EXERCISE 1 P#15

1. How the processor uses the address bus, the data bus, and the control bus to communicate with the system memory?

Address bus

The group of bits that the processor uses to inform the memory about which element to read or write is collectively known as address bus.

Data bus

Data bus is used to move the data from the memory to the processor in a read operation and from the processor to the memory in a write operation. *The third group consists of miscellaneous independent lines used for control purposes. For example, one line of the bus is used to inform the memory about whether to do the read operation or the write operation. These lines are collectively known as the *control bus*. *

Control bus

The third group consists of miscellaneous independent lines used for control purposes. For example, one line of the bus is used to inform the memory about whether to do the read operation or the write operation. These lines are collectively known as the *control bus*.

Control bus is special and relatively complex, because different lines comprising it behave differently. Some take information from the processor to a peripheral and some take information from the peripheral to the processor.

2. Which of the following are unidirectional and which are bidirectional?

a. Address Bus

The address bus is unidirectional. Address always travels from processor to memory.

b. Data Bus

Data bus is bidirectional. Data moves from both, processor to memory and memory to processor

c. Control Bus

Control bus is a bidirectional bus. It can carry information from processor to memory as well as from memory to processor

3. What are registers and what are the specific features of the accumulator, index registers, program counter, and program status word?

Accumulator

There is a central register in every processor called the accumulator. All mathematical and logical operations are performed on accumulator. The word size of a processor is defined by the width of its accumulator. A 32bit processor has an accumulator of 32 bits.

Index Register

Index register is used in such a situation to hold the address of the current array location. Now the value in the index register cannot be treated as data, but it is the address of data. In general whenever we need access to a memory location whose address is not known until runtime we need an index register. Without this register we would have needed to explicitly code each, iteration separately. In newer architectures the distinction between

accumulator and index registers has become vague. They have general registers, which are more versatile and can do both functions. They do have some specialized behaviors but basic operations can be done on all general registers.

Program counter

A register in the control unit of the CPU that is used to keep track of the address of the current or next instruction. Typically, the program counter is advanced to the next instruction, and then the current instruction is executed. Also known as a "sequence control register" and the "instruction pointer."

Everything must translate into a binary number for our dumb processor to understand it, be it an operand or an operation itself. Therefore the instructions themselves must be translated into numbers. For example to add numbers we understand the word "add." We translate this word into a number to make the processor understand it. This number is the actual instruction for the computer. All the objects, inheritance and encapsulation constructs in higher-level languages translate down to just a number in assembly language in the end. Addition, multiplication, shifting; all big programs are made using these simple building blocks. A number is at the bottom line since this is the only thing a computer can understand. (Page#4)

Program Status Word

This is a special register in every, architecture called the flags register or the program status word. Like the accumulator it is an 8, 16, or 32 bits register but unlike the accumulator it is meaningless as a unit, rather the individual bits carry different meanings. The bits of the accumulator work in parallel as a unit and each bit mean the same thing. The bit of the flags register work register work independently and individually, and combined its value is meaningless.

4. What is the size of the accumulator of a 64bit processor?

Size of the accumulator of a 64bit processor is 64-bits.

5. What is the difference between an instruction mnemonic and its opcode?

Opcode

Short for *operational code*, it is a number that determines the computer instruction to be executed. (machine language representation of an instruction)

6. How are instructions classified into groups?

INSTRUCTION GROUPS

Data Movement Instructions

These instructions are used to move data from one place to another. These places can be registers, memory, or even inside peripheral devices. Some examples are:

mov ax, bx

lad 1234

Arithmetic and Logic Instructions

Arithmetic instructions like addition, subtraction, multiplication, division and Logical instructions like logical and, logical or, logical xor, or complement are part of this group. Some examples are:

and ax, 1234

add bx, 0534

add bx, [1200]

The bracketed form is a complex variation meaning to add the data placed at address 1200.
In certain other cases we want the processor to first execute a separate block of code and then come back to resume processing where it left.
These are instructions that control the program execution and flow by playing with the instruction pointer and altering its normal behavior to point to the next instruction. Some examples are:
cmp ax, 0
jne 1234

Special Instructions

Another group called special instructions works like the special service commandos. They allow changing specific processor behaviors and are used to play with it. They are used rarely but are certainly used in any meaningful program. Some examples are:
cli
sti
Where cli clears the interrupt flag and sti sets it. Without delving deep into it, consider that the cli instruction instructs the processor to close its ears from the outside world and never listen to what is happening outside, possibly to do some very important task at hand, while sti restores normal behavior. Since these instructions change the processor behavior they are placed in the special instructions group.

7. A combination of 8bits is called a byte. What is the name for 4bits and for 16bits?

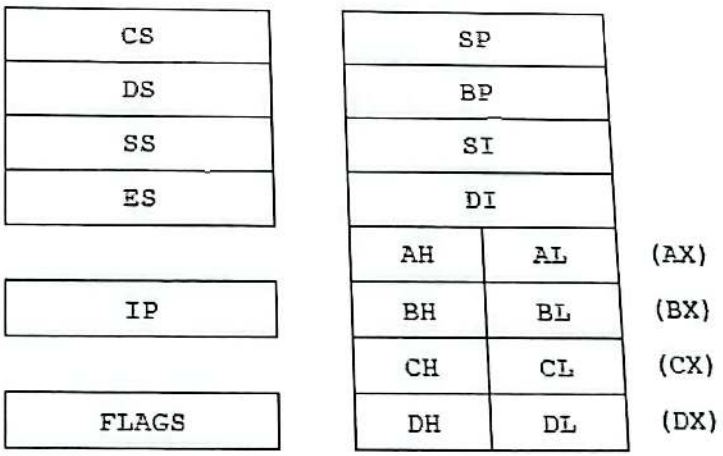
A number in base 16 is called a hex number and can be represented by 4 bits. The collection of 4 bits is called a nibble. One hexadecimal digit takes 4 bits so 4 hexadecimal digits make one word or two bytes.
4 bits = 1 nibble
8 bits =1 byte = 2 nibbles
16 bits = 2 bytes = 4 nibbles

8. What is the maximum memory 8088 can access?

8088 processor can access 1MB of memory.

9. List down the 14 registers of the 8088 architecture and briefly describe their uses.

1. **CS**
2. **DS**
3. **SS**
4. **ES**
5. **IP**
6. **SP**
7. **BP**
8. **SI**
9. **DI**
10. **AH AL (AX)**



11. BH BL (BX)
12. CH CL (CX)
13. DH DL (DX)
14. FLAGS

General Registers (AX, BX, CX, and DX)

The registers AX, BX, CX, and DX behave as general purpose registers in Intel architecture and do some specific functions in addition to it. X in their names stand for extended meaning 16bit registers. For example AX means we are referring to the extended 16bit "A" register. Its upper and lower byte are separately accessible as AH (A high byte) and AL (A low byte). All general-purpose registers can be accessed as one 16bit register or as two 8bit registers. The two registers AH and AL are part of the big whole AX. Any change in AH or AL is reflected in AX as well. AX is a composite or extended register formed by gluing together the two parts AH and AL.

The A of AX stands for Accumulator. Even though all general-purpose registers can act as accumulator in most instructions there are some specific variations, which can only work on AX, which is why it is named the accumulator. The B of BX stands for Base because of its role in memory addressing as discussed in the next chapter. The C of CX stands for Counter as there are certain instructions that work with an automatic count in the

CX register. The D of DX stands for Destination as it acts as the destination in I/O operations. The A, B, C, and D are in letter sequence as well as depict some special functionality of the register.

Index Registers (SI and DI)

These are the index registers of the Intel architecture, which hold address of data and used in memory access. Being an open and flexible architecture, Intel allows many mathematical and logical operations on these registers as well like the general registers. The source and destination are named because of their implied functionality as the source or the destination in a special class of instructions called the string instructions. However their use is not at all restricted to string instructions. SI and DI are 16bit and cannot be used as

8bit register pairs like AX, BX, CX, and DX.

Instruction Pointer (IP)

This is the special register containing the address of the next instruction to be executed. No mathematics or memory access can be done through this register. It is out of our direct control and is automatically used. Playing with it is dangerous and needs special care. Program control instructions change the IP register.

Stack Pointer (SP)

It is a memory pointer and is used indirectly by a set of instructions. This register will be explored in the discussion of the system stack.

Base Pointer (BP)

It is also a memory pointer containing the address in a special area of memory called the stack and will be explored alongside SP in the discussion of the stack.

Flags Register

The flags register as previously discussed is not meaningful as a unit rather it is bit wise significant and accordingly each bit is named separately.

The bits not named are unused. The Intel FLAGS register has its bits organized as follows:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

O D I T S Z A P C

The individual flags are explained in the following table.

Segment Registers (CS, DS, SS, and ES)

The code segment register, data segment register, stack segment register, and the extra segment register are special registers related to the Intel segmented memory model.

10. What flags are defined in the 8088 FLAGS register? Describe the function of the zero flag, the carry flag, the sign flag, and the overflow flag.

Zero flag

The Zero flag is set if the last mathematical or logical instruction has produced a zero in its destination.

Carry flag

When two 16bit numbers are added the answer can be 17 bits long or when two 8bit numbers are added the answer can be 9 bits long. This extra bit that won't fit in the target register is placed in the carry flag where it can be used and tested.

Sign flag

A signed number is represented in its two's complement form in the computer. The most significant bit (MSB) of a negative number in this representation is 1 and for a positive number it is zero. The sign bit of the last mathematical or logical operation's destination is copied into the sign flag.

Overflow flag

The overflow flag is set during signed arithmetic, e.g. addition or subtraction, when the sign of the destination changes unexpectedly. The actual process sets the overflow flag whenever the carry into the MSB is different from the carry out of the MSB.

11. Give the value of the zero flag, the carry flag, the sign flag, and the overflow flag after each of the following instructions if AX is initialized with 0x1254 and BX is initialized with 0x0FFF.

a. add ax, 0xEDAB

b. add ax, bx

c. add bx, 0xF001

- Carry Flag=1
- Zero Flag=1
- Sign Flag=0
- Overflow Flag=0

12. What is the difference between little endian and big endian formats? Which format is used by the Intel 8088 microprocessor?

Little endian	Big endian
• It can be least significant, more significant, more significant, and most significant called the little-endian order and is used by Intel. • The little-endian have the argument that this scheme places the less significant value at a lesser address and more significant value at a higher address.	• 32bit numbers either the order can be most significant; less significant, lesser significant, and least significant called the big-endian order used by Motorola. • The big-endian have the argument that it is more natural to read and comprehend

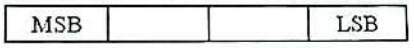
- Little-endian order is used by Intel 8088 microprocessor.

13. For each of the following words identify the byte that is stored at lower memory address and the byte that is stored at higher memory address in a little endian computer.

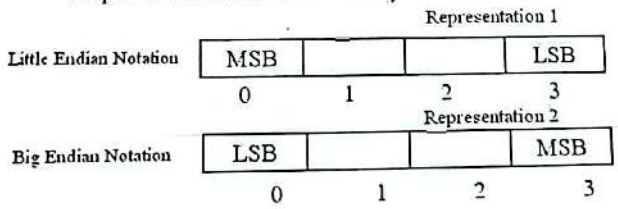
a. 1234
b. ABFC
c. B100
d. B800

Word Representation

- 4 Byte Word

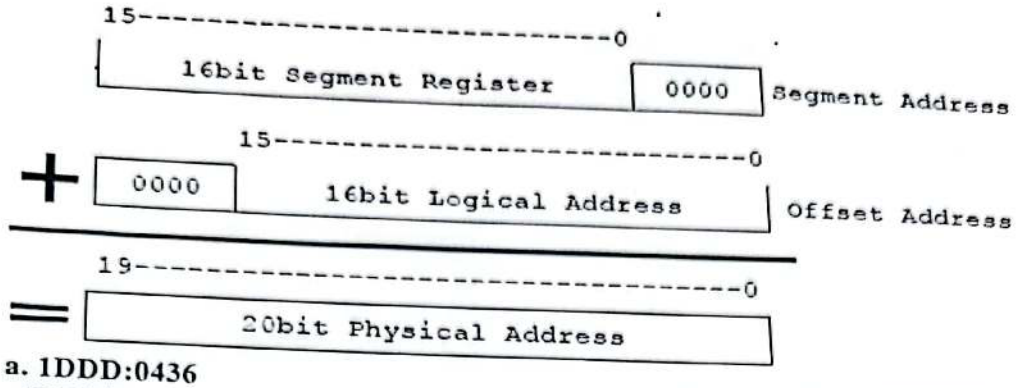


Representation in Memory



a. 1234
1 and 2 are stored at lower memory address and 3,4 stored at higher memory address.
b. ABFC
A and B are stored at lower memory address and F, C are stored at higher memory address.
c. B100
B and 1 are stored at lower memory and 0,0 are stored at higher memory address.
d. B800
B and 8 are stored at lower memory address and 0,0 are stored at higher memory address.

(MSB) **(LSB)**



a. 1DDD:0436

1DDD0 segment
+00436 offset
=1E206 physical memory address

Hexadecimal	Decimal
A	10
B	11
C	12
D	13
E	14
F	15

b. 1234:7920

12340
+07920
=19C60

c. 74F0: 2123

74F00
+02123
=77023

d. 0000:6727

00000
+06727
=06727

e. FFFF: 4336

FFFF0
+04336
=104326

f. 1080:0100

10800
+00100
=10900

g. AB01: FFFF

AB010
+0FFFF
=BB00F

1. **What is the difference between DATA LABEL and CODE LABEL?**

Data Label is the label that we use to define data as we defined memory locations num1,num2 ....etc in our programs. Code Label is the label that we have on code as we see in the case of conditional jump (Label l1) and is normally used for loop control statements.

2. **List the seven addressing modes available in the 8088 architecture.**

1. Direct
2. Base register Indirect
3. Indexed register Indirect
4. Base register indirect + offset
5. Index register indirect +offset
6. Base+index+offset

3. Differentiate between effective address and physical address.

The **effective address** is the address generated by the program, after all, transformations, such as index registers, offsets, addressing mode, etc. have been made. The **physical address** is the address generated by the hardware, after performing whatever lookups through the page table, etc. have been made. The effective address, or a virtual address, is the concern of the program. The physical address, or real address, is the concern of the operating system.

4. What is the effective address generated by the following instructions? Every instruction is independent of others. Initially BX=0x0100, num1=0x1001, [num1]=0x0000, and SI=0x0100

a. mov ax, [bx+12]
b. mov ax, [bx+num1]
c. mov ax, [num1+bx]
d. mov ax, [bx+si]

a. bx+12 = 0x0100 + 0xc = 0x010c
b. bx+num1 = 0x0100 + 0x1001 = 0x1101
c. num1+bx = 0x1001 + 0x0100 = 0x1101
d. bx+si = 0x0100 + 0x100 = 0x0200

5. What is the effective address generated by the following combinations if they are valid. If not give reason. Initially BX=0x0100, SI=0x0010, DI=0x0001, BP=0x0200, and SP=0xFFFF

a. bx-si
b. bx-bp
c. bx+10

d. bx-10
e. bx+sp
f. Bx+di

6.Identify the problems in the following instructions and correct them by replacing them with one or two instructions having the same effect.

a. mov [02], [22]
b. mov [wordvar], 20
c. mov bx, al
d. mov ax, [si+di+100]

a)
mov ax, [22]
mov [02], ax
OR
mov al, [22]
mov [02], [al]
b)
mov al, 22
mov [wordvar], al
c)
mov bl, al
d)
Mov bx, si
Mov ax, [bx+di+100]

7. What is the function of segment override prefix and what changes it brings to the opcode?

Solution:

The function of Segment Override Prefix:

The segment override prefix, which may be added to almost any instruction in any memory addressing mode, allows the programmer to deviate from the default segment. The segment override prefix is an additional byte that appends the front of an instruction to select an alternate segment register. The only instructions that cannot be prefixed are the jump and call instructions that must use the code segment register for address generation. For example, the MOV AX, [DI] instruction accesses data within the data segment by default. If required by a program it can be changed by prefixing the

instruction. Suppose that the data are in the extra segment instead of in the data segment. This instruction addresses the extra segment if changed to MOV AX, ES:[DI].

Changes to Opcode:

Opcode never changes, but the prefix byte modifies the default association to association with the desired segment register for the instruction.

8. What are the two types of address wraparound? What physical address is accessed with [BX+SI] if FFFF is loaded in BX, SI, and DS.

Solution:

There are two types of address wraparounds:

1. Within a single segment
2. Inside the whole megabyte/physical memory

Within a single segment

During the effective address calculation when a carry is generated then Segment Wraparound occurs. This carry is dropped giving the effect that when we try to access beyond the segment limit, we are actually wrapped around to the first cell in the segment. Just like a circle when we reached the end then we started again from the beginning. An arc at 370 degrees is the same as an arc at 10 degrees. We tried to cross the segment boundary and it pushed us back to the start. This is called segment wraparound.

For example if BX=9100, DS=1500 and the access is [bx+0x7000] we form the effective address 9100 + 7000 =10100. The carry generated is dropped giving the actual effective address of 0100. The physical address in this example will be 15100.

Inside the whole physical memory: Just like segment wraparound, the whole physical memory wraparound occurs. In Segment Wraparound memory address is pushed back to the start of the segment while in other cases address is pushed back to the very start of the physical memory. The following example, best describes the idea of a whole memory wraparound.

EXAMPLE: BX=0100, DS=FFF0 and the access under consideration is [bx+0x0100]. The effective address will be 0200 and the physical address will be 100100. This is a 21bit answer and cannot be sent on the address bus which is 20 bits wide. The carry is dropped and just like the segment wraparound, our physical memory has wrapped around at its very top. When we tried to access beyond limits the actual access is made at the very start.

Physical Address Calculation:
As
SI = FFFF
BX = FFFF
Segment : DS = FFFF
SI+BX= FFFF+FFFF= FFFE= offset address
Physical address = segment x 10 + offset address
= FFFF0+FFFE
= 0FFEE

9. Write instructions to do the following.
a. Copy contents of memory location with offset 0025 in the current data segment into AX.
b. Copy AX into memory location with offset 0FFF in the current data segment.
c. Move contents of memory location with offset 0010 to memory location with offset 002F in the current data segment.
Solution:

```
a. mov SI, 0x0025
 mov AX, [SI]
b. mov [DI], 0x0FFF
 mov [DI], AX
c. mov DI, 0x002F
 mov SI, 0x0010
 mov AX, [SI]
 mov [DI], AX
```

10. Write a program to calculate the square of 20 by using a loop that adds 20 to the accumulator 20 times.

Solution:

```
[org 0x0100]
mov bx,20
mov cx,20
mov ax,0
l1:
 add ax, bx
 sub cx, 1
 jnz l1
 mov [total], ax
```

```
mov ax,0x4c00
int 0x21
total: dw 0
```

Ch#03

1. Which registers are changed by the CMP instruction?

The compare instruction subtracts the source operand from the destination operand, updating the flags without changing either the source or the destination. So, it only changes the flag register.

2. What are the different types of jumps available? Describe position relative addressing.

The types of jumps are:

- Near Jumps
- Short Jumps
- Far Jumps

Position relative addressing: position relative addressing in contrast to absolute addressing does not tell the exact address rather it is telling how much forward or backward to go from the current position of IP in the current code segment.

3.If AX=8FFF and BX=0FFF and "cmp ax, bx" is executed, which of the following jumps will be taken? Each part is independent of others. Also give the value of Z, S, and C flags.

a. jg greater

b. jl smaller

c. ja above

d. jb below

Instructions	Jump	ZF	SF	CF
Jg greater	Not taken	0	1	0
Jl smaller	Taken	0	1	0

Ja above	Taken	0	1	0
Jb below	Not taken	0	1	0

4.Write a program in Assembly Language to find the maximum number and the minimum number from an array of ten numbers.

Solution:

```
[org 0x0100]
        jmp start               ; unconditionally jump over data
array1: dw  10, 5, 30, 4, 50, 1, 20, 6, 40, 8
min:    dw  0
max:    dw  0

start:

        mov  bx, 0              ; initialize array index to zero
        mov  ax, 0              ; initialize min to zero
        mov  ax, [array1+bx]    ; minimum number to ax
        mov cx,10

top1:   cmp  ax, [array1+bx]    ; are we find the minimum number
        jle  end1               ; if less or equal number
        mov ax,[array1+bx]      ;ax contains the minimum number

end1:
        add  bx, 2              ; advance bx to next index
        loop top1
        mov  [min], ax          ; write back minimum in memory
        mov  bx, 0              ; initialize array index to zero
        mov  ax, 0              ; initialize max to zero
        mov  ax, [array1+bx]    ; maximum number to ax
        mov cx,10

top2:   cmp  ax, [array1+bx]    ; are we find the maximum number
        jge  end2               ; if greater or equal number
        mov ax,[array1+bx]      ;ax contains the maximum number


end2:
        add  bx, 2              ; advance bx to next index
```

```
        loop top2

        mov [max], ax       ; write back maximum number in memory
        mov ax, 0x4c00      ; terminate program
        int 0x21
```

5. Write a program to search a particular element from an array using binary search. If the element is found set AX to one and otherwise to zero.

```
;Binary Search

[org 0x0100]

        jmp start1

data:  db 1,2,3,4,5,6,7,8,9,10,11
start: db 0
end:   db 10
key:   db -1

start1:  mov al,[key]

loop1: mov cl,[start]
        cmp cl,[end]
        ja end1                          ;Checking if(start<=end), if not then jump to
end1

        mov dl,[start]
        add dl,[end]            ;dl is basically now start + end
        sar dl,1                        ;here dl is being divided by 2
        mov bl,dl                       ;bl is mid and is calculated by (start + end)/2


        cmp  al, [data + bx]
        je store                        ; agar data mil gaya tw program end kar do
        ja step1                        ; agar data greater hai current element sey
        jb step2                        ; agar data smaller hai current element sey

step1: add dl,1                 ;mid + 1 kar do
        mov [start],dl          ;start ko ab mid + 1 kar do taakey hum mid se
aagey jaga par dekhein
        jmp loop1

step2: sub dl,1                 ;mid -1 kar do
        mov[end],dl             ;end ko ab mid - 1 kar do taakey hum mid se
previous jaga par dekhein
```

```
        jmp loop1

store: mov ax, 1
        mov ax,0x4c00
        int 21h

end1:  mov ax,0
        mov ax,0x4c00
        int 21h
```

6.Write a program to calculate the factorial of a number where factorial is defined as: factorial(x) = x*(x-1)*(x-2)*...*1 factorial(0) = 1

```
[org 0x0100]
mov bx,0
mov si,[l]
l1:
mov ax,[n+bx]
mov cx,ax
sub cx,1
l2:mul cx
        sub cx,1
        jnz l2
        mov [fact_num+bx],ax
        add bx,2
sub si,1
jne l1

mov ax, 0x4c00
int 0x21
n: dw 3,5,4,8,7
fact_num: dw 0,0,0,0,0
l: dw 5
```

14. What are the contents of memory locations 200, 201, 202, and 203 if the word 1234 is stored at offset 200 and the word 5678 is stored at offset 202?

In Little Endean, least significant byte is stored first at lower addresses; most significant byte is stored after it. Like for example the number 0x1234 is stored at memory address 0x123 then it will appear like as below:

Address	Contents
...	
0x123	34
0x124	12
...	

Do try it for the given memory addresses with given numbers.

15. What is the offset at which the first executable instruction of a COM file must be placed?

The very first instruction in our assembly programs i.e. [org 0x0100] tell that its .com file shall be loaded at an offset address 100 in the segment. This is done to maintain backward compatibility.

16. Why was segmentation originally introduced in 8088 architecture?

Four windows of 64K
Code window
Data window
Stack window

The segmented memory model allows multiple functional windows into the main memory, a code window, a data window etc. The processor sees code from the code window and data from the data window. The size of one window is restricted to 64K. 8085 software fits in just one such window. It sees code, data, and stack from this one window, so downward compatibility is attained. **(Try to find better answer)**

17. Why a segment start cannot start from the physical address 55555.

For any segment base address, segment first physical address will have 0 in the least significant position in hexadecimal format.

Let say, our Segment base =0x1234, and we calculate segment first physical address as 0x12340 <- 0x12340 + 0x00000 (Segment First Address)

Thus all segments starting physical address has 0 at its least significant position. In case of 55555 as segment first physical address there in no 0 at least significant positions so this cannot be a segment starting physical address.

18. Calculate the physical memory address generated by the following segment offset pairs.

19. What are the first and the last physical memory addresses accessible using the following segment values?

First 0x0000 and last 0xFFFF

First physical address	Last physical address
a. 1000	
0x10000	0x10000
+0x00000	+0x0FFFF
=0x10000	**=0x1FFFF**
b. 0FFF	
0x0FFF0	0x0FFF0
+0x00000	+0x0FFFF
=0x0FFF0	**=0x1FFEF**
c. 1002	
0x10020	0x10020
+0x00000	+0x0FFFF
=0x10020	**=0x2001F**
d. 0001	
0x00010	0x00010
+0x00000	+0x0FFFF
=0x00010	**=0x1000F**
e. E000	
0xE0000	0xE0000
+0x00000	+0x0FFFF
=0xE0000	**=0xEFFFF**

Note:

- That we have added Base & Offset addresses to calculate Physical addresses. Moreover also notice that we placed a Zero in Base & Offset address to produce a 20-bit address.
- I have used hexadecimal representation as shown by 0x used in the beginning of a number.

20. Write instructions that perform the following operations.

a. Copy BL into CL

mov CL, BL

b. Copy DX into AX

mov AX, DX Moreover DX can not be copied into AL because of size mis-match.

c. Store 0x12 into AL

add AL, 0x12

d. Store 0x1234 into AX

add AX,0x1234
c. Store 0xFFFF into AX
add AX, 0xFFFF

21. Write a program in assembly language that calculates the square of six by adding six to the accumulator six times.

[Move 6 into ax register in first instruction and then keep adding 6 in next 5 instructions. You will get square of 6.]

```
; Calculate square of six by adding six to accumulator six times
;[org 0x0100]

        mov ax,6        ; load first number in ax
        add ax,6        ; accumulate sum
        add ax,6
        add ax,6
        add ax,6
        add ax,6

        mov ax, 0x4c00  ; terminate program
        int 0x21
```

```
 1     ; calculates square of six by adding six to accumulator six times
 2
 3                                 -                   [org 0x0100]
 4
 5  00000000 B80600                                    mov ax,6
 6  00000003 050600                                    add ax,6
 7  00000006 B80600                                    mov ax,6
 8  00000009 B80600                                    mov ax,6
 9  0000000C B80600                                    mov ax,6
10  0000000F B80600                                    mov ax,6
11  00000012 B80600                                    mov ax,6
12
13  00000015 B8004C                                    mov ax, 0x4c00
14  00000018 CD21                                      int 0x21
```

Q:What is the instruction of this COPY AX INTO MEMORY LOCATION WITH OFFSET 0FFF IN THE CURRENT DATA SEGMENT?

The instruction that copy ax into memory location with offset 0FFF in the current data segment is given below:

Mov [DS:0FFF], AX

CH#04

1. Write a program to swap every pair of bits in the AX register.

Solution:;[org 0x0100]

```
; start:          mov ax, ABCD
;                       mov bx, 1010101010101010b
                        mov dx, 0101010101010101b
;
;                       and bx,ax
                        and dx,ax
;
;                       shr bx,1
                        shl dx,1
;
                        or bx,dx
;
                        mov ax,bx
; end:
;                   mov ax, 0x4c00
                        int 21h
```

2.Give the value of the AX register and the carry flag after each of the following instructions.

stc
mov ax, <your rollnumber>
adc ah, <first character of your name>
cmc
xor ah, al
mov cl, 4
shr al, cl
rcr ah, cl

```
[org 0x0100]

stc  ; AX:0, CF: 1

mov ax, 0x2365  ; AX:2365, CF: 1

adc ah, 0xA  ; AX:2E65,  CF: 0

cmc  ; AX:2E65,  CF: 1

xor ah, al  ; AX:4B65, CF: 0
```

```
mov cl, 4  ; AX:4B65, CF: 0

shr al, cl  ; AX:4B06,  CF: 0

rcr ah, cl  ; AX:6406,  CF: 1
mov ax, 0x4c00

int 21h
```

3.Write a program to swap the nibbles in each byte of the AX register

```
;[org 0x0100]

 ; start:          mov ax, ABCD
 ;                              mov bx, 1111000011110000b
 ;                              mov dx, 0000111100001111b

  ;                             and bx,ax
  ;                             and dx,ax

  ;                             shr bx,4
  ;                             shl dx,4

   ;                            or bx,dx

    ;                           mov ax,bx

    ; end:                 mov ax, 0x4c00
    ;                            int 21h
```

4. Calculate the number of one bits in BX and complement an equal number of least significant bits in AX. HINT: Use the XOR instruction

```
[org 0x0100]

                        mov ax, 0x1234
                        mov bx, 0xABCD
                        mov dx, 1000000000000000b
                        mov si, 0
                        mov cx, 0

   ;Calculating the no. of bits in bx

   loop1:          cmp dx,0
                        jz  part2

                        test bx,dx
                        jz  skip_inc

                        inc  si
```

```
skip_inc:       shr  dx,1
                jmp  loop1

;complementing from left to right the least significant bits of ax (one at a time)

part2:    cmp si,0
          jz  end

          mov dx, 0000000000000001b

loop2:    xor ax,dx
          shl dx,1
          inc cx
          cmp cx,si
          jnz loop2

end:            mov ax, 0x4c00
                int 21h
```

5.Write a program to multiply two 32bit numbers and store the answer in a 64bit location.

```
[org 0x0100]

          jmp start

buffer:             db
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32
startingBit:   db 105

start:              mov dx,1111111100000000b
                    mov bx,8
                    mov cx,0
                    mov ax,0
                    mov si,0

                    mov al, [startingBit]

                    div bl

                    mov bx, 0

                    mov bl,al
                    mov si,bx
```

```
                    cmp ah,0
                    jz  scenario0
                    jnz scenario1
;Desired Byte doesn't split into two bytes
scenario0:      mov bl, [buffer + si]

                    mov ax,bx
                    jmp end
;Desired Byte splits into two bytes
scenario1:
                mov cl, ah
                    mov bh, [buffer + si]
                    mov bl, [buffer + si + 1]   ;Done to
                    shl bx, cl                  ;maintain the order of bytes in bx
                    and dx, bx

                    mov ax, 0
                    mov al, dh
                    jmp end
end:
                mov ax,0x4c00
                    int 21h
```

6. Declare a 32byte buffer containing random data. Consider for this problem that the bits in these 32 bytes are numbered from 0 to 255. Declare another byte that contains the starting bit number. Write a program to copy the byte starting at this starting bit number in the AX register. Be careful that the starting bit number may not be a multiple of 8 and therefore the bits of the desired byte will be split into two bytes

```
[org 0x0100]

        jmp start

buffer:             db
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32
startingBit:    db 105

start:
                mov dx,1111111100000000b
                    mov bx,8
                    mov cx,0
                    mov ax,0
                    mov si,0

                    mov al, [startingBit]
```

```
                    div bl

                    mov bx, 0

                    mov bl,al
                    mov si,bx

                    cmp ah,0
                    jz  scenario0
                    jnz scenario1

;Desired Byte doesn't split into two bytes

scenario0:      mov bl, [buffer + si]

                    mov ax,bx
                    jmp end

;Desired Byte splits into two bytes

scenario1:
                mov cl, ah
                    mov bh, [buffer + si]      ;Done to
                    mov bl, [buffer + si + 1]  ;maintain the order of bytes in bx
                    shl bx, cl
                    and dx, bx

                    mov ax, 0
                    mov al, dh
                    jmp end

end:
                mov ax,0x4c00
                    int 21h
```

7. AX contains a number between 0-15. Write code to complement the corresponding bit in BX. For example if AX contains 6; complement the 6th bit of BX.

```
[org 0x0100]

start:
            mov ax, 7
                mov bx, 0xABCD

                mov cx,ax
                mov dx,1000000000000000b
                shr dx,cl
                xor bx,dx
```

```
end:            mov ax, 0x4c00
                int 21h
```

8. AX contains a non-zero number. Count the number of ones in it and store the result back in AX. Repeat the process on the result (AX) until AX contains one. Calculate in BX the number of iterations it took to make AX one. For example BX should contain 2 in the following case:
AX = 1100 0101 1010 0011 (input – 8 ones)
AX = 0000 0000 0000 1000 (after first iteration – 1 one)
AX = 0000 0000 0000 0001 (after second iteration – 1 one) STOP

```
[org 0x0100]
        jmp start

arr: dw
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64

start:          mov ax, 13
                sub sp,2
                push ax
                call myalloc
                pop ax

                push ax                 ;Index
                push 13         ;Bits
                call myfree

end:            mov ax, 0x4c00
                int 21h

;---------------------------------------------------

myalloc:        push bp
                mov bp,sp
                sub sp,4
;creating space for two local variables
```

```
                shr dh,1
                cmp dh,0
                jz  update

                jmp loop1


update:         mov dx, 1000000000000000b
                add bx, 1
                jmp loop1


reset:          mov ax, 0
                mov word [bp - 2], -1
                mov word [bp - 4],  0
                jmp l1

dreturn:        jmp return              ;Used because of short range jump issue at line 53

; After finding that many consecutive zero bits in the array , making them one

changeto1:      mov ax,0
                mov bx,0
                mov cx,0
                mov dx,8

                mov ax, [bp - 2] ;starting bit (index)

                div dl

                mov dx, 0

                mov dl,al
                mov bx,dx       ;Now bx contains the byte number which contains
index

                mov dx,1000000000000000b

                cmp ah,0
                jnz scenario1


scenario0:      ;Desired Byte doesn't split into two bytes

loop3:          or byte [arr + bx], dh

                inc cl
                cmp cl, [bp + 4]
```

```
;bp - 2 will be used to store the index temporarily
;bp - 4 will be used to hold the status of zeroes whether they
;                       are currently being checked or not.

                        pusha

                        mov ax, 0
                        mov bx, 0
                        mov cx, 0

                        mov word [bp - 2], -1           ;index is -1 by default

                        mov si, [bp + 4]
checked                                                 ;No. of zeroes to be

                        cmp si, 0                       ;If no. zero bits to be checked is 0, then do nothing
                        jz  dreturn

                        mov dx, 1000000000000000b                       ;mask for testing bits

                        mov word [bp - 4], 0            ;Currently we don't have a zero at hand

loop1:                  test byte [arr + bx] , dh
                        jnz  reset

                        cmp word [bp - 4],1             ;If some zeroes are at hand then don't store a
new index
                        jz  loop2

zeroFound:              mov word [bp - 2], cx                   ;Storing the index of the first
zero found
                        mov word [bp - 4], 1            ;Currently a zero is found .
                                                                ;No. of zeroes
loop2:                  inc ax
currently checked
                        cmp ax,si
                        jz  changeto1

                        inc cx
l1:                                                             ;0x400bits is
                        cmp cx, 0x400
 equivalent to 1024 bits
                        jz  return                      ;Means you are at the end of the arr
```

```
                    mov dx, 0

                    mov dl,al
                    mov bx,dx        ;Now bx contains the byte number which contains the starting
index

                    mov dx,0111111111111111b

                    cmp ah,0
                    jnz _scenario1

_scenario0:     ;Desired Byte doesn't split into two bytes

_loop3:         and byte [arr + bx], dh

                        inc cl
                        cmp cl, [bp + 4]
                        jz  _return

                        shr dh,1
                        cmp dh,0
                        jz  _update1
                        jmp _loop3

;Desired Byte splits into two bytes

_scenario1:     mov cl, ah
                        shr dx, cl

                        jmp _loop3


_update1:       mov dx,0111111111111111b
                        add bx,1
                        jmp _loop3


_return:        popa
                                pop bp
                                ret 4

;-----------------------------------------
```

```
                jz return

                shr dh,1
                cmp dh,0
                jz update1
                jmp loop3

;Desired Byte splits into two bytes

scenario1:      mov cl, ah
                shr dx, cl

                jmp loop3

update1:        mov dx,1000000000000000b
                add bx,1
                jmp loop3

return:         mov ax, [bp - 2]
                mov [bp + 6], ax

                popa

                add sp, 4
                pop bp

                ret 2

;-----------------------------------------

myfree:         push bp
                mov bp,sp
                pusha

                mov ax,0
                mov bx,0
                mov cx,0
                mov dx,8

                mov ax, [bp + 6]

                div dl
```

Chapter 8
Qno1

```
[org 0x0100]

        jmp start
isTSR:  dw 0
oldISR: dd 0
;-----------------------------------------------------------

mylSR:          cmp ah, 0x31                    ;Agar tw 0x31 wali service call
huwi tw kuch kaam karo, nahi tw humein koi kaam karney ki zaroorat
                                                :hi nahi
because hum ne tw sirf 31h wali service pe kaam karna hai

                jnz chain

                cmp word [cs:isTSR], 0          ;Agar tw humari mylSR TSR ban chuki hai, phir
ab kisi aur naye program ko TSR nahi banney dena
                jz makeitTSR                    ;Lekin agar humari mylSR TSR nahi bani, tw pehley
usey TSR banao

                mov ah, 0x4c

chain:          jmp far [cs:oldISR]
                .

makeitTSR:      mov word [cs:isTSR], 1
                jmp far [cs:oldISR]

;-----------------------------------------------------------

start:          xor ax,ax
                mov es,ax

                ;Saving the OLD ISR
                mov ax, [es:21h*4]
                mov [oldISR], ax

                mov ax, [es:21h*4 + 2]
                mov [oldISR + 2], ax

                ;Hooking our ISR
                mov word [es:21h*4], mylSR
                mov word [es:21h*4+2], cs

                mov dx, start
                add dx, 15
                mov cl, 4
                shr dx, cl

exit:           mov ax, 0x3100
                int 21h
```

qno2

```
[org 0x0100]
        jmp start

address:    dd 0
;-------------------------------------------------------------------------

;Clear Screen
clrscr:     push bp
            mov bp, sp

            pusha
            push es

            mov ax, 0xb800
            mov es, ax
            xor di,di
            mov ax,0x0720
            mov cx,2000

            cld
            rep stosw

            pop es

return:     mov ax, 0
            mov cx, 0

            cmp     ax, [bp+4]
            jz nearReturn

            cmp ax, [bp+6]
            jz farReturn

            cmp ax, [bp+8]
            jz interruptReturn

nearReturn:     popa
                pop bp
                ret 2

farReturn:      popa
                pop bp
                retf 2

interruptReturn: mov ax, [bp + 6]
                 mov [bp + 8], ax

                 mov ax, [bp + 4]
                 mov [bp + 6], ax
```

```
                mov ax, [bp + 2]
                mov [bp + 4], ax

                popa
                pop bp

                add sp, 2

                iret
;---------------------------------------------------------------------------

start:      xor ax, ax
            mov es, ax

            ;Hooking the interrupt
            mov word [es:80h*4], clrscr
            mov word [es:80h*4+2], cs

            ;Saving address for far call
            mov word [address], clrscr
            mov word [address + 2], cs

            push 0
            call clrscr                         ;Near Call

            push 0
            call far [address]                  ;Far Call

            push 0
            int 80h                             ;Interrupt Call (Extended Far Call)

            mov ax, 0x4c00
            int 21h
```

Qno3

```
[org 0x0100]

        jmp start

XISR_Offset:    dw 0x0000
XISR_Segment:   dw 0x0000
N:                      dw 0x80
;---------------------------------------------------------------------------

hooker:     push bp
            mov bp, sp
```

```
sub sp, 4                    ;Making two local variables, one for old
offset and one for old segment of the ISR previously hooked at N

pusha
push es


;bp - 2                      ;Old segment
;bp - 4                      ;Old offset

;bp + 4                      ;XISR Offset
;bp + 6                      ;XISR Segment
;bp + 8                      ;Interrupt No. 'N'


xor ax, ax
mov es, ax


mov di, [bp + 8]            ;Interrupt No. 'N'

;First of all saving the offset, segment of the ISR previously hooked at N

shl di, 2                    ;Multiplying by 4

;Saving the offset
mov bx, [es:di]
mov [bp - 4], bx

;Saving the segment
mov bx, [es:di + 2]
mov [bp - 2], bx


;Loading the segment of XISR in es
mov es, [bp + 6]


;Chaining the XISR to the old ISR previously hooked at N

mov bx, [bp + 4]            ;Offset of XISR

mov ax, [bp - 4]            ;Offset
mov dx, [bp - 2]            ;Segment


mov [es:bx + 2], ax
mov [es:bx + 4], dx


;Now hooking XISR at N
mov ax, 0
mov es, ax
```

```
                mov di,[bp + 8]
                shl di, 2                                   ;Multiplying by 4

                mov ax,[bp + 4]
                mov [es:di], ax                 ;Offset of XISR
                mov ax, [bp + 6]
                mov [es:di+ 2], ax                      ;Segment of XISR

return:         pop es
                popa
                add sp, 4

                pop bp
                ret 6
;-------------------------------------------------------------------------------

;-------------------------------------------------------------------------------
XISR:   pushf
                call 0:0

                popf
                ret

;-------------------------------------------------------------------------------
start:          push word [N]

                mov  word [XISR_Offset], XISR
                mov  word [XISR_Segment], cs

                push word [XISR_Segment]
                push word [XISR_Offset]

                call hooker


                mov ax, 0x4c00
                int 21h
```

CHAPTER NO 9

Qno 3
:Write a program to make an asterisk travel the border of the screen,
:from upper left to upper right to lower right to lower left and back to upper left indefinitely.

```
[org 0x0100]
jmp start

start:          call clrscr
                call borderAsterisk

                mov ax, 0x4c00
                int 21h

;Clear Screen
clrscr:         mov ax, 0xb800
                mov es, ax
                xor di,di
                mov ax,0x0720
                mov cx,2000

                cld
                rep stosw

                ret

;Delay
delay:          pusha
                mov cx, 0xFFFF

b1:             loop b1

                popa
                ret

borderAsterisk: push bp
                mov bp, sp
                pusha

                ;Loading the video memory
                mov ax, 0xb800
                mov es, ax

                mov di, 0

                mov ah, 01110000b
```

```
                        mov al, '*'
                        mov bh, 0x07
                        mov bl, 0x20
LefttoRight:    mov cx, 80
l1:                     mov [es:di], ax
                        call delay
                        mov [es:di], bx
                        call delay
                        add di, 2
                        loop l1
                        sub di, 2

RightToBottom:  mov cx, 25
l2:                     mov [es:di], ax
                        call delay
                        mov [es:di], bx
                        call delay
                        add di, 160
                        loop l2
                        sub di, 160

BottomToLeft:   mov cx, 80
l3:                     mov [es:di], ax
                        call delay
                        mov [es:di], bx
                        call delay
                        sub di, 2
                        loop l3
                        add di, 2
```

```
LefttoTop:          mov cx, 25
l4:                 mov [es:di], ax
                    call delay
                    mov [es:di], bx
                    call delay
                    sub di, 160
                    loop l4
                    add di, 160
                    ;Then repeat the whole process again resulting in an infinite loop
                    jmp LefttoRight

return:             popa
                    pop bp
                    ret
```

```
;---------------------------------------------------------------
; ALTERNATE SOLUTION
; Solution to this problem was developed by https://github.com/farhana1i
;---------------------------------------------------------------

; ; to display asterick movement every after 1 second
; [org 0x0100]

; jmp main

; seconds:  dw 0   ; number of seconds
; ticks:    dw 0   ; count of ticks
; isLeft:   db 0   ; left movement flag
; isRight:  db 0   ; right movement flag
; isTop:    db 0   ; up movement flag
; isBottom: db 0   ; down movement flag
; col:      db 0   ; current row number
; row:      db 0   ; current column number

; ; to clear video screen
; clrscr:
; push  es
; push  ax
; push  di

; mov   ax, 0xb800
; mov   es, ax
; mov   di, 0
```

```
; nextchar:
; mov   word [es:di], 0x720
; add   di, 2
; cmp   di, 4000
; jne   nextchar

; pop   di
; pop   ax
; pop   es
; ret


; ; to print asteric
; ; DI == position
; printAsterick:
; push  ax
; push  es

; mov   ax, 0xb800
; mov   es, ax        ; points to video memory

; mov   word [es: di], 0x0720  ; clear previous location

; cmp   byte [col], 0
; JNE   nextCmp

; cmp   byte [row], 0
; JNE    checkUp
; mov   byte [isLeft], 1
; mov   byte [isRight], 0
; mov   byte [isTop], 0
; mov   byte [isBottom], 0
; jmp   update

; checkUp:
; cmp   byte [row], 24
; JNE   nextCmp
; mov   byte [isLeft], 0
; mov   byte [isRight], 0
; mov   byte [isTop], 1
; mov   byte [isBottom], 0
; jmp   update

; nextCmp:
; cmp   byte [col], 158
; JNE   update

; cmp   byte [row], 0
; JNE   checkRight
; mov   byte [isLeft], 0
; mov   byte [isRight], 0
; mov   byte [isTop], 0
; mov   byte [isBottom], 1
; jmp   update

; checkRight:
```

```
; cmp    byte [row], 24
; JNE    update
; mov    byte [isLeft], 0
; mov    byte [isRight], 1
; mov    byte [isTop], 0
; mov    byte [isBottom], 0
; jmp    update

; update:
; cmp    byte [isLeft], 1
; JNE    checkRightFlag
; add    di, 2
; add    byte [col], 2
; jmp    printScreen

; checkRightFlag:
; cmp    byte [isRight], 1
; JNE    checkUpFlag
; sub    di, 2
; sub    byte [col], 2
; jmp    printScreen

; checkUpFlag:
; cmp    byte [isTop], 1
; JNE    checkDownFlag
; sub    di, 160
; sub    byte [row], 1
; jmp    printScreen

; checkDownFlag:
; cmp    byte [isBottom], 1
; JNE    printScreen
; add    di, 160
; add    byte [row], 1
; jmp    printScreen


; printScreen:
; mov    ah, 0x07    ; attribute
; mov    al, '*'
; mov    word [es: di], ax

; pop es
; pop ax
; ret


; ; hook timer interrupt service routine
; timer:
; push   ax

; inc    word [cs: ticks]
; cmp    word [cs: ticks], 18      ; 18.2 ticks per second
; jne    exitTimer
```

```
; inc    word [cs: seconds]        ; increase total seconds by 1
; mov    word [cs: ticks], 0
; CALL   printAsterick

; exitTimer:
; mov    al, 0x20       ; send EOI
; out    0x20, al
; pop    ax
; iret


; main:
; ;call   clrscr     ; to clear screen
; mov    di, 0
; xor    ax, ax
; mov    es, ax

; ; hook interrupt
; cli
; mov    word [es: 8*4], timer
; mov    [es: 8*4+2], cs
; sti

; ; to make program TSR
; mov    dx, main
; add    dx, 15
; mov    cl, 4
; shr    dx, cl
; mov    ax,-0x3100
; INT    0x21
```

Qno 8

```
; Solution to this problem was developed by https://github.com/farhana1i

[org 0x0100]

   jmp main

oldisr: dd 0       ; old isr offset and segment
buffer: times 2000 dw 0 ; buffer to save video memory

; to clear video screen
clrscr:
push   es
push   ax
push   di

mov    ax, 0xb800
mov    es, ax
mov    di, 0
nextchar:
mov    word [es:di], 0x720
add    di, 2
```

```
cmp   di, 4000
jne   nextchar

pop   di
pop   ax
pop   es
ret

; to add some delay
delay:
push  cx
push  di

mov   cx, 0xFF
delay1:
mov   di, 0xFFF
delay2:
dec   di
jnz   delay2
loop  delay1

pop   di
pop   cx
ret

; to store video memory in buffer
store_buffer:
push  bp
mov   bp, sp
push  ax
push  cx
push  si
push  di
push  es
push  ds

mov   ax, 0xb800  ; points to video memory
mov   ds, ax
mov   si, 0
mov   ax, cs
mov   es, ax
mov   di, buffer
mov   cx, 2000

cld
rep   movsw   ; move data from video memory to buffer

pop   ds
pop   es
pop   di
pop   si
pop   cx
pop   ax
pop   bp
ret
```

```
; load buffer
load_buffer:
push   bp
mov    bp, sp
push   ax
push   cx
push   si
push   di
push   es
push   ds

mov    ax, 0xb800     ; points to video memory
mov    es, ax
mov    di, 0
; points to buffer
mov    ax, cs
mov    ds, ax
mov    si, buffer
mov    cx, 2000

cld
rep    movsw      ; load buffer in video memory

pop    ds
pop    es
pop    di
pop    si
pop    cx
pop    ax
pop    bp
ret


; hook key board interrupt with interrupt chaining
kbISR:
push   ax
in     al, 0x60    ; read a char from keyboard

cmp    al, 00011101b  ; snap code of ctrl == 29
JNE    nextCmp

CALL   store_buffer   ; store video memory in a buffer
CALL   clrscr         ; clear screen
jmp    exit

nextCmp:
cmp    al, 10011101b   ; snap code of ctrl == 29
JNE    noMatch

CALL   delay          ; add some delay
CALL   load_buffer    ; load buffer in video memory
jmp    exit

noMatch:
pop    ax
jmp    far [cs:oldisr] ; CALL the original ISR
```

```
count:  dw 0, 0, 0, 0, 0                ;Count of the characters typed
tCount: dw -1                           ;Note: The tCount of first second is initialized to -1 for one time because
when you type the command and press ENTER
                                        ;       then the program gets loaded. And it takes you
a few milliseconds to release the ENTER key
                                        ;       and since the program was loaded before, it
counts this release of ENTER key as one. So this release count
                                        ;       is ignored by initializing the count to -1

iNo : dw 0

location: db 0                          ;Location where the next star is to be printed

;-------------------------------------------------------------------------------------------

;Clear Screen
clrscr:         pusha
                push es

                mov ax, 0xb800
                mov es, ax
                xor di,di
                mov ax,0x0720
                mov cx,2000

                cld
                rep stosw

                pop es
                popa
                ret
;-------------------------------------------------------------------------------------------

;Program to print the stars
printStars:     pusha
                push es

                mov ax, 0xb800
                mov es, ax

                mov al, 80
                mul byte [cs:location]
                add ax, 159
                shl ax, 1

                mov di, ax

                mov cx, [cs:tCount]

                cmp cx, 0
                jle return
```

```
l1:                     mov byte [es:di], '*'
                        inc byte [cs:location]
                        add di, 160
                        loop l1

return:         pop es
                        popa
                        ret

;-------------------------------------------------------------------------------

CTS:            pusha

                        ;These lines will execute for the very first five seconds
                        cmp word [cs:iNo], 10
                        jz l2

                        add word [cs:ms], 55
                        cmp word [cs:ms], 1000
                        jl EOI2

                        mov word [cs:ms], 0             ;Resetting the MilliSeconds to
zero
                        call printStars                 ;Because the count is to be
updated every second i.e
stars are to be printed after every second                              ;the

                        mov ax, [cs:tCount]
                        mov bx, [cs:iNo]

                        mov word [cs:count + bx], ax

                        mov word [cs:tCount], 0
                        add word [cs:iNo], 2

                        jmp EOI2


l2:                     add word [cs:ms], 55
                        cmp word [cs:ms], 1000
                        jl EOI2

                        mov word [cs:ms], 0             ;Resetting the MilliSeconds to
zero
                        ;Shifting the counts towards the right, to create a space for this current
second

                        mov dx, 0
```

```
        mov ax, [cs:count + 2]
        add dx, ax
        mov [cs:count], ax

        mov ax, [cs:count + 4]
        add dx, ax
        mov [cs:count + 2], ax

        mov ax, [cs:count + 6]
        add dx, ax
        mov [cs:count + 4], ax

        mov ax, [cs:count + 8]
        add dx, ax
        mov [cs:count + 6], ax

        mov ax, [cs:tCount]
        add dx, ax
        mov [cs:count + 8], ax

        jmp a1

EOI2.   jmp EOI
        ;Intermediate Jump

                ;Now dx contains the count of the last five seconds

a1              mov [cs:tCount], dx

                call clrscr

                mov byte [cs:location], 0
                call printStars

                mov word [cs:tCount], 0

EOI             mov al, 0x20
                out 0x20, al

exit            popa
                iret
```

Keyboard ISR

```
kbisr   push ax
```

```
                in al, 0x60

                shl al, 1
                jnc EOI1

                inc word [cs:tCount]        ;If a key is released, only then increase the count

EOI1    mov al, 0x20
                out 0x20, al

                pop ax

                iret
```

```
start           mov ax, 0
                mov es, ax

                mov bx, 0

                call clrscr

                ;Hooking the interrupts
                cli

                mov word [es: 9*4], kbisr
                mov [es:9*4+2], cs

                mov word [es:8*4], CTS
                mov [es:8*4+2], cs

                sti


                ;Code for making it TSR
                mov dx, start                   ;End of resident portion
                add dx, 15                      ;round up to
next para
                mov cl, 4
                shr dx, cl                      ;number of
paras

end:            mov ax, 0x3100                  ;terminate and stay resident
                int 21h
```

```
    cmp     di, 4000
    jne     nextchar

    pop     di
    pop     ax
    pop     es
    ret

    , to add some delay
    delay:
    push    cx
    push    di

    mov     cx, 0xFF
    delay1:
    mov     di, 0xFFF
    delay2:
    dec     di
    jnz     delay2
    loop    delay1

    pop     di
    pop     cx
    ret

    ; to store video memory in buffer
    store_buffer:
    push    bp
    mov     bp, sp
    push    ax
    push    cx
    push    si
    push    di
    push    es
    push    ds

    mov     ax, 0xb800  ; points to video memory
    mov     ds, ax
    mov     si, 0
    mov     ax, cs
    mov     es, ax
    mov     di, buffer
    mov     cx, 2000

    cld
    rep     movsw   ; move data from video memory to buffer

    pop     ds
    pop     es
    pop     di
    pop     si
    pop     cx
    pop     ax
    pop     bp
    ret
```

Write a function "addtoset" that takes offset of a function and remembers this offset in an array that can hold a maximum of 8 offsets. It does nothing if there are already eight offsets in the set. Write another function "callset" that makes a call to all functions in the set one by one.

```
[org 0x0100]
                jmp start
arr     dw 0,0,0,0,0,0,0,0
start           push word testFunct
                call addtoset
                push word testFunct
                call addtoset
                call callset
end             mov ax, 0x4c00
                int 21h
;An implied operand say any register which stores the count of the offsets in the array
;will make the solution simpler
addtoset:       push bp
                mov bp, sp
                pusha
                mov ax, 0
                mov bx, 0
                mov dx, [bp + 4]        ;Offset to be copied
loop1:          cmp ax, [arr + bx]
                jnz  skip
                mov [arr + bx], dx
                jmp return
skip:           add bx, 2
                cmp bx, 16
                jnz loop1
return:         popa
                pop bp
                ret 2
;An implied operand say any register which stores the count of the offsets in the array
;will make the solution simpler
callset:        push bp
                mov bp, sp
                pusha
                mov ax, 0
                mov bx, 0
_loop1:         cmp ax, [arr + bx]
                jz  skipcall
                call [arr + bx]
skipcall:       add bx, 2
                cmp bx, 16
                jnz _loop1
_return:        popa
                pop bp
                ret
testFunct:      ;Does nothing.
                ret
```

Do the above exercise such that "callset" does not use a CALL or a JMP to invoke the functions. HINT: Setup the stack appropriately such that the RET will execute the first function, its RET execute the next and so on till the last RET returns to the caller of "callset."

```
[org 0x0100]
                jmp start
arr:    dw 0,0,0,0,0,0,0,0
start:          push word testFunct
                call addtoset
                push word testFunct
                call addtoset
                pusha
```

```
                call callset
                popa
                mov ax, 0x4c00
end:            int 21h
;An implied operand say any register which stores the count of the offsets in the array
;will make the solution simpler
addtoset:       push bp
                mov bp, sp
                pusha
                mov ax, 0
                mov bx, 0
                mov dx, [bp + 4]        ;Offset to be copied
                cmp ax, [arr + bx]
loop1:          jnz  skip
                mov [arr + bx], dx
                jmp return
                add bx, 2
skip:           cmp bx, 16
                jnz loop1
                popa
return:         pop bp
                ret 2
;An implied operand say any register which stores the count of the offsets in the array
;will make the solution simpler
callset:    mov ax, 0
                mov bx, 14
_loop1:     cmp ax, [arr + bx]
                jz _skip
                push word [arr + bx]
_skip:      sub bx,  2
                cmp bx, -2
                jnz _loop1
_return:    ret
testFunct:  ;Does nothing.
                ret
```

CHAPTER 6

Write an infinite loop that shows two asterisks moving from right and left centers of the screen to the middle and then back. U
empty nested loops with large counters to introduce some delay so that the movement is noticeable.

```
[org 0x0100]
                jmp start
character: dw '*'
start:          call clrscr
                push word [character]
                call clash
end:            mov ax, 0x4c00
                int 21h
clrscr:         mov ax, 0xb800
                mov es, ax
                xor di,di
                mov ax,0x0720
                mov cx,2000
                cld
                rep stosw
                ret
clash:      push bp
                mov bp,sp
                pusha
                mov ax, 0xb800
                mov es, ax
```

```
                                        mov bx, 12
                                        ;Calculating the starting position
                                        mov al, 80
                                        mul bl
                                        shl ax, 1
                                        mov si, ax
                                        mov di, si
                                        add di, 158
                                        mov cx, 38
                                        ;Loading the characters
Printing1:                              mov al, [bp + 4]
                                        mov ah, 0x07
                                mov word [es:si], ax
                                        mov word [es:di], ax
                                        call _delay
                                        call _delay
                                        mov word [es:si], 0x0720
                                        mov word [es:di], 0x0720
                                        add si, 2
                                        sub di, 2
Printing2:                              loop Printing1
                                        mov cx, 38
                                mov word [es:si], ax
                                        mov word [es:di], ax
                                        call _delay
                                        call _delay
                                        mov word [es:si], 0x0720
                                        mov word [es:di], 0x0720
                                        sub si, 2
                                        add di, 2
                                        loop Printing2
                                        mov cx, 38
_delay:                                 jmp Printing1
l1:                             mov dx, 0xFFFF
                                        dec dx
                                        jnz l1
                                        ret
return:                         popa
                                        pop bp
                                        ret 2
```

Write a function "printaddr" that takes two parameters, the segment and offset parts of an address, via the stack. The function should print the physical address corresponding to the segment offset pair passed at the top left of the screen. The address sh be printed in hex and will therefore occupy exactly five columns. For example, passing 5600 and 7800 as parameters should re in 5D800 printed at the top left of the screen.

```
[org 0x0100]
        jmp start
_segment:   dw 0xF8AB
_offset:    dw 0xFFFF
start:      call clrscr
                push word [_segment]
                push word [_offset]
                call printaddr
end:        mov ax, 0x4c00
                int 21h
clrscr:                 mov ax, 0xb800
                                mov es, ax
                                xor di,di
                                mov ax,0x0720
                                mov cx,2000
                                cld
                                rep stosw
```

```
                    ret

;A mini sub routine to used by printaddr
                    cmp bl, 9
print:              jle Decimal
                    jg  Hex


Decimal:    add bl, 0x30
                    jmp l1

                add bl, 55
Hex:                jmp l1
                    mov word [es:di], bx

l1:                 add di, 2

                ret
return:
;Main sub-routine
printaddr:  push bp
                    mov bp,sp
                    pusha
                    mov ax, 0xb800
                    mov es, ax
                    ;Calculating the Physical Address
                    mov ax, [bp + 6]            ;segment address
                    mov bx, 0x10
                    mul bx
                    add ax, [bp + 4]    ;adding the offset
                    adc dx, 0
                    mov di, 0
                    mov bh, 0x07
;Printing Most Signicant Nibble of PA present in dx
Nibble_1st:         mov bl, 00001111b
                    and bl, dl
                    call print

;Printing the ax part of PA
Nibble_2nd:         mov bl, 11110000b
                    and bl, ah
                    shr bl, 4
                    call print
Nibble_3rd:         mov bl, 00001111b
                    and bl, ah
                    call print
Nibble_4th:         mov bl, 11110000b
                    and bl, al
                    shr bl, 4
                    call print
Nibble_5th:         mov bl, 00001111b
                    and bl, al
                    call print
_return:    popa
                    pop bp
                    ret 4
```

Write code that treats an array of 500 bytes as one of 4000 bits and for each blank position on the screen (i.e. space) sets the corresponding bit to zero and the rest to one.

```
[org 0x0100]
        jmp start
arr: times 500 db 0
start:      call spacechecker
end:        mov ax, 0x4c00
                int 21h
spacechecker:           pusha
                        mov ax, 0xb800
                        mov es, ax
                        ;Attributes wali byte locations par tw kabhi space nahi miley gi.
```

```
                                        ;Isi liye array ki odd numbered bits ko pehley hi 1 kar do.
                                        ;Firstly setting all bits to 1
                                        mov cx, 2000
                                        mov ax, 1111111111111111b
                        or [arr + bx], ax
_setAllBits:                            add bx, 2
                                        loop _setAllBits
                                        mov bl, 0x20                ;loading the space character
                                        ;Now checking for spaces
                                        mov cx, 2000
                                        mov di, 0
                                        mov si, 0
                                        mov dl, 10000000b
                                        mov al, 01111111b
                                cmp byte [es:di], bl
checkSpace:                             jnz _bit1
                                        jz  _bit0
                                and [arr + si], al
_bit0:                                  jmp l1
                                or [arr + si], dl
_bit1:                                  jmp l1
                                        shr dl, 2       ;Skipping the odd numbered bits as they are already set
l1:
to 1                                    shr al, 2       ;previously
                                        cmp dl, 0
                                        jnz l2
                                        mov dl, 10000000b
                                        mov al, 01111111b
                                        inc si
l2:                                     add di, 2
                                        loop  checkSpace
return:                         popa
                                        ret
```

Write a function "drawrect" that takes four parameters via the stack. The parameters are top, left, bottom, and right in this order. The function should display a rectangle on the screen using the characters + - and |.

```
[org 0x0100]
        jmp start
top:    dw 10          ;Starting Row
bottom: dw 20                  ;Ending   Row
left:   dw 30                          ;Starting Column
right:  dw 60                  ;Ending Column
start:              call clrscr
                            push word [top]
                            push word [bottom]
                            push word [left]
                            push word [right]
                            call drawrect
end:                mov ax, 0x4c00
                            int 21h
clrscr:                     mov ax, 0xb800
                                    mov es, ax                  ;Loading the video memory
                                    xor di,di
                                    mov ax,0x0720
                                    mov cx,2000
                                    cld
                                    rep stosw
                                    ret
drawrect:           push bp
                                    mov  bp, sp
                                    pusha
                                    ; bp + 4  =  right
```

```
                    ; bp + 6  =  left
                    ; bp + 8  =  bottom
                    ; bp + 10 =  top
                    ;Calculating the top left position of the rectangle
                    mov al, 80
                    mul byte [bp + 10]
                    add ax,  [bp + 6]
                    shl ax, 1
                    mov di, ax                          ;Saving for later use
                    push di
                    mov ah, 0x07                        ;Storing the attribute
                    ;Calculating the width of the rectangle
                    mov cx, [bp + 4]
                    sub cx, [bp + 6]                    ;Saving for later use
                    push cx
                    mov al, '+'
                rep stosw
loop1:              pop bx
                    pop di
                    push bx
                    dec bx
                    shl bx, 1
                    add di, 160
                    ;Calculating the height of the rectangle
                    mov cx, [bp + 8]
                    sub cx, [bp + 10]                   ;Excluding the top and bottom row
                    sub cx, 2
                    mov al, '|'
                mov si, di
loop2:              mov word [es:si], ax
                    add si, bx
                    mov word [es:si], ax
                    sub si, bx
                    add di, 160
                    loop loop2
                    pop cx
                    mov al, '-'
                rep stosw
loop3:          popa
return:             pop bp
                    ret 8
```

Write code to find the byte in AL in the whole megabyte of memory such that each memory location is compared to AL only once.

```
[org 0x0100]
        jmp start
flag:   db 0                                            ;Byte to find
start:  mov al, 0x0F                                    ;Starting from segment 0x0000
                mov bx, 0x0000
                mov es, bx
l1:             mov cx, 0xFFFF
                mov di, 0
                repne scasb
                je found
                add bx, 1000
                cmp bx, 0000
                jz notFound
                jnz l1
found:      mov byte [flag], 1
                jmp exit
notFound:   jmp exit
exit:       mov ax, 0x4c00
                int 21h
```

Write a far procedure to reverse an array of 64k words such that the first element becomes the last and the last becomes the first and so on. For example if the first word contained 0102h, this value is swapped with the last word. The next word is swapped with the second last word and so on. The routine will be passed two parameters through the stack; the segment and offset of the first element of the array.

```
[org 0x0100]
                jmp start
_segment:               dw 0x3000
_offset :       dw 0x1000
start:                  push word [_segment] ;
                        push word [_offset] ;
                        call reverseArray
                mov ax, 0x4c00
end:                    int  21h

reverseArray:   push bp
                        mov bp, sp
                        pusha
                        mov cx, 0xFFFF;To compare 64k words, the comparision should be done for 32k words onl
                        mov ds, [bp + 6]                ;Starting Segment
                        mov si, [bp + 4]                ;Starting Offset
                        mov di, si                              ;Ending Offset
                        mov ax, ds
                        add ax, 0x2000                  ;going to the third segment
                        mov es, ax
                        std                                             ;set direction flag
                        cmp di, 0
scenario0:              jnz loop1                       ;We have three overlapping segments
scneario1:              mov dx, es
                        sub dx, 0x1000          ;We have two non-overlapping segments
                        mov es, dx
                        mov di, 0xFFFE
loop1:                  mov ax, [es : di]
                        movsw
                        add si, 2                       ;because 2 has been subtracted by movsw
                        mov [si], ax            ;swapping values
                        add si, 2
                        cmp si, 0xFFFF          ;check if the segment has ended
                        jne l1
                        mov dx, ds
                        add dx, 0x1000          ;if ended move on to the next segment
                        mov ds, dx
                        mov si, 0                       ;resetting si
l1:                     cmp di, 0xFFFE          ;if the last seg has ended
                        jne l2
                        mov dx, es
                        sub dx, 0x1000          ;going to the 2nd segment backward
                        mov es, dx
                        mov di, 0xFFFE          ;resetting to last word
l2:                     loop loop1
return:         popa
                        pop bp
                        ret 4
```

Write a near procedure to copy a given area on the screen at the center of the screen without using a temporary array. The routine will be passed top, left, bottom, and right in that order through the stack. The parameters passed will always be within range, the height will be odd and the width will be even so that it can be exactly centered.

```
[org 0x0100]
        jmp start
top:    dw 17
bottom: dw 20
left:   dw 15
right:  dw 30
```

```
start:          push word [top]
                push word [left]
                push word [bottom]
                push word [right]
                call copyAtCenter

end:            mov ax, 0x4c00
                int 21h

copyAtCenter:   push bp
                mov bp, sp
                pusha
                push es
                push ds
                ;bp+4 = right
                ;bp+6 = bottom
                ;bp+8 = left
                ;bp+10 = top
                mov ax, 0xB800
                mov es, ax
                ;Center of screen
                ;Row = 12                                   ;Mid Col
                ;Col = 39,40                                ;Mid Row
                mov bx, 39
                mov dx, 12
                ;Calculating Width
                mov ax, [bp + 4]
                sub ax, [bp + 8]                            ;Saving width for later use
                push ax
                sub ax, 2
                shr ax, 1
                ;Getting to the required starting column
                sub bx, ax
                ;Calculating height
                mov ax, [bp + 6]
                sub ax, [bp + 10]                           ;Saving height for later use
                push ax
                sub ax, 1
                shr ax, 1
                ;Getting to the required starting row
                sub dx, ax
                ;Staring position of source
                mov al , 80
                dec byte [bp + 10]
                mul byte [bp + 10]   ;Top
                dec byte [bp + 8]
                add ax, [bp + 8]     ;Left
                shl ax, 1
                mov si, ax
                ;Starting position of destination
                mov al, 80                                  ;Load al with columns per row
                mul dl                                      ;Multiply with y position
                add ax, bx                                  ;add x position
                shl ax, 1
                mov di, ax
                pop ax                      ;Height
                pop cx                      ;Width
                push es
                pop  ds
                mov bx, 0
                ;Now moving the area to the center
l1:             push si
                push di
```

```
                push cx
                rep movsw
                pop cx
                pop di
                pop si
                add si, 160
                add di, 160
                inc bx
                cmp bx, ax
                jnz l1
                pop ds
                pop es
return:         popa
                pop bp
                ret 8
```

Write code to find two segments in the whole memory that are exactly the same. In other words find two distinct values which if loaded in ES and DS then for every value of SI [DS:SI]=[ES:SI].

```
[org 0x0100]
start:          sub sp, 2                                   ;return value
                call findEqualSegments
                pop bx
        ;ax = 1 indicates two equal segments are found otherwise
        ;bx = 0
end:            mov ax, 0x4c00
                int  21h
findEqualSegments:push bp
                mov bp, sp
                pusha
                mov word [bp + 4], 0
                mov ax, 0
                mov dx, 0
                mov si, 0
                mov di, 0
                mov cx, 0xFFFF
                ;Finding tw non-overlapping and equal segments
                ;There are a total of 16 distinct segments in a memory of 1MB
                mov ds, ax              ;Starting from the segment 0x0000 (1st Segment)
                mov ax, 0x1000
                mov es, ax                          ;2nd Segment
                cld
loop1:          repe cmpsb              ;repeat while equal cx times
                je areEqual             ;if the segments were equal
check_ES:       mov ax, es
                cmp ax, 0xF000          ;checking for the last segment (16th Segment)
                jz  check_DS
                mov ax, es                  ;Next non-overlapping segment
                add ax, 0x1000
                mov es, ax
                mov di, 0
                mov si, 0
                mov cx, 0xFFFF
                jmp loop1
check_DS:       mov ax, ds
                cmp ax, 0xF000 ;If DS = 0xF000, it means we are at the last segment, and this
                ;segment doesn't need to be compared with itself. So no further
                ;processing is to be done and we haven't found two ;equal segments
                jz return
                mov ax, ds                  ;Next non-overlapping segment
                add ax, 0x1000
                mov ds, ax
                add ax, 0x1000
```

```
                                mov es, ax
                                mov si, 0
                                mov di, 0
                                mov cx, 0xFFFF
                                jmp loop1
                                        ;Two equal segments are found
areEqual:       mov word [bp + 4], 1
                        popa
return:                         pop bp
                                ret
```

;Two overlapping and equal segments can be found, but the processing takes too much time.
;Anyways, the code for that is given below
;instead of adding 0x1000, now 0x0001 is being added and instead of comparing with 0xF000, now the comparision
;is being done with 0xFFFF

```
                                mov ds, ax              ;Starting from the segment 0x0000 (1st Segment)
                                mov ax, 0x0001                  ;2nd Segment
                                mov es, ax
                                cld                     ;repeat while equal cx times
                        repe cmpsb                      ;if the segments were equal
;loop1:                                 je areEqual                     ;checking for the last segment
                        cmp es, 0xFFFF
;check_ES:                              jz  check_DS            ;Next overlapping segment
                                mov ax, es
                                add ax, 0x0001
                                mov es, ax
                                mov di, 0
                                mov si, 0
                                mov cx, 0xFFFF
                                jmp loop1
                        cmp ds, 0xFFFF  ;If DS = 0xFFFF, it means we are at the last segment, and this
;check_DS:
        ;segment doesn't need to be compared with itself. So no further
                                        ;processing is to be done and we haven't found two equal

segments                        jz return                               ;Next overlapping
                                mov ax, ds

segment                         add ax, 0x0001
                                mov ds, ax
                                add ax, 0x0001
                                mov es, ax
                                mov si, 0
                                mov di, 0
                                mov cx, 0xFFFF
                                jmp loop1
```

Write a function "strcpy" that takes the address of two parameters via stack, the one pushed first is source and the second is the destination. The function should copy the source on the destination including the null character assuming that sufficient space is reserved starting at destination.

```
[org 0x0100]
start:          push src
                        push dest
                        call strcpy
end:            mov ax, 0x4c00
                        int 21h
strLen:                 push bp
                                mov bp, sp
                                pusha
                                push es
                                push ds
                                pop es
                                mov di, [bp+4]          ;Point di to string
                                mov cx, 0xFFFF          ;Load Maximum No. in cx
```

ways left scrolling

```
[org 0x0100]
jmp start

movepixels:

push bp
mov bp,sp

push ax
push bx
push cx
push si
push di
push es
push ds

 mov ax, 0xb800
 mov es, ax
 mov ds, ax
 mov si, [BP+4]  <- no. of cols to move left.
 shl si,1        ;MUL BY 2
 sub si,2
 mov di, 0
 mov bx, 0

  loop1:

  mov cx, 80



   rep movsb
   add si, 80
   add di, 80
   add bx, 1
   cmp bx, 25
   jne loop1


;_____________________________________
pop ds
pop es
pop di
pop si
pop cx
pop bx
pop ax
pop bp

 ret 2
```

```
start:

mov ax,5        ;how many cols to move left
push ax  ;[BP+4]

call movepixels

mov ax, 0x4c00
int 0x21
```

copying right half of the screen to the left

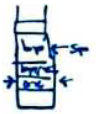
```
[org 0x0100]
jmp start

movepixels:

push ax
push bx
push cx
push si
push di
push es
push ds

 mov ax, 0xb800
 mov es, ax
 mov ds, ax
 mov si, 80
 mov di, 0
 mov bx, 0

 loop1:

 mov cx, 80
 cld
 rep movsb
 add si, 80
 add di, 80
 add bx, 1
 cmp bx, 25
 jne loop1

  pop ds
  pop es
  pop di
  pop si
  pop cx
  pop bx
  pop ax

  ret

  start:

  call movepixels
  mov ax, 0x4c00
  int 0x21
```

```
;copying left half of the screen to the right

[org 0x0100]
jmp start

movepixels:

push ax
push bx
push cx
push si
push di
push es
push ds

mov ax, 0xb800
mov es, ax
mov ds, ax
mov si, 0
mov di, 80
mov bx, 0

loop1:

mov cx, 80
cld
rep movsb
add si, 80
add di, 80
add bx, 1
cmp bx, 25
jne loop1

pop ds
pop es
pop di
pop si
pop cx
pop bx
pop ax

ret

start:

call movepixels
mov ax, 0x4c00
int 0x21
```

Asc
a : 9
A : 6
spac
hex

Write a recursive function to calculate the Fibonacci of a number. The number is passed as a parameter via the stack and the calculated Fibonacci number is returned in the AX register. A local variable should be used to store the return value from the first recursive call. Fibonacci function is defined as follows:

Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)



```
[org 0x0100]
start:      mov ax, 4
            sub sp,2
            push ax
            call fibonacci
            pop ax
end:        mov ax, 0x4c00
            int 21h
fibonacci:  push bp
            mov bp,sp
            sub sp,2
            pusha
            mov ax, [bp + 4]
basecase1:  cmp ax,1
            jnz basecase2
            mov word [bp + 6],1
            jmp return
basecase2:  cmp ax,0
            jnz calls
            mov word [bp + 6],0
            jmp return
calls:      sub sp,2
            dec ax
            push ax
call1:      call fibonacci
            pop word [bp - 2]      ;A local variable used to store the return value from the first
                                   ;recursive call
            sub sp,2
            dec ax
            push ax
call2:      call fibonacci
            pop dx
            add dx, [bp - 2]
            mov [bp + 6],dx
return:     popa
            add sp,2
            pop bp
            ret 2
```

Write the above Fibonacci function iteratively. HINT: Use two registers to hold the current and the previous Fibonacci numbers in a loop.

```
[org 0x0100]
start:      mov ax, 6
            sub sp,2
            push ax
            call fibonacci
            pop ax
end:        mov ax, 0x4c00
            int 21h

fibonacci:  push bp
            mov bp,sp
            pusha
```

```
                    mov ax, [bp + 4]
                    mov word [bp + 6], 0        ;Initializing the return value to 0
case1:      cmp ax,1
                    jnz case0
                    mov word [bp + 6], 1
                    jmp return
case0:      cmp ax,0
                    jnz l1
                    mov word [bp + 6], 0
                    jmp return
l1:                 mov dx, 0
                    mov bx, 0                   ;bx= F(0) = 0
                    mov cx, 1                   ;cx= F(1) = 1
loop1:      cmp ax,1
                    jz return
                    mov dx,cx
                    add dx,bx
                    mov [bp + 6],dx
                    mov bx, cx
                    mov cx, [bp + 6]
                    dec ax
                    jmp loop1
return:     popa
                    pop bp
                    ret 2
```

Write a function switch_stack meant to change the current stack and will be called as below. The function should destroy no registers.

```
push word [new_stack_segment]
push word [new_stack_offset]
call switch_stack
[org 0x0100]
            jmp start
new_stack_segment: dw 0x1234
new_stack_offset:  dw 0xFFFE
start:      mov ax,0xABCD   ;Test values
            mov cx,0
            push ax
            push 123
            push word [new_stack_segment]
            push word [new_stack_offset]
            call switch_stack
            pop cx
            pop bx
end:        mov ax, 0x4c00
            int 21h
switch_stack:   push bp
            mov bp,sp
            pusha
            mov bx,sp
            sub bx,2
            mov si,0xFFFC       ;si will be used to make a copy of the old stack and it is
currently pointing at the bottom element of the old stack
            mov sp, [bp + 4]        ;new offset
            mov ss, [bp + 6]        ;new stack segment
oop1:       push word [si]
            sub si,2
            cmp si,bx
            jnz loop1
eturn:      popa
            pop bp
            ret 4
```