



# 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

1	Experiment 1 - Arithmetic operations on 8-bit and 16-bit numbers. . . . .	2
2	Experiment 2 - Addition and subtraction of Arrays . . . . .	14
3	Experiment 3 - Arithmetic operations on matrices . . . . .	19
4	Experiment 4 - Sorting a given array . . . . .	24
5	Experiment 5 - Searching a number in a given array . . . . .	29
6	Experiment 6 - Count positive and negative number in a given array . . . . .	32
7	Experiment 7 - Generate Arithmetic Series . . . . .	35
8	Experiment 8 - Pattern - INT21H/ INT10H . . . . .	38
9	Experiment 9 - Find area of rectangle . . . . .	44
10	Experiment 10 - Get string input and reverse it . . . . .	49
11	Experiment 11 - Display "8086 LAB" on seven segment display. . . . .	53
12	Experiment 12 - Use 8255 PPI to check for pressed key and turn on the corresponding LED. . . . .	56
13	Experiment 13 - Generate a square wave of 1ms. . . . .	58
14	Experiment 14 - Compute and display the area of right angled triangle. . . . .	61

## 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 8085 Microprocessor. 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

```
1 ;Full Segment Definition
2 assume cs:code,ds:data
3 data segment
4     a db 11h
5     b db 88h
6     c db ?
7 data ends
8 code segment
9 start:
```

```
10     mov ax,data
11     mov ds,ax
12     mov al,a
13     mov bl,b
14     add al,bl
15     mov c,al
16 code ends
17 end start
18
19 ;Simplified Segment Definition
20 .model small
21 .data
22     a db 11h
23     b db 88h
24     c db ?
25 .code
26 .startup
27     mov al,a
28     mov bl,b
29     add al,bl
30     mov c,al
31 .exit
32 end
```

Listing 2: Addition of 16-bit numbers

```
1 ;Full Segment Definition
2 assume cs:code,ds:data
3 data segment
4     num1 dw 0011h
5     num2 dw 3388h
6     num3 dw ?
7 data ends
8
9 code segment
10 start:
11     mov ax,data
12     mov ds,ax
13     mov ax,num1
14     mov bx,num2
15     add ax,bx
16     mov num3,ax
17 code ends
18 end start
19
20 ;Simplified Segment Definition
```

```
21 .model small
22 .data
23     num1 dw 0011h
24     num2 dw 3388h
25     num3 dw ?
26
27 .code
28 .startup
29     mov ax,num1
30     mov bx,num2
31     add ax,bx
32     mov num3,ax
33 .exit
34 end
```

Listing 3: Subtraction of 8-bit numbers

```
1
2 ;Full Segment Definition
3 assume cs:code,ds:data
4 data segment
5     a db 99h
6     b db 09h
7     c db ?
8 data ends
9
10 code segment
11 start:
12     mov ax,data
13     mov ds,ax
14     mov al,a
15     mov bl,b
16     sub al,bl
17     mov c,al
18 code ends
19 end start
20
21 ;Simplified Segment Definition
22 .model small
23 .data
24     a db 99h
25     b db 09h
26     c db ?
27 .code
28 .startup
29     mov al,a
```

```
30     mov bl,b
31     sub al,bl
32     mov c,al
33     .exit
34     end
```

Listing 4: Subtraction of 16-bit numbers

```
1  ;Full Segment Definition
2  assume cs:code,ds:data
3  data segment
4      num1 dw 9999h
5      num2 dw 1111h
6      num3 dw ?
7  data ends
8
9  code segment
10 start:
11     mov ax,data
12     mov ds,ax
13     mov ax,num1
14     mov bx,num2
15     sub ax,bx
16     mov num3,ax
17 code ends
18 end start
19
20 ;Simplified Segment Definition
21 .model small
22 .data
23     num1 dw 9999h
24     num2 dw 1111h
25     num3 dw ?
26 .code
27 .startup
28
29     mov ax,num1
30     mov bx,num2
31     sub ax,bx
32     mov num3,ax
33     .exit
34     end
```

Listing 5: Multiplication of 8-bit numbers

```
1 ;Full Segment Definition
2 assume cs:code,ds:data
3 data segment
4     a db 11h
5     b db 09h
6     c db ?
7 data ends
8 code segment
9 start:
10     mov ax,data
11     mov ds,ax
12     mov al,a
13     mov bl,b
14     mul bl
15     mov c,al
16 code ends
17 end start
18
19 ;Simplified Segment Definition
20 .model small
21 .data
22     a db 11h
23     b db 09h
24     c db ?
25 .code
26 .startup
27
28     mov al,a
29     mov bl,b
30     mul bl
31     mov c,al
32 .exit
33 end
```

Listing 6: Multiplication of 16-bit numbers

```
1 ;Full Segment Definition
2 assume cs:code,ds:data
3 data segment
4     n1 dw 0099h
5     n2 dw 0005h
6     ans dw ?
7
8 data ends
9 code segment
10    begin:
```

```
11     mov ax,data
12     mov ds,ax
13     mov ax,n1
14     mov bx,n2
15     mul bx
16     mov ans,ax
17 code ends
18 end begin
19
20 ;Simplified Segment Definition
21 .model small
22 .data
23     n1 dw 0099h
24     n2 dw 0005h
25     ans dw ?
26
27 .code
28 .startup
29     mov ax,n1
30     mov bx,n2
31     mul bx
32     mov ans,ax
33 .exit
34 end
```

Listing 7: Division of 8-bit numbers

```
1 ;Full Segment Definition
2 assume cs:code,ds:data
3 data segment
4     a db 99
5     b db 8
6     quo db ?
7     rem db ?
8 data ends
9 code segment
10 start:
11     mov ax,data
12     mov ds,ax
13     mov ah,0h
14     mov al,a
15     mov bl,b
16     div bl
17     mov quo,al
18     mov rem,ah
19 code ends
```



```
20 end start
21
22 ;Simplified Segment Definition
23 .model small
24 .data
25     a db 99
26     b db 8
27     quo db ?
28     rem db ?
29 .code
30 .startup
31     mov ah,0h
32     mov al,a
33     mov bl,b
34     div bl
35     mov quo,al
36     mov rem,ah
37 .exit
38 end
```

Listing 8: Division of 16-bit numbers

```
1 ;Full Segment Definition
2 assume cs:code,ds:data
3 data segment
4     n1 dw 1899h
5     n2 dw 0099h
6     Q dw ?
7     R dw ?
8 data ends
9 code segment
10     begin:
11     mov ax,data
12     mov ds,ax
13     mov ax,n1
14
15     mov bx,n2
16     div bx
17     mov Q,ax
18     mov R,dx
19 code ends
20 end begin
21
22 ;Simplified Segment Definition
23 .model small
24 .data
```

```

25     n1 dw 1899h
26     n2 dw 0099h
27     Q dw ?
28     R dw ?
29 .code
30 .startup
31     mov ax,n1
32     mov bx,n2
33     div bx
34     mov Q,ax
35     mov R,dx
36 .exit
37 end

```

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

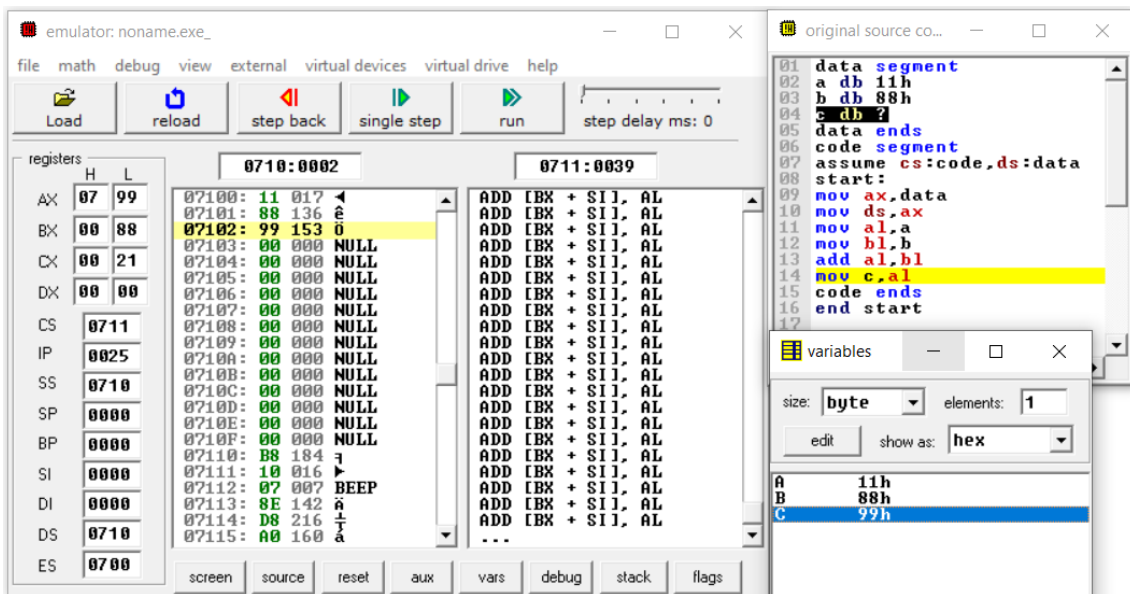


Figure 1: Addition of 8-bit numbers.

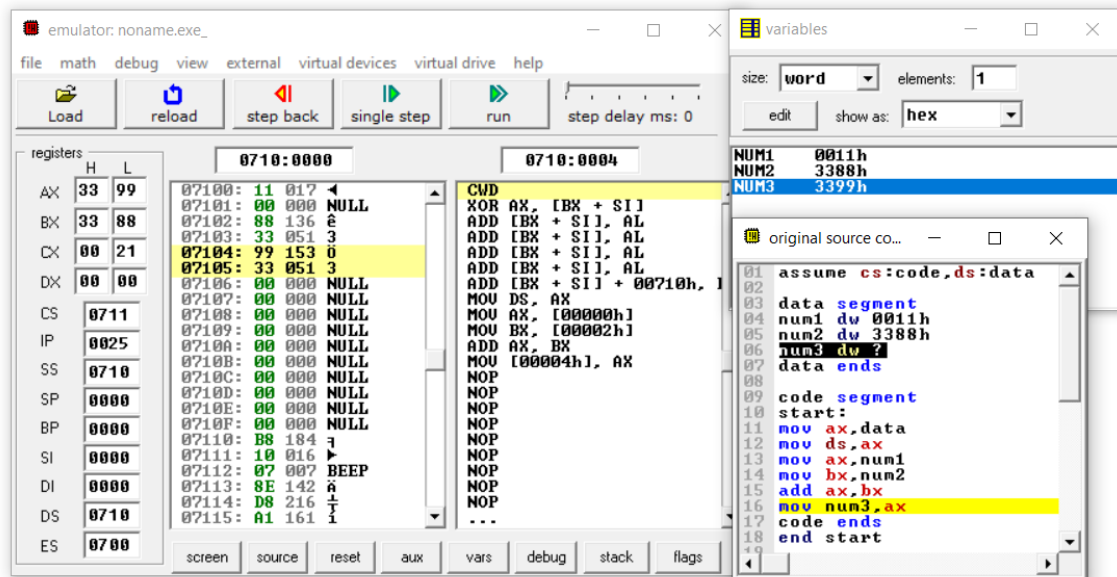


Figure 2: Addition of 16-bit numbers.

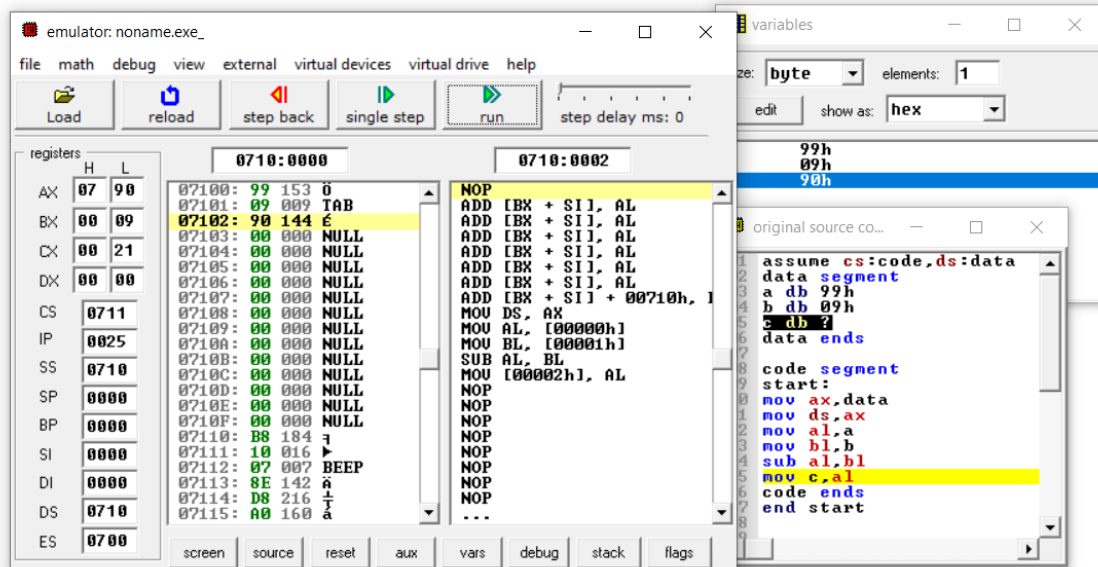


Figure 3: Subtraction of 8-bit numbers.

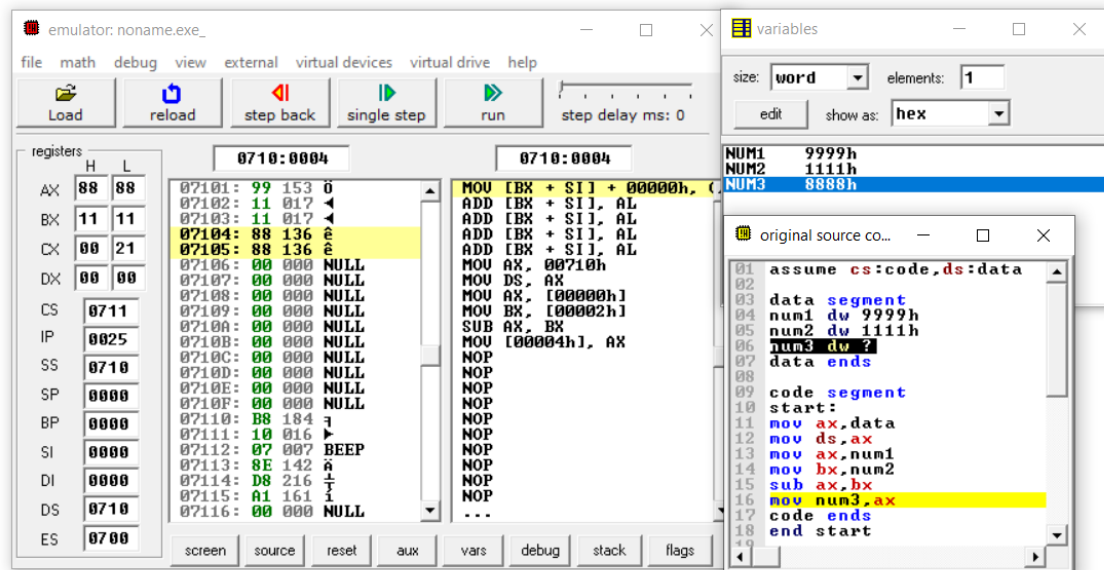


Figure 4: Subtraction of 16-bit numbers.

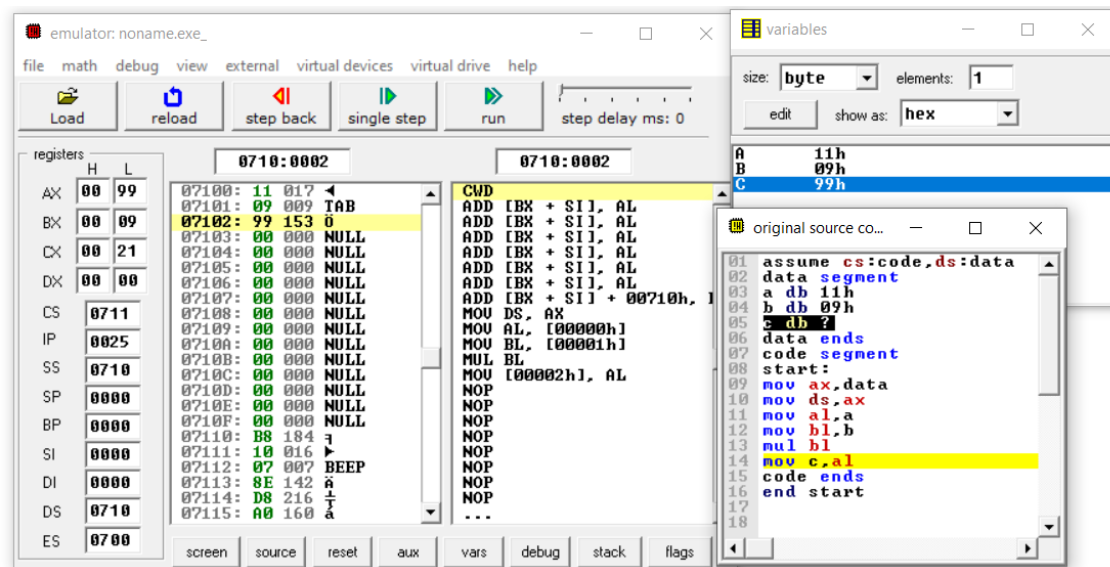


Figure 5: Multiplication of 8-bit numbers.

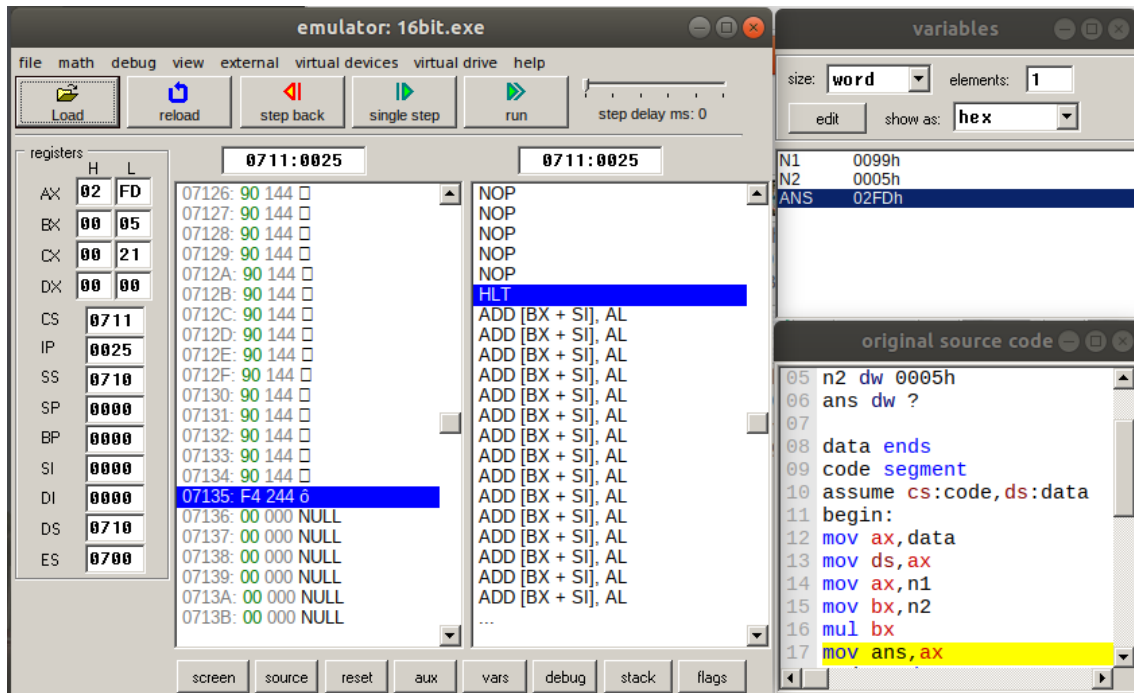


Figure 6: Multiplication of 16-bit numbers.

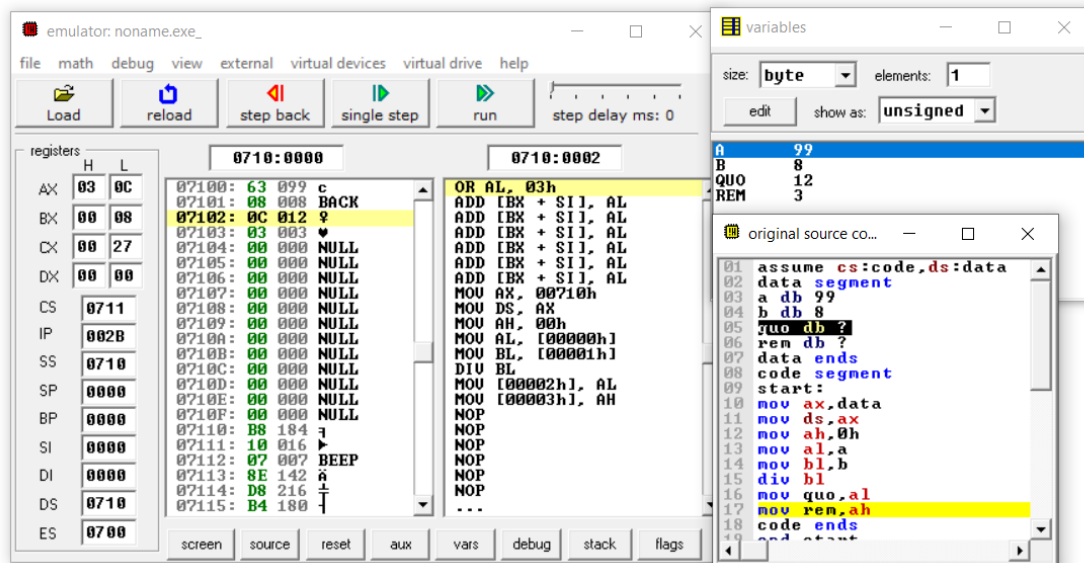


Figure 7: Division of 8-bit numbers.

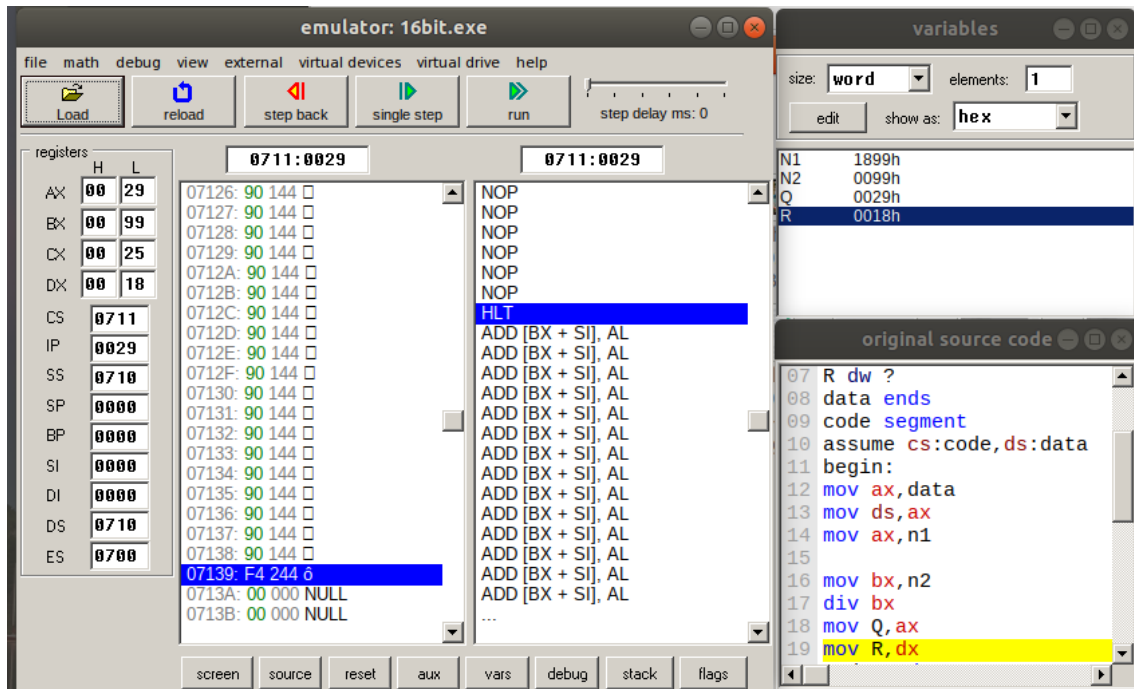


Figure 8: Division of 16-bit numbers.

**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

```
1  ;addition of 2 arrays
2
3  ;Full segment definition
4  assume cs:code ds: data
5  data segment
6      arr1 db 7,4,5,6
7      arr2 db 1,2,4,3
8      res db 4 dup <0> ;0,0,0,0
9  data ends
10
11 code segment
12     start:
13     mov ax, data
14     mov ds, ax
15
16     lea si, arr1 ; mov si, offset arr1
17     lea di, arr2
18     lea bx, res
19     mov cx, 4
20
21 loop1:
22     mov al, [si]
23     add al, [di]
24     mov [bx], al
25     inc bx
```

```
26     inc si
27     inc di
28     dec cl
29     jnz loop1
30
31     mov ah, 4ch
32     int 21h
33
34 code ends
35 end start
36
37 ;Simplified segment definition
38 .model small
39 .data
40     arr1 db 7,4,5,6
41     arr2 db 1,2,4,3
42     res db 4 dup <0> ;0,0,0,0
43 .code
44 .startup
45
46     lea si, arr1 ; mov si, offset arr1
47     lea di, arr2
48     lea bx, res
49     mov cx, 4
50
51 loop1:
52     mov al, [si]
53     add al, [di]
54     mov [bx], al
55     inc bx
56     inc si
57     inc di
58     dec cl
59     jnz loop1
60
61     mov ah, 4ch
62     int 21h
63
64 .exit
65 end
```

Listing 10: Subtraction of Arrays

```
1
2 ; Subtract matrices
3
```



```
4 ;Full segment definition
5 assume cs:code ds: data
6 data segment
7     arr1 db 9, 9, 7, 8
8     arr2 db 0, 0, 5, 3
9     res db 4 dup <0> ;0,0,0,0
10 data ends
11
12 code segment
13     start:
14     mov ax, data
15     mov ds, ax
16
17     lea si, arr1 ; mov si, offset arr1
18     lea di, arr2
19     lea bx, res
20     mov cx, 4
21
22 loop1:
23     mov al, [si]
24     sub al, [di]
25     mov [bx], al
26     inc bx
27     inc si
28     inc di
29     dec cl
30     jnz loop1
31
32     mov ah, 4ch
33     int 21h
34
35 code ends
36 end start
37
38 ;Simplified segment definition
39 .model small
40 .data
41     arr1 db 9, 9, 7, 8
42     arr2 db 0, 0, 5, 3
43     res db 4 dup <0> ;0,0,0,0
44 .code
45 .startup
46
47     lea si, arr1 ; mov si, offset arr1
48     lea di, arr2
49     lea bx, res
50     mov cx, 4
51
52 loop1:
```

```

53     mov al, [si]
54     add al, [di]
55     mov [bx], al
56     inc bx
57     inc si
58     inc di
59     dec cl
60     jnz loop1
61
62     mov ah, 4ch
63     int 21h
64
65 .exit
66 end

```

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

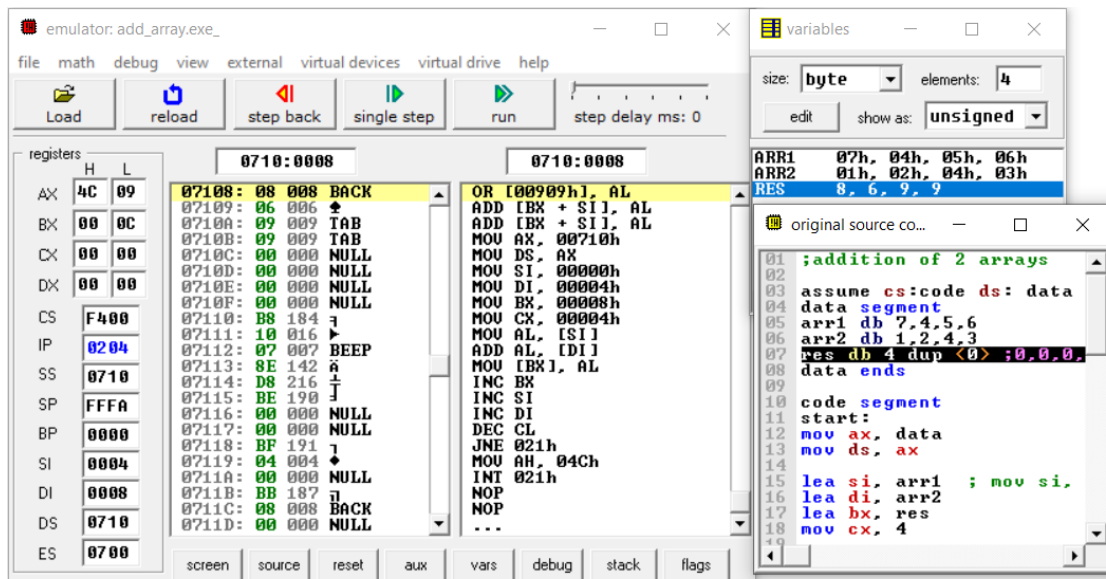


Figure 9: Addition of arrays

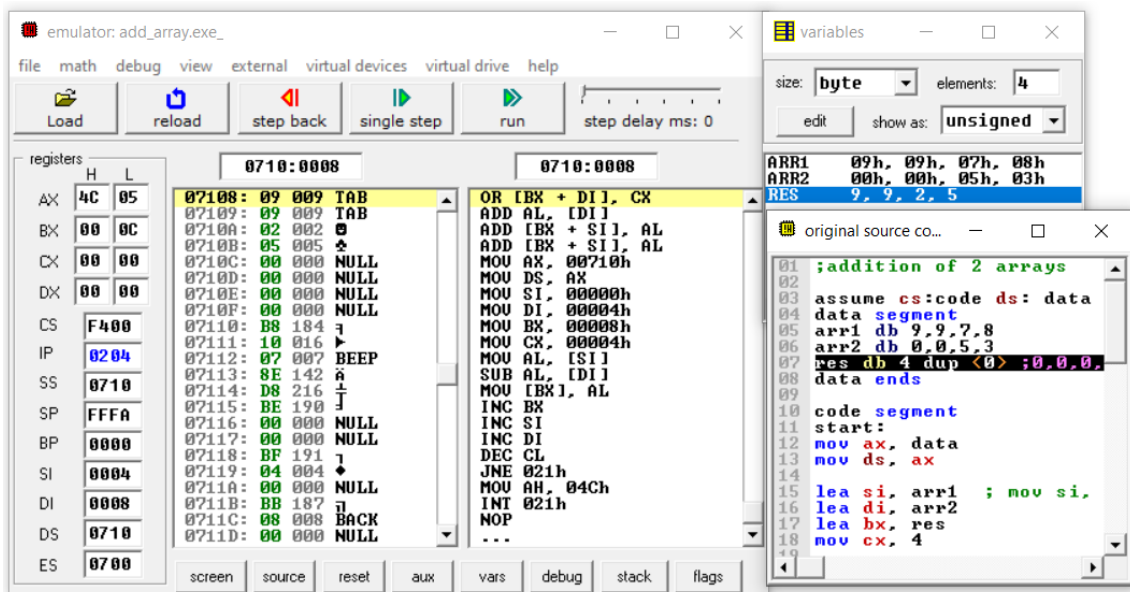


Figure 10: Subtraction of arrays

**Conclusion:** 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

```
1
2 ; addition matrices
3
4 ;Full segment definition
5 assume cs:code ds: data
6
7 data segment
8     matrix1 dw 01h, 02h, 03h
9             dw 04h, 05h, 06h
10            dw 07h, 08h, 09h
11
12     matrix2 dw 08h, 07h, 06h
13            dw 05h, 04h, 03h
14            dw 02h, 01h, 00h
15
16     result dw 9 dup<?>
17 data ends
18
19 code segment
20 start:
21     mov ax,data
22     mov ds, ax
23
24     mov si,0
25     mov ax,0
```

```
26     mov bx,0
27     mov cx,3
28
29 loop1:
30     mov si, 0
31     push cx
32     mov cx, 3
33
34 loop2:
35     mov ax,matrix1[bx][si]
36     add ax,matrix2[bx][si]
37     mov result[bx][si], ax
38     inc si
39     inc si
40     loop loop2
41     add bx, 6
42     pop cx
43     loop loop1
44
45 code ends
46 end start
47
48 ;Simplified segment definition
49 .model small
50 .data
51     matrix1 dw 01h, 02h, 03h
52             dw 04h, 05h, 06h
53             dw 07h, 08h, 09h
54
55     matrix2 dw 08h, 07h, 06h
56             dw 05h, 04h, 03h
57             dw 02h, 01h, 00h
58
59     result dw 9 dup<?>
60 .code
61 .startup
62
63     mov si,0
64     mov ax,0
65     mov bx,0
66     mov cx,3
67
68 loop1:
69     mov si, 0
70     push cx
71     mov cx, 3
72
73 loop2:
74     mov ax,matrix1[bx][si]
```

```
75     add ax,matrix2[bx][si]
76     mov result[bx][si], ax
77     inc si
78     inc si
79     loop loop2
80     add bx, 6
81     pop cx
82     loop loop1
83 .exit
84 end
```

Listing 12: Subtraction of Matrices

```
1
2 ; Subtract matrices
3
4 ;Full segment definition
5 assume cs:code ds: data
6
7 data segment
8     matrix1 dw 0Ah, 0Bh, 0Ch
9             dw 0Dh, 0Eh, 0Fh
10            dw 11h, 12h, 13h
11
12     matrix2 dw 01h, 03h, 04h
13            dw 05h, 01h, 20h
14            dw 02h, 03h, 02h
15
16     result dw 9 dup<?>
17 data ends
18
19 code segment
20 start:
21     mov ax,data
22     mov ds, ax
23
24     mov si,0
25     mov ax,0
26     mov bx,0
27     mov cx,3
28
29 loop1:
30     mov si, 0
31     push cx
32     mov cx, 3
33
```

```
34 loop2:
35     mov ax,matrix1[bx][si]
36     sub ax,matrix2[bx][si]
37     mov result[bx][si], ax
38     inc si
39     inc si
40     loop loop2
41     add bx, 6
42     pop cx
43     loop loop1
44
45 code ends
46 end start
47
48 ; Simplified segment definition
49 .model small
50 .data
51     matrix1 dw 0Ah, 0Bh, 0Ch
52             dw 0Dh, 0Eh, 0Fh
53             dw 11h, 12h, 13h
54
55     matrix2 dw 01h, 03h, 04h
56             dw 05h, 01h, 20h
57             dw 02h, 03h, 02h
58
59     result dw 9 dup<?>
60 .code
61 .startup
62
63     mov si,0
64     mov ax,0
65     mov bx,0
66     mov cx,3
67
68 loop1:
69     mov si, 0
70     push cx
71     mov cx, 3
72
73 loop2:
74     mov ax,matrix1[bx][si]
75     sub ax,matrix2[bx][si]
76     mov result[bx][si], ax
77     inc si
78     inc si
79     loop loop2
80     add bx, 6
81     pop cx
82     loop loop1
```

```

83 .exit
84 end

```

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

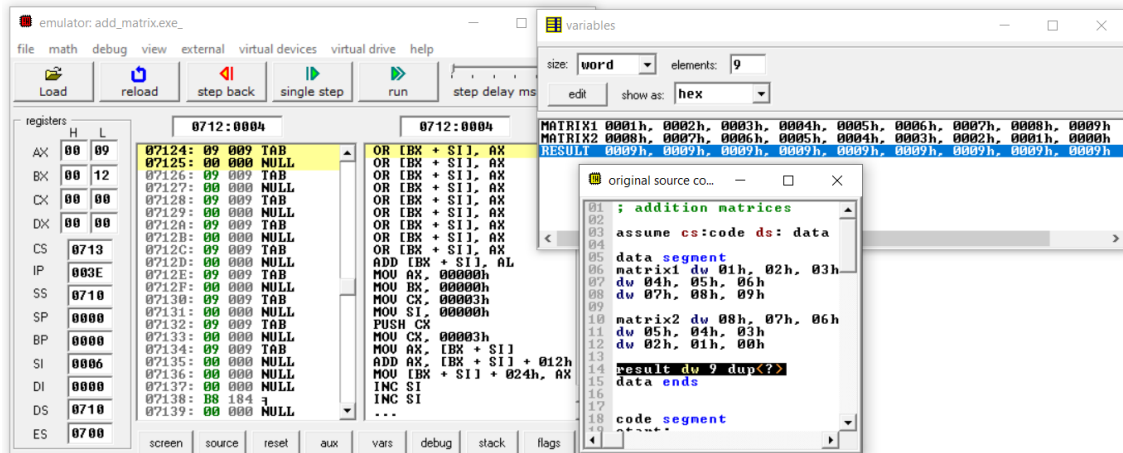


Figure 11: Addition of Matrices.

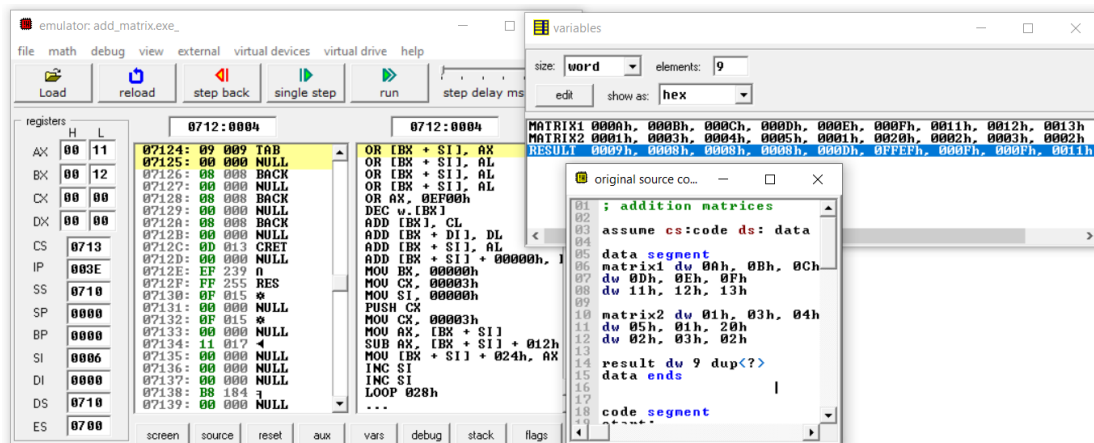


Figure 12: Subtraction of Matrices.

**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

```
1
2 ;Full Segment Definition
3 assume cs:code,ds:data
4 data segment
5 string1 db 99,12,56,45,36 ;storing unsorted array
6 data ends
7
8 code segment
9 start:
10     mov ax,data
11     mov ds,ax
12     mov ch,04h
13     up2:
14     mov cl,04h
15     lea si,string1
16     up1:
17     mov al,[si]
18     mov bl,[si+1]
19     cmp al,bl
20     jc down
21     mov dl,[si+1]
22     xchg [si],dl
23     mov [si+1],dl
24     down:
25     inc si
26     dec cl
27     jnz up1
```

```
28     dec ch
29     jnz up2
30 code ends
31 end start
32
33 ;Simplified Segment Definition
34 .model small
35 .data
36 string1 db 99,12,56,45,36 ;storing unsorted array
37 .code
38 .startup
39     mov ch,04h
40     up2:
41     mov cl,04h
42     lea si,string1
43     up1:
44     mov al,[si]
45     mov bl,[si+1]
46     cmp al,bl
47     jc down
48     mov dl,[si+1]
49     xchg [si],dl
50     mov [si+1],dl
51     down:
52     inc si
53     dec cl
54     jnz up1
55     dec ch
56     jnz up2
57 .exit
58 end
```

Listing 14: Descending Order sort

```
1
2 ; sort array in descending order
3 ;Full Segment Definition
4 assume cs:code ds: data
5 data segment
6     string1 db 99,12,56,45,36
7 data ends
8
9 code segment
10     start: mov ax,data
11     mov ds,ax
12     mov ch,04h
```

```
13     up2: mov cl,04h
14
15     lea si,string1
16
17     up1:mov al,[si]
18     mov bl,[si+1]
19     cmp al,bl
20     jnc down
21     mov dl,[si+1]
22     xchg [si],dl
23     mov [si+1],dl
24
25     down: inc si
26     dec cl
27     jnz up1
28     dec ch
29     jnz up2
30 code ends
31 end start
32
33 ; Simplified segment definition
34 .model small
35 .data
36     string1 db 99,12,56,45,36
37 .code
38 .startup
39     mov ch,04h
40     up2: mov cl,04h
41
42     lea si,string1
43
44     up1: mov al,[si]
45     mov bl,[si+1]
46     cmp al,bl
47     jnc down
48     mov dl,[si+1]
49     xchg [si],dl
50     mov [si+1],dl
51
52     down: inc si
53     dec cl
54     jnz up1
55     dec ch
56     jnz up2
57 .exit
58 end
```

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

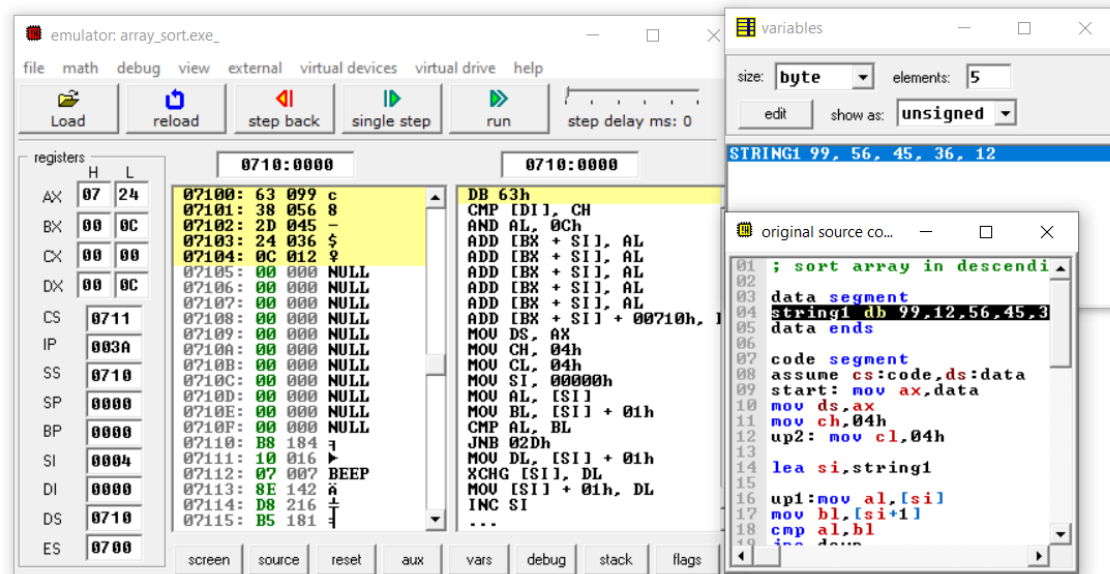


Figure 13: Sorting of array (Descending sort)

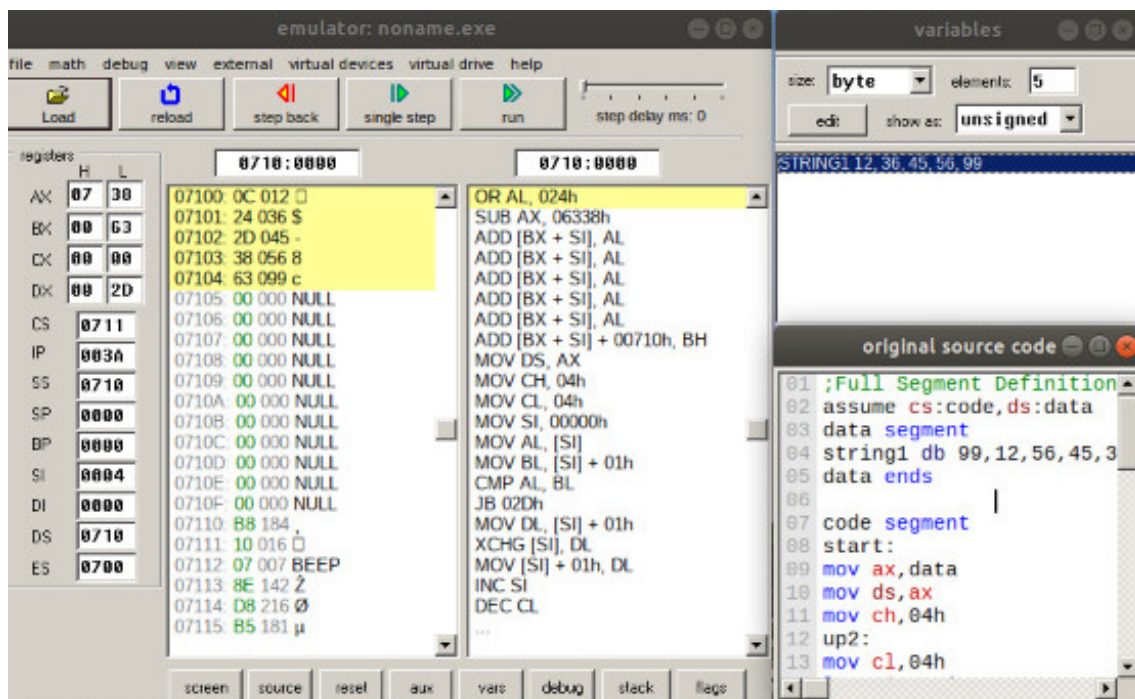


Figure 14: Sorting of array (Ascending sort)

**Conclusion:** 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

```
1  ;finding an element in array
2  ;if found display num available else num not available
3
4  ;Full Segment Definition
5  assume cs:code ds: data
6  data segment
7      arr1 db 12,1,14,99,15,22
8      cnt db 99
9      msg1 db 13,10,'Num available$',13,10 ;0dh,0ah
10     msg2 db 13,10,'Num not available$',13,10
11 data ends
12
13 code segment
14 start:
15     mov ax,data
16     mov ds, ax
17
18     mov cl,cnt
19     mov di,offset arr1
20     chk:
21     mov al,99 ;number to check in array
```

```
22     mov ah,[di]
23     cmp al,ah
24     jz disp1
25     inc di
26     dec cl
27     jnz chk
28
29     disp2: mov dx,offset msg2
30     mov ah,9
31     int 21h
32     .exit
33
34     disp1: mov dx,offset msg1
35     mov ah,9
36     int 21h
37 code ends
38 end start
39
40 ;Simplified segment definition
41 .model small
42 .data
43     arr1 db 12,1,14,99,15,22
44     cnt db 99
45     msg1 db 13,10,'Num available$',13,10 ;0dh,0ah
46     msg2 db 13,10,'Num not available$',13,10
47 .code
48 .startup
49 start:
50     mov cl,cnt
51     mov di,offset arr1
52     chk:
53     mov al,15 ;number to check in array
54     mov ah,[di]
55     cmp al,ah
56     jz disp1
57     inc di
58     dec cl
59     jnz chk
60
61     disp2: mov dx,offset msg2
62     mov ah,9
63     int 21h
64     .exit
65
66     disp1: mov dx,offset msg1
67     mov ah,9
68     int 21h
69 .exit
70 end
```

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

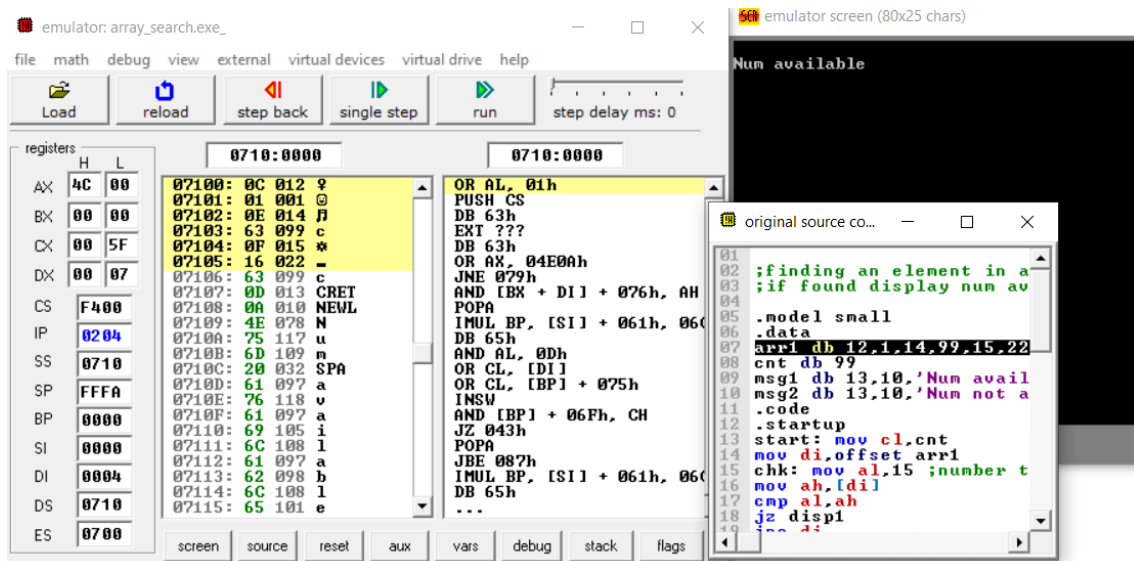


Figure 15: Linear Search in a array

**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

```
1  ;Full Segment Definition
2  assume cs:code ds:data
3  data segment
4      arr1 db -13, 21, 17,-50, -5, 99 ; array
5      negnos db 0 ; count of negative numbers
6      posnos db 0 ; count of positive numbers
7  data ends
8
9  code segment
10 start:
11     MOV AX,data
12     MOV ds, AX ;load data to DS
13     MOV SI, offset arr1 ; load address of array
14     MOV CL, 6 ; count of number os elements in array
15     l1:
16         MOV AL, [SI] ; array element in AL
17         CMP AL, 0 ; compare with 0
18         JG pos ; if greater go to pos
19         INC negnos ; increment count of - numbers
20         JMP here ; jmp to here label
21     pos:
22         INC posnos ; increment count of + numbers
23
24     here:
```

```
25         INC SI ; increment array pointer
26         LOOP l1 ; loop L1 till CL=0
27 code ends
28 end start
29
30 ;Simplified segment definition
31 .model small
32 .data
33     arr1 db -13, 21, 17, -50, -5, 99 ; array
34     negnos db 0 ; count of negative numbers
35     posnos db 0 ; count of positive numbers
36
37 .code
38 .startup
39 start:
40     MOV AX, data
41     MOV ds, AX ; load data to DS
42     MOV SI, offset arr1 ; load address of array
43     MOV CL, 6 ; count of number of elements in array
44     l1:
45         MOV AL, [SI] ; array element in AL
46         CMP AL, 0 ; compare with 0
47         JG pos ; if greater go to pos
48         INC negnos ; increment count of - numbers
49         JMP here ; jmp to here label
50     pos:
51         INC posnos ; increment count of + numbers
52
53     here:
54         INC SI ; increment array pointer
55         LOOP l1 ; loop L1 till CL=0
56 .exit
57 end
```

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

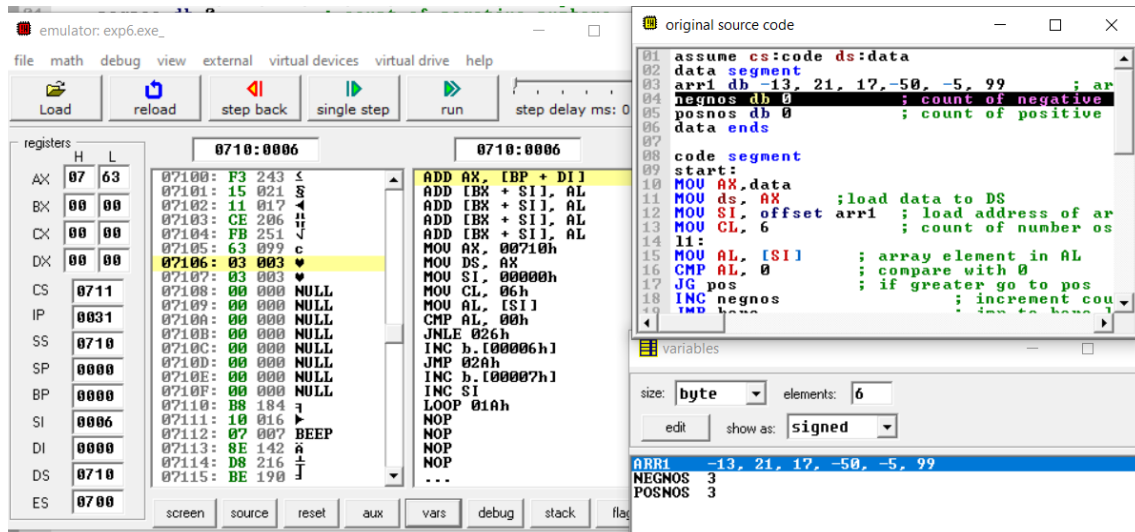


Figure 16: Count positive and negative numbers in a array

**Conclusion:** 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

```

1  ;Arithmetic Series
2  ;Full Segment Definition
3  assume CS:code DS:data
4  data segment
5      n db 9 ; number of elements in series
6      a db 2 ; first element
7      d db 3 ; difference
8      arr1 db 9 dup <0> ; STORES ap
9  data ends
10
11 code segment
12 start:
13     MOV CL, n ; store number of elements in cl
14     MOV CH,00
15     MOV AL, a ; move first element in AL
16
17     MOV DI, 6500h ; destination pointer
18     MOV [DI], AL ; store first element
19     INC DI ; increment pointer
20     DEC CL ; decrement count
21     MOV BL, d ; move difference in BL
22
23     LEA SI, arr1 ; load arr1 address in si

```

```

24     MOV [SI], AL ; store first value
25     INC SI
26
27 AP:  ADD AL, BL ; add difference to previous element
28     MOV [SI], AL ; store in arr1
29     MOV [DI], AL ; store at destination
30     INC DI ; increment destination pointer
31     INC SI
32
33     loop AP ; loop for n elements
34 code ends
35     end start

```

Listing 18: Arithmetic Series- Simplified Segment Definition

```

1  ; Arithmetic Series
2  ; Simplified Small Definition
3
4  .model small
5  .data
6      n db 9 ; number of elements in series
7      a db 2 ; first element
8      d db 3 ; difference
9      arr1 db 9 dup <0> ; STORES ap
10 .code
11 .startup
12     MOV CL, n ; store number of elements in cl
13     MOV CH, 00
14     MOV AL, a ; move first element in AL
15
16     MOV DI, 6500h ; destination pointer
17     MOV [DI], AL ; store first element
18     INC DI ; increment pointer
19     DEC CL ; decrement count
20     MOV BL, d ; move difference in BL
21
22     LEA SI, arr1 ; load arr1 address in si
23     MOV [SI], AL ; store first value
24     INC SI
25
26 AP:  ADD AL, BL ; add difference to previous element
27     MOV [SI], AL ; store in arr1
28     MOV [DI], AL ; store at destination
29     INC DI ; increment destination pointer
30     INC SI
31

```

```

32     loop AP ; loop for n elements
33 .exit
34 end

```

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

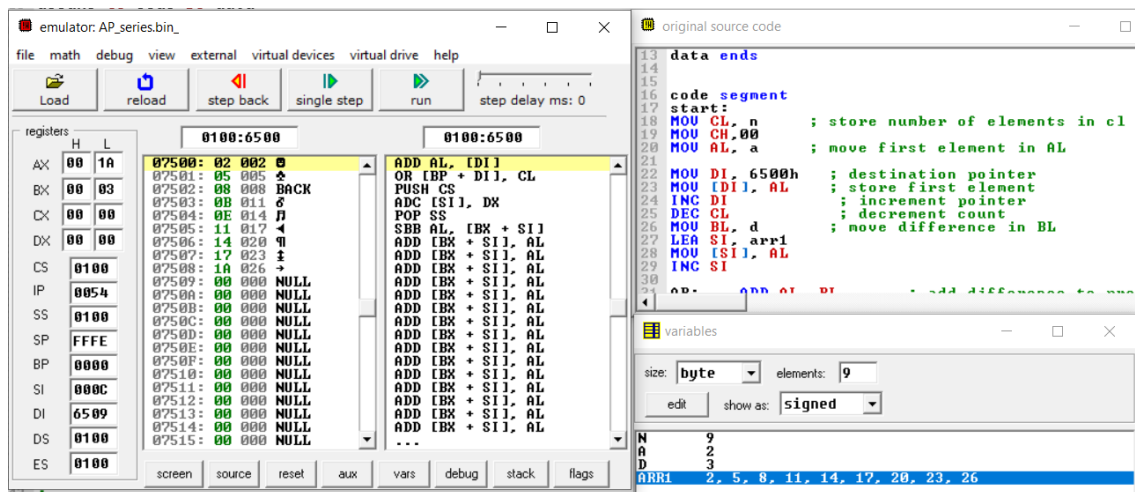


Figure 17: Arithmetic Series

**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

```
1  ;int21h
2  ;Full segment Definition
3
4  Assume cs:code ds:dsta
5  data segment
6      cnt db 7
7      star db 42,"$"
8      ;smiley db 1,"$" ; 0dh,0ah
9      ;heart db 3,"$"
10     new db 13,10,'$' ; new line
11 data ends
12
13 code segment
14 start:
15     MOV AX,data
16     MOV ds, AX ;load data to DS
17
18     mov cl,cnt ; rows count
19     mov bl, 1
20     loop1:
21         mov ch, bl ; load bl to count
22     loop2:
23         mov dx,offset star ;move address of smiley variable
24         mov ah,9 ;display character
25         int 21h
```

```

26
27     dec ch ; print till count 0
28     jnz loop2
29
30     mov dx,offset new ; print new line
31     mov ah,9 ; display
32     int 21h
33
34     inc bl ; increment bl
35     dec cl ; decrement cl
36     jnz loop1
37
38     mov dx,offset new ; move address of new line var
39     mov ah,9 ; display char
40     int 21h
41
42 code ends
43 end start
44
45
46 ; Simplified Segment Definition
47
48 .model small
49 .data
50     cnt db 7
51     star db 42,"$"
52     ;smiley db 1,"$" ;0dh,0ah
53     ;heart db 3,"$"
54     new db 13,10,'$' ; new line
55
56 .code
57 .startup
58 start:
59     mov cl,cnt ; rows count
60     mov bl, 1
61 loop1:
62     mov ch, bl ; load bl to count
63 loop2:
64     mov dx,offset star ;move address of smiley variable
65     mov ah,9 ;display character
66     int 21h
67
68     dec ch ; print till count 0
69     jnz loop2
70
71     mov dx,offset new ; print new line
72     mov ah,9 ; display
73     int 21h
74

```



```

75     inc bl ; increment bl
76     dec cl ; decrement cl
77     jnz loop1
78
79     mov dx,offset new ; move address of new line var
80     mov ah,9 ; display char
81     int 21h
82
83     .exit
84     end

```

Listing 20: Right angled triangle Pattern using INT 10H

```

1  ;display of right angled triangle
2  ; Full Segment Definition
3
4  Assume cs:code ds:dsta
5  data segment
6      disp1 db "$"
7      limit db 1
8  data ends
9
10 code segment
11 start:
12     MOV AX,data
13     MOV ds, AX ;load data to DS
14     mov dl,5 ;column
15     mov dh,5 ;row
16     mov bh,0
17
18     l1:
19         mov ah,2 ;cursor position
20         int 10h
21
22         mov ah,9 ;display star
23         mov al,disp1
24         mov bl,1111101b ; white bg & light magenta color
25         mov cl, limit ; times the disp1 will be displayed
26         int 10h
27
28         inc dh
29         inc limit ; increment limit
30         cmp limit,15 ; till 15 rows
31         jne l1
32     code ends
33 end start

```

```
34
35 ; Simplified Segment Definition
36 .model small
37 .data
38 disp1 db "$"
39 limit db 1
40
41 .code
42 .startup
43
44 mov dl,5 ;column
45 mov dh,5 ;row
46 mov bh,0
47
48 l1:
49     mov ah,2 ;cursor position
50     int 10h
51
52     mov ah,9 ;display star
53     mov al,disp1
54     mov bl,11111101b ; white bg & light magenta color
55     mov cl, limit ; times the disp1 will be displayed
56     int 10h
57
58     inc dh
59     inc limit ; increment limit
60     cmp limit,15 ; till 15 rows
61     jne l1
62 end
63
64 .exit
```

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



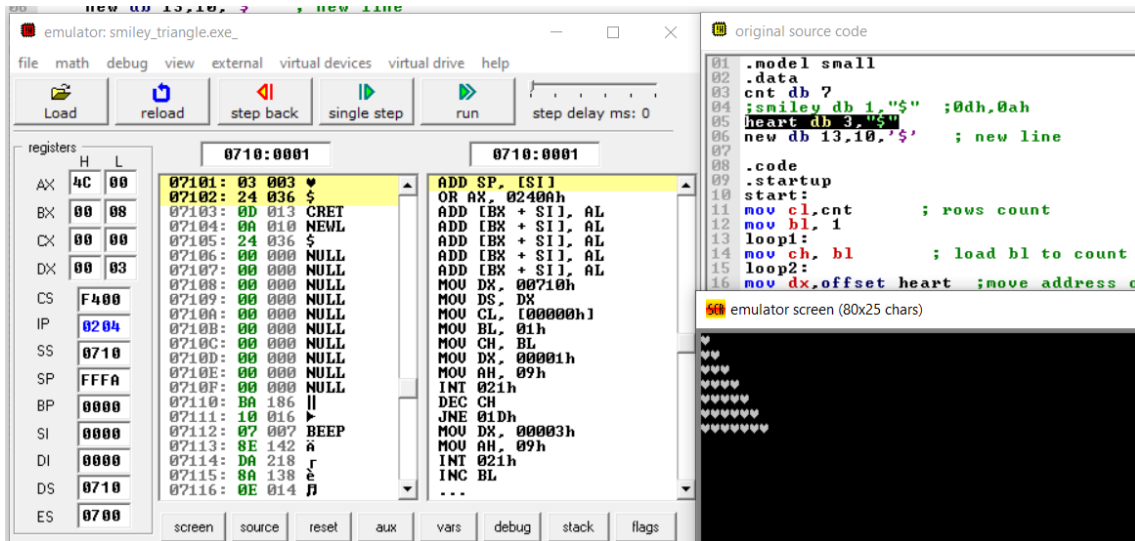


Figure 20: Heart Pattern using INT 21H

**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). These digits are displayed one after other using INT 21H.

**Code:**

Listing 21: Area of Rectangle - Full Segment Definition

```
1  ;area
2  ;Full segment Definition
3
4  Assume CS:Code, DS:Data
5
6  Data Segment
7      msg1 db "Enter the Length : $"
8      msg2 db 13, 10, "Enter the Breadth : $"
9      msg3 db 13, 10, "Area is : $"
10     area1 db ?
11     area2 db 00h
12 Data Ends
13
14 Code Segment
15
16 Start: MOV AX, Data
17         MOV DS, AX ; load data to DS
18
19         MOV AH, 09h
20         LEA DX, msg1 ; display msg1
21         INT 21h
22
23         MOV AH, 01h ; get single character input
24         INT 21h
25
26         SUB AL, 30h ; ASCII to number
```

```
27     MOV CL, AL ; move length to count
28     SUB CL, 01h
29
30     MOV AH, 09h
31     LEA DX, msg2 ; display msg2
32     INT 21h
33
34
35     MOV AH, 01h ; get single character input
36     INT 21h
37
38     SUB AL, 30h ; ASCII to number
39
40     MOV AH, 00h
41
42     MOV CH, 00h
43     MOV BX, AX ; move breadth to bc
44
45     Here: ADD AX, BX
46           DAA ; BCD adjust after addition
47           Loop Here ; add breadth cl times
48
49     disp: MOV AH, AL
50           MOV CL, 04h
51           SHL AL, CL
52           SHR AL, CL
53           SHR AH, CL
54           ADD AX, 3030h
55
56           MOV area1, AL
57           MOV area2, AH
58
59           MOV AH, 09h
60           LEA DX, msg3 ; display msg3
61           INT 21h
62
63           MOV AH, 02h
64           MOV DL, area2 ; display area2 tens place
65           MOV DH, 00h
66           INT 21h
67
68           MOV AH, 02h
69           MOV DL, area1 ; display area1 ones place
70           MOV DH, 00h
71           INT 21h
72     Code Ends
73     End Start
```

Listing 22: Area of Rectangle - Simplified segment Definition

```
1  .model small
2  .data
3      msg1 db "Enter the Length : $"
4      msg2 db 13, 10, "Enter the Breadth : $"
5      msg3 db 13, 10, "Area is : $"
6      msg4 db 13,10,'21$', 13,10
7  .code
8  .startup
9  start:
10     mov dx,offset msg1
11     mov ah,9 ; display msg1
12     int 21h
13
14     mov ah, 1h ; take input
15     int 21h
16
17     sub al, 30h ; ASCII to decimal conversion
18     mov bl, al
19
20     mov dx,offset msg2
21     mov ah,9 ; display msg2
22     int 21h
23
24     mov ah, 1h ; take input
25     int 21h
26
27     sub al, 30h ; ASCII to decimal conversion
28
29     mul bl ; multiply length
30     mov bl, al ; store in BL
31
32     mov dx,offset msg3
33     mov ah,9 ; display msg3
34     int 21h
35
36     mov al, bl
37     mov ah, 0
38     mov bl, 10
39     div bl ; divide al by 10
40     mov bl, ah ; get ones place
41     add al, 30h
42
43
44     mov offset msg4[2], al
45     add ah, 30h
46     mov offset msg4[3], ah
```

```

47     mov dx,offset msg4
48     mov ah,9
49     int 21h
50 .exit
51 end

```

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

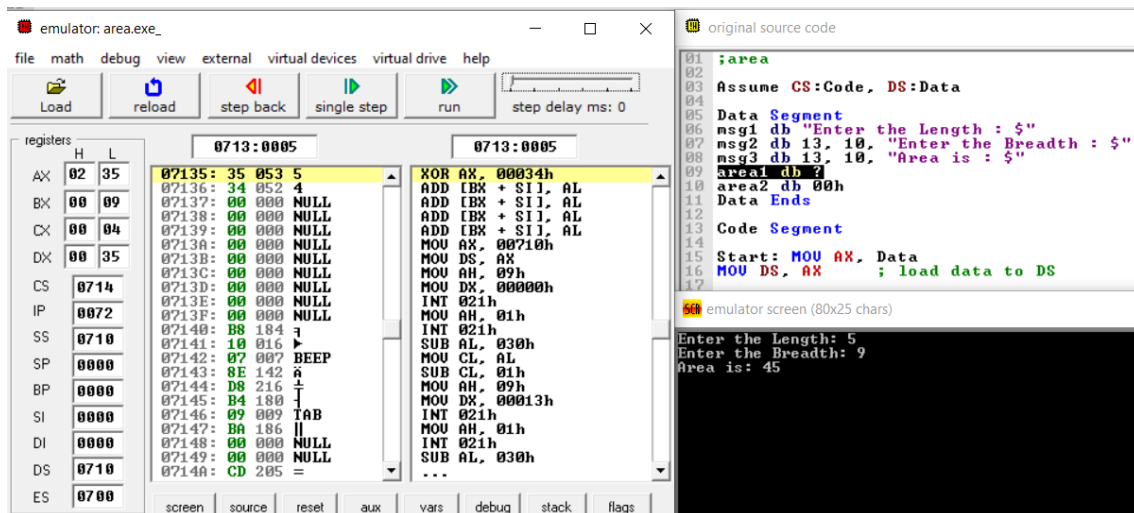


Figure 21: Area of rectangle - fsd

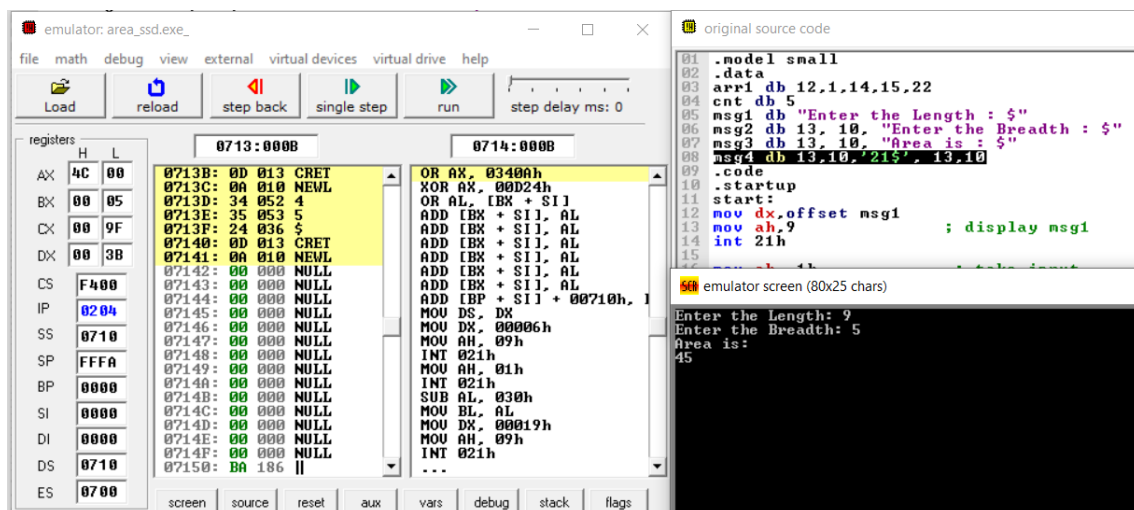


Figure 22: Area of rectangle - ssd



**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

```
1
2 ; ssd
3
4 .model small
5 .data
6 msg1 db "Enter the string:$"
7 msg2 db 13,10,"The reversed string is:$"
8 str db ?
9
10 .code
11 .startup
12
13 mov ah,9h
14 lea dx,msg1 ;display msg1
15 int 21h
16
17 lea si,str ; load address off str1
18 mov cl,1h ; counter value 1
19 jmp scanstring
20
21 scanstring:
22     mov ah, 01h
23     int 21h ; take single character input
24
25     cmp al,13 ; check if pressed enter
26     je display ; if pressed go to display
27     mov [si],al ; else continue scanning
28     inc si
```

```

29     inc cl ; increment count
30     jmp scanstring
31
32 display:
33     lea dx,msg2
34     mov ah,9h
35     int 21h ; display msg2
36     jmp reverse
37
38 reverse:
39     mov al,[si]
40
41     mov dl,al
42     mov dh,0
43     mov ah,02h ; display character
44     int 21h
45
46     dec si ; increment array pointer
47     dec cl ; decrement counter value
48     cmp cl,0 ; check if CX=0
49     jne reverse ; if not 0 continue reverse loop
50 .exit
51 end

```

Listing 24: Reverse string - Full segment Definition

```

1
2 ;Get a string as an input from the user and reverse it and display on the screen.
3
4 Assume CS:Code, DS:Data, ES:Extra
5 Data Segment
6     msg1 db "Enter a String : $"
7     msg2 db 0Ah, 0Dh, "The reversed string is : $"
8     str db 08 dup('$')
9 Data Ends
10
11 Extra Segment
12     rev_str db 07 dup(00), '$' ; For Reverse string
13 Extra Ends
14
15 Code Segment
16
17 Start: mov ax, Data
18         mov ds, ax ; Load data to DS
19         mov ax, Extra
20         mov es, ax ; Load extra to ES

```

```
21
22     mov ah, 09h
23     lea dx, msg1 ; display msg1
24     int 21h
25
26     lea dx, str
27     mov ah, 0Ah ; take string input
28     mov data, 04h
29     int 21h
30
31     MOV al, str[01h] ; AL will now contain the elements.
32     MOV ah, 00h ; Now AX would contain the count.
33     mov cx, ax ; Transferring the count to the CX register.
34
35     lea si, str + 02h ;load effective address
36     lea di, rev_str + 06h
37
38     Here:
39     cld ;clear direction flag
40     lodsb ;read from source string
41     std ;set direction flag
42     stosb ;write into destination string cld
43     loop here
44
45     MOV al, str[01h] ; AL will now contain the elements.
46     MOV ah, 00h ; Now AX would contain the count.
47     mov cx, ax ; Transferring the count to the CX register.
48     inc cx
49
50     mov si, di
51     inc si
52
53     lea di, str ; effective address of string
54
55     mov ax, data ; move data to ES
56     mov es, ax
57
58     mov ax, extra ; move extra to ES
59     mov ds, ax
60
61     cld ; clear direction flag
62
63     rep movsb ; move byte from string 1 to string2 till CX =0
64
65     mov ax, data ; move data to DS
66     mov ds, ax
67
68     mov ah, 09h
69     lea dx, msg2 ; display msg2
```

```

70      int 21h
71
72      mov ah, 09h
73      lea dx, str ; display reverse string
74      int 21h
75
76 Code Ends
77      End Start

```

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

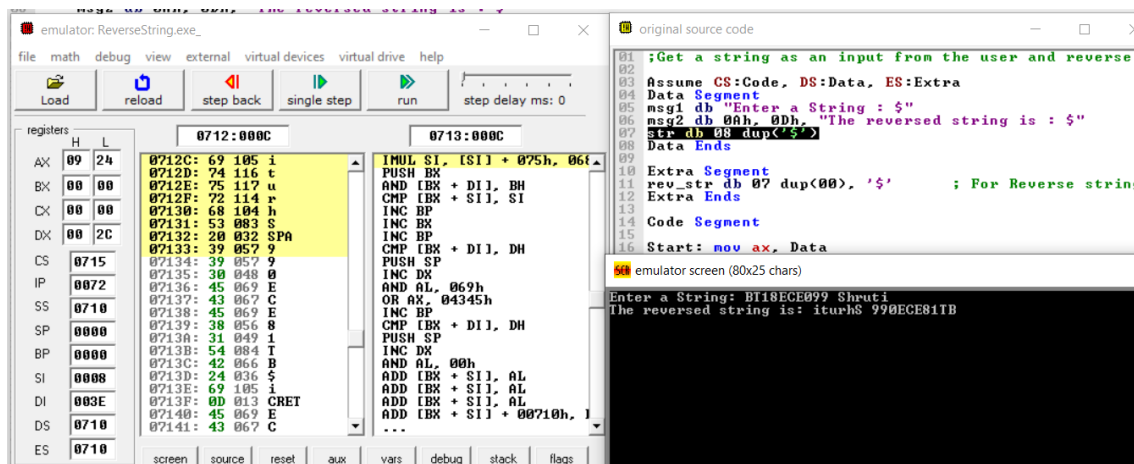


Figure 23: Reverse String

**Conclusion:** The experiment 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

```
1  ASSUME CS:CODE, DS:DATA
2
3  DATA SEGMENT
4      PORTA EQU 00H
5      PORTB EQU 02H
6      PORTC EQU 04H
7      PORT_CON EQU 06H
8  DATA ENDS
9
10 CODE SEGMENT
```

```

11      MOV AX,DATA
12      MOV DS, AX
13
14      ORG 0000H
15  START:
16
17      MOV DX, PORT_CON
18      MOV AL, 10000000B ; port C (output), port A (output) in mode 0 and PORT ...
          B (INPUT) in mode 0
19      OUT DX, AL
20      JMP XX
21  XX:
22      MOV AL, 0FFH
23      MOV DX, PORTA
24      OUT DX,AL
25      CALL DELAY
26
27      MOV AL, 80H ; 8
28      MOV DX, PORTA
29      OUT DX,AL
30      CALL DELAY
31
32      MOV AL, 0COH ; 0
33      MOV DX, PORTA
34      OUT DX,AL
35      CALL DELAY
36
37      MOV AL, 80H ; 8
38      MOV DX, PORTA
39      OUT DX,AL
40      CALL DELAY
41
42      MOV AL, 82H ; 6
43      MOV DX, PORTA
44      OUT DX,AL
45      CALL DELAY
46
47      MOV AL, 0C7H ; L
48      MOV DX, PORTA
49      OUT DX,AL
50      CALL DELAY
51
52      MOV AL, 88H ; A
53      MOV DX, PORTA
54      OUT DX,AL
55      CALL DELAY
56
57      MOV AL,83H ; B
58      MOV DX, PORTA

```

```

59  OUT DX,AL
60  CALL DELAY
61
62  JMP XX
63
64  DELAY PROC NEAR
65      MOV CX,0DF36H ; Delay
66      loop1:loop loop1
67      RET
68  DELAY ENDP
69
70
71  CODE ENDS
72  END

```

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

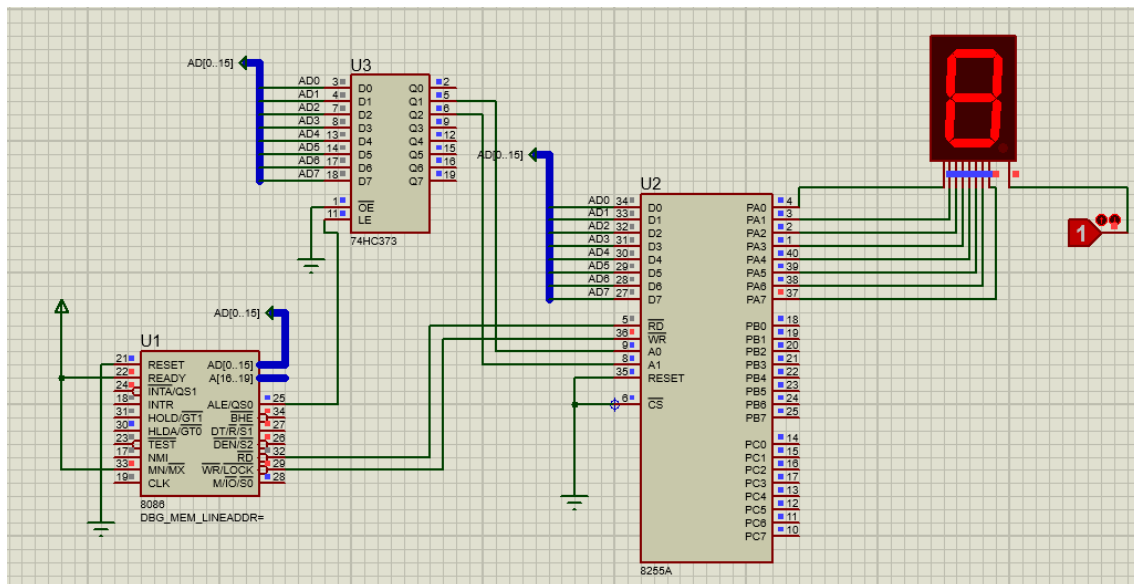


Figure 24: 8255 with 8086

**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

```
1  assume cs:code, ds:data
2  data segment
3      porta equ 00H
4      portb equ 02H
5      portc equ 04H
6      cwr equ 06H
7  data ends
8
9  code segment
10     mov ax, data
11     mov ds, ax
12     org 0000H
13
14  start:
15     mov al, 91H
16     out cwr, al
17     l1:
18         mov al, 00H
19         in al, porta ; switch input
20         out portb, al ; output to led
21     jmp l1
22 code ends
23 end
```

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

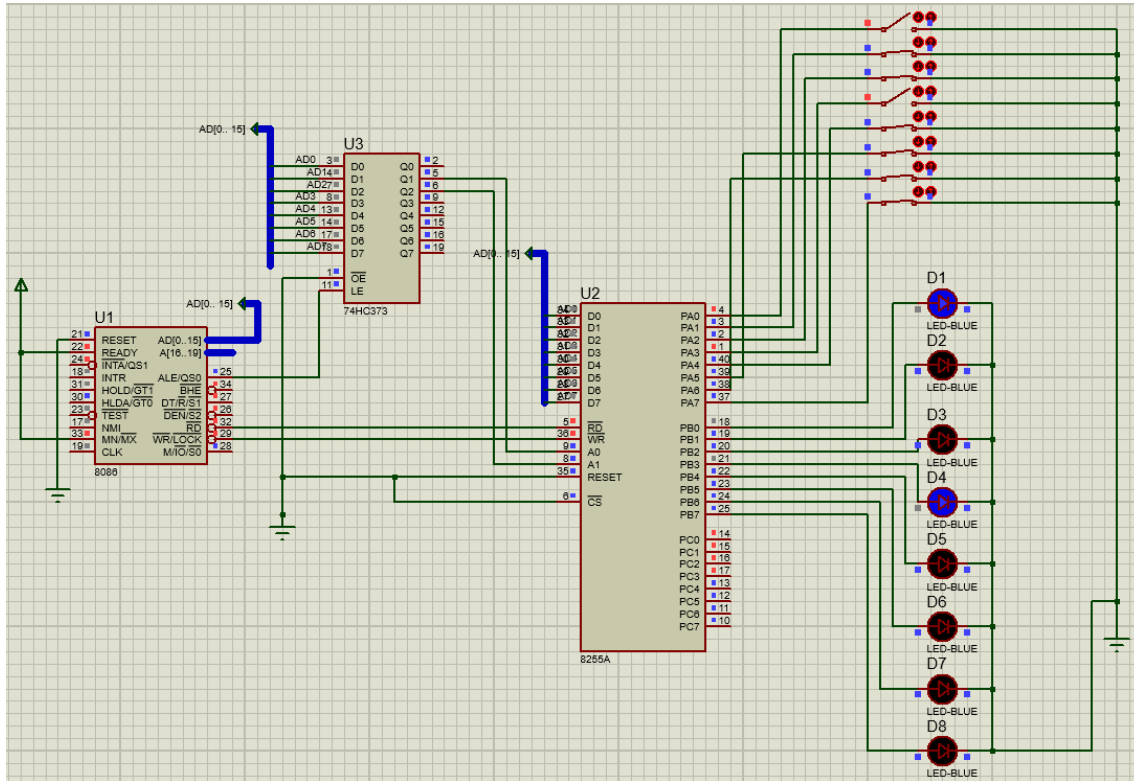


Figure 25: LED with Switches

**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:

$$T = 1\text{ms}$$

$$f = 1/T = 1\text{KHz}$$

Input clock frequency to 8053 = 2Mhz

Count for register = clock frequency / square wave frequency

$$= 2\text{Mhz} / 1\text{KHz}$$

$$= 2000 = 070D \text{ H}$$

**Code:**

Listing 27: Square wave of 1ms using 8253

```

1  ASSUME CS:CODE, DS:DATA
2
3  DATA SEGMENT
4      COUNTER0 EQU 69H
5      COUNTER1 EQU 6BH
6      COUNTER2 EQU 6DH
7      CWR EQU 6FH
8  DATA ENDS
9
10 CODE SEGMENT
11     MOV AX, DATA
12     MOV DS, AX
13     ORG 0000H
14
15 START:
16     ; --> Square Wave :: MOD 3 of 8253
17     MOV AL, 10110111b ; 10 [counter 2], 11[ 2 byte data], 011 [ with mod 3], ...
        1 [BCD]
```

```

18      OUT CWR, AL ; configure the counter 2 of 8253
19
20      MOV AL, 0D0h
21      OUT COUNTER2, AL ; Send first byte of Counter 2
22      MOV AL, 07h
23      OUT COUNTER2, AL ; Send second byte of Counter 2
24
25  ENDLESS:
26      JMP ENDLESS
27  CODE ENDS
28      END START

```

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

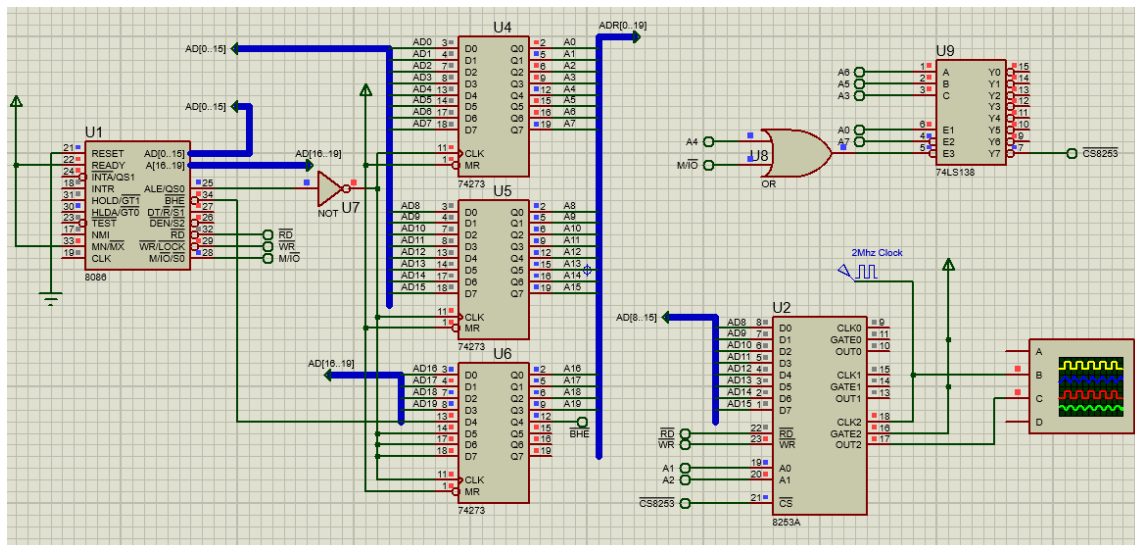


Figure 26: 8253 with 8086

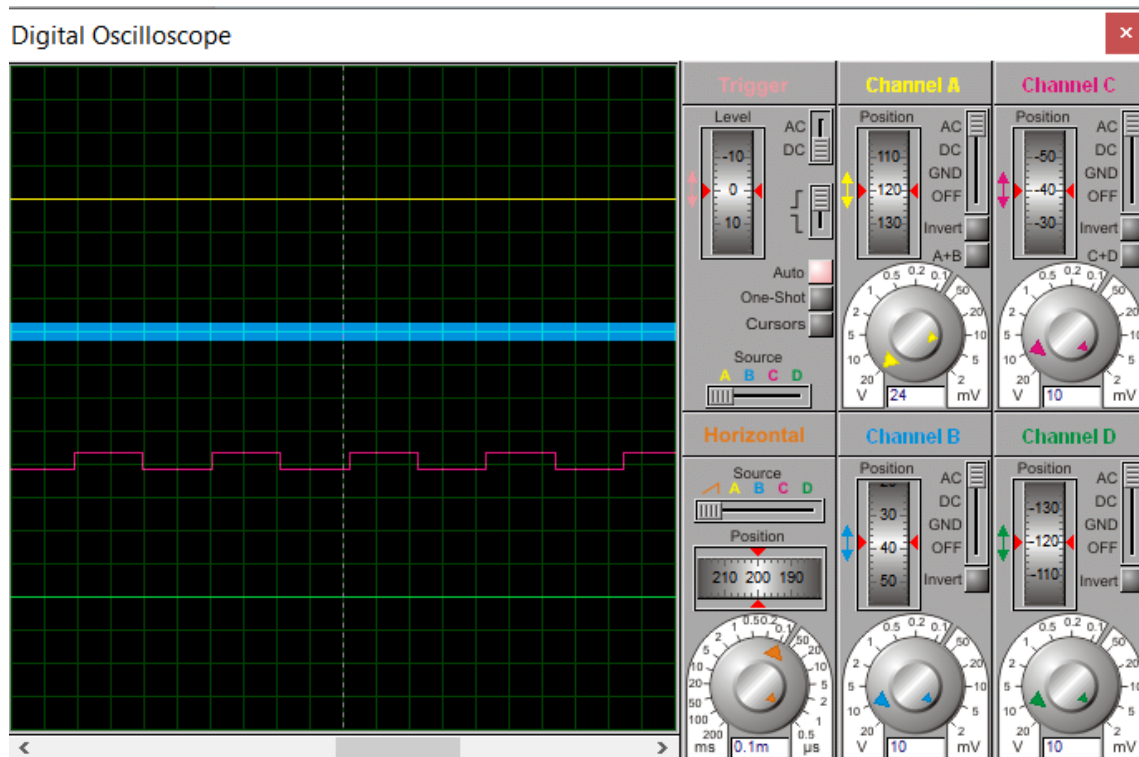


Figure 27: Oscilloscope [B - 2us input clock, C - 1ms square wave]

**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  $27 \mu s$  and can also calculate square-root in  $35 \mu s$ . It follows IEEE floating point standards.

Area of right angled triangle =  $0.5 * \text{base} * \text{height}$

**Code:**

Listing 28: Area of triangle

```

1  Data Segment
2      base dd 9.9
3      height dd 6.2
4      half dd 0.5
5      area dt ?
6      areamsg db "Area of right angled triangle is: $"
7      a dw ?
8      b dt 100.00
9      dot db ".$"
10     msg4 db 10d,13d,"$"
11 Data Ends
12
13
14 Code Segment
15     Assume CS:Code, DS:Data
16 Start:
17
18     scall macro xx, yy ;macro to display
19         lea dx,xx
20         mov ah,yy
21         int 21h
22     endm
23
24     displayreal macro xx,yy
25         fld xx

```

```
26      scall msg4,09h
27      scall yy,09h
28
29      fld b ;load 100.00
30      fmulp st[1],st ;multiply by 100.00
31      fbstp xx ;convert to BCD
32
33      mov ax,word ptr xx
34      call display4digit ;display ax contents
35  endm
36
37  mov ax, Data
38  mov ds, ax
39
40  finit
41  fld base ; load base
42  fld height ; load height
43  fmul ; multiply base and height
44  fld half ; load 0.5
45  fmul ; multiply half to result
46  fstp area ; store result in area
47
48  displayreal area, areamsg
49
50
51  mov ah, 4ch ; terminate program
52  int 21h
53
54
55  display4digit proc
56      mov a,ax ;ax has 4 digits to diaplay
57      mov dl,ah ;to display ah first
58
59      mov ch,02h ;count of digits
60      rol dl,01
61      rol dl,01
62      rol dl,01
63      rol dl,01
64  loop3: and dl,0fh
65      add dl,30h
66
67      mov ah,02h
68      int 21h
69
70      mov ax,a ;restore ax
71      mov dl,ah
72
73      dec ch
74      jnz loop3
```

```
75
76     scall dot,09h ;display dot
77
78     mov ax,a
79     mov dl,al ;to display al
80
81     mov ch,02h ;count of digits
82     rol dl,01
83     rol dl,01
84     rol dl,01
85     rol dl,01
86
87 loop4: and dl,0fh ;anding to extract last digit
88         add dl,30h ;convert to ascii-hex
89
90     mov ah,02h ;display dl contents
91     int 21h
92
93     mov ax,a
94     mov dl,al
95
96     dec ch
97     jnz loop4
98 RET
99 display4digit Endp
100
101 Code Ends
102     End Start
```

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



```
D:\>area
Area of right angled triangle is: 30.69
```

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.