**CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**FACULTY OF TECHNOLOGY & ENGINEERING**

**DEVANG PATEL INSTITUTE OF ADVANCED TECHNOLOGY AND RESEARCH**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**SUBJECT NAME:** MICROPROCESSOR ARCHITECTURE & ASSEMBLY LANGUAGE PROGRAMMING

**SUBJECT CODE:** CE341

**SEMESTER:** V

**ACADEMIC YEAR:** 2020-21

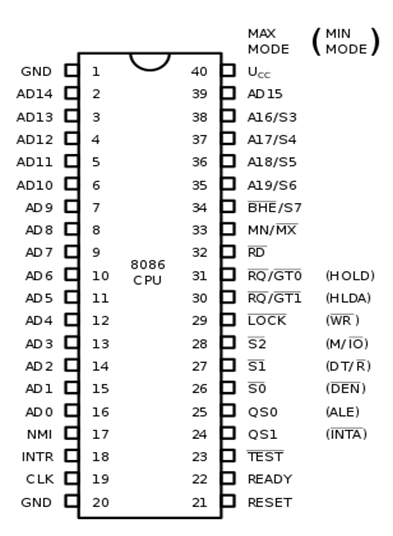
**PRACTICAL - 1**

**AIM : Introduction to 8086** **Microprocessor & Assembly Language Programming**

**8086 Microprocessor:**

* **8086 microprocessor is the enhanced version of 8085 microprocessor. It was designed by Intel in 1976.**
* **The 8086 microprocessor is a16-bit, N-channel, HMOS microprocessor. Where the HMOS is used for "High-speed Metal Oxide Semiconductor".**
* **8086 is built on a single semiconductor chip and packaged in a 40-pin IC package. The type of package is DIP (Dual Inline Package).**
* **8086 uses 20 address lines and 16 data- lines. It can directly address up to 220 = 1 Mbyte of memory.**
* **It consists of a powerful instruction set, which provides operation like division and multiplication very quickly.**
* **8086 is designed to operate in two modes, i.e., Minimum and Maximum mode.**

**8086 Pin Configuration:**

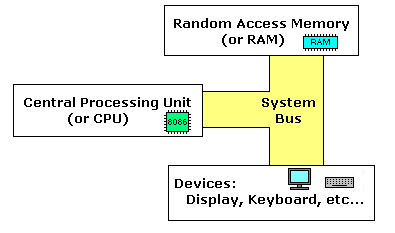


**Assembly Language Programming:**

Assembly language is a low level programming language. In this there is a very strong correspondence between the instructions in the language and the [architecture's](https://en.wikipedia.org/wiki/Computer_architecture) [machine code](https://en.wikipedia.org/wiki/Machine_code) [instructions](https://en.wikipedia.org/wiki/Instruction_set_architecture). Because assembly depends on the machine code instructions, every assembler has its own assembly language which is designed for exactly one specific computer architecture. Assembly language may also be called symbolic machine code.

Assembly code is converted into executable machine code by a [utility program](https://en.wikipedia.org/wiki/Utility_software) referred to as an [assembler](https://en.wikipedia.org/wiki/Assembly_language#Assembler). The conversion process is referred to as assembly, as in assembling the [source code](https://en.wikipedia.org/wiki/Source_code). Assembly language usually has one statement per machine instruction but [comments](https://en.wikipedia.org/wiki/Comment_(computer_programming)) and statements that are assembler [directives](https://en.wikipedia.org/wiki/Directive_(programming)), [macros](https://en.wikipedia.org/wiki/Macro_instruction), and symbolic [labels of program](https://en.wikipedia.org/wiki/Label_(programming)) and [memory locations](https://en.wikipedia.org/wiki/Memory_location) are often also supported.

You need to get some knowledge about computer structure in order to understand anything. The simple computer model as I see it:



* The system bus (shown in yellow) connects the various components of a computer.
* The CPU is the heart of the computer, most of computations occur inside the CPU.
* RAM is a place to where the programs are loaded in order to be executed.

**CONCLUSION:**

In this practical we studied the basics of 8086 Microprocessor about its pin diagram and also we were introduced to the assembly language programming.

**PRACTICAL – 2**

PROGRAM 2.1

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

**CODE:**

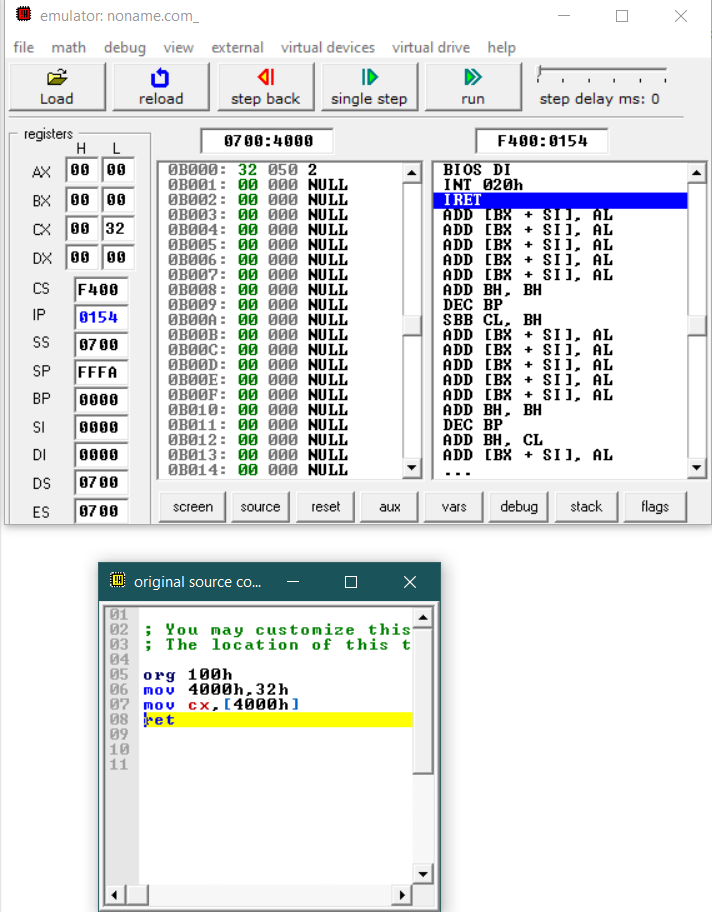
org 100h

mov 4000h,32h

mov cx,[4000h]

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned how to store hexadecimal value in a memory location.

PROGRAM 2.2

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

**CODE:**

org 100h

mov 2000h,10h

mov 4000h,20h

mov al,[2000h]

mov ah,[4000h]

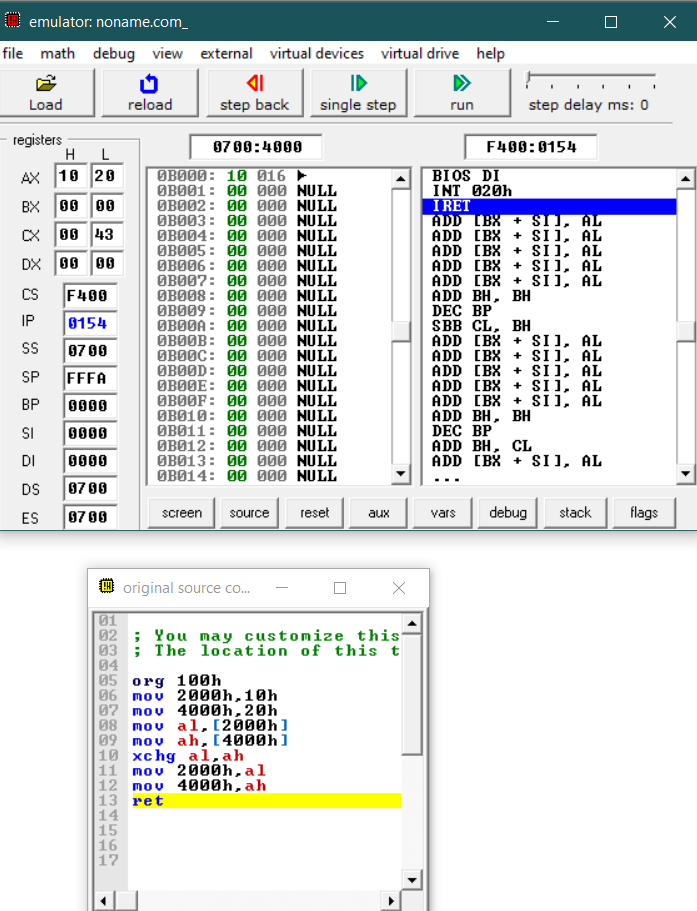
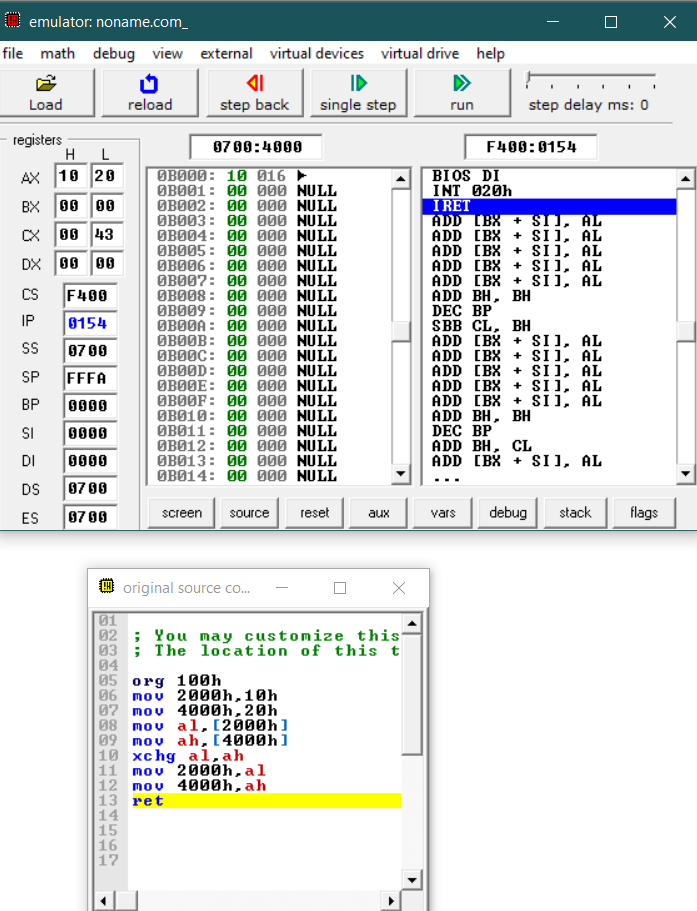
xchg al,ah

mov 2000h,al

mov 4000h,ah

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned how to exchange values between two memory locations.

PROGRAM 2.3

**AIM : Convert the below given C Program into Assembly Language**.

main()

{

int l,m,n,o,p;

l=m+n-o+p;

}

**CODE:**

org 100h

mov [1000h],1h

mov [1010h],2h

mov [1020h],3h

mov [1030h],4h

mov al,[1000h]

mov bl,[1010h]

mov cl,[1020h]

mov dl,[1030h]

add al,bl

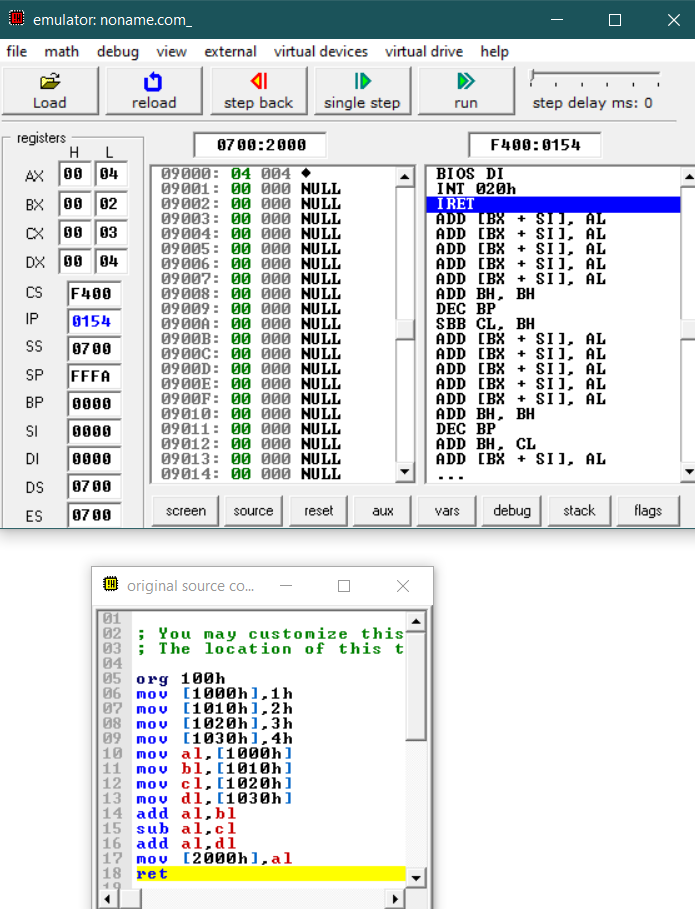
sub al,cl

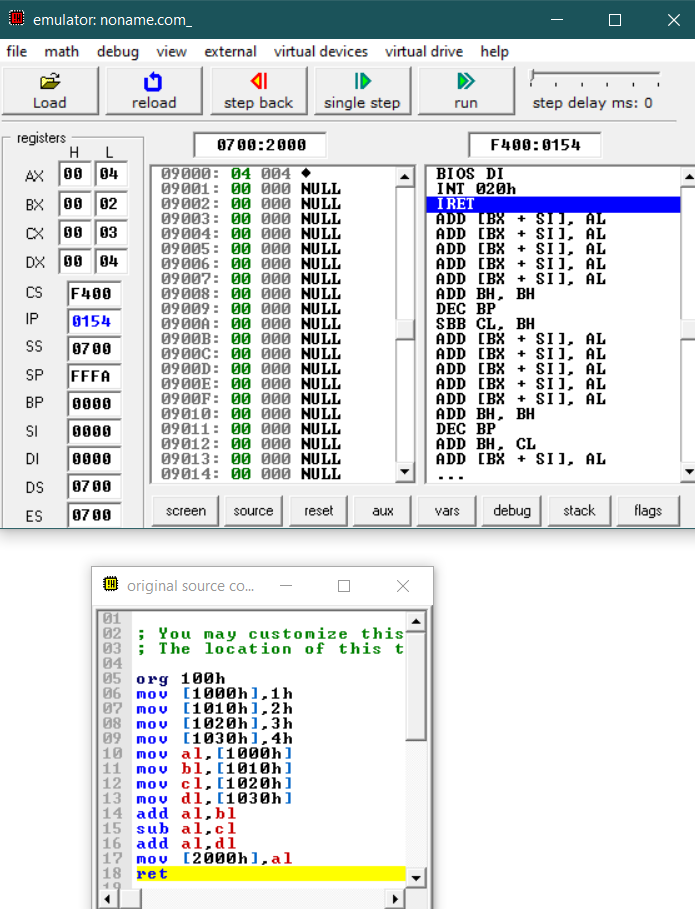
add al,dl

mov [2000h],al

ret

**OUTPUT:**





**CONCLUSION:**

In this practical we converted a given C-language code of addition & subtraction multiple numbers into assembly language.

PROGRAM 2.4

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

**CODE:**

org 100h

mov [2000h],10h

mov [4001h],12h

mov al,[4001h]

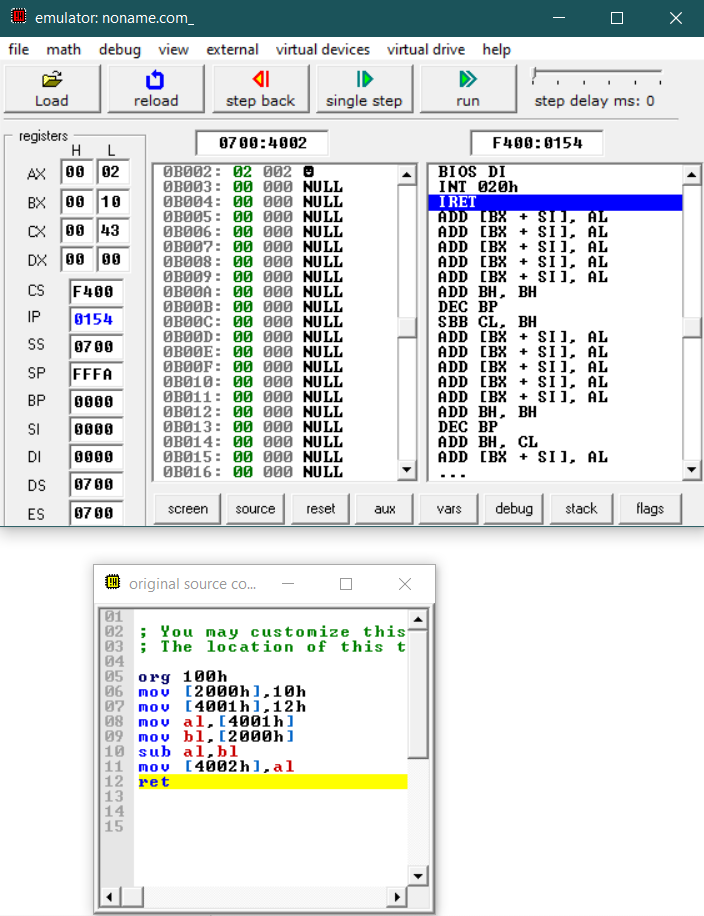
mov bl,[2000h]

sub al,bl

mov [4002h],al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned to subtract the data of one memory location from another memory location.

PROGRAM 2.5

**AIM : Subtract 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],0F0h

mov [4001h],30h

mov [4002h],40h

mov [4003h],50h

mov ax,[4000h]

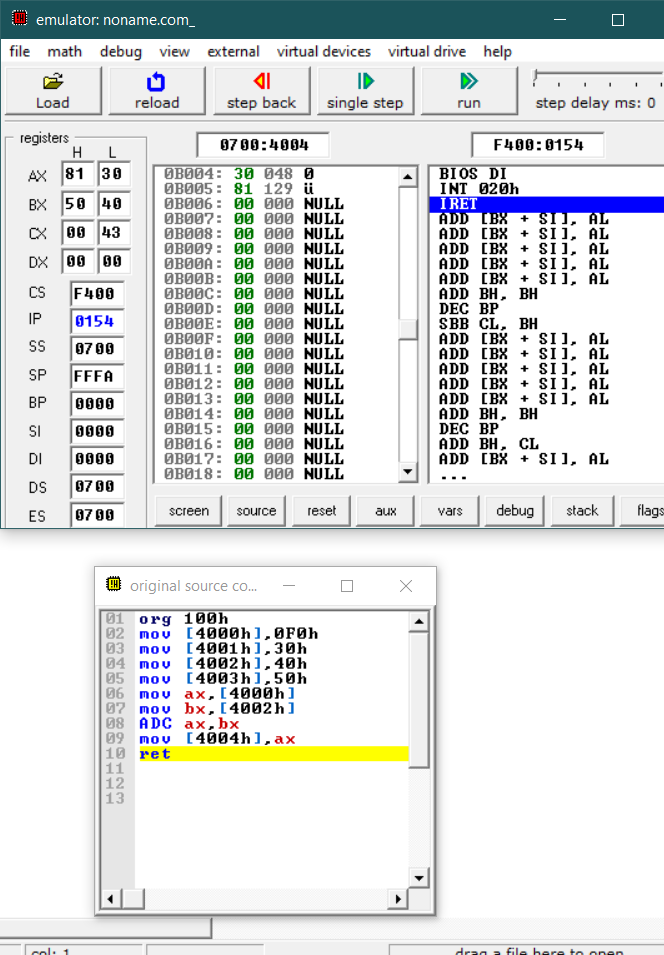
mov bx,[4002h]

ADC ax,bx

mov [4004h],ax

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned to add two 16-bit hexadecimal numbers stored in two continuous memory locations

PROGRAM 2.6

**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],40h

mov [4001h],30h

mov [4002h],10h

mov [4003h],20h

mov ax,[4000h]

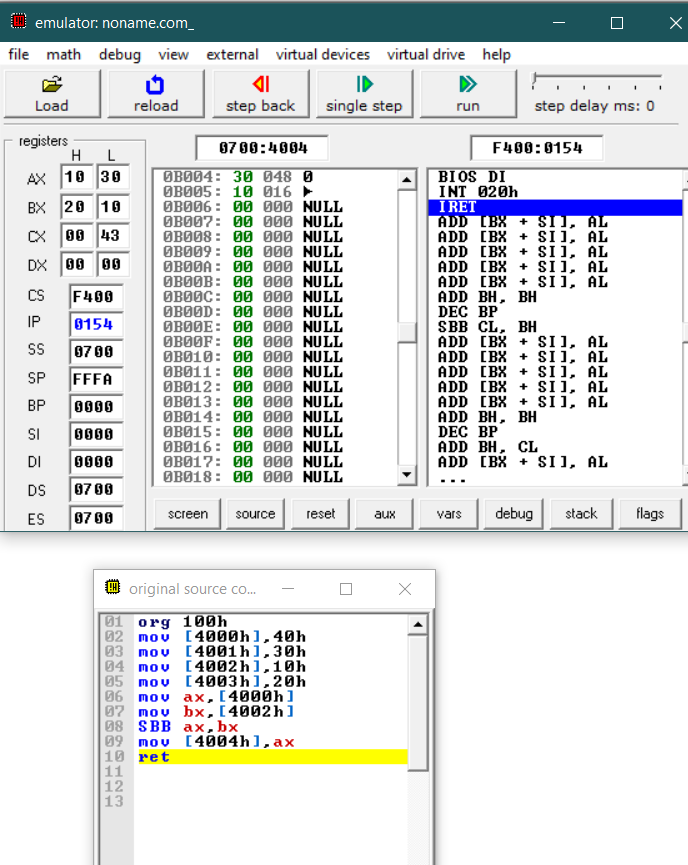
mov bx,[4002h]

SBB ax,bx

mov [4004h],ax

ret

**OUTPUT:**



PROGRAM 2.7

**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],10h

mov [4001h],20h

mov [4002h],30h

mov [4003h],40h

mov [4004h],50h

mov [4005h],60h

mov [4006h],70h

mov [4007h],80h

mov ax,[4000h]

mov bx,[4002h]

mov cx,[4004h]

mov dx,[4006h]

ADC ax,cx

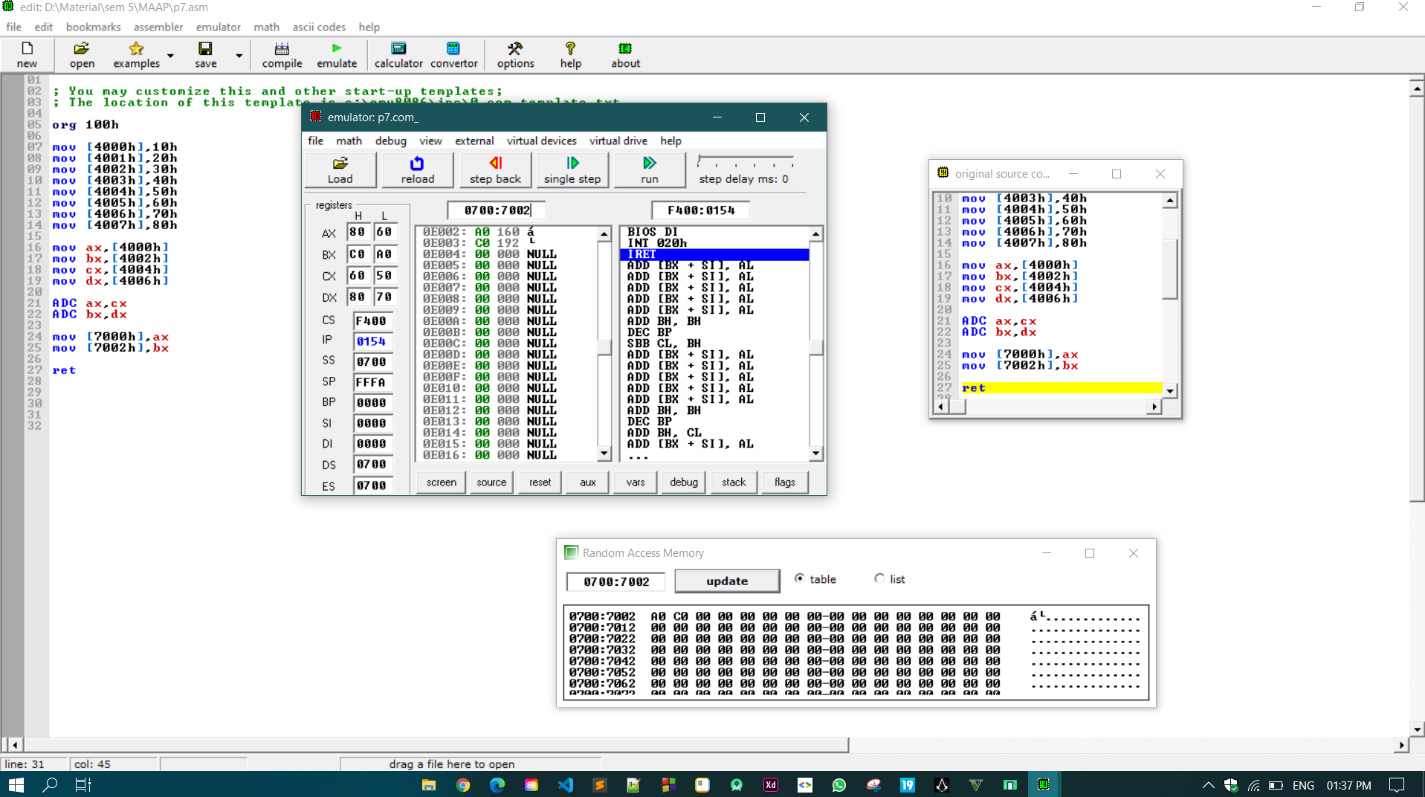
ADC bx,dx

mov [7000h],ax

mov [7002h],bx

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned to add two 32-bit hexadecimal numbers stored in four continuous memory locations.

PROGRAM 2.8

**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],80h

mov [4001h],70h

mov [4002h],60h

mov [4003h],50h

mov [4004h],40h

mov [4005h],30h

mov [4006h],20h

mov [4007h],10h

mov ax,[4000h]

mov bx,[4002h]

mov cx,[4004h]

mov dx,[4006h]

SBB ax,cx

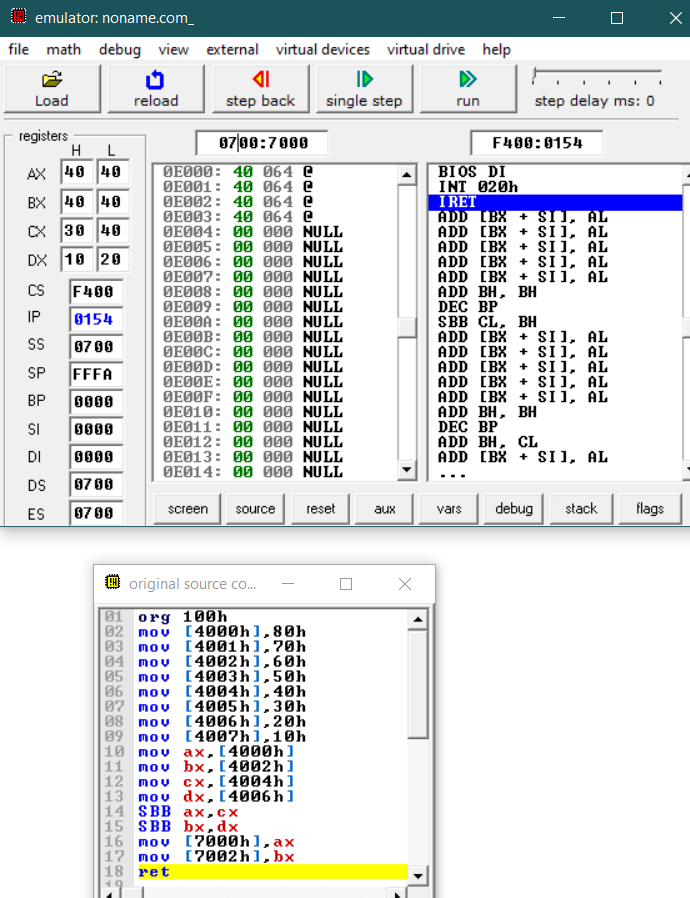
SBB bx,dx

mov [7000h],ax

mov [7002h],bx

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned to subtract two 32-bit hexadecimal numbers stored in four continuous memory locations.

PROGRAM 2.9

**AIM : Write an assembly language program to convert temperature in F to C.**

**CODE:**

org 100h

mov [4000h],104

mov al,[4000h]

mov bl,5

sbb al,32

mul bl

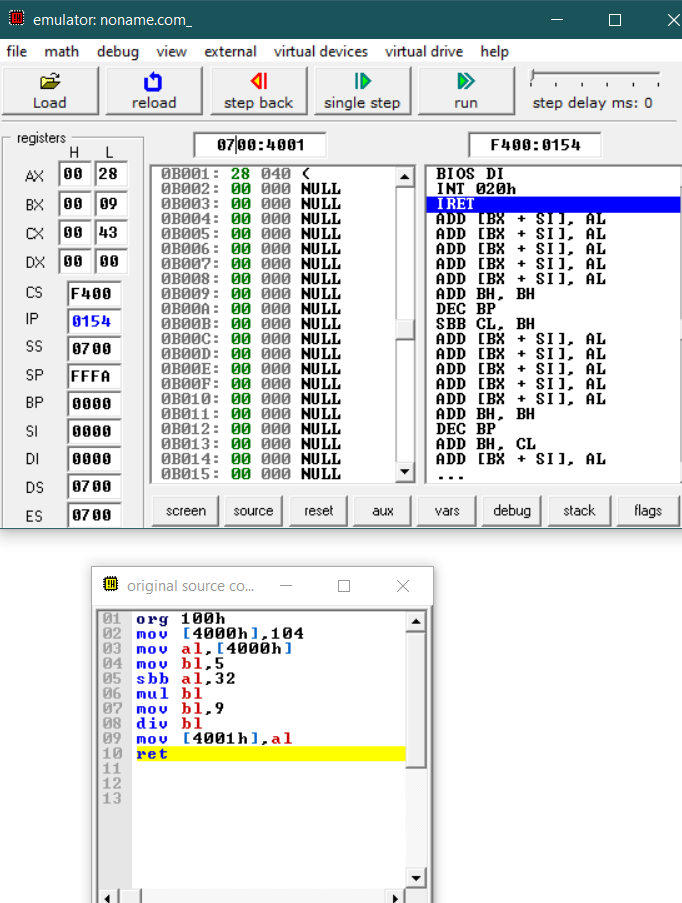
mov bl,9

div bl

mov [4001h],al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned to code a assembly language program to convert temperature from Fahrenheit to Celsius.

**PRACTICAL – 3**

PROGRAM 3.1

**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],01010101b

mov [4001h],10101010b

mov al,[4000h]

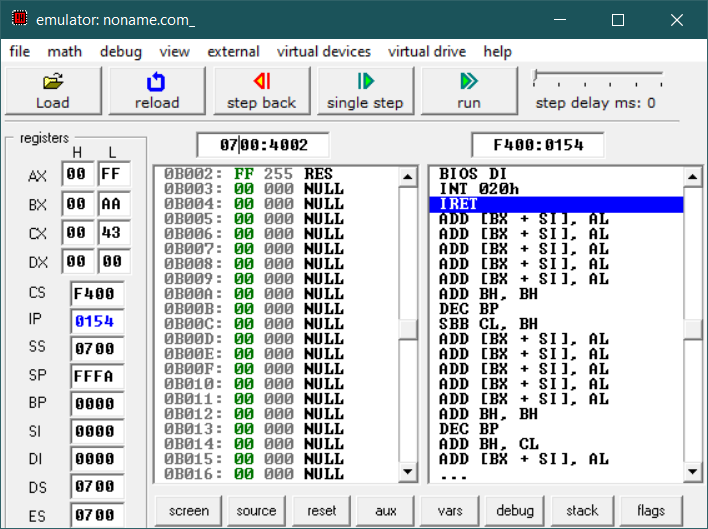
mov bl,[4001h]

OR al,bl

mov [4002h],al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we applied bit-wise OR operator to perform a selective set operation.

PROGRAM 3.2

**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],01010101b

mov [4001h],10101010b

mov al,[4000h]

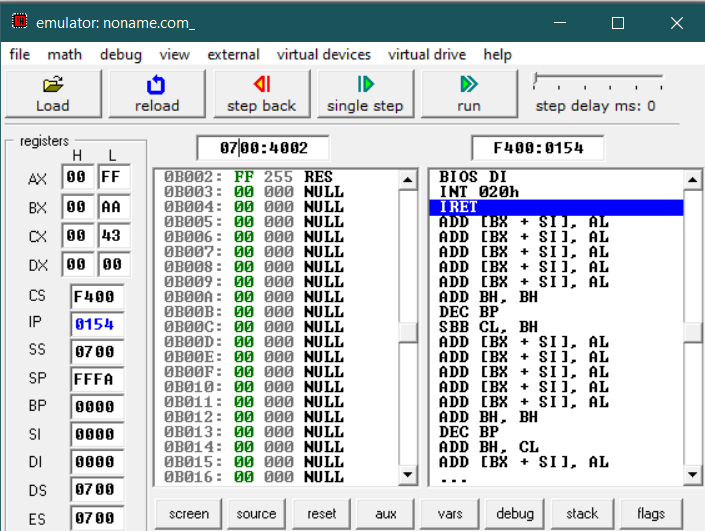
mov bl,[4001h]

XOR al,bl

mov [4002h],al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we applied bit-wise XOR operator to perform a selective set operation.

PROGRAM 3.3

**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],01010101b

mov [4001h],01010101b

mov al,[4000h]

mov bl,[4001h]

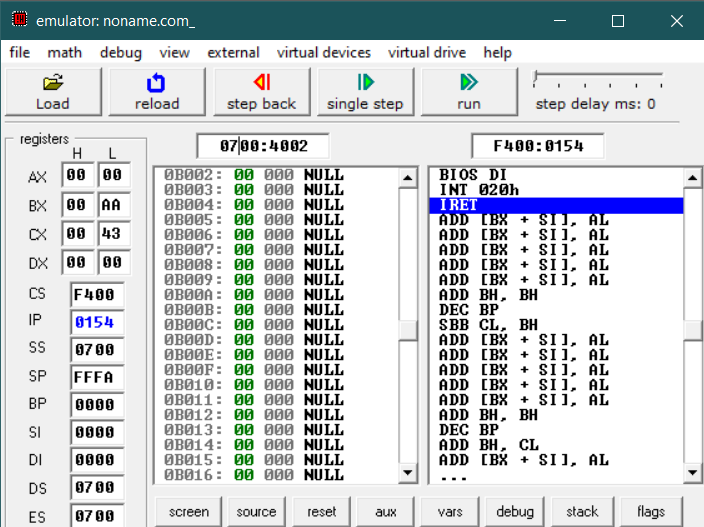
NOT bl

AND al,bl

mov [4002h],al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we applied a specific set of bit-wise operation to perform a selective clear operation.

PROGRAM 3.4

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

**CODE:**

org 100h

mov [2000h],01010101b

mov [2001h],10101010b

mov al,[2000h]

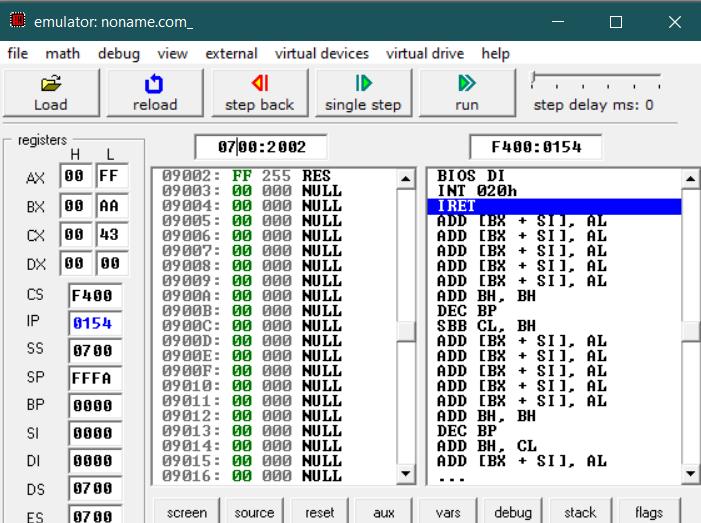
mov bl,[2001h]

XOR al,bl

mov [2002h],al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we applied bit-wise XOR operator on two memory locations.

PROGRAM 3.5

**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],010b

mov al,[4000h]

mov bl,[4000h]

shl al,1

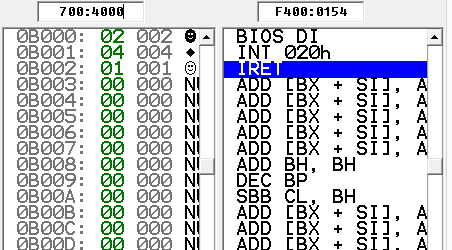
shr bl,1

mov 4001h,al

mov 4002h,bl

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we multiplied and divided a number by 2 stored in a memory location by using arithmetic shift operations.

PROGRAM 3.6

**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],00010111b

mov [4002h],00010011b

mov al,[4001h]

mov bl,[4002h]

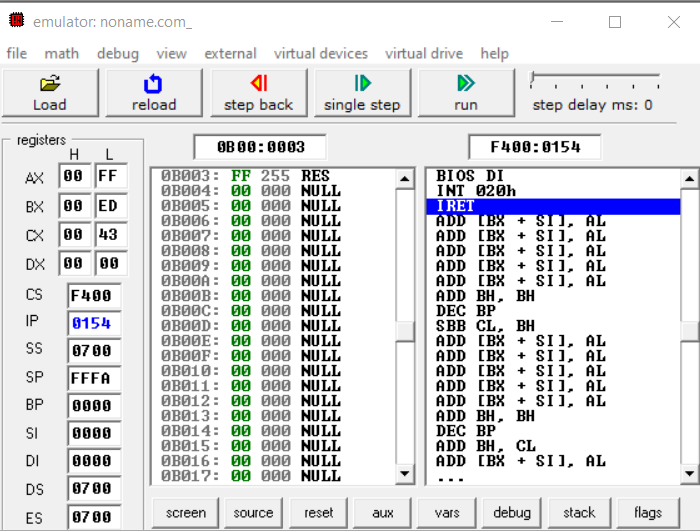
NEG bl

OR al,bl

mov [4003h],al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned to subtract of one memory location from other memory location without the use of SUB operation, using bit-wise operators.

PROGRAM 3.7

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

**CODE:**

org 100h

mov [4001h],11001101b

mov al,[4001h]

mov cl,al

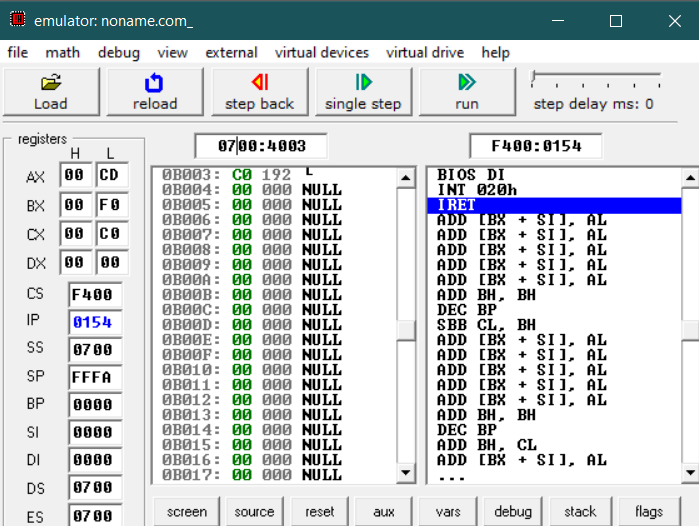
mov bl,11110000b

AND cl,bl

mov [4003h],cl

ret

**OUTPUT:**



**CONCLUSION:**

In this program we learned to mask the lower four bits of a memory location using AND bit-wise operator.

PROGRAM 3.8

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

**CODE:**

org 100h

mov [4001h],11001101b

mov al,[4001h]

mov cl,al

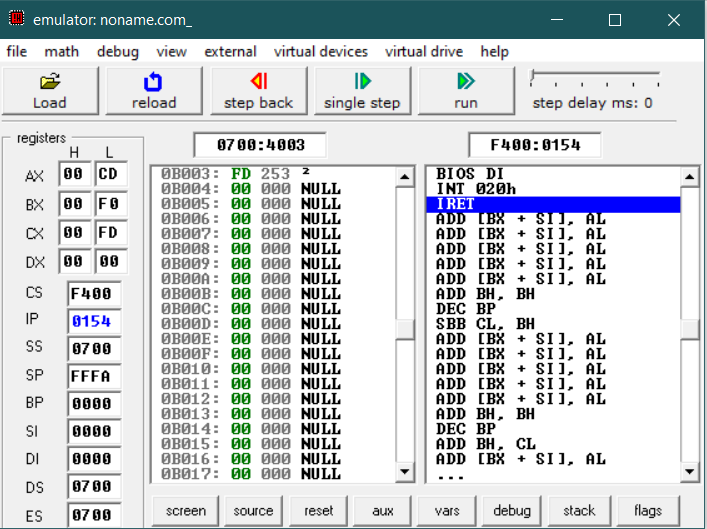
mov bl,11110000b

OR cl,bl

mov [4003h],cl

ret

**OUTPUT:**



**CONCLUSION:**

In this program we learned to set the first four bits of a memory location to 1 using OR bit-wise operator

PROGRAM 3.9

**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 AX, 200h

mov DS, AX

mov 4000h,1h

mov 4001h,2h

mov 4002h,3h

mov 4003h,4h

mov 4004h,5h

mov SI, 4000h

mov DL, [SI]

mov CL, 5h

L1:

inc SI

mov Bl, [SI]

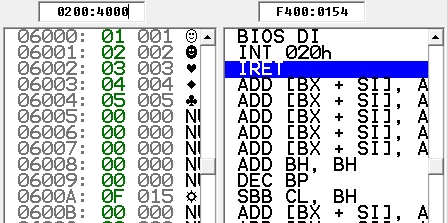
add Dl, BL

loop L1

mov 400Ah, DL

ret

**OUTPUT:**



**CONCLUSION:**

In this program we added a list of numbers stored in continuous memory location using the loop function

PROGRAM 3.10

**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], 15h

MOV [4001h], 20h

MOV [4002h], 35h

MOV [4003h], 0F0h

MOV [4004h], 0FH

MOV CL, 5

MOV SI, 4000h

MOV AL, 0h

LABEL1:

ADC AL,[SI]

JC LABEL2

INC SI

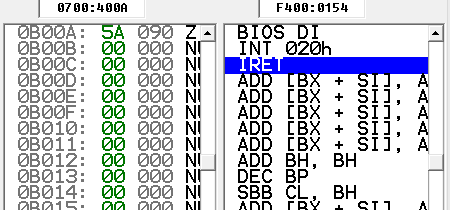
LOOP LA

LABEL2:

MOV [400Ah], AL

ret

**OUTPUT:**



**CONCLUSION:**

In this the program we add a feature to the last practical in which if carry is generated we would halt the program and store the result at 400AH location.

PROGRAM 3.11

**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], 8h

MOV [4006h], 4h

MOV CL, [4006h]

MOV AX, 00h

LABEL1:

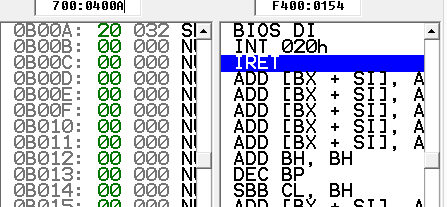
ADC AX, [4001h]

LOOP LABEL1

MOV [400Ah], AX

ret

**OUTPUT:**



**CONCLUSION:**

In this program we multiplied two numbers stored at particular location by repeated addition using for loop.

PROGRAM 3.12

**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], 15h

MOV [4001h], 20h

MOV [4002h], 35h

MOV [4003h], 40h

MOV [4004h], 0FH

MOV CL, 5

MOV DL, CL

MOV SI, 4000h

MOV AL, 0h

LABEL1:

ADC AL,[SI]

INC SI

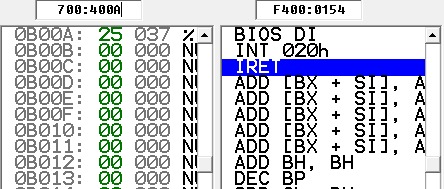
LOOP LABEL1

DIV DL

MOV [400Ah], AL

Ret

**OUTPUT:**



**CONCLUSION:**

In this program we learned how we can find out an average of ‘n’ numbers

PROGRAM 3.13

**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], 15h

MOV [4001h], 20h

MOV [4002h], 35h

MOV [4003h], 40h

MOV [4004h], 0FH

MOV CL, 5

MOV SI, 4000h

LABEL1:

MOV DL, [SI]

SHR DL, 1

JC ODD

JNC EVEN

BACK:

INC SI

LOOP LABEL1

JMP FINISH

ODD:

INC [400Ah]

JMP BACK

EVEN:

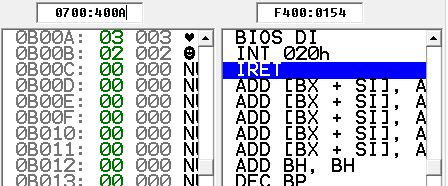
INC [400Bh]

JMP BACK

FINISH:

Ret

**OUTPUT:**



**CONCLUSION:**

In this program we learned how to find how many numbers of odd numbers and even numbers are present in given array of n numbers.

**PRACTICAL – 4**

PROGRAM 4.1

**AIM : Divide 8-bit number stored in memory locations 4009h by data stored at memory location 4001h & store the result of division at memory location 400Ah.**

**CODE:**

org 100h

mov 4009h,42

mov 4001h,5

mov al,[4009h]

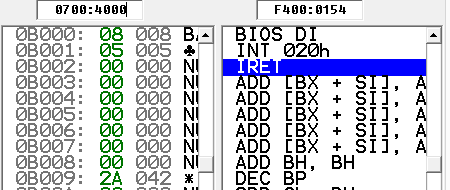
mov bl,[4001h]

div bl

mov 4000h,al

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we applied bit-wise OR operator to perform a selective set operation.

PROGRAM 4.2

**AIM : Divide 8-bit number stored in memory locations 4009h by data stored at memory location 4001h & store the result of module operation at memory location 400Ah.**

**CODE:**

org 100h

mov 4009h,42

mov 4001h,5

mov al,[4009h]

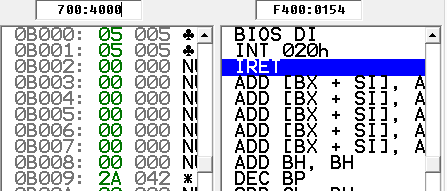
mov bl,[4001h]

div bl

mov 4000h,bl

ret

**OUTPUT:**



**CONCLUSION:**

In this program we learned how to divide data stored at one memory location from the data stored at another memory location and stored module operation result at specific location by using a specific dataset.

PROGRAM 4.3

**AIM : Write an assembly language program to find the largest number in an array.**

**CODE:**

org 100h

MOV [4000H], 50H

MOV [4001H], 15H

MOV [4002H], 30H

MOV [4003H], 72H

MOV [4004H], 14H

MOV [4005H], 78H

MOV [4006H], 39H

MOV CX, 7H

MOV SI, 4000H

MOV [400AH], 0H

LABEL1:

MOV AL, [SI]

CMP [400AH], AL

JC CHANGE

BACK:

INC SI

LOOP LABEL1

JMP FINISH

CHANGE:

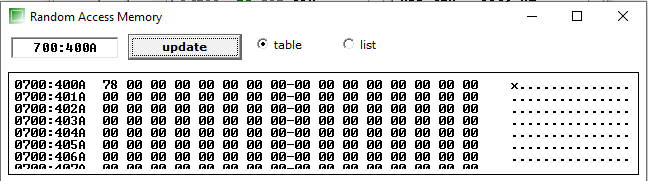
MOV [400AH], AL

JMP BACK

FINISH:

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we found the largest number in array using assembly programming.

PROGRAM 4.4

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

**CODE:**

mov [4001h], 3h

mov [4002h], -1h

mov [4003h], 4h

mov [4004h], -5h

mov [4005h], 2h

mov DL, 0h

mov AX, 200h

mov DS, AX

mov SI, 4001h

mov DI, 400Ah

mov CL, 5h

Label:

mov AL, [SI]

CMP AL, 0

JGE L1:

INC SI

DEC CL

Loop Label:

L1:

INC DL

INC SI

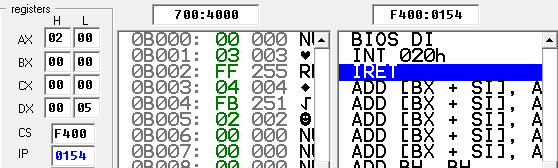
DEC CL

JNZ Label:

mov DH, 5h

SUB DH, DL

**OUTPUT:**



**CONCLUSION:**

In this program we learned to find the number of positive and negative numbers in an array by using counters and conditional statements.

PROGRAM 4.5

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

.

**CODE:**

org 100h

MOV AX, 4362H

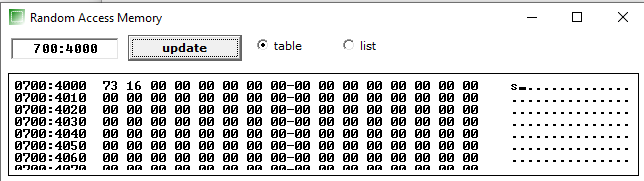
MOV BX, 1673H

MUL BX

MOV [4000H], BX

ret

**OUTPUT:**



**CONCLUSION:**

In this practical we learned to multiply 2 16bit number and stored in it memory.

PROGRAM 4.6

**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.**

.

**CODE:**

org 100h

#start=led\_display.exe#

name "led"

MOV AX, 20H

LABEL0:

MOV CX, 0005H

LABEL1:

MOV DX, 0010H

LABEL2:

DEC DX

JNZ LABEL2

DEC CX

JNZ LABEL1

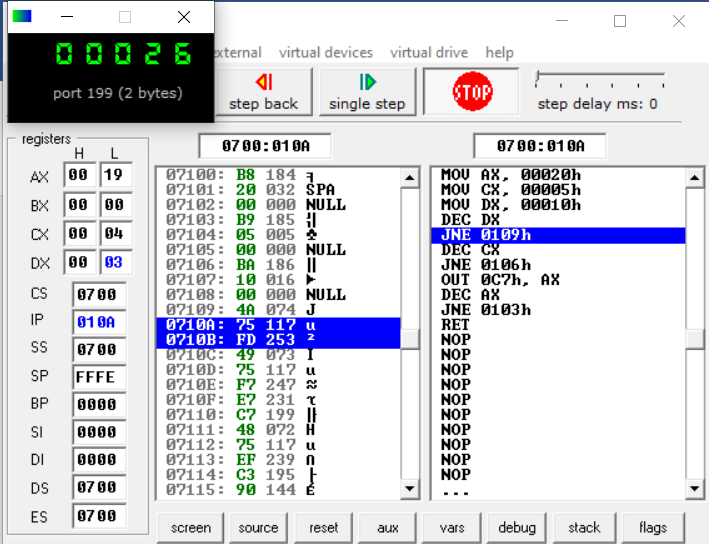
OUT 199, AX

DEC AX

JNZ LABEL0

ret

**OUTPUT:**



**CONCLUSION:**

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

PROGRAM 4.7

**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**

**CODE:**

org 100h

mov ax, 3

int 10h

mov ax, 1003h

mov bx, 0

int 10h

mov ax, 0b800h

mov ds, ax

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'

mov cx, 19

mov di, 03h

c: mov [di], 00001111b

add di, 2

mov bx, 0020h

back:

dec bx

jnz back

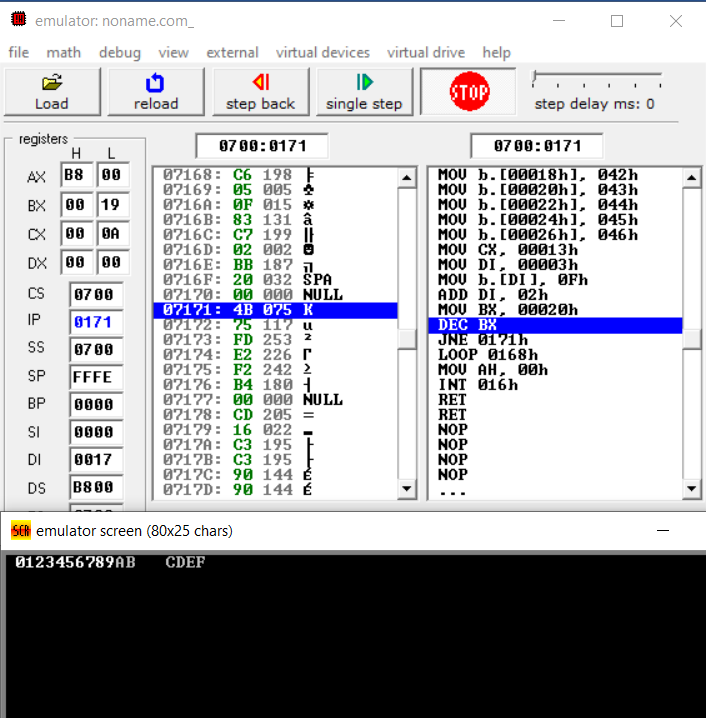
loop c

mov ah, 0

int 16h

ret

**OUTPUT:**



**CONCLUSION:**

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

PROGRAM 4.8

**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.**

.

**CODE:**

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

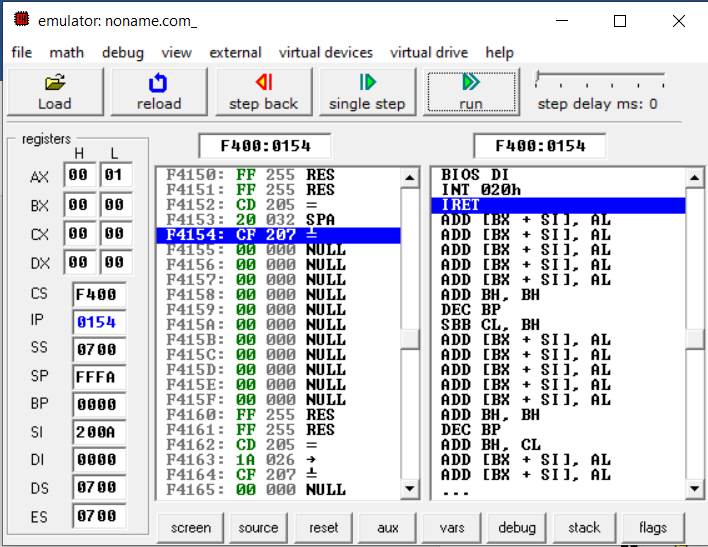
OUT 01H, AX

JMP BACK

FINISH:

Ret

**OUTPUT:**



**CONCLUSION:**

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

**PRACTICAL – 5**

PROGRAM 5.1

**AIM : Write a program which sets the parity bit..**

**CODE:**

org 100h

PUSHF

POP DX

OR DX, 100B

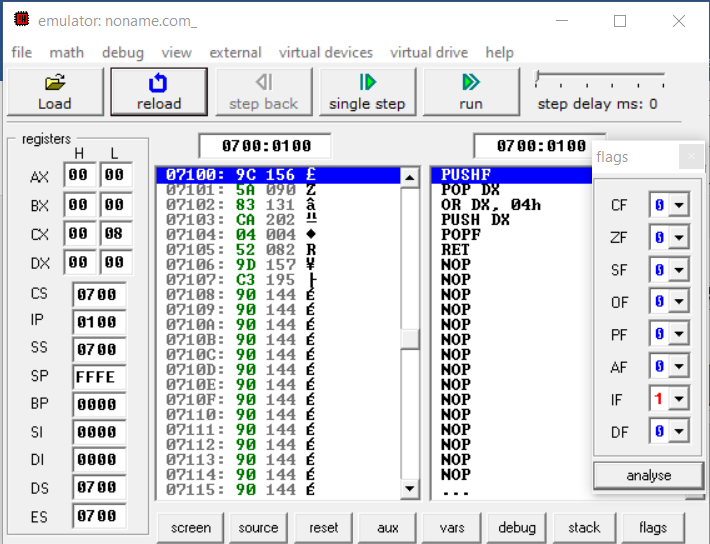
PUSH DX

POPF

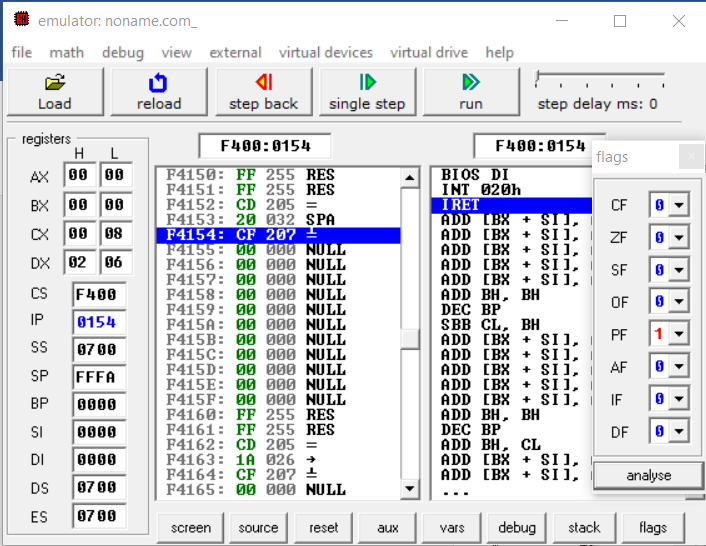
Ret

**OUTPUT:**

Before



After



**CONCLUSION:**

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

PROGRAM 5.2

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

**CODE:**

org 100h

PUSHF

POP DX

OR DX, 100B

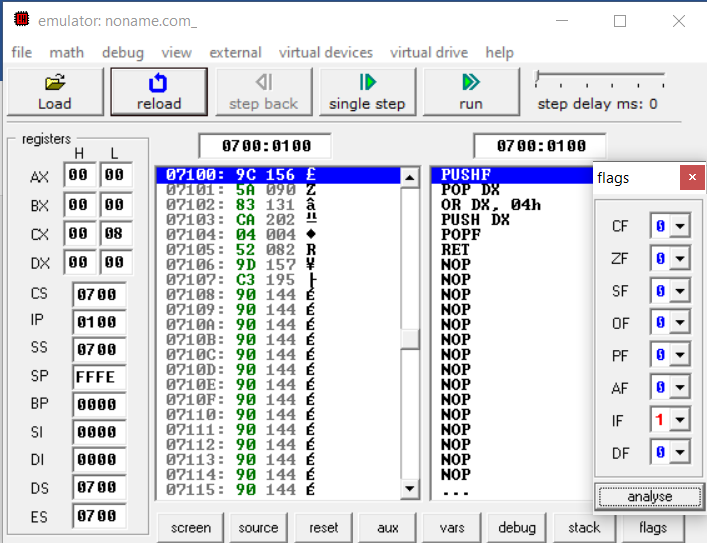
PUSH DX

POPF

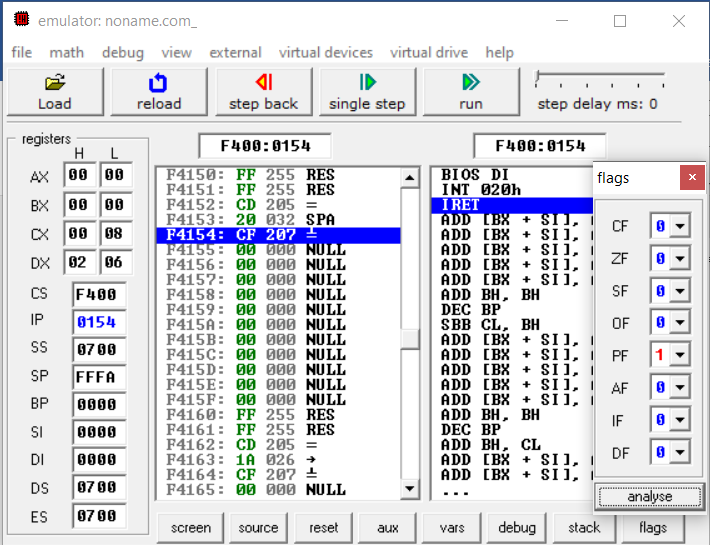
ret

**OUTPUT:**

Before



After



**CONCLUSION:**

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

PROGRAM 5.3

**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.**

.

**CODE:**

org 100h

MOV AX, 7AH

MOV BX, 46H

ADC AX, BX

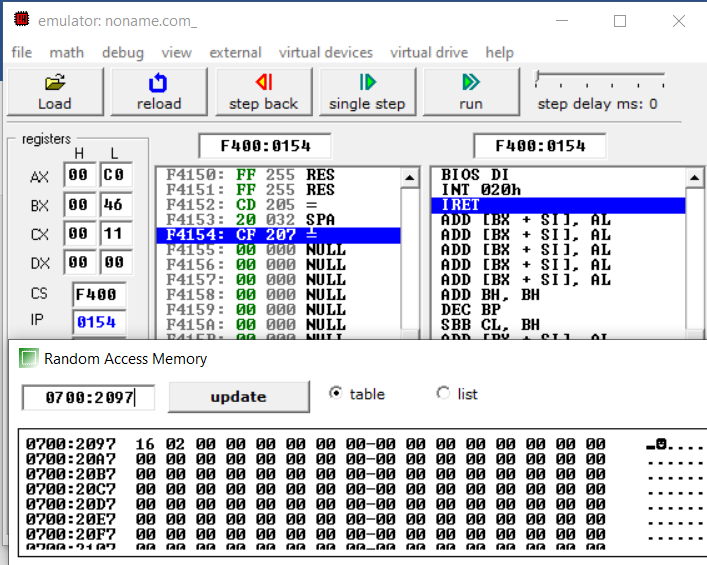
MOV [2098H], AX

PUSHF

POP [2097H]

ret

**OUTPUT:**



**CONCLUSION:**

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

PROGRAM 5.4

**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..**

**CODE:**

org 100h

CMP AX, 0000H

MOV CX, 208D

LABEL1:

DEC CX

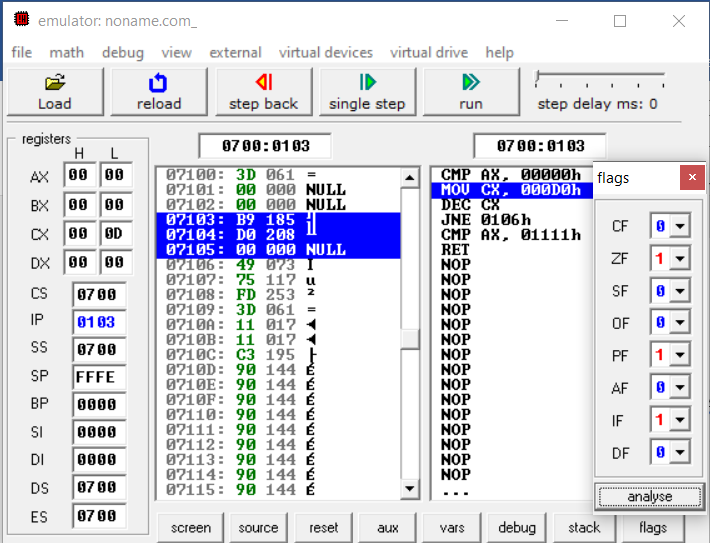
JNZ LABEL1

CMP AX, 1111H

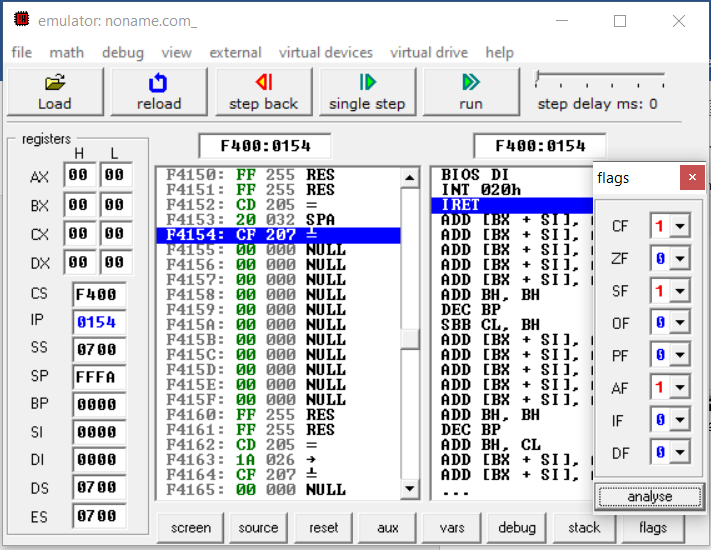
ret

**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.

PROGRAM 5.5

**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.**

.

**CODE:**

org 100h

MOV DX, 40H

PUSH DX

POPF

call p1

CMP CX, 1111H

ret

p1 PROC

MOV AX, 10H

MOV BX, 0F0H

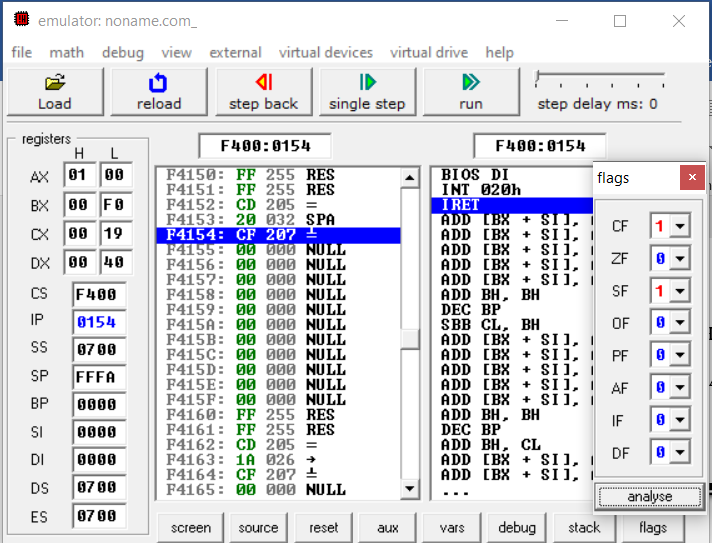
ADC AX, BX

MOV [2040H], AX

RET

p1 ENDP

**OUTPUT:**



**CONCLUSION:**

In this practical, we implemented addition of two hex numbers and cleared zero flag after the subroutine.

PROGRAM 5.6

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

.

**CODE:**

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

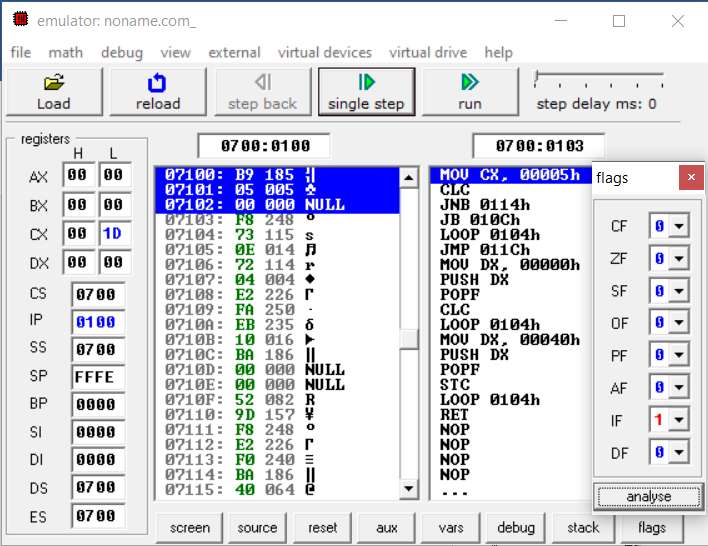
LOOP LABEl1

FINISH:

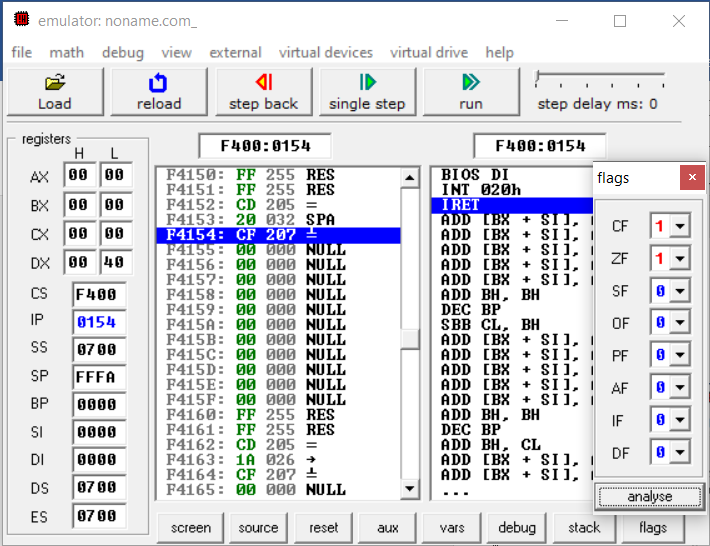
ret

**OUTPUT:**

Zero Flag at odd iterations



Clear Zero Flag at even iterations



**CONCLUSION:**

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

PROGRAM 5.7

**AIM : Write a program to provide the given on/off time of three traffic lights. 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).**

**CODE:**

#start=Traffic\_Lights.exe#

name "traffic"

mov ax, all\_red

out 4, ax

mov si, offset situation

next:

mov ax, [si]

out 4, ax

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

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.

PROGRAM 5.8

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

**CODE:**

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

L1:

PUSH [3000H] DI

INC DI

LOOP L1

MOV DI, 0H

MOV CX, 5H

L2:

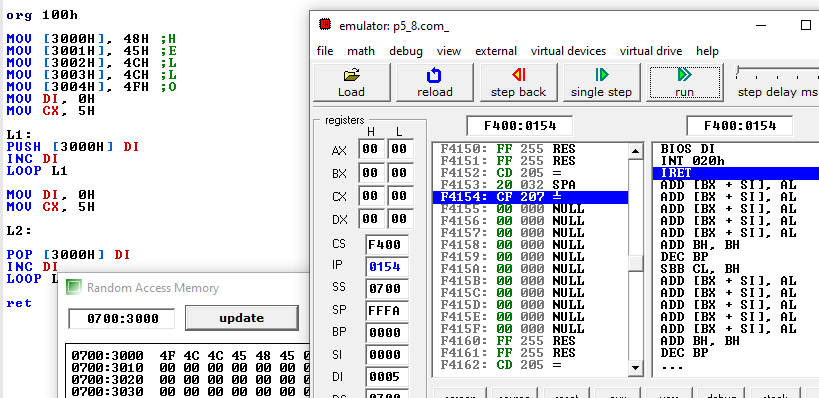
POP [3000H] DI

INC DI

LOOP L2

Ret

**OUTPUT:**



**CONCLUSION:**

In this practical, we reverse the given string and store at the same location using stack operation i.e. push and pop.

PROGRAM 5.9

**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.**

**CODE:**

org 100h

mov [2200h],5 mov [2201h],1234h mov [2203h],1235h mov [2205h],13h

mov si,2200h mov cl,[si]

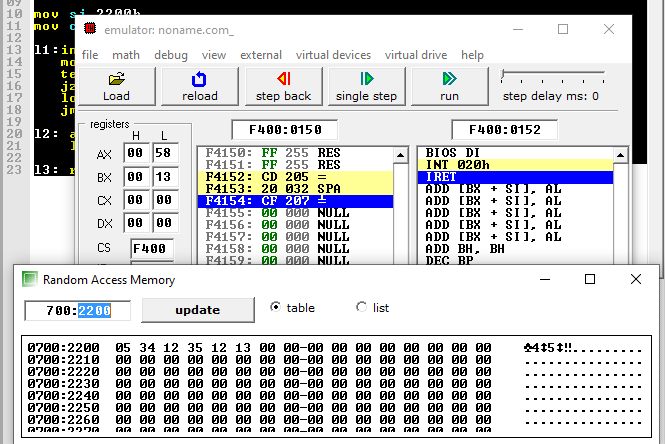
l1:inc si mov bl,[si]

test bl,1 ;LSB=0 then ZF=1 jz l2

loop l1 jmp l3

l2: add al,bl loop l1

l3: ret

**OUTPUT:**

**CONCLUSION:**

In this practical, we calculated the sum of the even series assuming the 8 bit number

PROGRAM 5.10

**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.**

**CODE:**

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

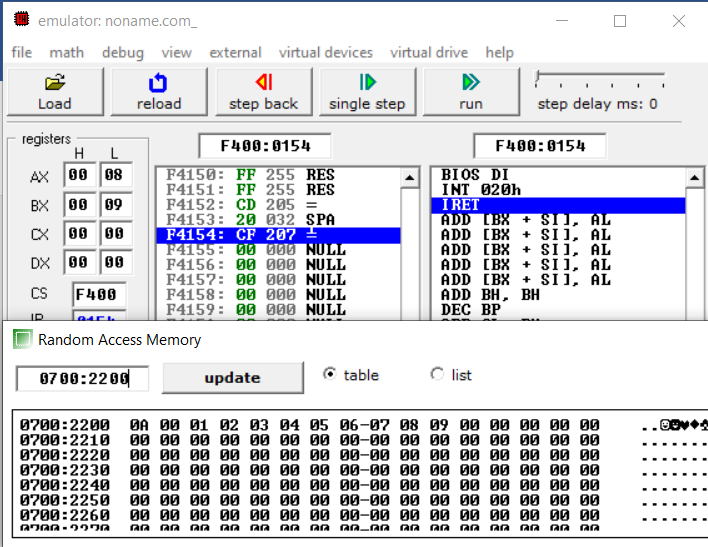
MOV [SI+1], AL

JMP BACK

FINISH:

ret

**OUTPUT:**



**CONCLUSION:**

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

PROGRAM 5.11

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

**CODE:**

org 100h

mov CL,09

mov SI,3000h

mov AL,00

mov [SI],AL

inc SI

mov BL,01

add [SI],BL

L1: add AL,BL

mov DL,AL

inc SI

mov [SI],DL

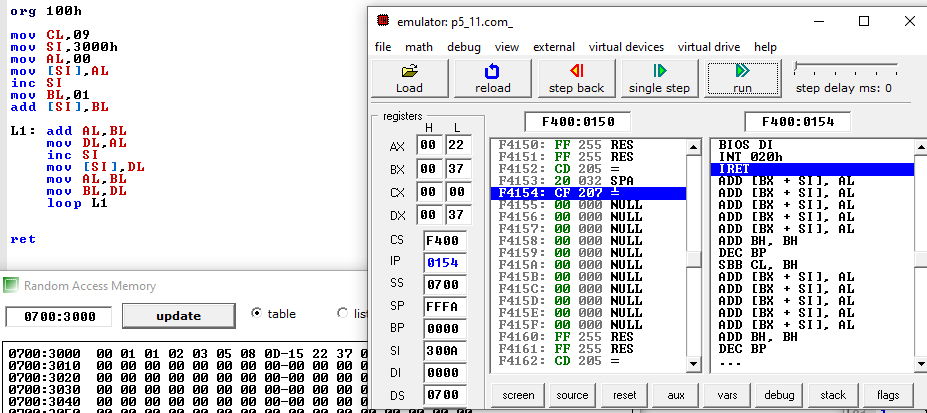
mov AL,BL

mov BL,DL

loop L1

ret

**OUTPUT:**



**CONCLUSION:**

In this practical, we write a program for Fibonacci series in assembly language.

**PRACTICAL – 6**

**Aim – (A) Exploiting multiple core for compilation of various applications.**

A multi-core processor is an integrated circuit to which two or more processors have been attached for enhanced performance, reduced power consumption and efficient simultaneous processing of multiple tasks. A dual core set-up is somewhat comparable to having multiple, separate processors installed in the same computer, but because the two processors are actually plugged into the same socket, the connection between them is faster. Ideally, a dual core or a quad core processor is nearly twice as powerful as a single core processor. In practice, performance gains are said to be about fifty percent. A dual core processor is likely to be about one-and-a-half times as powerful as a single core processor. Multi-core processing is a growing industry trend as single core processors rapidly reach the physical limits of possible complexity and speed. Companies that have produced or are working on multi-core products include AMD, ARM, Broadcom, Intel, and VIA.

For the most part, having a higher core count processor is generally better if your software and typical use cases support it. For the most part, a dual-core or quad-core processor will be more than enough power for a basic computer user. The majority of consumers will see no tangible benefits from going beyond four processor cores as there is so little non-specialized software that can take advantage of it. The best use case for high-core-count processors relates to machines that perform complex tasks such as desktop video editing, some forms of high-end gaming, or complicated science and math programs.

**CONCLUSION**

**In this practical we learned about exploiting multiple core for compilation of various applications.**

**Aim – (B) Affinity allocation: Windows and Linux Machines**

Affinity allocation in Linux

The ability in Linux to bind one or more processes to one or more processors, called CPU affinity, is a long-requested feature. The idea is to say “always run this process on processor one” or “run these processes on all processors but processor zero”. The scheduler then obeys the order, and the process runs only on the allowed processors.

Other operating systems, such as Windows NT, have long provided a system call to set the CPU affinity for a process. Consequently, demand for such a system call in Linux has been high. Finally, the 2.5 kernel introduced a set of system calls for setting and retrieving the CPU affinity of a process.

In this article, I look at the reasons for introducing a CPU affinity interface to Linux. I then cover how to use the interface in your programs. If you are not a programmer or if you have an existing program you are unable to modify, I cover a simple utility for changing the affinity of a given process using its PID. Finally, we look at the actual implementation of the system call.

Soft vs. Hard CPU Affinity

There are two types of CPU affinity. The first, soft affinity, also called natural affinity, is the tendency of a scheduler to try to keep processes on the same CPU as long as possible. It is merely an attempt; if it is ever infeasible, the processes certainly will migrate to another processor. The new O(1) scheduler in 2.5 exhibits excellent natural affinity. On the opposite end, however, is the 2.4 scheduler, which has poor CPU affinity. This behavior results in the ping-pong effect. The scheduler bounces processes between multiple processors each time they are scheduled and rescheduled.

Affinity Allocation in windows

**Processor** **affinity**, or **CPU** pinning or "cache affinity", enables the binding and unbinding of a [process](https://en.wikipedia.org/wiki/Process_(computing)) or a [thread](https://en.wikipedia.org/wiki/Thread_(computing)) to a [central processing unit](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Scheduling_algorithm) in a [symmetric multiprocessing](https://en.wikipedia.org/wiki/Symmetric_multiprocessing) operating system. Each item in the queue has a tag indicating its kin [processor](https://en.wikipedia.org/wiki/Central_processing_unit). At the time of resource allocation, each task is allocated to its kin processor in preference to others.

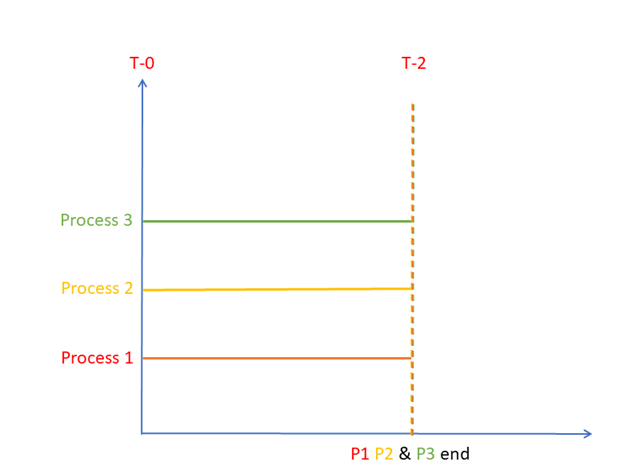
Processor affinity takes advantage of the fact that remnants of a process that was run on a given processor may remain in that processor's state (for example, data in the [cache memory](https://en.wikipedia.org/wiki/CPU_cache)) after another process was run on that processor. Scheduling that process to execute on the same processor improves its performance by reducing performance-degrading events such as [cache misses](https://en.wikipedia.org/wiki/Cache_miss). A practical example of processor affinity is executing multiple instances of a non-threaded application, such as some graphics-rendering software.

**CONCLUSION**

**In this practical we got to know about the affinity allocation: Windows and Linux Machines**

**Aim – (C) Shifting all processes on single core.**

In parallel execution, the tasks to be performed by a process are broken down into sub-parts, and multiple CPUs (or multiple cores) process each sub-task at precisely the same time.

[](https://www.guru99.com/images/1/082319_0623_CPUCoreMult2.png)

**Parallel Execution**

As you can see, at any given time, all processes are in execution. In reality, it is the sub-tasks of a process which are executing in parallel, but for better understanding, you can visualize them as processes.

Therefore, parallelism is the real way in which multiple tasks can be processed at the same time. This type of situation can be found in systems having multicore processors, which includes almost all modern, commercial processors.

1. Mostly can’t be single-core processors anymore. All my desktop computers have multi-core processors, oh, so does my phone and tablet. And the single-board Orange Pi computers I have (like raspberry Pi, but better for the same price). The only single core “thing” I have is a esp2866 development board, and even that has a second chip on the same board for some other functions. Actually, I have an old Pentium 4 in the cupboard that is broken. that’s a single core system. But the Pi’s are faster than it anyway, cheaper and smaller and use less power.
2. It used to be that some single-processor systems where better at single-thread workloads than multi-processor systems were at those workloads. So if that’s the software / work you’re doing, then single processor was the best choice.
3. For every other reason multi-processor systems are better. They’re more efficient (flops/watt) provided that the software can use multiple processors. And these days, even if it can’t it’s equally good as a single processor system.

**CONCLUSION**

**In this practical we learnt that how to shift all processes on single core.**

**PRACTICAL – 7**

**Aim - Intel and AMD: Architectural Differences**

**Processing Power - Intel processors, on average, are far more powerful than AMD processors. This doesn’t necessarily mean you will need the Intel though. For a gamer, the AMD is plenty powerful enough. When I upgraded from my AMD 8150 to my i5–6500, I only saw a 2–3 FPS increase in most games, even though on paper the i5–6500 is about 50% more powerful. Where I see the increase is in boot time, program loading time, and intense engineering tools such as Solidworks and Catia. I don’t know much about the new Ryzen line of AMD processors, but they are supposed to pose some serious competition to Intel, and it’s about time. When I bought my i5–6500, the newest AMD processor was still running on technology 4 years older than the Intel.**

**Power Consumption - While this has never been something I’ve cared about too awful much, the Intel processors, on average consume far less electrical power. This allows you to buy cheaper PSU’s for the same reliability, and potentially spend less on your power bill.**

**Integrated Graphics - Again, something I don’t care about, but if you don’t want to buy an expensive dedicated graphics card, Intel is your only option here. While the Intel HD line of video processors are not sufficient for serious gamers, it’s actually extremely powerful compared to integrated graphics of the past, and can run many modern games on low settings. I’ve used my Surface Pro 2 and 4 successfully with Solid works.**

**Price - Traditionally, AMD processors were far cheaper than Intel, but that gap is closing. When I upgraded to the i5–6500, it was still a good bit more expensive than the AMD counterpart. But looking on Amazon now, the i7–7700k is $348.88, whereas it’s closest AMD competition, the Ryzen 7 1700 is $329.99. That’s not an appreciable difference for anyone that really cares about building a PC with the performance that these processors would provide.**

**CONCLUSION**

**In this practical we got to know about the architectural differences between Intel and AMD**

**PRACTICAL – 8**

**Aim – (A) Multicore Programming in Linux**

**Multicore programming in Linux**

**Multicore Application Programming is a comprehensive, practical guide to high-performance multicore programming that any experienced developer can use.**

**Author Darryl Gove covers the leading approaches to parallelization on Windows, Linux, and Oracle Solaris. Through practical examples, he illuminates the challenges involved in writing applications that fully utilize multicore processors, helping you produce applications that are functionally correct, offer superior performance, and scale well to eight cores, sixteen cores, and beyond.**

**The book reveals how specific hardware implementations impact application performance and shows how to avoid common pitfalls. Step by step, you’ll write applications that can handle large numbers of parallel threads, and you’ll master advanced parallelization techniques.**

**CONCLUSION**

**In this practical we learnt about what exactly is Multicore Programming in Linux**

**Aim – (B) Multicore Programming in Window**

**Multicore programming in Windows**

* **Identify your best opportunities to use parallelism**
* **Share data safely between multiple threads**
* **Write applications using POSIX or Windows threads**
* **Hand-code synchronization and sharing**
* **Take advantage of automatic parallelization and OpenMP**
* **Overcome common obstacles to scaling**
* **Apply new approaches to writing correct, fast, scalable parallel code**

**Multicore Application Programming isn’t wedded to a single approach or platform: It is for every experienced C programmer working with any contemporary multicore processor in any leading operating system environment.**

**CONCLUSION**

**In this practical we learnt about what exactly is Multicore Programming in Windows**

**PRACTICAL – 9**

**Aim - Hardware I/O and Pen drive I/O Performance Measurement**

**I/O performance has measures that have no counterparts in CPU design. One of**

**these is diversity: Which I/O devices can connect to the computer system? An-**

**other is capacity: How many I/O devices can connect to a computer system?**

**In addition to these unique measures, the traditional measures of performance,**

**namely response time and throughput, also apply to I/O. (I/O throughput is some-**

**times called I/O bandwidth, and response time is sometimes called latency.)**

**Another measure of I/O performance is the interference of I/O with CPU execution. Transferring data may interfere with the execution of another process. There is also overhead due to handling I/O interrupts. Our concern here is how much longer a process will take because of I/O for another process.**

**Throughput versus Response Time**

**An interaction, or transaction, with a computer is divided into three parts:**

* **Entry time—The time for the user to enter the command. The graphics system in Figure 7.26 required 0.25 seconds on average to enter a command versus 4.0 seconds for the keyboard system.**
* **System response time—The time between when the user enters the command and the complete response is displayed.**
* **Think time—The time from the reception of the response until the user begins to enter the next command.**

**The sum of these three parts is called the transaction time. Several studies report**

**that user productivity is inversely proportional to transaction time; transactions**

**per hour are a measure of the work completed per hour by the user.**

**CONCLUSION**

**In this practical we learned about Performance Measurements of Hardware I/O and Pen drive I/O**

**PRACTICAL – 10**

**Aim - L1, L2 & L3 Cache Rebuilt and performance measurement**

**Cache memory, also called CPU memory, is high-speed static random access memory (**[**SRAM**](https://whatis.techtarget.com/definition/SRAM-static-random-access-memory)**) that a computer microprocessor can access more quickly than it can access regular random access memory (**[**RAM**](https://searchstorage.techtarget.com/definition/RAM-random-access-memory)**). This memory is typically integrated directly into the CPU chip or placed on a separate chip that has a separate**[**bus**](https://searchstorage.techtarget.com/definition/bus)**interconnect with the CPU. The purpose of cache memory is to store program instructions and data that are used repeatedly in the operation of programs or information that the CPU is likely to need next. The computer processor can access this information quickly from the cache rather than having to get it from computer's main memory. Fast access to these instructions increases the overall speed of the program.**

* **A CPU cache is a smaller faster memory used by the central processing unit (CPU) of a computer to reduce the average time to access memory.**
* **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, it’s still faster than the RAM.**

**CONCLUSION**

**In this practical we learned about L1, L2 & L3 Cache Rebuilt and performance measurement**