For ADDITION:

Update Address in 0100 : 1000

```
MOV AX,[1000h]
MOV BX,[1002h]
MOV CL,00h
ADD AX,BX

MOV [1004h],AX
JNC jump

INC CL
jump:
MOV [1006h],CL
HLT
```

FOR SUBTRACTION

```
MOV AX,[1000H]
MOV BX,[1002H]
MOV CL,00H
SUB AX,BX
JNC jump:
INC CL
NOT AX
ADD AX,0001H
jump:
MOV [1004H],AX
MOV [1006H],CL
HLT
```

REMEMBER NOT TO ADD SPACE BEFORE COLON:
0710 to view the DS for Sorting in MEMORY

Sorting

ASCENDING : JC DOWN
DESCENDING : JNC DOWN

```
DATA SEGMENT
STRING1 DB 99H,45H,36H,55H,12H
DATA ENDS

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

MOV CH,04H  ; Outer loop counter initialization

UP2:
MOV CL,04H  ; Inner loop counter initialization
LEA SI,STRING1  ; Load effective address of STRING1 into SI

UP1:
MOV AL,[SI]  ; Load value at SI into AL
MOV BL,[SI+1]  ; Load next value into BL
CMP AL,BL  ; Compare AL and BL

; Jump to DOWN if AL < BL (carry flag CF set)
JC DOWN / JNC DOWN

; If no jump, swap values at SI and SI+1
MOV DL,[SI+1]
XCHG [SI],DL
MOV [SI+1],DL

DOWN:
INC SI
DEC CL
JNZ UP1  ; Jump to UP1 if inner loop counter is not zero
DEC CH
JNZ UP2  ; Jump to UP2 if outer loop counter is not zero

CODE ENDS
END START
```

BLOCK TRANSFER

```
data segment
seg1 db 01h,02h,03h
data ends

extra segment
seg2 db ?
extra ends

code segment
start:
;setting the segements register
mov ax,data
mov ds,ax
mov ax,extra
mov es,ax


lea si,seg1
lea di,seg2

mov cx,03H

x:
   mov ah,ds:[si]
   mov es:[di],ah

inc si
inc di
dec cx
jnz x

int 3 ; interrupt to halt

code ends
end start
```

MIN MAX

```
DATA SEGMENT
ARR DB 5,3,7,1,9,2,6,8,4
LEN DW $-ARR
MIN DB ?
MAX DB ?
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE,DS:DATA
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
```

Factorial (WIthout Macro)

```
DATA SEGMENT
A DB 5        ; Input number for which factorial is calculated
fact DB ?     ; Variable to store the factorial result
DATA ENDS

CODE SEGMENT
   ASSUME DS:DATA, CS:CODE
START:
   MOV AX, DATA
   MOV DS, AX
   MOV AH, 00
   MOV AL, A   ; AL is initialized with the input number A

L1:
   DEC A      ; Decrement the value of A
   MUL A      ; Multiply AL by A, result in AX:DX
   MOV CL, A  ; Move the current value of A into CL for comparison
   CMP CL, 01 ; Compare CL with 1
   JNZ L1     ; Jump to L1 if CL is not equal to 1 (repeat the loop)

   MOV fact, AL ; Move the result (factorial) from AL to 'fact'

CODE ENDS
END START
```

Factorial with Macro

```
fact macro f
    up:
    mul f
    dec f
    jnz up

endm

data segment
    num dw 05h
    result dw ?
data ends

stack segment
    dw 128 dup(0)
ends

code segment
    start:
    mov ax,data
    mov ds,ax

    mov cx,num
    mov ax,0001h

    fact num

    mov result,ax
 code ends
 end start
```

DOS Interrupt

Display input :
lea dx,msg
 mov ah,09h

Read character : mov ah,01

Display character :
Mov dl,al
mov ah,02

Terminate program : mov ah,4ch

```
data segment
    msg db "Enter a character : $"
data ends

code segment
assume cs:code,ds:data
start:
mov ax,data
mov ds,ax

;displays the input
lea dx,msg
mov ah,09h
INT 21h

;takes the input character
mov ah,01
INT 21h

; displays the output character
mov dl,al
mov ah,02
INT 21h

;terminates the program
mov ah,4ch
INT 21h
code ends
end start
```

LRU

```
count_memory=int(input("ENter number of memory blocks : "))
seq=list(map(int,input("Enter a sequence : ").split(' ')))

hits=0
faults=0
block=[None for _ in range(count_memory)]
timestamp=[-1 for _ in range(count_memory)]
for i,j in enumerate(seq):
    if j not in block:
        #page fault
        minimum=min(timestamp)
        index=timestamp.index(minimum)

        block[index]=j
        timestamp[index]=i
        faults+=1
    else:
        #page hit
        index=block.index(j)
        timestamp[index]=i
        hits+=1
    print(j,"\t",block)

print(f"Hits {hits}")
print(f"Faults {faults}")
```

FIFO

```
count_memory=int(input("Enter number of memory blocks : "))
seq=list(map(int,input("Enter a sequence ").split(' ')))

block=[None for _ in range (count_memory)]

hits=0
faults=0
head=0

for i,j in enumerate(seq):
    if j not in block:
       #Page Fault
        block[head]=j
        head=(head+1)%count_memory
        faults+=1
    else:
        hits+=1
    print(j," :\t",block)

print(f"Hits : {hits} \n Faults: {faults}")
```

Best Fit

```
def best_fit(memory_blocks,process_sizes):
    for(process_id,process_size) in enumerate(process_sizes):
        best_fit_index=-1
        choosen_block=-1
        min_remaining_space=float('inf')

        for(i,block_size) in enumerate(memory_blocks):
            if(block_size>=process_size and
block_size-process_size<min_remaining_space):
                best_fit_index=i
                min_remaining_space=block_size-process_size
                choosen_block=block_size
        if best_fit_index!=-1:
            memory_blocks[best_fit_index]-=process_size
            print(f"Process {process_id} {process_size} allocated to
block {best_fit_index} {choosen_block}")
        else:
```

```
            print(f"Process {process_id} {process_size} cannot be
alocated to any block")

memory_blocks = [100, 500, 200, 300, 600]
process_sizes = [212, 417, 112, 426]

best_fit(memory_blocks, process_sizes)
```

Worst Fit

```
def worst_fit(memory_blocks,process_sizes):
    for (process_id,process_size) in enumerate(process_sizes):
        worst_fit_index=-1
        max_remaining_space=-1
        choosen_block=-1
        for(i,block_size) in enumerate(memory_blocks):
            if(block_size>process_size and block_size-process_size >
max_remaining_space):
                worst_fit_index=i
                max_remaining_space=block_size-process_size
                choosen_block=block_size
        if worst_fit_index!=-1:
            memory_blocks[worst_fit_index]-=process_size
            print(f"Process {process_id} {process_size} allocated to
block {worst_fit_index} {choosen_block}")
        else:
            print(f"Process {process_id} {process_size} not alocated to
any block")

memory_blocks = [100, 500, 200, 300, 600]
process_sizes = [212, 417, 112, 426]

worst_fit(memory_blocks, process_sizes)
```

First Fit

```
def first_fit(memory_blocks,process_sizes):
    for (process_id,process_size) in enumerate(process_sizes):
        allocated=False
        for (i,block_size) in enumerate(memory_blocks):
            if(block_size>=process_size):
                memory_blocks[i]-=process_size
                print(f"Process {process_id} {process_size} allocated
to block {i}")
```

```
                allocated=True
                break

        if not allocated:
            print(f"Process {process_id} {process_size} cannot be
allocated to any block")

memory_blocks = [100, 500, 200, 300, 600]
process_sizes = [212, 417, 112, 426]

first_fit(memory_blocks, process_sizes)
```

Booths
if(n==0):
Return

Def add:
For i in range(max_len-1,-1,-1):
r=carry
r+=1 if (a[i]==”1”) else 0
A,m,q,q1,n,n_len
Input Q first
Then M

```
def booth(a,m,q,q1,n,n_len):
    print(f"{n}\t\t{a}\t\t{q}\t\t{q1}")
    if(n==0):
        return f"Answer is {a,q} deci={int(a+q,2)} " if a[0]==0 else
f"Answer is -ve {a,q} \n 2's complement {complement(a+q)} \n Decimal
{int(complement(a+q),2)}"
    if (q[-1]=="1" and q1[-1]=="0"):
        a=add(a,complement(m.zfill(n_len)))
        if(len(a)!=n_len):
            a=a[1:]
        a,q,q1=ars(a,q,q1)
    elif (q[-1]=="0" and q1[-1]=="1"):
        a=add(a,m)
        if(len(a)!=n_len):
            a=a[1:]
        a,q,q1=ars(a,q,q1)
    elif ( (q[-1]=="0" and q1[-1]=="0") or  (q[-1]=="1" and
q1[-1]=="1")):
        a,q,q1=ars(a,q,q1)

    return booth(a,m,q,q1,n-1,n_len)

def add(a,b):
    result=''
    carry=0
    max_len=max(len(a),len(b))
    a=a.zfill(max_len)
    b=b.zfill(max_len)
    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
        carry=0 if r<2 else 1
    if carry!=0:
        result='1'+result
    return result.zfill(max_len)

def ars(a,q,q1):
    q1=q[-1]
    q=a[-1]+q[:-1]
    a=a[0]+a[:-1]
    return (a,q,q1)

def complement(a):
    res=""
    for i in a:
        if i=="1":
            res+='0'
        elif i=="0":
            res+='1'
    res=add(res,'1')
    return res

# Input
a = int(input("Enter Q : "))
b = int(input("Enter M : "))
n = len(bin(max(abs(a),abs(b)))[2:]) + 1
a = bin(a)[2:].zfill(n) if a >= 0 else complement(bin(a)[3:].zfill(n))
b = bin(b)[2:].zfill(n) if b >= 0 else complement(bin(b)[3:].zfill(n))

# Initial display
print(f"M = {a}, Q = {b}, A={'0'*n}, Count={n}")
print("Count\tA\t\tQ\t\tQ1")
print("-----------------------------------------------------------------"
)

# Call the booth function
print(booth('0'*n, a.zfill(n), b.zfill(n), '0', n, n))
```

Restoring
a,q,m,n,n_len

```
def restoring(a,q,m,n,n_len):
    print(f"{n} \t {a} \t\t {q}")
    if(n==0):
        return f"Quotient is {q} deci = {int(q,2)}\n Remainder is {a}
dec={int(a,2)}"
    a,q=ls(a,q)
    a=add(a,complement(m.zfill(n_len)))
    if(len(a)!=n_len):
        a=a[1:]
    if(a[0]=="1"):
        a=add(a,m)
        if(len(a)!=n_len):
            a=a[1:]
        q=q.replace('_','0')
    elif(a[0]=="0"):
        q=q.replace('_','1')
    return restoring(a,q,m,n-1,n_len)
```

```
def add(a,b):
    result=""
    carry=0
    max_len=max(len(a),len(b))
    a=a.zfill(max_len)
    b=b.zfill(max_len)
    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
        carry=0 if r<2 else 1
    if carry!=0:
        result='1'+result
    return result.zfill(max_len)
```

```
def complement(a):
    res=''
    for i in a:
        if i=="0":
            res+='1'
        elif i=='1':
            res+='0'
    res=add(res,'1')
    return res

def ls(a,q):
    a=a[1:]+q[0]
    q=q[1:]+"_"
    return a,q

def ars(a,q,q1):
    q1=q[-1]
    q=a[-1]+q[:-1]
    a=a[0]+a[:-1]

a=int(input("Enter Numerator : "))
b=int(input("Enter Denominator : "))
n=len(bin(max(abs(a),abs(b)))[2:])+1
a=bin(a)[2:].zfill(n) if a>=0 else complement(bin(a)[3:].zfill(n))
b=bin(b)[2:].zfill(n) if b>=0 else complement(bin(b)[3:].zfill(n))

print(restoring('0'*n,a.zfill(n),b.zfill(n),n,n))
```

Non Restoring
Taking Input n-1

```
q=q.replace("_",str(int(not(int(a[0])))))
```

```
def nonRestoring(a,q,m,n,n_len):
    print(f"{n} \t {a} \t {q}")
    if n==0:
        if a[0]=="1":
            a=add(a,m)
            if(len(a)!=n_len):
                a=a[1:]
        return f"Quotient is {q} deci = {int(q,2)} \n Remainder is {a}
dec= {int(a,2)}"

    a,q=ls(a,q)
    if(a[0]=="0"):
        a=add(a,complement(m))
        if(len(a)!=n_len):
                a=a[1:]
    elif (a[0]=="1"):
        a=add(a,m)
        if(len(a)!=n_len):
                a=a[1:]
    q=q.replace("_",str(int(not(int(a[0])))))
    return nonRestoring(a,q,m,n-1,n_len)


def add(a,b):
    max_len=max(len(a),len(b))
    a=a.zfill(max_len)
    b=b.zfill(max_len)
    carry=0
    result=""
    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
        carry=  0 if r<2 else 1
    if carry!=0:
        result="1"+result
    return result.zfill(max_len)
```