Visvesvaraya National Institute of Technology (VNIT), Nagpur

Advanced microprocessors and interfacing lab (ECP426)

Lab Journal

Submitted by :
Shruti Murarka (BT18ECE099)
Semester VII

Submitted to :
Dr. Neha K. Nawandar
(Lab Instructor)
Department of Electronics and Communication Engineering,
VNIT Nagpur

Contents

Experiment 1 - Arithmetic operations on 8-bit and 16-bit numbers.

Aim: Write ALP using both simplified and full segment definition to perform Arithmetic Operations(Addition, subtraction, multiplication and division) on 8-bit and 16-bit numbers.

Components/ setup: emu8086 Emulator

Theory: 8086 Microprocessor is an enhanced version of 8085Microprocessor. The 8086 microprocessor supports both 8-bit and 16-bit arithmetic operations. Addition and Subtraction operations can be carried out using the op code **ADD** and **SUB** respectively. Both of these op codes store the final result in AX register. Multiplication and Division operations are carried out using **MUL** and **DIV** instructions, respectively. For 8-bit Multiplication, the result is stored in AX register as the result can be a 16-bit number and lower and higher bytes are stored in AL and AH respectively. Whereas, for 16-bit Multiplication, the result is a 32-bit number and the lower and upper word are stored in AX and DX register respectively. For 8-bit Division, the quotient and remainder are stored in AL and AH respectively. Whereas, for 16-bit Division, the quotient and remainder are both 16-bit numbers and stored in AX and DX register respectively. Results and codes can be seen below.
Also, the following codes are written using both simplified and full segment definitions, both of which are optimal ways to write codes. These formats are preferred as the code and data remain segregated. The main difference between full and simplified segment definitions are that in full segment, the address where the data is stored must be specified, but simplified segment does not require the programmer to do so.

Code:

Listing 1: Addition of 8-bit numbers

```
;Full Segment Definition
assume cs:code,ds:data
data segment
    a db 11h
    b db 88h
    c db ?
data ends
code segment
start:
```

```
    mov ax,data
    mov ds,ax
    mov al,a
    mov bl,b
    add al,bl
    mov c,al
code ends
end start

;Simplified Segment Definition
.model small
.data
    a db 11h
    b db 88h
    c db ?
.code
.startup
    mov al,a
    mov bl,b
    add al,bl
    mov c,al
.exit
end
```

Listing 2: Addition of 16-bit numbers

```
;Full Segment Definition
assume cs:code,ds:data
data segment
    num1 dw 0011h
    num2 dw 3388h
    num3 dw ?
data ends

code segment
start:
    mov ax,data
    mov ds,ax
    mov ax,num1
    mov bx,num2
    add ax,bx
    mov num3,ax
code ends
end start

;Simplified Segment Definition
```

```
.model small
.data
    num1 dw 0011h
    num2 dw 3388h
    num3 dw ?

.code
.startup
    mov ax,num1
    mov bx,num2
    add ax,bx
    mov num3,ax
.exit
end
```

Listing 3: Subtraction of 8-bit numbers

```

;Full Segment Definition
assume cs:code,ds:data
data segment
    a db 99h
    b db 09h
    c db ?
data ends

code segment
start:
    mov ax,data
    mov ds,ax
    mov al,a
    mov bl,b
    sub al,bl
    mov c,al
code ends
end start

;Simplified Segment Definition
.model small
.data
    a db 99h
    b db 09h
    c db ?
.code
.startup
    mov al,a
```

```
    mov bl,b
    sub al,bl
    mov c,al
.exit
end
```

Listing 4: Subtraction of 16-bit numbers

```
;Full Segment Definition
assume cs:code,ds:data
data segment
    num1 dw 9999h
    num2 dw 1111h
    num3 dw ?
data ends

code segment
start:
    mov ax,data
    mov ds,ax
    mov ax,num1
    mov bx,num2
    sub ax,bx
    mov num3,ax
code ends
end start

;Simplified Segment Definition
.model small
.data
    num1 dw 9999h
    num2 dw 1111h
    num3 dw ?
.code
.startup

    mov ax,num1
    mov bx,num2
    sub ax,bx
    mov num3,ax
.exit
end
```

Listing 5: Multiplication of 8-bit numbers

```
;Full Segment Definition
assume cs:code,ds:data
data segment
    a db 11h
    b db 09h
    c db ?
data ends
code segment
start:
    mov ax,data
    mov ds,ax
    mov al,a
    mov bl,b
    mul bl
    mov c,al
code ends
end start

;Simplified Segment Definition
.model small
.data
    a db 11h
    b db 09h
    c db ?
.code
.startup

    mov al,a
    mov bl,b
    mul bl
    mov c,al
.exit
end
```

Listing 6: Multiplication of 16-bit numbers

```
;Full Segment Definition
assume cs:code,ds:data
data segment
    n1 dw 0099h
    n2 dw 0005h
    ans dw ?

data ends
code segment
    begin:
```

```
    mov ax,data
    mov ds,ax
    mov ax,n1
    mov bx,n2
    mul bx
    mov ans,ax
code ends
end begin

;Simplified Segment Definition
.model small
.data
    n1 dw 0099h
    n2 dw 0005h
    ans dw ?

.code
.startup
    mov ax,n1
    mov bx,n2
    mul bx
    mov ans,ax
.exit
end
```

Listing 7: Division of 8-bit numbers

```
;Full Segment Definition
assume cs:code,ds:data
data segment
    a db 99
    b db 8
    quo db ?
    rem db ?
data ends
code segment
start:
    mov ax,data
    mov ds,ax
    mov ah,0h
    mov al,a
    mov bl,b
    div bl
    mov quo,al
    mov rem,ah
code ends
```

```
end start

;Simplified Segment Definition
.model small
.data
    a db 99
    b db 8
    quo db ?
    rem db ?
.code
.startup
    mov ah,0h
    mov al,a
    mov bl,b
    div bl
    mov quo,al
    mov rem,ah
.exit
end
```

Listing 8: Division of 16-bit numbers

```
;Full Segment Definition
assume cs:code,ds:data
data segment
    n1 dw 1899h
    n2 dw 0099h
    Q dw ?
    R dw ?
data ends
code segment
    begin:
    mov ax,data
    mov ds,ax
    mov ax,n1

    mov bx,n2
    div bx
    mov Q,ax
    mov R,dx
code ends
end begin

;Simplified Segment Definition
.model small
.data
```

```
    n1 dw 1899h
    n2 dw 0099h
    Q dw ?
    R dw ?
.code
.startup
    mov ax,n1
    mov bx,n2
    div bx
    mov Q,ax
    mov R,dx
.exit
end
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

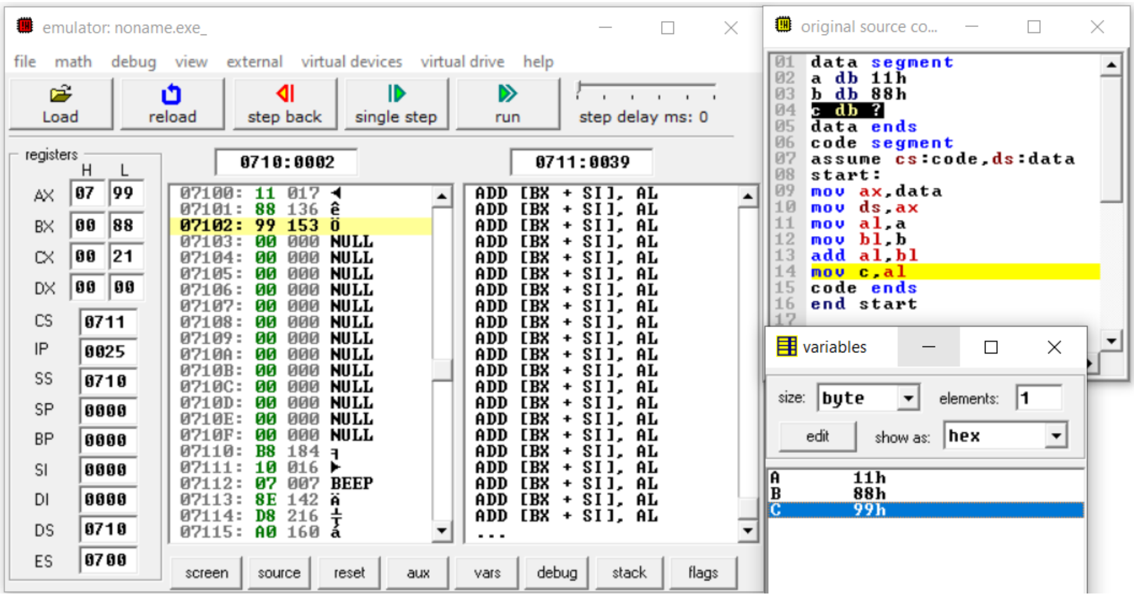

Figure 1: Addition of 8-bit numbers.

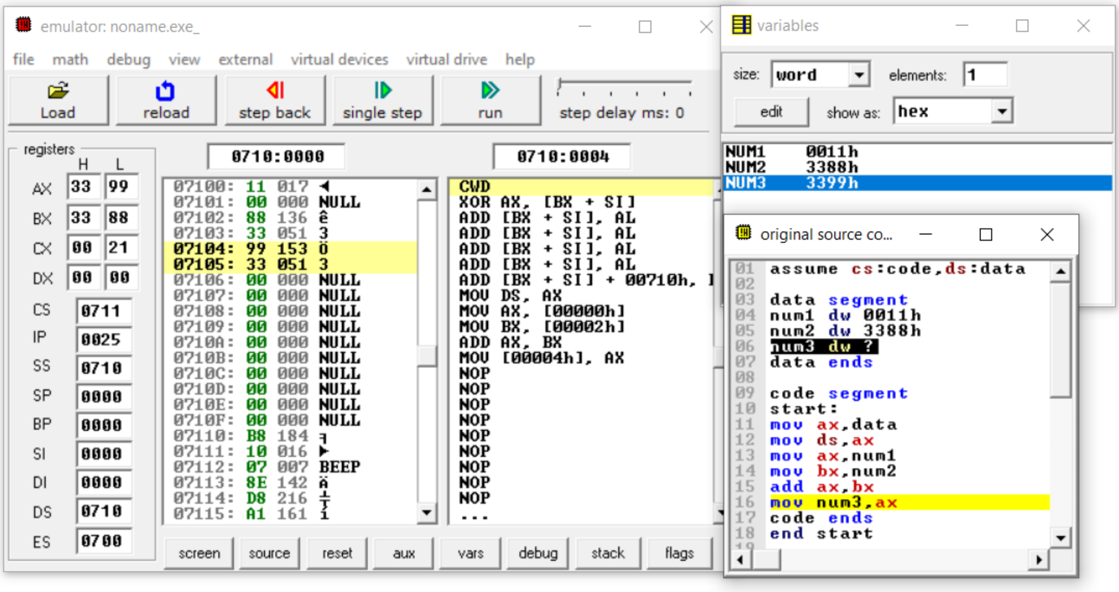

Figure 2: Addition of 16-bit numbers.

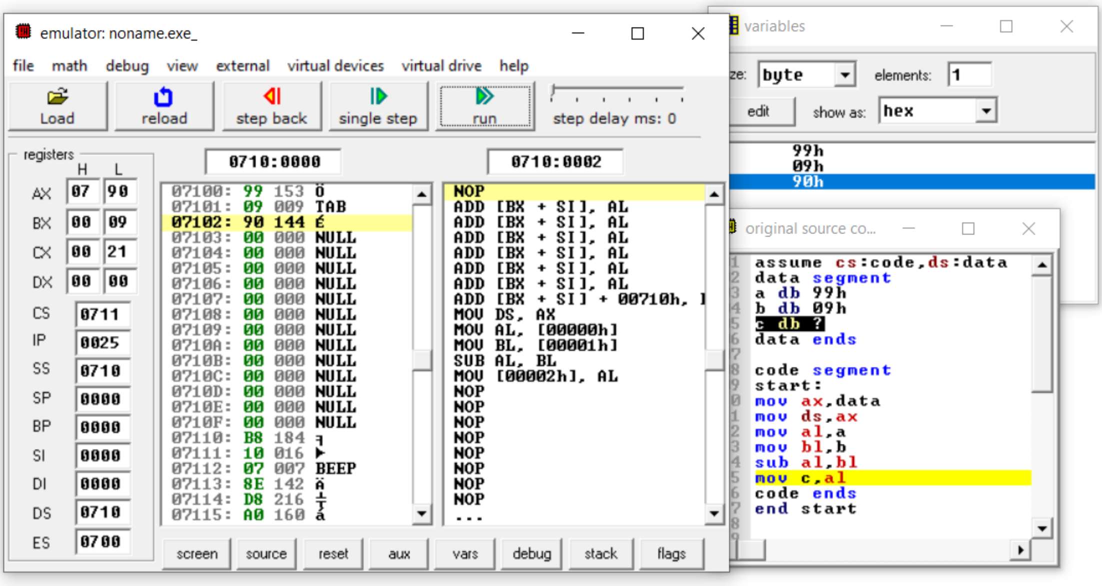

Figure 3: Subtraction of 8-bit numbers.

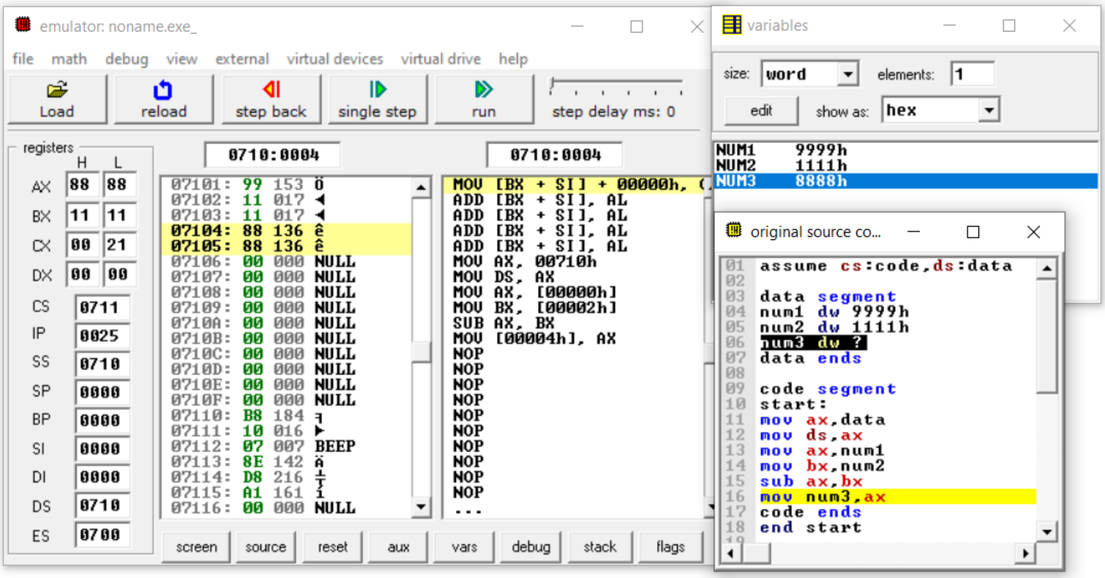

Figure 4: Subtraction of 16-bit numbers.

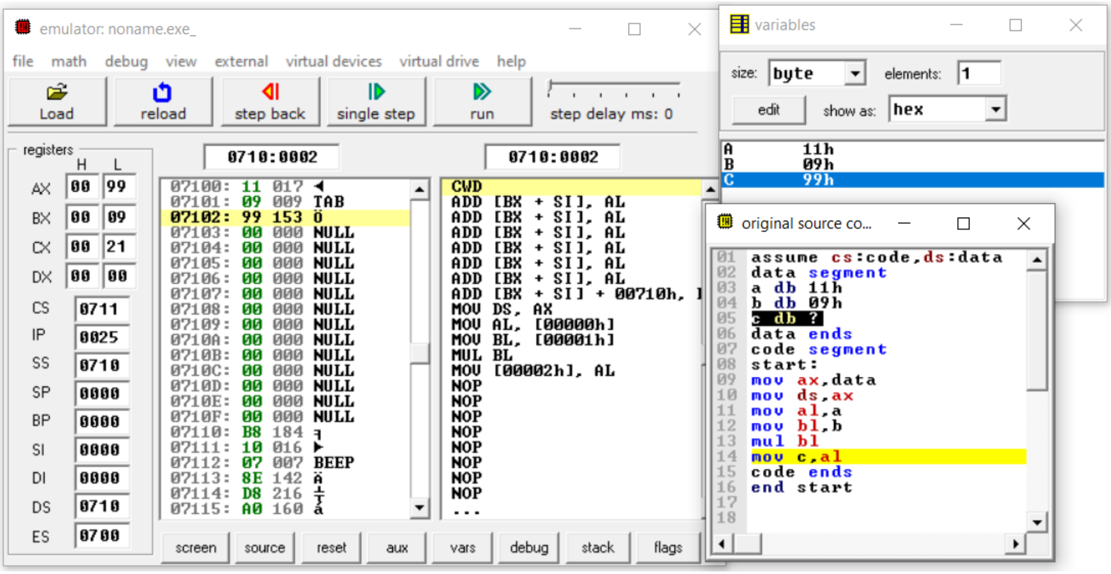

Figure 5: Multiplication of 8-bit numbers.

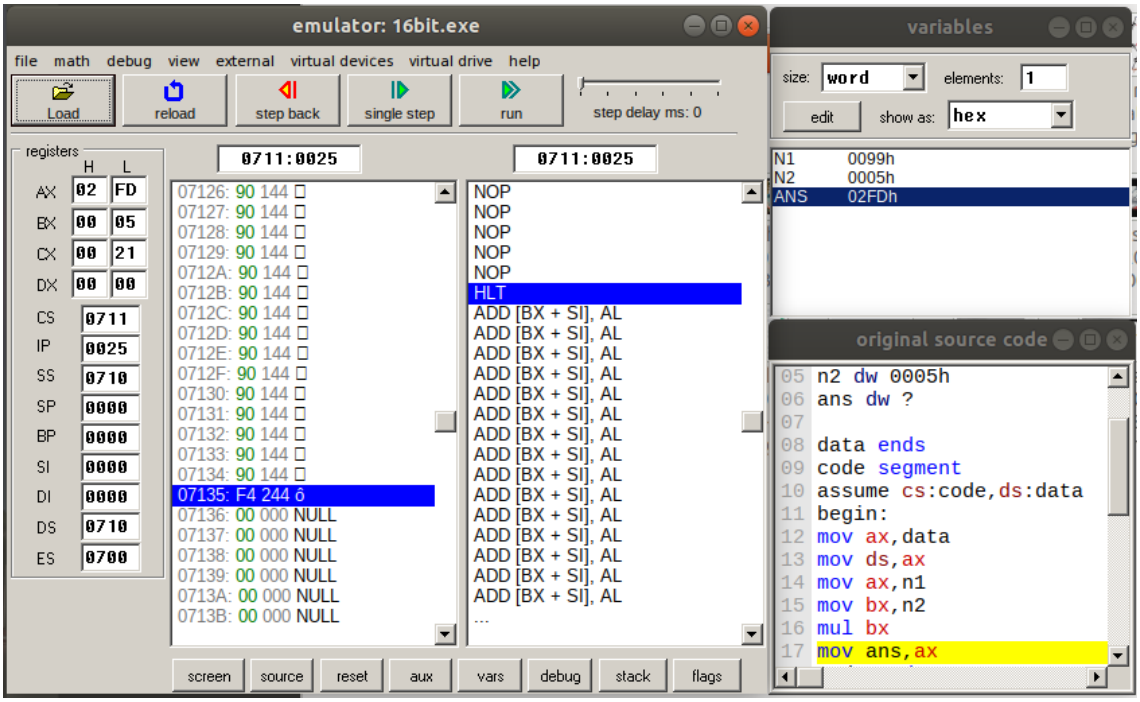

Figure 6: Multiplication of 16-bit numbers.

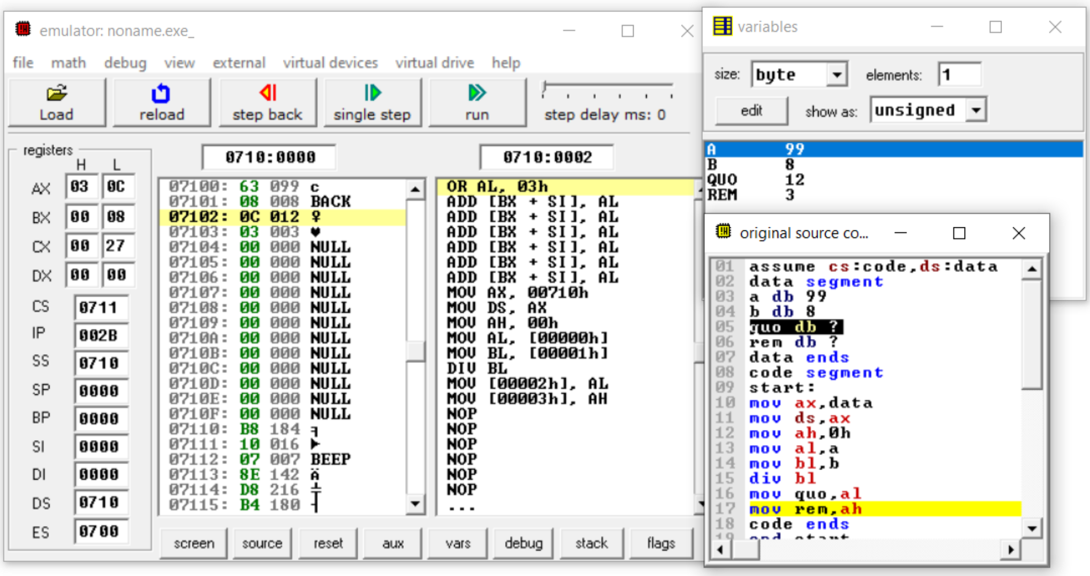

Figure 7: Division of 8-bit numbers.

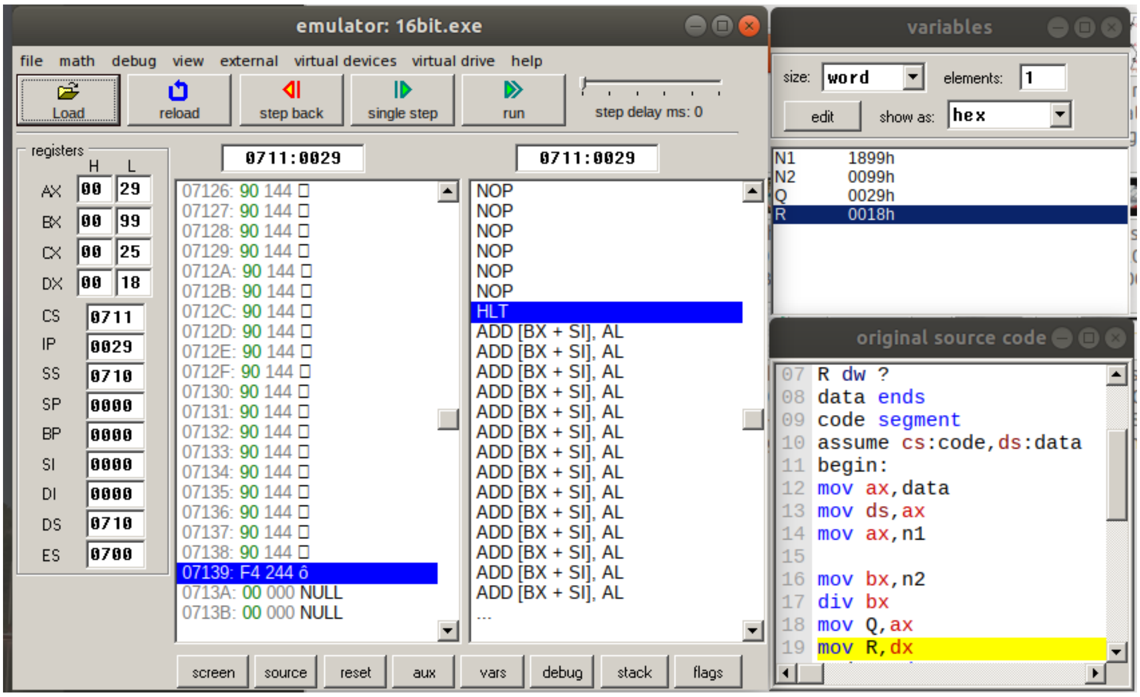

Figure 8: Division of 16-bit numbers.

Conclusion: The Arithmetic Operations of addition, subtraction, multiplication and division for both 8-bit and 16-bit numbers was successfully performed using x86 instructions on the emu8086 emulator.

Note: Either input or output of above codes is 99, which is my roll no.(BT18ECE099)

Experiment 2 - Addition and subtraction of Arrays

Aim: Write ALP using both simplified and full segment definition to perform Addition and Subtraction operations on array.

Components/ setup: emu8086 Emulator

Theory: The 8086 microprocessor supports unitary as well as array data. Working with normal 8-bit or 16-bit data is fairly simple, but working with arrays require knowledge about sizes of data elements and also offsets. Basic arithmetic operations like addition and subtraction can be easily done using this knowledge. In the below codes 'arr1' and 'arr2' are the given arrays and 'res' array contains the final arithmetic result. Also, for arithmetic operations on arrays, loops are used and CX register is used as counter and stored with the length of given array.

Code:

Listing 9: Addition of Arrays

```
;addition of 2 arrays

;Full segment definition
assume cs:code ds: data
data segment
    arr1 db 7,4,5,6
    arr2 db 1,2,4,3
    res db 4 dup <0> ;0,0,0,0
data ends

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

    lea si, arr1 ; mov si, offset arr1
    lea di, arr2
    lea bx, res
    mov cx, 4

loop1:
    mov al, [si]
    add al, [di]
    mov [bx], al
    inc bx
```

```
    inc si
    inc di
    dec cl
    jnz loop1

    mov ah, 4ch
    int 21h

code ends
end start

;Simplified segment definition
.model small
.data
    arr1 db 7,4,5,6
    arr2 db 1,2,4,3
    res db 4 dup <0> ;0,0,0,0
.code
.startup

    lea si, arr1 ; mov si, offset arr1
    lea di, arr2
    lea bx, res
    mov cx, 4

loop1:
    mov al, [si]
    add al, [di]
    mov [bx], al
    inc bx
    inc si
    inc di
    dec cl
    jnz loop1

    mov ah, 4ch
    int 21h

.exit
end
```

Listing 10: Subtraction of Arrays

```

; Subtract matrices

```

```
;Full segment definition
assume cs:code ds: data
data segment
    arr1 db 9, 9, 7, 8
    arr2 db 0, 0, 5, 3
    res db 4 dup <0> ;0,0,0,0
data ends

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

    lea si, arr1 ; mov si, offset arr1
    lea di, arr2
    lea bx, res
    mov cx, 4

loop1:
    mov al, [si]
    sub al, [di]
    mov [bx], al
    inc bx
    inc si
    inc di
    dec cl
    jnz loop1

    mov ah, 4ch
    int 21h

code ends
end start

;Simplified segment definition
.model small
.data
    arr1 db 9, 9, 7, 8
    arr2 db 0, 0, 5, 3
    res db 4 dup <0> ;0,0,0,0
.code
.startup

    lea si, arr1 ; mov si, offset arr1
    lea di, arr2
    lea bx, res
    mov cx, 4

loop1:
```

```
    mov al, [si]
    add al, [di]
    mov [bx], al
    inc bx
    inc si
    inc di
    dec cl
    jnz loop1

    mov ah, 4ch
    int 21h

.exit
end
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

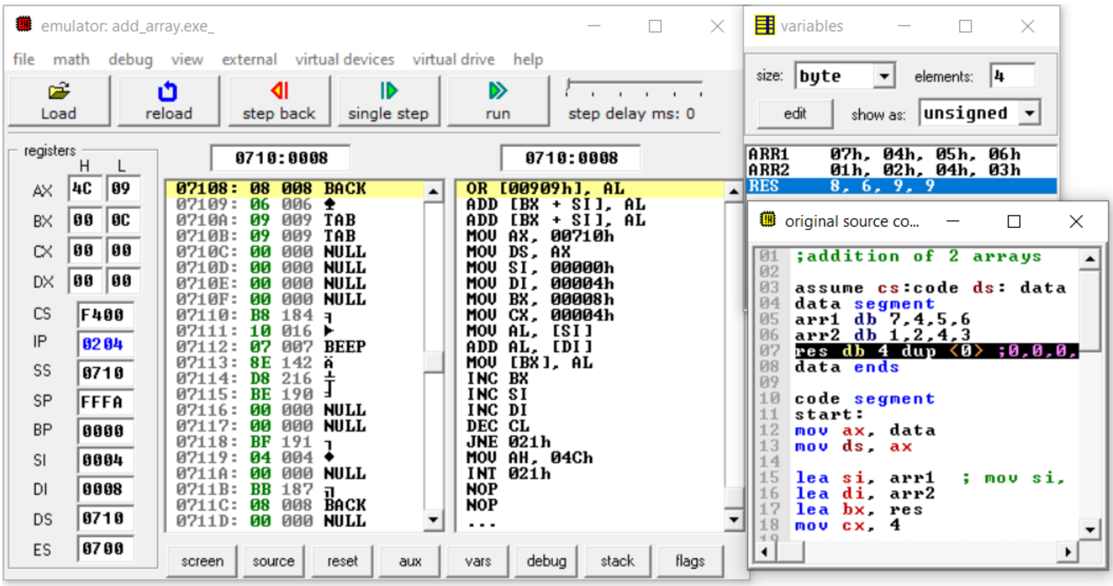

Figure 9: Addition of arrays

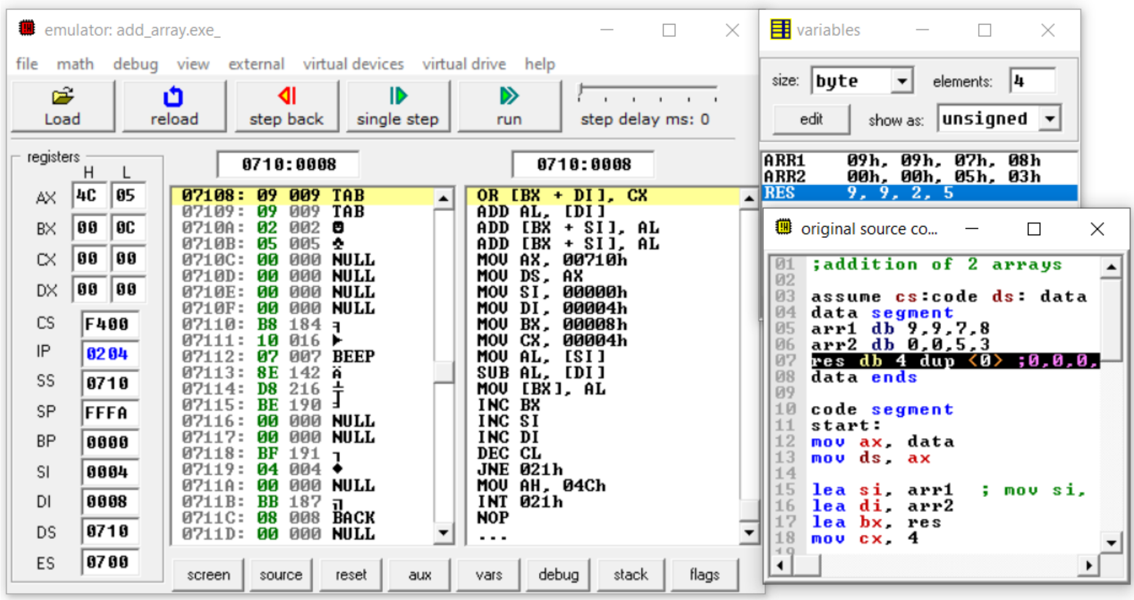

Figure 10: Subtraction of arrays

<u>Conclusion</u>: The Arithmetic Operations of addition and subtraction on arrays was successfully performed using x86 instructions on the emu8086 emulator.

Note: The output of above codes is 99, which is my roll no.(BT18ECE099)

Experiment 3 - Arithmetic operations on matrices

Aim: Write ALP using both simplified and full segment definition to perform Addition and Subtraction operations on 3x3 Matrix.

Components/ setup: emu8086 Emulator

Theory: In 8086 we can add two data which can be stored in the form of rows and columns (the difference between array and matrix is that matrices can have more than one rows). We traverse through the matrices of same size and add/sub the corresponding elements. In the following program we make use of corresponding elements but if a constant is to be added or matrices have to added in such a way that corresponding elements don't have to be added then that can also be achieved. This can be achieved by the following code.

Code:

Listing 11: Addition of Matrices

```

; addition matrices

;Full segment definition
assume cs:code ds: data

data segment
    matrix1 dw 01h, 02h, 03h
            dw 04h, 05h, 06h
            dw 07h, 08h, 09h

    matrix2 dw 08h, 07h, 06h
            dw 05h, 04h, 03h
            dw 02h, 01h, 00h

    result dw 9 dup<?>
data ends

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

    mov si,0
    mov ax,0
```

```
    mov bx,0
    mov cx,3

loop1:
    mov si, 0
    push cx
    mov cx, 3

loop2:
    mov ax,matrix1[bx][si]
    add ax,matrix2[bx][si]
    mov result[bx][si], ax
    inc si
    inc si
    loop loop2
    add bx, 6
    pop cx
    loop loop1

code ends
end start

;Simplified segment definition
.model small
.data
    matrix1 dw 01h, 02h, 03h
            dw 04h, 05h, 06h
            dw 07h, 08h, 09h

    matrix2 dw 08h, 07h, 06h
            dw 05h, 04h, 03h
            dw 02h, 01h, 00h

    result dw 9 dup<?>
.code
.startup

    mov si,0
    mov ax,0
    mov bx,0
    mov cx,3

loop1:
    mov si, 0
    push cx
    mov cx, 3

loop2:
    mov ax,matrix1[bx][si]
```

```
    add ax,matrix2[bx][si]
    mov result[bx][si], ax
    inc si
    inc si
    loop loop2
    add bx, 6
    pop cx
    loop loop1
.exit
end
```

Listing 12: Subtraction of Matrices

```

; Subtract matrices

;Full segment definition
assume cs:code ds: data

data segment
    matrix1 dw 0Ah, 0Bh, 0Ch
            dw 0Dh, 0Eh, 0Fh
            dw 11h, 12h, 13h

    matrix2 dw 01h, 03h, 04h
            dw 05h, 01h, 20h
            dw 02h, 03h, 02h

    result dw 9 dup<?>
data ends

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

    mov si,0
    mov ax,0
    mov bx,0
    mov cx,3

loop1:
    mov si, 0
    push cx
    mov cx, 3

```

```
loop2:
    mov ax,matrix1[bx][si]
    sub ax,matrix2[bx][si]
    mov result[bx][si], ax
    inc si
    inc si
    loop loop2
    add bx, 6
    pop cx
    loop loop1

code ends
end start

; Simplified segment definition
.model small
.data
    matrix1 dw 0Ah, 0Bh, 0Ch
            dw 0Dh, 0Eh, 0Fh
            dw 11h, 12h, 13h

    matrix2 dw 01h, 03h, 04h
            dw 05h, 01h, 20h
            dw 02h, 03h, 02h

    result dw 9 dup<?>
.code
.startup

    mov si,0
    mov ax,0
    mov bx,0
    mov cx,3

loop1:
    mov si, 0
    push cx
    mov cx, 3

loop2:
    mov ax,matrix1[bx][si]
    sub ax,matrix2[bx][si]
    mov result[bx][si], ax
    inc si
    inc si
    loop loop2
    add bx, 6
    pop cx
    loop loop1
```

```
.exit
end
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

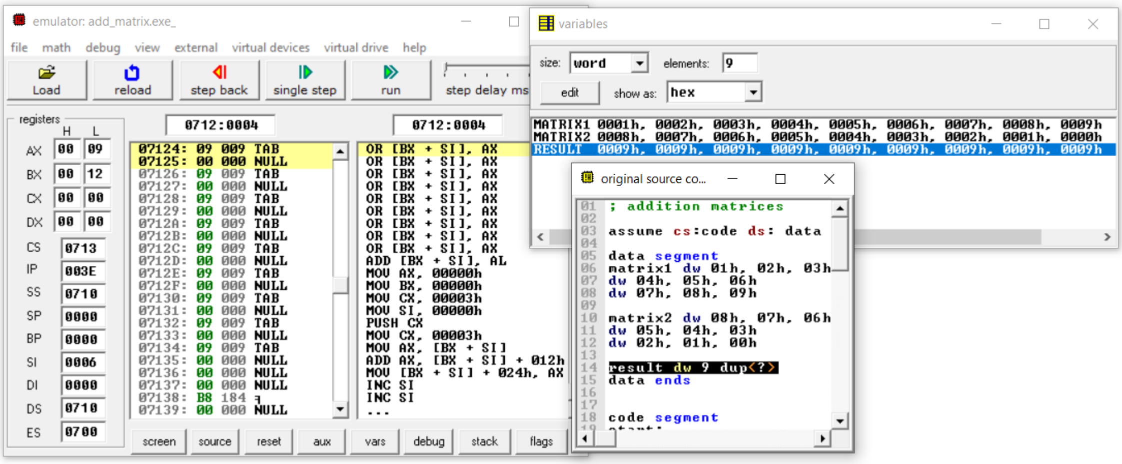

Figure 11: Addition of Matrices.

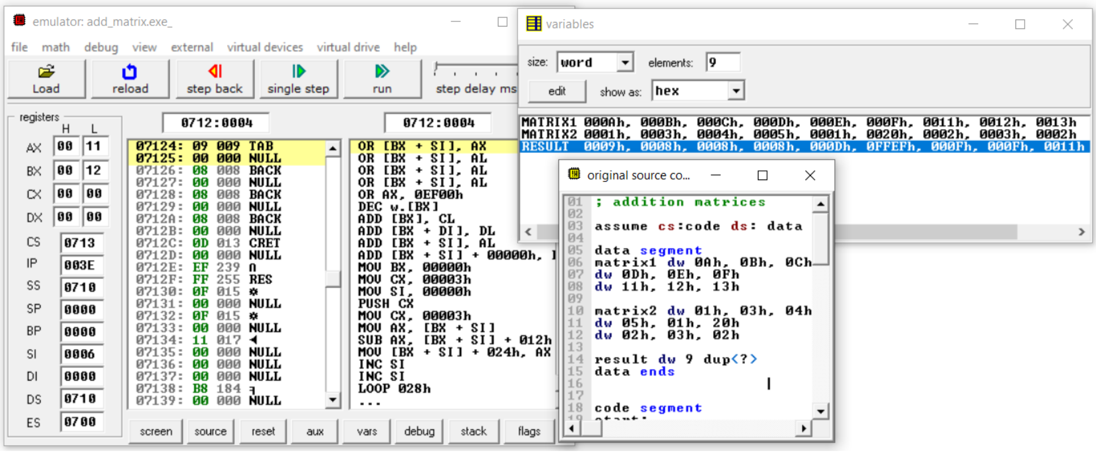

Figure 12: Subtraction of Matrices.

Conclusion: The Arithmetic Operations of addition and subtraction for 3x3 matrices was successfully performed using x86 instructions on the emu8086 emulator.

Note: The output of above codes is 99, which is my roll no.(BT18ECE099)

Experiment 4 - Sorting a given array

Aim: Write ALP using both simplified and full segment definition to sort an array in Ascending and Descending order.

Components/ setup: emu8086 Emulator

Theory: A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element. Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Code:

Listing 13: Ascending Order Sort

```

;Full Segment Definition
assume cs:code,ds:data
data segment
string1 db 99,12,56,45,36 ;storing unsorted array
data ends

code segment
start:
    mov ax,data
    mov ds,ax
    mov ch,04h
    up2:
    mov cl,04h
    lea si,string1
    up1:
    mov al,[si]
    mov bl,[si+1]
    cmp al,bl
    jc down
    mov dl,[si+1]
    xchg [si],dl
    mov [si+1],dl
    down:
    inc si
    dec cl
    jnz up1
```

```
    dec ch
    jnz up2
code ends
end start

;Simplified Segment Definition
.model small
.data
string1 db 99,12,56,45,36 ;storing unsorted array
.code
.startup
    mov ch,04h
    up2:
    mov cl,04h
    lea si,string1
    up1:
    mov al,[si]
    mov bl,[si+1]
    cmp al,bl
    jc down
    mov dl,[si+1]
    xchg [si],dl
    mov [si+1],dl
    down:
    inc si
    dec cl
    jnz up1
    dec ch
    jnz up2
.exit
end
```

Listing 14: Descending Order sort

```

; sort array in descending order
;Full Segment Definition
assume cs:code ds: data
data segment
    string1 db 99,12,56,45,36
data ends

code segment
    start: mov ax,data
    mov ds,ax
    mov ch,04h
```

```
    up2: mov cl,04h

    lea si,string1

    up1:mov al,[si]
    mov bl,[si+1]
    cmp al,bl
    jnc down
    mov dl,[si+1]
    xchg [si],dl
    mov [si+1],dl

    down: inc si
    dec cl
    jnz up1
    dec ch
    jnz up2
code ends
end start

; Simplified segment definition
.model small
.data
    string1 db 99,12,56,45,36
.code
.startup
    mov ch,04h
    up2: mov cl,04h

    lea si,string1

    up1: mov al,[si]
    mov bl,[si+1]
    cmp al,bl
    jnc down
    mov dl,[si+1]
    xchg [si],dl
    mov [si+1],dl

    down: inc si
    dec cl
    jnz up1
    dec ch
    jnz up2
.exit
end
```

Results : For developing & compiling the assembly output, 8086 Emulator is used.

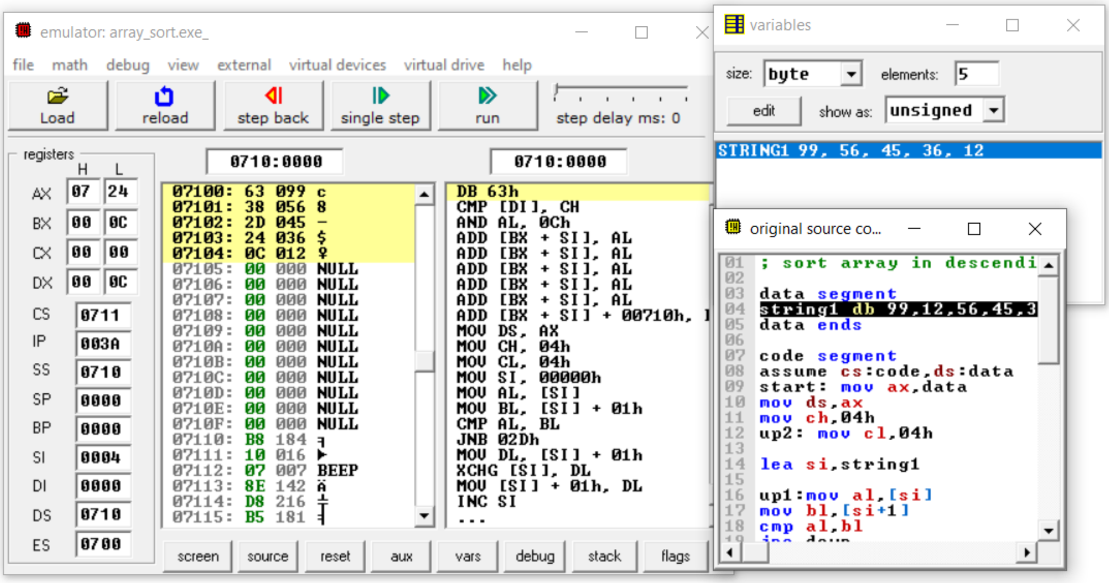

Figure 13: Sorting of array (Descending sort)

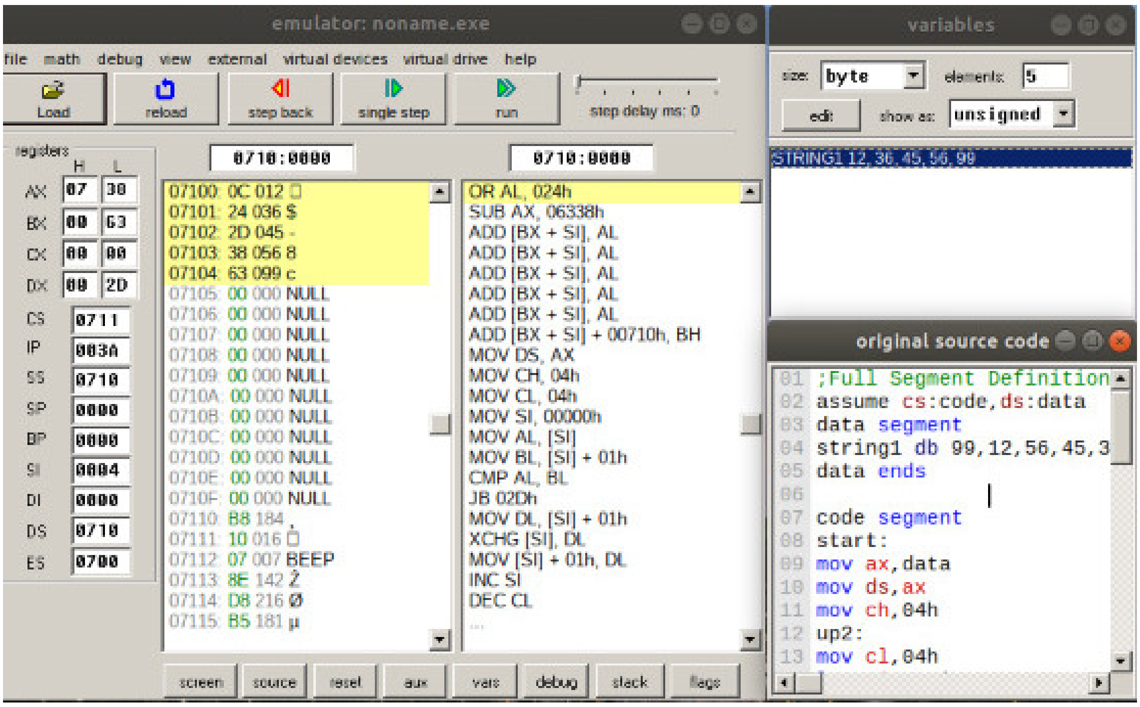

Figure 14: Sorting of array (Ascending sort)

<u>Conclusion</u>: Sorting of an array in Ascending and Descending order was successfully performed using x86 instructions on the emu8086 emulator.

Note: The unsorted array has 99, which is my roll no.(BT18ECE099)

Experiment 5 - Searching a number in a given array

Aim: Write ALP using both simplified and full segment definition to check if required number is present in a given array or not and display 'Num available' if found, else 'Num not available'.

Components/ setup: emu8086 Emulator

Theory: Searching Algorithms are designed to check for an element or retrieve an element from an array. Here, the list or array is traversed sequentially and every element is checked.
Different elements are stored in array of 8086. If we want to check whether the given element is present in array, we have to compare the element with each element of array until we found the element in array. If element is not found till the end of array, the given element is not present in the array. The following program performs the same task in 8086. It also displays the string whether the given number is found or not in array.

Code:

Listing 15: Linear Search

```
;finding an element in array
;if found display num available else num not available

;Full Segment Definition
assume cs:code ds: data
data segment
    arr1 db 12,1,14,99,15,22
    cnt db 99
    msg1 db 13,10,'Num available$',13,10 ;0dh,0ah
    msg2 db 13,10,'Num not available$',13,10
data ends

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

    mov cl,cnt
    mov di,offset arr1
    chk:
    mov al,99 ;number to check in array
```

```
    mov ah,[di]
    cmp al,ah
    jz disp1
    inc di
    dec cl
    jnz chk

    disp2: mov dx,offset msg2
    mov ah,9
    int 21h
    .exit

    disp1: mov dx,offset msg1
    mov ah,9
    int 21h
code ends
end start

;Simplified segment definition
.model small
.data
    arr1 db 12,1,14,99,15,22
    cnt db 99
    msg1 db 13,10,'Num available$',13,10 ;0dh,0ah
    msg2 db 13,10,'Num not available$',13,10
.code
.startup
start:
    mov cl,cnt
    mov di,offset arr1
    chk:
    mov al,15 ;number to check in array
    mov ah,[di]
    cmp al,ah
    jz disp1
    inc di
    dec cl
    jnz chk

    disp2: mov dx,offset msg2
    mov ah,9
    int 21h
    .exit

    disp1: mov dx,offset msg1
    mov ah,9
    int 21h
.exit
end
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

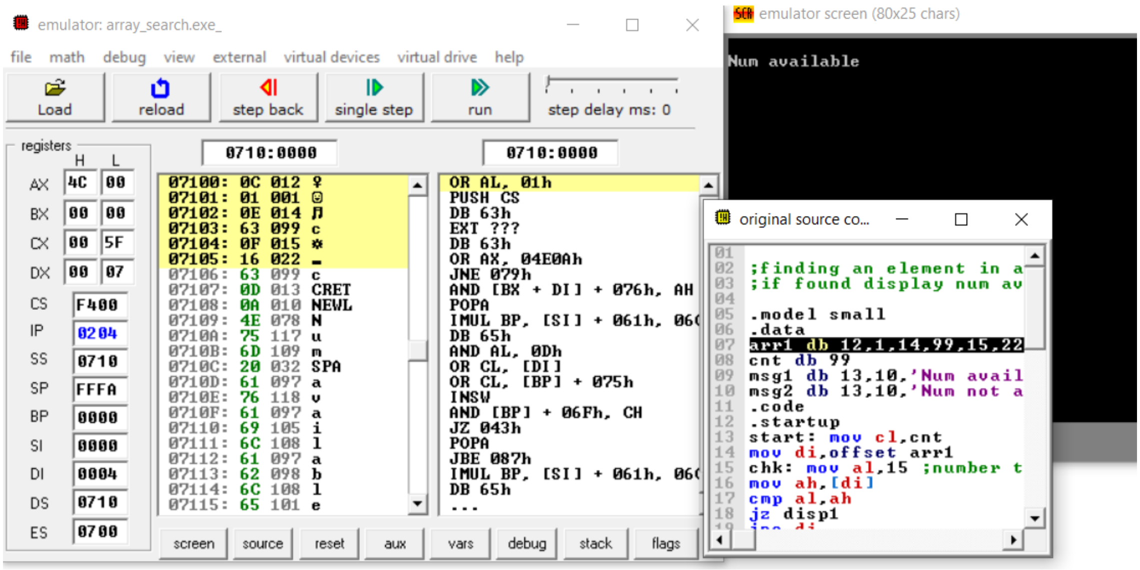

Figure 15: Linear Search in a array

Conclusion: The experiment for searching an element in the given array was successfully performed using x86 instructions on the emu8086 emulator.

Note: The key element to be searched in array is 99, which is my roll no.(BT18ECE099)

Experiment 6 - Count positive and negative number in a given array

Aim: Write an ALP to find out number of positive and negative numbers in a given array.

Components/ setup: emu8086 Emulator

Theory: In 8086, we can store both positive and negative integers as signed integers. In signed integers, the most significant bit represents the sign of the integer. If MSB is 1, then the number is negative, else the number is positive.
Take the i^{th} number in any of the registers. And subtract it with 0. The status of carry flag, overflow and zero flags are checked. If number greater, the number is positive; or else, it is negative.

Code:

Listing 16: Count of positive and negative numbers

```
;Full Segment Definition
assume cs:code ds:data
data segment
    arr1 db -13, 21, 17,-50, -5, 99 ; array
    negnos db 0 ; count of negative numbers
    posnos db 0 ; count of positive numbers
data ends

code segment
start:
    MOV AX,data
    MOV ds, AX ;load data to DS
    MOV SI, offset arr1 ; load address of array
    MOV CL, 6 ; count of number os elements in array
    l1:
        MOV AL, [SI] ; array element in AL
        CMP AL, 0 ; compare with 0
        JG pos ; if greater go to pos
        INC negnos ; increment count of - numbers
        JMP here ; jmp to here label
        pos:
            INC posnos ; increment count of + numbers

        here:
```

```
        INC SI ; increment array pointer
        LOOP l1 ; loop L1 till CL=0
code ends
end start

;Simplified segment definition
.model small
.data
    arr1 db -13, 21, 17,-50, -5, 99 ; array
    negnos db 0 ; count of negative numbers
    posnos db 0 ; count of positive numbers

.code
.startup
start:
    MOV AX,data
    MOV ds, AX ;load data to DS
    MOV SI, offset arr1 ; load address of array
    MOV CL, 6 ; count of number os elements in array
    l1:
        MOV AL, [SI] ; array element in AL
        CMP AL, 0 ; compare with 0
        JG pos ; if greater go to pos
        INC negnos ; increment count of - numbers
        JMP here ; jmp to here label
        pos:
            INC posnos ; increment count of + numbers

        here:
        INC SI ; increment array pointer
        LOOP l1 ; loop L1 till CL=0
.exit
end
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

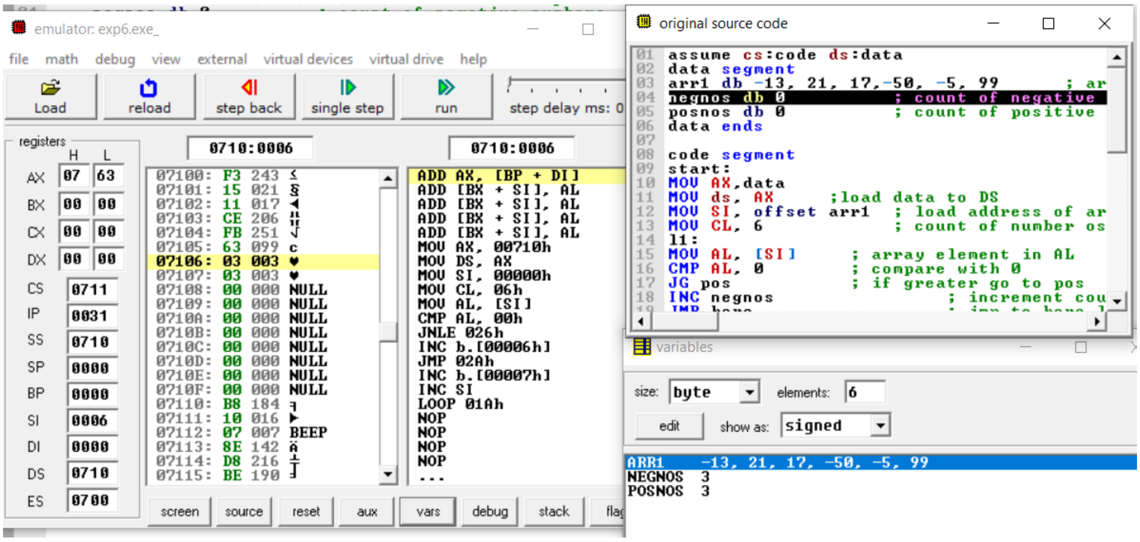

Figure 16: Count positive and negative numbers in a array

<u>Conclusion</u>: The experiment for counting positive and negative numbers in the given array was successfully performed using x86 instructions on the emu8086 emulator.

Note: One of the element in array is 99, which is my roll no.(BT18ECE099)

Experiment 7 - Generate Arithmetic Series

Aim: Write an ALP to generate first 'n' elements of arithmetic series where a=2 and d=3. Store the elements from 6500H onwards.

Components/ setup: emu8086 Emulator

Theory: a_n = a + (n-1)d
Above formula is arithmetic series formula which represents the sum of the first n terms in an arithmetic sequence.
The variable n, a and d are stored in CL, AL and BL respectively. Destination address is stored in DI pointer. N acts as counter value, the first term is already loaded into AL, and BL is holding the common difference d. The AL is placing as it is, then repeatedly add BL with AL and store it into memory pointed by DI to generate the sequence.

Code:

Listing 17: Arithmetic Series - Full Segment Definition

```
;Arithmetic Series
;Full Segment Definition
assume CS:code DS:data
data segment
    n db 9 ; number of elements in series
    a db 2 ; first element
    d db 3 ; difference
    arr1 db 9 dup <0> ; STORES ap
data ends

code segment
start:
      MOV CL, n ; store number of elements in cl
      MOV CH,00
      MOV AL, a ; move first element in AL

      MOV DI, 6500h ; destination pointer
      MOV [DI], AL ; store first element
      INC DI ; increment pointer
      DEC CL ; decrement count
      MOV BL, d ; move difference in BL

      LEA SI, arr1 ; load arr1 address in si
```

```
       MOV [SI], AL ; store first value
       INC SI

AP: ADD AL, BL ; add difference to previous element
       MOV [SI],AL ; store in arr1
       MOV [DI], AL ; store at destination
       INC DI ; increment destination pointer
       INC SI

       loop AP ; loop for n elements
code ends
     end start
```

Listing 18: Arithmetic Series- Simplified Segment Definition

```
; Arithmetic Series
; Simplified Small Definition

.model small
.data
     n db 9 ; number of elements in series
     a db 2 ; first element
     d db 3 ; difference
     arr1 db 9 dup <0> ; STORES ap
.code
.startup
       MOV CL, n ; store number of elements in cl
       MOV CH,00
       MOV AL, a ; move first element in AL

       MOV DI, 6500h ; destination pointer
       MOV [DI], AL ; store first element
       INC DI ; increment pointer
       DEC CL ; decrement count
       MOV BL, d ; move difference in BL

       LEA SI, arr1 ; load arr1 address in si
       MOV [SI], AL ; store first value
       INC SI

AP: ADD AL, BL ; add difference to previous element
       MOV [SI],AL ; store in arr1
       MOV [DI], AL ; store at destination
       INC DI ; increment destination pointer
       INC SI

```

```
        loop AP ; loop for n elements
.exit
end
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

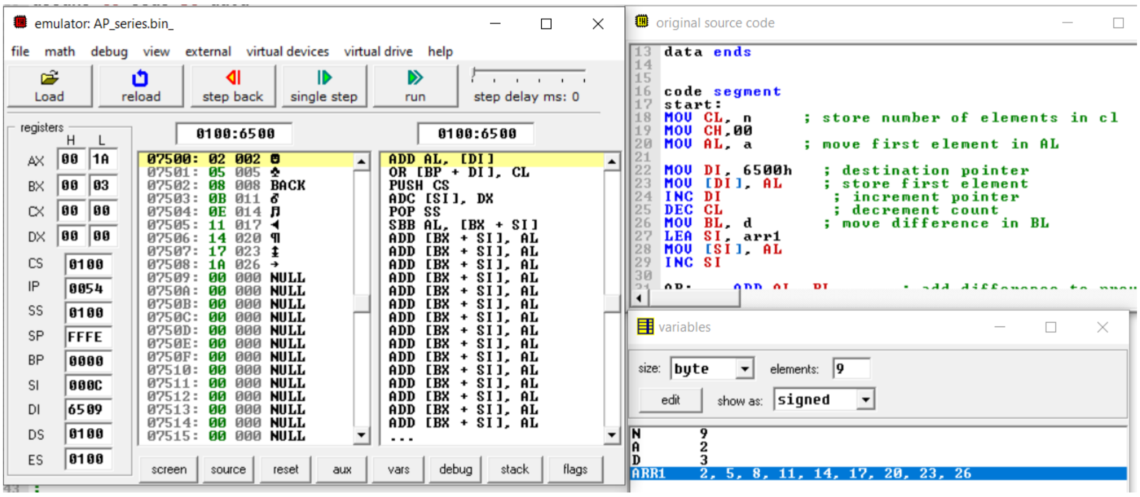

Figure 17: Arithmetic Series

Conclusion: The experiment for generating first 'n' elements of arithmetic series where a=2 and d=3 was successfully performed using x86 instructions on the emu8086 emulator. The result was stored from 6500H onwards.

Experiment 8 - Pattern - INT21H/ INT10H

Aim: Write an ALP to display triangle pattern using int 21h and int 10h

Components/ setup: emu8086 Emulator

Theory: Int 10h is a video service bios interrupt. It includes services like setting the video mode, character and string output, and reading and writing pixels in graphics mode. Int 21h is a DOS interrupt for keyboard operations.
In all calls, on entry AH defines the function. Other parameters may also be required in other registers. Where a memory block is used by the call this is specified in the normal segment:offset form. In all cases the general programming technique is to set AH to the function pointer, set up the required register contents (and the memory block if necessary) then to issue the call by the assembly code INT instruction.

Code:

Listing 19: Pattern using INT 21H

```
;int21h
;Full segment Definition

Assume cs:code ds:dsta
data segment
    cnt db 7
    star db 42,"$"
    ;smiley db 1,"$" ; 0dh,0ah
    ;heart db 3,"$"
    new db 13,10,'$' ; new line
data ends

code segment
start:
    MOV AX,data
    MOV ds, AX ;load data to DS

    mov cl,cnt ; rows count
    mov bl, 1
    loop1:
        mov ch, bl ; load bl to count
    loop2:
        mov dx,offset star ;move address of smiley variable
        mov ah,9 ;display character
        int 21h
```

```

        dec ch ; print till count 0
        jnz loop2

        mov dx,offset new ; print new line
        mov ah,9 ; display
        int 21h

        inc bl ; increment bl
        dec cl ; decrement cl
        jnz loop1

        mov dx,offset new ; move address of new line var
        mov ah,9 ; display char
        int 21h

code ends
end start


; Simplified Segment Definition

.model small
.data
    cnt db 7
    star db 42,"$"
    ;smiley db 1,"$" ;0dh,0ah
    ;heart db 3,"$"
    new db 13,10,'$' ; new line

.code
.startup
start:
    mov cl,cnt ; rows count
    mov bl, 1
loop1:
    mov ch, bl ; load bl to count
loop2:
    mov dx,offset star ;move address of smiley variable
    mov ah,9 ;display character
    int 21h

    dec ch ; print till count 0
    jnz loop2

    mov dx,offset new ; print new line
    mov ah,9 ; display
    int 21h

```

```
    inc bl ; increment bl
    dec cl ; decrement cl
    jnz loop1

    mov dx,offset new ; move address of new line var
    mov ah,9 ; display char
    int 21h

.exit
end
```

Listing 20: Right angled triangle Pattern using INT 10H

```
;display of right angled triangle
; Full Segment Definition

Assume cs:code ds:dsta
data segment
    disp1 db "*$"
    limit db 1
data ends

code segment
start:
    MOV AX,data
    MOV ds, AX ;load data to DS
    mov dl,5 ;column
    mov dh,5 ;row
    mov bh,0

    l1:
        mov ah,2 ;cursor position
        int 10h

        mov ah,9 ;display star
        mov al,disp1
        mov bl,11111101b ; white bg & light magenta color
        mov cl, limit ; times the disp1 will be displayed
        int 10h

        inc dh
        inc limit ; increment limit
        cmp limit,15 ; till 15 rows
        jne l1
    code ends
end start
```

```

; Simplified Segment Definition
.model small
.data
disp1 db "*$"
limit db 1

.code
.startup

mov dl,5 ;column
mov dh,5 ;row
mov bh,0

l1:
    mov ah,2 ;cursor position
    int 10h

    mov ah,9 ;display star
    mov al,disp1
    mov bl,11111101b ; white bg & light magenta color
    mov cl, limit ; times the disp1 will be displayed
    int 10h

    inc dh
    inc limit ; increment limit
    cmp limit,15 ; till 15 rows
    jne l1
end

.exit
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

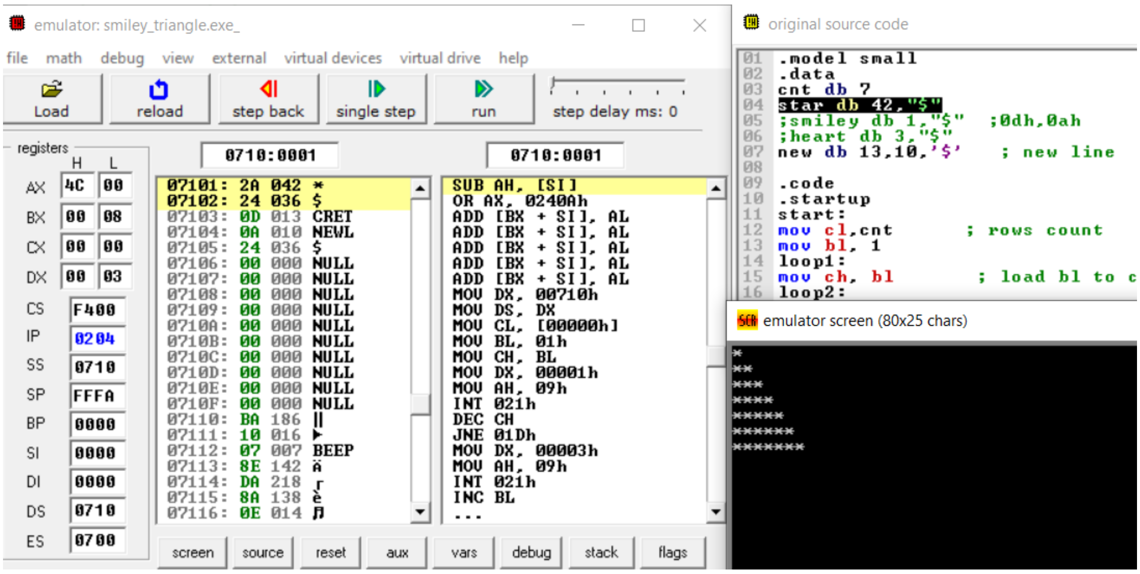

Figure 18: Star Pattern using INT 21H

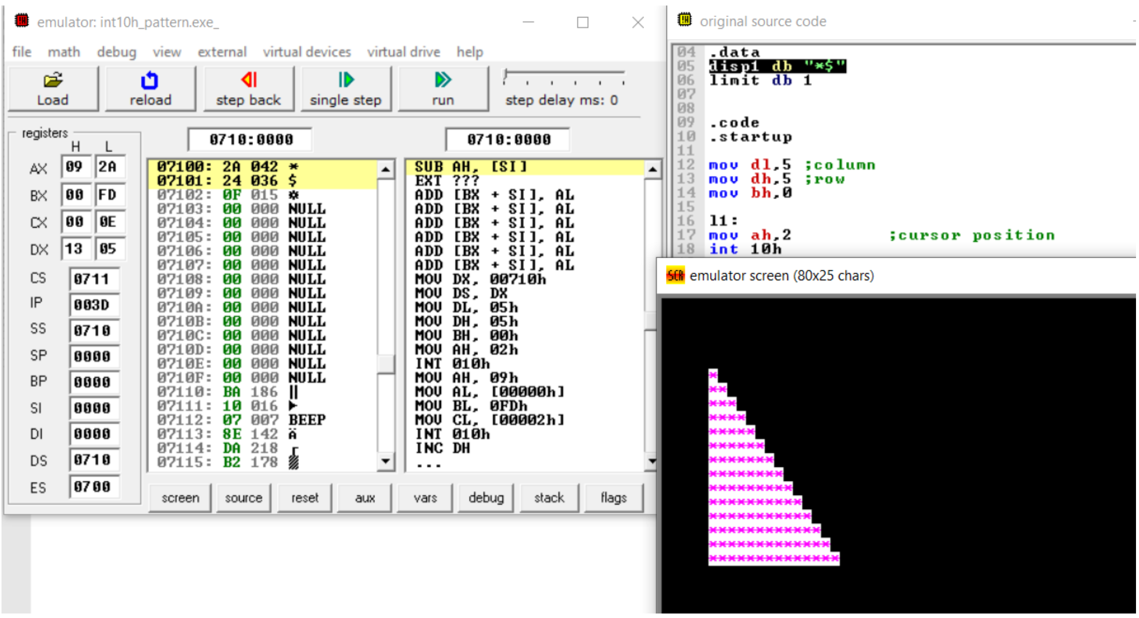

Figure 19: Star Pattern using INT 10H

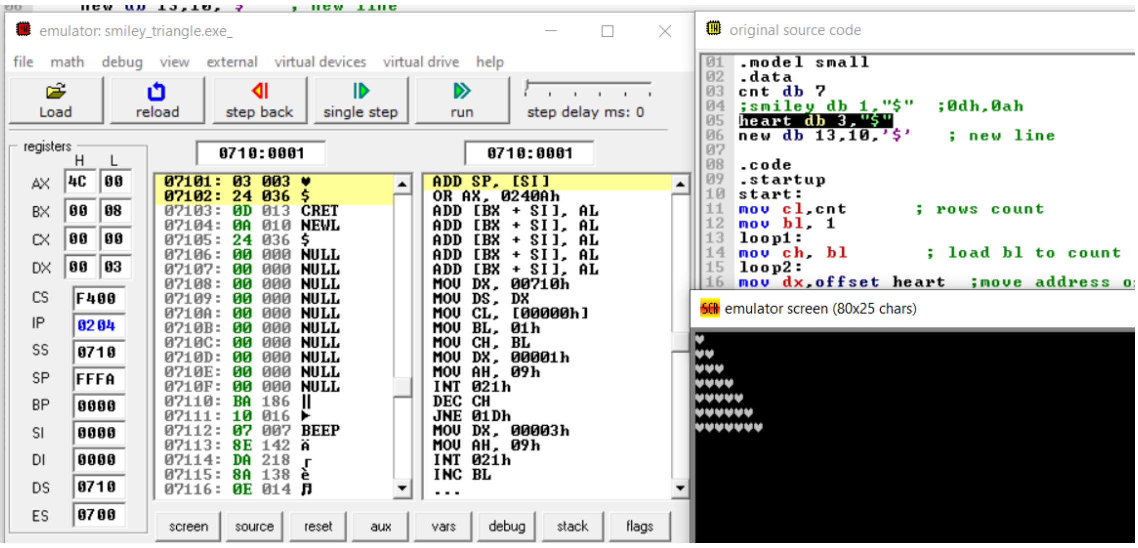

Figure 20: Heart Pattern using INT 21H

Conclusion: The experiment for displaying triangle pattern using int 21h and int 10H was successfully performed using x86 instructions on the emu8086 emulator. Heart and star patterns using INT 21H are showed in fig 20 and fig 18 respectively. Star pattern using INT 10H is shown in fig 19.

Experiment 9 - Find area of rectangle

Aim: Write an ALP to find area of rectangle/square by getting the data from the user.

Components/ setup: emu8086 Emulator

Theory: Area = length * breadth
Length and breadth of rectangle are taken as input from user using INT 21H. They are considered to be single digit number. Later they are multiplied using MUL or length is added breadth times to get area. This area is then divided by 10 to get quotient (tens place) and remainder (ones place). This digits are displayed one after other using INT 21H.

Code:

Listing 21: Area of Rectangle - Full Segment Definition

```
;area
;Full segment Definition

Assume CS:Code, DS:Data

Data Segment
    msg1 db "Enter the Length : $"
    msg2 db 13, 10, "Enter the Breadth : $"
    msg3 db 13, 10, "Area is : $"
    area1 db ?
    area2 db 00h
Data Ends

Code Segment

Start: MOV AX, Data
       MOV DS, AX ; load data to DS

       MOV AH, 09h
       LEA DX, msg1 ; display msg1
       INT 21h

       MOV AH, 01h ; get single character input
       INT 21h

       SUB AL, 30h ; ASCII to number
```

```
        MOV CL, AL ; move length to count
        SUB CL, 01h

        MOV AH, 09h
        LEA DX, msg2 ; display msg2
        INT 21h


        MOV AH, 01h ; get single character input
        INT 21h

        SUB AL, 30h ; ASCII to number

        MOV AH, 00h

        MOV CH, 00h
        MOV BX, AX ; move breadth to bc

 Here: ADD AX, BX
        DAA ; BCD adjust aftter addition
        Loop Here ; add breadth cl times

 disp: MOV AH, AL
        MOV Cl, 04h
        SHL AL, Cl
        SHR AL, Cl
        SHR AH, Cl
        ADD AX, 3030h

        MOV area1, AL
        MOV area2, AH

        MOV AH, 09h
        LEA DX, msg3 ; display msg3
        INT 21h

        MOV AH, 02h
        MOV DL, area2 ; display area2 tens place
        MOV DH, 00h
        INT 21h

        MOV AH, 02h
        MOV DL, area1 ; display area1 ones place
        MOV DH, 00h
        INT 21h
Code Ends
     End Start
```

Listing 22: Area of Rectangle - Simplified segment Definition

```
.model small
.data
    msg1 db "Enter the Length : $"
    msg2 db 13, 10, "Enter the Breadth : $"
    msg3 db 13, 10, "Area is : $"
    msg4 db 13,10,'21$', 13,10
.code
.startup
start:
    mov dx,offset msg1
    mov ah,9 ; display msg1
    int 21h

    mov ah, 1h ; take input
    int 21h

    sub al, 30h ; ASCII to decimal conversion
    mov bl, al

    mov dx,offset msg2
    mov ah,9 ; display msg2
    int 21h

    mov ah, 1h ; take input
    int 21h

    sub al, 30h ; ASCII to decimal conversion

    mul bl ; multiply length
    mov bl, al ; store in BL

    mov dx,offset msg3
    mov ah,9 ; display msg3
    int 21h

    mov al, bl
    mov ah, 0
    mov bl, 10
    div bl ; divide al by 10
    mov bl, ah ; get ones place
    add al, 30h


    mov offset msg4[2], al
    add ah, 30h
    mov offset msg4[3], ah
```

```
    mov dx,offset msg4
    mov ah,9
    int 21h
.exit
end
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

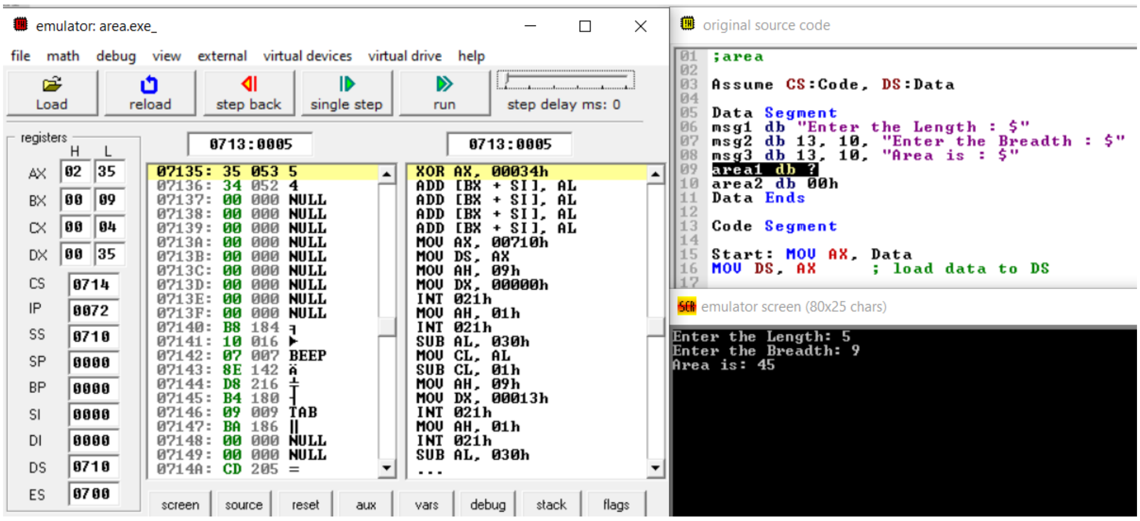

Figure 21: Area of rectangle - fsd

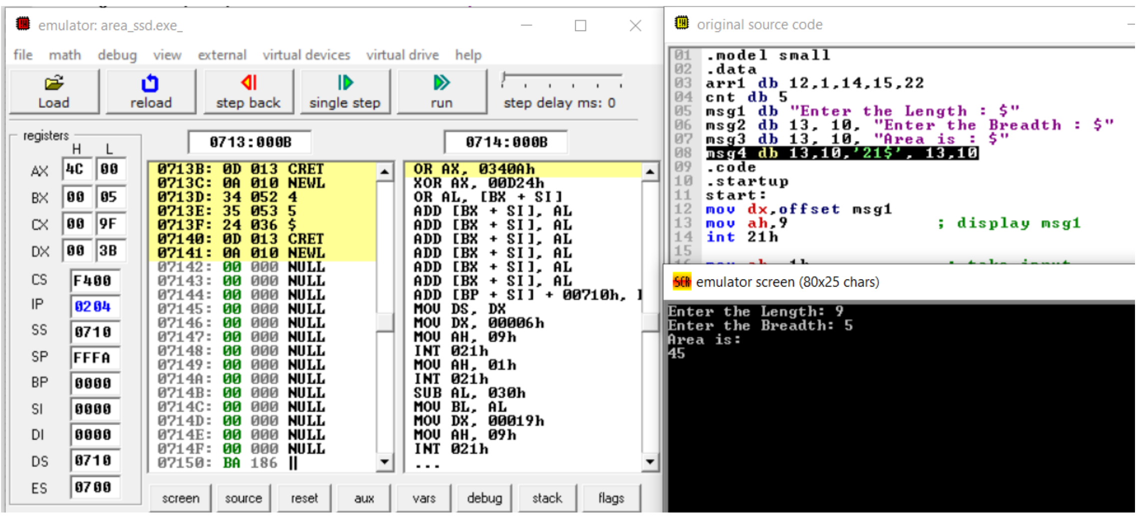

Figure 22: Area of rectangle - ssd

Conclusion: The experiment to find area of rectangle/square by getting the data from the user was successfully performed using x86 instructions on the emu8086 emulator.

Experiment 10 - Get string input and reverse it

Aim: Write an ALP to get a string as an input from the user and reverse it and display on the screen.

Components/ setup: emu8086 Emulator

Theory: String is taken from user using INT 21H/ AH 0AH or looping single character input INT 21H/ AH 9H. The address of last character of string is stored in pointer SI. The values are then transferred to DL and is printed using INT 21H/ AH 02H. The process continues till CX = 0.

Code:

Listing 23: Reverse string - Simplified segment Definition

```

; ssd

.model small
.data
msg1 db "Enter the string:$"
msg2 db 13,10,"The reversed string is:$"
str db ?

.code
.startup

mov ah,9h
lea dx,msg1 ;display msg1
int 21h

lea si,str ; load address off str1
mov cl,1h ; counter value 1
jmp scanstring

scanstring:
    mov ah, 01h
    int 21h ; take single character input

    cmp al,13 ; check if pressed enter
    je display ; if pressed go to display
    mov [si],al ; else continue scanning
    inc si
```

```
    inc cl ; incrrement count
    jmp scanstring

display:
    lea dx,msg2
    mov ah,9h
    int 21h ; display msg2
    jmp reverse

reverse:
    mov al,[si]

    mov dl,al
    mov dh,0
    mov ah,02h ; display character
    int 21h

    dec si ; increment array pointer
    dec cl ; decrement counter value
    cmp cl,0 ; check if CX=0
    jne reverse ; if not 0 continue reverse loop
.exit
end
```

Listing 24: Reverse string - Full segment Definition

```

;Get a string as an input from the user and reverse it and display on the screen.

Assume CS:Code, DS:Data, ES:Extra
Data Segment
    msg1 db "Enter a String : $"
    msg2 db 0Ah, 0Dh, "The reversed string is : $"
    str db 08 dup('$')
Data Ends

Extra Segment
    rev_str db 07 dup(00), '$' ; For Reverse string
Extra Ends

Code Segment

Start: mov ax, Data
       mov ds, ax ; Load data to DS
       mov ax, Extra
       mov es, ax ; Load extra to ES
```

```

    mov ah, 09h
    lea dx, msg1 ; display msg1
    int 21h

    lea dx, str
    mov ah, 0Ah ; take string input
    mov data, 04h
    int 21h

    MOV al, str[01h] ; AL will now contain the elements.
    MOV ah, 00h ; Now AX would contain the count.
    mov cx, ax ; Transfering the coung to the CX register.

    lea si, str + 02h ;load effective address
    lea di, rev_str + 06h

Here:
    cld ;clear direction flag
    lodsb ;read from source string
    std ;set direction flag
    stosb ;write into distination string cld
    loop here

    MOV al, str[01h] ; AL will now contain the elements.
    MOV ah, 00h ; Now AX would contain the count.
    mov cx, ax ; Transfering the coung to the CX register.
    inc cx

    mov si, di
    inc si

    lea di, str ; effective address of string

    mov ax, data ; move data to ES
    mov es, ax

    mov ax, extra ; move extra to ES
    mov ds, ax

    cld ; clear direction flag

    rep movsb ; move byte from string 1 to string2 till CX =0

    mov ax, data ; move data to DS
    mov ds, ax

    mov ah, 09h
    lea dx, msg2 ; display msg2
```

```
        int 21h

        mov ah, 09h
        lea dx, str ; display reverse string
        int 21h

Code Ends
    End Start
```

Results: For developing & compiling the assembly output, 8086 Emulator is used.

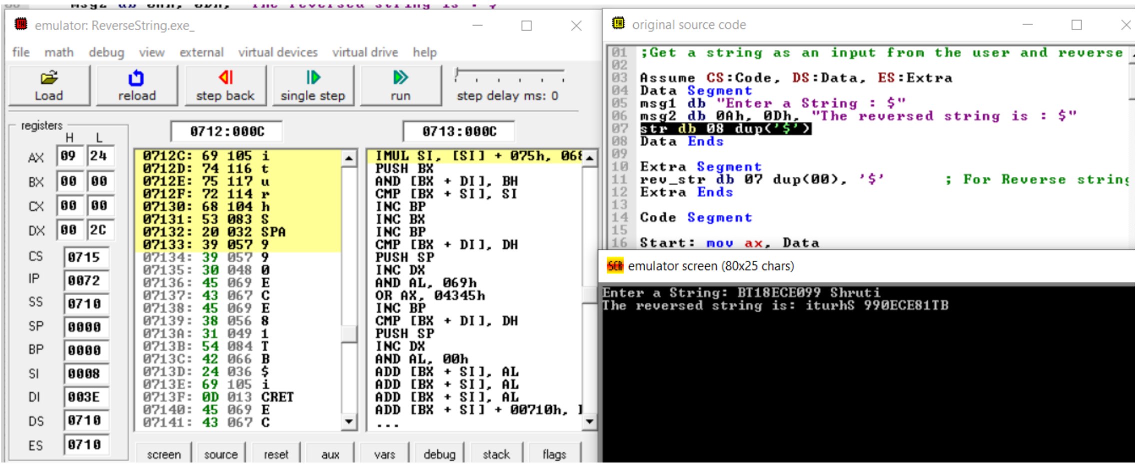

Figure 23: Reverse String

Conclusion: The experiment to to get a string as an input from the user and reverse it and display on the screen was successfully performed using x86 instructions on the emu8086 emulator.

Experiment 11 - Display "8086 LAB" on seven segment display.

Aim: Use 8255 PPI to display "8086 LAB" on seven segment display.

Components/ setup: Proteus 8 Professional v 8.10

Theory: PPI 8255 is a general purpose programmable I/O device designed to interface the CPU with its outside world such as ADC, DAC, keyboard etc. It consists of three 8-bit bidirectional I/O ports i.e. PORT A, PORT B and PORT C. These three ports are further divided into two groups, i.e. Group A includes PORT A and upper PORT C. Group B includes PORT B and lower PORT C. We can assign different ports as input or output functions.
To interface 7 segment display, 8255 is configured in Mode 0 [input-output mode]. A seven-segment LED is a kind of LED(Light Emitting Diode) consisting of 7 small LEDs. Common Anode 7 segment Led is used. Port A is connected with 7-segment display as output.
Here we are using a common anode display therefore 0 logic is needed to activate the segment. Suppose to display number 9 at the seven-segment display, therefore the segments F, G, B, A, C, and D have to be activated.

- We store the 99H in the accumulator i.e. 10010000.
- Port A address is loaded in register D.
- Finally, By OUT instruction we are sending the data stored in the accumulator to the port A.

Code:

Listing 25: Display "8086 LAB" on 7 - seg

```
ASSUME CS:CODE, DS:DATA

DATA SEGMENT
     PORTA EQU 00H
     PORTB EQU 02H
     PORTC EQU 04H
     PORT_CON EQU 06H
DATA ENDS

CODE SEGMENT
```

```
    MOV AX,DATA
    MOV DS, AX

    ORG 0000H
START:

    MOV DX, PORT_CON
    MOV AL, 10000000B ; port C (output), port A (output) in mode 0 and PORT ...
        B (INPUT) in mode 0
    OUT DX, AL
    JMP XX
XX:
  MOV AL, 0FFH
  MOV DX, PORTA
  OUT DX,AL
  CALL DELAY

  MOV AL, 80H ; 8
  MOV DX, PORTA
  OUT DX,AL
  CALL DELAY

  MOV AL, 0C0H ; 0
  MOV DX, PORTA
  OUT DX,AL
  CALL DELAY

  MOV AL, 80H ; 8
  MOV DX, PORTA
  OUT DX,AL
  CALL DELAY

  MOV AL, 82H ; 6
  MOV DX, PORTA
  OUT DX,AL
  CALL DELAY

  MOV AL, 0C7H ; L
  MOV DX, PORTA
  OUT DX,AL
  CALL DELAY

  MOV AL, 88H ; A
  MOV DX, PORTA
  OUT DX,AL
  CALL DELAY

  MOV AL,83H ; B
  MOV DX, PORTA
```

```
  OUT DX,AL
  CALL DELAY

JMP XX

DELAY PROC NEAR
    MOV CX,0DF36H ; Delay
    loop1:loop loop1
    RET
DELAY ENDP


CODE ENDS
END
```

Results: For developing & compiling the assembly output, Proteus is used.

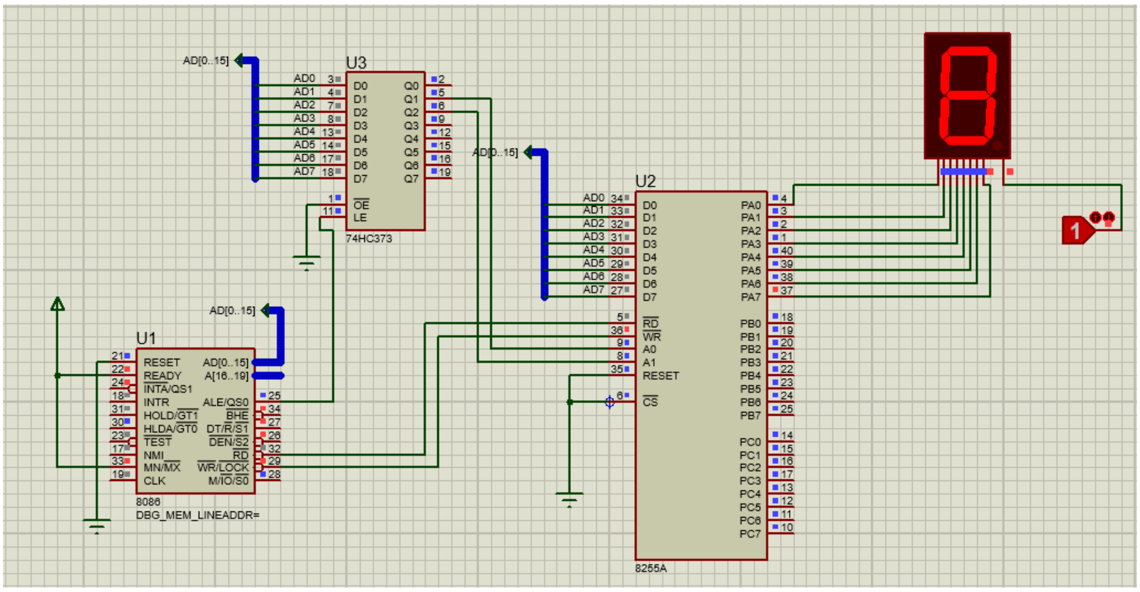

Figure 24: 8255 with 8086

Conclusion: The experiment to display '8086 LAB' on 7-segment display was successfully performed on the Proteus using 8086 and 8255.

Experiment 12 - Use 8255 PPI to check for pressed key and turn on the corresponding LED.

Aim: Use 8255 PPI to check for pressed key and turn on the corresponding LED.

Components/ setup: Proteus 8 Professional v 8.10

Theory: 8255 is used to interface switches and LEDs. To interface LED and switch, 8255 is configured in Mode 0 [input-output mode]. Port A is connected with switches and is acting as input. Port B is connected to LEDs and is acting as output. Input from port A is taken using IN and is passed to port B using OUT.

Code:

Listing 26: LED with switches

```
assume cs:code, ds:data
data segment
    porta equ 00H
    portb equ 02H
    portc equ 04H
    cwr equ 06H
data ends

code segment
    mov ax, data
    mov ds, ax
    org 0000H

start:
    mov al, 91H
    out cwr, al
    l1:
        mov al, 00H
        in al, porta ; switch input
    out portb, al ; output to led
    jmp l1
code ends
end
```

Results: For developing & compiling the assembly output, Proteus is used.

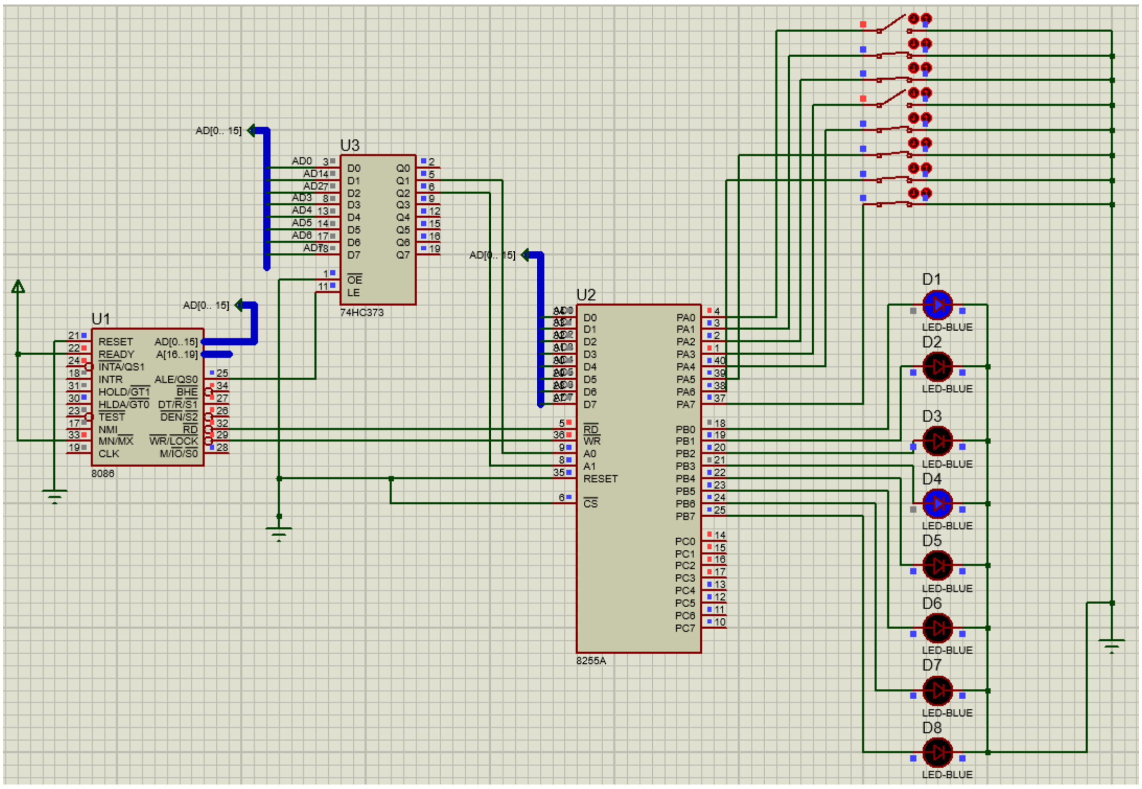

Figure 25: LED with Switches

Conclusion: The experiment to turn LED for corresponding switch was successfully performed on the Proteus using 8086 and 8255.

Experiment 13 - Generate a square wave of 1ms.

Aim: Use 8253 PIT to generate a square wave of 1 ms.

Components/ setup: Proteus 8 Professional v 8.10

Theory: 8253 and 8254 are Programmable Interval Timers (PTIs) designed for microprocessors to perform timing and counting functions using three 16-bit registers. Each counter has 2 input pins, i.e. Clock Gate, and 1 pin for "OUT" output. It has three independent 16-bit down counters. These three counters can be programmed for either binary or BCD count. It can handle inputs from DC to 10 MHz.
8253 is configured in mode 3 i.e. square wave generator.
Calculation for count value:

T = 1ms
f = 1/T = 1KHz
Input clock frequency to 8053 = 2Mhz
Count for register = clock frequency / square wave frequency
= 2Mhz/ 1KHz
= 2000 = 070D H

Code:

Listing 27: Square wave of 1ms using 8253

```
 ASSUME CS:CODE, DS:DATA

 DATA SEGMENT
    COUNTER0 EQU 69H
    COUNTER1 EQU 6BH
    COUNTER2 EQU 6DH
    CWR EQU 6FH
DATA ENDS

CODE SEGMENT
     MOV AX, DATA
     MOV DS, AX
    ORG 0000H

START:
    ; --> Square Wave :: MOD 3 of 8253
    MOV AL, 10110111b ; 10 [counter 2], 11[ 2 byte data], 011 [ with mod 3], ...
        1 [BCD]
```

```
    OUT CWR, AL   ; configure the counter 2 of 8253

    MOV AL, 0D0h
    OUT COUNTER2, AL   ; Send first byte of Counter 2
    MOV AL, 07h
    OUT COUNTER2, AL   ; Send second byte of Counter 2

ENDLESS:
       JMP ENDLESS
CODE ENDS
       END START
```

Results: For developing & compiling the assembly output, Proteus is used.

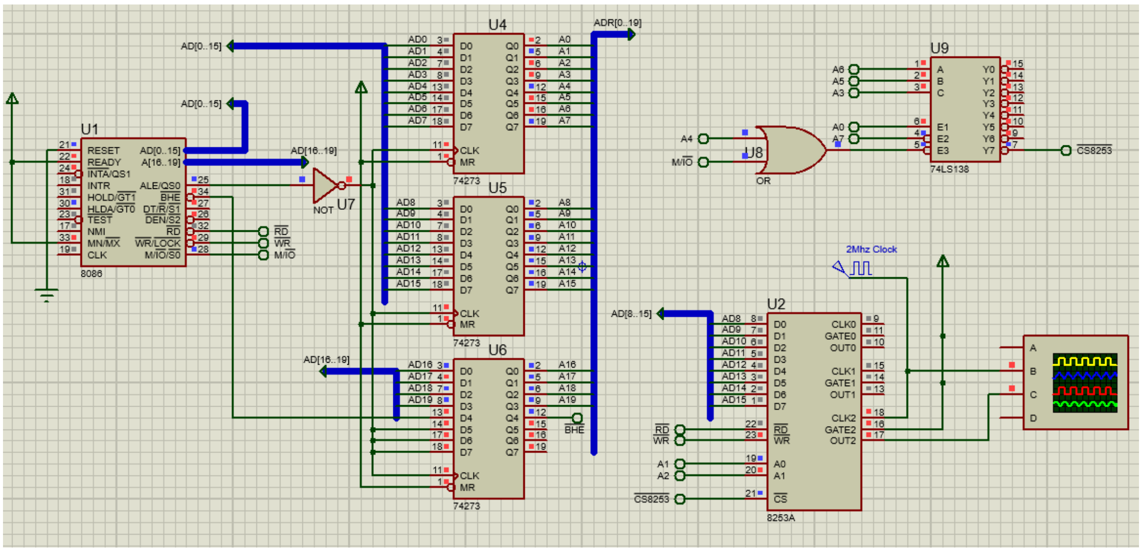

Figure 26: 8253 with 8086

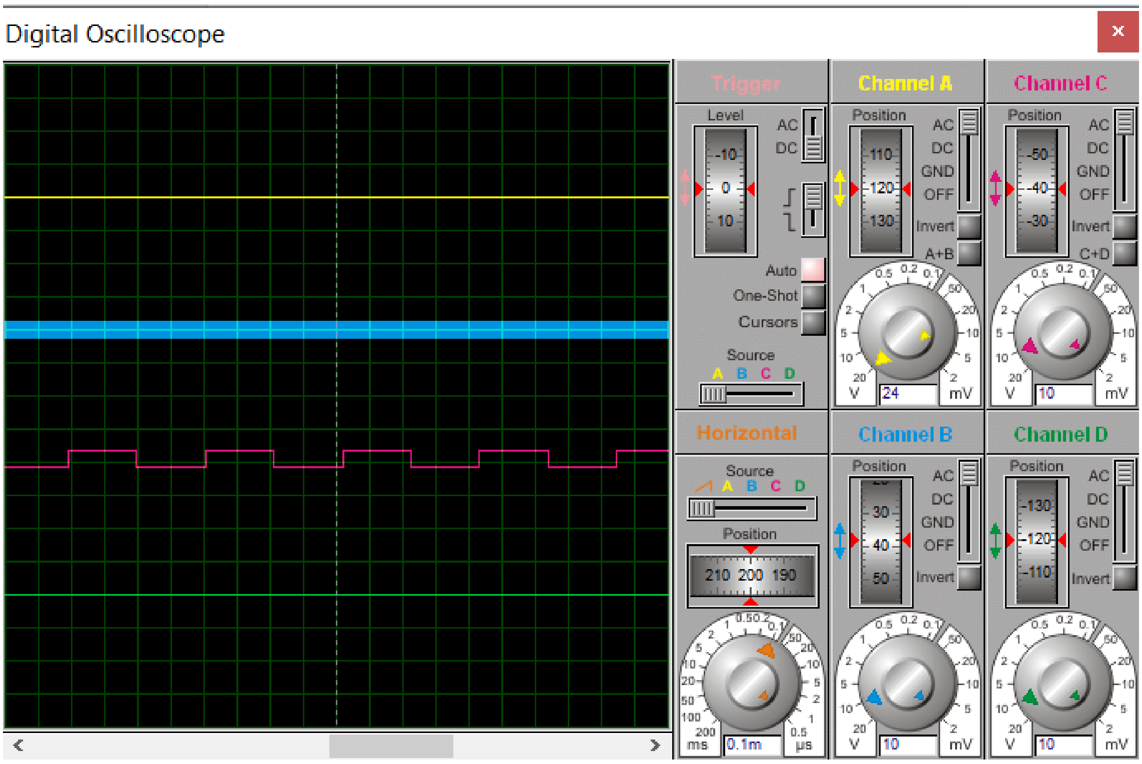

Figure 27: Oscilloscope [B - 2us input clock, C - 1ms square wave]

Conclusion: The experiment to generate a square wave of 1 ms was successfully performed on the Proteus using 8086 and 8253.

Experiment 14 - Compute and display the area of right angled triangle.

Aim: Compute and display the area of right angled triangle, given its base and height.

Components/ setup: emu8086, DOSBox

Theory: 8087 is a Maths co processor. It supports data of type integer, float, and real types ranging from 2-10 bytes. The processing speed is so high that it can calculate multiplication of two 64-bits real numbers in 27 μs and can also calculate square-root in 35 μs. It follows IEEE floating point standards.

Area of right angled triangle = 0.5*base*height

Code:

Listing 28: Area of triangle

```
Data Segment
    base dd 9.9
    height dd 6.2
    half dd 0.5
    area dt ?
    areamsg db "Area of right angled triangle is: $"
    a dw ?
    b dt 100.00
    dot db ".$"
    msg4 db 10d,13d,"$"
Data Ends


Code Segment
    Assume CS:Code, DS:Data
Start:

    scall macro xx, yy ;macro to display
        lea dx,xx
        mov ah,yy
        int 21h
    endm

    displayreal macro xx,yy
        fld xx
```

```
        scall msg4,09h
        scall yy,09h

        fld b ;load 100.00
        fmulp st[1],st ;multiply by 100.00
        fbstp xx ;convert to BCD

        mov ax,word ptr xx
        call display4digit ;display ax contents
    endm

    mov ax, Data
    mov ds, ax

    finit
    fld base ; load base
    fld height ; load height
    fmul ; multiply base and height
    fld half ; load 0.5
    fmul ; multiply half to result
    fstp area ; store result in area

    displayreal area, areamsg


    mov ah, 4ch ; terminate program
    int 21h


display4digit proc
    mov a,ax ;ax has 4 digits to diaplay
    mov dl,ah ;to display ah first

    mov ch,02h ;count of digits
    rol dl,01
    rol dl,01
    rol dl,01
    rol dl,01
loop3: and dl,0fh
    add dl,30h

    mov ah,02h
    int 21h

    mov ax,a ;restore ax
    mov dl,ah

    dec ch
    jnz loop3
```

```

    scall dot,09h ;display dot

    mov ax,a
    mov dl,al ;to display al

    mov ch,02h ;count of digits
    rol dl,01
    rol dl,01
    rol dl,01
    rol dl,01

loop4: and dl,0fh ;anding to extract last digit
    add dl,30h ;convert to ascii-hex

    mov ah,02h ;display dl contents
    int 21h

    mov ax,a
    mov dl,al

    dec ch
    jnz loop4
RET
display4digit Endp

Code Ends
     End Start
```

Results: For developing & compiling the assembly output, 8086 Emulator & DOS-Box are used respectively.

Figure 28: Area of right angled triangle

Conclusion: The experiment to calculate area of right angled triangle was successfully calculated using 8087 co-processor and 8086.