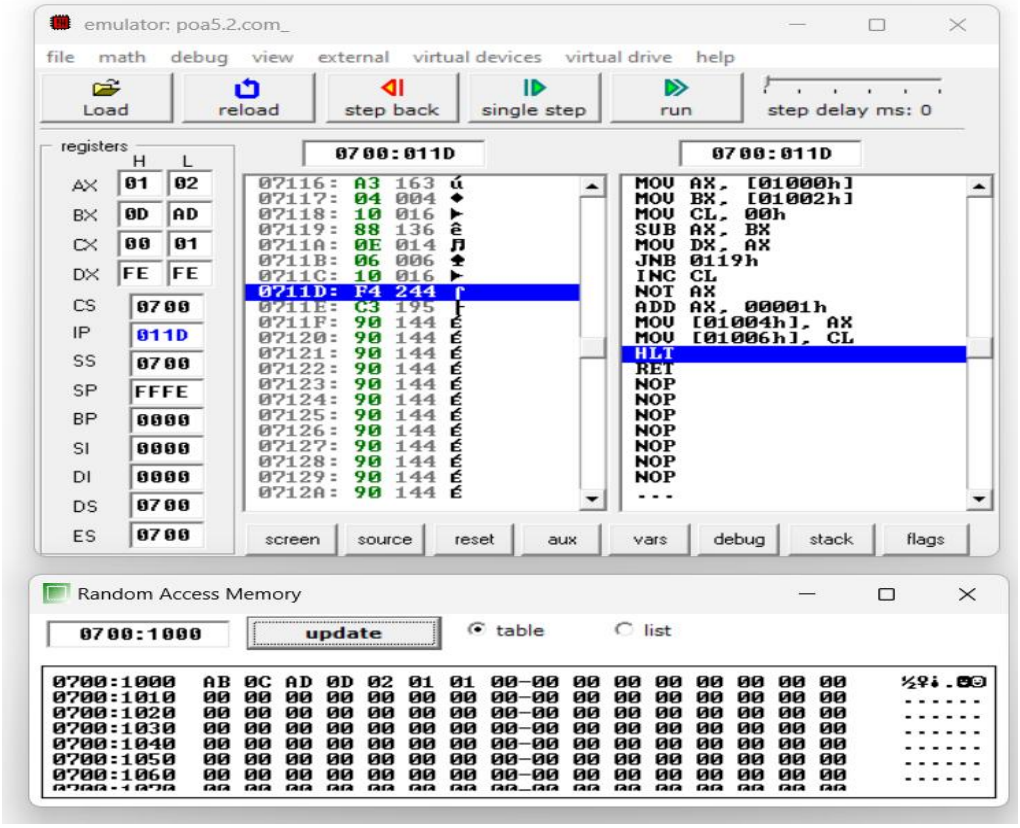


3. **Subtraction using Direct Addressing mode** :

```
org 100h

MOV AX, 0ffffh
MOV BX, 00099h
MOV CL,00h
SUB AX,BX
MOV DX,AX
JNC borrow
INC CL
NOT AX
ADD AX,0001h
MOV [1004h],AX
borrow:
MOV [1006h],CL
HLT
ret
```

- Subtraction of CABh - DADh which gives -102 has the answer which can be seen at address 1004h,1003h



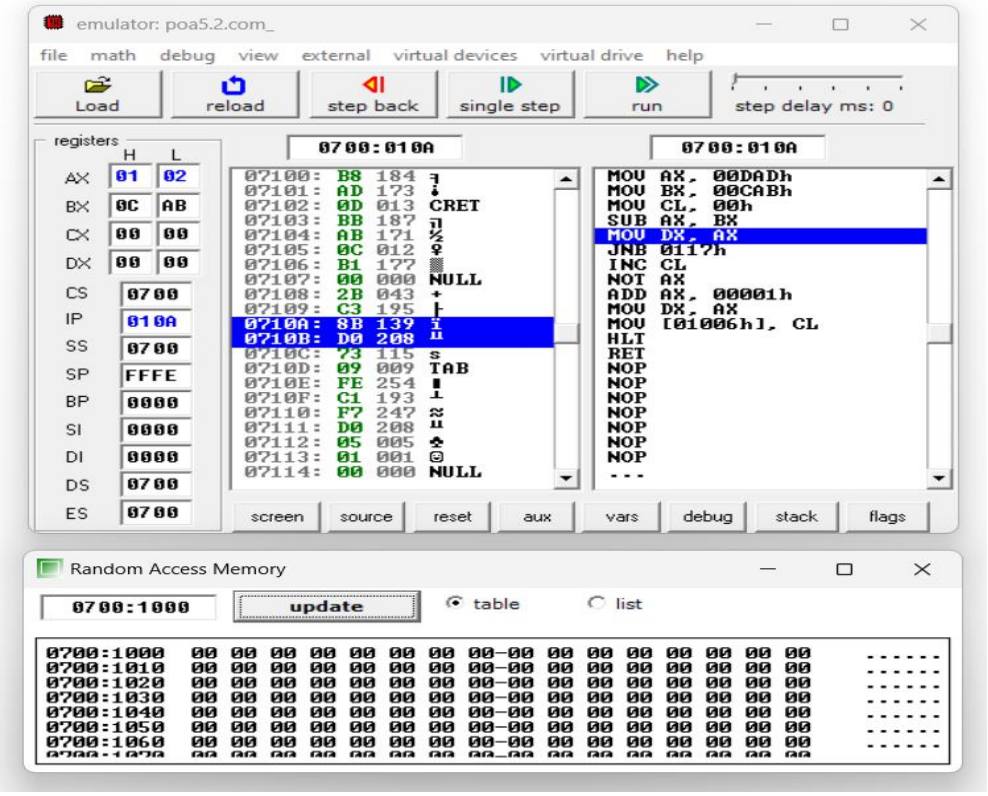


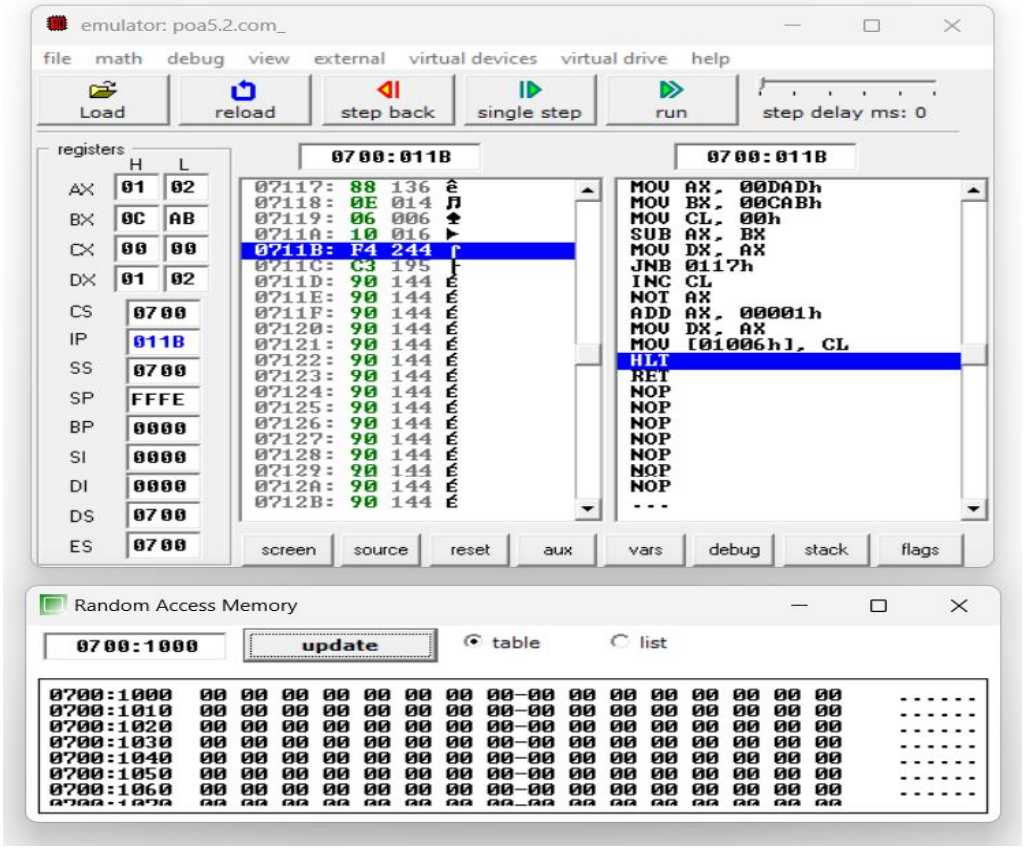
4. **Subtraction using Immediate Addressing mode** :

```
org 100h
```

```
MOV AX, 0dadH
MOV BX, 0cabH
MOV CL,00h
SUB AX,BX
MOV DX,AX
JNC borrow
INC CL
NOT AX
ADD AX,0001h
MOV DX,AX
borrow:
MOV [1006h],CL
HLT
ret
```

- Subtraction of DADh - CABh which gives 102 has the answer which can be seen at register DX




emulator: poa5.2.com_
file math debug view external virtual devices virtual drive help
Load
reload
step back
single step
run
step delay ms: 0
registers
H L
AX 01 02
BX 0C AB
CX 00 00
DX 01 02
CS 0700
IP 011B
SS 0700
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0700
ES 0700
0700:011B
07117: 88 136 ê
07118: 0E 014 ♫
07119: 06 006 ♠
0711A: 10 016 ►
0711B: F4 244 ⌠
0711C: C3 195 ├
0711D: 90 144 É
0711E: 90 144 É
0711F: 90 144 É
07120: 90 144 É
07121: 90 144 É
07122: 90 144 É
07123: 90 144 É
07124: 90 144 É
07125: 90 144 É
07126: 90 144 É
07127: 90 144 É
07128: 90 144 É
07129: 90 144 É
0712A: 90 144 É
0712B: 90 144 É
0700:011B
MOV AX, 00DADh
MOV BX, 00CABh
MOV CL, 00h
SUB AX, BX
MOV DX, AX
JNB 0117h
INC CL
NOT AX
ADD AX, 00001h
MOV DX, AX
MOV [01006h], CL
HLT
RET
NOP
NOP
NOP
NOP
NOP
NOP
NOP
---
screen source reset aux vars debug stack flags
Random Access Memory
0700:1000
update
table
list
0700:1000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ------
0700:1010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ------
0700:1020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ------
0700:1030 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ------
0700:1040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ------
0700:1050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ------
0700:1060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ------

Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 6

60004210155
Jigar Siddhpura

Experiment 6 : Sorting Numbers in 8086

Aim : Implement program to sort numbers in ascending / descending order.

Theory :

Bubble sort is implement in 8086 microprocessor to arrange an array of values. Here, element are compared to next adjacent element and swaped if they are in wrong order; until the array is sorted. The assembly code shows both ascending & descending sorting.

Ascending :

Here, array (segment) is defined in DATA segment & iterated through using loops. Outer loop 'L1' iterates through array & inner loop 'L2' is used for comparison. If current element is greater than next element, swapping is performed using XCHG instruction. This continues until entire array is sorted.

Descending : Here

Here, the code is almost same as above, however the primary difference is on Conditional Jump instructions. Here, swap is performed if current element is less

than the next.

Here instructions used are – MOV, LEA (Load effective address), DEC, CMP (compare), JNZ, JC, XCHG (swap operation)

Here, code segment contains instructions whereas data segment is used to define array. In code, DATA refers to 0710h address, CH refers to counter, LEA SI, N1 stores starting address of N1. Under the hood, CMP performs subtraction to detect the greater element. JC means Jump if Carry i.e. if carry flag is set, jump to particular procedure. JNZ means Jump if not zero i.e. jump if flag is non-zero.

Conclusion: Here, we perform sorting in both possible orders on 8086 microprocessor using bubble sort with the help of instruction set of 8086.

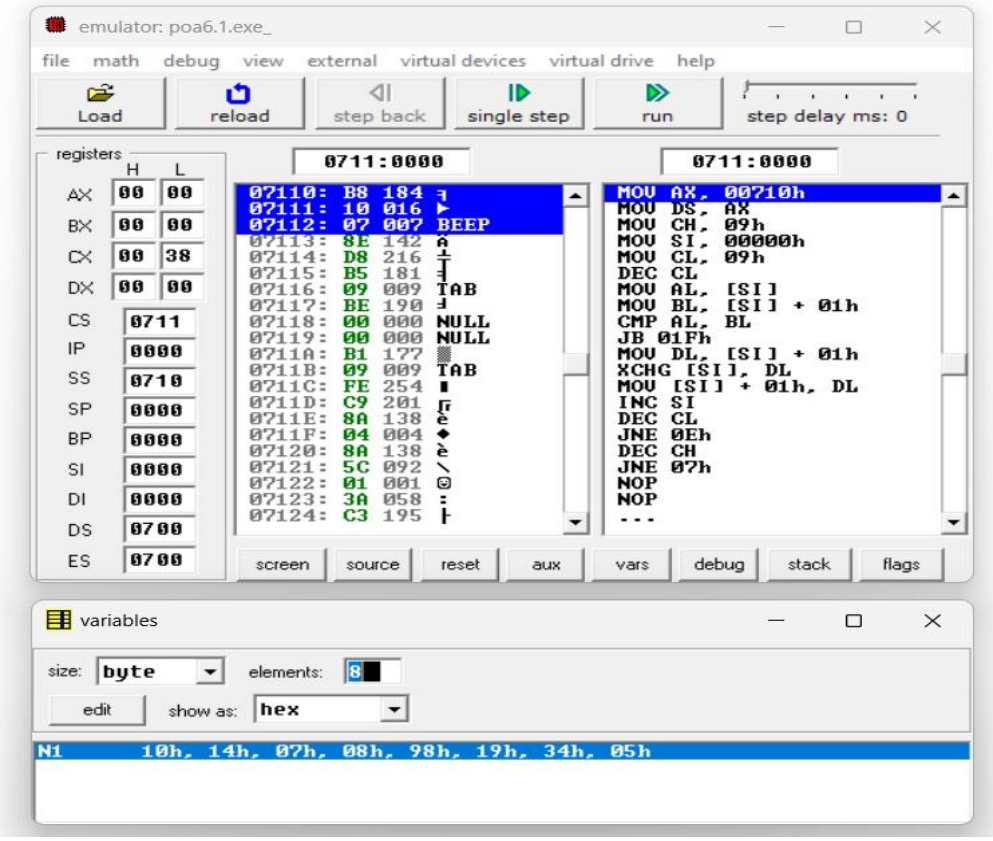
Code :

5. **Sorting in Ascending Order** :

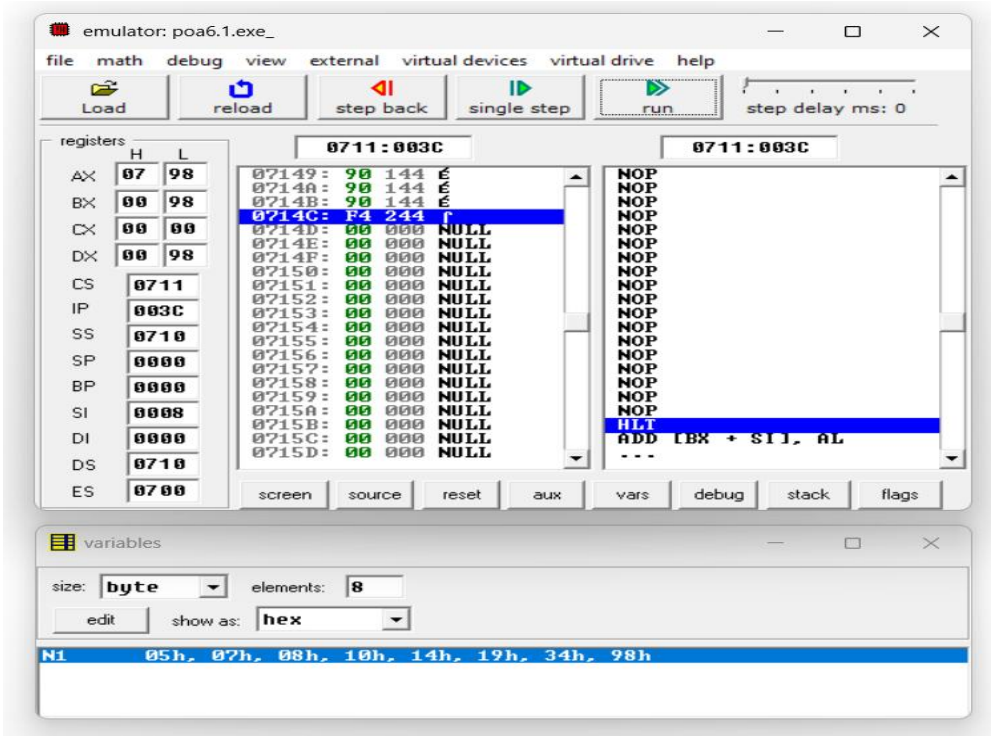
```
DATA SEGMENT
      N1 DB 10h,14h,7h,8h,98h,19h,34h,5h, 98h
      SIZE equ $ - N1
ENDS


CODE SEGMENT
      START:
      MOV AX,DATA
      MOV DS,AX

      MOV CH,SIZE

      L1:
      LEA SI,N1
      MOV CL,
      SIZE DEC CL

      L2:
      MOV AL, [SI]
      MOV BL, [SI+1]
      CMP AL,BL
      JC DOWN
      MOV DL,[SI+1]
      XCHG [SI],DL
      MOV [SI+1],DL

      DOWN:
      INC SI
      DEC CL
      JNZ L2

      DEC CH
      JNZ L1

CODE ENDS
END START
```

- Here the elements are sorted in ascending order in N1 data segment

Before Sorting -



After sorting -



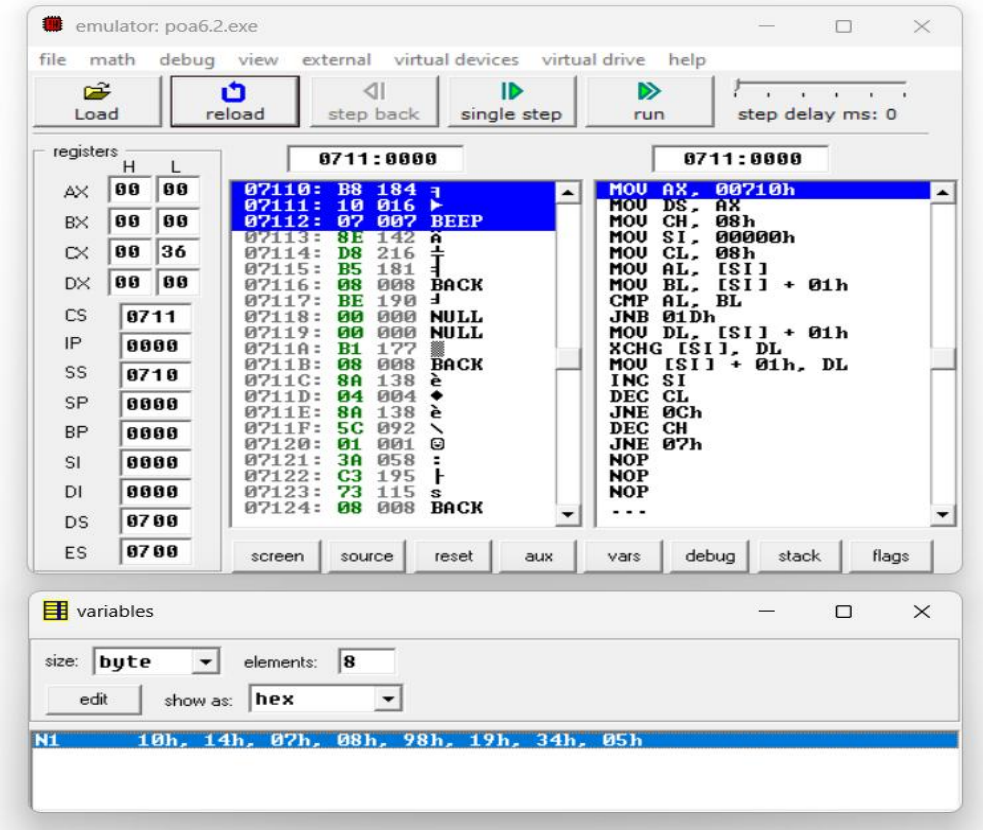
Code :

6. **Sorting in Descending Order** :

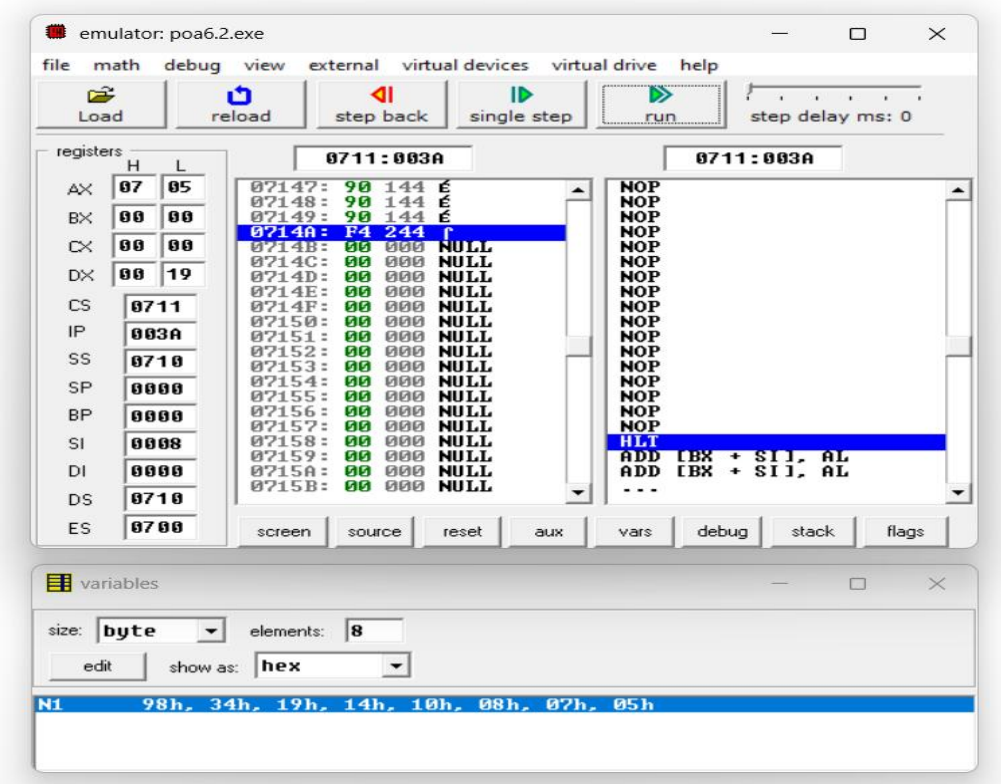
```
DATA SEGMENT
    N1 DB 10h,14h,7h,8h,98h,19h,34h,5h
    SIZE equ $ - N1
ENDS
CODE SEGMENT
    START:
    MOV AX,DATA
    MOV DS,AX

    MOV CH,SIZE

    L1:
    LEA SI,N1
    MOV CL,SIZE

    L2:
    MOV AL, [SI]
    MOV BL, [SI+1]
    CMP AL,BL
    JNC DOWN
    MOV DL,[SI+1]
    XCHG [SI],DL
    MOV [SI+1],DL

    DOWN:
    INC SI
    DEC CL
    JNZ L2
    DEC CH
    JNZ L1

CODE ENDS
END START
ret
```

- Here the elements are sorted in descending order in N1 data segment

Before Sorting -



After sorting -



Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 7

Jigar Siddhpura
60004210155

Experiment 7: Transfer 'n' blocks of data from segments

Aim: Implement assembly program to transfer 'n' blocks of data segment from 1 block to other.

Theory:

Code is written to tranfer 'n' blocks of data from Seg 1 to seg 2 (here 3 bytes of data). First section contains data & extra segment where seg 1 & 2 are defined respectively. In code segment, firstly actual execution is done where firstly data & extra segment are loaded i.e. DS & ES, then loading source & destination indices. Later, tranfer loop is carried on where

→ MOV AH, DS:[SI] means Loading byte from source into AH register, where SI acts as offset & gets added with base DS.

→ MOV ES:[DI], AH is same as above where DI acts as offset. The loop counter (CX) ensures that specified no. of bytes is transferred.

→ INT 3 is an interrupt.

Conclusion:

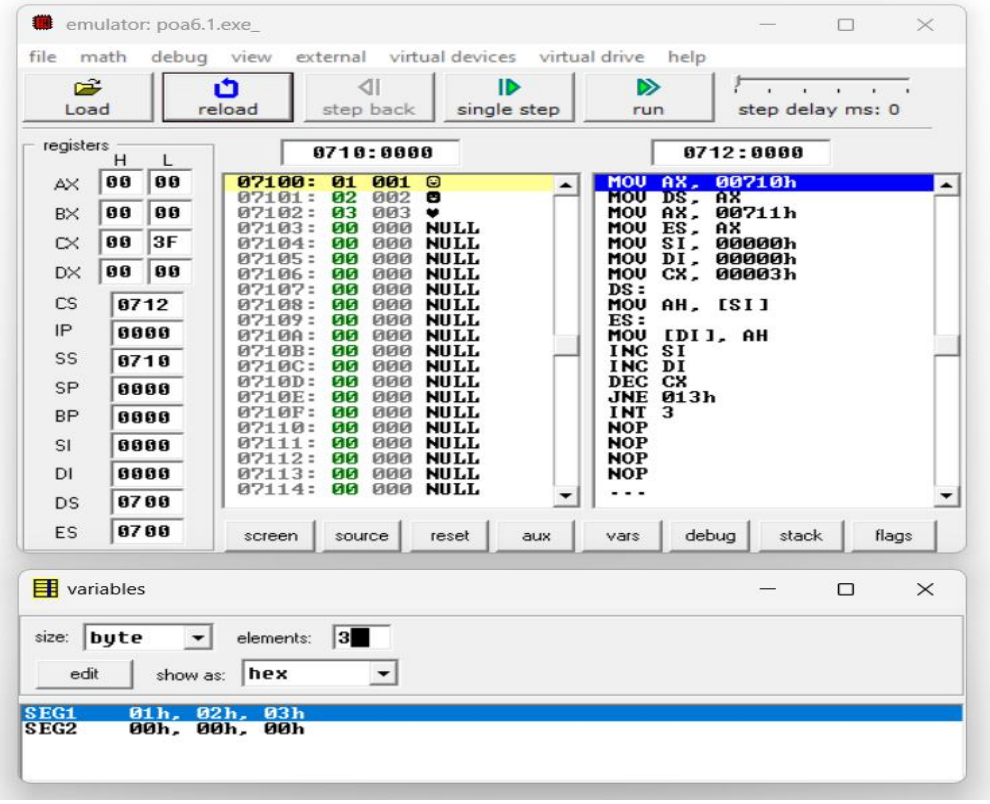
We implemented data transfer from seg 1 to 2 using assembly language on 8086 microprocessor.

Code :

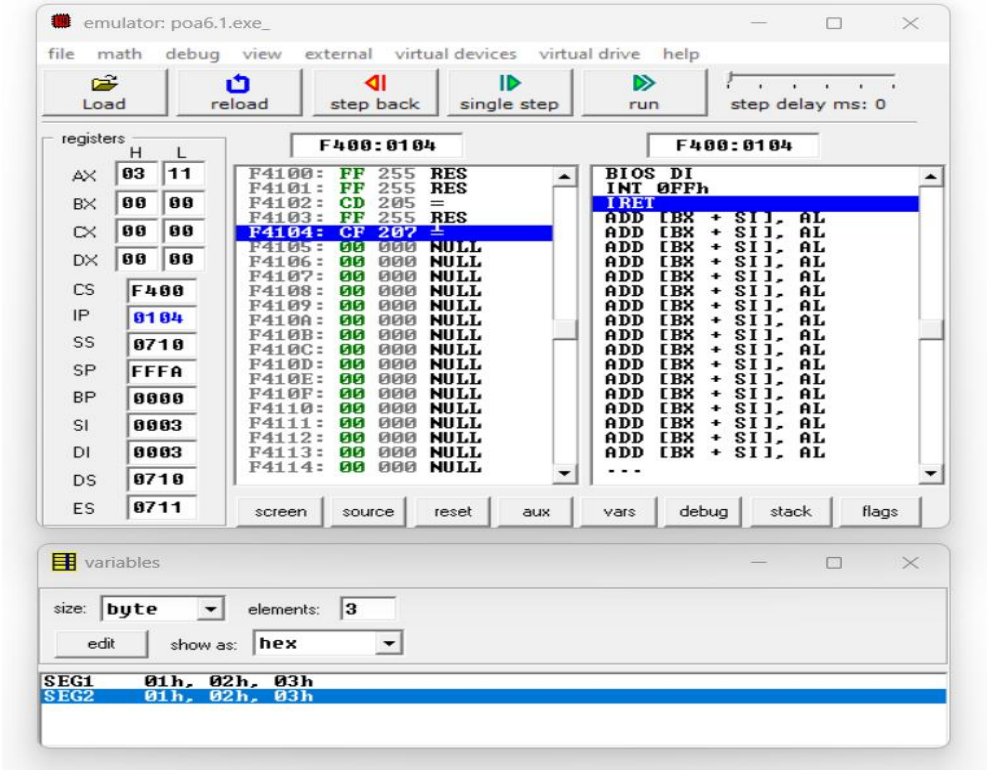
```
DATA SEGMENT
      SEG1 DB 1H, 2H, 3H
DATA ENDS

EXTRA SEGMENT
    SEG2 DB ?
EXTRA ENDS

CODE SEGMENT
      START:
      MOV AX,DATA
      MOV DS,AX
      MOV AX,EXTRA
      MOV ES,AX
      LEA SI, SEG1
      LEA DI, SEG2
      MOV CX,03H

      L1:
      MOV AH,DS:[SI]
      MOV ES:[DI],AH
      INC SI
      INC DI
      DEC CX
      JNZ L1
      INT 3

ENDS
END START
```

Output :

- Before (seg1 has blocks and they are to transferred to seg2) -



- After (blocks are transferred to seg2) -


emulator: poa6.1.exe_
file math debug view external virtual devices virtual drive help
Load
reload
step back
single step
run
step delay ms: 0
registers
H L
AX 03 11
BX 00 00
CX 00 00
DX 00 00
CS F400
IP 0104
SS 0710
SP FFFA
BP 0000
SI 0003
DI 0003
DS 0710
ES 0711
F400:0104
F4100: FF 255 RES
F4101: FF 255 RES
F4102: CD 205 =
F4103: FF 255 RES
F4104: CF 207 ±
F4105: 00 000 NULL
F4106: 00 000 NULL
F4107: 00 000 NULL
F4108: 00 000 NULL
F4109: 00 000 NULL
F410A: 00 000 NULL
F410B: 00 000 NULL
F410C: 00 000 NULL
F410D: 00 000 NULL
F410E: 00 000 NULL
F410F: 00 000 NULL
F4110: 00 000 NULL
F4111: 00 000 NULL
F4112: 00 000 NULL
F4113: 00 000 NULL
F4114: 00 000 NULL
F400:0104
BIOS DI
INT 0FFh
IRET
ADD [BX + SI], AL
---
screen source reset aux vars debug stack flags
variables
size: byte
elements: 3
edit
show as: hex
SEG1 01h, 02h, 03h
SEG2 01h, 02h, 03h

Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 8

60004210155
Jigar Siddhpura

Experiment 8 : Maximum / Minimum no. in Array

Aim : Implement max / min. no. from a array in assembly program

Theory : 1. Experiment is about finding max & min. no. from an array.

2. For which ALP program utilizes the REPEAT loop instr. to iterate through each element of the array, comparing current element with max & min. element previously determined.

3. Firstly the declaration & initialization of array named ARR with set of integers LEN, MIN, MAX.

4. CHECK MAX is part of conditional logic.

5. After comparing the current element with min. value, it checks whether it is greater than max value.

6. If yes, program jumps to CHECK MAX to update max value ensuring both min. & max. are updated in each iteration properly.

7. If current element is smaller than ~~current~~, min. recorded value & is greater than ~~gw~~ current max. ; max value is updated.

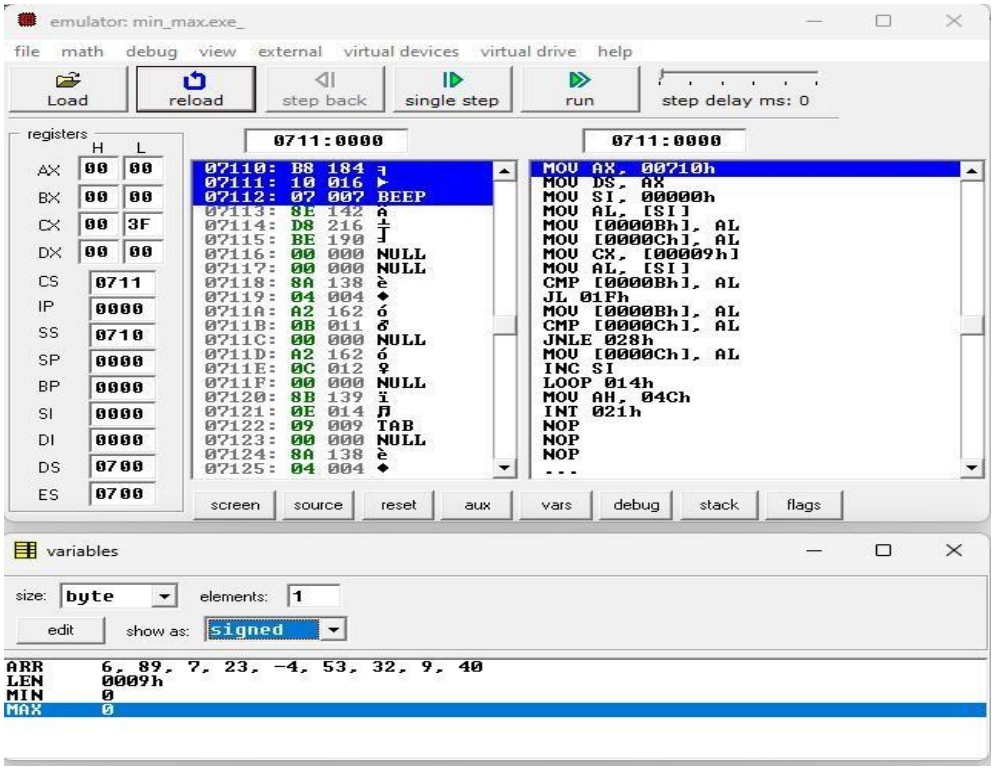
8. Here MOV, LEA, CMP, JL, JG (Jump if less & Jump if greater), LOOP, INT 21 H instr are used

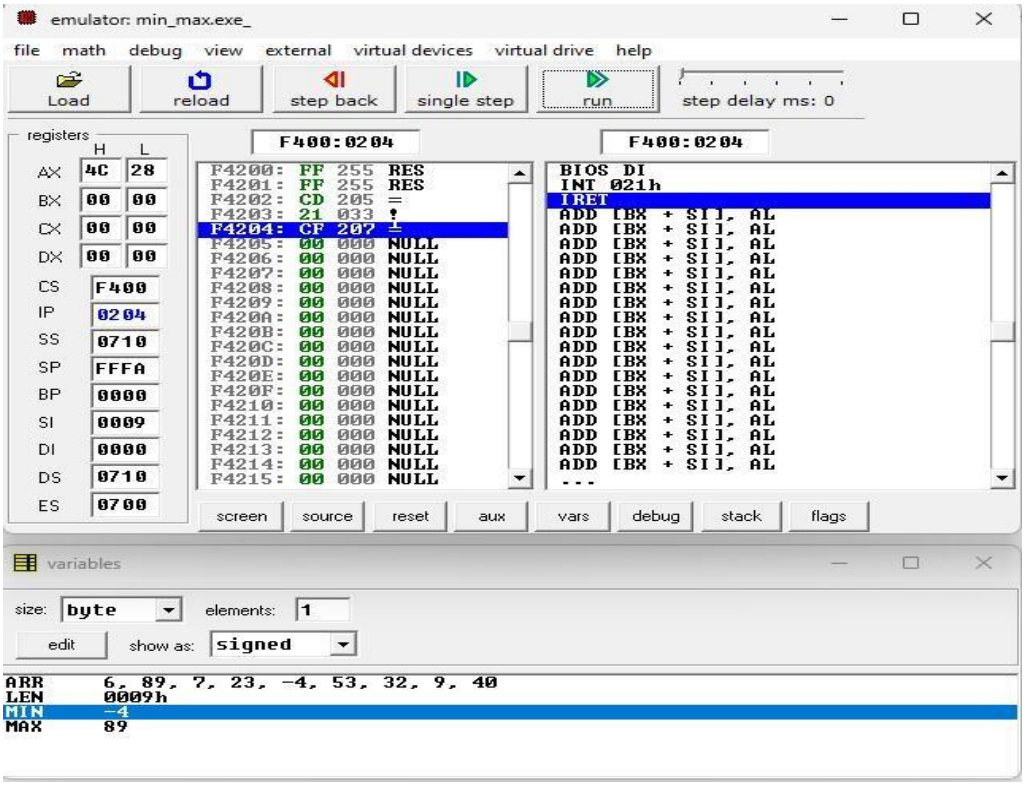
Conclusion : Hence, we implemented assembly program to find max. & min. from an array.

Code :

```
org 100h

DATA SEGMENT
      ARR DB 6,89,7,23,-4,53,32,9,40
      LEN DW SI $-ARR
      MIN DB ?
      MAX DB ?
      DATA ENDS

CODE SEGMENT
      START:
      MOV AX,DATA
      MOV DS,AX
      LEA SI,ARR
      MOV AL,ARR[SI]
      MOV MIN,AL
      MOV MAX,AL
      MOV CX,LEN

      REPEAT:
      MOV AL,ARR[SI]
      CMP MIN,AL
      JL CHECKMAX
      MOV MIN,AL
      CHECKMAX:
      CMP MAX,AL
      JG DONE
      MOV MAX,AL
      DONE:
      INC SI
      LOOP REPEAT


      MOV AH,4CH
      INT 21H

      CODE ENDS
END START

Ret
```

Output :


emulator: min_max.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
H L
AX 00 00
BX 00 00
CX 00 3F
DX 00 00
CS 0711
IP 0000
SS 0710
SP 0000
BP 0000
SI 0000
DI 0000
DS 0700
ES 0700
0711:0000
0711:0000
07110: B8 184
07111: 10 016
07112: 07 007 BEEP
07113: 8E 142
07114: D8 216
07115: BE 190
07116: 00 000 NULL
07117: 00 000 NULL
07118: 8A 138
07119: 04 004
0711A: A2 162
0711B: 0B 011
0711C: 00 000 NULL
0711D: A2 162
0711E: 0C 012
0711F: 00 000 NULL
07120: 8B 139
07121: 0E 014
07122: 09 009 TAB
07123: 00 000 NULL
07124: 8A 138
07125: 04 004
MOV AX, 00710h
MOV DS, AX
MOV SI, 00000h
MOV AL, [SI]
MOV [0000Bh], AL
MOV [0000Ch], AL
MOV CX, [00009h]
MOV AL, [SI]
CMP [0000Bh], AL
JL 01Fh
MOV [0000Bh], AL
CMP [0000Ch], AL
JNLE 028h
MOV [0000Ch], AL
INC SI
LOOP 014h
MOV AH, 04Ch
INT 021h
NOP
NOP
NOP
screen source reset aux vars debug stack flags
variables
size: byte elements: 1
edit show as: signed
ARR 6, 89, 7, 23, -4, 53, 32, 9, 40
LEN 0009h
MIN 0
MAX 0


emulator: min_max.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
H L
AX 4C 28
BX 00 00
CX 00 00
DX 00 00
CS F400
IP 0204
SS 0710
SP FFFA
BP 0000
SI 0009
DI 0000
DS 0710
ES 0700
F400:0204
F400:0204
F4200: FF 255 RES
F4201: FF 255 RES
F4202: CD 205 =
F4203: 21 033 !
F4204: CF 207 ╧
F4205: 00 000 NULL
F4206: 00 000 NULL
F4207: 00 000 NULL
F4208: 00 000 NULL
F4209: 00 000 NULL
F420A: 00 000 NULL
F420B: 00 000 NULL
F420C: 00 000 NULL
F420D: 00 000 NULL
F420E: 00 000 NULL
F420F: 00 000 NULL
F4210: 00 000 NULL
F4211: 00 000 NULL
F4212: 00 000 NULL
F4213: 00 000 NULL
F4214: 00 000 NULL
F4215: 00 000 NULL
BIOS DI
INT 021h
IRET
ADD [BX + SI], AL
screen source reset aux vars debug stack flags
variables
size: byte elements: 1
edit show as: signed
ARR 6, 89, 7, 23, -4, 53, 32, 9, 40
LEN 0009h
MIN -4
MAX 89

Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 9

Jigar Siddhpura
60004210155

POA Experiment 9 : Factorial

Aim : Implement assembly program to find factorial with or without macros.

Theory : 1. In assembly program, concept of macros offer a powerful tool for code abstraction & simplification.
2. First implementation utilizes, traditional language program initializes a data segment with single variable 'A', the no. of which factorial is to be calculated.
3. The program iteratively ~~dec~~ decrements 'A' by 1 & multiplies it with the value in 'A' ~~unit~~ until 'A' equals 1.
4. Later storing it in 'FACT' variable. This approach relies on explicit instr & repetitive code

Macros : 1. Macros in ALP acts as a preprocessor directive that allows programmer to define reusable code segments.
2. They maintain code reusability, readability & reduce redundancy by encapsulating repitive instr. into single named entity.

→ Second program utilizes MACRO instruction to find factorial using FACT label.
It takes a single argument 'F' representing the no. for which factorial is to be calculated.
It encapsulates the iterative factorial calculation process.
This implementation initializes a data segment with number 'num' & utilizes 'FACT' macro to calculate factorial storing result in RESULT variable.

1. By using MACRO, it reduces the no. of explicit instructions enhancing code readability, making it easier to understant & maintain.
2. The without macro implementation, while achieving same result involves more manual instr.; leading to code duplication & chaces of errors.

Conclusions :

Hence, we implemented both programs, with & without macros to find factorial of a no. & understood the use of macros & how it is beneficial for code org., readability & maintainibity.

Code :

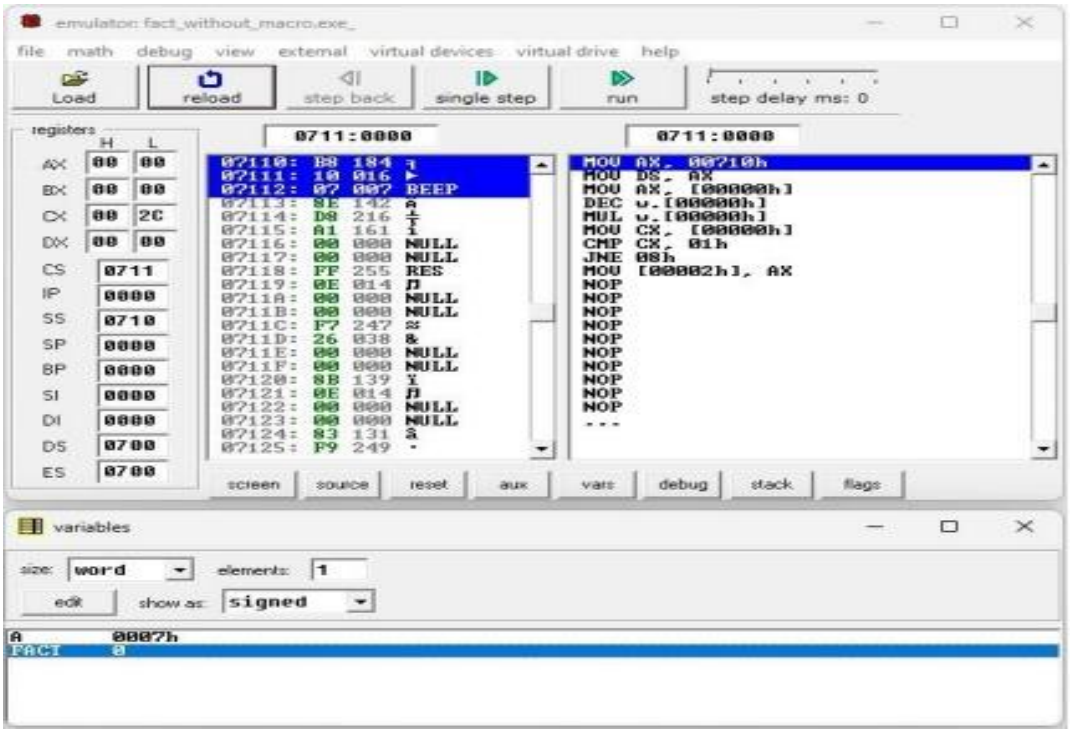
Factorial of a number without using macro instruction

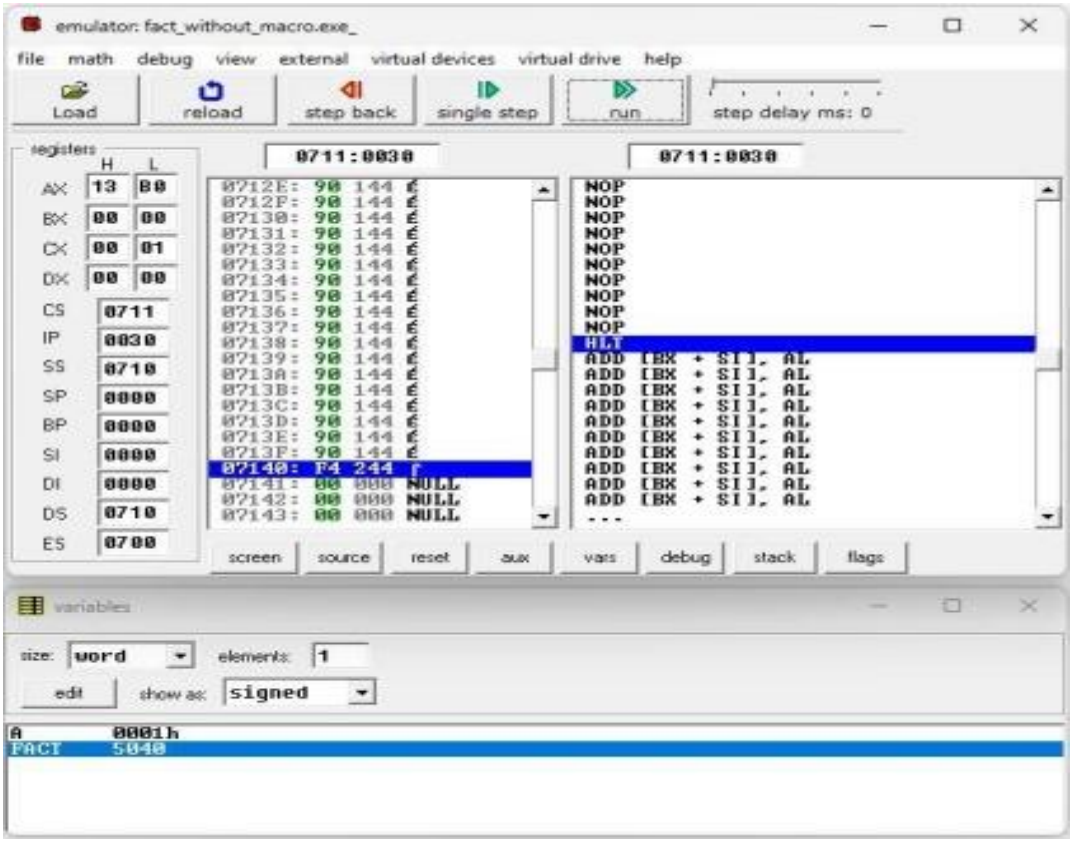
```
org 100h

DATA SEGMENT
    A DW 7
    FACT DW ?
    DATA ENDS
CODE SEGMENT
    START:
    MOV AX,DATA
    MOV DS,AX
    MOV AX,A
    L1:
    DEC A
    MUL A
    MOV CX,A
    CMP CX,01
    JNZ L1
    MOV FACT,AX
    CODE ENDS
END START

Ret
```

Factorial of a number using macro instruction :

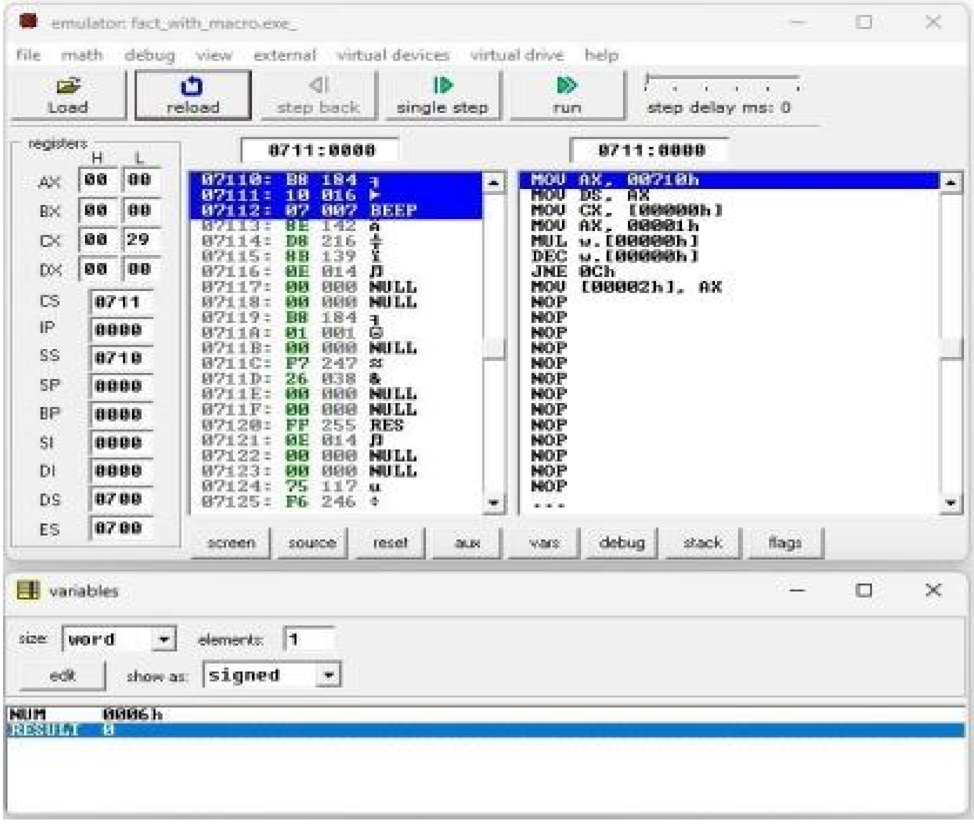
```
FACT MACRO F
    UP:
    MUL F
    DEC F
    JNZ UP
ENDM
    DATA SEGMENT
    NUM DW 06H
    RESULT DW ?
ENDS
    CODE SEGMENT
    START:
    MOV AX, DATA
    MOV DS,AX
    MOV CX,NUM
    MOV AX,0001H
    FACT NUM
    MOV RESULT,AX

ENDS
END START
```

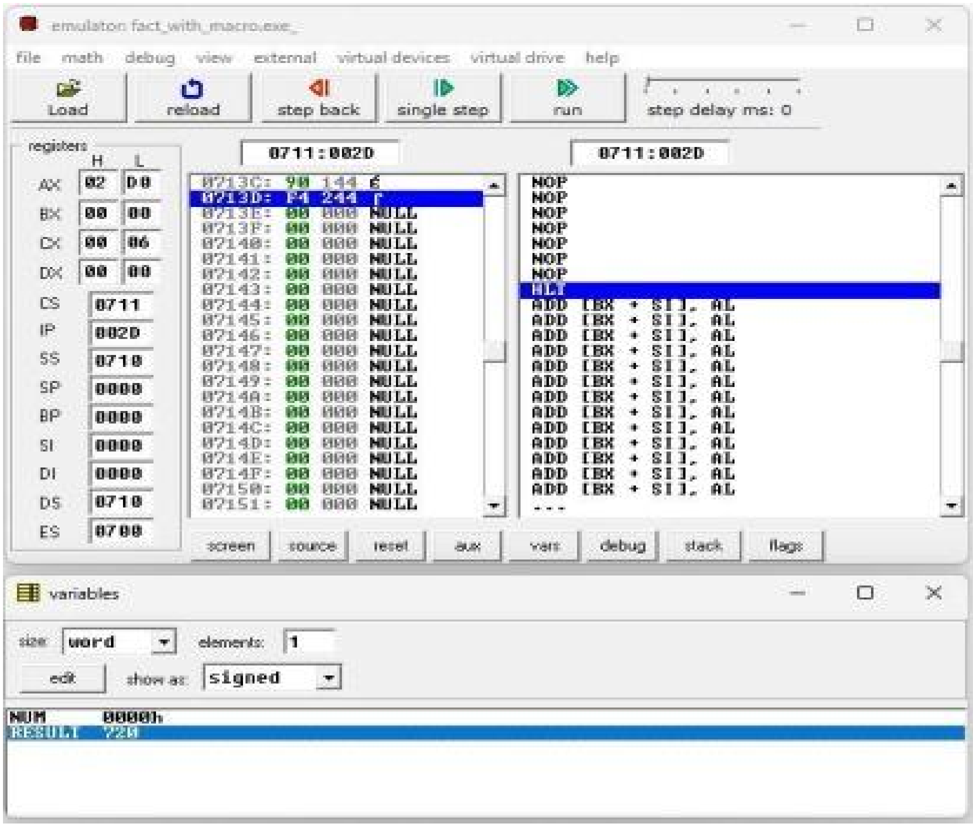
Output (without macro):





Output (with Macro) :





Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

POA EXPERIMENT 10

Jigar Siddhpura
60004210155

POA

Experiment 10 : Interrupts

Aim : Implement assembly program to demonstrate few DOS interrupts.

Theory : 1. In 8086 microprocessor, interrupts are event that cause normal sequence of program execution to be temporarily halted so that specific set of instructions (Instruction Service Routine) can be executed. The 'INT' instruction is used to generate interrupts.

2. Few common interrupts in 8086 :
   a) INT 21H : DOS function call. It is a software interrupt to invoke various DOS services.
   b) INT 10H : Video services. It is used for video related services such as displaying characters on screen
   c) INT 16H : Keyboard services, used for keyboard related services, like reading a character from keyboard.
   d) INT 4CH : Program termination, used to terminate a program & return control to OS.

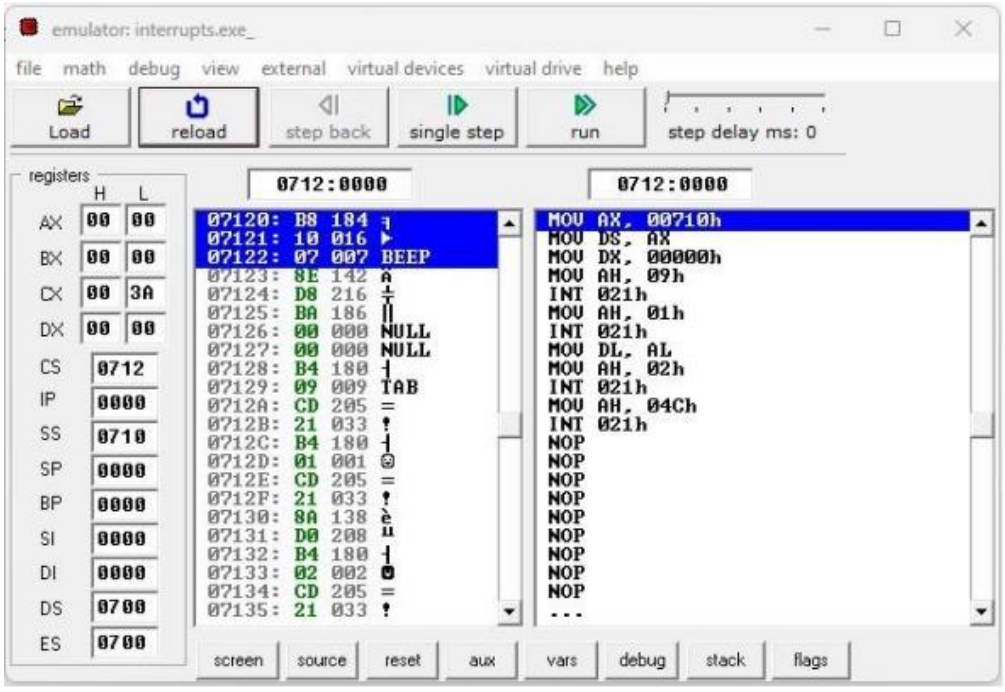
3. The Interrupt vector table in 8086 contains addresses pointing to an ISR.
4. When interrupts occurs, processor looks up corresponding address in IVT & jumps to associated routine.
5. The routine is responsible for handling specific interrupt & returning control to main program.

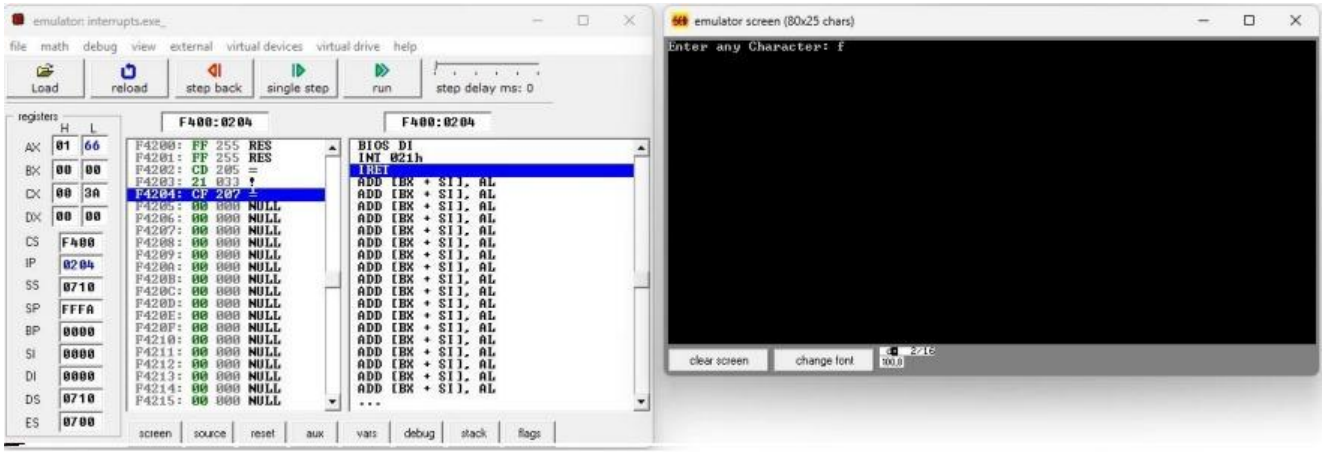
Conclusion : Hence, we have implemented various DOS interrupts on 8086 processor & understood how interrupt works in 8086.

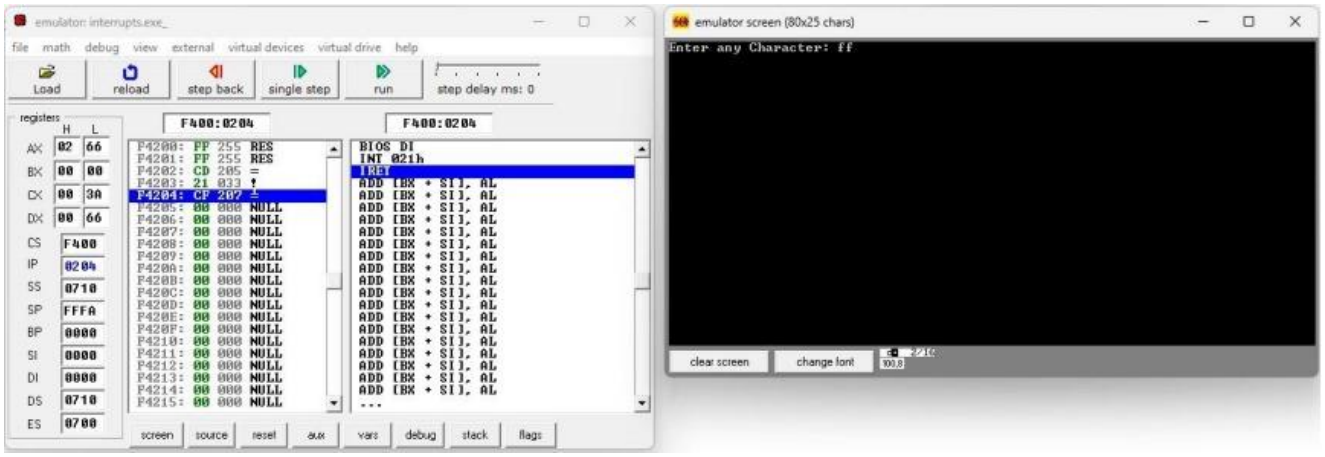
Code :

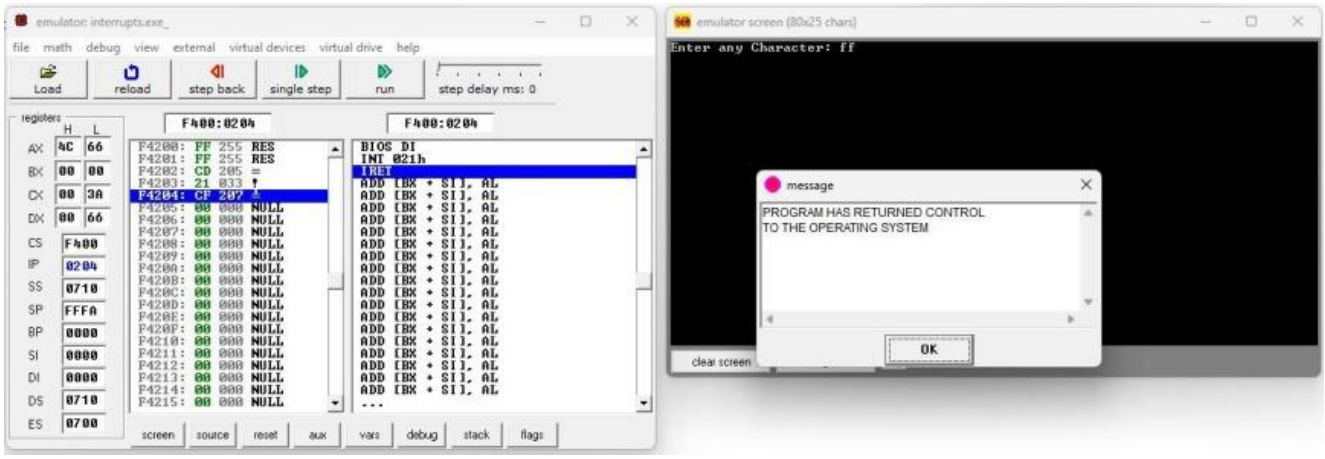
```
org 100h

DATA SEGMENT
	MSG DB "Enter any Character : $"
	DATA ENDS
CODE SEGMENT
	START:
	MOV AX,DATA
	MOV DS,AX
	LEA DX,MSG
	MOV AH,09H
	INT 21H
	MOV AH,01
	INT 21H
	MOV DL,AL
	MOV AH,02
	INT 21H


	MOV AH,4CH
	INT 21H
	CODE ENDS
END START

ret
```

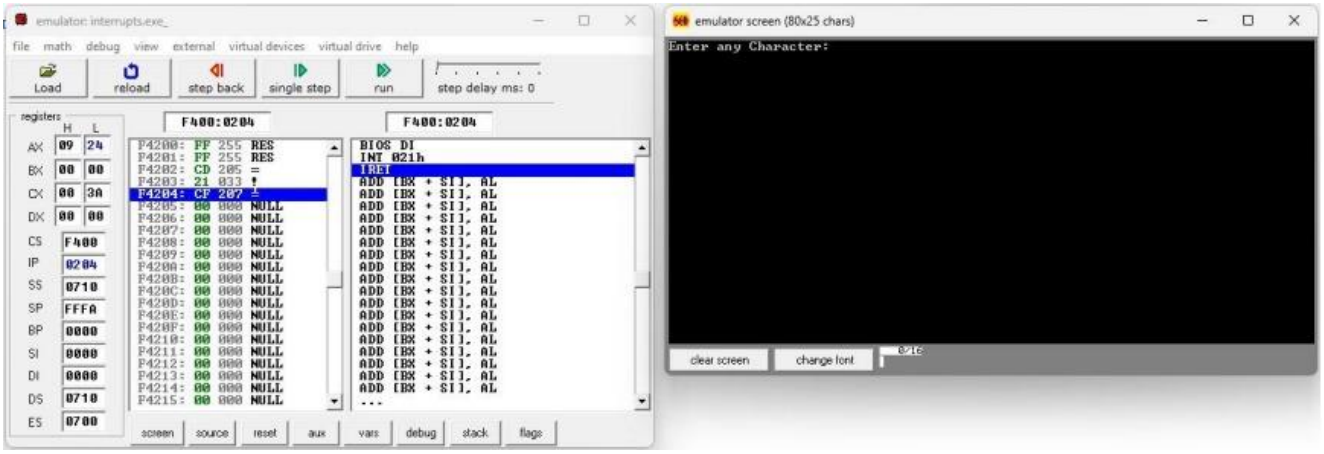
Output:




emulator: interrupts.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
H L
AX 01 66
BX 00 00
CX 00 3A
DX 00 00
CS F400
IP 0204
SS 0710
SP FFFA
BP 0000
SI 0000
DI 0000
DS 0710
ES 0700
F400:0204
F400:0204
F4200: FF 255 RES
F4201: FF 255 RES
F4202: CD 205 =
F4203: 21 033 !
F4204: CF 207 ≟
BIOS DI
INT 021h
IRET
ADD [BX + SI], AL
screen source reset aux vars debug stack flags
emulator screen (80x25 chars)
Enter any Character: f
clear screen
change font


emulator: interrupts.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
H L
AX 02 66
BX 00 00
CX 00 3A
DX 00 66
CS F400
IP 0204
SS 0710
SP FFFA
BP 0000
SI 0000
DI 0000
DS 0710
ES 0700
F400:0204
F400:0204
F4200: FF 255 RES
F4201: FF 255 RES
F4202: CD 205 =
F4203: 21 033 !
F4204: CF 207 ≟
BIOS DI
INT 021h
IRET
ADD [BX + SI], AL
screen source reset aux vars debug stack flags
emulator screen (80x25 chars)
Enter any Character: ff
clear screen
change font


emulator: interrupts.exe_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
H L
AX 4C 66
BX 00 00
CX 00 3A
DX 00 66
CS F400
IP 0204
SS 0710
SP FFFA
BP 0000
SI 0000
DI 0000
DS 0710
ES 0700
F400:0204
F400:0204
F4200: FF 255 RES
F4201: FF 255 RES
F4202: CD 205 =
F4203: 21 033 !
F4204: CF 207 ≟
BIOS DI
INT 021h
IRET
ADD [BX + SI], AL
screen source reset aux vars debug stack flags
emulator screen (80x25 chars)
Enter any Character: ff
message
PROGRAM HAS RETURNED CONTROL
TO THE OPERATING SYSTEM
OK
clear screen


emulator: interrupts.exe_
file math debug view external virtual devices virtual drive help
Load
reload
step back
single step
run
step delay ms: 0
registers
H L
AX 09 24
BX 00 00
CX 00 3A
DX 00 00
CS F400
IP 0204
SS 0710
SP FFFA
BP 0000
SI 0000
DI 0000
DS 0710
ES 0700
F400:0204
F400:0204
F4200: FF 255 RES
F4201: FF 255 RES
F4202: CD 205 =
F4203: 21 033 !
F4204: CF 207 ±
F4205: 00 000 NULL
F4206: 00 000 NULL
F4207: 00 000 NULL
F4208: 00 000 NULL
F4209: 00 000 NULL
F420A: 00 000 NULL
F420B: 00 000 NULL
F420C: 00 000 NULL
F420D: 00 000 NULL
F420E: 00 000 NULL
F420F: 00 000 NULL
F4210: 00 000 NULL
F4211: 00 000 NULL
F4212: 00 000 NULL
F4213: 00 000 NULL
F4214: 00 000 NULL
F4215: 00 000 NULL
BIOS DI
INT 021h
IRET
ADD [BX + SI], AL
---
screen source reset aux vars debug stack flags
emulator screen (80x25 chars)
Enter any Character:
clear screen
change font
0/16