PRACTICAL: 1(A)

AIM: Introduction to 8086 microprocessor and assembly language programming.

THEORY:

- Assembly language is a low level programming language.
- 8086 CPU has 8 general purpose registers, each register has its own name:
  - AX - the accumulator register (divided into AH / AL).
  - BX - the base address register (divided into BH / BL).
  - CX - the count register (divided into CH / CL).
  - DX - the data register (divided into DH / DL).
  - SI - source index register.
  - DI - destination index register.
  - BP - base pointer.
  - SP - stack pointer.
- Segment registers
  - CS - points at the segment containing the current program.
  - DS - generally points at segment where variables are defined.
  - ES - extra segment register, it's up to a coder to define its usage.
  - SS - points at the segment containing the stack.
- Special purpose registers
  - IP - the instruction pointer.
  - flags register - determines the current state of the microprocessor.

CONCLUSION:

We learned about 8086 and it’s architecture.

PRACTICAL: 2(A1)

AIM: Store the data byte 32H into memory location 4000H

CODE:

```
org 100h

; add your code here

mov ax, 5000h
mov ds, ax
mov [4000h], 32h

ret
```

OUTPUT:

CONCLUSION:

We learned about how to store data into certain location.

PRACTICAL: 2(A2)

AIM: Exchange the contents of memory locations 2000H and 4000H

CODE:

```
org 100h

; add your code here

mov ax, 4000h
mov ds, ax

mov [2000h], 40h
mov [4000h], 10h

mov bl, [2000h] ; transfer data of 2000 location to the register bl

xchg bl, [4000h] ; exchanging values of reg bl and 4000 location

mov [2000h], bl ;

ret
```

OUTPUT:

CONCLUSION:

We learned to exchange the data of two locations.

PRACTICAL: 2(A3)

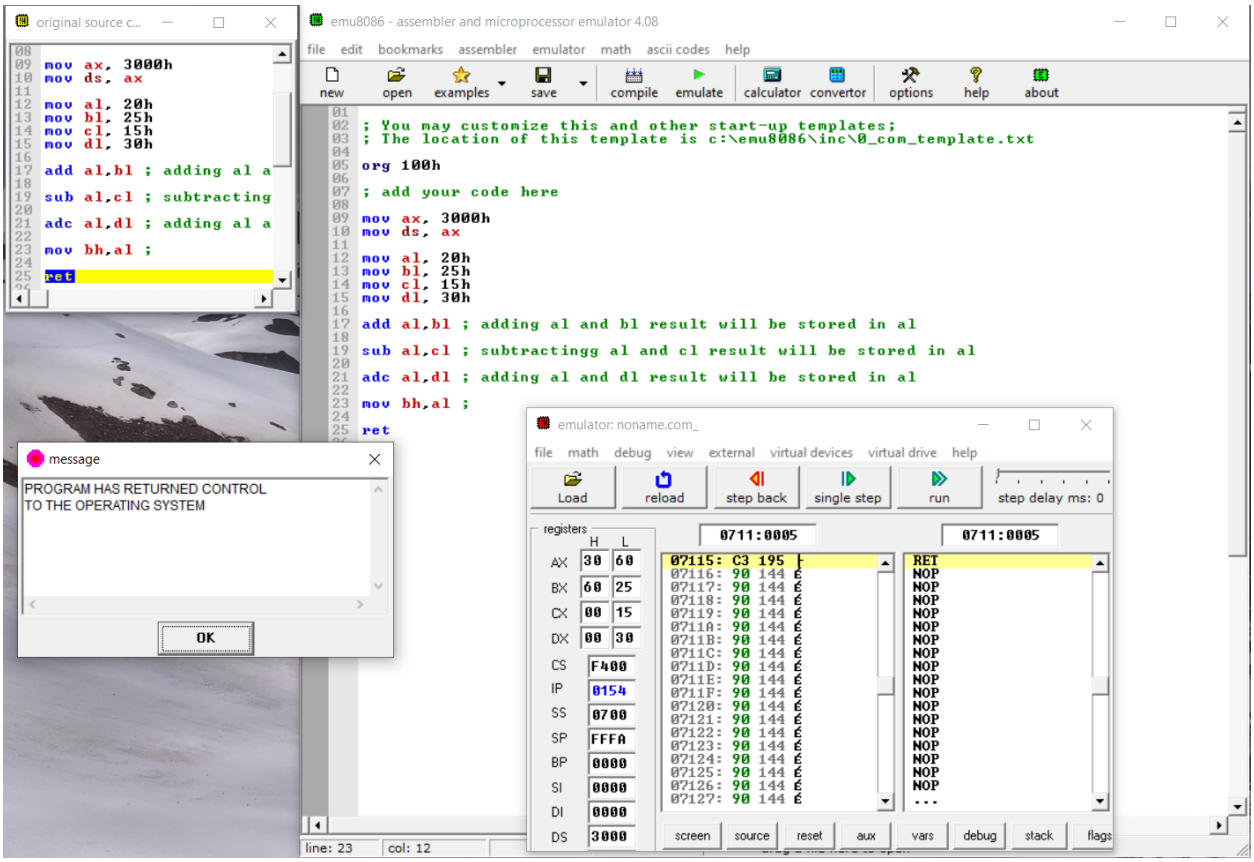
AIM: Convert the C program into Assembly language

```
main() {
int
l,m,n,o,p
; l=m+n-
o+p; }
```

CODE:

```
org 100h

mov ax, 3000h
mov ds, ax

mov al, 20h
mov bl, 25h
mov cl, 15h
mov dl, 30h

add al,bl ; adding al and bl result will be stored in al

sub al,cl ; subtractingg al and cl result will be stored in al

adc al,dl ; adding al and dl result will be stored in al

mov bh,al ;

ret
```

OUTPUT:



CONCLUSION:

We learned to interpret and convert code into C language from Assembly language.

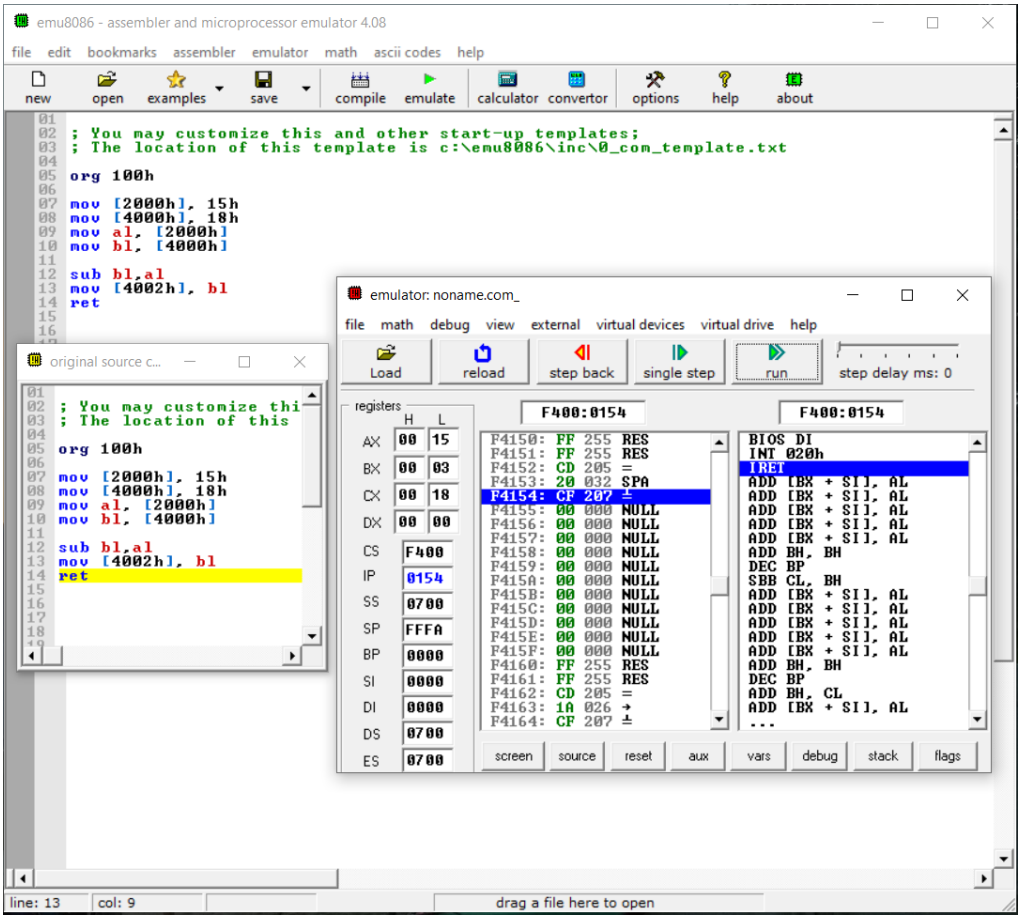
PRACTICAL: 2(B4)

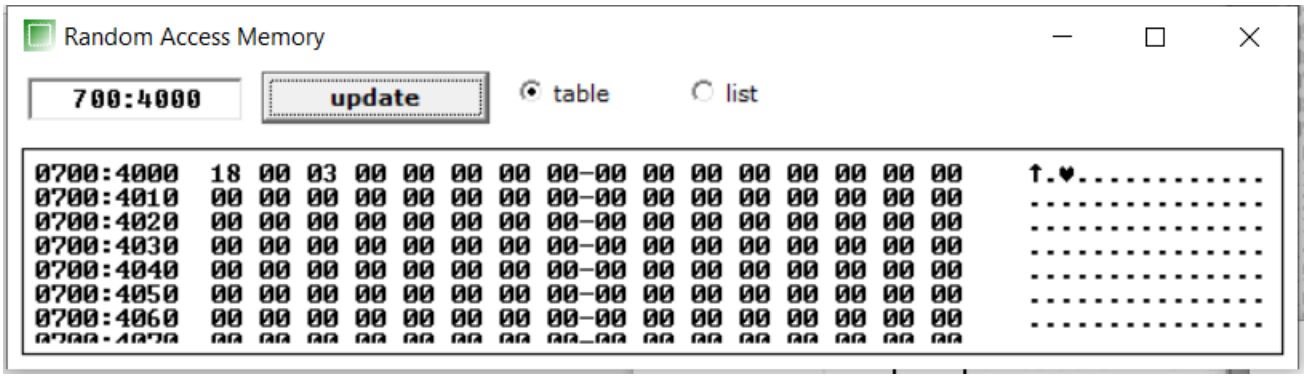
AIM: Subtract the contents of memory location 4001H from the memory location 2000H and place the result in memory location 4002H.

CODE:

```
org 100h

mov [2000h], 15h
mov [4000h], 18h
mov al, [2000h]
mov bl, [4000h]

sub bl,al
mov [4002h], bl
ret
```

OUTPUT:





CONCLUSION:

We learned about SBB command and it's implementation.

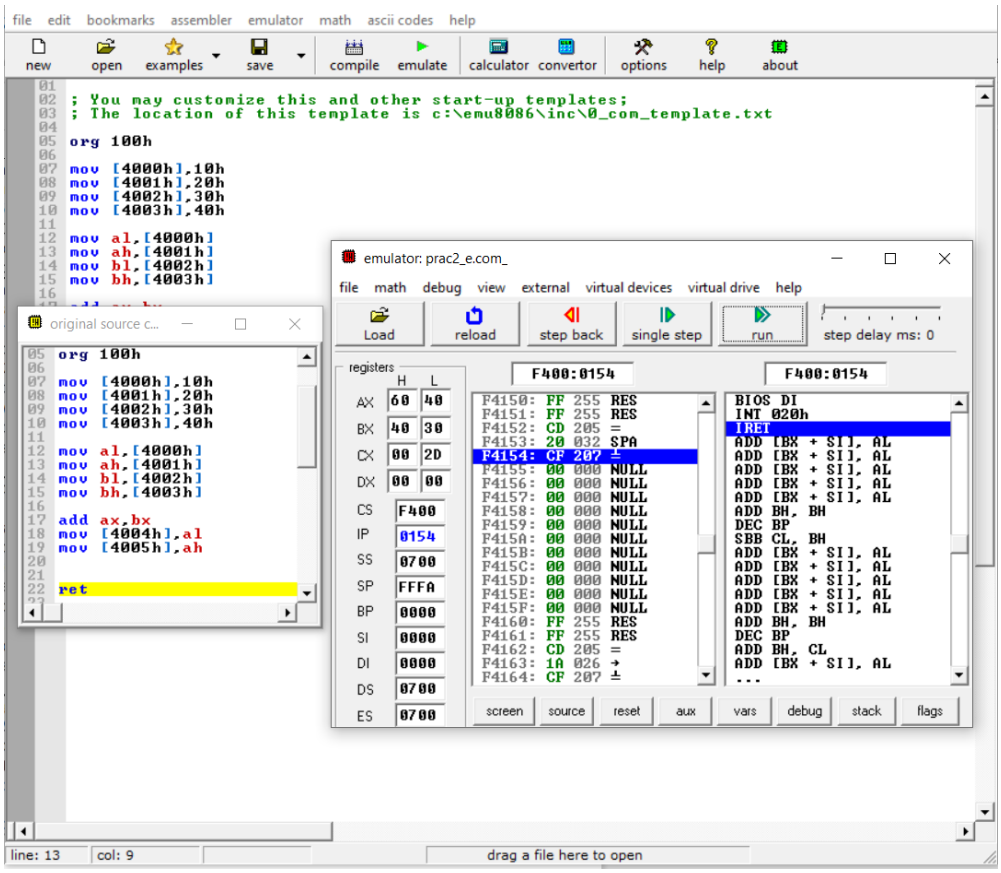
PRACTICAL: 2(B5)

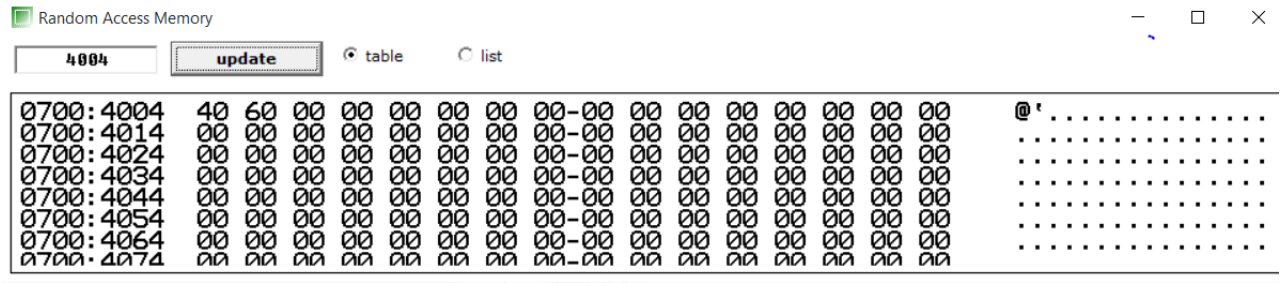
AIM: Add the 16-bit number in memory locations 4000H and 4001H to the 16-bit number in memory locations 4002H and 4003H. The most significant eight bits of the two numbers to be added are in memory locations 4001H and 4003H. Store the result in memory locations 4004H and 4005H with the most significant byte in memory location 4005H.

CODE:

```
org 100h

mov [4000h],10h
mov [4001h],20h
mov [4002h],30h
mov [4003h],40h

mov al,[4000h]
mov ah,[4001h]
mov bl,[4002h]
mov bh,[4003h]

add ax,bx
mov [4004h],al
mov [4005h],ah


ret
```

OUTPUT:





CONCLUSION:

We learned about ADC command and its implementation for 16-Bit Numbers.

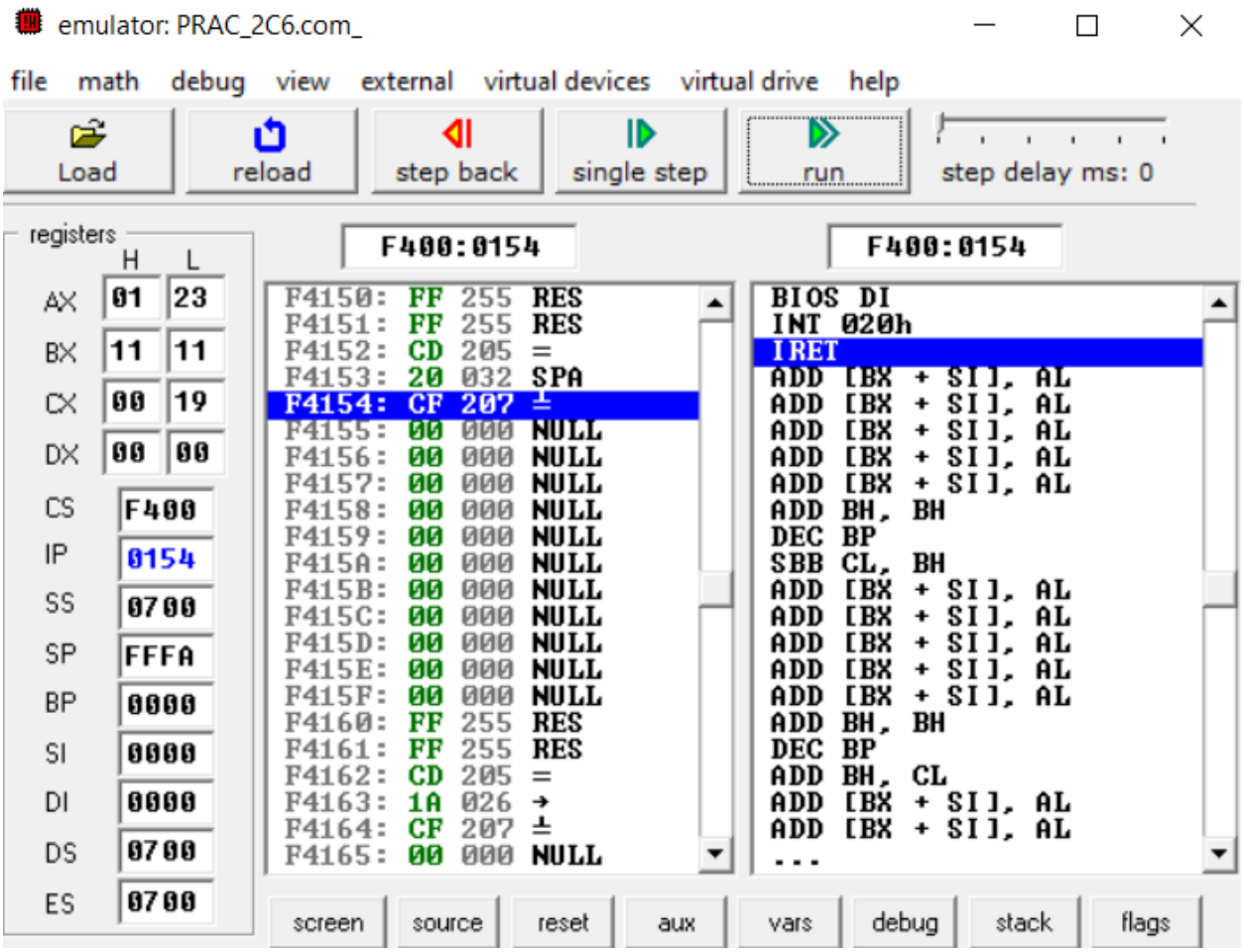
PRACTICAL: 2(C6)

AIM: Subtract the 16-bit number in memory locations 4002H and 4003H from the 16-bit number in memory locations 4000H and 4001H. The most significant eight bits of the two numbers are in memory locations 4001H and 4003H. Store the result in memory locations 4004H and 4005H with the most significant byte in Memory location 4005H.

CODE:

```
org 100h

MOV [4000H], 1234H
MOV [4002H], 1111H

MOV AX, [4000H]
MOV BX, [4002H]

SBB AX, BX
MOV [4004H], AX

ret
```

OUTPUT:



CONCLUSION:

We learned about SBB command and its implementation for 16-Bit Numbers.

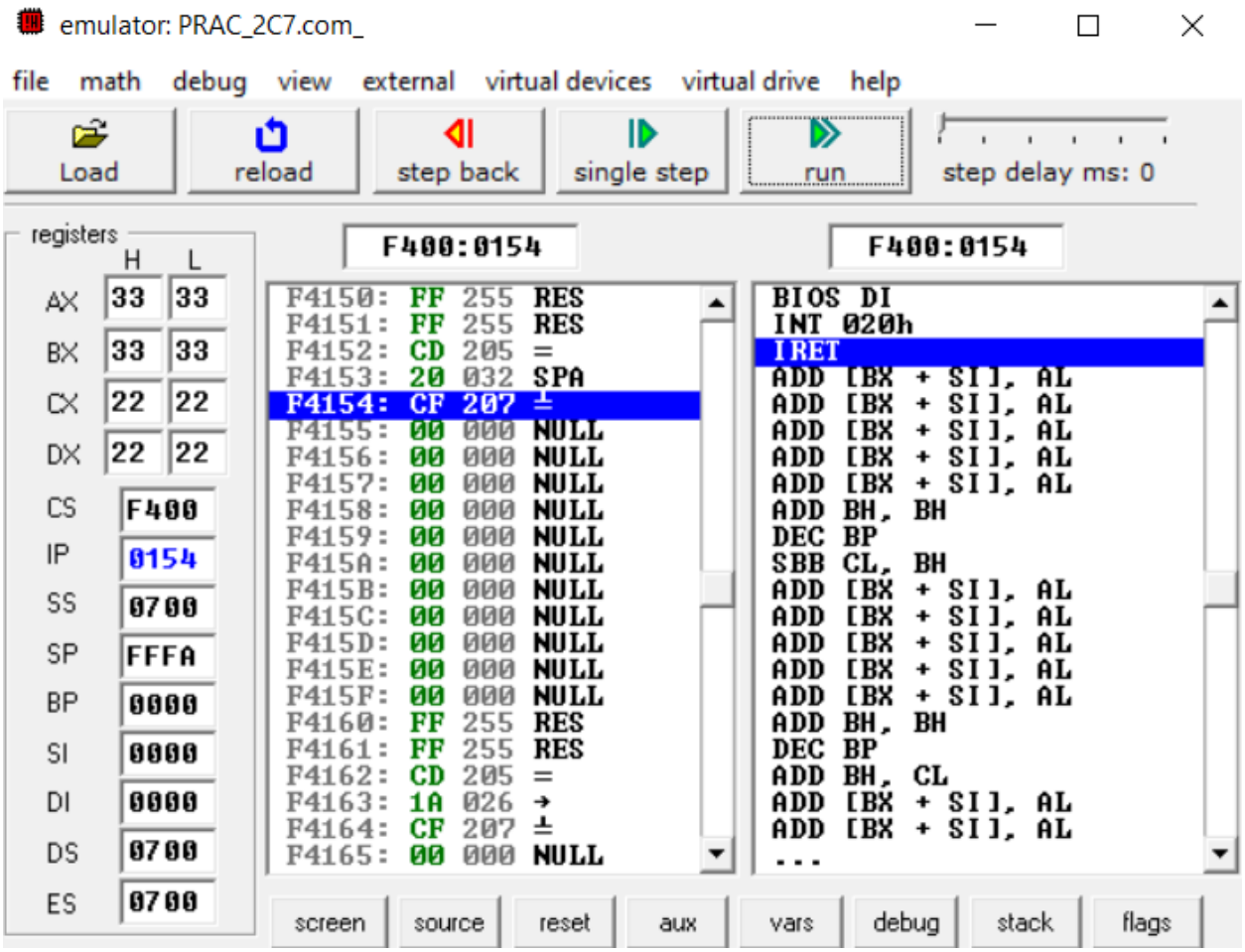
PRACTICAL: 2(C7)

AIM: Add Two 32-bit numbers stored in consecutive memory locations and store the result in memory locations starting from 7000H

CODE:

```
org 100h

MOV [4000H], 1111H
MOV [4002H], 1111H
MOV [4004H], 2222H
MOV [4006H], 2222H

MOV AX, [4000H]
MOV BX, [4002H]
MOV CX, [4004H]
MOV DX, [4006H]

ADD AX, CX    ;ADC SUPPORTS 16BIT ADDITION ONLY
ADC BX, DX    ;MSB

MOV [7000H], AX
MOV [7002H], BX

ret
```

OUTPUT:



CONCLUSION:

We learned about ADC command and its implementation for 32-Bit Numbers.

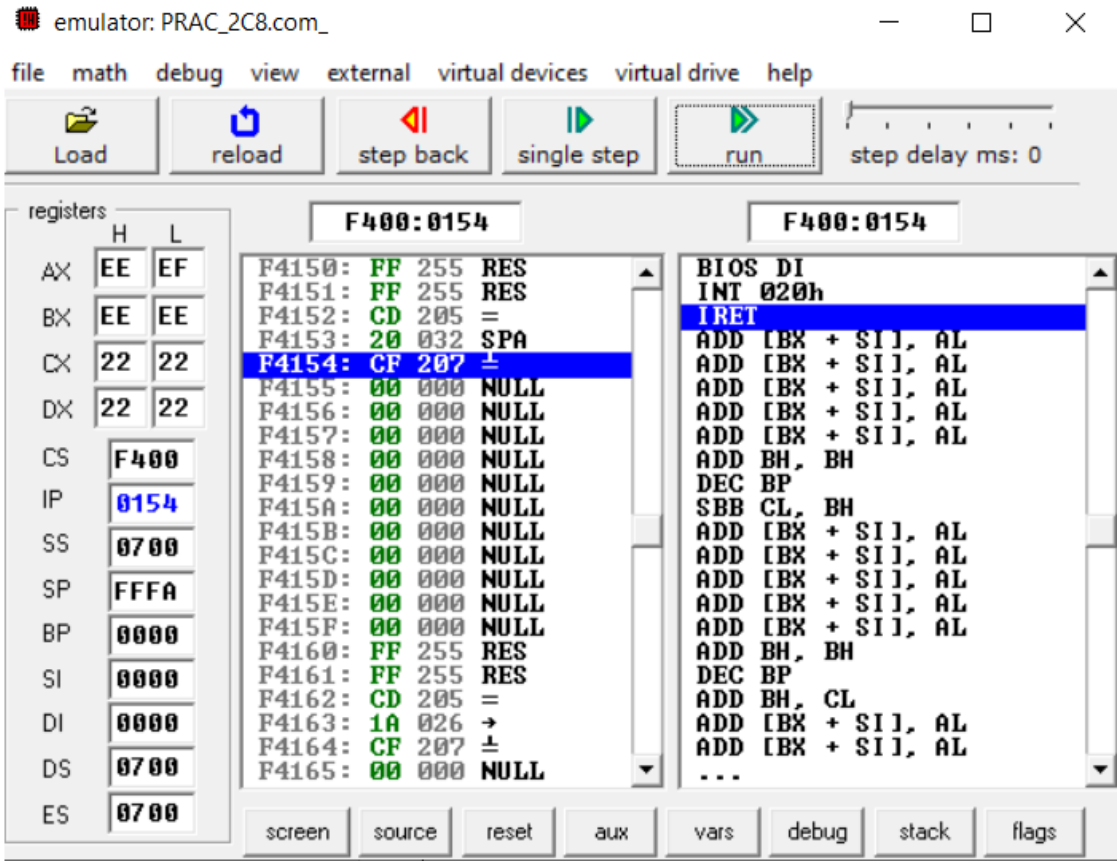
PRACTICAL: 2(C8)

AIM: Subtract Two 32-bit numbers stored in consecutive memory locations and store the result in memory locations starting from 7000H.

CODE:

```
org 100h

MOV [4000H], 1111H
MOV [4002H], 1111H
MOV [4004H], 2222H
MOV [4006H], 2222H

MOV AX, [4000H]
MOV BX, [4002H]
MOV CX, [4004H]
MOV DX, [4006H]

SBB AX, CX
SBB BX, DX

MOV [7000H], AX
MOV [7002H], BX

ret
```

OUTPUT:



CONCLUSION:

We learned about SBB command and its implementation for 32-Bit Numbers.

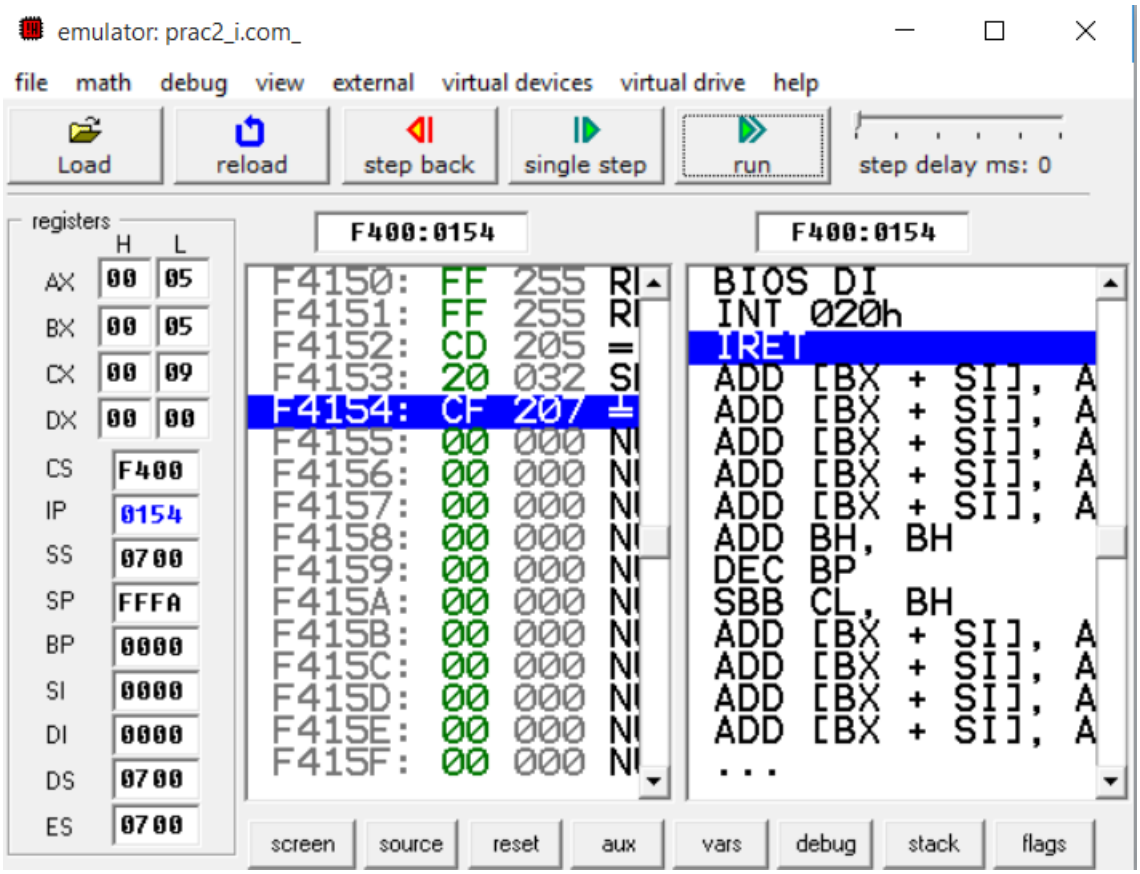
PRACTICAL: 2(C9)

AIM: Write an assembly language program to convert temperature in F to C. C=(F-32)*5/9.

CODE:

```
org 100h

MOV Al, 41
MOV Bl, 5
MOV CL, 9

SUB AL, 32
MUL BL
DIV CL

ret
```

OUTPUT:



CONCLUSION:

We learned to build Temperature Converter code using assembly language.

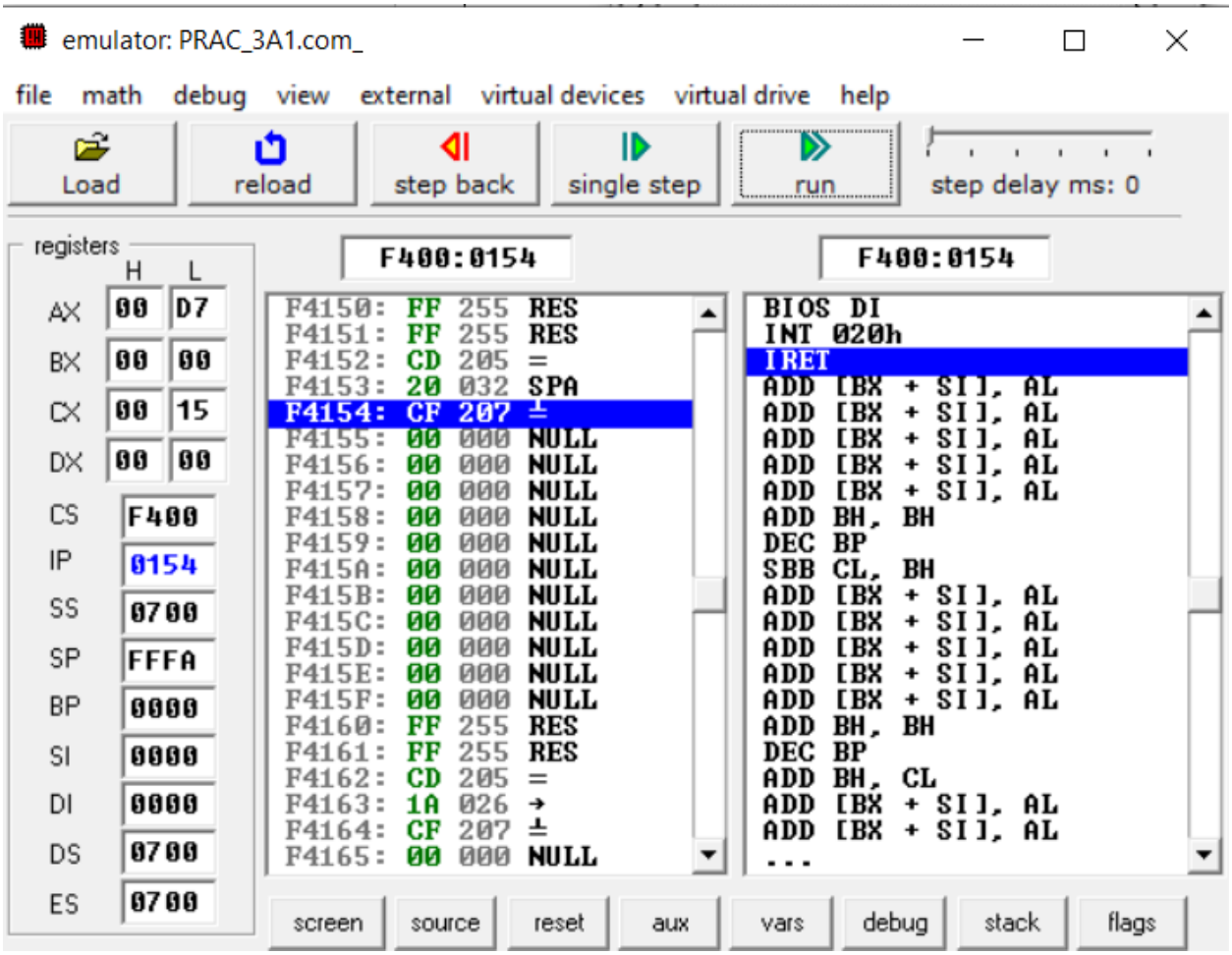
PRACTICAL: 3(A1)

AIM: Write a program to perform selective set operation on data stored at 4000H with the data stored at 4001H and store the result at 4002H. Verify the result and write bite wise operation of this program. (OR)

CODE:

```
org 100h

MOV [4000H], 10010101B
MOV [4001H], 01010010B

MOV AL, [4000H]
OR AL, [4001H]

MOV [4002H], AL

ret
```

OUTPUT:



CONCLUSION:

We learned about OR command and its implementation.

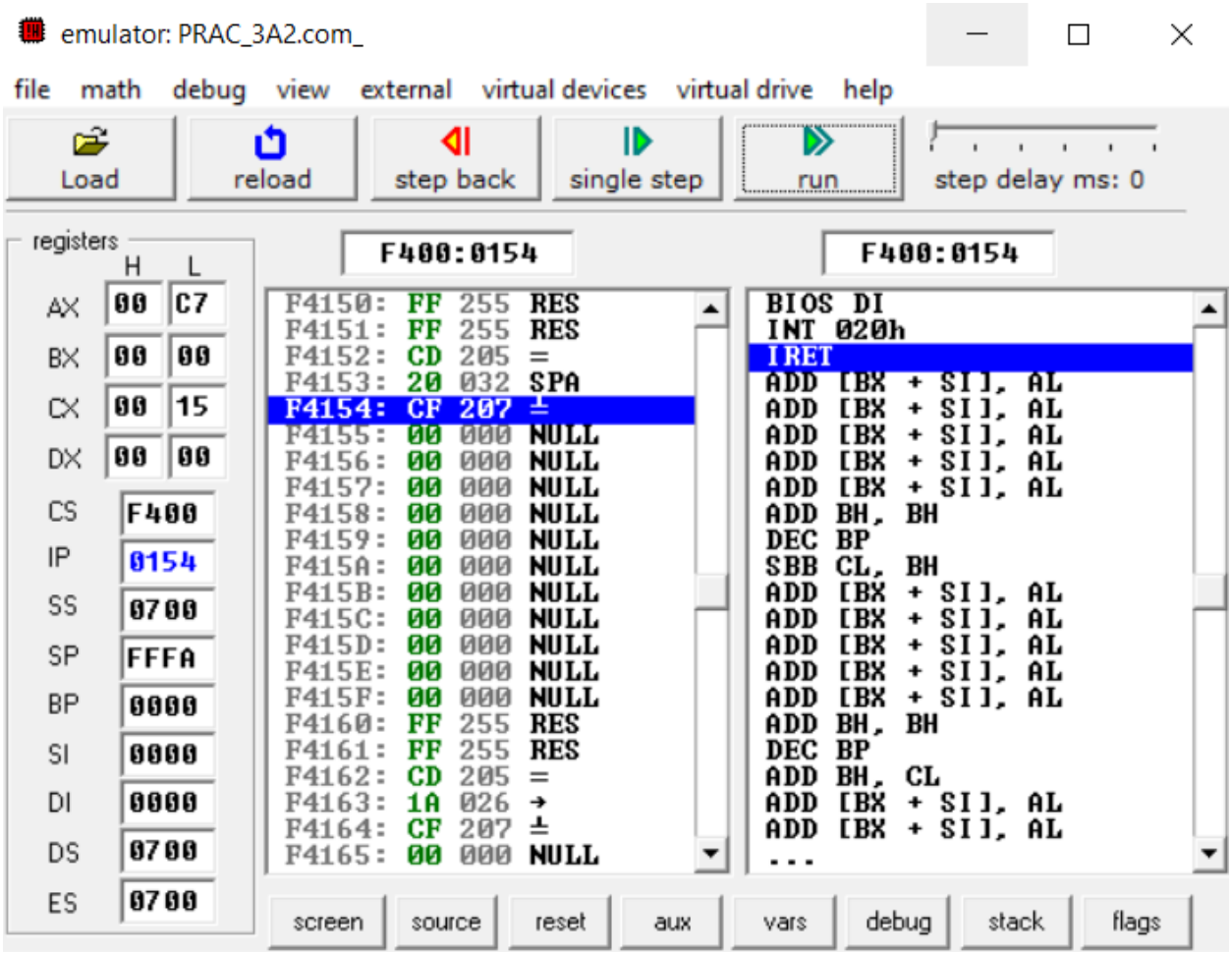
PRACTICAL: 3(A2)

AIM: Write a program to perform selective compliment operation on data stored at 4000H corresponding to the data stored at 4001H and store the result at 4002H. Verify the result and write bite wise operation of this program. (XOR)

CODE:

```
org 100h

MOV [4000H], 10010101B   ;149
MOV [4001H], 01010010B   ;82

MOV AL, [4000H]
XOR AL, [4001H]

MOV [4002H], AL

ret
```

OUTPUT:



CONCLUSION:

We learned about XOR command and its implementation.

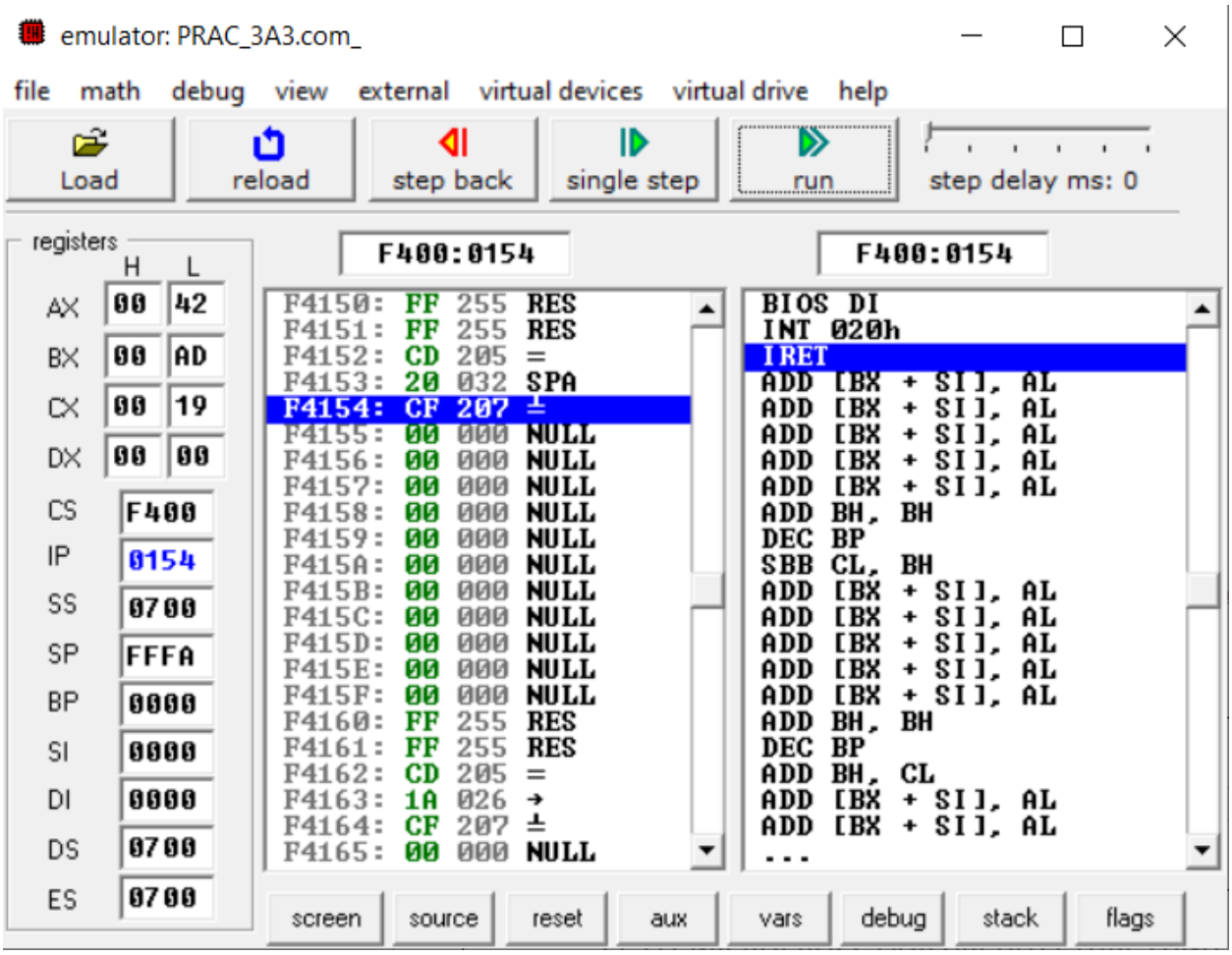
PRACTICAL: 3(A3)

AIM: Write a program to perform selective clear operation on data stored at 4000H corresponding to the data stored at 4001H and store the result at 4002H.Verify the result and write bite wise operation of this program. (A AND B')

CODE:

```
org 100h

MOV [4000H], 10010101B   ;149
MOV [4001H], 01010010B   ;82

MOV AL, [4000H]
MOV BL, [4001H]

NOT BL

ADD AL, BL
MOV [4002H], AL

ret
```

OUTPUT:



CONCLUSION:

We learned about NOT command and its implementation.

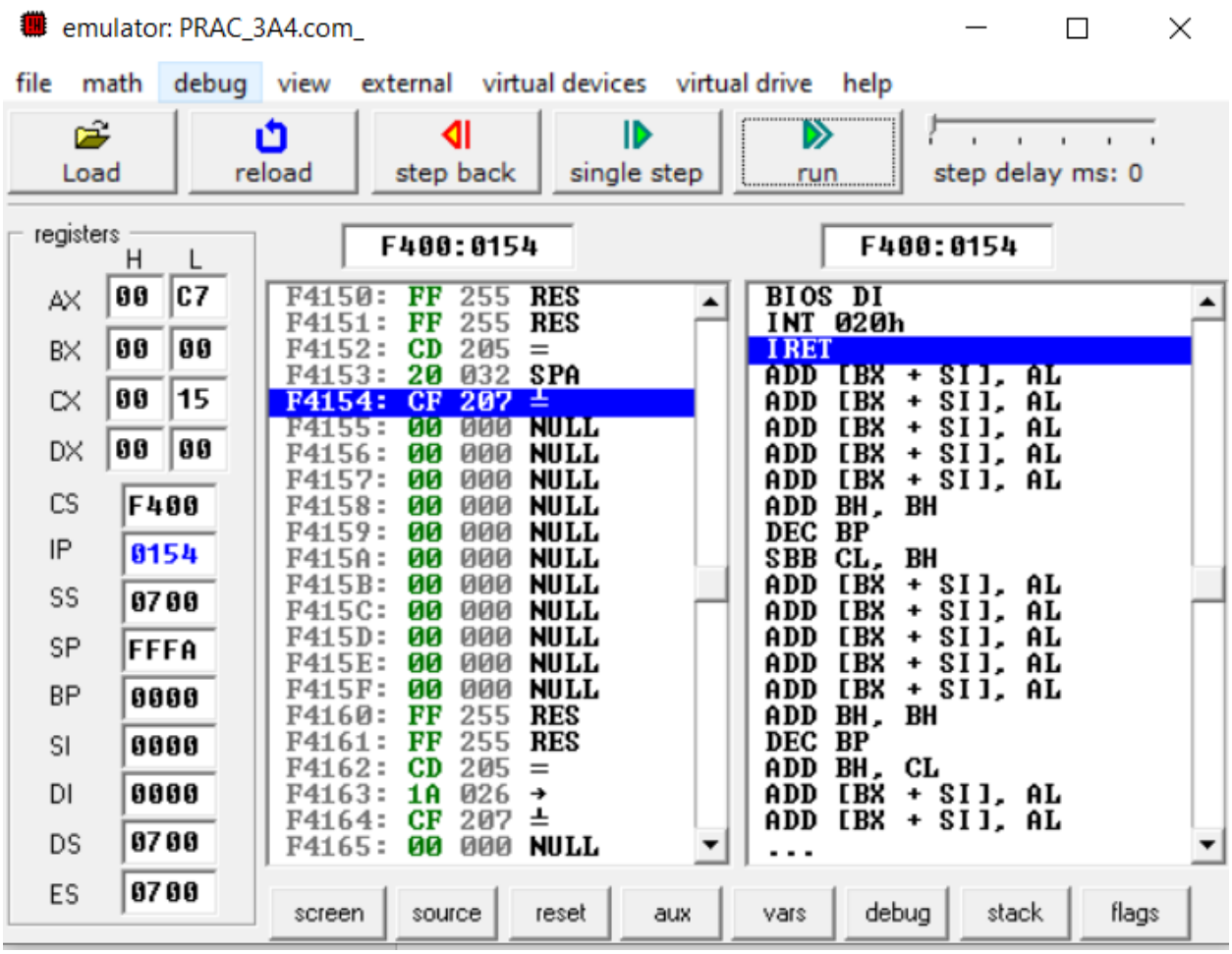
PRACTICAL: 3(A4)

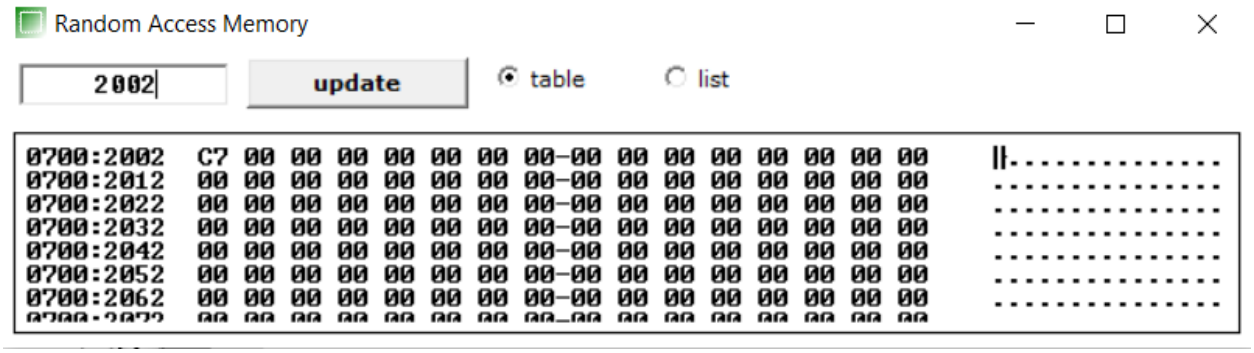
AIM: Write an assembly language program the data at memory locations 2000H & 2001H. (Use XOR)

CODE:

```
org 100h

MOV [2000H], 10010101B   ;149
MOV [2001H], 01010010B   ;82

MOV AL, [2000H]
XOR AL, [2001H]

MOV [2002H], AL

ret
```

OUTPUT:





CONCLUSION:

We learned about XOR command and its implementation.

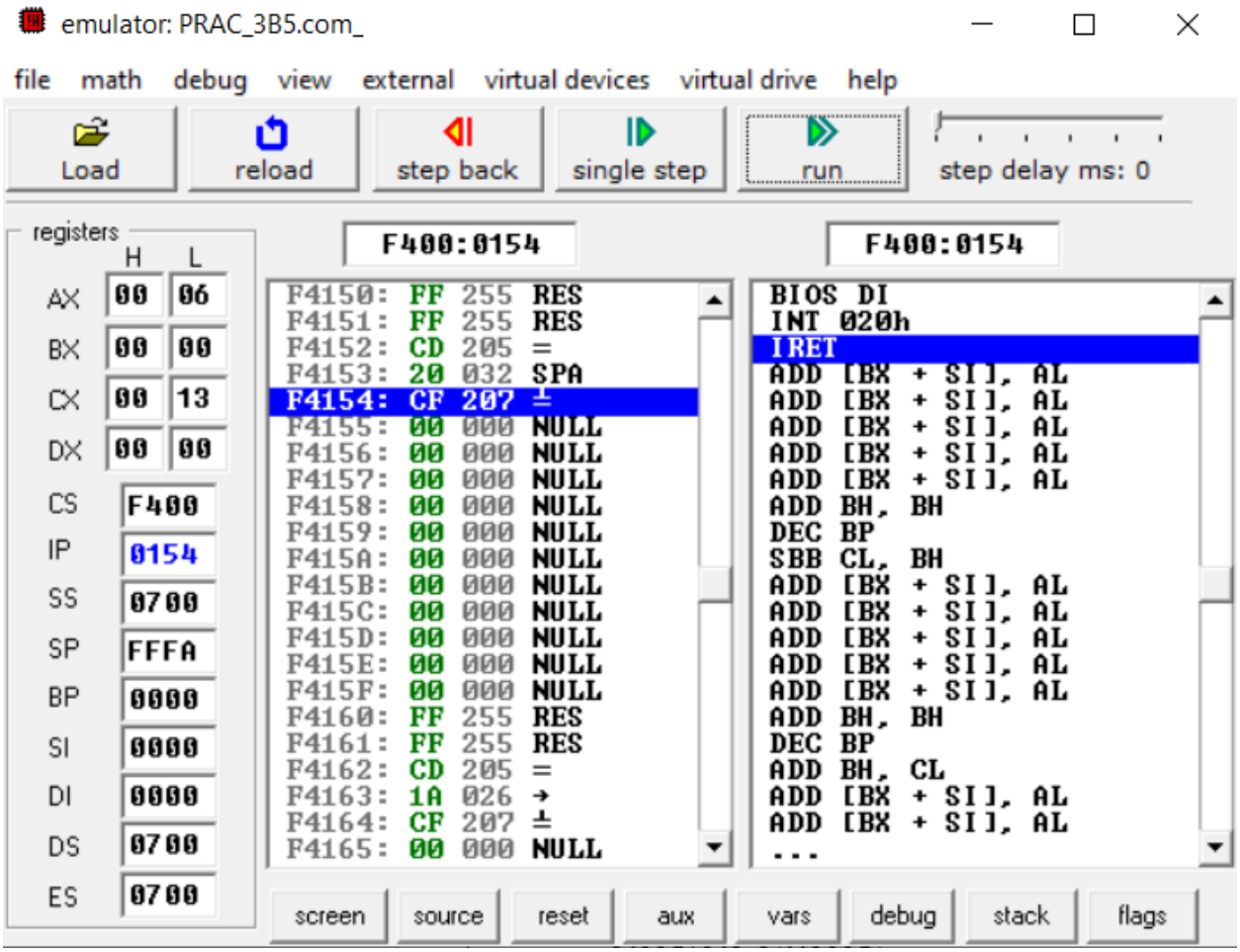
PRACTICAL: 3(B5)

AIM: Write a program to multiply & divide the number stored at 4000H by 2 and store the result at 4001H & 4002H .(Use Shift instructions).

CODE:

```
org 100h

MOV [4000H], 0000110B   ;149

MOV AX, [4000H]
SHL AX ,1
MOV [4001H], AX
SHR AX ,1
MOV [4002H], AX

ret
```

OUTPUT:



CONCLUSION:

We learned about SHL and SHR commands and its implementation.

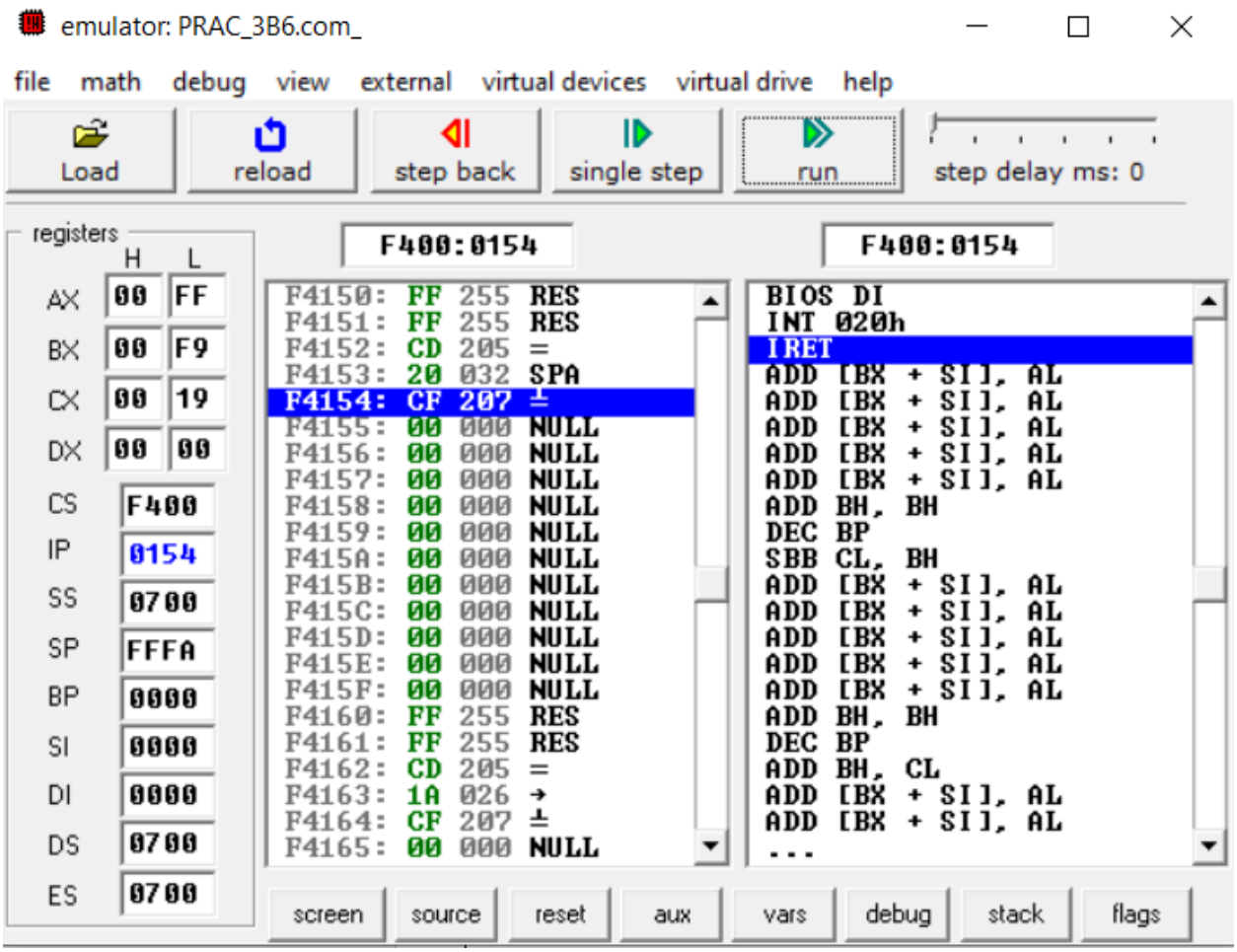
PRACTICAL: 3(B6)

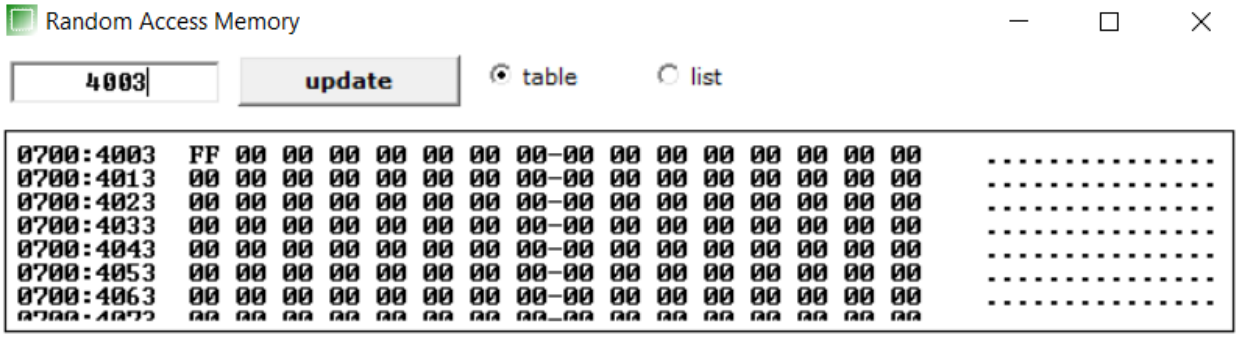
AIM: Write a Program to subtract the contents of memory location 4001H from the memory location 4002H and place the result in memory location 4003H without SUB instruction.

CODE:

```
org 100h

MOV [4001H], 0110B   ;149
MOV [4002H], 0110B   ;82

MOV AL, [4001H]
MOV BL, [4002H]

NOT BL

ADD AL, BL
MOV [4003H], AL

ret
```

OUTPUT:





CONCLUSION:

We learned about Subtraction Operation without SUB command. (A – B = A + B').

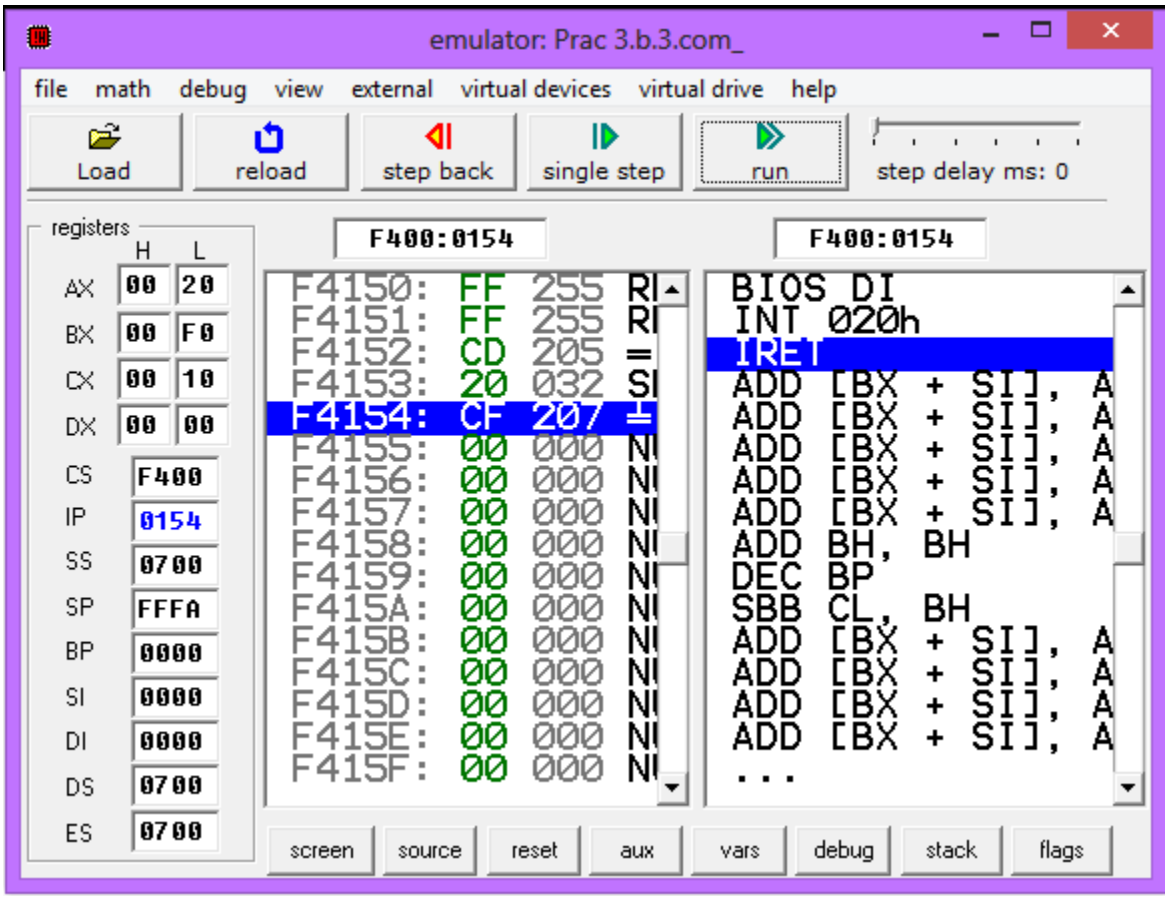
PRACTICAL: 3(B7)

AIM: Implement a program to mask the lower four bits of content of the memory Location.

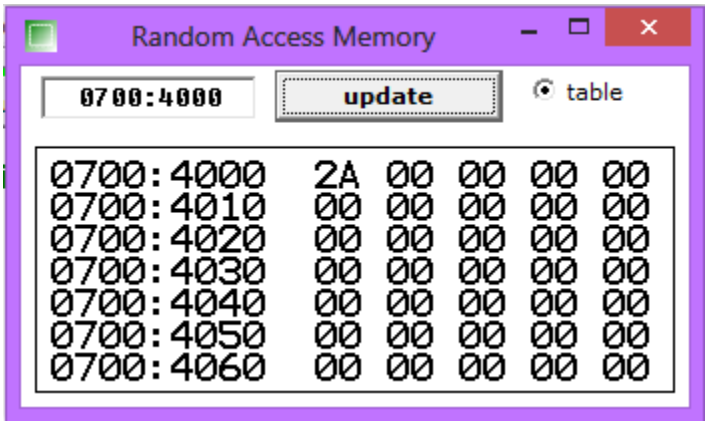
CODE:

```
org 100h

MOV [4000H], 00101010b
MOV AL, [4000H]
MOV BL, 11110000b
AND AL,BL
MOV [4000H], AL

ret
```

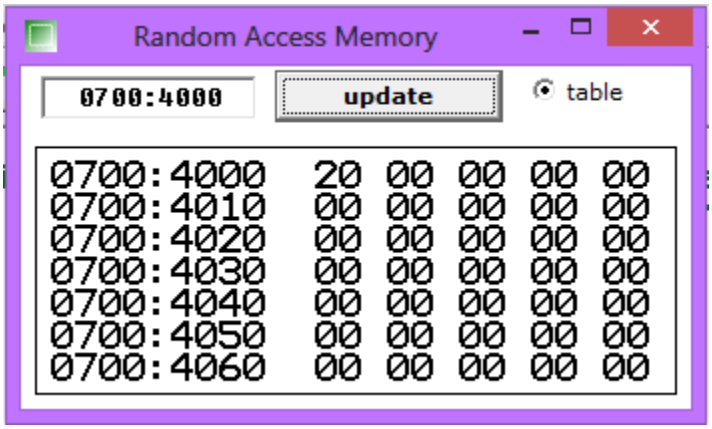
OUTPUT:



Before:



After:



CONCLUSION:

We learned that How to mask lower 4 bits of content of memory location by using AND command.

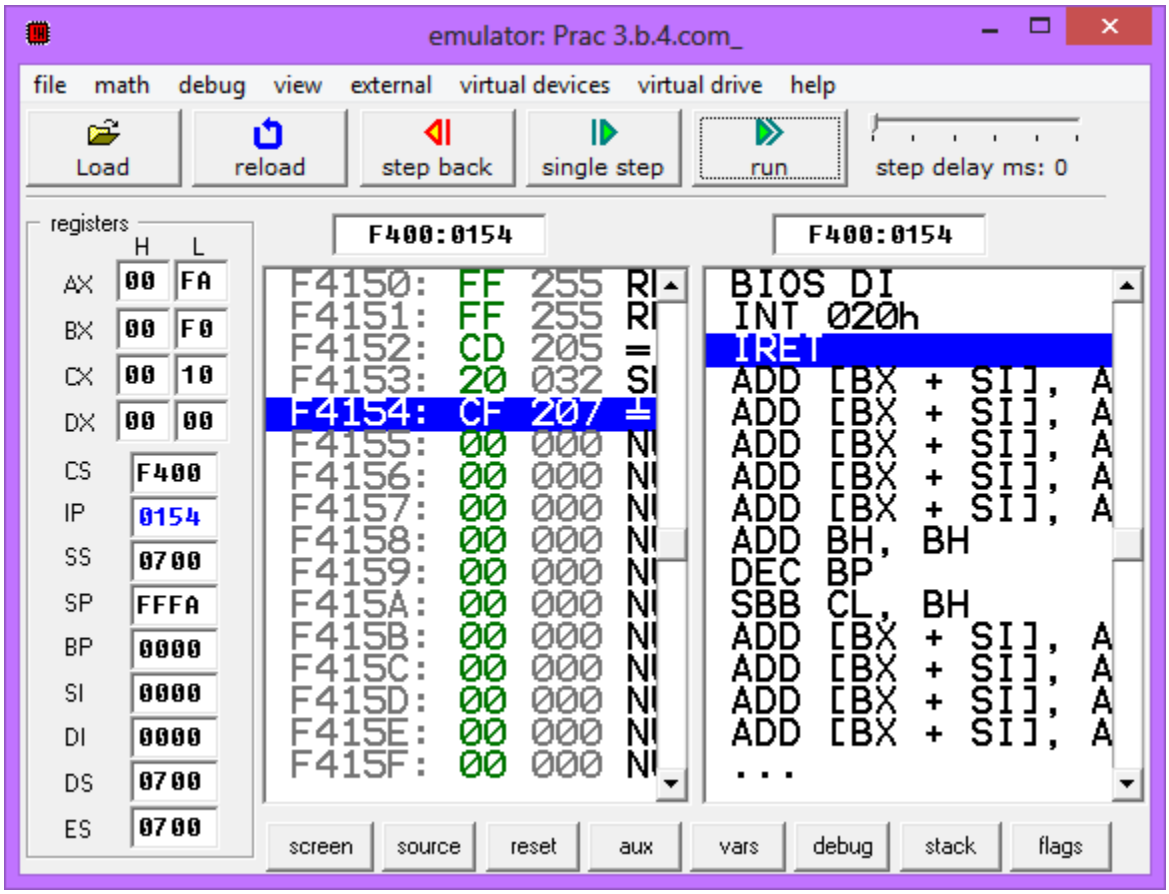
PRACTICAL: 3(B8)

AIM: Implement a program to set higher four bits of content of the memory location to 1.

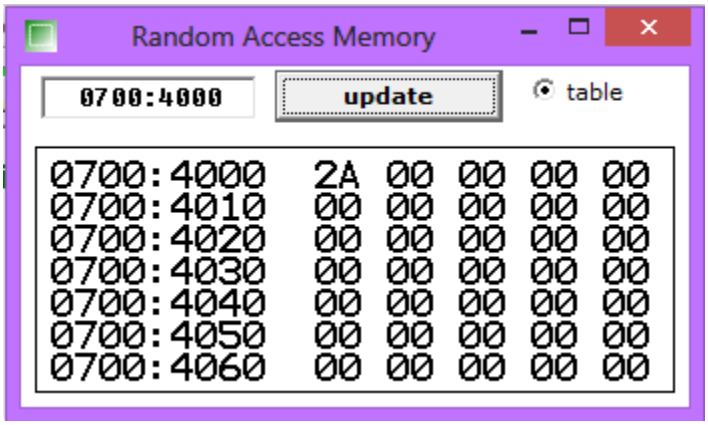
CODE:

```
org 100h

MOV [4000H], 00101010b
MOV AL, [4000h]
MOV BL, 11110000b
OR AL, BL
MOV [4000H], AL

ret
```

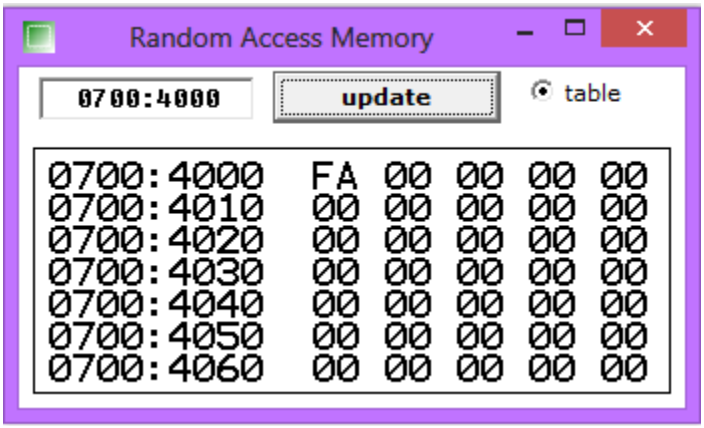
OUTPUT:



Before:



After:



CONCLUSION:

We learned that How to set 1 at higher 4 bits of content of memory location by using OR command.

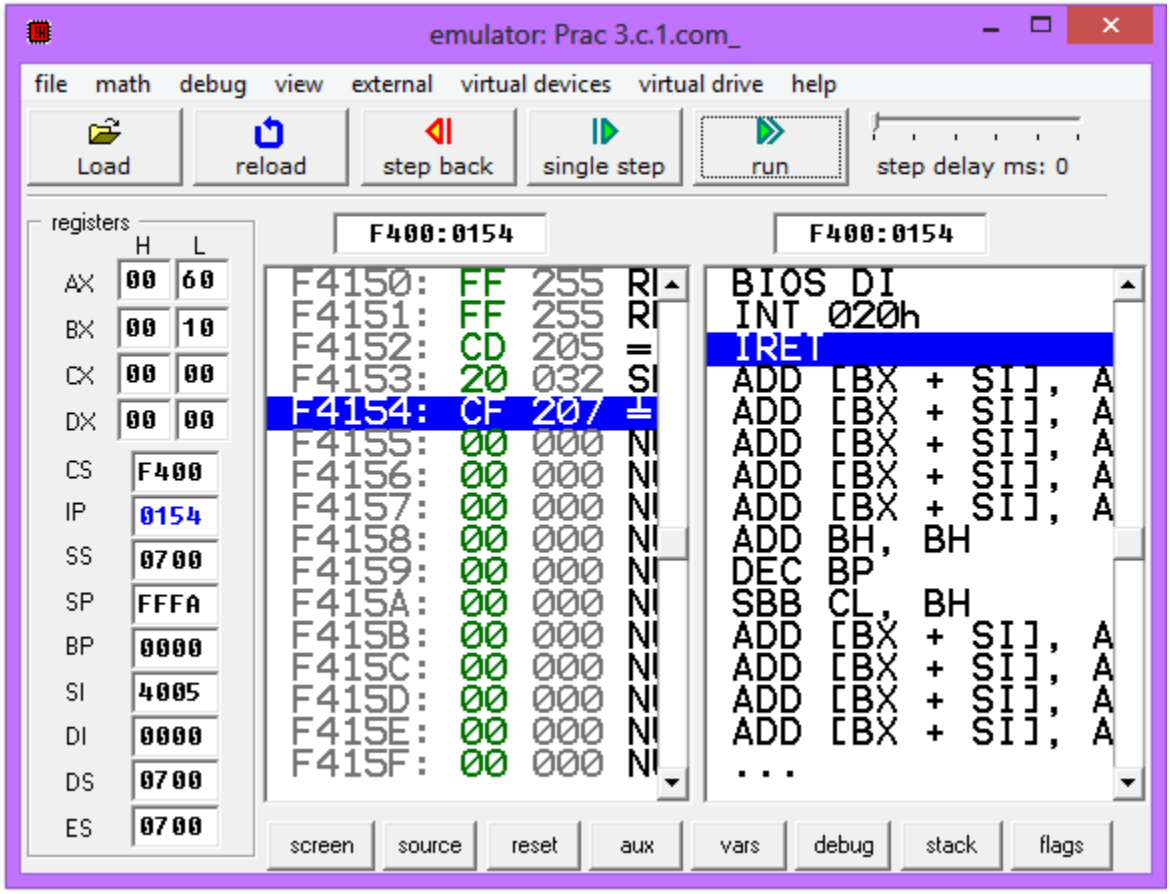
PRACTICAL: 3(C9)

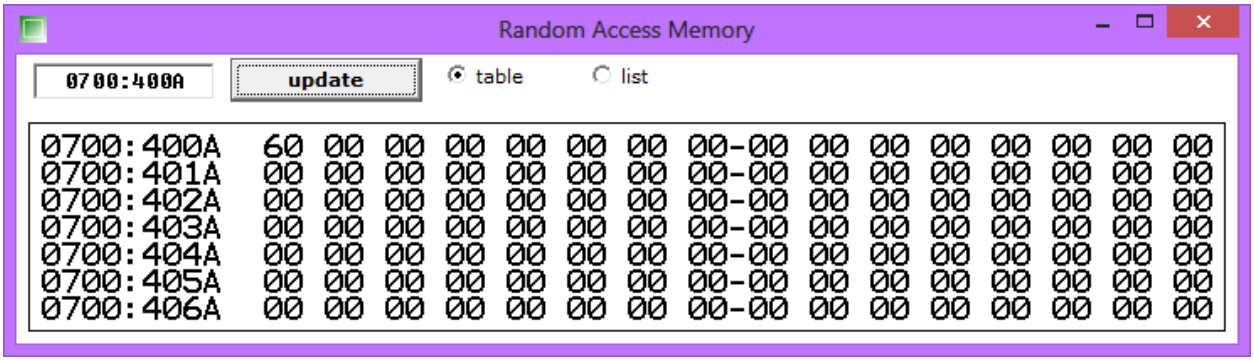
AIM: Calculate the sum of series of numbers (Data set-1) from the memory location listed below & store the result at 400AH location.

CODE:

```
org 100h

MOV [4000H],10H
MOV [4001H],10H
MOV [4002H],10H
MOV [4003H],10H
MOV [4004H],10H
MOV [4005H],10H

MOV SI,4000H
MOV AL,[SI]
MOV CL,5

L1: INC SI
   MOV BL,[SI]
   ADD AL,BL
   DEC CL
   JNZ L1

MOV [400AH], AL

ret
```

OUTPUT:





CONCLUSION:

We learned How to calculate the sum of series of numbers by using Loop and we also learned about INC, DEC and JNZ command and its implementation.

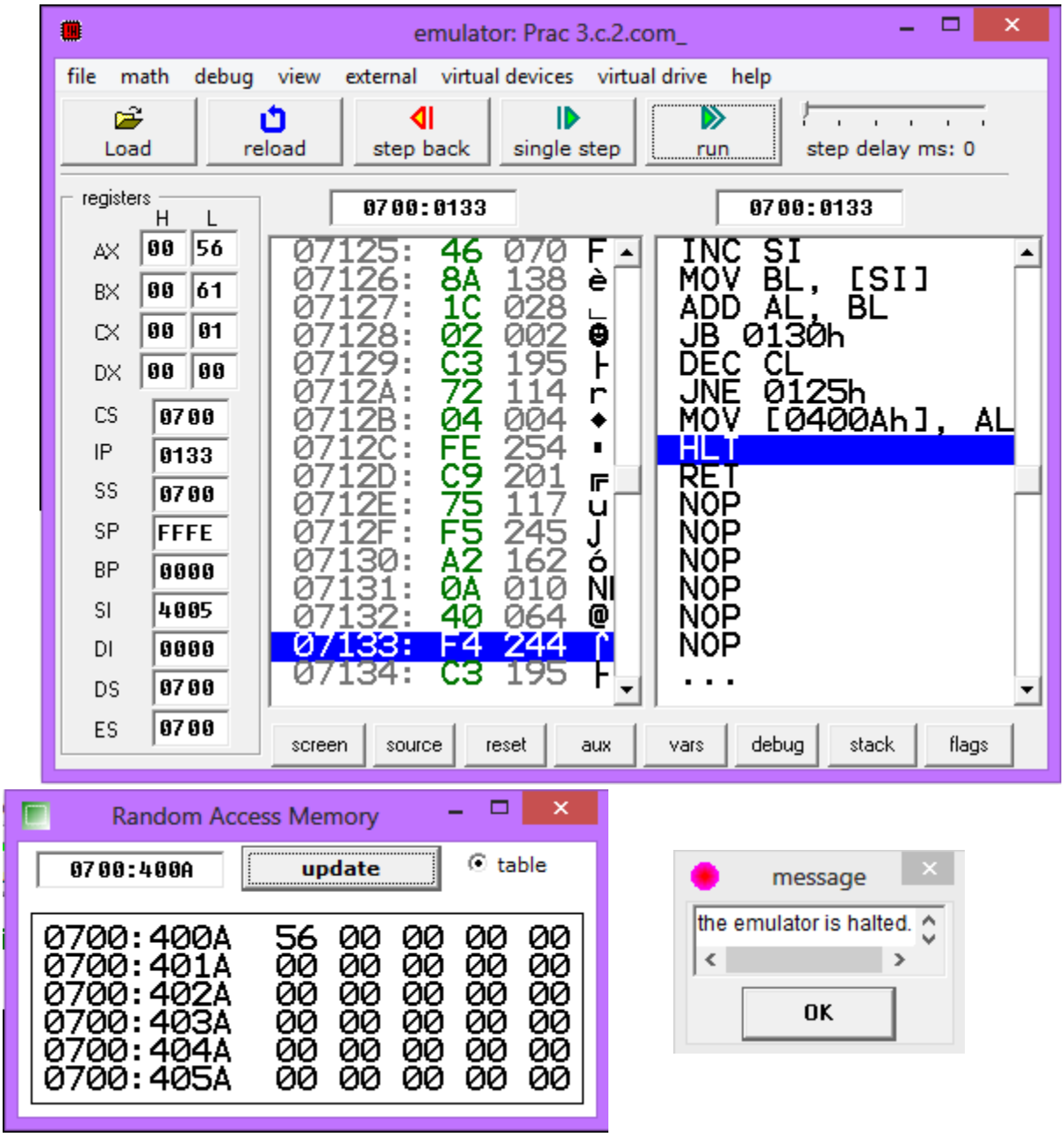
PRACTICAL: 3(C10)

AIM: Modify above the program such a way that it halts the execution if carry generated & stores the intermediate result at 400AH location. (Data set-2) (Note: Student need to implement FOR loop in this program: initialization, Compare, Decrement/Increment; also need to use JMP, JMx instructions.)

CODE:

```
org 100h

MOV [4000H],11H
MOV [4001H],21H
MOV [4002H],31H
MOV [4003H],41H
MOV [4004H],51H
MOV [4005H],61H

MOV SI,4000H
MOV AL,[SI]
MOV CL,5

L1: INC SI
   MOV BL,[SI]
   ADD AL,BL
   JC L2
   DEC CL
   JNZ L1

L2: MOV [400AH],Al
   HLT

ret
```

OUTPUT:



CONCLUSION:

We learned How to HALT the program when carry generated by using HALT and JC command.

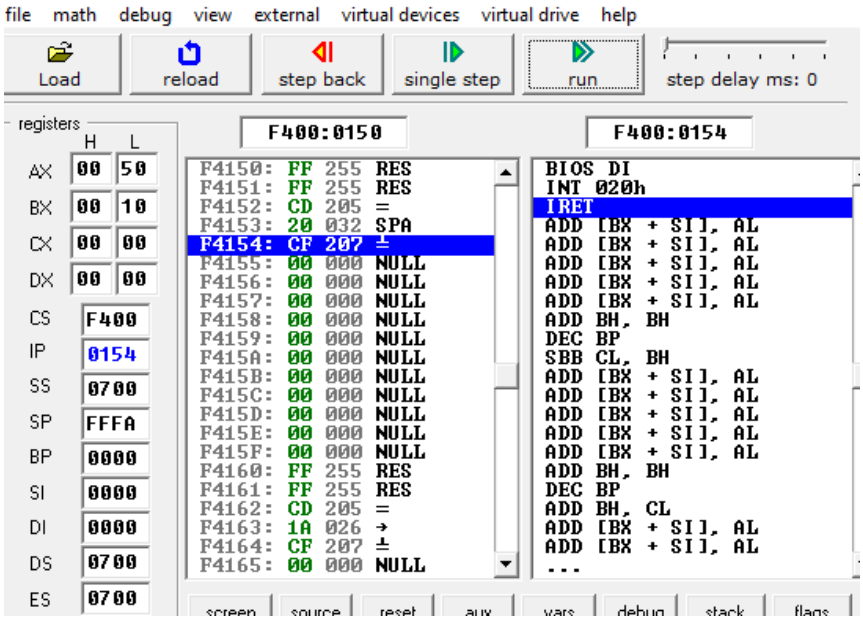
PRACTICAL: 3(C11)

AIM: Multiply two 8-bit numbers stored in memory locations 4001H and 4006H by repetitive addition and store the result at 400AH location.(Use Data Set -3) (Note: Student need to implement FOR loop in this program: initialization, Compare, Decrement/Increment; also need to use JMP, JMx instructions.)

CODE:

```
org 100h
MOV [4001h],10h
mov bl,[4001h]
mov [4000h],0h

MOV cl,0005h
MOV al,[4000h]
l1:
add al,bl
loop l1

ret
```

OUTPUT:



CONCLUSION:

I learned how to use loop concept.

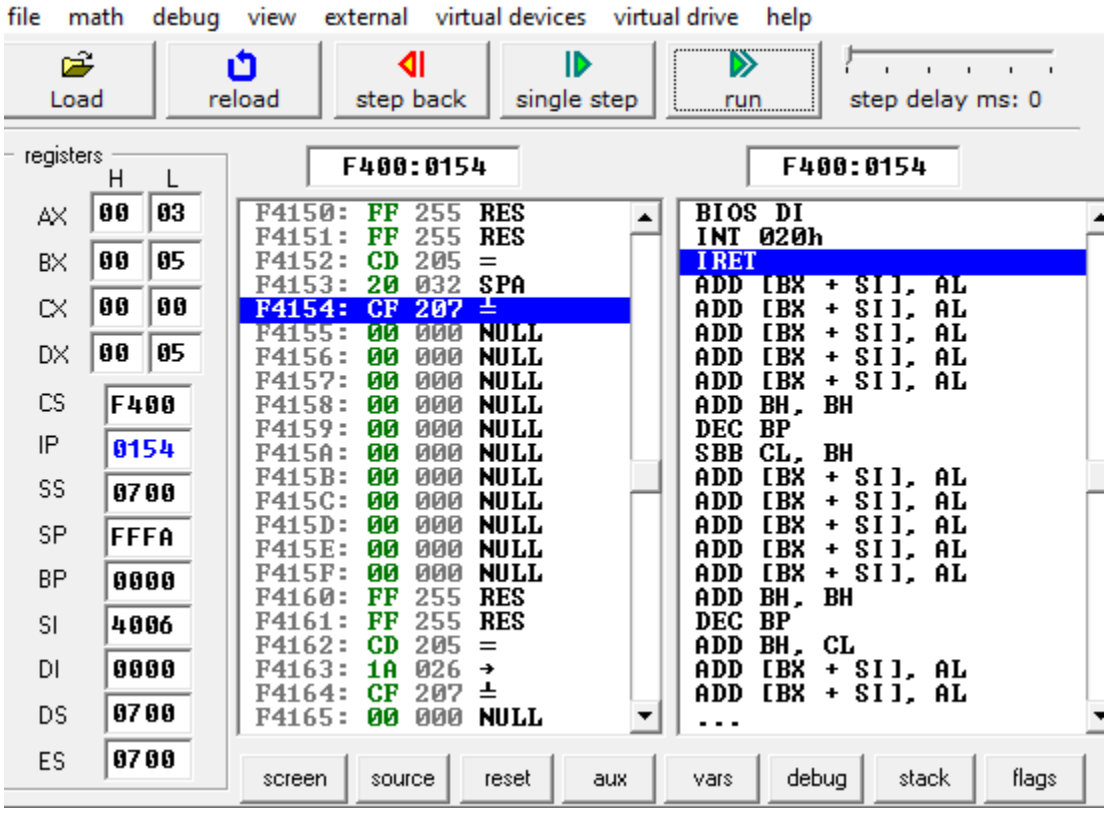
PRACTICAL: 3(C12)

AIM: Program to find average of n numbersorg 100h

CODE:

```
mov [4001h],1h
mov [4002h],2h
mov [4003h],3h
mov [4004h],4h
mov [4005h],5h

mov si,4001h
mov al,0h
mov cl,5h
mov dl,5h

l1:
mov bl,[si]
add al,bl
inc si
loop l1

div dl

mov [6000h],al

ret
```

OUTPUT:



CONCLUSION:

I learned how to use loop concept using Source index.

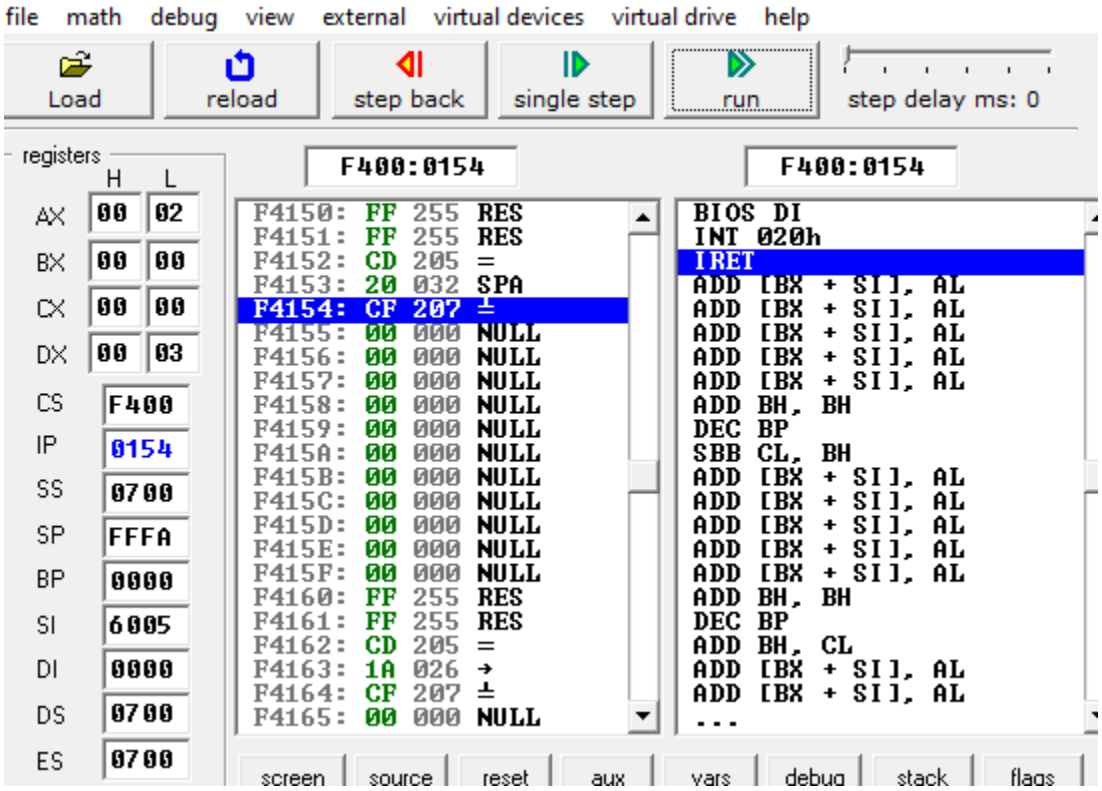
PRACTICAL: 3(C13)

AIM: Write an assembly language program to find the no. of odd numbers and even numbers, given an array of n numbers.

CODE:

```
org 100h

mov [6000h], 01h
mov [6001h], 02h
mov [6002h], 03h
mov [6003h], 04h
mov [6004h], 05h
mov SI, 6000h
mov cl, 05h
mov dl, 00h
l1:
mov al, [SI]
INC SI
shr al, 1
JNC even
INC dl
even:
loop l1
mov [400Ah], dl
```

OUTPUT:



CONCLUSION:

I learned how to check whether given number is odd or even.

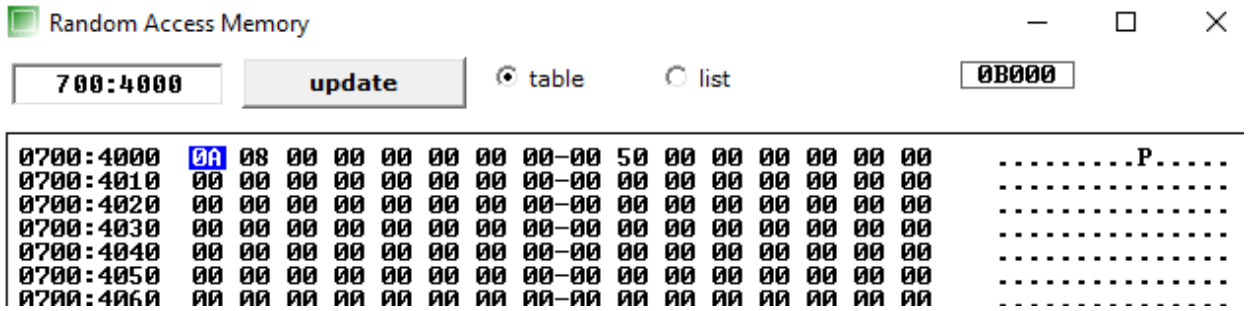
PRACTICAL: 4(A1)

AIM: : Divide 8-bit number stored in memory locations 4009H by data stored at memory location 4001H & store result of division at memory location 400AH. (Use Data Set -4)

CODE:

```
org 100h
MOV [4009h],50h
mov [4001h],8h
mov al,[4009h]
mov bl,[4001h]

div bl;
mov [4000h],al;

ret
```

OUTPUT:



CONCLUSION:

In this I have learned that how to divide number of 8-bit.

PRACTICAL: 4(A2)

AIM: : Divide 8-bit number stored in memory locations 4009H by data stored at memory location 4001H & store result of module operation at memory location 400AH. .(Use Data Set - 2,4)

CODE:

```
org 100h
MOV [4009h],50h
mov [4001h],8h
mov al,[4009h]
mov bl,[4001h]

div bl;
mov [4000h],ah;

ret
```

OUTPUT:

CONCLUSION:

In this I have learned that how to find module of 8-bit.

PRACTICAL: 4(A4)

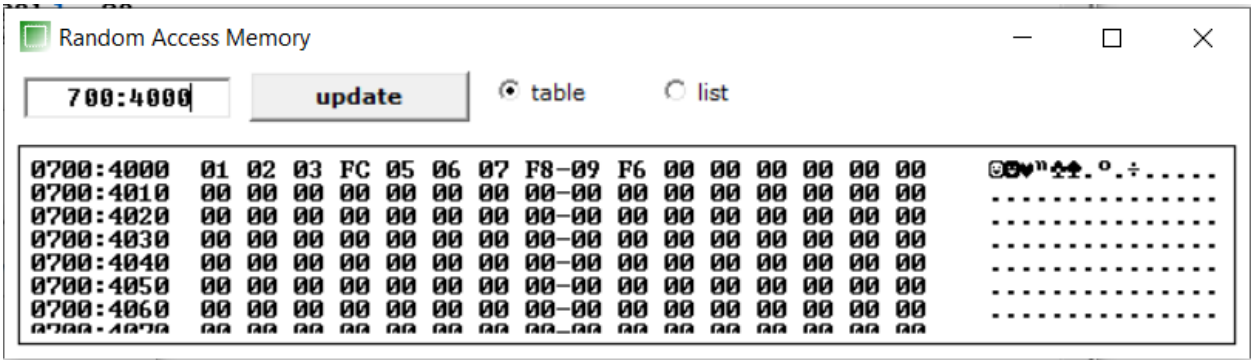
AIM: Write an assembly language program to count the numbers in an array (negative & positive).

PROGRAM:

```
org 100h
mov [4000h], 01
mov [4001h], 02
mov [4002h], 03
mov [4003h], -04
mov [4004h], 05
mov [4005h], 06
mov [4006h], 07
mov [4007h], -08
mov [4008h], 09
mov [4009h], -10

mov si, 4000h
sft:
mov al, [si]
shl al, 1
jnc positive
inc bl
jmp return
positive:
inc dl
return:
```

```
inc si

cmp si, 400ah

jne sft


ret
```

OUTPUT:



CONCLUSION:

In this I have learned that how to count number in array.

PRACTICAL: 4(B5)

AIM: Write an assembly language program to multiply two 16-bit numbers in memory and store the result in memory.

PROGRAM:

```
org 100h
mov [4000h], 3000h
mov [4002h], 2000h
mov ax, [4000h]
mov bx, [4002h]
mul bx
mov [4004h], dx
mov [4006h], ax
ret
```

OUTPUT:

CONCLUSION:

In this I have learned that how to multiply two 16-bit number.

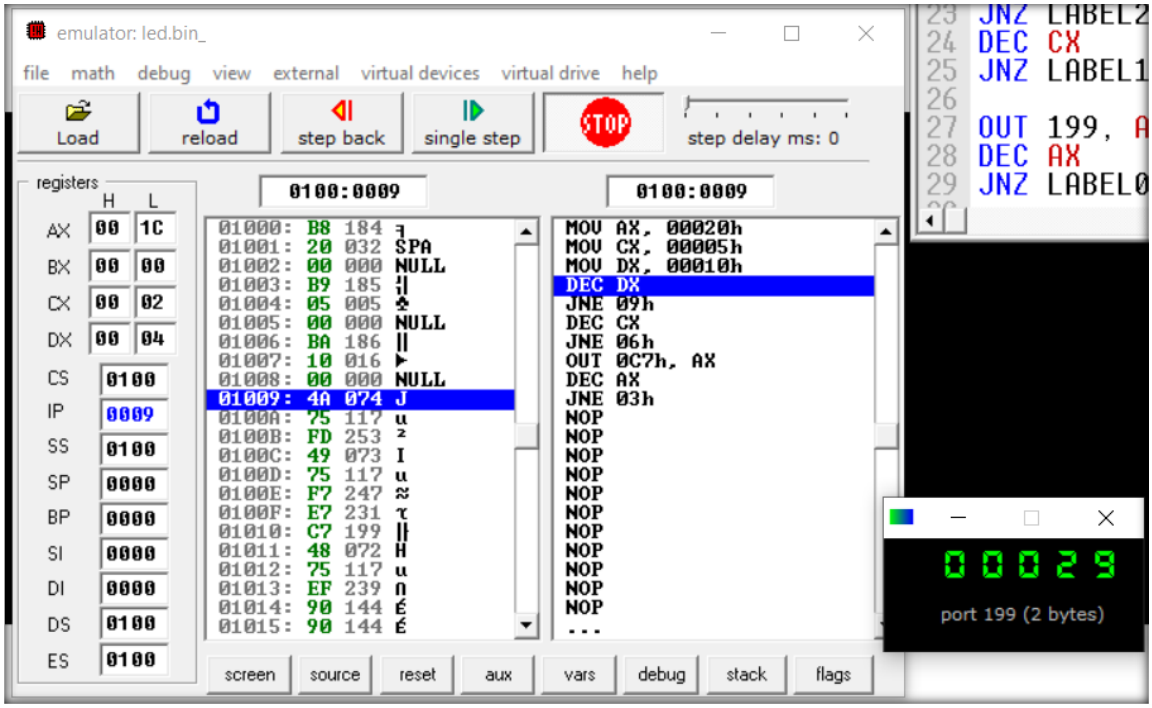
PRACTICAL: 4(B6)

AIM: Write a program with nested loop which will display the decimal down counter .on Port 1 with a one second delay between each count.

PROGRAM:

```
#start=led_display.exe#

name "led"

MOV AX, 20H

;Values are kept small for smaller delay

LABEL0:

MOV CX, 0005H

LABEL1:

MOV DX, 0010H

LABEL2:

DEC DX

JNZ LABEL2

DEC CX

JNZ LABEL1


OUT 199, AX

DEC AX

JNZ LABEL0
```

OUTPUT:



CONCLUSION:

In this practical, we interfaces LED with 8086 and made a decimal down counter.

PRACTICAL: 4(B7)

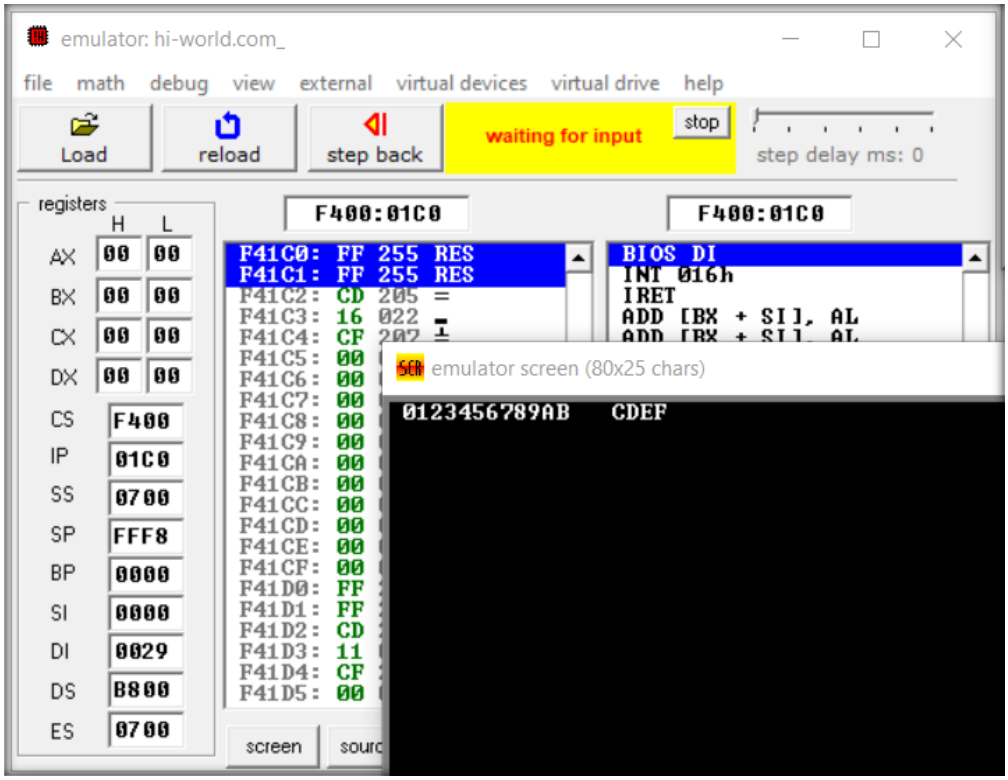
AIM: Write an assembly language program to Display Digits 0 1 2 3 4 5 6 7 8 9 A B C D E F on port 01H with 500ms of delay.

PROGRAM:

```
org 100h

; set video mode

mov ax, 3     ; text mode 80x25, 16 colors, 8 pages (ah=0, al=3)

int 10h       ; do it!

; cancel blinking and enable all 16 colors:

mov ax, 1003h

mov bx, 0

int 10h

; set segment register:

mov     ax, 0b800h

mov     ds, ax

; print "hello world"

; first byte is ascii code, second byte is color code.

mov [02h], '0'

mov [04h], '1'

mov [06h], '2'

mov [08h], '3'

mov [0ah], '4'

mov [0ch], '5'

mov [0eh], '6'

mov [10h], '7'

mov [12h], '8'

mov [14h], '9'
```

```
mov [16h], 'A'
mov [18h], 'B'
mov [20h], 'C'
mov [22h], 'D'
mov [24h], 'E'
mov [26h], 'F'
; color all characters:
mov cx, 19  ; number of characters.
mov di, 03h ; start from byte after 'h'
c:  mov [di], 00001111b   ; light red(1100) on yellow(1110)
    add di, 2 ; skip over next ascii code in vga memory.
    mov bx, 0020h
    back:
    dec bx
    jnz back
    loop c
; wait for any key press:
mov ah, 0
int 16h
ret
```

OUTPUT:



CONCLUSION:

In this practical, we implemented loop to generate delay and printed characters on the screen.

PRACTICAL: 4(C8)

AIM: Design an 8086 microprocessor based system with input device getting input from memory address starting from 2000 h to 2009 h. Three LEDs (common cathode): LED-1(Green) at D0 bit, LED-2 (Yellow) at D3 bit and LED-3 (Red) at D6 bit of the output device connected at I/O mapped address 01h. Write an assembly program to take data from input device,

Glow LED-1 ; if data <=50H

LED-2 ; if 50H >data<=A0H

LED-3; if data>A0H.

Take data from input device at every 10 ms time.

PROGRAM:

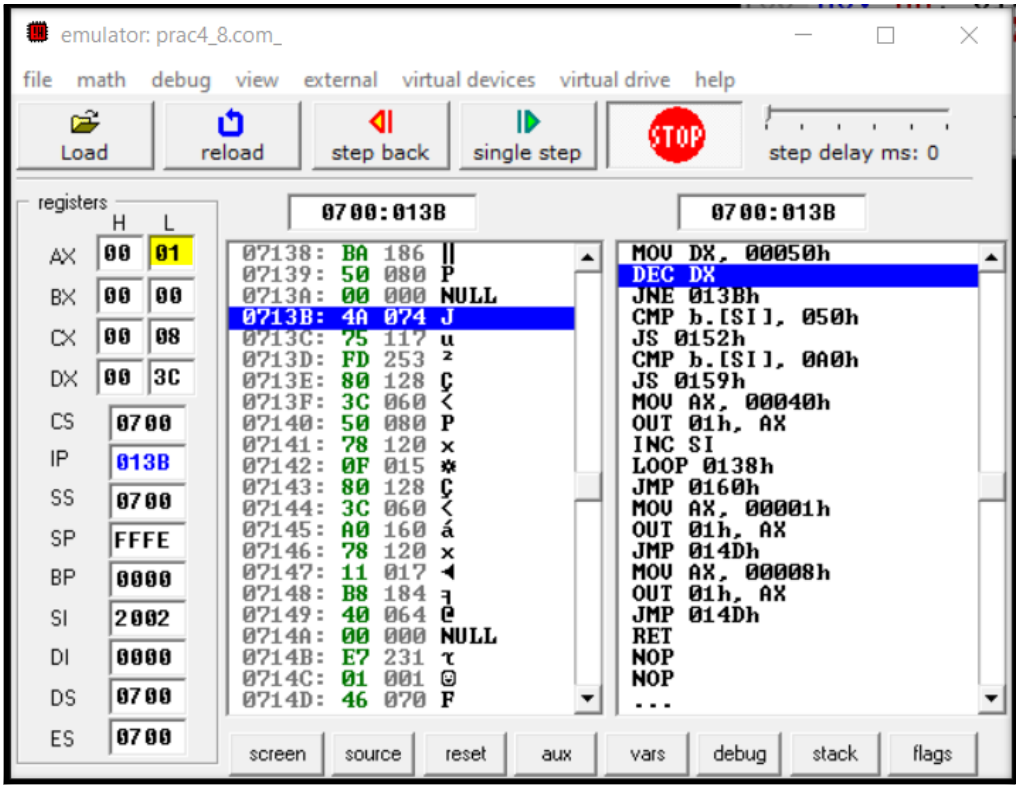
```
org 100h

MOV CX, 0AH

MOV [2000H], 16H

MOV [2001H], 48H

MOV [2002H], 0B0H

MOV [2003H], 27H

MOV [2004H], 64H

MOV [2005H], 25H

MOV [2006H], 0C2H

MOV [2007H], 63H

MOV [2008H], 0ADH

MOV [2009H], 9H

MOV SI, 2000H
```

```
LABEL1:
MOV DX, 0050H
LABEL2:
DEC DX
JNZ LABEL2
CMP [SI], 50H
JS CASE1
CMP [SI], 0A0H
JS CASE2
MOV AX, 01000000B
OUT 01H, AX
BACK:
INC SI
LOOP LABEL1
JMP FINISH

CASE1:
MOV AX, 00000001B
OUT 01H, AX
JMP BACK

CASE2:
MOV AX, 00001000B
OUT 01H, AX
JMP BACK
```

FINISH:

ret

OUTPUT:



CONCLUSION:

In this practical, we generated delay and output different values on different input conditions.

PRACTICAL: 5(A1)

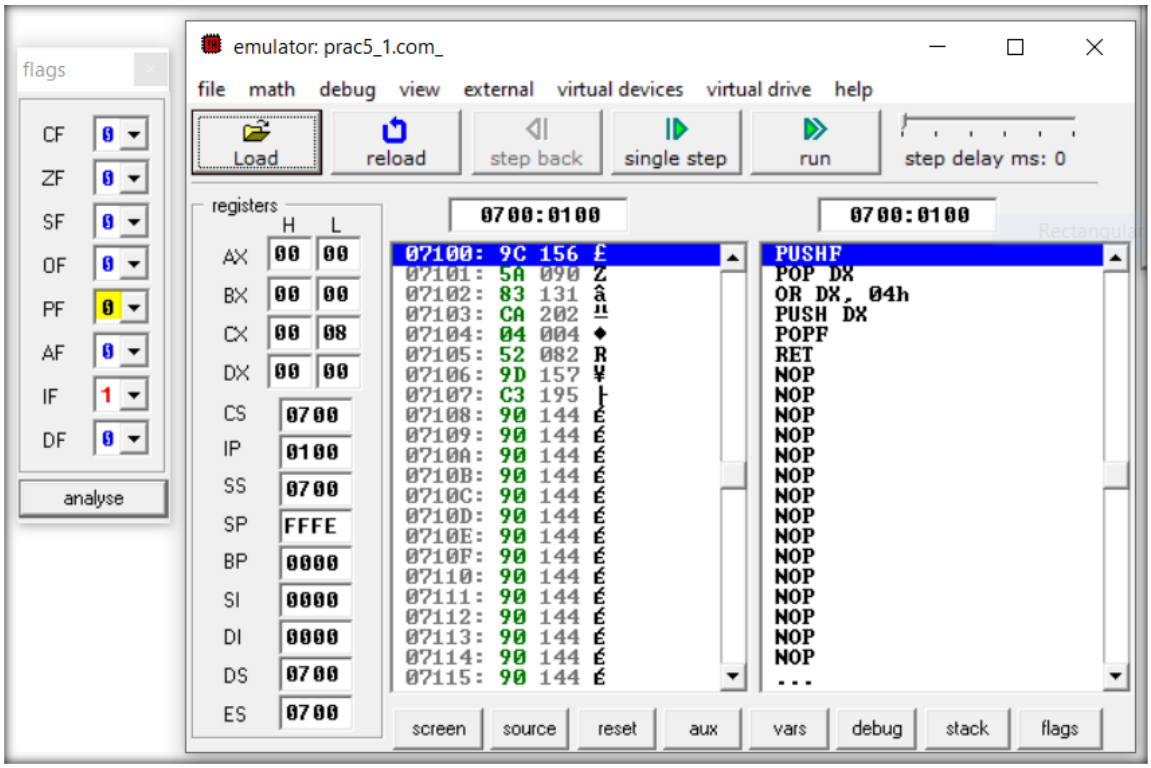
AIM: Write a program which sets the parity bit.

PROGRAM:

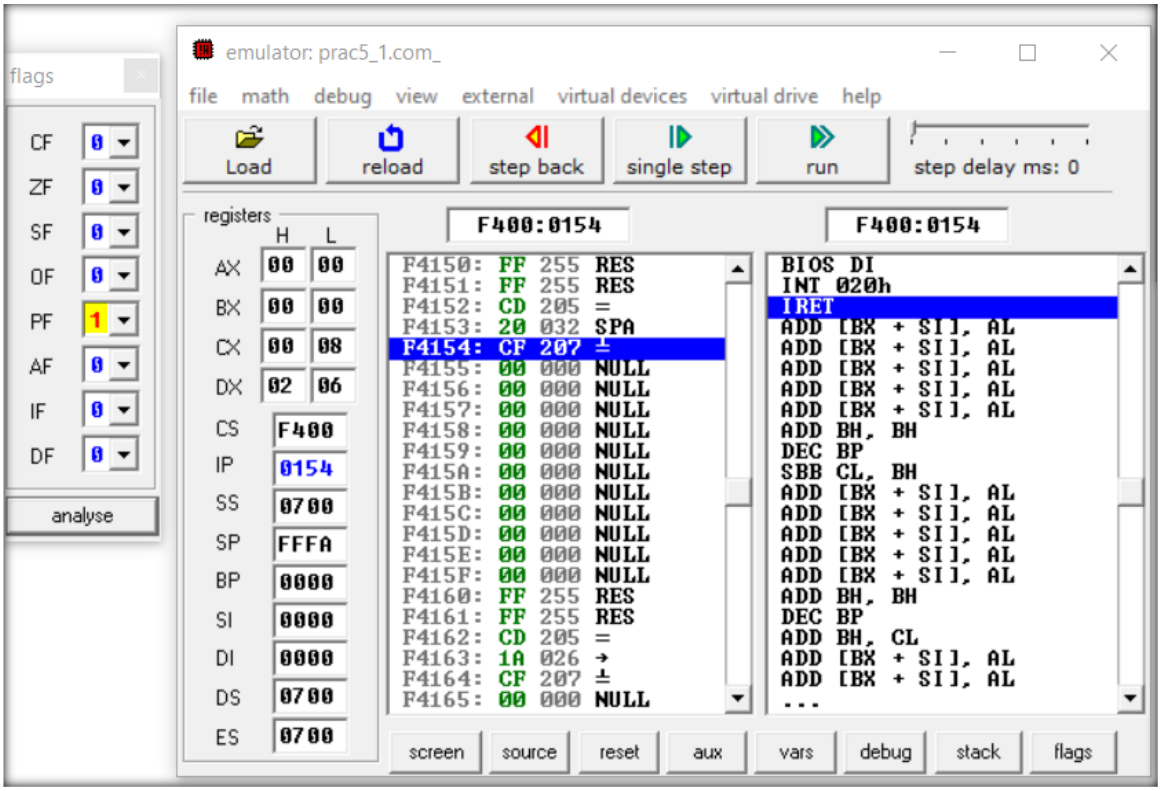
```
org 100h

PUSHF

POP DX

OR DX, 100B

PUSH DX

POPF

ret
```

OUTPUT:

Before:



After:



CONCLUSION:

In this practical, we learned about flag register and modified the parity bit.

PRACTICAL: 5(A2)

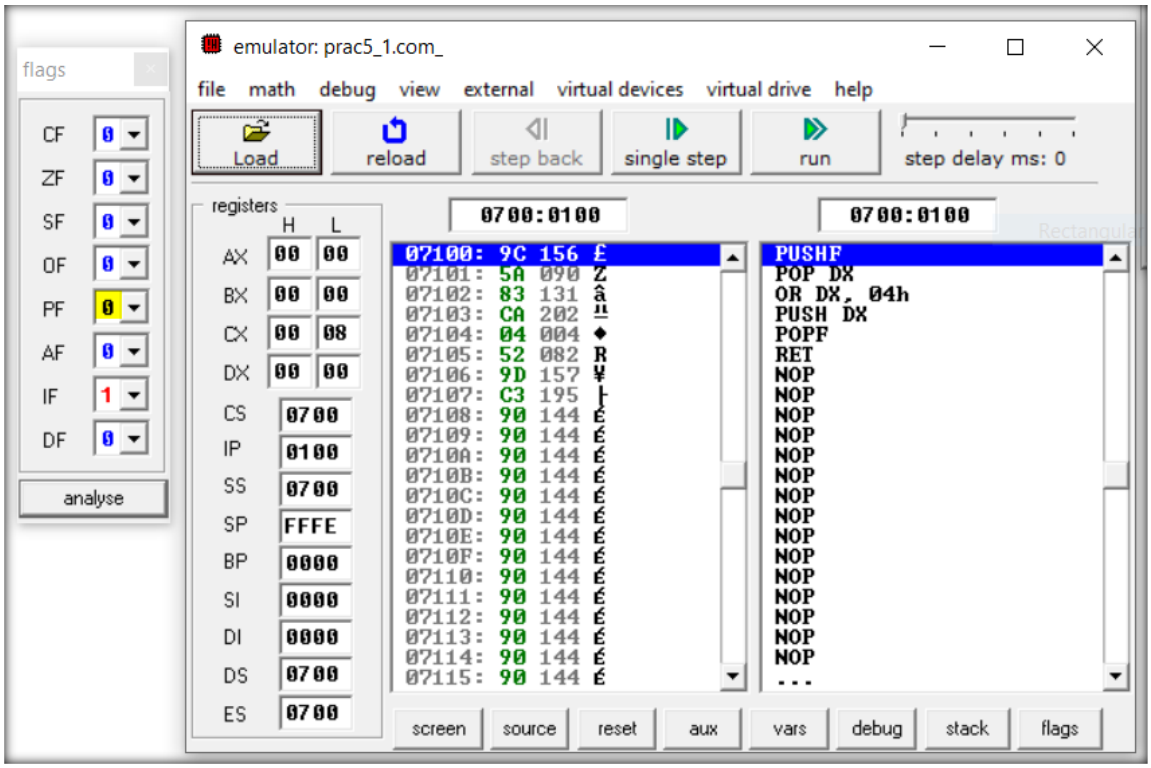
AIM: Write a program which transfers content of Flags to Register.

PROGRAM:

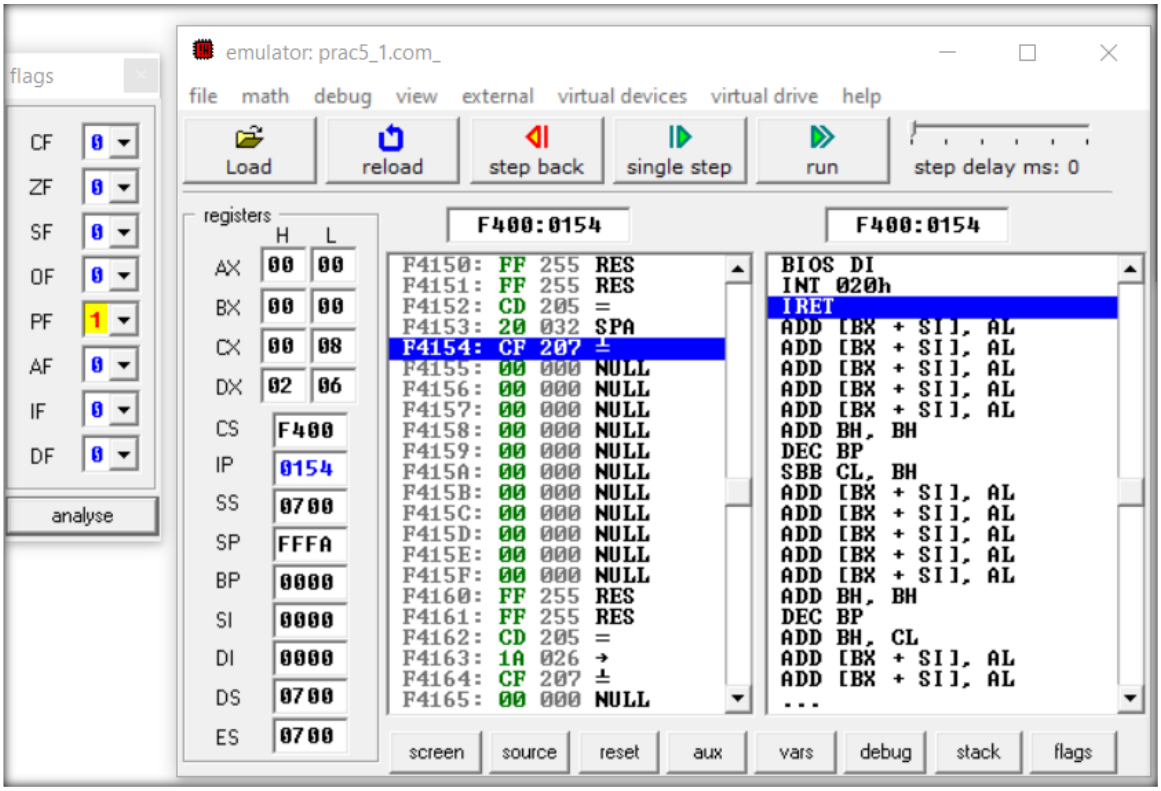
```
org 100h

PUSHF

POP DX

OR DX, 100B

PUSH DX

POPF

ret
```

OUTPUT:

Before:



After:



CONCLUSION:

In this practical, we learned how to change flag bits using stack instructions.

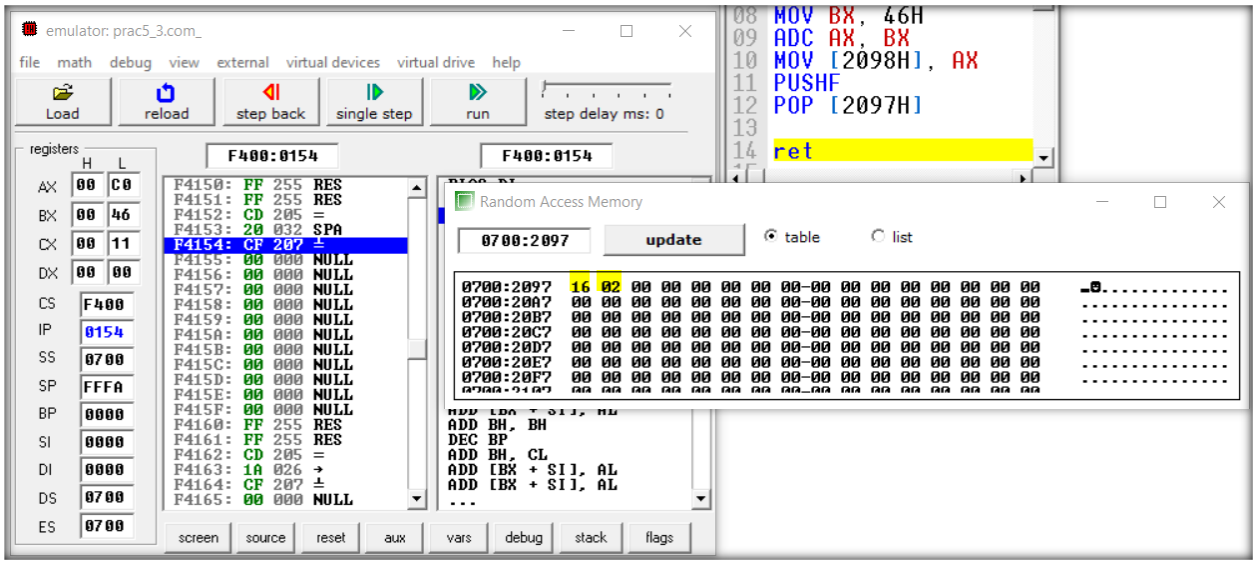
PRACTICAL: 5(A3)

AIM: Write a program to add the two Hex Numbers 7AH and 46H and to store the sum at memory location 2098 and flags status at 2097 location.

PROGRAM:

```
org 100h
MOV AX, 7AH
MOV BX, 46H
ADC AX, BX
MOV [2098H], AX
PUSHF
POP [2097H]
ret
```

OUTPUT:



CONCLUSION:

In this practical, we learned how to store flag status into the register using PUSHF and POP instructions.

PRACTICAL: 5(A4)

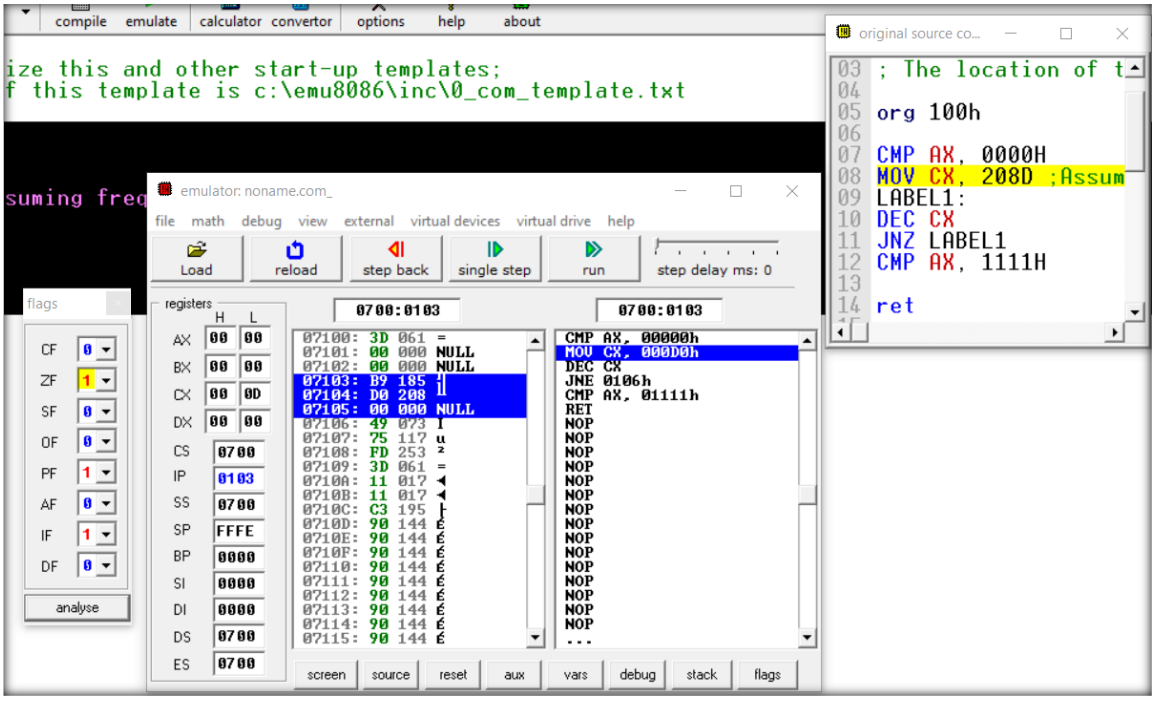
AIM: Write a 20 ms time delay subroutine using register pair BC. At the end of subroutine, clear the flag Z without affecting other flags and return to main program.

PROGRAM:

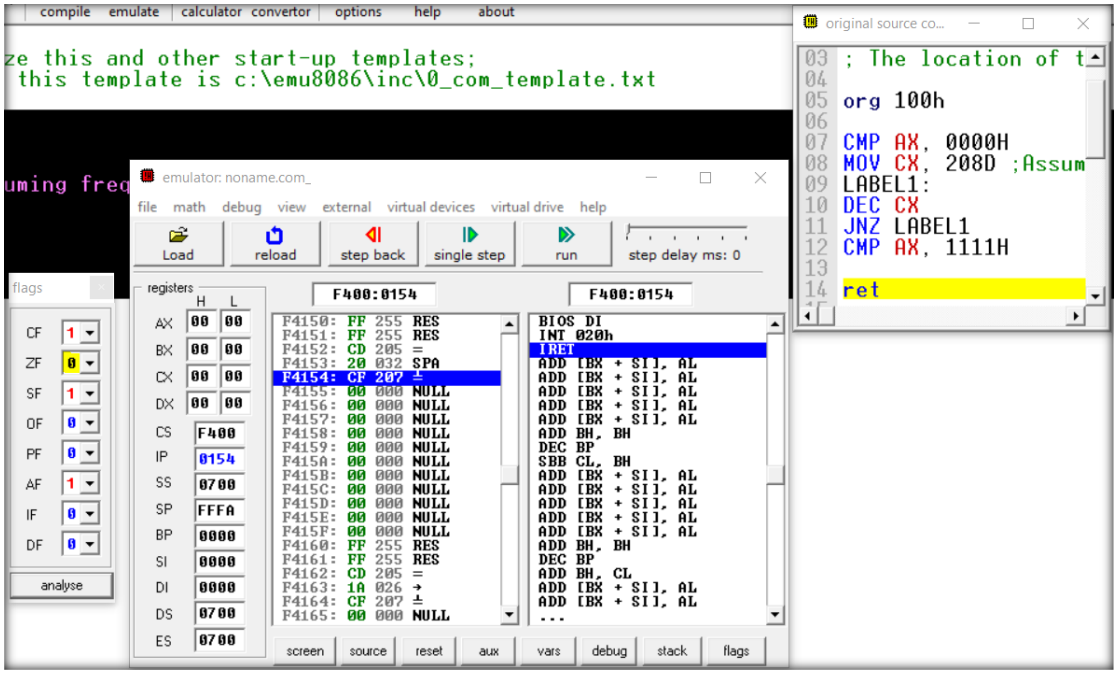
```
org 100h

CMP AX, 0000H

MOV CX, 208D ;Assuming freq=5MHz and n=12

LABEL1:

DEC CX

JNZ LABEL1

CMP AX, 1111H

ret
```

OUTPUT:

Set Zero flag at starting



Clear zero flag after delay



CONCLUSION:

In this practical, we implemented traffic delay loop and cleared zero flag after the subroutine of 20ms delay.

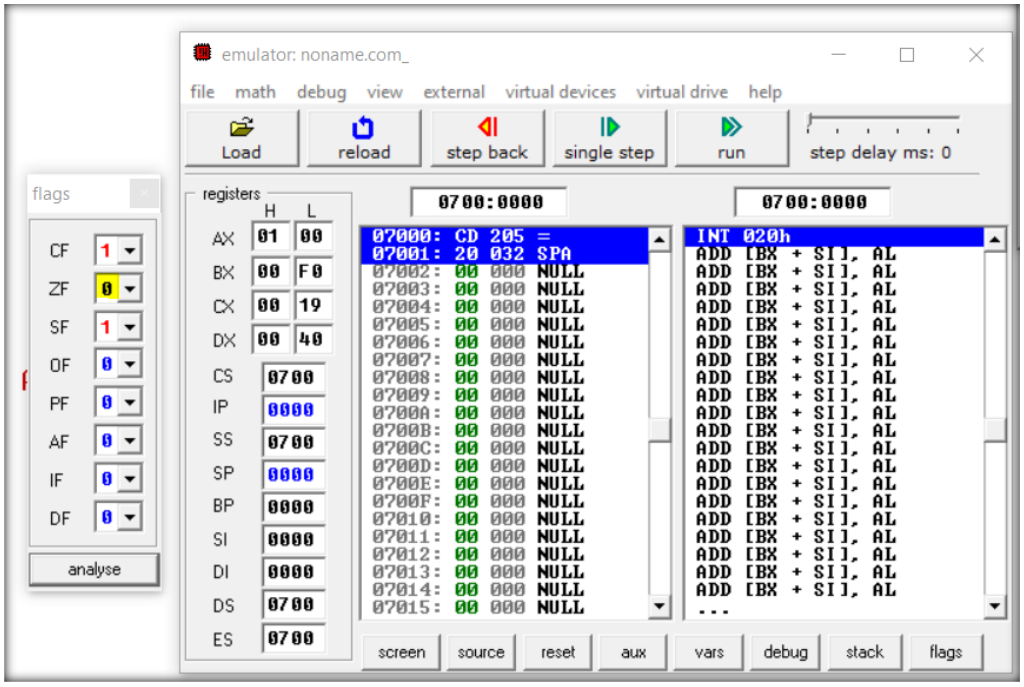
PRACTICAL: 5(A5)

AIM: Using a Subroutine, write a program which adds two hex number 10H and F0H and store result at 2040H location in memory. At the end of subroutine, clear the flag Z without affecting other flags and return to main program.

PROGRAM:

```
org 100h

MOV DX, 40H

PUSH DX

POPF

call p1

CMP CX, 1111H

ret

p1 PROC

   MOV AX, 10H

   MOV BX, 0F0H

   ADC AX, BX

   MOV [2040H], AX

   RET

p1 ENDP
```

OUTPUT:



CONCLUSION:

In this practical, we implemented traffic delay loop and cleared zero flag after the subroutine of 20ms delay.

PRACTICAL : 5(B6)

AIM: Write a program which set and resets zero flag at next iteration. (take number of iteration equal to 5)

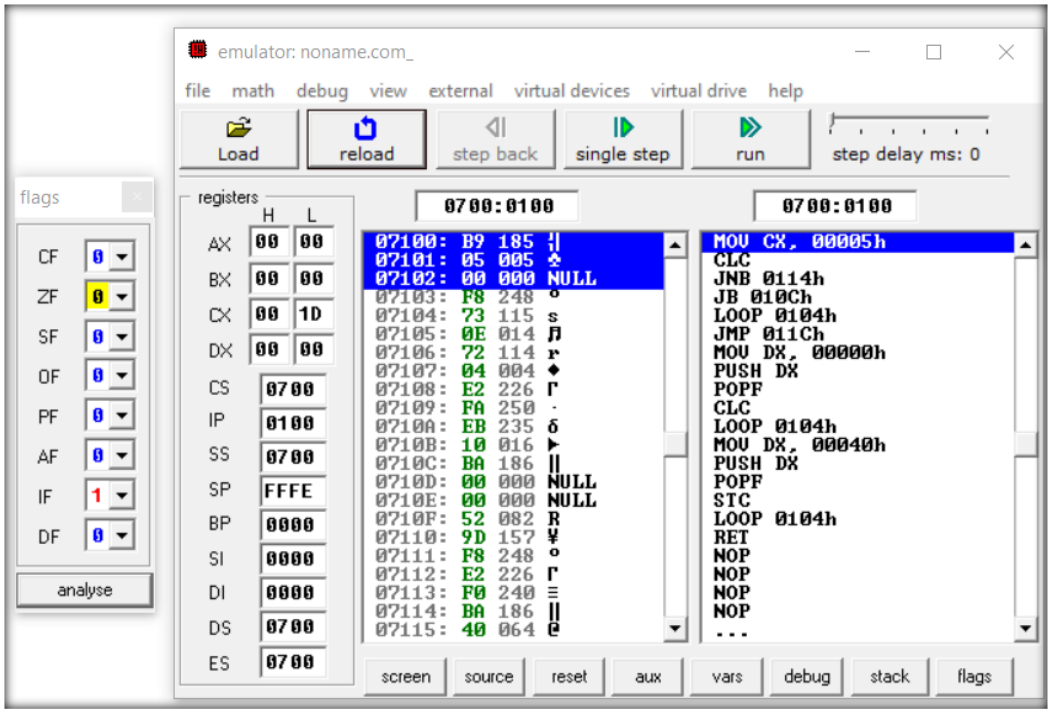
PROGRAM:

```
org 100h

MOV CX, 5H

CLC

LABEL1:

JNC SETZ

JC CLEARZ

LOOP LABEL1

JMP FINISH

CLEARZ:

MOV DX, 00H

PUSH DX

POPF

CLC

LOOP LABEL1

SETZ:

MOV DX, 40H

PUSH DX

POPF

STC
```

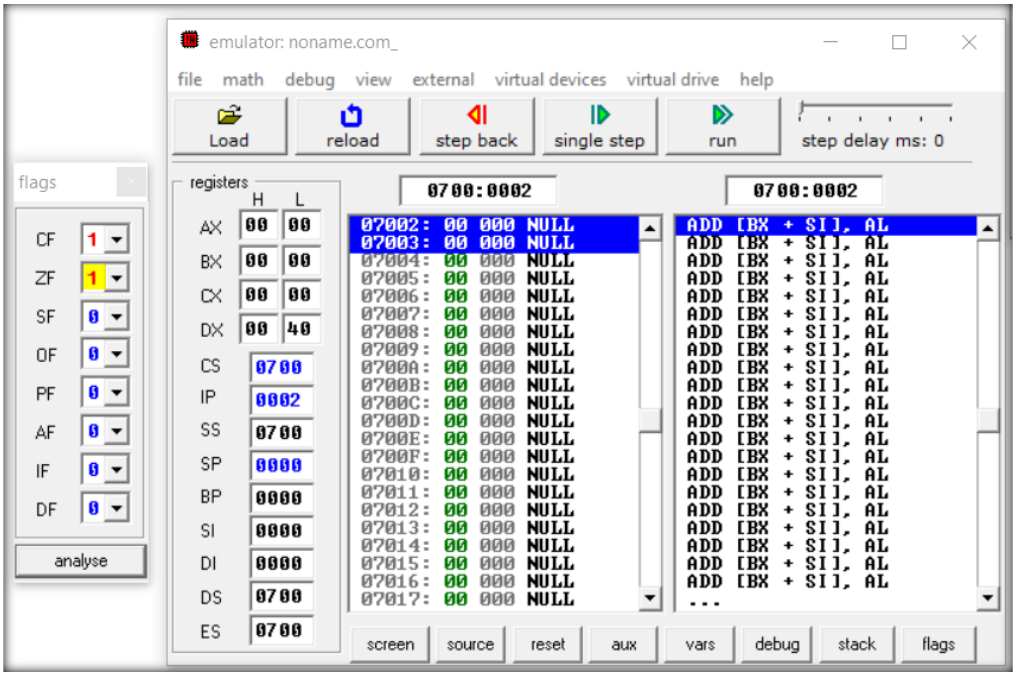
```
LOOP LABEl1

FINISH:

ret
```

OUTPUT:

Zero flag at odd iterations



Clear zero flag at even iteration



CONCLUSION:

In this practical, we toggled the zero flag at each iteration.

PRACTICAL: 5(B7)

AIM: Write a program to provide the given on/off time of three traffic lights. (Green, Yellow, Red). The signal sings are turned on/off by the data bits of an output port 1 as shown below:

Green will be ON for 15 ms otherwise OFF (On port 1 Value 1 will stay for 15 ms)

Yellow will be ON for 5 ms otherwise OFF (On port 1 Value 1 will stay for 5 ms)

Red will be ON 20 ms otherwise OFF (On port 1 Value 1 will stay for 20 ms).

PROGRAM:

```
; controlling external device with 8086 microprocessor.

; realistic test for c:\emu8086\devices\Traffic_Lights.exe

#start=Traffic_Lights.exe#

name "traffic"

mov ax, all_red

out 4, ax

mov si, offset situation

next:

mov ax, [si]

out 4, ax

; wait 5 seconds (5 million microseconds)

mov     cx, 4Ch    ;    004C4B40h = 5,000,000

mov     dx, 4B40h

mov     ah, 86h

int     15h
```

```
add si, 2 ; next situation
cmp si, sit_end
jb  next
mov si, offset situation
jmp next
;                   FEDC_BA98_7654_3210
situation       dw      0000_0011_0000_1100b
s1              dw      0000_0110_1001_1010b
s2              dw      0000_1000_0110_0001b
s3              dw      0000_1000_0110_0001b
s4              dw      0000_0100_1101_0011b
sit_end = $
all_red         equ     0000_0010_0100_1001b
```

OUTPUT:

CONCLUSION:

In this practical, we implemented traffic light program in EMU8086.

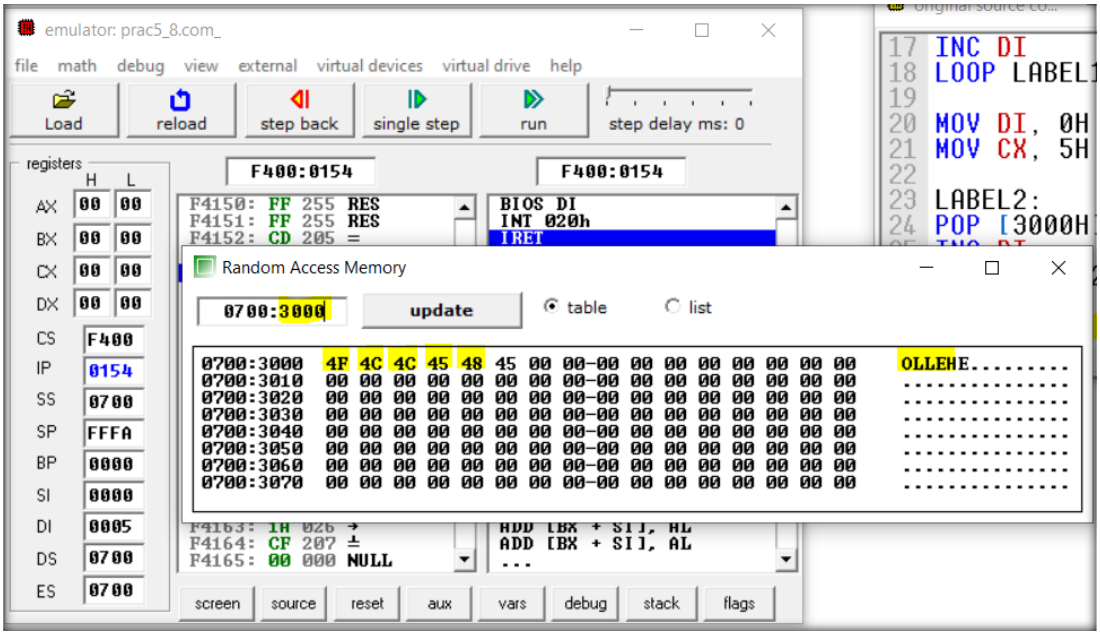
PRACTICAL: 5(C8)

AIM: Implement a program to reverse a string using stack operations and stored in same memory area.

PROGRAM:

```
org 100h

MOV [3000H], 48H ;H

MOV [3001H], 45H ;E

MOV [3002H], 4CH ;L

MOV [3003H], 4CH ;L

MOV [3004H], 4FH ;O

MOV DI, 0H

MOV CX, 5H

LABEL1:

PUSH [3000H] DI

INC DI

LOOP LABEL1

MOV DI, 0H

MOV CX, 5H

LABEL2:

POP [3000H] DI

INC DI

LOOP LABEL2

ret
```

OUTPUT:



CONCLUSION:

In this practical, we accessed the whole string using LOOP and stored it into stack using PUSH. We again accessed the same string using POP and stored the reverse string at same memory location.

PRACTICAL: 5(C9)

AIM: Calculate the sum of series of even numbers from the list of numbers. The length of the list is in memory location 2200H and the series itself begins from memory location 2201H. Assume the sum to be 8 bit number so you can ignore carries and store the sum at memory location 2210H.

PROGRAM:

```
org 100h

MOV [2200H], 8H

MOV [2201H], 1H

MOV [2202H], 5H

MOV [2203H], 8H

MOV [2204H], 12H

MOV [2205H], 0CH

MOV [2206H], 0BH

MOV [2207H], 4H

MOV [2208H], 7H

MOV CL, [2200H]

MOV SI, 2201H

MOV [2210H], 0H

LABEL1:

MOV AL, [SI]

TEST AL, 1B

JZ LABEL2

BACK:

INC SI
```

LOOP LABEL1

JMP FINISH

LABEL2:

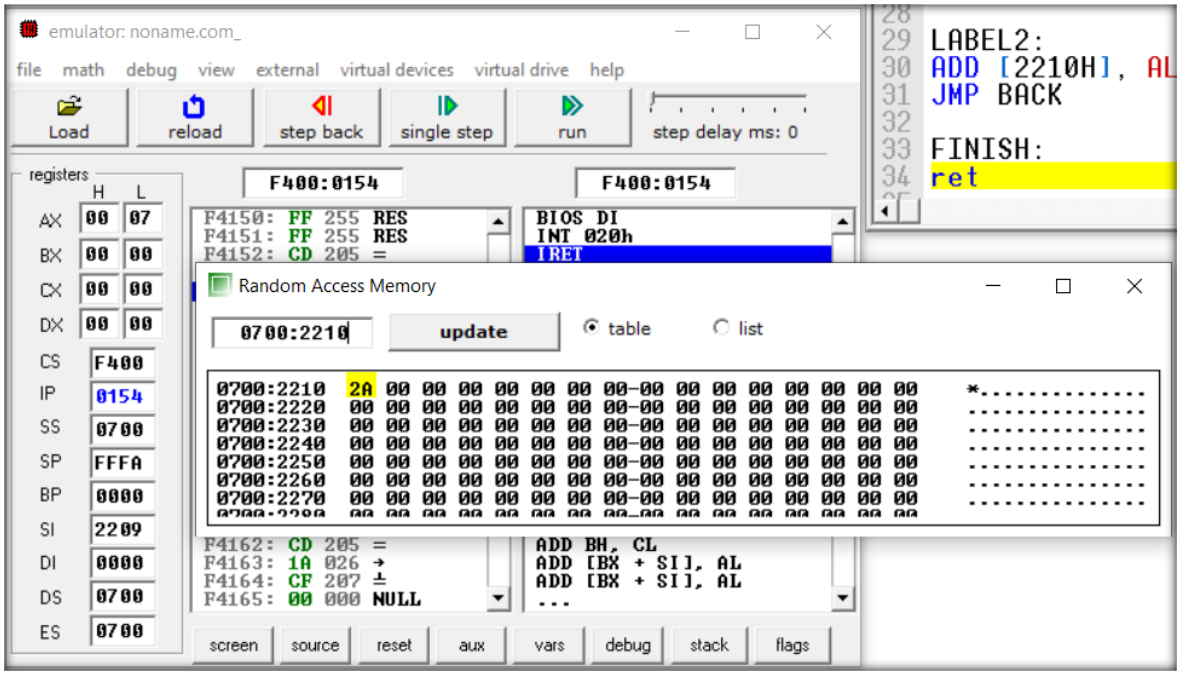
ADD [2210H], AL

JMP BACK

FINISH:

ret

OUTPUT:



CONCLUSION:

In this practical, we accessed the whole string using LOOP and computed the sum of even numbers using ADD and TEST instructions.

PRACTICAL: 5(C10)

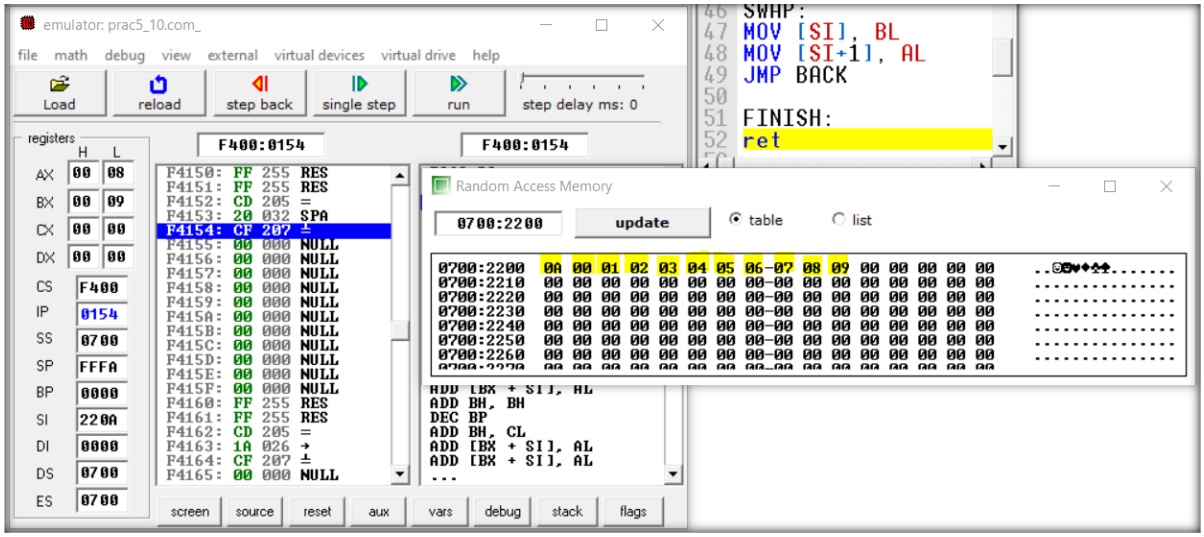
AIM: Write an assembly language program to arrange an array of 10 data in ascending order. The length of the list is in memory location 2200H and the series itself begins from memory location 2201H.

PROGRAM:

```
org 100h

MOV [2200H], 10

MOV [2201H], 4

MOV [2202H], 5

MOV [2203H], 7

MOV [2204H], 1

MOV [2205H], 0

MOV [2206H], 2

MOV [2207H], 6

MOV [2208H], 8

MOV [2209H], 9

MOV [220AH], 3

MOV CL, [2200H]

MOV DL, [2200H]

SUB DL, 1

MOV SI, 2201H

MOV AX, 40H

PUSH AX

POPF
```

```
LABEL1:
MOV SI, 2201H
MOV DL, [2200H]
SUB DL, 1
LABEL2:
MOV AL, [SI]
MOV BL, [SI+1]
CMP AL, BL
JNS SWAP
BACK:
INC SI
DEC DL
TEST DL, 0FFH
JNZ LABEL2
DEC CL
TEST CL, 0FFH
JNZ LABEL1
JMP FINISH
SWAP:
MOV [SI], BL
MOV [SI+1], AL
JMP BACK
FINISH:
ret
```

OUTPUT:



CONCLUSION:

In this practical, we sorted array using bubble sort with help of loops and index.

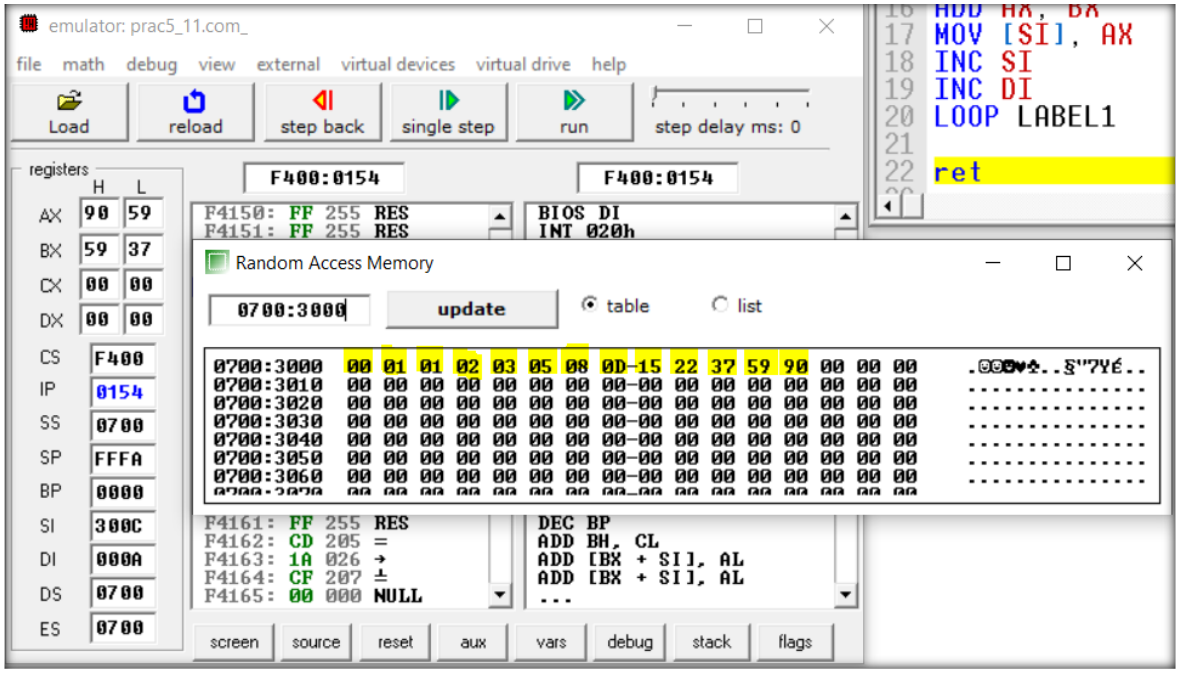
PRACTICAL: 5(C11)

AIM: Write an assembly language program to fill the memory locations starting from 3000h, with n Fibonacci numbers.

PROGRAM:

```
org 100h

MOV [3000H], 0H

MOV [3001H], 1H

MOV CX, 0AH

MOV SI, 3002H

MOV DI, 0H


LABEL1:

MOV AX, [3000H] DI

MOV BX, [3001H] DI

ADD AX, BX

MOV [SI], AX

INC SI

INC DI

LOOP LABEL1

ret
```

OUTPUT:



CONCLUSION:

In this practical, we implemented LOOP and Index Registers to calculate Fibonacci series.

PRACTICAL : 6(A)

AIM: Exploiting multiple core for compilation of various applications.

THEORY:

- A multi-core processor is a computer processor integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions, as if the computer had several processors.

- Coarse grain concurrency was all the rage for Java 5. The hardware reality has changed. The number of cores is increasing so applications must now search for fine grain parallelism (fork-join)
- As hardware becomes more parallel, more and more cores, software has to look for techniques to find more and more parallelism to keep the hardware busy.
- Clock rates have been increasing exponentially over the last 30 years or so. Allowed programmers to be lazy because a faster processor would be released that saved your butt. There wasn't a need to tune programs.
- That wait for faster processor game is up. Around 2003 clock rates stopped increasing. Hit the power wall. Faster processors require more power. Thinner chip conductor lines were required and the thinner lines can't dissipate the increased power without causing overheating which effects the resistance characteristics of the conductors. So you can't keep increasing clock rate.
- Fastest Intel CPU 4 or 5 years ago was 3.2 Ghz. Today it's about the same or even slower.
- Easier to build 2.6 Ghz or 2.8 Ghz chips. Moore's law wasn't repealed so we can cram more transistors on each wafer. So more processing power could be put on a chip which leads to putting more and more processing cores on a chip. This is multicore.
- Multicore systems are the trend. The number of cores will grow at exponential rate for the next 10 years. 4 cores at the low end. The high end 256 (Sun) and 800 (Azul) core systems.
- More cores per chip instead of faster chips. Moore's law has been redirected to multicore.
- The problem is it's harder to make a program go faster on a multicore system. A faster chip will run your program faster. If you have a 100 cores you program won't go faster unless you explicitly design it to take advantage of those chips.
- No free lunch anymore. Must now be able to partition your program so it can run faster by running on multiple cores. And you must be able keep doing that as the number of cores keeps improving.
- We need a way to specify programs so they can be made parallel as topologies change by adding more cores.
- As hardware evolves platforms must evolve to take advantage of the new hardware. Started off with course grain tasks which was sufficient given the number of cores. This approach won't work as the number cores increase.
- Must find finer-grained parallelism. Example sorting and searching data. Opportunities around data. The data can for sorting can be chunked and sorted and the brought together with a merge sort. Searching can be done in parallel by searching subregions of the data and merging the results.

- Parallel solutions use more CPU in aggregate because of the coordination needed and that data needs to be handled more than once (merge). But the result is faster because it's done in parallel. This adds business value. Faster is better for humans.

CONCLUSION:

In this practical, we learned about multiple core CPU and notes why parallelism is important now.

PRACTICAL : 6(B)

AIM: Affinity allocation: Windows and Linux Machines

THEORY:

Processor affinity, or CPU pinning or "cache affinity", enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU.

This can be viewed as a modification of the native central queue scheduling algorithm in a symmetric multiprocessing operating system.

Each item in the queue has a tag indicating its kin processor.

At the time of resource allocation, each task is allocated to its kin processor in preference to others.

Processor affinity can effectively reduce cache problems, but it does not reduce the persistent load-balancing problem.

Also note that processor affinity becomes more complicated in systems with non-uniform architectures.

For example, a system with two dual-core hyper-threaded CPUs presents a challenge to a scheduling algorithm.

On Linux, the CPU affinity of a process can be altered with the taskset(1) program and the sched_setaffinity(2) system call.

The affinity of a thread can be altered with one of the library functions: pthread_setaffinity_np(3) or pthread_attr_setaffinity_np(3).

On Windows NT and its successors, thread and process CPU affinities can be set separately by using SetThreadAffinityMask and SetProcessAffinityMask API calls or via the Task Manager interface (for process affinity only).

CONCLUSION:

In this practical, we learned affinity allocation and different system calls for Linux and Windows for that.

PRACTICAL : 6(C)

AIM: Shifting all processes on single core.

THEORY:

- On a dual-core processor, you can entirely disable the second core through the System Configuration menu or partially disable it using processor affinity options.
- System Configuration lets you set how many processor cores Windows accesses, while processor affinity lets you control which cores any given program can use.
- This approach can be useful you're trying to keep processes from monopolizing all your computer resources.
- These options do not appear on single-core processor computers.
- Processor affinity forces specific program threads to run on specific processor cores.
- By default, all programs have access to all cores when they launch.
- Windows resets any changes to processor affinity settings when a program is closed, so the next time it's launched it has access to all cores again.
- If you are running two tasks at the same time, like editing a large photograph in Adobe Photoshop and watching an HD movie in Windows Media Player, affinity creates the option to assign each of these tasks to one of the cores to keep them from interfering with each other.
- This way, if one program runs a large data crunch, it won't cause the other to slow down.
- You control processor affinity through the Task Manager, and the functionality is available in all versions of Windows after XP.
- Launch the Task Manager by pressing "Ctrl-Shift-Esc," select the "Processes" tab, and then open the affinity menu by right-clicking on any program task listed and selecting "Set Affinity." The Processes tab's Description column displays program names to help you identify individual programs, while the processor affinity menu features a list of all the cores on the computer and an All Processors option.
- The selected program can use any core that has a check mark in the box next to it; disable core access by unchecking individual core boxes, and then choosing "OK."

CONCLUSION:

In this practical, we understood how to use affinity control to stick process to single core so that we can force just one core to do all the work.

PRACTICAL : 7

AIM: Intel and AMD: Architectural Differences.

THEORY:

- Intel® is the world's oldest and mostestablished microprocessor company, producing the world's most popular microprocessor chips.
- Although perhaps best known for its PC processors, Intel devices are used in virtually every field of electronics, including automotive, industrial, automation, robotics, consumer electronics, image processing, networking, encryption, military, construction, medical, energy, and other industries.
- Designers unfamiliar with the Intel architecture may have concerns about the architecture's fundamental concepts, its inner workings, or its complexity.
- The goal of this paper is to educate skilled developers with no previous exposure to the Intel architecture and to provide guidance regarding system components and concepts.
- The ARM architecture processor is an advanced reduced instruction set computing [RISC] machine and it's a 32bit reduced instruction set computer (RISC) microcontroller. It was introduced by the Acron computer organization in 1987.
- This ARM is a family of microcontroller developed by makers like ST Microelectronics,Motorola, and so on. The ARM architecture comes with totally different versions like ARMv1, ARMv2, etc., and, each one has its own advantage and disadvantages.
- Intel processors (commonly referred to as X86 in correlation with Windows 32-bit programs) use Complex Instruction Set Computing while ARM uses Reduced Instruction Set Computing. While both perform commands rather quickly in 2020, the former uses slightly more complex instruction with several cycles.
- Intel processors are commonly found in larger tech like desktop computers while ARM is often found in mobile devices. One contributing factor for this is that ARM processors rely heavily on software for performance features while Intel relies on hardware.
- ARM works better in smaller tech that does not have access to a power source at all times and Intel focuses more on performance making it the better processor for larger tech.
- The ARM processors not only use less battery life thanks to their single-cycle computing set, but they also have a reduced operating temperature than the Intel processors. Intel processors are focused on performance, and for most PC or laptop users this isn't a problem at all because the computer is constantly connected to power.
- ARM chips are usually slower than their Intel counterparts. This is largely due to the fact that they are designed to commute with low power consumption. While most users wouldn't notice a difference in their respective devices, Intel processors are designed for faster computing.
- Intel was once a part of a few Android mobile devices but the ARM processors still reign in this market.

CONCLUSION:

In this practical, we learned about Intel and ARM architecture and learned about difference between them.

PRACTICAL : 8(A)

AIM: Multicore Programming in Linux.

THEORY:

- GNU Linux is a symmetric multiprocessing (SMP) operating system - it know how to
- manage a multiple processor with a single shared main memory.
- Linux used to use a "big kernel lock" to provide the concurrency control required by symmetric multiprocessing (SMP) systems, used whenever a thread entered kernel space, and is released when the thread returns to user space (a system call is the archetypal example).
- BKL threads in user space can run concurrently on any available processors or processor cores, but no more than one thread can run in kernel space. The Linux kernel had a big kernel lock until it was finally replaced by more fine-grained locking mechanisms by Arnd Bergmann in 2011.
- By 2.2 big kernel lock for most of kernel, interrupts own lock. 2.4 more fine grained locking, still several common global locks. 2.6 further BLK reductions, new subsystems (multi queue scheduler, multi flow networking) 2.6.35 included (thank you Google) Receive Packet Steering (RPS) and Receive Flow Steering (RFS) to spread the load of network handling across the CPUs available in the system.
- Network cards have improved the bandwidth to the point where it's hard for a single modern CPU to keep up. In 2.6.36 Tilera architecture support (Tile CPUs) was added. In 2.6.38 automatic process grouping was added, allowing processes with the same sessionID to be grouped on the same processor as a single scheduling entity.

CONCLUSION:

In this practical, we had brief insight about multicore programming in Linux.

PRACTICAL : 8(B)

AIM: Multicore Programming in Windows.

THEORY:

- Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor or the ability to allocate tasks between them.
- There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple dies in one package, multiple packages in one system unit, etc.).
- Pthreads is a POSIX standard for threads, communicating though shared memory. Pthreads defines a set of C programming language types, functions and constants.
- It is implemented with a pthread.h header and a thread library. Programmers can use Pthreads to create, manipulate and manage threads.
- In particular, programmers can synchronize between threads using mutexes, condition variables and semaphores.
- A C++ library that run on top of native threads Programmers specify tasks rather than threads: Tasks are objects.
- Each task object has an input parameter and an output parameter. One needs to define (at least) the methods: one for creating a task and one for executing it.
- Tasks are launched by a spawn or a spawn and wait for all statement. Tasks are automatically load-balanced across the threads using the work stealing principle. TBB Developed by Intel and focus on performance.
- Several compilers available, both open-source and Visual Studio. Runs on top of native threads Linguistic extensions to C/C++ or Fortran in the form of compiler pragmas (compiler directives): # pragma omp task shared(x) implies that the next statement is an independent task; moreover sharing of memory is managed explicitly other pragmas express directives for scheduling, loop parallelism amd data aggregation.
- Supports loop parallelism and, more recently in Version 3.0, task parallelism with dynamic scheduling.
- OpenMP provides a variety of synchronization constructs (barriers, mutual-exclusion locks, etc.)

CONCLUSION:

In this practical, we briefly learned about multicore programming using Windows.

PRACTICAL : 9

AIM: Harddisk I/O performance measurement, Pendrive I/O performance measurement.

THEORY:

- IOPS (input/output operations per second) is the standard unit of measurement for the maximum number of reads and writes to non-contiguous storage locations.
- IOPS is pronounced EYE-OPS.
- IOPS is frequently referenced by storage vendors to characterize performance in solid-state drives (SSD), hard disk drives (HDD) and storage area networks. However, an IOPS number is not an actual benchmark, and numbers promoted by vendors may not correspond to real-world performance.
- IOPS is often measured with an open source network testing tool called an Iometer. An Iometer determines peak IOPS under differing read/write conditions.
- Measuring both IOPS and latency can help a network administrator predict how much load a network can handle without performance being negatively affected. Intel discontinued work on Iometer in 2006, so some may find the tool to be dated.
- It is possible to calculate IOPS without an Iometer, but results will vary depending on the performance category of the workload. In general, there are three types of workload performance: random, sequential and a mixture of the two. RAID can also impact IOPS calculations, since each write operation results in multiple writes to the storage array.
- IOPS can be measured using an online IOPS calculator, which determines IOPS based on the drive speed, average read seek time and average write seek time.
- HDDs use the standard equation to determine IOPS, but SSDs perform differently. With HDDs, IOPS is dependent on the seek time, but SSDs are primarily dependent on the device's internal controller. SSD performance changes over time, peaking early on. However, even after it drops into the steady state, SSDs still outperform HDDs in terms of IOPS. HDDs also grapple with higher latency and longer read/write times.

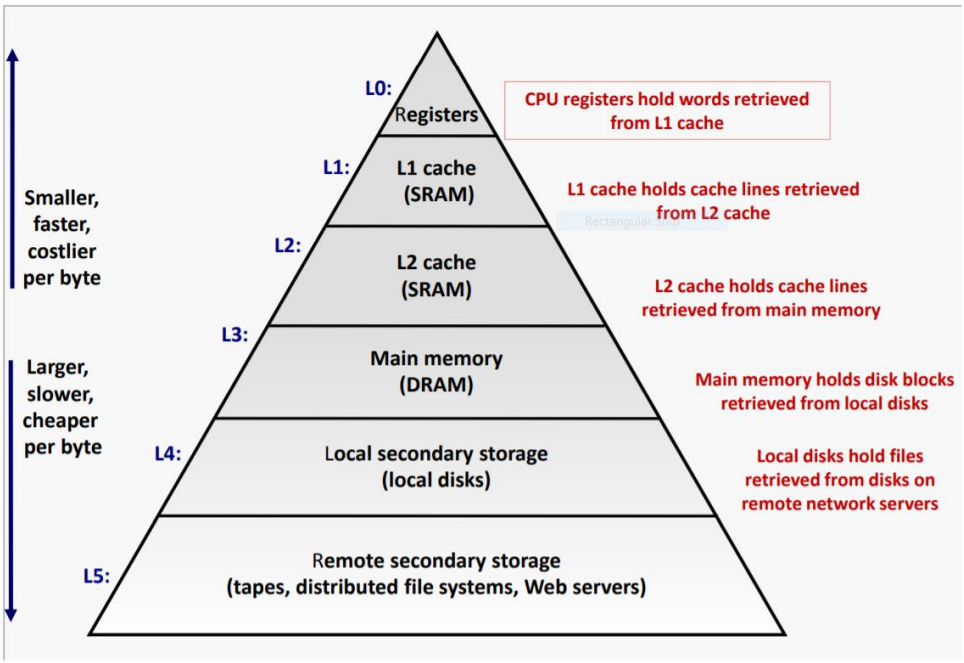
CONCLUSION:

In this practical, we got to know about IOPS and how parameters of Input/Output devices are measured and compared.

PRACTICAL : 10

AIM: L1, L2 & L3 Cache Rebuilt and performance measurement.

THEORY:



- Cache: a smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy: – For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work? – Because of locality, programs tend to access the data at level k more often than they access the data at level k+1. – Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Primary cache attached to CPU – Small, but fast Level-2 cache services misses from primary cache – Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache
- L1 (Level 1), L2, L3 cache are some specialized memory which work hand in hand to improve computer performance. When a request is made to the system, CPU has some set of instructions to execute, which it fetches from the RAM. Thus to cut down delay, CPU maintains a cache with some data which it anticipates it will be needed.(L1) Level 1 Cache(2KB - 64KB) - Instructions are first searched in this cache.
- L1 cache very small in comparison to others, thus making it faster than the rest.(L2) Level 2 Cache(256KB - 512KB) - If the instructions are not present in the L1 cache then it looks in the L2 cache, which is a slightly larger pool of cache, thus accompanied by

some latency.(L3) Level 3 Cache (1MB -8MB) - With each cache miss, it proceeds to the next level cache.

- This is the largest among the all the cache, even though it is slower, its still faster than the RAM.Now you know what cache is and what different level of cache are.And that 6MB value of the L3 Cache in your Intel 4700MQ microprocessor is actually the memory size of that Cache. Thus Cache improves the overall performance of the CPU but these numbers shouldn't be considered while purchasing any system. Look at the benchmarks of the CPU as a whole.
- A CPU with similar architecture but with more cache wouldn't make any noticeable difference. The technology these days have advanced to such a point that the specs of a CPU are just meaningless.

CONCLUSION:

In this practical, we learned about Cache and different levels of Cache and we measures their performance.