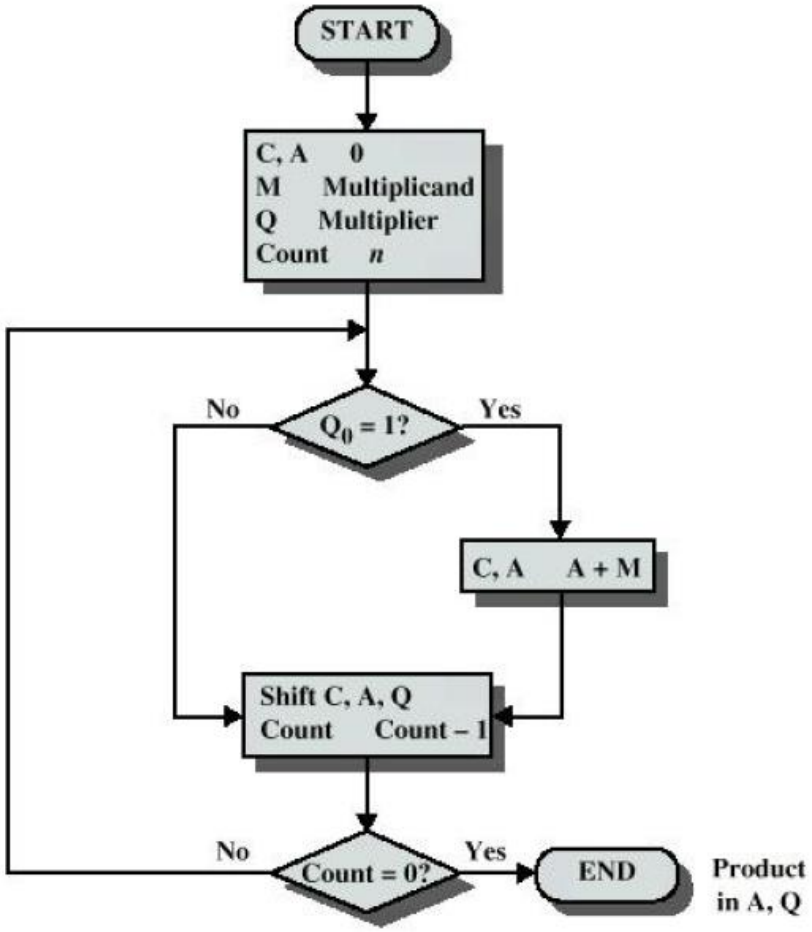
Experiment 1

Aim : To implement shift and add method of multiplication

Theory:

Multiplication involves generation of partial products, one for each digit of the multiplier. These partial products are then summed to produce the final product. In binary multiplication when multiplier bit id 0 partial product is 0, else if the bit is 1. The partial product is the multiplicand. The final answer is obtained by summing the partial products. However before summing each successive partial product is shifted one position left relative to the preceding partial product. Thus the product of two n digit numbers can be accommodated in 2n digits.

```
   1011   Multiplicand (11 dec)
 x 1101   Multiplier     (13 dec)
------
   1011   Partial products
  0000    Note: if multiplier bit is 1 copy
 1011     multiplicand (place value)
1011      otherwise zero
--------
10001111  Product (143 dec)
Note: need double length result
```


START
C, A 0
M Multiplicand
Q Multiplier
Count n
No
$Q_0 = 1$?
Yes
C, A A + M
Shift C, A, Q
Count Count – 1
No
Count = 0?
Yes
END
Product in A, Q

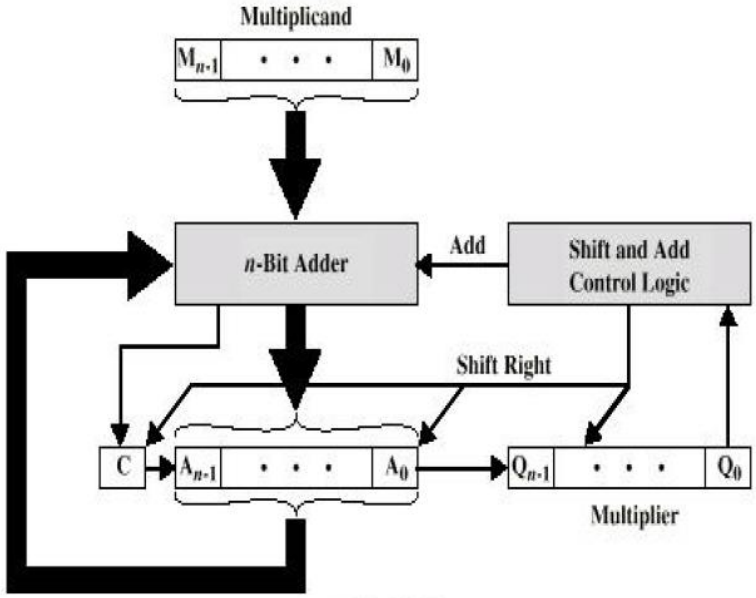



**FIG-8a**

| C | A | Q | M | comment |
|---|---|---|---|---|
| 0 | 0000 | 1010 | 1100 | Initial value |
| 0 | 0000 | 0101 | 1100 | Since Q0=0<br>Shift C,A,Q by 1bit |
| 0 | 1100 | 0101 | 1100 | Since Q0=1<br>C,A A+M |
| 0 | 0110 | 0010 | 1100 | Shift C,A,Q by 1 bit |
| 0 | 0011 | 0001 | 1100 | Since Q0=0<br>Shift C,A,Q by 1 bit |
| 0 | 1111 | 0001 | 1100 | Since Q0=1<br>C,A A+M |
| 0 | 0111 | 1000 | 1100 | Shift C,A,Q by 1 bit |

Answer(AQ)=01111000

```
import java.util.*; //Importing all classes of Util Package
public class BitMultiply //Creating a class named BitMultiply
```

```
 {
 public static void main(String[] args) //Main Method
  {
  System.out.println("Enetr the numbers in decimal form which you want to
multiply");
  Scanner InpOb = new Scanner(System.in); //Making Object InpOb
  int num1 = InpOb.nextInt();
  int num2 = InpOb.nextInt();
  int i1=0, i2=0, i3=0;
         int arr1[] = new int[100];
         int arr2[] = new int[100];
         int arr3[] = new int[100];
  int a = num1;
  System.out.println(num1 + " in binary form is ");
  while (a != 0)      //Converting num1 in Binary
          {
          i1++;
          arr1[i1] = a % 2;
              a = a / 2;
          }
         for (int j = i1; j > 0; j--)
          {
          System.out.print(arr1[j]);
          }
  System.out.println();
  a = num2;
  System.out.println(num2 + " in binary form is ");
  while (a != 0)       //Converting num2 in Binary
          {
          i2++;
          arr2[i2] = a % 2;
              a = a / 2;
          }
```

```
        for (int j = i2; j > 0; j--)
         {
         System.out.print(arr2[j]);
         }
  System.out.println();
  System.out.println("Multiplying " +num1 +" & " + num2 +" in we get");
  int result = 0;
  while (num2 != 0) // Iterate the loop till b==0
         {
             if ((num2 & 01) != 0) // Logical ANDing of the value of num2
with 01
             {
                 result = result + num1; // Update the result with the new
value of num1.
             }
             num1 <<= 1;              // Left shifting the value contained
in 'num1' by 1.
             num2 >>= 1;             // Right shifting the value contained
in 'num2' by 1.
         }
         System.out.println(result);
  System.out.println();
  a = result;
  System.out.println(result + " in binary form is ");
  while (a != 0)
         {
         i3++;
         arr3[i3] = a % 2;
             a = a / 2;
         }
        for (int j = i3; j > 0; j--)
         {
         System.out.print(arr3[j]);
```

```
        }

  }
 }

/*Result
Enetr the numbers in decimal form which you want to multiply
23
54
23 in binary form is
10111
54 in binary form is
110110
Multiplying 23 & 54 in we get
1242

1242 in binary form is
1001101101
*/
```

Conclusion : Thus we have implemented shift and add method of multiplication.

## Experiment 2

Aim: Write a program to implement Booth's multiplication

Theory:

Booth's algorithm is used to multiply signed numbers.It generates a 2n bit product and treats both positive and negative numbers uniformly. The algorithm suggests that we can reduce the number of operations required for multiplication by representing the multiplier as a difference between two numbers.

In general , -1 times the multiplicand is selected when moving from 0 to 1, +1 times the multiplicand is selected when moving from 1 to 0 and 0 times the multiplicand is selected for none of the above cases as the multiplier is scanned from right to left

Flowchart:

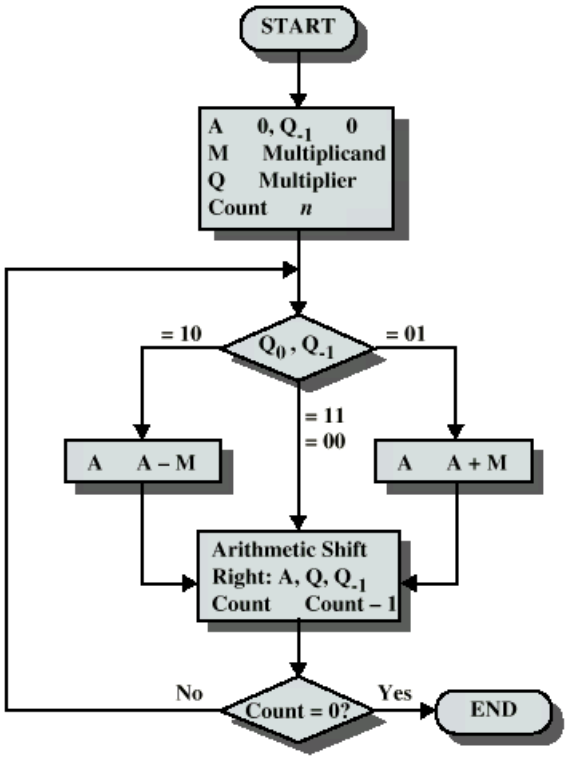



**Figure 8.12 Booth's Algorithm for Twos Complement Multiplication**

Example of Booth's Algorithm (7*3):-

| A | Q | $Q_{-1}$ | M | Comment |
|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial value |

| | | | | |
|---|---|---|---|---|
| 1001 | 0011 | 0 | 0111 | $Q_0,Q_{-1}=10$<br>A<- A-M |
| 1100 | 1001 | 1 | 0111 | Shift right<br>A,Q,$Q_{-1}$ |
| 1110 | 0100 | 1 | 0111 | $Q_0,Q_{-1}=11$<br>Shift right<br>A,Q,$Q_{-1}$ |
| 0101 | 0100 | 1 | 0111 | $Q_0,Q_{-1}=01$<br>A<- A+M |
| 0010 | 1010 | 0 | 0111 | Shift right<br>A,Q,$Q_{-1}$ |
| 0001 | 0101 | 0 | 0111 | $Q_0,Q_{-1}=00$<br>Shift right<br>A,Q,$Q_{-1}$ |

```
import java.util.*;
class BOOTH
{
public static int[] add(int a[],int m1[])
{
int carry=0;
int sum[]=new int [4];
for(int i=3;i>=0;i–)
{
sum[i]=(a[i]+m1[i]+carry)%2;
carry=(a[i]+m1[i]+carry)/2;
}
return sum;
}

public static void shift(int a[],int q[])
{
int b=a[3];
for(int i=2;i>=0;i–)
{
a[i+1]=a[i];
q[i+1]=q[i];
}
```

```
q[0]=b;
}

public static int[] comp(int m1[])
{
int z[]={0,0,0,1};
for(int i=0;i<4;i++)
{
if(m1[i]==0)
m1[1]=1;
else
m1[i]=0;
}
int c[]=add(m1,z);
return c;
}

public static void display(int a[],int q[],int Q1,int m[])
{
for(int i=0;i<4;i++)
{
System.out.print(a[i]);
}
System.out.print("\t");
for(int i=0;i<4;i++)
{
System.out.print(q[i]);
}
System.out.print("\t");
System.out.print(Q1);
System.out.print("\t");
for(int i=0;i<4;i++)
{
System.out.print(m[i]);
}
System.out.print("\t");
}

public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
int m[]=new int[4];
```

```
int q[]=new int[4];
int a[]={0,0,0,0};
int Q1=0;
int count=4;
int m1[]=new int[4];
System.out.println("ENTER MULTIPLICAND:");
for(int i=0;i<4;i++)
{
m[i]=sc.nextInt();
}
System.out.println("ENTER MULTIPLIER:");
for(int i=0;i<4;i++)
{
q[i]=sc.nextInt();
}
System.out.println("A \t Q \tQ-1\t M \toperation");
display(a,q,Q1,m);
System.out.print("Initial\n");
for(int i=count;i>0;i–)
{
for(int j=0;j<4;j++)
{
m1[j]=m[j];
}
if(q[3]==0&&Q1==1)
{
a=add(a,m1);
display(a,q,Q1,m);
System.out.print("A<~A+M\n");
}
if((q[3]==1&&Q1==1)||(q[3]==3&&Q1==0))
{ }
if(q[3]==1&&Q1==0)
{
int c[]=comp(m1);
a=add(a,c);
display(a,q,Q1,m);
System.out.print("A<~A-M\n");
}
Q1=q[3];
shift(a,q);
display(a,q,Q1,m);
```

```
System.out.print("shift\n");
}
}
}
```

/*
**booths Algorithm program in java OUTPUT**
BOOTH Algorithm program output:
ENTER MULTIPLICAND:
1
1
1
1
ENTER MULTIPLIER:
1
1
1
1
A Q Q-1 M operation
0000 1111 0 1111 Initial
0001 1111 0 1111 A<~A-M
0000 1111 1 1111 shift
0000 0111 1 1111 shift
0000 0011 1 1111 shift
0000 0001 1 1111 shift *

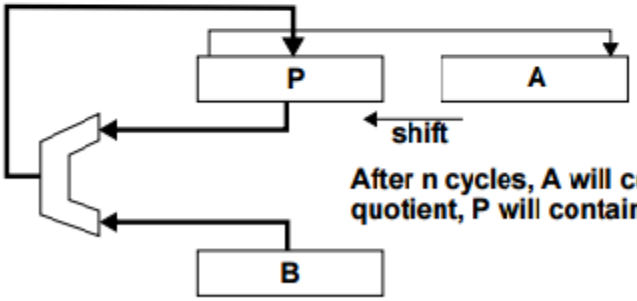
Conclusion : Thus we have successfully implemented Booth's algorithm for multiplication

## Experiment 3

Aim: To study and implement Restoring division

# Restoring Division Algorithm

- Put $x$ in register A, $d$ in register B, $0$ in register P, and perform $n$ divide steps ($n$ is the quotient wordlength)
- Each step consists of
  - (i) Shift the register pair (P,A) one bit left
  - (ii) Subtract the contents of B from P, put the result back in P
  - (iii) If the result is -ve, set the low-order bit of A to 0 otherwise to 0
  - (iv) If the result is -ve, restore the old value of P by adding the contents of B back in P




After n cycles, A will contain the quotient, P will contain the remainder

# Restoring Division Example

| P | A | Operation |
|---|---|---|
| 00000 | 1110 | Divide 14 = 1110 by 3 = 11. B register always contains 0011 |
| 00001 | 110 | step 1(i): shift |
| -00011 | | step 1(ii): subtract |
| --------- | | |
| -00010 | 1100 | step 1(iii): quotient is -ve, set quotient bit to 0 |
| 00001 | 1100 | step 1(iv): restore |
| 00011 | 100 | step 2(i): shift |
| -00011 | | step 2(ii): subtract |
| --------- | | |
| 00000 | 1001 | step 2(iii): quotient is +ve, set quotient bit to 1 |
| 00001 | 001 | step 3(i): shift |
| -00011 | | step 3(ii): subtract |
| --------- | | |
| -00010 | 0010 | step 3(iii): quotient is -ve, set quotient bit to 0 |
| 00001 | 0010 | step 3(iv): restore |
| 00010 | 010 | step 4(i): shift |
| -00011 | | step 4(ii): subtract |
| --------- | | |
| -00001 | 0100 | step 4(iii): quotient is -ve, set quotient bit to 0 |
| 00010 | 0100 | step 4(iv): restore |

- **The quotient is 0100 and the remainder is 00010**
- **The name *restoring* because if subtraction by *b* yields a negative result, the *P* register is restored by adding *b* back**

```
import java.util.*;
class RESTORING
{
public static void lshift(int a[],int q[])
{
for(int i=0;i<3;i++)
{
a[i]=a[i+1];
}
a[3]=q[0];
for(int i=0;i<3;i++)
{
q[i]=q[i+1];
}
q[3]=0;
}

public static int[] add(int a[],int m1[])
{
int carry =0;
```

```
int sum[]=new int [4];
for(int i=3;i>=0;i–)
{
sum[i]=(a[i]+m1[i]+carry)%2;
carry=(a[i]+m1[i]+carry)/2;
}
return sum;
}

public static int [] comp2(int m1[])
{
int z[]={0,0,0,1};
for(int i=0;i<3;i++)
{
if(m1[i]==0)
m1[i]=1;
else
m1[i]=0;
}
m1=add(m1,z);
return m1;
}

public static void display(int a[],int q[],int m[])
{
for(int i=0;i<4;i++)
{
System.out.print(a[i]);
}
System.out.print(“\t”);
for(int i=0;i<4;i++)
{
System.out.print(q[i]);
}
System.out.print(“\t”);

for(int i=0;i<4;i++)
{
System.out.print(m[i]);
}
System.out.print(“\t”);
}
```

```
public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
int a[]={0,0,0,0};
int m[]=new int[4];
int q[]=new int[4];
int count=4;
int m1[]=new int[4];
System.out.println("ENTER DIVISOR:");
for(int i=0;i<=3;i++)
{
m[i]=sc.nextInt();
}

System.out.println("ENTER DIVIDEND:");
for(int i=0;i<=3;i++)
{
q[i]=sc.nextInt();
}
System.out.println("A \t Q \t M \toperation");
display(a,q,m);
System.out.print("Initial\n");

for(int i=count;i>0;i–)
{
for(int j=0;j<4;j++)
{
m1[j]=m[j];
}
lshift(a,q);
display(a,q,m);
System.out.print("shift\n");
int c[]=new int[4];
c=comp2(m1);
a=add(a,c);
display(a,q,m);
System.out.print("Subtract\n");
if(a[0]==1)
{
q[3]=0;
a=add(a,m);
display(a,q,m);
```

```
System.out.print("Restore\n");
}
else
{
q[3]=1;
display(a,q,m);
System.out.print("Set Q0=1\n");
}
}
}
}
```

```
/* RESTORING Division program in java OUTPUT:
ENTER DIVISOR:
0
0
1
1
ENTER DIVIDEND:
0
1
1
1
A Q M operation
0000 0111 0011 Initial
0000 1110 0011 shift
1110 1110 0011 Subtract
0001 1110 0011 Restore
0011 1100 0011 shift
0001 1100 0011 Subtract
0001 1101 0011 Set Q0=1
0011 1010 0011 shift
0001 1010 0011 Subtract
0001 1011 0011 Set Q0=1
0011 0110 0011 shift
0001 0110 0011 Subtract
0001 0111 0011 Set Q0=1 */
```

Conclusion: Hence we have studied and implemented Restoring division.

Experiment 4

Aim: To study and implement Non- restoring Division.

## Non-Restoring Division Algorithm

- A variant that skips the restoring step and instead works with negative residuals
- If P is negative
  - (i-a) Shift the register pair (P,A) one bit left
  - (ii-a) Add the contents of register B to P
- If P is positive
  - (i-b) Shift the register pair (P,A) one bit left
  - (ii-b) Subtract the contents of register B from P
- (iii) If P is negative, set the low-order bit of A to 0, otherwise set it to 1
- After n cycles
  - The quotient is in A
  - If P is positive, it is the remainder, otherwise it has to be restored (add B to it) to get the remainder

# Non-Restoring Division Example

| P | A | Operation |
|---|---|---|
| 00000 | 1110 | Divide 14 = 1110 by 3 = 11. B register always contains 0011 |
| 00001 | 110 | step 1(i-b): shift |
| +00011 | | step 1(ii-b): subtract b (add two's complement) |
| 11110 | 1100 | step 1(iii): P is negative, so set quotient bit to 0 |
| 11101 | 100 | step 2(i-a): shift |
| +00011 | | step 2(ii-a): add b |
| 00000 | 1001 | step 2(iii): P is +ve, so set quotient bit to 1 |
| 00001 | 001 | step 3(i-b): shift |
| +11101 | | step 3(ii-b): subtract b |
| 11110 | 0010 | step 3(iii): P is -ve, so set quotient bit to 0 |
| 11100 | 010 | step 4(i-a): shift |
| +00011 | | step 4(ii-a): add b |
| 11111 | 0100 | step 4(iii): P is -ve, set quotient bit to 0 |
| +00011 | | Remainder is negative, so do final restore step |
| 00010 | | |

- The quotient is 0100 and the remainder is 00010
- *restoring* division seems to be more complicated since it involves extra addition in step (iv)
- This is not true since the sign resulting from the subtraction is tested at adder o/p and only if the sum is +ve, it is loaded back to the P register

```
import java.io.*;
class NONRESTORING
{
public static int[] lshift(int s1[],int s2)
{
int s[]=new int [4];
for(int i=0;i<3;i++)
{
s[i]=s1[i+1];
}
s[3]=s2;
return(s);
}

public static int[] add(int A[],int B[])
{
int s[]=new int [4];
int c=0;
for(int i=3;i>=0;i–)
{
```

```
s[i]=((A[i]+B[i]+c)%2);
c=(A[i]+B[i]+c)/2;
}
return(s);
}

public static void display(int a[],int q[],int m[])
{
for(int i=0;i<4;i++)
System.out.print(a[i]);
System.out.print(" ");
for(int i=0;i<4;i++)
System.out.print(q[i]);
System.out.print(" ");
for(int i=0;i<4;i++)
System.out.print(m[i]);
System.out.print(" ");
}

public static void main(String args[])throws Exception
{
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(isr);
int ac[]={0,0,0,0};
int M[]={0,0,0,0};
int Q[]={0,0,0,0};
int K[]={0,0,0,1};
int M1[]=new int[4];
int i;
int count=4;
System.out.println("ENTER THE DIVIDEND :");
for(i=0;i<4;i++)
Q[i]=Integer.parseInt(br.readLine());
System.out.println("ENTER THE DIVISOR :");
for(i=0;i<4;i++)
M[i]=Integer.parseInt(br.readLine());
System.out.println(" A Q M");
display(ac,Q,M);
System.out.println();
while(count>0)
{
if(ac[0]==0)
```

```
{
int q1=Q[0];
ac=lshift(ac,q1);
Q=lshift(Q,0);
display(ac,Q,M);
System.out.println("LEFT SHIFT");
for(int j=0;j<4;j++)
{
if(M[j]==0)
M1[j]=1;
else
M1[j]=0;
}
M1=add(M1,K);
ac=add(ac,M1);
display(ac,Q,M);
System.out.println("SUBTRACTION");
}
else
{
int q1=Q[0];
ac=lshift(ac,q1);
Q=lshift(Q,0);
display(ac,Q,M);
System.out.println("LEFT SHIFT");
ac=add(ac,M);
display(ac,Q,M);
System.out.println("ADDITION");
}
if(ac[0]==0)
{
Q[3]=1;
display(ac,Q,M);
System.out.println("AFTER Q[3]=1″);
}
else
{
Q[3]=0;
display(ac,Q,M);
System.out.println(" AFTER Q[3]=0″);
}
count–;
```

```
}
if(count==0)
{
if(ac[0]==0)
{
}
else
{
ac=add(ac,M);
display(ac,Q,M);
System.out.println("ADDITION");
}
}
}
}
```

/*

**non restoring division algorithm in java OUTPUT:**

```
ENTER THE DIVIDEND :
1
1
1
1
ENTER THE DIVISOR :
0
0
1
1
A Q M
0000 1111 0011
0001 1110 0011 LEFT SHIFT
1110 1110 0011 SUBTRACTION
1110 1110 0011 AFTER Q[3]=0
1101 1100 0011 LEFT SHIFT
0000 1100 0011 ADDITION
0000 1101 0011 AFTER Q[3]=1
0001 1010 0011 LEFT SHIFT
1110 1010 0011 SUBTRACTION
1110 1010 0011 AFTER Q[3]=0
1101 0100 0011 LEFT SHIFT
0000 0100 0011 ADDITION
0000 0101 0011 AFTER Q[3]=1 */
```

Experiment 5

Aim: To study LRU page replacement policy

Theory:

If the required page is not in the main memory a page fault occurs and the required page is loaded into the main memory from the secondary memory. However if there is no vacant space in the main memory to copy the page, it is necessary to replace the required page with an existing page in the main memory which is not in use. This is achieved by page replacement algorithms.

LRU(Least Recently Used) page replacement:

- In this algorithm, the page which has not been used for the longest time or least recently used is replaced.

Example:

```
#include<stdio.h>
main()
{
        int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];
        printf("Enter no of pages:");
        scanf("%d",&n);
        printf("Enter the reference string:");
        for(i=0;i<n;i++)
        scanf("%d",&p[i]);
        printf("Enter no of frames:");
        scanf("%d",&f);
        q[k]=p[k];
        printf("\n\t%d\n",q[k]);
        c++;
        k++;
        for(i=1;i<n;i++)
        {
                c1=0;
                for(j=0;j<f;j++)
                {
                        if(p[i]!=q[j])
                        c1++;
                }
                if(c1==f)
                {
```

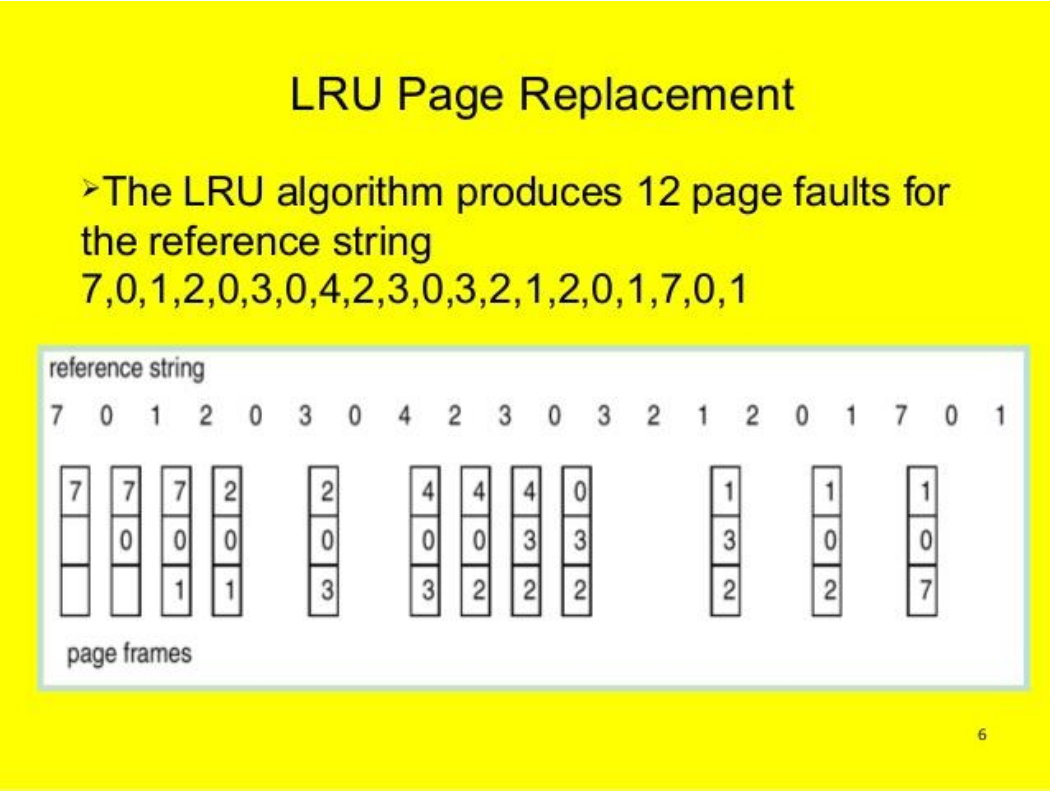
```
                        c++;
                        if(k<f)
                        {
                                q[k]=p[i];
                                k++;
                                for(j=0;j<k;j++)
                                printf("\t%d",q[j]);
                                printf("\n");
                        }
                        else
                        {
                                for(r=0;r<f;r++)
                                {
                                        c2[r]=0;
                                        for(j=i-1;j<n;j--)
                                        {
                                        if(q[r]!=p[j])
                                        c2[r]++;
                                        else
                                        break;
                                }
                        }
                        for(r=0;r<f;r++)
                         b[r]=c2[r];
                        for(r=0;r<f;r++)
                        {
                                for(j=r;j<f;j++)
                                {
                                        if(b[r]<b[j])
                                        {
                                                t=b[r];
                                                b[r]=b[j];
                                                b[j]=t;
                                        }
                                }
                        }
                        for(r=0;r<f;r++)
                        {
                                if(c2[r]==b[0])
                                q[r]=p[i];
                                printf("\t%d",q[r]);
                        }
                        printf("\n");
                }
        }
}
printf("\nThe no of page faults is %d",c);
}
```

## **OUTPUT:**

Enter no of pages:10
Enter the reference string:7 5 9 4 3 7 9 6 2 1
Enter no of frames:3

| | | |
|---|---|---|
| 7 | | |
| 7 | 5 | |
| 7 | 5 | 9 |
| 4 | 5 | 9 |
| 4 | 3 | 9 |
| 4 | 3 | 7 |
| 9 | 3 | 7 |
| 9 | 6 | 7 |
| 9 | 6 | 2 |
| 1 | 6 | 2 |

The no of page faults is 10




Conclusion: Thus we have successfully implemented the LRU page replacement algorithm.

# Experiment 6

Aim: To study FIFO page replacement policy

Theory:

If the required page is not in the main memory a page fault occurs and the required page is loaded into the main memory from the secondary memory. However if there is no vacant space in the main memory to copy the page, it is necessary to replace the required page with an existing page in the main memory which is not in use. This is achieved by page replacement algorithms.

FIFO(First In First Out) page replacement:

- This is the simplest page replacement algorithm.
- This replaces the new page with the oldest page in the main memory.
- However its performance is not always good as it may replace the most needed page which might be the oldest page

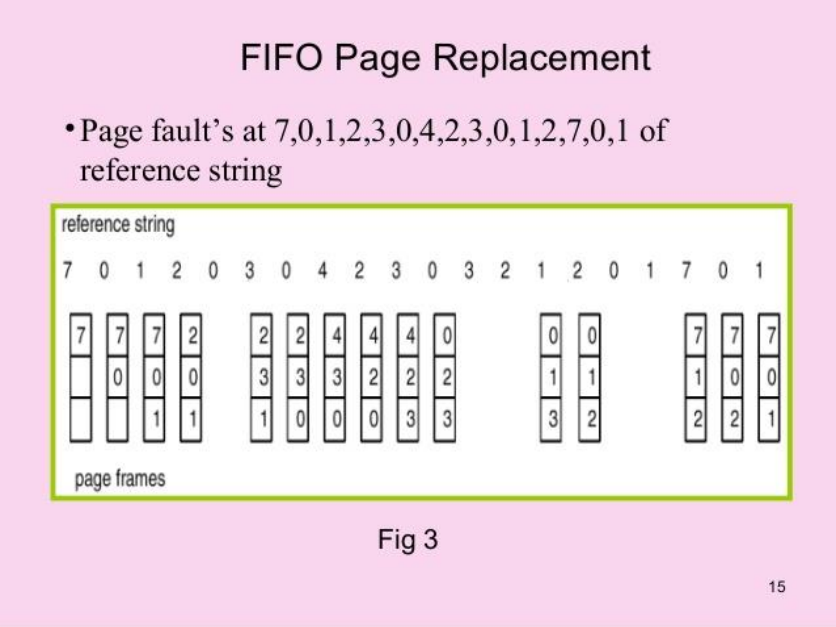
Example:

```
#include<stdio.h>
int main()
{
        int i,j,n,a[50],frame[10],no,k,avail,count=0;
        printf("\n ENTER THE NUMBER OF PAGES:\n");
        scanf("%d",&n);
        printf("\n ENTER THE PAGE NUMBER :\n");
        for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
        printf("\n ENTER THE NUMBER OF FRAMES :");
        scanf("%d",&no);
        for(i=0;i<no;i++)
        frame[i]= -1;
                j=0;
                printf("\tref string\t page frames\n");
for(i=1;i<=n;i++)
                {
                        printf("%d\t\t",a[i]);
                        avail=0;
                        for(k=0;k<no;k++)
if(frame[k]==a[i])
                                avail=1;
                        if (avail==0)
                        {
                                frame[j]=a[i];
```

```
                                        j=(j+1)%no;
                                        count++;
                                        for(k=0;k<no;k++)
                                        printf("%d\t",frame[k]);
        }
                                printf("\n");
        }
                printf("Page Fault Is %d",count);
                return 0;
}
```

## OUTPUT:

```
ENTER THE NUMBER OF PAGES:  20
ENTER THE PAGE NUMBER :       7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
ENTER THE NUMBER OF FRAMES :3
    ref string       page frames
7           7     -1    -1
0           7     0     -1
1           7     0     1
2           2     0     1
0
3           2     3     1
0           2     3     0
4           4     3     0
2           4     2     0
3           4     2     3
0           0     2     3
3
2
1           0     1     3
2           0     1     2
0
1
7           7     1     2
0           7     0     2
1           7     0     1
Page Fault Is 15
```




Conclusion: Thus we have successfully implemented the FIFO page replacement algorithm.

## Experiment 7

Aim: To implement Memory allocation policies:

Best and first fit.

Theory:

First Fit

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage

Fastest algorithm because it searches as little as possible.

Disadvantage

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

Best Fit

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers

the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage

Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage

It is slower and may even tend to fill up memory with tiny useless holes.

## **Implementation**:

```
#include<stdio.h>
#include<process.h>
void main()
{
	int a[20],p[20],i,j,n,m;
	printf("Enter no of Blocks.\n");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		printf("Enter the %dst Block size:",i);
		scanf("%d",&a[i]);
	}
	printf("Enter no of Process.\n");
	scanf("%d",&m);
	for(i=0;i<m;i++)
	{
		printf("Enter the size of %dst Process:",i);
		scanf("%d",&p[i]);
	}
	for(i=0;i<n;i++)
	{
	for(j=0;j<m;j++)
		{
			if(p[j]<=a[i])
			{
				printf("The Process %d allocated to %d\n",j,a[i]);
				p[j]=10000;
				break;
			}
		}
}
	for(j=0;j<m;j++)
	{
	if(p[j]!=10000)
		{
	printf("The Process %d is not allocated\n",j);
		}
```

```
        }
}
```

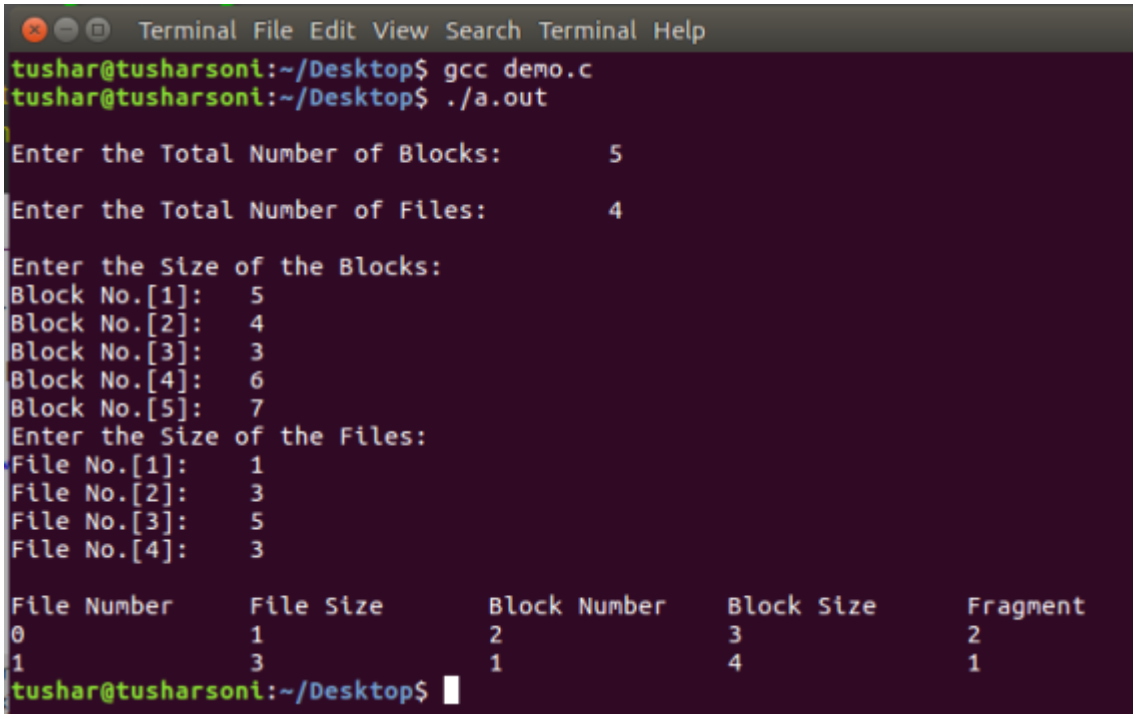
## OUTPUT:

Enter no of Blocks.
5
Enter the 0st Block size:500
Enter the 1st Block size:400
Enter the 2st Block size:300
Enter the 3st Block size:200
Enter the 4st Block size:100
Enter no of Process.
5
Enter the size of 0st Process:100
Enter the size of 1st Process:350
Enter the size of 2st Process:400
Enter the size of 3st Process:150
Enter the size of 4st Process:200
The Process 0 allocated to 500
The Process 1 allocated to 400
The Process 3 allocated to 200
The Process 2 is not allocated
The Process 4 is not allocated

```
#include<stdio.h>

int main()
{
        int fragments[10], block[10], file[10], m, n, number_of_blocks, number_of_files, temp, lowest = 10000;
        static int block_arr[10], file_arr[10];
        printf("\nEnter the Total Number of Blocks:\t");
        scanf("%d", &number_of_blocks);
        printf("\nEnter the Total Number of Files:\t");
        scanf("%d", &number_of_files);
        printf("\nEnter the Size of the Blocks:\n");
        for(m = 0; m < number_of_blocks; m++)
        {
                printf("Block No.[%d]:\t", m + 1);
                scanf("%d", &block[m]);
        }
        printf("Enter the Size of the Files:\n");
        for(m = 0; m < number_of_files; m++)
        {
                printf("File No.[%d]:\t", m + 1);
                scanf("%d", &file[m]);
        }
        for(m = 0; m < number_of_files; m++)
        {
                for(n = 0; n < number_of_blocks; n++)
                {
                        if(block_arr[n] != 1)
```

```
                {
                        temp = block[n] - file[m];
                        if(temp >= 0)
                        {
                                if(lowest > temp)
                                {
                                        file_arr[m] = n;
                                        lowest = temp;
                                }
                        }
                }
                fragments[m] = lowest;
                block_arr[file_arr[m]] = 1;
                lowest = 10000;
        }
}
printf("\nFile Number\tFile Size\tBlock Number\tBlock Size\tFragment");
for(m = 0; m < number_of_files && file_arr[m] != 0; m++)
{
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", m, file[m], file_arr[m], block[file_arr[m]], fragments[m]);
}
printf("\n");
return 0;
```




## Experiment 8

Aim: To study and implement Hamming code error detection and correction.

Theory:

In telecommunication, Hamming codes are a family of linear error-correcting codes that generalize the Hamming (7, 4)-code, and were invented by Richard Hamming in 1950. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors. By contrast, the simple parity code cannot correct errors, and can detect only an odd number of bits in error. Hamming codes are perfect codes, that is, they achieve the highest possible rate for codes with their block length and minimum distance of three.

In mathematical terms, Hamming codes are a class of binary linear codes. For each integer $r \geq 2$ there is a code with block length $n = 2r - 1$ and message length $k = 2r - r - 1$. Hence the rate of Hamming codes is $R = k / n = 1 - r / (2r - 1)$, which is the highest possible for codes with minimum distance of three (i.e., the minimal number of bit changes needed to go from any code word to any other code word is three) and block length $2r - 1$. The parity-check matrix of a Hamming code is constructed by listing all columns of length r that are non-zero, which means that the dual code of the Hamming code is the shortened Hadamard code. The parity-check matrix has the property that any two columns are pairwise linearly independent.

Due to the limited redundancy that Hamming codes add to the data, they can only detect and correct errors when the error rate is low. This is the case in computer memory (ECC memory), where bit errors are extremely rare and Hamming codes are widely used. In this context, an extended Hamming code having one extra parity bit is often used. Extended Hamming codes achieve a Hamming distance of four, which allows the decoder to distinguish between when at most one one-bit error occurs and when any two-bit errors occur. In this sense, extended Hamming codes are single-error correcting and double-error detecting, abbreviated as SECDED.

```
import java.util.*;


class Hamming {
        public static void main(String args[]) {
                Scanner scan = new Scanner(System.in);
                System.out.println("Enter the number of bits for the Hamming data:")
;
                int n = scan.nextInt();
                int a[] = new int[n];


                for(int i=0 ; i < n ; i++) {
```

```
                        System.out.println("Enter bit no. " + (n-i) + ":");
                        a[n-i-1] = scan.nextInt();
                }

                System.out.println("You entered:");
                for(int i=0 ; i < n ; i++) {
                        System.out.print(a[n-i-1]);
                }
                System.out.println();

                int b[] = generateCode(a);

                System.out.println("Generated code is:");
                for(int i=0 ; i < b.length ; i++) {
                        System.out.print(b[b.length-i-1]);
                }
                System.out.println();

                // Difference in the sizes of original and new array will give us th
e number of parity bits added.
                System.out.println("Enter position of a bit to alter to check for er
ror detection at the receiver end (0 for no error):");
                int error = scan.nextInt();
                if(error != 0) {
                        b[error-1] = (b[error-1]+1)%2;
                }
                System.out.println("Sent code is:");
                for(int i=0 ; i < b.length ; i++) {
                        System.out.print(b[b.length-i-1]);
                }
                System.out.println();
                receive(b, b.length - a.length);
        }
```

```
        static int[] generateCode(int a[]) {
                // We will return the array 'b'.
                int b[];

                // We find the number of parity bits required:
                int i=0, parity_count=0 ,j=0, k=0;
                while(i < a.length) {
                        // 2^(parity bits) must equal the current position
                        // Current position is (number of bits traversed + number of
parity bits + 1).
                        // +1 is needed since array indices start from 0 whereas we
need to start from 1.

                        if(Math.pow(2,parity_count) == i+parity_count + 1) {
                                parity_count++;
                        }
                        else {
                                i++;
                        }
                }

                // Length of 'b' is length of original data (a) + number of parity b
its.
                b = new int[a.length + parity_count];

                // Initialize this array with '2' to indicate an 'unset' value in pa
rity bit locations:

                for(i=1 ; i <= b.length ; i++) {
                        if(Math.pow(2, j) == i) {
                        // Found a parity bit location.
                        // Adjusting with (-1) to account for array indices starting
from 0 instead of 1.

                                b[i-1] = 2;
```

```
                                j++;
                        }
                        else {
                                b[k+j] = a[k++];
                        }
                }
                for(i=0 ; i < parity_count ; i++) {
                        // Setting even parity bits at parity bit locations:

                        b[((int) Math.pow(2, i))-1] = getParity(b, i);
                }
                return b;
        }

        static int getParity(int b[], int power) {
                int parity = 0;
                for(int i=0 ; i < b.length ; i++) {
                        if(b[i] != 2) {
                                // If 'i' doesn't contain an unset value,
                                // We will save that index value in k, increase it
by 1,
                                // Then we convert it into binary:

                                int k = i+1;
                                String s = Integer.toBinaryString(k);

                                //Nw if the bit at the 2^(power) location of the bi
nary value of index is 1
                                //Then we need to check the value stored at that lo
cation.
                                //Checking if that value is 1 or 0, we will calcula
te the parity value.

                                int x = ((Integer.parseInt(s))/((int) Math.pow(10,
power)))%10;
```

```
                                if(x == 1) {
                                        if(b[i] == 1) {
                                                parity = (parity+1)%2;
                                        }
                                }
                        }
                }
                return parity;
        }

        static void receive(int a[], int parity_count) {
                // This is the receiver code. It receives a Hamming code in array 'a
'.
                // We also require the number of parity bits added to the original d
ata.
                // Now it must detect the error and correct it, if any.

                int power;
                // We shall use the value stored in 'power' to find the correct bits
to check for parity.

                int parity[] = new int[parity_count];
                // 'parity' array will store the values of the parity checks.

                String syndrome = new String();
                // 'syndrome' string will be used to store the integer value of erro
r location.

                for(power=0 ; power < parity_count ; power++) {
                // We need to check the parities, the same no of times as the no of
parity bits added.

                        for(int i=0 ; i < a.length ; i++) {
                                // Extracting the bit from 2^(power):
```

```
                                int k = i+1;
                                String s = Integer.toBinaryString(k);
                                int bit = ((Integer.parseInt(s))/((int) Math.pow(10
, power)))%10;
                                if(bit == 1) {
                                        if(a[i] == 1) {
                                                parity[power] = (parity[power]+1)%2
;
                                        }
                                }
                        }
                        syndrome = parity[power] + syndrome;
                }
                // This gives us the parity check equation values.
                // Using these values, we will now check if there is a single bit er
ror and then correct it.

                int error_location = Integer.parseInt(syndrome, 2);
                if(error_location != 0) {
                        System.out.println("Error is at location " + error_location
+ ".");
                        a[error_location-1] = (a[error_location-1]+1)%2;
                        System.out.println("Corrected code is:");
                        for(int i=0 ; i < a.length ; i++) {
                                System.out.print(a[a.length-i-1]);
                        }
                        System.out.println();
                }
                else {
                        System.out.println("There is no error in the received data."
);
                }

                // Finally, we shall extract the original data from the received (an
d corrected) code:
                System.out.println("Original data sent was:");
```

```
                power = parity_count-1;
                for(int i=a.length ; i > 0 ; i--) {
                        if(Math.pow(2, power) != i) {
                                System.out.print(a[i-1]);
                        }
                        else {
                                power--;
                        }
                }
                System.out.println();
        }
}
```

## Output

```
Output:
Enter the number of bits for the Hamming data:
7
Enter bit no. 7:
1
Enter bit no. 6:
0
Enter bit no. 5:
1
Enter bit no. 4:
0
Enter bit no. 3:
1
Enter bit no. 2:
0
Enter bit no. 1:
1
You entered:
1010101
```

```
Generated code is:

10100101111

Enter position of a bit to alter to check for error detection at the receiver end (0
for no error):

5

Sent code is:

10100111111

Error is at location 5.

Corrected code is:

10100101111

Original data sent was:

1010101
```

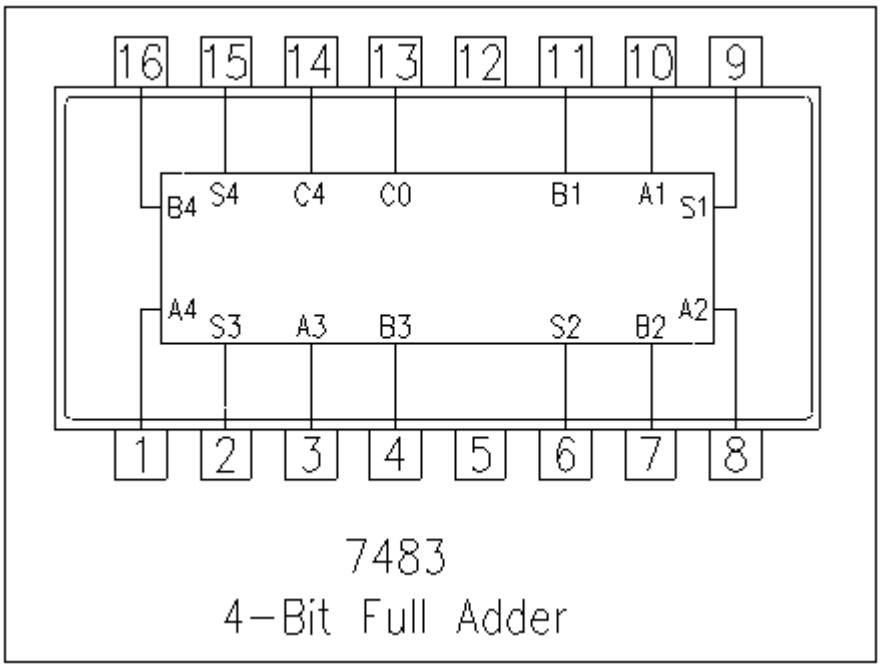
## Experiment 9

Aim:- To Study Full Adder using IC 7483.

Theory:-

This type of adder is a little more difficult to implement than a half-adder. The main difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs. The first two inputs are A and B and the third input is an input carry designated as CIN. When a full adder logic is designed we will be able to string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

**Pin Diagram of IC 7483:-**

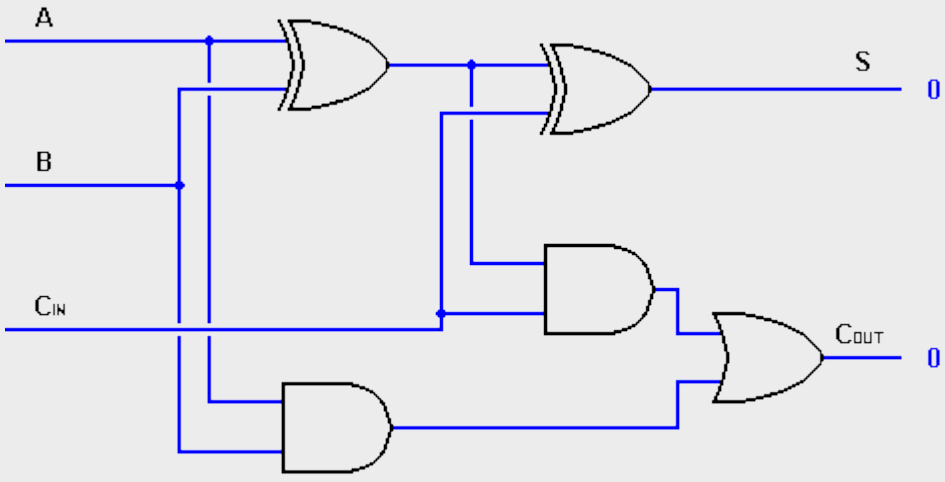



TRUTH TABLE:

| INPUTS | | OUTPUTS | | |
|---|---|---|---|---|
| **A** | **B** | **CIN** | **COUT** | **S** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

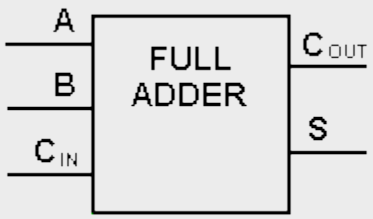
From the above truth-table, the full adder logic can be implemented. We can see that the output S is an EXOR between the input A and the half-adder SUM output with B and CIN inputs. We must also note that the COUT will only be true if any of the two inputs out of the three are HIGH.

Thus, we can implement a full adder circuit with the help of two half adder circuits. The first will half adder will be used to add A and B to produce a partial Sum. The second half adder logic can be used to add CIN to the Sum produced by the first half adder to get the final S output. If any of the half adder logic produces a carry, there will be an output carry. Thus, COUT will be an OR function of the half-adder Carry outputs. Take a look at the implementation of the full adder circuit shown below.




Full Adder Circuit

Though the implementation of larger logic diagrams is possible with the above full adder logic a simpler symbol is mostly used to represent the operation. Given below is a simpler schematic representation of a one-bit full adder.




Single-bit Full Adder

With this type of symbol, we can add two bits together taking a carry from the next lower order of magnitude, and sending a carry to the next higher order of magnitude. In a computer, for a multi-bit operation, each bit must be represented by a full adder and must be added simultaneously. Thus, to add two 8-bit numbers, you will need 8 full adders which can be formed by cascading two of the 4-bit blocks.

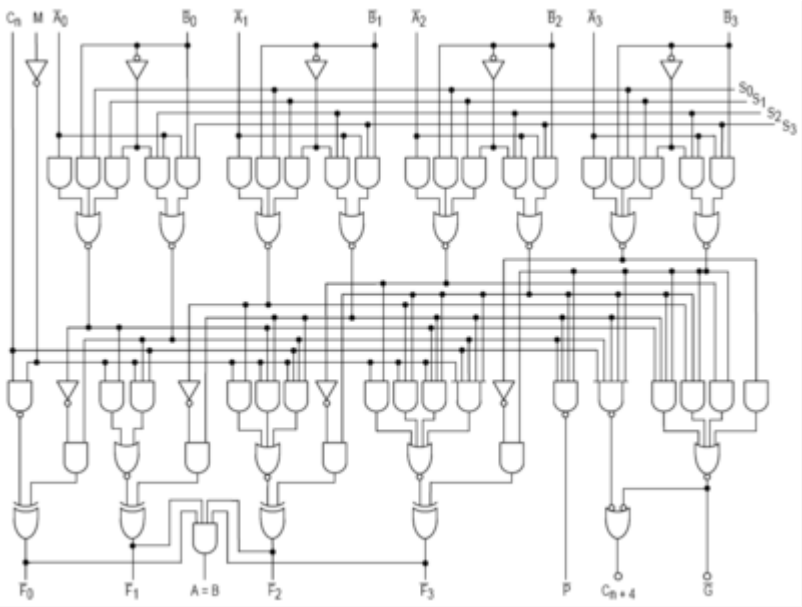
Conclusion:- Thus we have successfully studied full adder using IC 7483.

## Experiment 10

Aim: - To Study ALU using IC 74181..

Theory:- The **74181** is a bit slice arithmetic logic unit (ALU), implemented as a 7400 series TTL integrated circuit. The first complete ALU on a single chip,[1] it was used as the arithmetic/logic core in the CPUs of many historically significant minicomputers and other devices.

The 74181 represents an evolutionary step between the CPUs of the 1960s, which were constructed using discrete logic gates, and today's single-chip CPUs or microprocessors. Although no longer used in commercial products, the 74181 is still referenced in computer organization textbooks and technical papers. It is also sometimes used in 'hands-on' college courses, to train future computer architects.




The combinational logic circuitry of the 74181 integrated circuit

The 74181 is a 7400 series medium-scale integration (MSI) TTL integrated circuit, containing the equivalent of 75 logic gates[2] and most commonly packaged as a 24-pin DIP. The 4-bit wide ALU can perform all the traditional add / subtract / decrement operations with or without carry, as well as AND / NAND, OR / NOR,XOR, and shift. Many variations of these basic functions are available, for a total of 16 arithmetic and 16 logical operations on two four-bit words. Multiply and divide functions are not provided but can be performed in multiple steps using the shift and add or subtract functions. Shift is not an explicit function but can be derived from several available functions including (A+B) plus A, A plus AB[*clarification needed*].

The 74181 performs these operations on two four-bit operands generating a four-bit result with carry in 22 nanoseconds (45 MHz). The 74S181 performs the same

operations in 11 nanoseconds (90 MHz), while the 74F181 performs the operations in 7 nanoseconds (143 MHz) (typical).

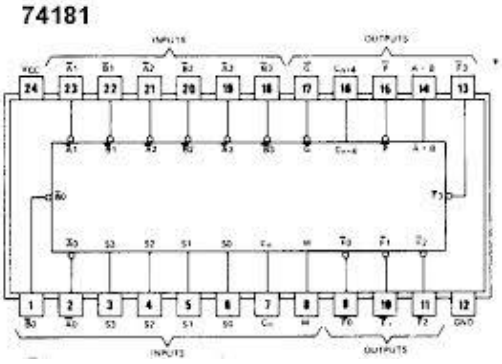
Multiple 'slices' can be combined for arbitrarily large word sizes. For example, sixteen 74S181s and five 74S182 look ahead carry generators can be combined to perform the same operations on 64-bit operands in 28 nanoseconds (36 MHz). Although overshadowed by the performance of today's multi-gigahertz 64-bit microprocessors, this was quite impressive when compared to the sub megahertz clock speeds of the early four and eight bit microprocessors.

## Significance

Although the 74181 is only an ALU and not a complete microprocessor it greatly simplified the development and manufacture of computers and other devices that required high speed computation during the late 1960s through the early 1980s, and is still referenced as a "classic" ALU design.[3]

Prior to the introduction of the 74181, computer CPUs occupied multiple circuit boards and even very simple computers could fill multiple cabinets. The 74181 allowed an entire CPU and in some cases, an entire computer to be constructed on a single large printed circuit board. The 74181 occupies a historically significant stage between older CPUs based on discrete logic functions spread over multiple circuit boards and modern microprocessors that incorporate all CPU functions in a single component. The 74181 was used in various minicomputers and other devices beginning in the 1970s, but as microprocessors became more powerful the practice of building a CPU from discrete components fell out of favor and the 74181 was not used in any new designs.

**Pin Diagram of IC 74181:-**




Conclusion: - Thus we have successfully studied ALU using IC 74181.