POA LAB EXPERIMENTS

Name	Junaid Girkar
Sap ID	60004190057
Division	A
Batch	A4
Branch	Computer Engineering
Semester	5
Subject	Processor Organization and Architecture
Subject Code	DJ19CEL502

Experiment List

EXPERIMENT - 1

AIM: To demonstrate Booth's Algorithm for multiplication of 2 signed binary numbers.

Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	01-10-2021	01-10-2021	

THEORY:

Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation in an efficient way, i.e., less number of additions/subtractions required. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^(k+1) to 2^m.

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values *A* and *S* to a product *P*, then performing a rightward arithmetic shift on *P*. Let **m** and **r** be the multiplicand and multiplier, respectively; and let *x* and *y* represent the number of bits in **m** and **r**.

1. Determine the values of *A* and *S*, and the initial value of *P*. All of these numbers should have a length equal to (*x* + *y* + 1).
    1. A: Fill the most significant (leftmost) bits with the value of **m**. Fill the remaining (*y* + 1) bits with zeros.
    2. S: Fill the most significant bits with the value of (−**m**) in two's complement notation. Fill the remaining (*y* + 1) bits with zeros.
    3. P: Fill the most significant *x* bits with zeros. To the right of this, append the value of **r**. Fill the least significant (rightmost) bit with a zero.
2. Determine the two least significant (rightmost) bits of *P*.
    1. If they are 01, find the value of *P* + *A*. Ignore any overflow.
    2. If they are 10, find the value of *P* + *S*. Ignore any overflow.
    3. If they are 00, do nothing. Use *P* directly in the next step.
    4. If they are 11, do nothing. Use *P* directly in the next step.

3. Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let *P* now equal this new value.
4. Repeat steps 2 and 3 until they have been done *y* times.
5. Drop the least significant (rightmost) bit from *P*. This is the product of **m** and **r**.

TIME COMPLEXITY: O(n) * (complexity_of_addition + complexity_of_shift)

EXAMPLE: Multiplication of 7 and 3

Q_n	Q_{n+1}	M = (0111) M' + 1 = (1001) & Operation	AC	Q	Q_{n+1}	SC
1	0	Initial	0000	0011	0	4
		Subtract (M' + 1)	1001			
			1001			
		Perform Arithmetic Right Shift operations (ashr)	1100	1001	1	3
1	1	Perform Arithmetic Right Shift operations (ashr)	1110	0100	1	2
0	**1**	Addition (A + M)	0111			
			0101	0100		
		Perform Arithmetic right shift operation	0010	1010	0	1
0	0	Perform Arithmetic right shift operation	**0001**	**0101**	0	0

Final Answer = $(0001\ 0101)_2$
= $(21)_{10}$

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void rightShift();

int main()
```

```
{
   printf("\n");
   printf("BOOTH's Algorithm\n");
   printf("\n");
   printf("Enter two numbers that are to be multiplied : ");//taking two numbers as inputs
   int a,b;
   scanf("%d %d",&a,&b);
   int ap=a,bp=b;
   if(ap<0) // Negetive values check
      ap*=-1;
   if(bp<0) bp*=-1;
   if(bp>ap) //taking greater VALUE as multiplicand
   {
      ap=bp+ap-(bp=ap);
      a=b+a-(b=a);
   }
   int t1=ap,t2=bp;
   int ab[35]={};
   int bb[35]={};
   int i=0;
   while(t1>0)
   {
      ab[i]=t1%2;
      i++;
      t1/=2;
   }
   ab[i]=0;
   int j=0;
   while(t2>0)
   {
      bb[j]=t2%2;
      j++;
      t2/=2;
   }
   while(j<=i) //equating bits to the previous(ab) binary number(ab will either be larger or equal to
bb).
      bb[j++]=0;
   int nb=i+1; //nb is number of bits
   i=0;j=0;
   while(i<nb/2) //converting VALUES to binary
   {
      ab[i]=ab[nb-i-1]+ab[i]-(ab[nb-i-1]=ab[i]);
      i++;
   }
   i=0;
   while(i<nb/2) { bb[i]=bb[nb-i-1]+bb[i]-(bb[nb-i-1]=bb[i]); i++; } int x[35]={0}; int y[35]={0}; i=0; if(a>=0)
//taking actual binary numbers
```

```
{ //x is multiplicand and y is multiplier
   while(i<nb)
   x[i]=ab[i+++1];
}
else //2's complimant
{
   while(i<nb) { if(ab[i]==0) x[i]=1; else x[i]=0; i++; } i=1; x[nb-i]++; while(x[nb-i]==2) { x[nb-i]=0; i++;
x[nb-i]++; } } i=0; if(b>=0)
{
   while(i<nb)
      y[i]=bb[i+++1];
}
else //2's complimant
{
   while(i<nb) { if(bb[i]==0) y[i]=1; else y[i]=0; i++; } i=1; y[nb-i]++; while(y[nb-i]==2) { y[nb-i]=0; i++;
y[nb-i]++; } } printf("\n"); //output starts here printf("Multiplicand (Q) %d -> ",a);
i=0;
printf("Multiplicand (Q) -> ");
while(i<nb) printf("%d",x[i++]); printf("\nMultiplier (M) %d -> ",b);
i=0;
while(i<nb)
   printf("%d",y[i++]);
printf("\n");
i=0;
int ym[35]={0}; //calculating -M
if(b<0)
{
   while(i<nb)
   ym[i]=bb[i+++1];
}
else
{
   while(i<nb) { if(bb[i]==0) ym[i]=1; else ym[i]=0; i++; } i=1; ym[nb-i]++; while(ym[nb-i]==2) {
ym[nb-i]=0; i++; ym[nb-i]++; } } printf("we use -(M) i.e. %d -> ",-b);
i=0;
while(i<nb)
   printf("%d",ym[i++]);
printf("\n");
int q0=0;
int p[35]={0}; //p here is value that is stored in accumulator. initially set to zero.
int steps=nb;
printf("\n");
printf("n\t");
i=0;
while(i<nb)
{
   if(i*2==nb || i*2==nb-1)
```

```
    printf("A");
    else
    printf(" ");
    i++;
}
printf(" ");
i=0;
while(i<nb)
{
    if(i*2==nb || i*2==nb-1)
        printf("Q\t");
    else
        printf(" ");
    i++;
}
printf(" Q-1");
printf("\n");
j=0;

while(steps--) //counting down steps.
{
    printf("%d        ",j++);
    i=0;
    while(i<nb)
        printf("%d",p[i++]);
    printf(" ");
    i=0;
    while(i<nb)
        printf("%d",x[i++]);
    printf(" ");
    printf("%d\n",q0);
    if(x[nb-1]==0 && q0==0) //0-0 condition
    {
        q0=x[nb-1];
        rightShift(p,x,nb);
    }
    else if(x[nb-1]==0 && q0==1) //0-1 condition
    {
        printf("    A + M ");
        i=0;
        while(i<nb)
            printf("%d",y[i++]);
        i=0;
        while(i<nb)
        {
            p[nb-i-1]+=y[nb-i-1];
            if(p[nb-i-1]==2)
```

```
        {
            p[nb-i-1]=0;
            if(nb-i-1!=0)
            p[nb-i-2]++;
        }
        if(p[nb-i-1]==3)
        {
            p[nb-i-1]=1;
            if(nb-i-1!=0)
            p[nb-i-2]++;
        }
        i++;
    }
    printf("\n         ");
    i=0;
    while(i<nb)
        printf("%d",p[i++]);
    printf("\n");
    q0=x[nb-1];
    rightShift(p,x,nb);
}
else if(x[nb-1]==1 && q0==0) //1-0 condition
{
    printf("    A - M ");
    i=0;
    while(i<nb)
        printf("%d",ym[i++]);
    i=0;
    while(i<nb)
    {
        p[nb-i-1]+=ym[nb-i-1];
        if(p[nb-i-1]==2)
        {
            p[nb-i-1]=0;
            if(nb-i-1!=0)
            p[nb-i-2]++;
        }
        if(p[nb-i-1]==3)
        {
            p[nb-i-1]=1;
            if(nb-i-1!=0)
            p[nb-i-2]++;
        }
        i++;
    }
    printf("\n         ");
    i=0;
```

```
        while(i<nb)
            printf("%d",p[i++]);
        printf("\n");
        q0=x[nb-1];
        rightShift(p,x,nb);
      }
      else if(x[nb-1]==1 && q0==1) //1-1 condition
      {
        q0=x[nb-1];
        rightShift(p,x,nb);
      }
   }
   printf("%d        ",j);
   i=0;
   while(i<nb)
      printf("%d",p[i++]);
   printf(" ");
   i=0;
   while(i<nb)
      printf("%d",x[i++]);
   printf(" ");
   printf("%d\n",q0);
   printf("\n");

   printf("Final Product in signed binary number is : ");
   i=0;
   while(i<nb)
      printf("%d",p[i++]);
   i=0;
   printf(" ");
   while(i<nb)
      printf("%d",x[i++]);
   printf("\n\n");
   return 0;
}

void rightShift(int p[],int x[],int nb)
{
   int i=0;
   while(nb-i-1)
   {
      x[nb-i-1]=x[nb-i-2];
      i++;
   }
   x[0]=p[nb-1];
   i=0;
   while(nb-i-1)
```

```
  {
    p[nb-i-1]=p[nb-i-2];
    i++;
  }
}
```

OUTPUT:

```
BOOTH's Algorithm

Enter two numbers that are to be multiplied : 20 35

Multiplicand (Q) -> 0100011
Multiplier (M) 20 -> 0010100
we use -(M) i.e. -20 -> 1101100

n          A       Q        Q-1
0         0000000 0100011 0
    A - M 1101100
          1101100
1         1110110 0010001 1
2         1111011 0001000 1
    A + M 0010100
          0001111
3         0000111 1000100 0
4         0000011 1100010 0
5         0000001 1110001 0
    A - M 1101100
          1101101
6         1110110 1111000 1
    A + M 0010100
          0001010
7         0000101 0111100 0

Final Product in signed binary number is : 0000101 0111100


...Program finished with exit code 0
Press ENTER to exit console.
```

EXPERIMENT - 2

AIM: To demonstrate the Restoring Division Algorithm for 2 unsigned binary numbers.

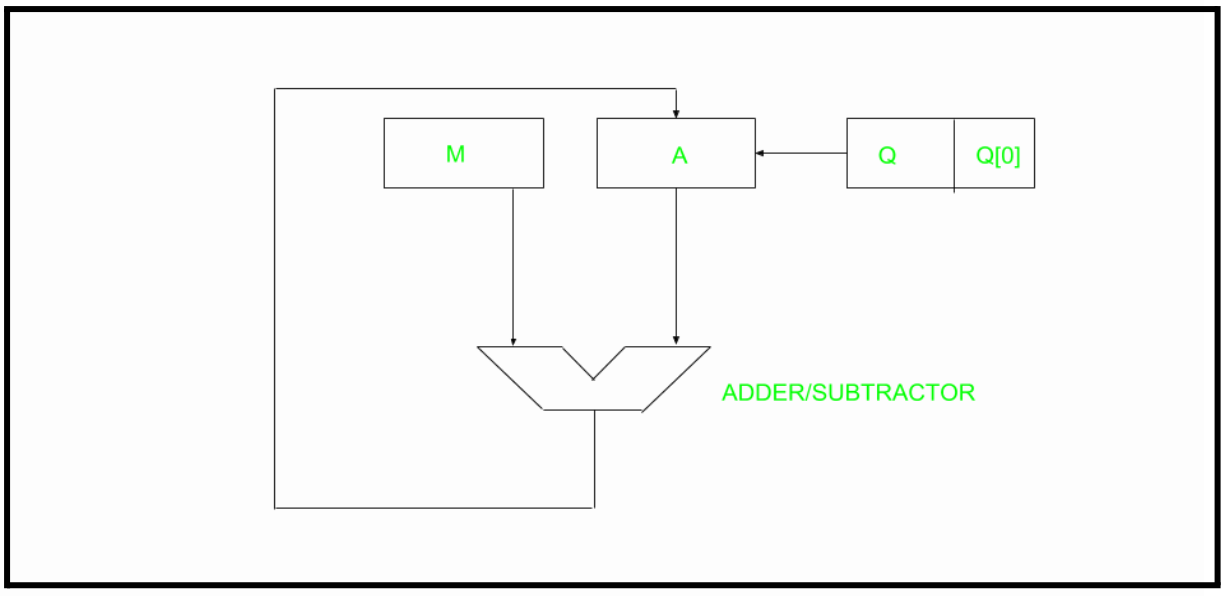
Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	08-10-2021	08-10-2021	

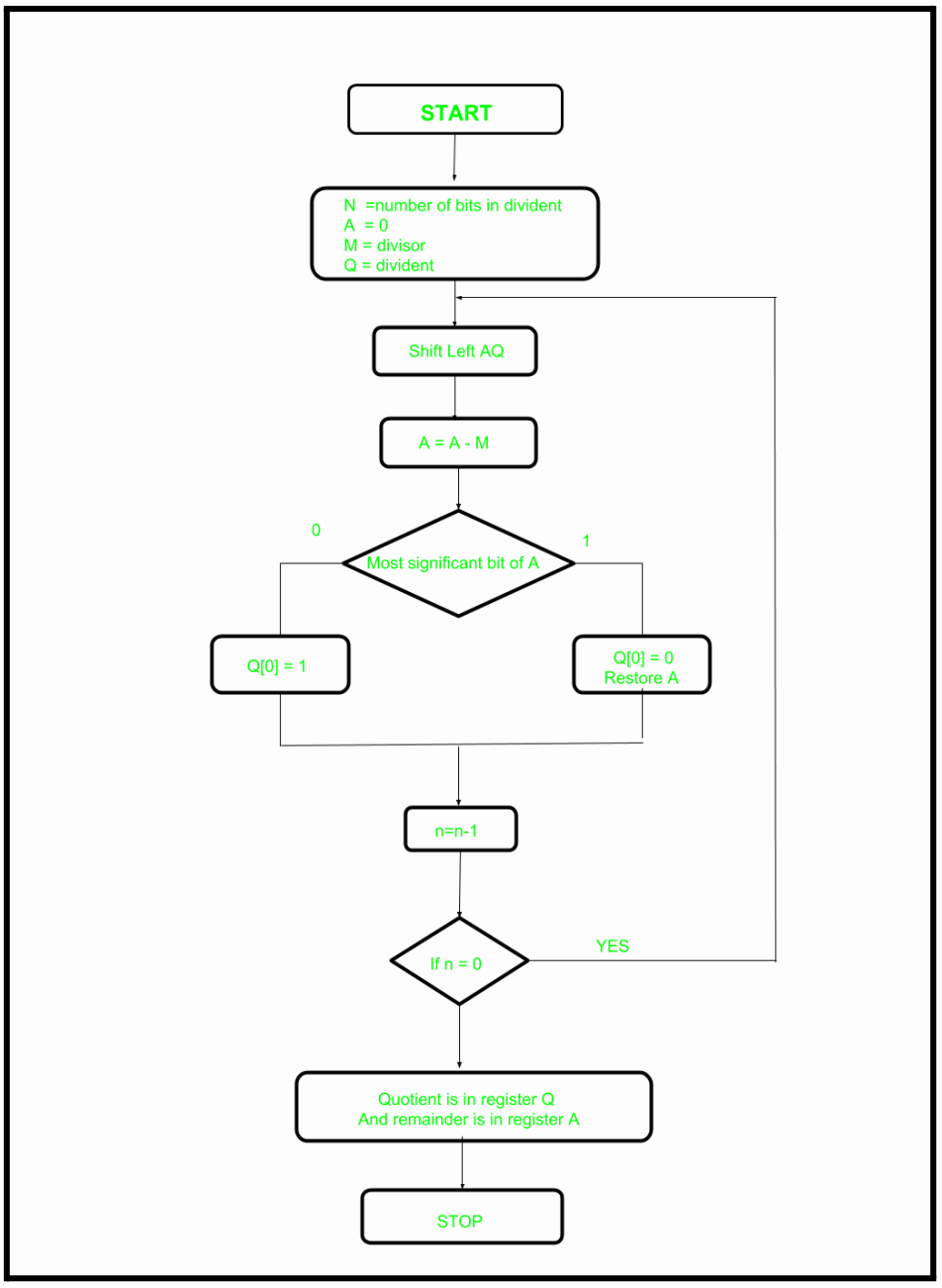
THEORY:

A division algorithm provides a quotient and a remainder when we divide two number. They are generally of two type **slow algorithm and fast algorithm**. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm and under fast comes Newton–Raphson and Goldschmidt.

In this article, will be performing restoring algorithm for unsigned integer. Restoring term is due to fact that value of register A is restored after each iteration.



Here, register Q contain quotient and register A contain remainder. Here, n-bit dividend is loaded in Q and divisor is loaded in M. Value of Register is initially kept 0 and this is the register whose value is restored during iteration due to which it is named Restoring.



Let's pick the step involved:

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted right as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat fro step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder

Examples:

Perform Division Restoring Algorithm
Dividend = 11

Divisor = 3

n	M	A	Q	Operation
4	00011	00000	1011	initialize
	00011	00001	011_	shift left AQ
	00011	11110	011_	A=A-M
	00011	00001	0110	Q[0]=0 And restore A
3	00011	00010	110_	shift left AQ
	00011	11111	110_	A=A-M
	00011	00010	1100	Q[0]=0
2	00011	00101	100_	shift left AQ
	00011	00010	100_	A=A-M
	00011	00010	1001	Q[0]=1
1	00011	00101	001_	shift left AQ
	00011	00010	001_	A=A-M
	00011	00010	0011	Q[0]=1

Remember to restore the value of A, most significant bit of A is 1. As that register Q contains the quotient, i.e. 3 and register A contains remainder 2.

CODE:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
```

```
int getsize(int x)
{
int c;
if(x<=1)
c = 2;
else if(x < 4)
c = 2;
else if(x< 8)
c = 3;
else if(x< 16)
c = 4;
else if(x< 32)
c = 5;
else if(x< 64)
c = 6;
else if(x< 128)
c = 7;
else if(x< 256)
c = 8;
else if(x< 512)
c = 9;
return c;
}

int max(int x,int y)
{
if(x< y)
return(y);
else
return(x);
}

void main()
{
int B,Q,Z,M,c,c1,e,f,g,h,i,j,x,y,ch,in,S,G,P;
int a[24],b[12],b1[12],q[12],carry=0,count=0,option;
long num;

printf("¦\t\tPROGRAM FOR RESTORING DIVISION\t\t¦\n");
```

```
printf("\n\nENTER DIVIDEND\t: ");
scanf("%d",&Q);
y = getsize(Q);
printf("ENTER DIVISOR\t: ");
scanf("%d",&M);
x = getsize(M);
Z = max(x,y);
printf("\n\tTOTAL BITS CONSIDERED FOR RESULT => %d",2*Z+1);
printf("\n\tINITiALLY A IS RESET TO ZERO:");
for(i=0;i<=Z;i++)
printf("%d ",a[i]=0);
for(i=Z;i>=0;i--)
{
b1[i] = b[i] = M%2;
M = M/2;
b1[i] = 1-b1[i];
}
carry = 1;
for(i=Z;i>=0;i--)
{
c1 = b1[i]^carry;
carry = b1[i]&&carry;
b1[i]=c1;
}
for(i=2*Z;i>Z;i--)
{
a[i] = Q%2;
Q = Q/2;
}
printf("\n\n\tDivisor\t\t(M)\t: ");
for(i=0;i<=Z;i++)
printf("%d ",b[i]);
printf("\n\t2'C Divisor\t(-M)\t: ");
for(i=0;i<=Z;i++)
printf("%d ",b1[i]);
printf("\n\tDividend\t(Q)\t: ");
for(i=Z+1;i<=2*Z;i++)
printf("%d ",a[i]);
printf("\n\n\tBITS CONSIDERED:[ A ]\t [ M ]");
```

```
printf("\n\t\t\t");
for(i=0;i<=Z;i++)
printf("%d ",a[i]);
printf(" ");
for(i=Z+1;i<=2*Z;i++)
printf("%d ",a[i]);
count = Z;
do{
for(i=0;i<2*Z;i++)
a[i] = a[i+1];
printf("\n\nLeft Shift\t\t");
for(i=0;i<=Z;i++)
printf("%d ",a[i]);
printf(" ");
for(i=Z+1;i< 2*Z;i++)
printf("%d ",a[i]);
carry=0;
for(i=Z;i>=0;i--)
{
S=a[i]^(b1[i]^carry);
G=a[i]&&b1[i];
P=a[i]^b1[i];
carry=G||(P&&carry);
a[i]=S ;
}
printf("\nA< -A-M \t\t");
for(i=0;i<=Z;i++)
printf("%d ",a[i]);
printf(" ");
for(i=Z+1;i< 2*Z;i++)
printf("%d ",a[i]);
ch=a[0];
printf("\nBIT Q:%d",ch);
switch (ch)
{
case 0: a[2*Z]=1;
printf(" Q0< -1\t\t");
for(i=0;i<=Z;i++)
printf("%d ",a[i]);
```

```
printf(" ");
for(i=Z+1;i<=2*Z;i++)
printf("%d ",a[i]);
break;

case 1: a[2*Z]=0;
printf(" Q0< -0\t\t");
for(i=0;i<=Z;i++)
printf("%d ",a[i]);
printf(" ");
for(i=Z+1;i< 2*Z;i++)
printf("%d ",a[i]);
carry=0;
for(i=Z;i>=0;i--)
{
S=a[i]^(b[i]^carry);
G=a[i]&&b[i];
P=a[i]^b[i];
carry=G||(P&&carry);
a[i]=S ;
}
printf("\nA< -A+M");
printf("\t\t\t");
for(i=0;i<=Z;i++)
printf("%d ",a[i]);
printf(" ");
for(i=Z+1;i<=2*Z;i++)
printf("%d ",a[i]);
break;
}
count--;
}while(count!=0);
num=0;
printf("\n\n\t\tQUOTIENT IN BITS :");
for(i=Z+1;i<=2*Z;i++)
{
printf("%d ",a[i]);
num=num+pow(2,2*Z-i)*a[i];
}
```

```
printf("\n\t\tOUOTIENT IN DECIMAL :%ld",num);
num=0;
printf("\n\t\tREMAINDER IN BITS :");
for(i=0;i<=Z;i++)
{
printf("%d ",a[i]);
num=num+pow(2,Z-i)*a[i];
}
printf("\n\t\tREMAINDER IN DECIMAL :%ld",num);
getch();

getch();
}
```

OUTPUT:

```
|               PROGRAM FOR RESTORING DIVISION          |


ENTER DIVIDEND   : 17
ENTER DIVISOR    : 4

        TOTAL BITS CONSIDERED FOR RESULT => 11
        INITiALLY A IS RESET TO ZERO:0 0 0 0 0 0

        Divisor          (M)      : 0 0 0 1 0 0
        2'C Divisor      (-M)     : 1 1 1 1 0 0
        Dividend         (Q)      : 1 0 0 0 1

        BITS CONSIDERED:[ A ]     [ M ]
                        0 0 0 0 0 0  1 0 0 0 1

Left Shift              0 0 0 0 0 1  0 0 0 1
A< -A-M                 1 1 1 1 0 1  0 0 0 1
BIT Q:1 Q0< -0          1 1 1 1 0 1  0 0 0 1
A< -A+M                 0 0 0 0 0 1  0 0 0 1 0

Left Shift              0 0 0 0 1 0  0 0 1 0
A< -A-M                 1 1 1 1 1 0  0 0 1 0
BIT Q:1 Q0< -0          1 1 1 1 1 0  0 0 1 0
A< -A+M                 0 0 0 0 1 0  0 0 1 0 0

Left Shift              0 0 0 1 0 0  0 1 0 0
A< -A-M                 0 0 0 0 0 0  0 1 0 0
BIT Q:0 Q0< -1          0 0 0 0 0 0  0 1 0 0 1

Left Shift              0 0 0 0 0 0  1 0 0 1
A< -A-M                 1 1 1 1 0 0  1 0 0 1
BIT Q:1 Q0< -0          1 1 1 1 0 0  1 0 0 1
A< -A+M                 0 0 0 0 0 0  1 0 0 1 0

Left Shift              0 0 0 0 0 1  0 0 1 0
A< -A-M                 1 1 1 1 0 1  0 0 1 0
BIT Q:1 Q0< -0          1 1 1 1 0 1  0 0 1 0
A< -A+M                 0 0 0 0 0 1  0 0 1 0 0

                QUOTIENT IN BITS :0 0 1 0 0
                OUOTIENT IN DECIMAL :4
                REMAINDER IN BITS :0 0 0 0 0 1
                REMAINDER IN DECIMAL :1
```

CONCLUSION: From this experiment, we learn how to use the restoring division algorithm to divide unsigned bits. We understood that it is a type of slow algorithm along with non-restoring division. We also learn how to write the code for the same algorithm in C language and implement it successfully.

EXPERIMENT - 3

Aim: To study and implement non-restoring division algorithms.

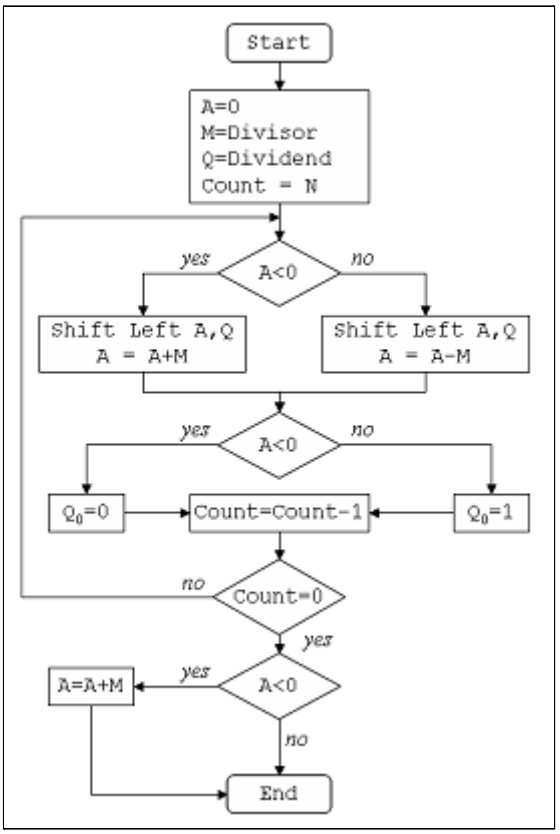
Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	30-10-2021	30-10-2021	

Theory

A division algorithm is an algorithm which, given two integers N and D, computes their quotient and/or remainder, the result of Euclidean division. Division algorithms fall into two main categories: slow division and fast division. Slow division algorithms produce one digit of the final quotient per iteration. Examples of slow division include restoring, non-performing restoring, non-restoring, and SRT division. Fast division methods start with a close approximation to the final quotient and produce twice as many digits of the final quotient on each iteration. Newton–Raphson and Goldschmidt algorithms fall into this category.

Non-Restoring division, it is less complex than the restoring one because simpler operations are involved i.e. addition and subtraction, also now the restoring step is performed. In the method, rely on the sign bit of the register which initially contains zero named as A.
Here is the flow chart given below.



Algorithm

```
1) Set the value of register A as 0 (N bits)
2) Set the value of register M as Divisor (N bits)
3) Set the value of register Q as Dividend (N bits)
4) Concatenate A with Q {A,Q}
5) Repeat the following "N" number of times (here N is no. of bits in
divisor):
    If the sign bit of A equals 0,
      shift A and Q combined left by 1 bit and subtract M from A,
    else shift A and Q combined left by 1 bit and add M to A
    Now if sign bit of A equals 0, then set Q[0] as 1, else set Q[0]
as 0
6) Finally if the sign bit of A equals 1 then add M to A.
7) Assign A as remainder and Q as quotient.
```

Example:

Dividend (A) = 101110, ie 46, and Divisor (B) = 010111, ie 23.

Initialization :
Set Register A = Dividend = 000000
Set Register Q = Dividend = 101110
(So AQ = 000000 101110 , Q0 = LSB of Q = 0)
Set M = Divisor = 010111, M' = 2's complement of M = 101001
Set Count = 6, since 6 digits operation is being done here.

After this we start the algorithm, which I have shown in a table below :
In the table, SHL(AQ) denotes shift left AQ by one position leaving Q0 blank.
Similarly, a square symbol in Q0 position denote, it is to be calculated later

Action	A	Q	Count
Initial	000 000	101 110	6
A > 0 => SHL (AQ)	000 001	011 10□	
A = A-M	101 010	011 10□	
A < 0 => Q0 = 0	101 010	011 100	5
A < 0 => SHL (AQ)	010 100	111 00□	
A = A+M	101 011	111 00□	
A < 0 => Q0 = 0	101 011	111 000	4
A < 0 => SHL (AQ)	010 111	110 00□	
A = A+M	101 110	110 00□	
A < 0 => Q0 = 0	101 110	110 000	3
A < 0 => SHL (AQ)	011 101	100 00□	
A = A+M	110 100	100 00□	
A < 0 => Q0 = 0	110 100	100 000	2
A < 0 => SHL (AQ)	101 001	000 00□	
A = A+M	000 000	000 00□	
A < 0 => Q0 = 1	000 000	000 001	1
A > 0 => SHL (AQ)	000 000	000 01□	
A = A+M	101 001	000 01□	
A < 0 => Q0 = 1	101 001	000 010	0
Count has reached Zero, So final steps			
A < 0 => A = A+M	**000 000**	**000 010**	
	Reminder	**Quotient**	

CODE:

```
#include<stdio.h>
#include<malloc.h>
int *a,*q,*m,*mc,*c,n,d;
int powr(int x,int y)
{
 int s=1,i;
 for(i=0;i<y;i++)
   s=s*x;
 return s;
}
void print(int arr[],int n)
{
 int i;
 for(i=0;i<n;i++)
  printf("%d ",arr[i]);
}
void bin(int n, int arr[]){
    int r, i = 0;
    do{
        r = n % 2;
        n /= 2;
        arr[i] = r;
        i++;
    }while(n > 0);
 }
void set(int array[], int x){
    int i,tmp[20]={0};
    for(i = x -1; i >=0; i--)
  tmp[x-1-i]=array[i];
 for(i=0;i<x;i++)
  array[i]=tmp[i];
 }
int len(int x)
{
 int i=0;
 while(powr(2,i)<=x) i++;
 return ++i;
}
```

```
void addBinary(int a1[], int a2[])
{
    int bi[2]={0},ca[20]={0};
    int t=len(n),tmp=0;
    int *su=(int*)malloc(sizeof(int)*len(n));
 while(t-->0)
 {
  tmp=a1[t]+a2[t]+ca[t];
  bin(tmp,bi);
  su[t]=bi[0];
  ca[t-1]=bi[1];
  bi[0]=0;bi[1]=0;
 }
 for(t=0;t<len(n);t++)
  a1[t]=su[t];
 free(su);
 }

void twoCom(int arr[]){
    int i;
    int *one=(int*)malloc(sizeof(int)*len(n));
    for(i=0;i<len(n)-1;i++)
  one[i]=0;
 one[i]=1;
    for(i = 0; i < len(n); i++){
        arr[i]=1-arr[i];
    }
    addBinary(arr, one);
    free(one);
}
void ls(int alen,int blen)
{
 int i=0;
 for(i=0;i<alen-1;i++)
  a[i]=a[i+1];
 a[i]=q[0];
 for(i=0;i<blen-1;i++)
  q[i]=q[i+1];
 q[i]=-1;
}
```

```
void printaq()
{
 print(a,len(n));
 printf("\t");
 print(q,len(n)-1);
 printf("\t");
 printf("\n");
}
int main()
{
 int i,cnt=0;
 printf("Enter The Numerator/Denominator: ");
 scanf("%d/%d",&n,&d);
 q=(int*)malloc(sizeof(int)*len(n)-1);
 bin (n,q);
 m=(int*)malloc(sizeof(int)*(len(n)));
 bin(d,m);
 a=(int*)malloc(sizeof(int)*(len(n)));
 for(i=0;i<len(n);i++)
  a[i]=0;
 mc=(int*)malloc(sizeof(int)*(len(n)));
 bin(d,mc);
 set(q,len(n)-1);
 set(m,len(n));
 set(mc,len(n));
 twoCom(mc);
 cnt=len(n)-1;
 printf("\t    A\t\t   Q\t\t    M\t    Count\n");
 printf("\t---------\t-------\t\t--------- ---------\n");
 while(cnt>0)
 {
  printf("\t");
  print(a,len(n));
  printf("\t");
  print(q,len(n)-1);
  printf("\t");
  print(m,len(n));
  printf("\t%d\n",cnt);
  if(a[0]==1)
  {
```

```
   ls(len(n),len(n)-1);
   printf("LSHIFT\t");
   printaq();
   addBinary(a,m);
   printf("A=A+M\t");
   printaq();
  }
  else
  {
   ls(len(n),len(n)-1);
   printf("LSHIFT\t");
   printaq();
   addBinary(a,mc);
   printf("A=A-M\t");
   printaq();
  }
  if(a[0]==1)
  {
   q[len(n)-2]=0;
   addBinary(a,m);
  }
  else
   q[len(n)-2]=1;
  printf("A=A+M\t");
  printaq();
  cnt-=1;
  printf("\n");
 }
 return 0;
}
```

OUTPUT:

```
Enter The Numerator/Denominator: 10/3
            A                 Q                 M          Count
         ---------         -------           ---------  ---------
         0 0 0 0 0         1 0 1 0           0 0 0 1 1          4
LSHIFT   0 0 0 0 1         0 1 0 -1
A=A-M    1 1 1 1 0         0 1 0 -1
A=A+M    0 0 0 0 1         0 1 0 0

         0 0 0 0 1         0 1 0 0           0 0 0 1 1          3
LSHIFT   0 0 0 1 0         1 0 0 -1
A=A-M    1 1 1 1 1         1 0 0 -1
A=A+M    0 0 0 1 0         1 0 0 0

         0 0 0 1 0         1 0 0 0           0 0 0 1 1          2
LSHIFT   0 0 1 0 1         0 0 0 -1
A=A-M    0 0 0 1 0         0 0 0 -1
A=A+M    0 0 0 1 0         0 0 0 1

         0 0 0 1 0         0 0 0 1           0 0 0 1 1          1
LSHIFT   0 0 1 0 0         0 0 1 -1
A=A-M    0 0 0 0 1         0 0 1 -1
A=A+M    0 0 0 0 1         0 0 1 1
```

Conclusion :

The non-restorative division algorithm is an efficient way to perform binary division compared to traditional subtractive based algorithms by using the faster processed bit shift commands in the CPU registers. The algorithm is simple enough to be implemented in hardware in equipment like Arithmometers while also generalising to complex modern day systems. The algorithm serves as a good example in showing that considering lower-level system dependencies and physical limitations can be used to optimize algorithms. Non-restorative algorithm is more efficient than restorative algorithm as it uses simpler commands in terms of addition and subtraction, however it is slower than other algorithms.

EXPERIMENT - 4

Aim:

Implement Sequential memory organization with following details:

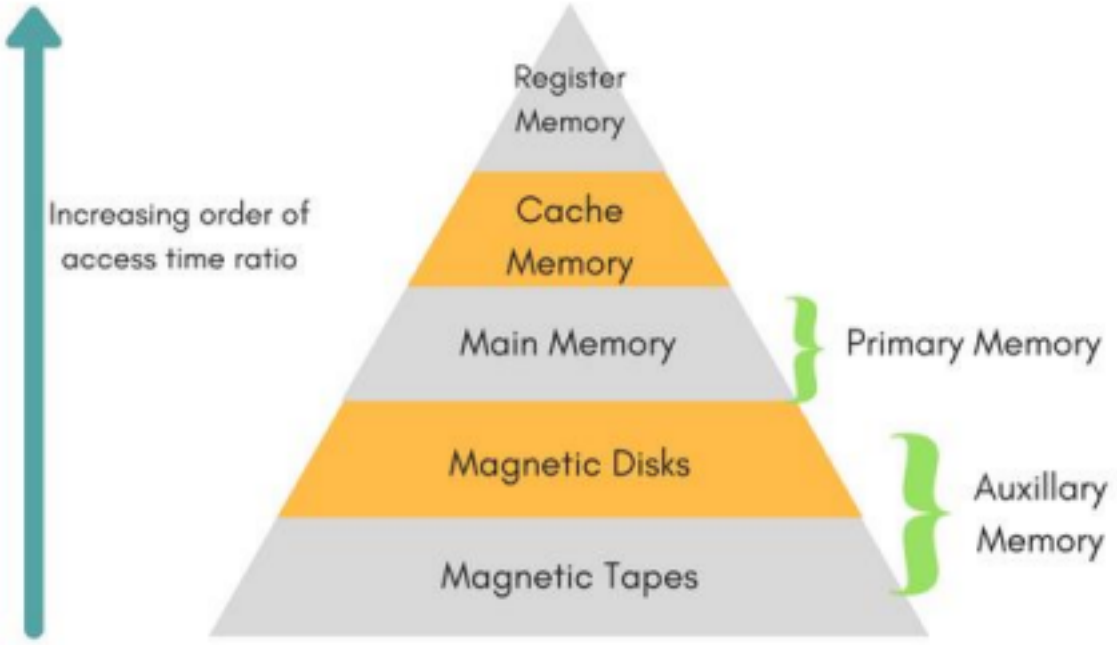
- Processor can access 1 word = 4 bytes.
- L1 cache can store max = 32 words.
- L2 cache can store max =128 words.
- Main memory can store max = 2048 bytes.

Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	30-10-2021	30-10-2021	

Theory:

In Computer System Design, Memory Hierarchy is an enhancement to organize the memory such that it can minimize the access time. The Memory Hierarchy was developed based on a program behaviour known as locality of references. The figure below clearly demonstrates the different levels of memory hierarchy:



This Memory Hierarchy Design is divided into 2 main types:

- External Memory or Secondary Memory – Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e., peripheral storage devices which are accessible by the processor via I/O Module.
- Internal Memory or Primary Memory – Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.

Levels of memory:

- Level 1 or Register –

  It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, Program counter, address register etc.

- Level 2 or Cache memory –

  It is the fastest memory which has faster access time where data is temporarily stored for faster access.

- Level 3 or Main Memory –

  It is memory on which computer works currently. It is small in size and once power is off data no longer stays in this memory.

- Level 4 or Secondary Memory –

  It is external memory which is not as fast as main memory but data stays permanently in this memory.

Types of Cache:

- Primary Cache –

  A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.

- Secondary Cache –

  Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

Code:

```
memory.cpp
#include<bits/stdc++.h>
#include<time.h>
using namespace std;
int main() {
 vector<double> lc1(32,-1);
 vector<double> lc2(64,-1);


 double lc1Time = 20;
 double lc2Time = 60;
 double mainMemoryTime = 120;
 double totalHits = 0;
 double lc1Hits = 0;
 double lc2Hits = 0;
 double count = 0;
 int fr1 = -1;
 int re1 = 0;
 int fr2 = -1;
 int re2 = 0;
 char key = 100;
 srand(time(0));
 cout<<"Sequential Memory Organization:"<<endl<<endl;
 cout<<" "<<"Requirement"<<setw(22)<<"Location of Hit"<<" "<<setw(15)<<"Avg.
Access Time"<<endl;
 while(key--) {
 count++;
```

```
 double avgTime = 0;
 double input = rand()%512;
 bool f1 = false;
 bool f2 = false;
 for(double i=0; i<32; i++) {
 if(lc1[i]==input) {
 lc1Hits++;
 f1 = true;
 f2 = true;
 avgTime += (lc1Hits/(count))*lc1Time + (1-
(lc1Hits/(count)))*(lc2Hits/(count-lc1Hits))*(lc1Time+lc2Time) + (1-
(lc1Hits/(count)))*(1-(lc2Hits/(count
lc1Hits)))*1*(lc1Time+lc2Time+mainMemoryTime);


 cout<<setw(12)<<input<<" "<<setw(20)<<"L1 Cache"<<"
"<<setw(8)<<avgTime<<endl;
 }
 }
 if(!f1) {
 double b = input/2;
 for(double i=0; i<64; i++) {
 if(lc2[i]==b) {
 lc2Hits++;
 f2 = true;
 // found in 2
 if(re1==fr1) {
 re1 = (re1+1)%32;
 }
 fr1 = (fr1+1)%32;
 lc1[fr1] = input;
 avgTime += (lc1Hits/(count))*lc1Time + (1-
(lc1Hits/(count)))*(lc2Hits/(count-lc1Hits))*(lc1Time+lc2Time) + (1-
(lc1Hits/(count)))*(1-(lc2Hits/(count
lc1Hits)))*1*(lc1Time+lc2Time+mainMemoryTime);
 cout<<setw(12)<<input<<" "<<setw(20)<<"L2 Cache"<<"  "<<setw(8)<<avgTime<<endl;
 }
 }
 }
 if(!f2) {
 double b = input/2;
 if(re2==fr2) {
 re2 = (re2+1)%64;
 }
 fr2 = (fr2+1)%64;
 lc2[fr2] = b;
 if(re1==fr1) {
 re1 = (re1+1)%32;
```

```
 }


 fr1 = (fr1+1)%32;
 lc1[fr1] = input;
 avgTime += (lc1Hits/(count))*lc1Time + (1-
(lc1Hits/(count)))*(lc2Hits/(count-lc1Hits))*(lc1Time+lc2Time) + (1-
(lc1Hits/(count)))*(1-(lc2Hits/(count
lc1Hits)))*1*(lc1Time+lc2Time+mainMemoryTime);
 cout<<setw(12)<<input<<" "<<setw(20)<<"Main Memory"<<"
"<<setw(8)<<avgTime<<endl;
 }
 }
 cout<<endl;
 cout<<"L1 Hit Ratio (H1) = "<<lc1Hits<<"/"<<count<<" =
"<<(double)lc1Hits/count<<endl;
 cout<<"L1 Access Time (T1) = "<<lc1Time<<" ns"<<endl<<endl;
 cout<<"L2 Hit Ratio (H2) = "<<lc2Hits<<"/"<<(count-lc1Hits)<<" =
"<<(double)lc2Hits/(count-lc1Hits)<<endl;
 cout<<"L2 Access Time (T2) = "<<lc2Time<<" ns"<<endl<<endl;
 cout<<"Main Memory Hit Ratio (Hm) = "<<1<<endl;
 cout<<"Main memory Access Time = "<<mainMemoryTime<<" ns"<<endl<<endl;
 cout<<"Average Access Time = [H1*T1] + [(1-H1)*H2*(T1+T2)] + [(1-H1)*(1-
H2)*Hm*(T1+T2+Tm)]"<<endl;
 double finalAns = ((double)lc1Hits/count)*lc1Time + (1-
((double)lc1Hits/count))*((double)lc2Hits/(100-lc1Hits))*(lc1Time+lc2Time) + (1-
((double)lc1Hits/count))*(1-((double)lc2Hits/(count
lc1Hits)))*1*(lc1Time+lc2Time+mainMemoryTime);
 cout<<"Average Access Time = "<<finalAns<<" ns"<<endl;

 return 0;
}
```

Output:

```
Sequential Memory Organization:
Requirement Location of Hit Avg. Access Time
 353 Main Memory 200
 372 Main Memory 200
 89 Main Memory 200
 189 Main Memory 200
 265 Main Memory 200
 423 Main Memory 200
 379 Main Memory 200
 107 Main Memory 200
 8 Main Memory 200
 217 Main Memory 200
```

```
155 Main Memory 200
294 Main Memory 200
192 Main Memory 200
47 Main Memory 200
348 Main Memory 200
217 L1 Cache 188.75
264 Main Memory 189.412
375 Main Memory 190
100 Main Memory 190.526
351 Main Memory 191
476 Main Memory 191.429
12 Main Memory 191.818
178 Main Memory 192.174
138 Main Memory 192.5
338 Main Memory 192.8
250 Main Memory 193.077
306 Main Memory 193.333
363 Main Memory 193.571
208 Main Memory 193.793
99 Main Memory 194
143 Main Memory 194.194
205 Main Memory 194.375
381 Main Memory 194.545
34 Main Memory 194.706
329 Main Memory 194.857
259 Main Memory 195
245 Main Memory 195.135
429 Main Memory 195.263
7 Main Memory 195.385
228 Main Memory 195.5
65 Main Memory 195.61
320 Main Memory 195.714
46 Main Memory 195.814
19 Main Memory 195.909
262 Main Memory 196
56 Main Memory 196.087
178 L1 Cache 192.34
132 Main Memory 192.5


252 Main Memory 192.653
189 L2 Cache 190.4
301 Main Memory 190.588
315 Main Memory 190.769
418 Main Memory 190.943
163 Main Memory 191.111
302 Main Memory 191.273
87 Main Memory 191.429
```

```
 49 Main Memory 191.579
 402 Main Memory 191.724
 467 Main Memory 191.864
 80 Main Memory 192
 462 Main Memory 192.131
 101 Main Memory 192.258
 410 Main Memory 192.381
 125 Main Memory 192.5
 3 Main Memory 192.615
 465 Main Memory 192.727
 116 Main Memory 192.836
 217 L2 Cache 191.176
 368 Main Memory 191.304
 71 Main Memory 191.429
 27 Main Memory 191.549
 262 L1 Cache 189.167
 190 Main Memory 189.315
 283 Main Memory 189.459
 166 Main Memory 189.6
 146 Main Memory 189.737
 469 Main Memory 189.87
 229 Main Memory 190
 347 Main Memory 190.127
 122 Main Memory 190.25
 335 Main Memory 190.37
 296 Main Memory 190.488
 452 Main Memory 190.602
 492 Main Memory 190.714
 44 Main Memory 190.824
 305 Main Memory 190.93
 261 Main Memory 191.034
 81 Main Memory 191.136
 455 Main Memory 191.236
 66 Main Memory 191.333
 364 Main Memory 191.429
 139 Main Memory 191.522
 8 Main Memory 191.613
 240 Main Memory 191.702
 74 Main Memory 191.789
 379 Main Memory 191.875
 476 Main Memory 191.959
 458 Main Memory 192.041
 1 Main Memory 192.121
 455 L1 Cache 190.4
L1 Hit Ratio (H1) = 4/100 = 0.04
L1 Access Time (T1) = 20 ns
```

```
L2 Hit Ratio (H2) = 2/96 = 0.0208333
L2 Access Time (T2) = 60 ns
Main Memory Hit Ratio (Hm) = 1
Main memory Access Time = 120 ns
Average Access Time = [H1*T1] + [(1-H1)*H2*(T1+T2)] + [(1-H1)*(1-
H2)*Hm*(T1+T2+Tm)]
Average Access Time = 190.4 ns
```

Conclusion:

We learnt about sequential memory organization and implemented it in C++ programming language successfully on the given problem statement in a 2-Level Cache Memory architecture for 100 different number words required by the processor.

EXPERIMENT - 5

AIM: Study various microprocessors and microcontrollers used in the market specifically targeting Aircrafts and Automated Vehicles.

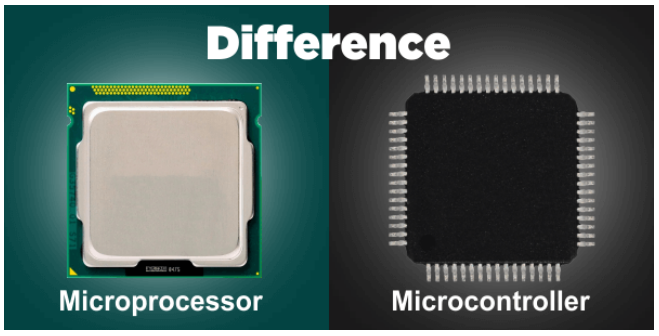
Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	17-12-2021	17-12-2021	

THEORY:

Microprocessors

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it. Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device. Register array consists of registers identified by letters like B, C, D, E, H, L and accumulator. The control unit controls the flow of data and instructions within the computer.



Microcontrollers

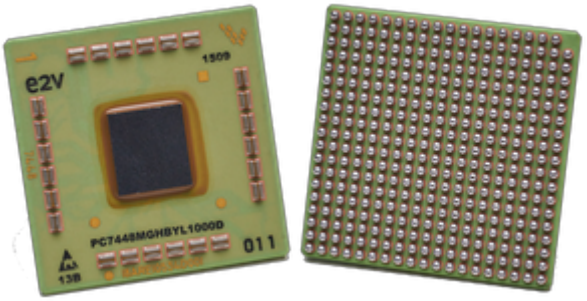
A microcontroller is a small and low-cost microcomputer, which is designed to perform the specific tasks of embedded systems like displaying microwave's information, receiving remote signals, etc. The general microcontroller consists of the processor, the memory (RAM, ROM, EPROM), Serial ports, peripherals (timers, counters), etc.

Microprocessors used in Aircrafts and Automated Vehicles

1. E2V's PC7448 Processor

In flight and mission control systems the combination of data from different on board sensors can be a computationally intensive operation. In particular image processing from optical and radar sensors requires large amount of processing power. The use of extended temperature range of PowerPC processors in an avionics environment allows for optimisation of data-link resources, eases the mission control task and also means that the time of response can be dramatically improved.

The PC7448 is a popular processor in the avionics industry and used extensively inflight computers. This superscalar microprocessor can provide 3000 Dhrystone MIPS performance and has integrated L1 and L2 cache. There are also multicore processors in the P and T series which are also now being used in avionic systems applications.



Major Features are :

1. Up to four instructions can be fetched from the instruction cache at a time
2. Up to three instructions plus a branch instruction can be dispatched to the issue queues at a time
3. Up to 12 instructions can be in the Instruction Queue (IQ)
4. Up to 16 instructions can be at some stage of execution simultaneously
5. Single-cycle execution for most instructions
6. One instruction per clock cycle throughput for most instructions
7. Seven-stage pipeline control
8. 32-Kbyte, eight-way set-associative instruction and data caches
9. On-chip, 1-Mbyte, eight-way set-associative unified instruction and data cache

2. E2V's PC5674 Processor

It is used for Engine Control. PC5674 microprocessor has an increased temperature range of -55°C ttems. The part contains embedded flash for program storage and 256kbyte of SRAM. Other functions typically used for engine control include a programmable timer interface, ao +150°C which makes it suitable for high temperature environments such as engine sys CAN bus interface and internal 12-bit analogueto-digital converters (ADCs).

Major Features :

1. 16 KB I-Cache and 16 KB D-Cache
2. 256 KB On-Chip General-Purpose SRAM Including 32 KB of Standby RAM
3. Interrupt Controller (INTC)

3. E2V's PC 8548 Processor

PC8548 is an integrated microprocessor with space level screening based on the recently introduced QML-Y standard. These processors offer clock speeds in excess of 1GHz and include advanced processing engines which accelerate the processing power of the circuit. They will be incorporated into satellite processing systems and will allow intensive on-board data processing which will improve the efficiency of future satellite systems.

More advanced multicore processors such as the P and T series offer the possibility to segment tasks within a particular core. Communication between tasks and cores is handled by on chip dedicated hardware which means that separate tasks can interact much more rapidly than if they were in separate processors. This has many applications in the complex control environments in space.

Major Features :

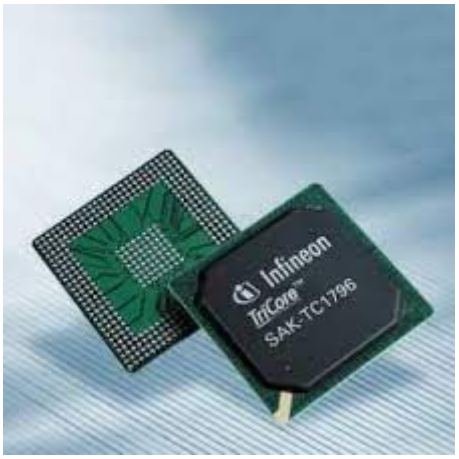
1. Dual Dispatch Superscalar, 7-stage Pipeline Design with out-of-order Issue and Execution
2. 3065 MIPS at 1333 MHz (Estimated Dhrystone 2.1)
3. 36-bit Physical Addressing
4. L1 Cache-32 KB Data and 32 KB Instruction Cache with Line-locking Support
5. L2 Cache-512 KB (8-Way Set Associative); 512 KB/256 KB/128 KB/64 KB Can Be Used As SRAM
6. 64-bit PCI 2.2 Bus Controller (Up to 66 MHz, 3.3V I/O)
7. 64-bit PCI-X Bus Controller (Up to 133 MHz, 3.3V I/O)

Microcontrollers used in Aircrafts and Automated Vehicles

Different Microcontrollers used in an automobile can communicate with one another through a multiplexing. These microcontrollers can manage related systems separately by using a BUS to communicate with other networks when they are required to perform a function. The combination of several linked networks includes the CAN (controller area networks). Present controller area networks permit complex interactions, that involve sensory systems, car speed, outdoor rainfall interactions, in car temperatures with performance controls for air conditioning maintenance, the audio visual multimedia systems and braking mechanisms.

1. Infineon Tri-core Microcontroller

Tri-core is a 32- bit microcontroller, which is developed by Infineon. These microcontrollers are assembled in over 50 automotive brands which means, every second vehicle designed today includes a Tri-core based microcontroller. It is responsible for keeping the exhaust emissions and fuel consumption as low as possible. Tri-core microcontrollers are used in the gear boxes to control the injection, central control units for combustion engines' ignition: Progressively, they are also being used in electrical and hybrid vehicle drives.

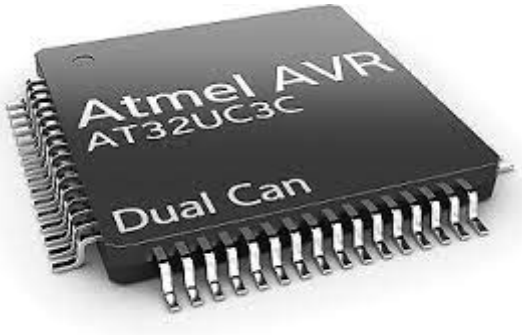


Major Features :

1. Triple Tri-Core with 200MHz
2. Tri-Core DSP functionality
3. Up to 4MB flash w/ECC protection
4. 64KB EEPROM at 500k cycles
5. Up to 472KB RAM w/ECC protection
6. Powerful Generic Timer Module (GTM)
7. SENT, PSI5, PSI5S sensor interfaces
8. Ethernet 100 Mbit
9. Programmable HSM (Hardware Security Module)

2. Atmel AVR Microcontroller

Atmel AVR (Alf-Egil-Bogen-VegardWollan-RISC) microcontrollers distribute the power, performance and flexibility for automobile applications. This microcontroller consists of the Harvard architecture. So the device runs very fast with a reduced number of machine level instructions. The AVR microcontrollers are classified into three types: Tiny AVR, Mega AVR and Xmega AVR. The main features of AVR microcontrollers compared to other microcontrollers include inbuilt ADC, 6-sleep modes serial data communication and internal oscillator, etc.



Major Features :

1. 2 Kilo bytes of internal Static RAM
2. 32 X 8 general working purpose registers
3. 32 Kilo bytes of in system self-programmable flash program memory.
4. 1024 bytes EEPROM
5. One 16-bit timer/counter with separate pre-scaler, compare mode and capture mode.
6. Two 8-bit timers/counters with separate pre-scalers and compare modes
7. 32 programmable I/O lines
8. Programmable watch dog timer with separate on-chip oscillator

3. PIC Microcontroller

The short form of the peripheral interface microcontroller is PIC. It is programmed and controlled in such a way that, it performs multiple tasks and controls a generation line. These microcontrollers are used in numerous applications like smart phones, automobiles, audio accessories and medical devices. The presently available PIC microcontrollers in the market are PIC16F84 to PIC16C84 which are affordable flash microcontrollers. Where, PIC18F458 and PIC18F258 microcontrollers are widely used in automobiles.

Major Features :

1. PIC microcontrollers are consistent and faulty of PIC percentage is very less. The performance of the PIC microcontroller is very fast because of using RISC architecture.
2. When comparing to other microcontrollers, power consumption is very less and programming is also very easy.
3. Interfacing of an analog device is easy without any extra circuitry.

4. Renesas Microcontroller

Renesas is the latest automotive microcontroller family, which offers high performance and low power consumption over a wide extent of items. This microcontroller offers embedded safety characteristics and functional security for advanced automotive applications. These microcontrollers offer the most useful microcontroller families in the world. For example, RX family offers various type of devices with memory variations from 32K flash/4K RAM to an incredible 8 flash/512K RAM. This RX family microcontroller uses 32-bit enhanced Harvard CISC architecture to attain very high performance.

Major Features :

1. 100MHz Arm Cortex-M33 with Trust Zone
2. 512kB to 1MB Flash memory
3. 64kB SRAM with parity
4. 64kB SRAM with ECC
5. 8kB Data Flash to store data as in EEPROM

6. 1kB Stand-by SRAM
7. USB 2.0 Full Speed
8. CAN 2.0B
9. -40℃ to +105℃ ambient operating temperature range
10. Scalable from 64-pin to 144-pin packages

5. 8051 Microcontroller

The 8051 microcontroller is 40 pin microcontroller and is based on Harvard architecture wherein the program memory and data memory is different. This microcontroller is used in a large number of machines like automobiles as it can be easily integrated around a machine.

Major Features :

1. 4KB bytes on-chip program memory (ROM)
2. 128 bytes on-chip data memory (RAM)
3. 128 user defined software flags
4. 8-bit bidirectional data bus
5. 16-bit unidirectional address bus
6. 32 general purpose registers each of 8-bit
7. 16-bit Timers (usually 2, but may have more or less)
8. Three internal and two external Interrupts
9. Four 8-bit ports,(short model have two 8-bit ports)
10. 16-bit program counter and data pointer.

Conclusion:

We learnt about microprocessors and microcontrollers. We then looked into their applications in the aviation industry and discussed the major features of each one of them.

EXPERIMENT - 6

Aim

To implement fully associative mapping and set associative mapping.

Submission Sheet

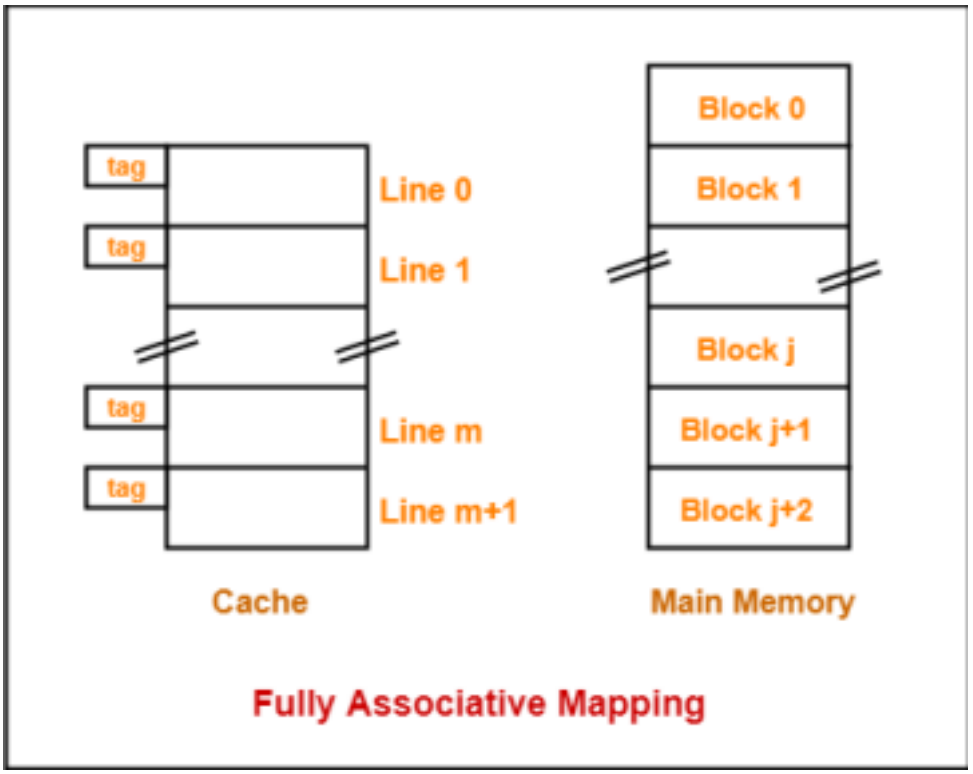
SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	6-11-2021	6-11-2021	

Theory

A CPU cache is a hardware used by the central processing unit (CPU) of a computer to reduce the average cost (time) of accessing data from the main memory. A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data from frequently used main memory locations. Cache memory is costlier than main memory or disk memory but economical than CPU registers. It is an extremely fast memory type that acts as a buffer between RAM and the CPU, and holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Fully associative mapping

Fully Associative Mapping is a cache mapping technique that allows to map a block of main memory to any freely available cache line. In this type of mapping, the associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.

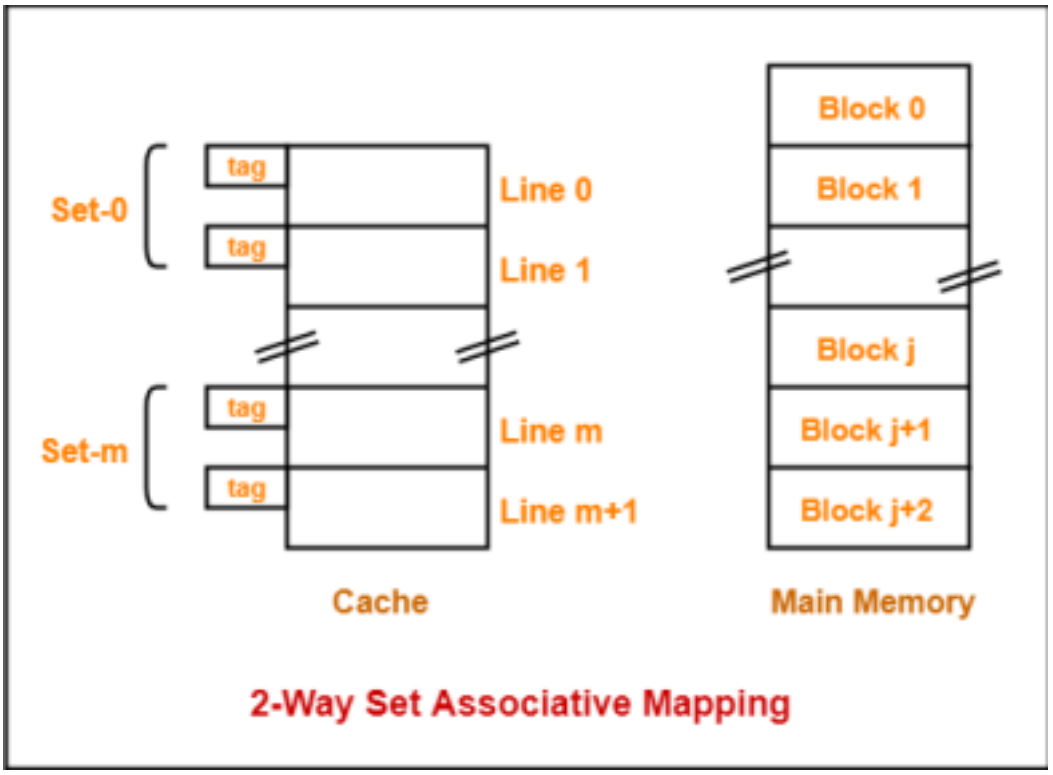


Fully Associative Mapping

Set associative mapping

Set-associative cache is a trade-off between direct-mapped cache and fully associative cache. This form of mapping is an enhanced form of direct mapping where the drawbacks of direct

mapping is removed. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set***. Then a block in memory can map to any one of the lines of a specific sea. Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques.



Prelab code for generating trace file

```
main: addu $t0,$0,$0
 addiu $t1,$0,80
 addu $t2,$0,$0
loop: lw $t3,array($t0)
 addu $t2,$t2,$t3
```

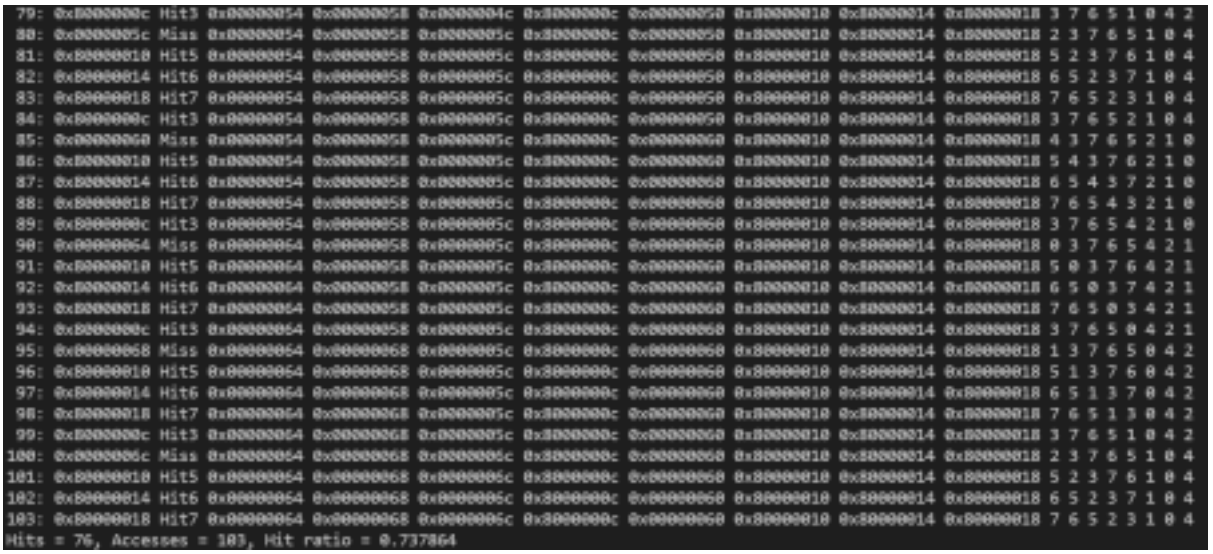
```
 addiu $t0,$t0,4
 bne $t0,$t1,loop
*done: beq $0,$0,done
array: .word 1,2,3,4,5,6,7,8,9,10
 .word 11,12,13,14,15,16,17,18,19,20
```

Fully Associative Mapping – 8 bit
Code:

```
#include <stdio.h>
int tag[8];
int mru[8] = {7,6,5,4,3,2,1,0};
void mruUpdate(int index)
{
int i;
// find index in mru
for (i = 0; i < 8; i++)
if (mru[i] == index)
break;
// move earlier refs one later
while (i > 0) {
mru[i] = mru[i-1];
i--;
}
mru[0] = index;


}
int main( )
{
int addr;
int i, j, t;
int hits, accesses;
FILE *fp;
fp = fopen("trace.txt", "r");
hits = 0;
accesses = 0;
while (fscanf(fp, "%x", &addr) > 0) {
/* simulate fully associative cache with 8 words */
accesses += 1;
printf("%3d: 0x%08x ", accesses, addr);
for (i = 0; i < 8; i++) {
if (tag[i] == addr) {
hits += 1;
printf("Hit%d ", i);
mruUpdate(i);
```

```
break;
}
}
if (i == 8) {
/* allocate entry */
printf("Miss ");
i = mru[7];
tag[i] = addr;
mruUpdate(i);
}
for (i = 0; i < 8; i++)
printf("0x%08x ", tag[i]);
for (i = 0; i < 8; i++)


printf("%d ", mru[i]);
printf("\n");
}
printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses,
((float)hits)/accesses);
pclose(fp);
}
```

Output:



Fully Associative Mapping – 16 bit

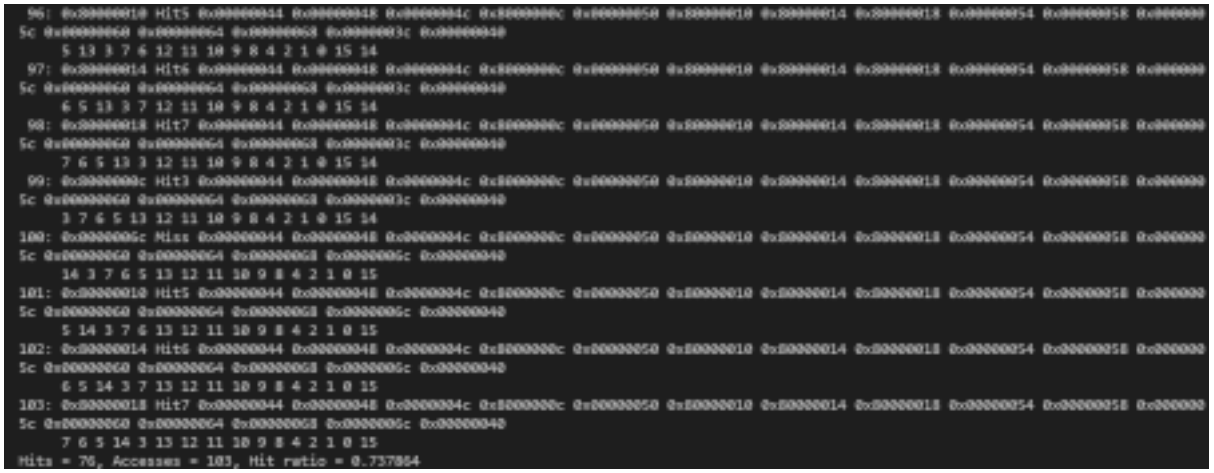
Code:

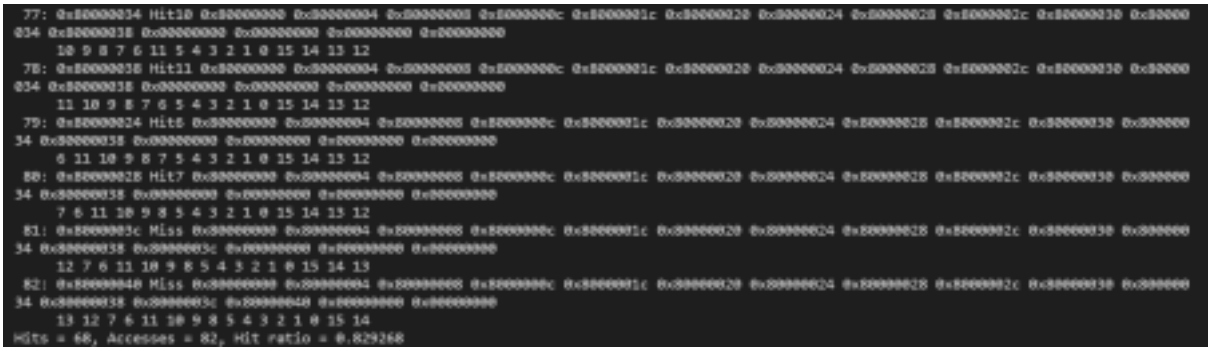
```
#include <stdio.h>
int tag[16];
int mru[16] = {15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0};
void mruUpdate(int index)
```

```
{
int i;
// find index in mru
for (i = 0; i < 16; i++)
if (mru[i] == index)
break;
// move earlier refs one later
while (i > 0) {
mru[i] = mru[i-1];


i--;
}
mru[0] = index;
}
int main( )
{
int addr;
int i, j, t;
int hits, accesses;
FILE *fp;
fp = fopen("trace.txt", "r");
hits = 0;
accesses = 0;
while (fscanf(fp, "%x", &addr) > 0) {
/* simulate fully associative cache with 16 words */
accesses += 1;
printf("%3d: 0x%08x ", accesses, addr);
for (i = 0; i < 16; i++) {
if (tag[i] == addr) {
hits += 1;
printf("Hit%d ", i);
mruUpdate(i);
break;
}
}
if (i == 16) {
/* allocate entry */
printf("Miss ");
i = mru[15];
tag[i] = addr;
mruUpdate(i);
}


for (i = 0; i < 16; i++)
printf("0x%08x ", tag[i]);
printf("\n ");
```

```
for (i = 0; i < 16; i++)
printf("%d ", mru[i]);
printf("\n");
}
printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses,
((float)hits)/accesses);
close(fp);
}
```

Output: Fully Associative Mapping – 16 bit





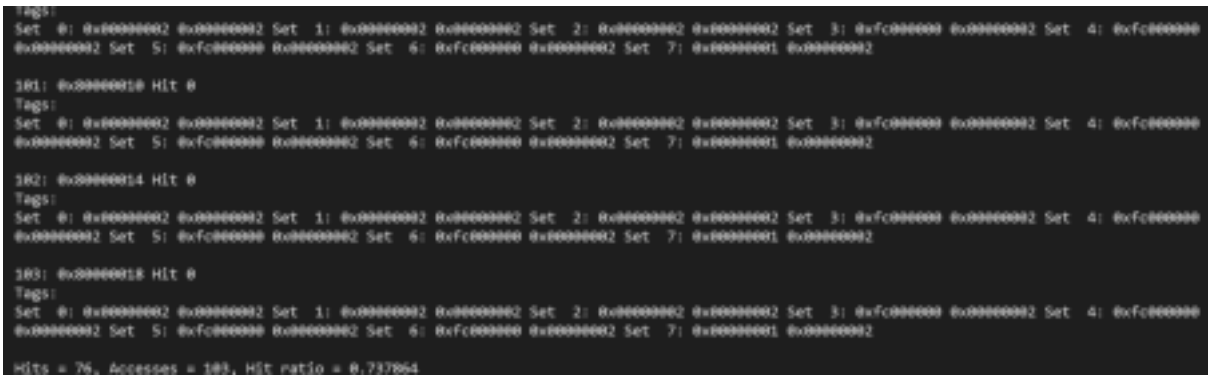
Set Associative Mapping – 16 bit

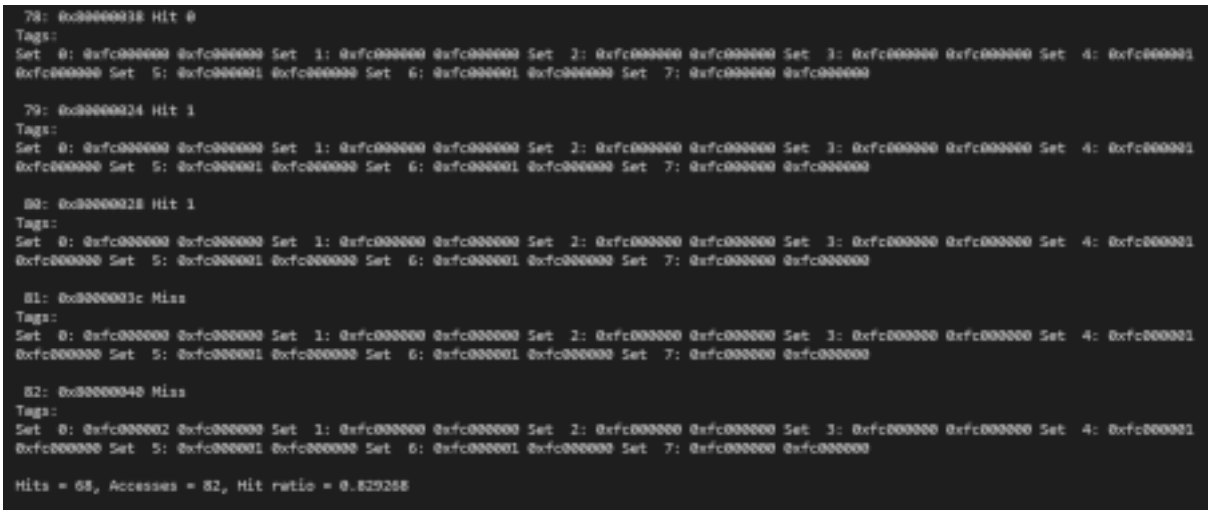
Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void main() {
int** tags = malloc(2 * sizeof(int*));
tags[0] = malloc(16 * sizeof(int));
tags[1] = malloc(16 * sizeof(int));
int mru[8] = {1,1,1,1,1,1,1,1};
int sets = 8;
int addr;


int hits, accesses;
FILE *fp;
fp = fopen("trace.txt", "r");
hits = 0;
accesses = 0;
while (fscanf(fp, "%x", &addr) > 0) {
accesses++;
int setNum = (addr >> 2) & (int)(pow(2, log2(sets)) - 1);
```

```
int addrTag = addr >> (2 + (int)log2(sets));
printf("\n%3d: 0x%08x ", accesses, addr);
int i;
for(i = 0; i < 2; ++i) {
if(tags[i][setNum] == addrTag) {
printf("Hit %d", i);
hits++;
mru[setNum] = i;
break;
}
}
if(i == 2) {
printf("Miss");
tags[(mru[setNum] + 1) % 2][setNum] = addrTag;
mru[setNum] = (mru[setNum] + 1) % 2;
}
printf("\nTags:\n");
for(int i = 0; i < sets; ++i) {
printf("Set %2d: 0x%08x 0x%08x ", i, tags[0][i], tags[0][1]);
}
printf("\n");
}
printf("\nHits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses,
((float)hits)/accesses);
fclose(fp);
}
```

Output: Set Associative Mapping – 16 bit





Conclusion

A cache is a smaller, quicker memory that stores copies of software instructions and data that are utilized regularly in the operation of programs and is positioned closer to the processor core. The program's overall speed is improved by having quick access to these commands. In this experiment, we learnt about fully associative mapping and set associative mapping and implemented them in a C program.

EXPERIMENT - 7

Aim: Study and Simulate (Stepwise) Experiments of MIPS Programming available at MIPS simulator. Modify the given programs to implement
1) Add 10 nos.
2) to print message "Hello MIPS"
3) To Reverse the input string (e.g."ABC" - "CBA").

Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	26/11/21	26/11/21	

Theory:
MIPS assembly language simply refers to the assembly language of the MIPS processor. The term MIPS is an acronym for Microprocessor without Interlocked Pipeline Stages. It is a reduced-instruction set architecture developed by an organization called MIPS Technologies. The MIPS assembly language is a very useful language to learn because many embedded systems run on the MIPS processor.

MIPS Architecture
Data Types
1. All the instructions in MIPS are 32 bits.
2. A byte in the MIPS architecture represents 8 bits; a halfword represents 2 bytes (16 bits) and a word represents 4 bytes (32 bits).
3. Each character used in the MIPS architecture requires 1 byte of storage. Each integer used requires 4 bytes of storage.

Literals
In the MIPS architecture, literals represent all numbers (e.g. 5), characters enclosed in single quotes (e.g. 'g') and strings enclosed in double quotes (e.g. "Deadpool").
Code 1: To add 10 numbers

Registers
MIPS architecture uses 32 general-purpose registers. Each register in this architecture is preceded by '$' in the assembly language instruction. You can address these registers in one of two ways. Either use the register's number (that is, from $0 to $31), or the register's name (for example, $t1).

MIPS Structure

General structure of a program created using the MIPS assembly language
A typical program created using the MIPS assembly language has two main parts. They are the data declaration section of the program and the code section of the program.

Data declaration section of a MIPS assembly language program
The data declaration section of the program is the part of the program identified with the assembler directive .data. This is the part of the program in which all the variables to be used in the program are created and defined. It is also the part of the program where storage is allocated in the main memory (RAM). The MIPS assembly language program declares variables as follows: name: .storage_type value(s).

The “name” refers to the name of the variable being created. The “storage_type” refers to the type of data that the variable is meant to store. The “value(s)” refers to the information to be stored in the variable being created.

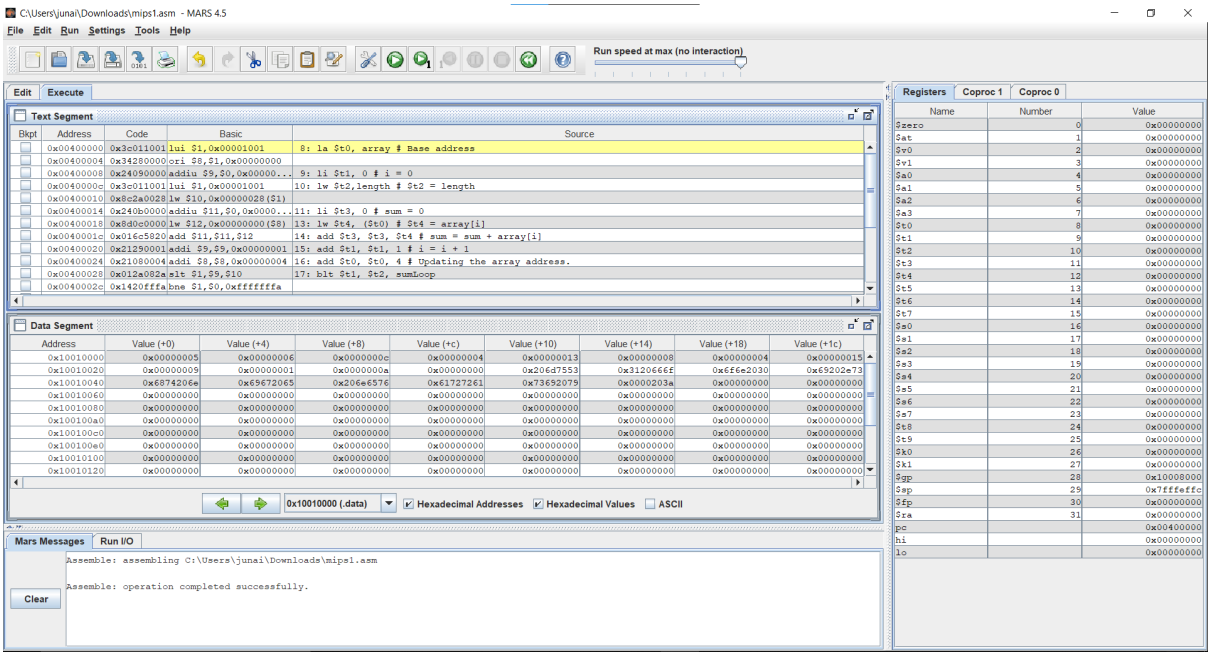
Code section of the MIPS assembly language program

The code section of the program is the part of the program in which the instructions to be executed by the program are written. It is placed in the section of the program identified with the assembler directive .text. The starting point for the code section of the program is marked with the label “main” and the ending point for the code section of the program is marked with an exit system call. This section of a MIPS assembly language program typically involves the manipulation of registers and the performance of arithmetic operations.

CODE 1: Addition of 10 numbers

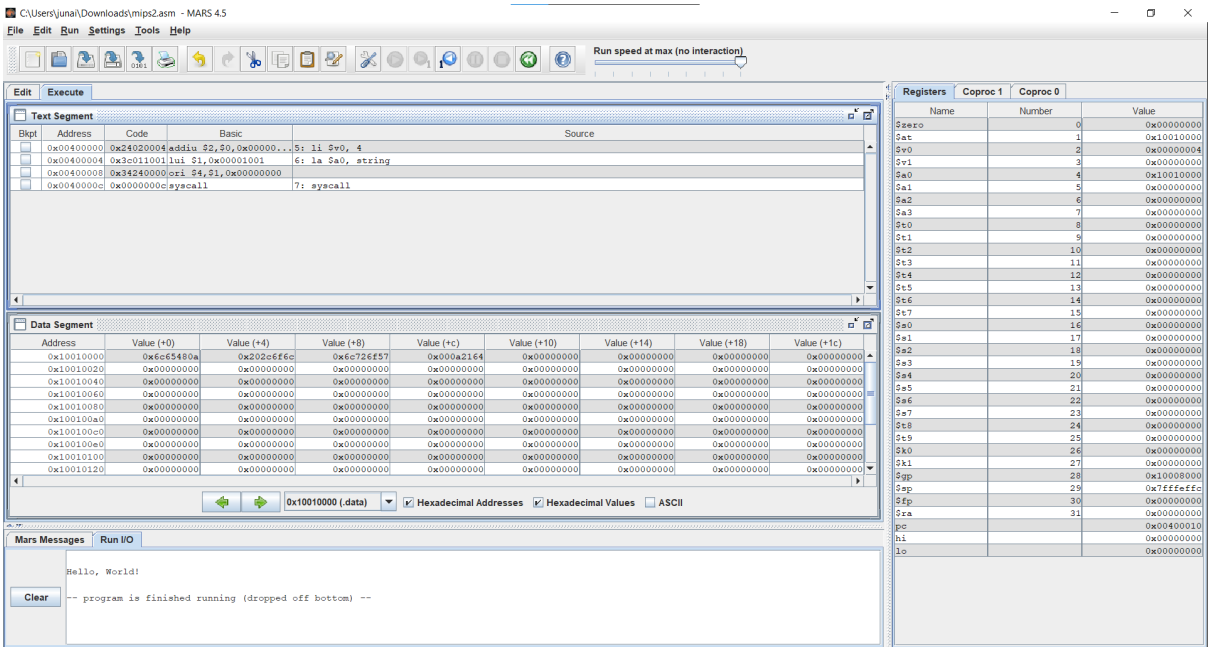
```
.data
array: .word 5,6,12,4,19,8,4,21,9,1
length: .word 10
sum: .word 0
myMessage: .ascii "Sum of 10 nos. in the given array is: "
.text
main:
la $t0, array # Base address
li $t1, 0 # i = 0
lw $t2,length # $t2 = length
li $t3, 0 # sum = 0
sumLoop:
lw $t4, ($t0) # $t4 = array[i]
add $t3, $t3, $t4 # sum = sum + array[i]
```

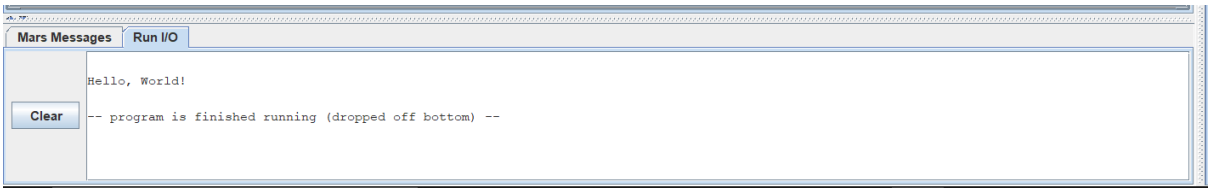
```
add $t1, $t1, 1 # i = i + 1
add $t0, $t0, 4 # Updating the array address.
blt $t1, $t2, sumLoop
sw $t3, sum
li $v0, 4
la $a0, myMessage
syscall
li $v0, 1
move $a0, $t3
syscall
```



Code 2: To print Hello World

```
.data
string: .asciiz "\nHello, World!\n"
.text
main:
li $v0, 4
la $a0, string
syscall
```



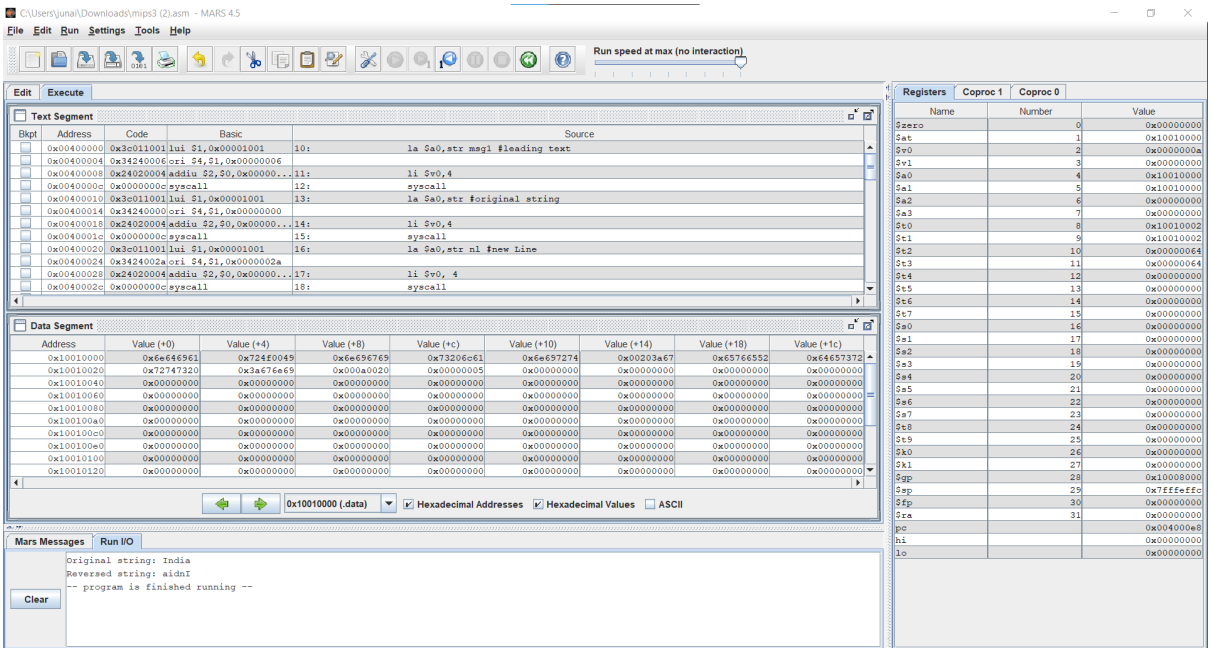


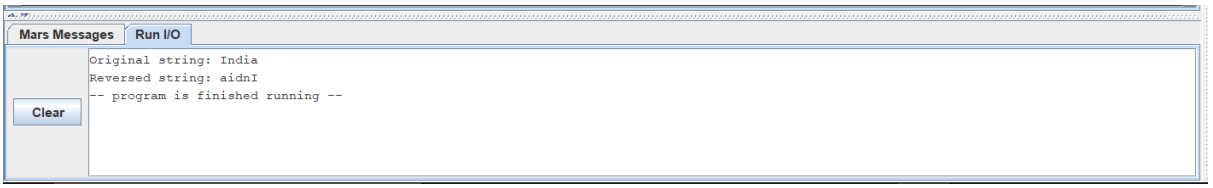
CODE 3: To Reverse the input string (e.g."ABC" – "CBA").

```
.data
        str: .asciiz "India"
        str_msg1: .asciiz "Original string: "
        str_msg2: .asciiz "Reversed string: "
        str_nl: .asciiz "\n"
        str_len: .word 0
.text
        main:
                #print original string
                la $a0,str_msg1 #leading text
                li $v0,4
                syscall
                la $a0,str #original string
                li $v0,4
```

```
syscall
la $a0,str_nl #new Line
li $v0, 4
syscall
#get length
add $t0,$zero,$zero #initialize registers when needed
add $a0,$zero,$zero
add $a1,$zero,$zero
la $a0,str #loads the address of the string
la $a1,str
getLen:
        lb $t0,0($a0) #load first byte
        beqz $t0,saveLen
        addi $a0,$a0,1
        j getLen #jump back to start of this loop
saveLen:
        subu $t0,$a0,$a1 #len = address of null terminator - str address
        sw $t0,str_len
#reverse the string
add $t0,$zero,$zero #address of the beginning of str
add $t1,$zero,$zero #address of the end of str
add $t2,$zero,$zero
add $t3,$zero,$zero
revString:
        #find the index of the last character before the end of the string
        la $t0,str #loads the address of the start of the string
        lw $t1,str_len #loads the length of the string
        addu $t1,$t0,$t1
        subi $t1,$t1,1
        loop:
                lb $t2,0($t0) #load the first character
                lb $t3,0($t1) #load the last character
                ble $t1,$t0,printRev
                sb $t3,0($t0)
                sb $t2,0($t1)
                addi $t0,$t0,1
                subi $t1,$t1,1 #and loop until we reach the middle
                j loop
#print the reversed version of the text
printRev:
        add $a0,$zero,$zero #initialize the a0 registry
        la $a0,str_msg2 #leading text
        li $v0,4
```

```
syscall
la $a0,str #reversed string
li $v0,4
syscall
li $v0,10 #exit program
syscall
```





EXPERIMENT - 8

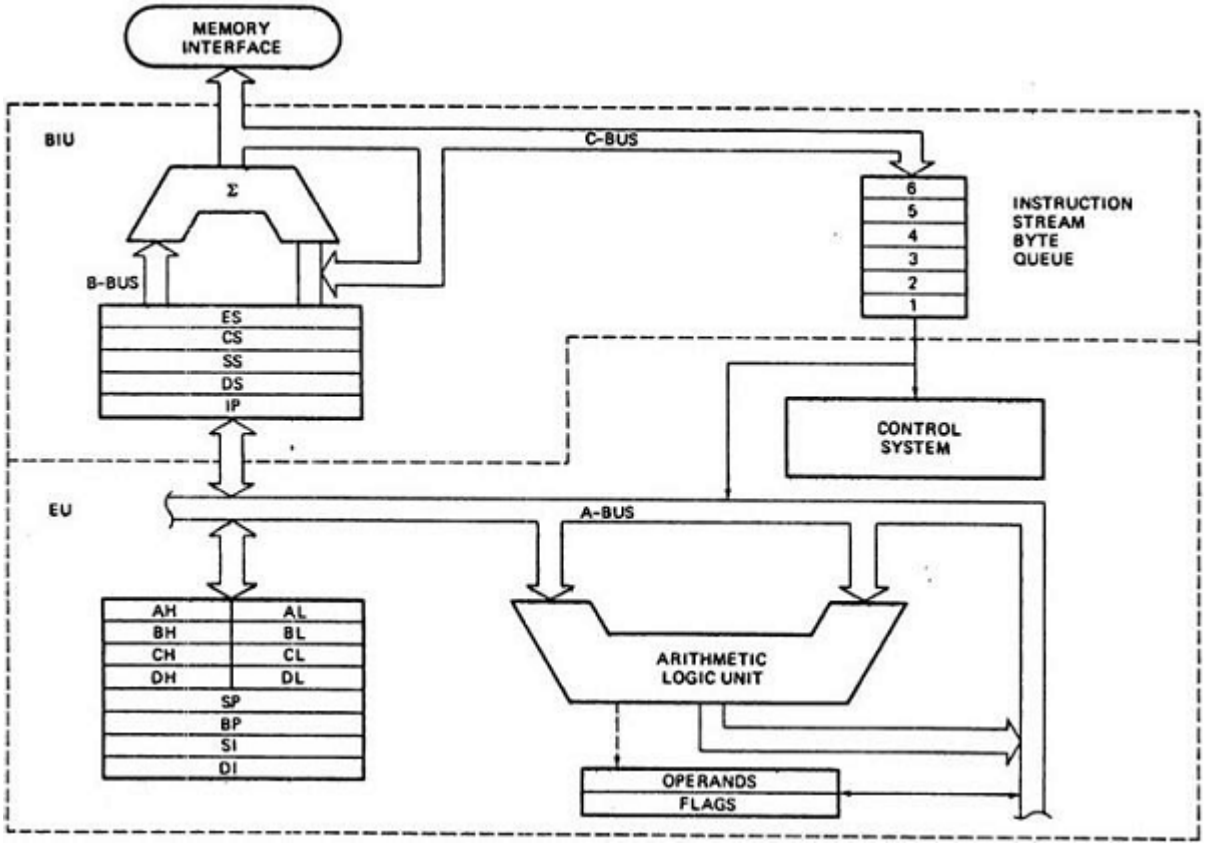
Aim : Run the simulator using the given link https://yjdoc2.github.io/8086-emulator-web/compile. Study and execute First 6 examples from the repository https://github.com/YJDoc2/8086-Emulator/tree/master/examples

Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	17/12/21	17/12/21	

Architecture of 8086

The following diagram depicts the architecture of a 8086 Microprocessor −



Pins on 8086:

AD0-AD15 (Address Data Bus): Bidirectional address/data lines. These are low order address bus. They are multiplexed with data.
When these lines are used to transmit memory address, the symbol A is used instead of AD, for example, A0- A15.
A16 - A19 (Output): High order address lines. These are multiplexed with status signals.
A16/S3, A17/S4: A16 and A17 are multiplexed with segment identifier signals S3 and S4.
A18/S5: A18 is multiplexed with interrupt status S5.
A19/S6: A19 is multiplexed with status signal S6.
BHE/S7 (Output): Bus High Enable/Status. During T1, it is low. It enables the data onto the most significant half of the data bus, D8-D15. 8-bit devices connected to the upper half of the data bus use BHE signal. It is multiplexed with status signal S7. S7 signal is available during T3 and T4.
RD (Read): For read operation. It is an output signal. It is active when LOW.
Ready (Input): The addressed memory or I/O sends acknowledgment through this pin. When HIGH, it denotes that the peripheral is ready to transfer data.
RESET (Input): System reset. The signal is active HIGH.
CLK (input): Clock 5, 8 or 10 MHz.
INTR: Interrupt Request.
NMI (Input): Non-maskable interrupt request.
TEST (Input): Wait for test control. When LOW the microprocessor continues execution otherwise waits.
VCC: Power supply +5V dc.
GND: Ground.

Operating Modes of 8086

There are two operating modes of operation for Intel 8086, namely the minimum mode and the maximum mode.
When only one 8086 CPU is to be used in a microprocessor system, the 8086 is used in the Minimum mode of operation.
In a multiprocessor system 8086 operates in the Maximum mode.

ADDRESSING:
The 8086 microprocessors have 8 addressing modes. Two addressing modes have been provided for instructions which operate on register or immediate data.
These two addressing modes are:

Register Addressing: In register addressing, the operand is placed in one of the 16-bit or 8-bit general purpose registers.
Example

- MOV AX, CX
- ADD AL, BL
- ADD CX, DX

Immediate Addressing: In immediate addressing, the operand is specified in the instruction itself.

Example

- MOV AL, 35H
- MOV BX, 0301H
- MOV [0401], 3598H
- ADD AX, 4836H

The remaining 6 addressing modes specify the location of an operand which is placed in a memory.
These 6 addressing modes are:
Direct Addressing: In direct addressing mode, the operand?s offset is given in the instruction as an 8-bit or 16-bit displacement element.
Example

- ADD AL, [0301]

The instruction adds the content of the offset address 0301 to AL. the operand is placed at the given offset (0301) within the data segment DS.

Register Indirect Addressing: The operand's offset is placed in any one of the registers BX, BP, SI or DI as specified in the instruction.
Example

- MOV AX, [BX]

It moves the contents of memory locations addressed by the register BX to the register AX.

Based Addressing: The operand's offset is the sum of an 8-bit or 16-bit displacement and the contents of the base register BX or BP. BX is used as base register for data segment, and the BP is used as a base register for stack segment.
Effective address (Offset) = [BX + 8-bit or 16-bit displacement].
Example

- MOV AL, [BX+05]; an example of 8-bit displacement.
- MOV AL, [BX + 1346H]; example of 16-bit displacement.

Indexed Addressing: The offset of an operand is the sum of the content of an index register SI or DI and an 8-bit or 16-bit displacement.
Offset (Effective Address) = [SI or DI + 8-bit or 16-bit displacement]
Example

- MOV AX, [SI + 05]; 8-bit displacement.
- MOV AX, [SI + 1528H]; 16-bit displacement.

Based Indexed Addressing: The offset of operand is the sum of the content of a base register BX or BP and an index register SI or DI.
Effective Address (Offset) = [BX or BP] + [SI or DI]
Here, BX is used for a base register for data segment, and BP is used as a base register for stack segment.
Example

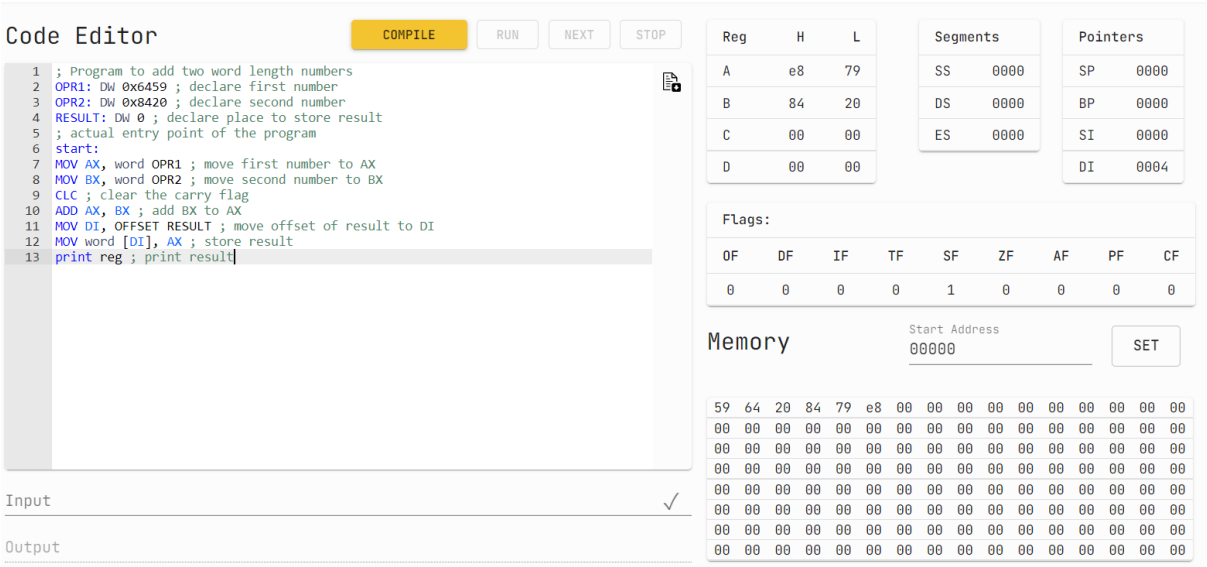
- ADD AX, [BX + SI]
- MOV CX, [BX + SI]

Based Indexed with Displacement: In this mode of addressing, the operand's offset is given by:
Effective Address (Offset) = [BX or BP] + [SI or DI] + 8-bit or 16-bit displacement
Example

- MOV AX, [BX + SI + 05]; 8-bit displacement
- MOV AX, [BX + SI + 1235H]; 16-bit displacement

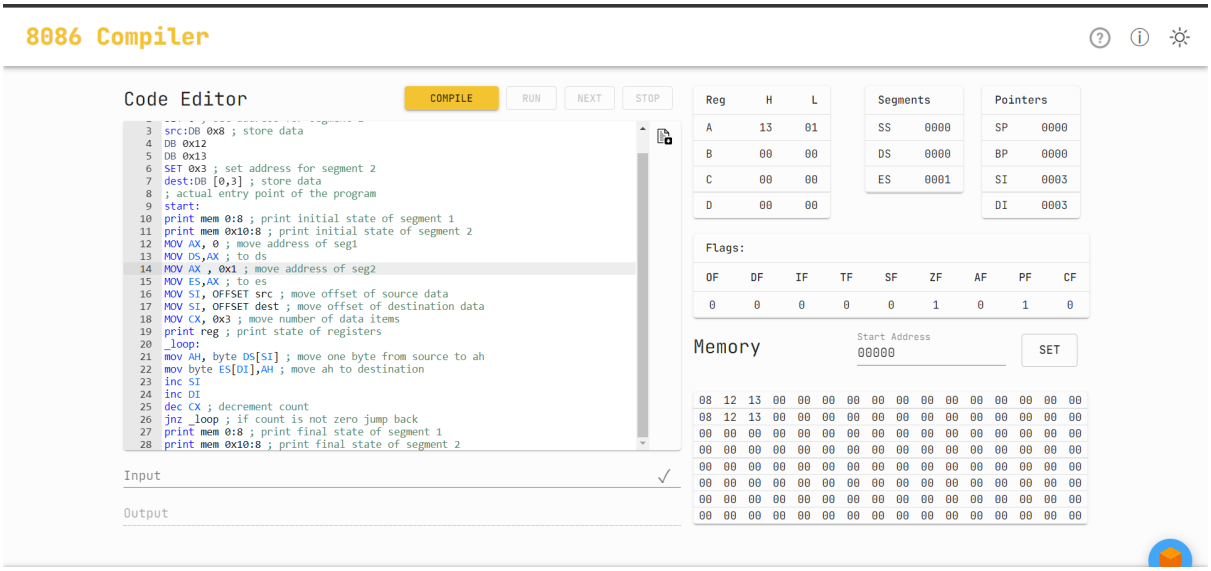
1. Program to add two word length number

```
; Program to add two word length numbers
OPR1: DW 0x6459 ; declare first number
OPR2: DW 0x8420 ; declare second number
RESULT: DW 0 ; declare place to store result
; actual entry point of the program
start:
MOV AX, word OPR1 ; move first number to AX
MOV BX, word OPR2 ; move second number to BX
CLC ; clear the carry flag
ADD AX, BX ; add BX to AX
MOV DI, OFFSET RESULT ; move offset of result to DI
MOV word [DI], AX ; store result
print reg ; print result
```



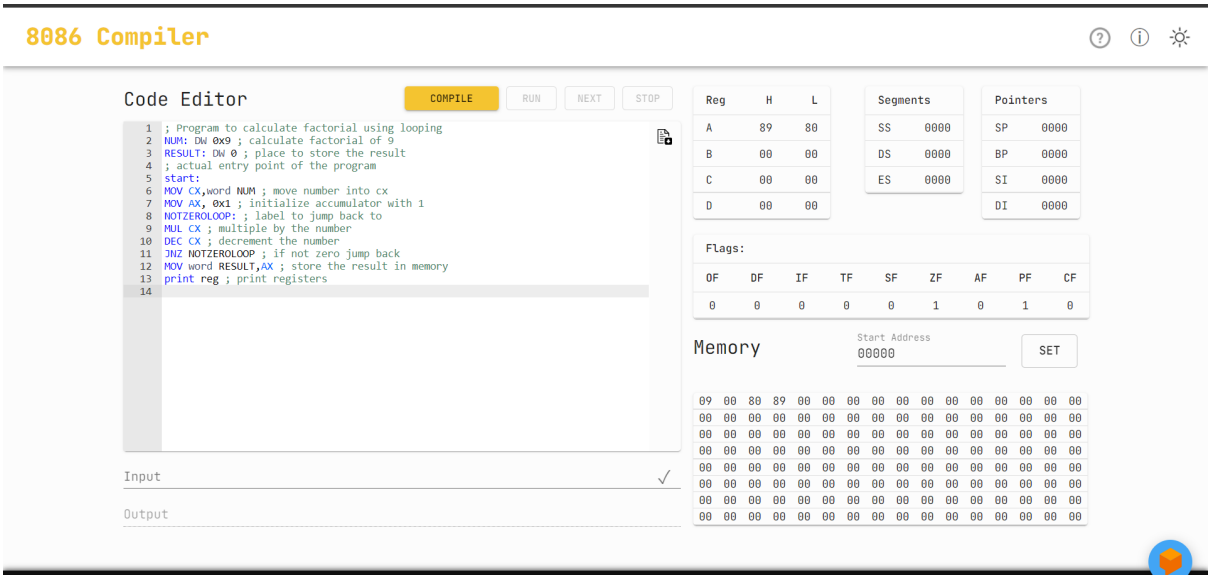
2. A Program to move data from one segment to another.

```
; A Program to move data from one segment to another
SET 0 ; set address for segment 1
src:DB 0x8 ; store data
DB 0x12
DB 0x13
SET 0x3 ; set address for segment 2
dest:DB [0,3] ; store data
; actual entry point of the program
start:
print mem 0:8 ; print initial state of segment 1
print mem 0x10:8 ; print initial state of segment 2
MOV AX, 0 ; move address of seg1
MOV DS,AX ; to ds
MOV AX , 0x1 ; move address of seg2
MOV ES,AX ; to es
MOV SI, OFFSET src ; move offset of source data
MOV SI, OFFSET dest ; move offset of destination data
MOV CX, 0x3 ; move number of data items
print reg ; print state of registers
_loop:
mov AH, byte DS[SI] ; move one byte from source to ah
mov byte ES[DI],AH ; move ah to destination
inc SI
inc DI
dec CX ; decrement count
jnz _loop ; if count is not zero jump back
print mem 0:8 ; print final state of segment 1
print mem 0x10:8 ; print final state of segment 2
```



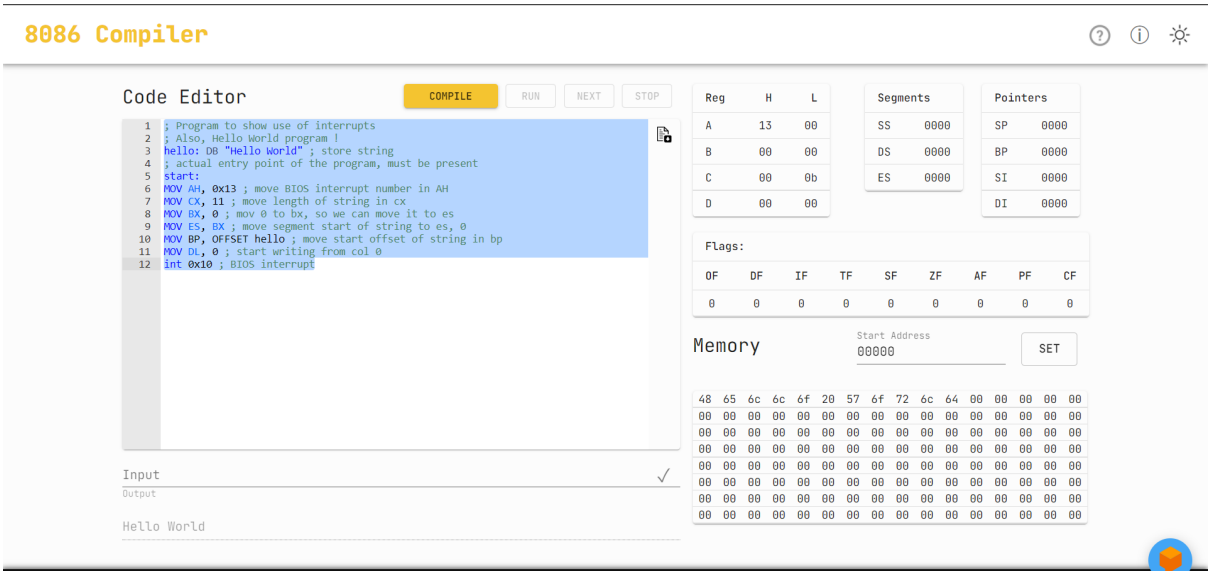
3. Program to calculate factorial using looping.

```
; Program to calculate factorial using looping
NUM: DW 0x9 ; calculate factorial of 9
RESULT: DW 0 ; place to store the result
; actual entry point of the program
start:
MOV CX,word NUM ; move number into cx
MOV AX, 0x1 ; initialize accumulator with 1
NOTZEROLOOP: ; label to jump back to
MUL CX ; multiple by the number
DEC CX ; decrement the number
JNZ NOTZEROLOOP ; if not zero jump back
MOV word RESULT,AX ; store the result in memory
print reg ; print registers
```



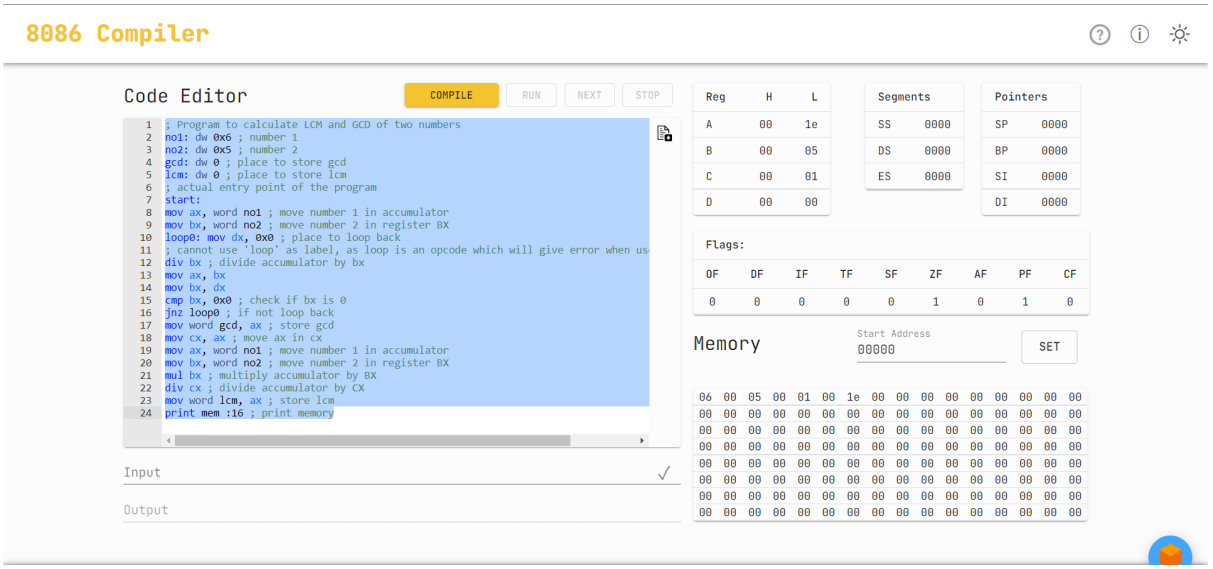
4. Program to show use of interrupts.

```
; Program to show use of interrupts
; Also, Hello World program !
hello: DB "Hello World" ; store string
; actual entry point of the program, must be present
start:
MOV AH, 0x13 ; move BIOS interrupt number in AH
MOV CX, 11 ; move length of string in cx
MOV BX, 0 ; mov 0 to bx, so we can move it to es
MOV ES, BX ; move segment start of string to es, 0
MOV BP, OFFSET hello ; move start offset of string in bp
MOV DL, 0 ; start writing from col 0
int 0x10 ; BIOS interrupt
```



5. Program to calculate LCM and GCD of two numbers.

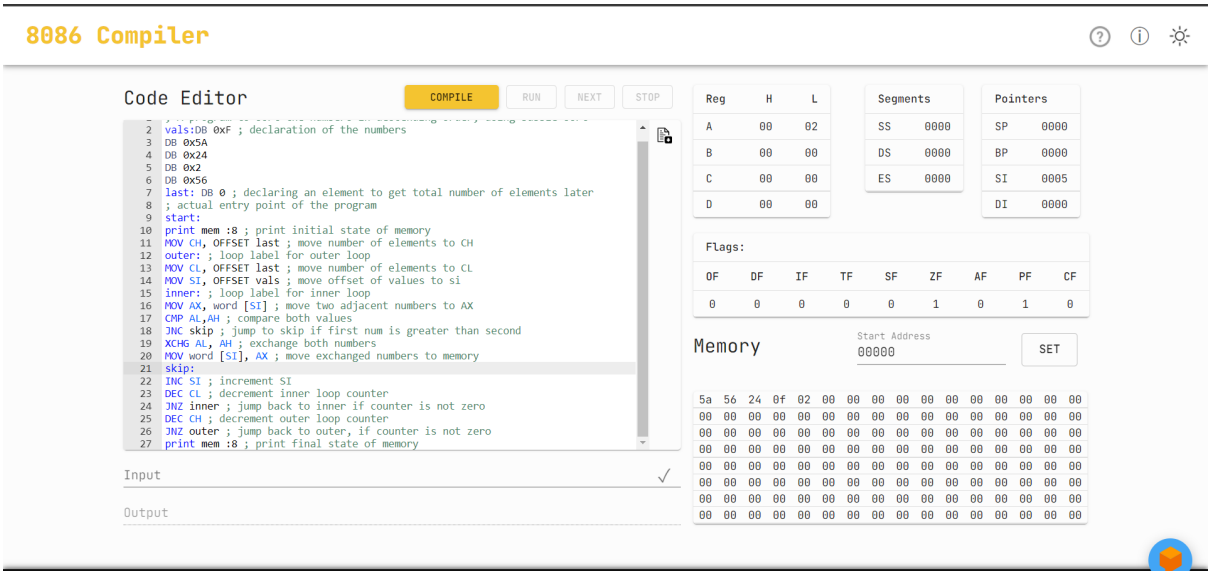
```
; Program to calculate LCM and GCD of two numbers
no1: dw 0x6 ; number 1
no2: dw 0x5 ; number 2
gcd: dw 0 ; place to store gcd
lcm: dw 0 ; place to store lcm
; actual entry point of the program
start:
mov ax, word no1 ; move number 1 in accumulator
mov bx, word no2 ; move number 2 in register BX
loop0: mov dx, 0x0 ; place to loop back
; cannot use 'loop' as label, as loop is an opcode which will give error
when used with jumps
div bx ; divide accumulator by bx
mov ax, bx
mov bx, dx
cmp bx, 0x0 ; check if bx is 0
jnz loop0 ; if not loop back
mov word gcd, ax ; store gcd
mov cx, ax ; move ax in cx
mov ax, word no1 ; move number 1 in accumulator
mov bx, word no2 ; move number 2 in register BX
mul bx ; multiply accumulator by BX
div cx ; divide accumulator by CX
mov word lcm, ax ; store lcm
print mem :16 ; print memory
```



6. A program to sort the numbers in descending order, using bubble sort

```
; A program to sort the numbers in descending order, using bubble sort
vals:DB 0xF ; declaration of the numbers
DB 0x5A
DB 0x24
DB 0x2
DB 0x56
last: DB 0 ; declaring an element to get total number of elements later
; actual entry point of the program
start:
print mem :8 ; print initial state of memory
MOV CH, OFFSET last ; move number of elements to CH
outer: ; loop label for outer loop
MOV CL, OFFSET last ; move number of elements to CL
MOV SI, OFFSET vals ; move offset of values to si
inner: ; loop label for inner loop
MOV AX, word [SI] ; move two adjacent numbers to AX
CMP AL,AH ; compare both values
JNC skip ; jump to skip if first num is greater than second
XCHG AL, AH ; exchange both numbers
MOV word [SI], AX ; move exchanged numbers to memory
skip:
INC SI ; increment SI
DEC CL ; decrement inner loop counter
JNZ inner ; jump back to inner if counter is not zero
```

```
DEC CH ; decrement outer loop counter
JNZ outer ; jump back to outer, if counter is not zero
print mem :8 ; print final state of memory
```



EXPERIMENT - 9

AIM: Write Programs using Assembly Language:
1. To implement Macros for calculating Factorial of a number.
2. To study and Implement DOS interrupts. Calculate and display Sum of 2 user entered inputs using DOS interrupt

Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	17/12/21	17/12/21	

Theory :
Macros : Writing a macro is another way of ensuring modular programming in assembly language. A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program. In NASM, macros are defined with %macro and %endmacro directives. The macro begins with the %macro directive and ends with the %endmacro directive. The Syntax for macro definition –

```
%macro macro_name number_of_params
<macro body>
%endmacro
```

Where, number_of_params specifies the number parameters, macro_name specifies the name of the macro. The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time

Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions. If you declared a macro and never used it in your code, compiler will simply ignore it. Unlike procedures, macros should be defined above the code that uses it.

The advantage of using Macro is that it avoids the overhead time involved in calling and returning (as in the procedures). Therefore, the execution of Macros is faster as compared to procedures. Another advantage is that there is no need for accessing stack or providing any separate memory to it for storing and returning the address locations while shifting the processor controls in the program.

But it should be noted that every time you call a macro, the assembler of the microprocessor places the entire set of Macro instructions in the mainline program from where the call to Macro is being made. This is known as Macro expansion. Due to this, the program code (which uses Macros) takes more memory space than the code which uses procedures for implementing the same task using the same set of instructions.

DOS Interrupts

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an ISR (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt. The following image shows the types of interrupts we have in a 8086 microprocessor –

DOS Interrupt is a Software Interrupt. INT 21H is a DOS interrupt. It is one of the most commonly used interrupts while writing code in 8086 assembly language. To use the DOS interrupt 21H load AH with the desired sub-function. Load other required parameters in other registers, and make a call to INT 21H.

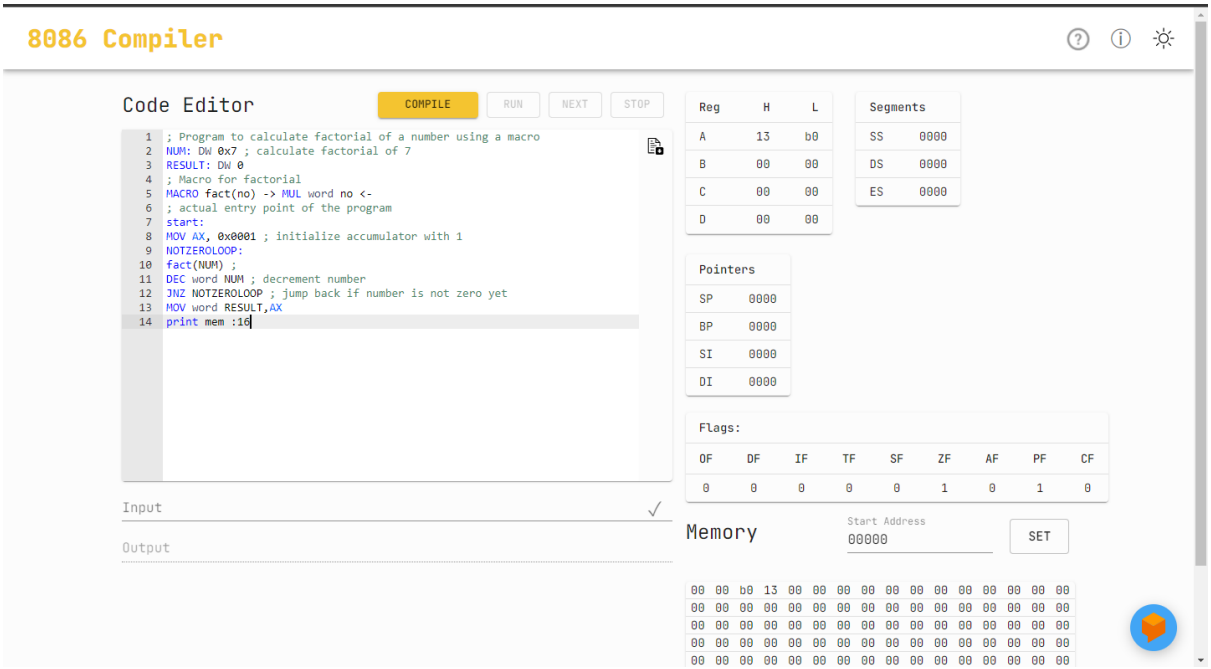
AH	Description	AH	Description
01	Read character from STDIN	02	Write character to STDOUT
05	Write character to printer	06	Console Input/Output
07	Direct char read (STDIN), no echo	08	Char read from STDIN, no echo
09	Write string to STDOUT	0A	Buffered input
0B	Get STDIN status	0C	Flush buffer for STDIN
0D	Disk reset	0E	Select default drive
19	Get current default drive	25	Set interrupt vector
2A	Get system date	2B	Set system date
2C	Get system time	2D	Set system time
2E	Set verify flag	30	Get DOS version
35	Get Interrupt vector		
36	Get free disk space	39	Create subdirectory

3A	Remove subdirectory	3B	Set working directory
3C	Create file	3D	Open file
3E	Close file	3F	Read file
40	Write file	41	Delete file
42	Seek file	43	Get/Set file attributes
47	Get current directory	4C	Exit program
4D	Get return code	54	Get verify flag
56	Rename file	57	Get/Set file date

CODE:

1. Factorial using Macros.

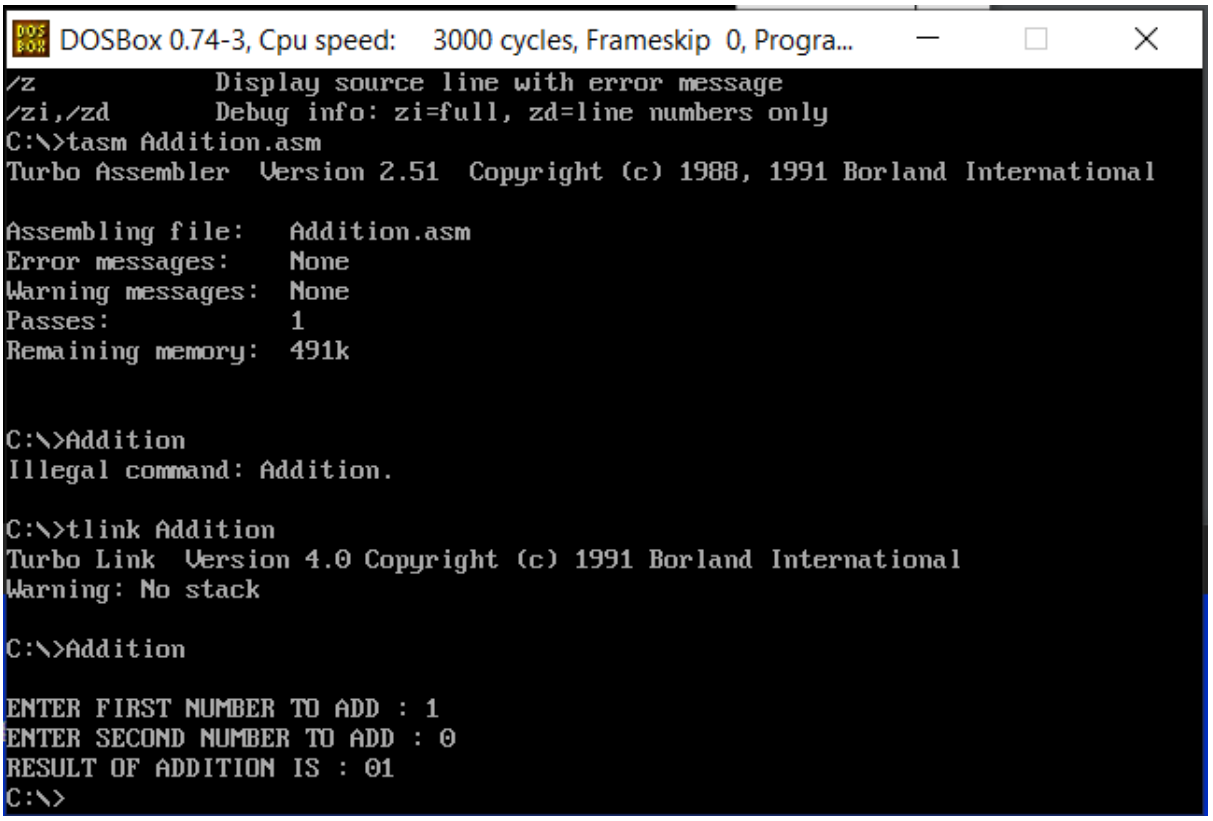
```
; Program to calculate factorial of a number using a macro
NUM: DW 0x7 ; calculate factorial of 7
RESULT: DW 0
; Macro for factorial
MACRO fact(no) -> MUL word no <-
; actual entry point of the program
start:
MOV AX, 0x0001 ; initialize accumulator with 1
NOTZEROLOOP:
fact(NUM) ;
DEC word NUM ; decrement number
JNZ NOTZEROLOOP ; jump back if number is not zero yet
MOV word RESULT,AX
print mem :16
```



2. DOS Interrupt

```
DATA SEGMENT
NUM1 DB ?
NUM2 DB ?
RESULT DB ?
MSG1 DB 10,13,"ENTER FIRST NUMBER TO ADD : $"
MSG2 DB 10,13,"ENTER SECOND NUMBER TO ADD : $"
MSG3 DB 10,13,"RESULT OF ADDITION IS : $"
ENDS
CODE SEGMENT
ASSUME DS:DATA, CS:CODE
START:
MOV AX,DATA
MOV DS,AX
LEA DX,MSG1
MOV AH,9
INT 21H
MOV AH,1
INT 21H
SUB AL,30H
MOV NUM1,AL
LEA DX,MSG2
MOV AH,9
INT 21H
```

```
MOV AH,1
INT 21H
SUB AL,30H
MOV NUM2,AL
ADD AL,NUM1
MOV RESULT,AL
MOV AH,0
AAA
ADD AH,30H
ADD AL,30H
MOV BX,AX
LEA DX,MSG3
MOV AH,9
INT 21H
MOV AH,2
MOV DL,BH
INT 21H
MOV AH,2
MOV DL,BL
INT 21H
MOV AH,4CH
INT 21H
ENDS
END START
```


DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Progra...
/z Display source line with error message
/zi,/zd Debug info: zi=full, zd=line numbers only
C:\>tasm Addition.asm
Turbo Assembler Version 2.51 Copyright (c) 1988, 1991 Borland International
Assembling file: Addition.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 491k
C:\>Addition
Illegal command: Addition.
C:\>tlink Addition
Turbo Link Version 4.0 Copyright (c) 1991 Borland International
Warning: No stack
C:\>Addition
ENTER FIRST NUMBER TO ADD : 1
ENTER SECOND NUMBER TO ADD : 0
RESULT OF ADDITION IS : 01
C:\>

EXPERIMENT - 10

Aim: Write a Program using ALP to Simulate Microcontroller interfacing with 7 segment display using http://vlabs.iitb.ac.in/vlabs-dev/labs/8051-Microcontroller-Lab/labs/exp1/simulation.php. Display your SAP ID using this tool

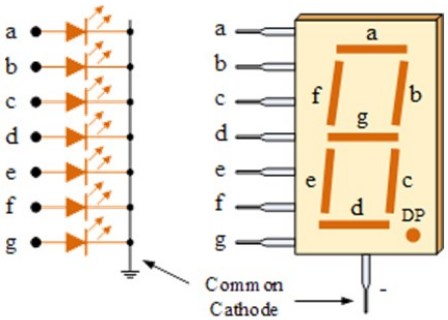
Submission Sheet

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004190057	Junaid Girkar	3/10/21	3/10/21	

THEORY:
The 7-segment display consists of seven LEDs arranged in a rectangular fashion. Each of the seven LEDs is called a segment because when illuminated the segment forms part of a numerical digit (both Decimal and Hex) to be displayed. An additional 8th LED is sometimes used within the same package which is the indication of a decimal point(DP), when two or more 7-segment displays are connected together numbers greater than ten can be displayed.
So by forward biasing the appropriate pins of the LED segments in a particular order, some segments will be glowing and others will remain as it is, allowing the desired character pattern of the number to be generated on the display. This then allows us to display each of the ten decimal digits 0 to 9 on the same 7-segment display.

In the common cathode display, all the cathode connections of the LED segments are joined together to logic “0” or ground. The individual segments are illuminated by application of a “HIGH”, or logic “1” signal via a current limiting resistor to forward bias the individual Anode terminals (a-g).

Common Cathode 7-segment Display



Commom Cathode							
Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	
1		1	1				
2	1	1		1	1		1
3	1	1	1	1			1
4		1	1			1	1
5	1		1	1		1	1
6	1		1	1	1	1	1
7	1	1	1				
8	1	1	1	1	1	1	1
9	1	1	1	1		1	1

Common Cathode Decoding Table

CHAR	A	B	C	D	E	F	G	HEX
SAP ID = 60004190057								
6	1		1	1	1	1	1	5F
0	1	1	1	1	1	1		7E
4		1	1			1	1	33
1		1	1					30
9	1	1	1	1		1	1	7B
5	1		1	1		1	1	5B
7	1	1	1					70
HEXADECIMAL = DF8874769								
D	1	1	1	1	1	1		7E
F	1				1	1	1	47
8	1	1	1	1	1	1	1	7F

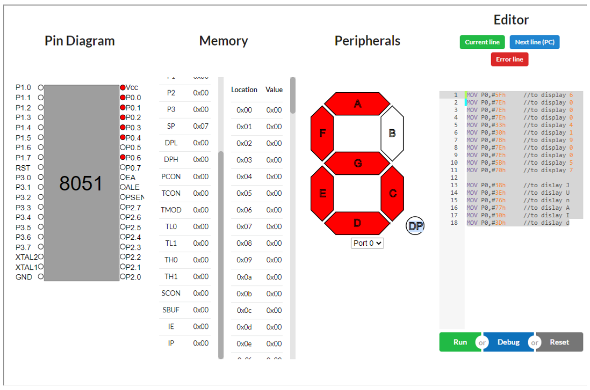
7	1	1	1					70
4		1	1			1	1	33
6	1		1	1	1	1	1	5F
9	1	1	1	1		1	1	7B
NAME = JUNAID								
J		1	1	1				38
U		1	1	1	1	1		3E
N	1	1	1		1	1		76
A	1	1	1		1	1	1	77
I		1	1					30
D		1	1	1	1		1	3D

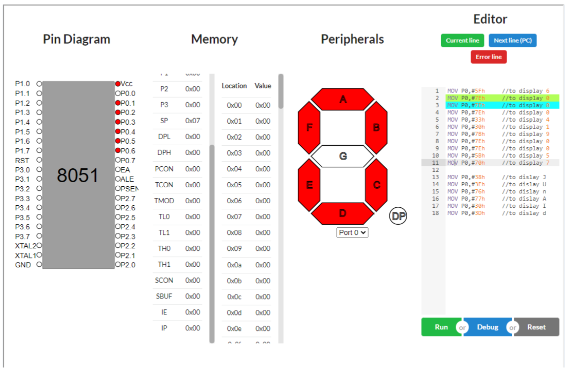
CODE:

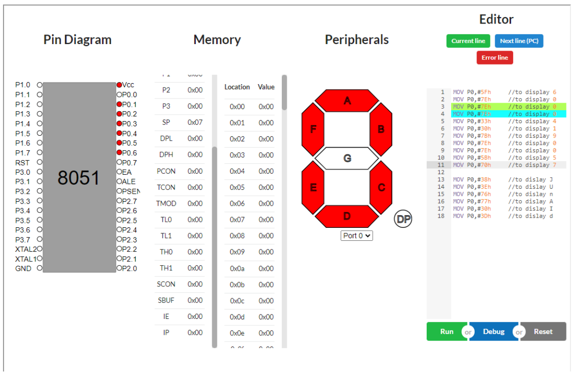
```
MOV P0,#5Fh     //to display 6
MOV P0,#7Eh     //to display 0
MOV P0,#7Eh     //to display 0
MOV P0,#7Eh     //to display 0
MOV P0,#33h     //to display 4
MOV P0,#30h     //to display 1
MOV P0,#7Bh     //to display 9
MOV P0,#7Eh     //to display 0
MOV P0,#7Eh     //to display 0
MOV P0,#5Bh     //to display 5
MOV P0,#70h     //to display 7

MOV P0,#3Dh     //to display d
MOV P0,#47h     //to display F
MOV P0,#7Fh     //to display 8
MOV P0,#7Fh     //to display 8
MOV P0,#70h     //to display 7
MOV P0,#33h     //to display 4
MOV P0,#70h     //to display 7
MOV P0,#5Fh     //to display 6
MOV P0,#7Bh     //to display 9

MOV P0,#38h     //to display J
MOV P0,#3Eh     //to display U
```

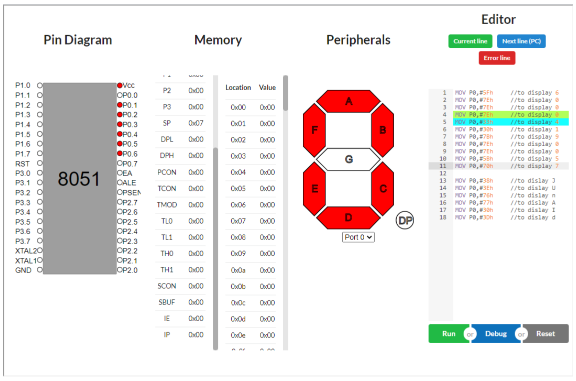
```
MOV P0,#76h      //to display n
MOV P0,#77h      //to display A
MOV P0,#30h      //to display I
MOV P0,#3Dh      //to display d
```

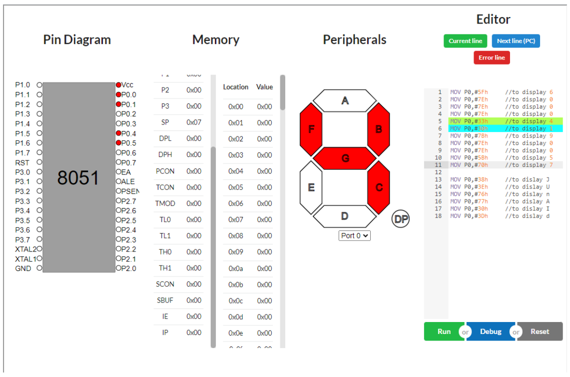
OUTPUT:

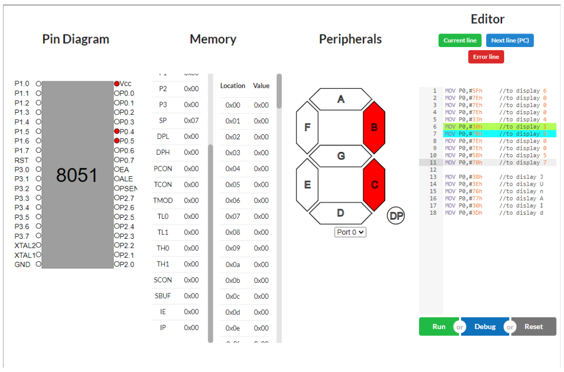


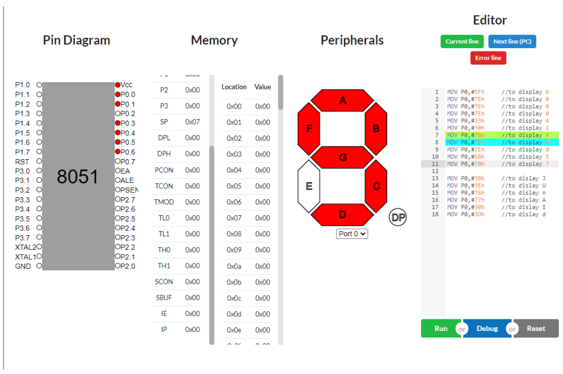


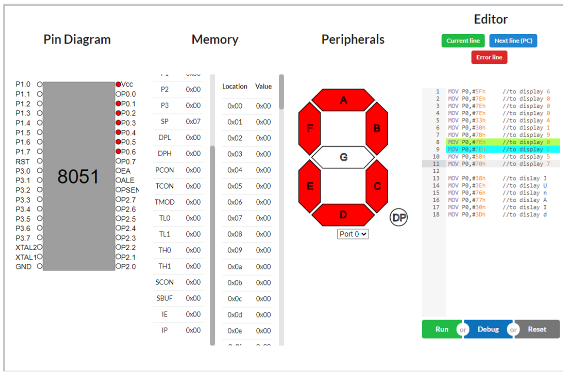


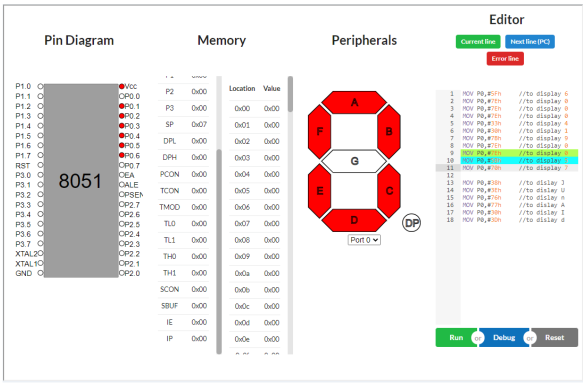


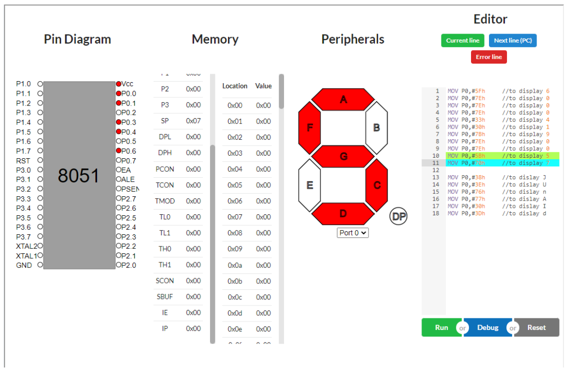


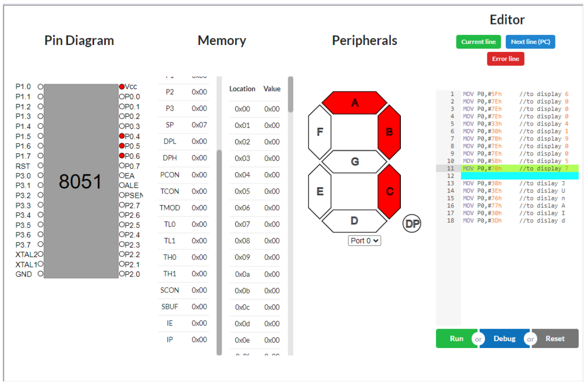


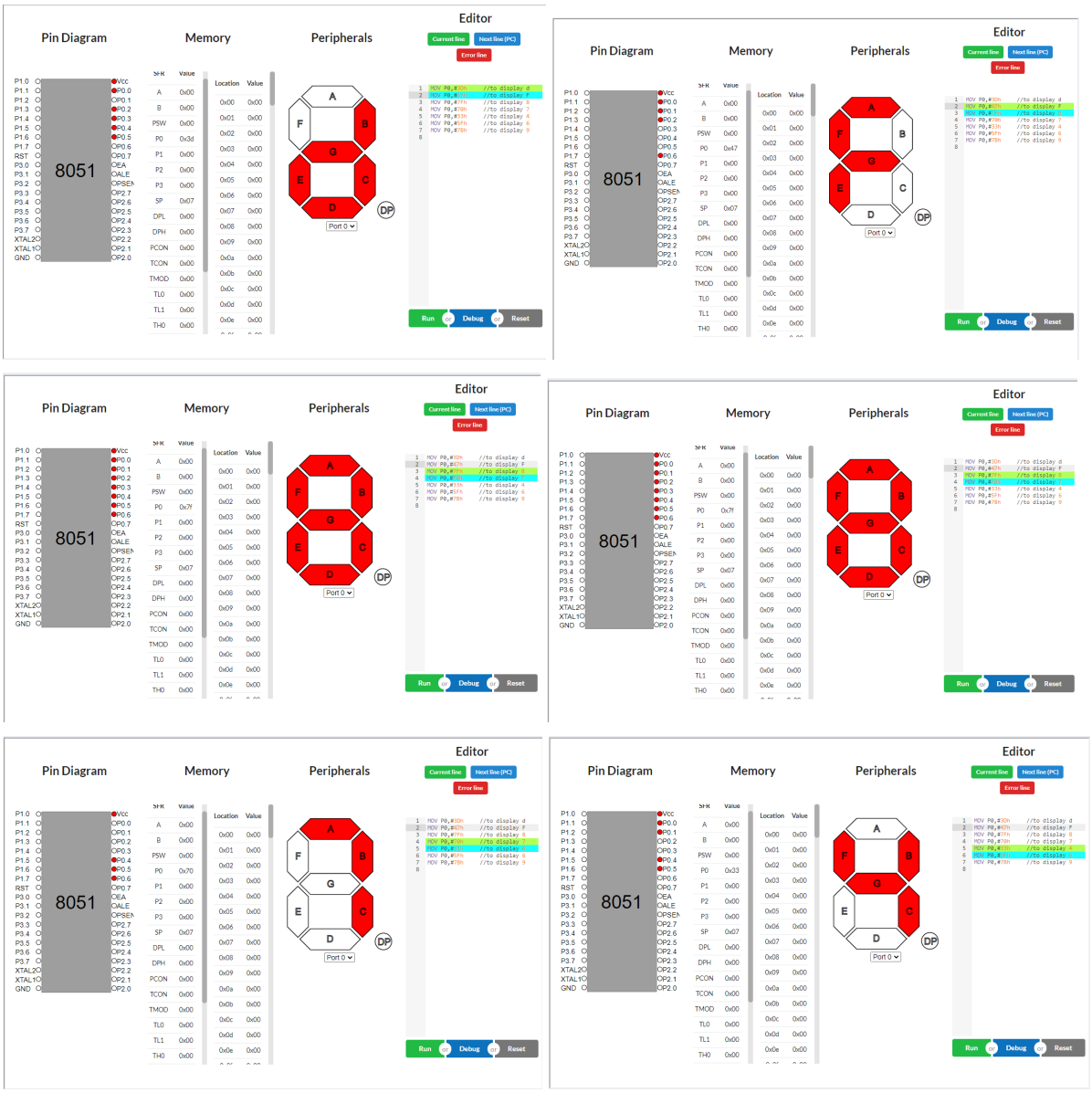

Pin Diagram
Memory
Peripherals
Editor
Current line
Next line (PC)
Error line
8051
Run
Debug
Reset

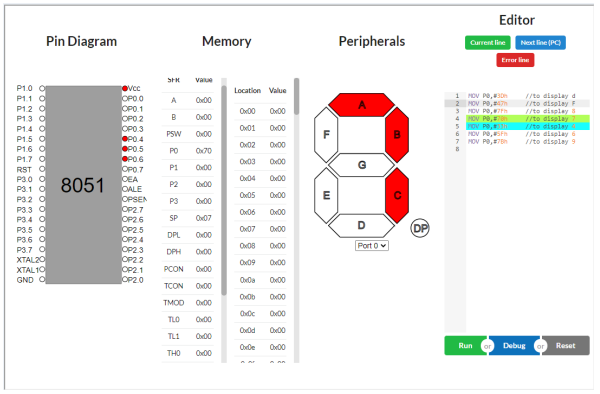

Pin Diagram
Memory
Peripherals
Editor
Current line
Next line (PC)
Error line
8051
Run
Debug
Reset

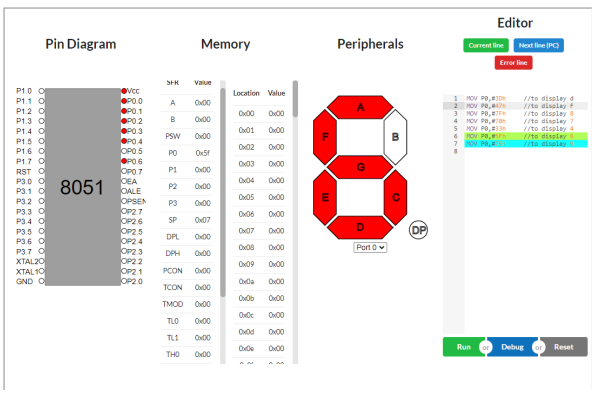

Pin Diagram
Memory
Peripherals
Editor
Current line
Next line (PC)
Error line
8051
Run
Debug
Reset

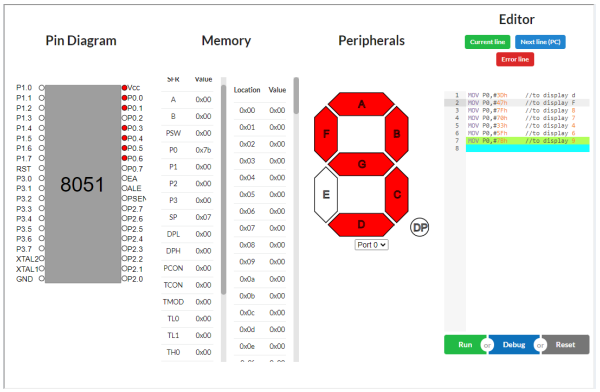

Pin Diagram
Memory
Peripherals
Editor
Current line
Next line (PC)
Error line
8051
Run
Debug
Reset

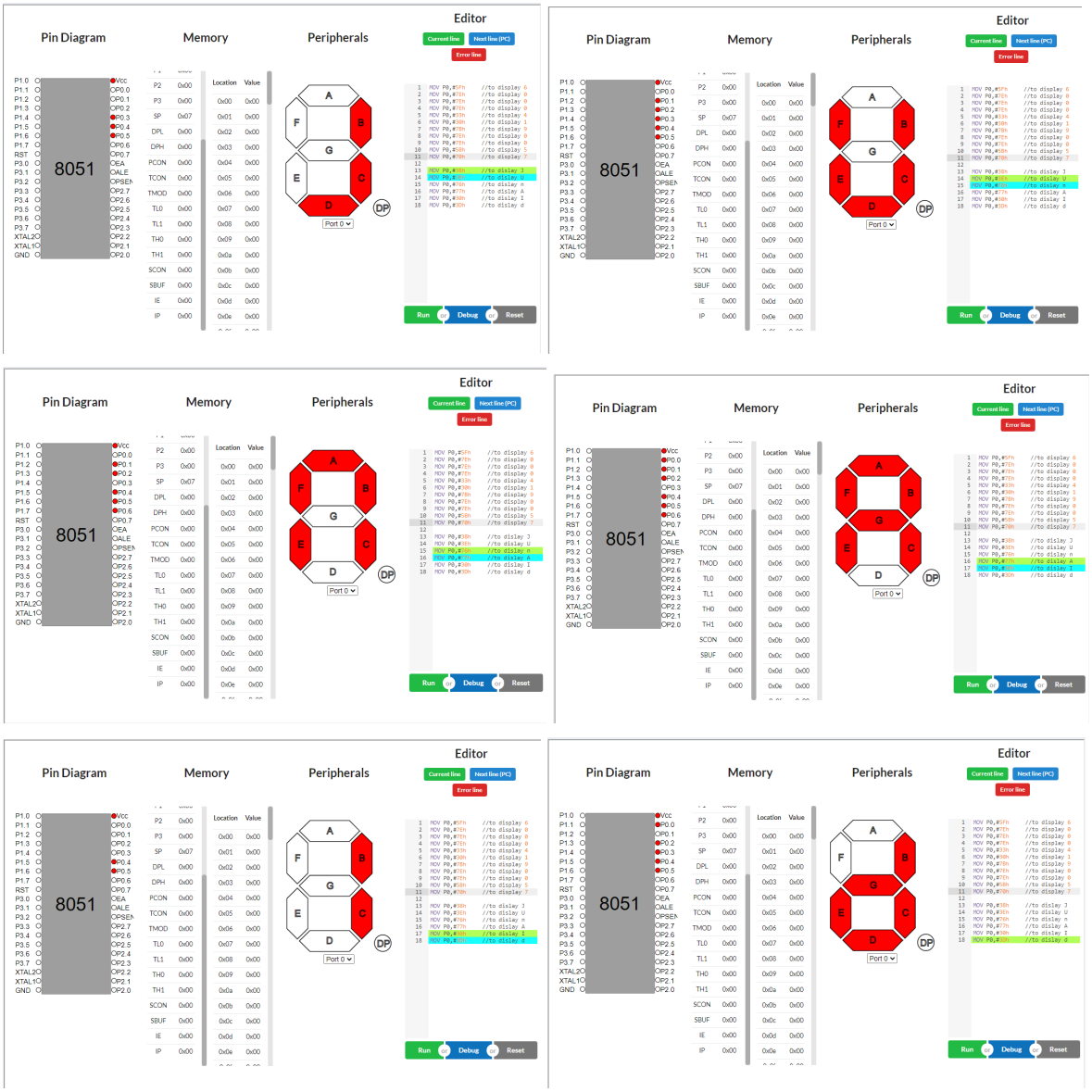

Pin Diagram
Memory
Peripherals
Editor
Current line
Next line (PC)
Error line
8051
Run
Debug
Reset


Pin Diagram
Memory
Peripherals
Editor
8051
Run
Debug
Reset


Pin Diagram
Memory
Peripherals
Editor
8051
Run
Debug
Reset


Pin Diagram
Memory
Peripherals
Editor
8051
Run
Debug
Reset


Pin Diagram
Memory
Peripherals
Editor
8051
Run
Debug
Reset



CONCLUSION: We learn about 7 segment display and simulations using the 8051 Microcontroller. We then simulated a few examples by writing the code.