COMPUTER ORGANIZATION AND ARCHITECURE PROJECT-2013

IMPLEMENTATION OF

1. BOOTH'S MULTIPLICATION ALGORITHM
2. DIVISION ALGORITHM

USING THE 'C' PROGRAMMING LANGUAGE

PROJECT SUBMITTED BY:
NAVIN THOMAS -12CO60
R.M.SUMUKHA -12CO67
SURAJ.K.RAJAN -12CO94
MUKUND.Y.R -12CO58

SUBMITTED ON:
15 NOVEMBER 2013

SUBMITTED TO:
MR. KARUN KARAN

1.BOOTH'S MULTIPLICATION ALGORITHM

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm was invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is of interest in the study of computer architecture.

The Algorithm

Booth's algorithm examines adjacent pairs of bits of the *N*-bit multiplier *Y* in signed two's complement representation, including an implicit bit below the least significant bit, $y_{-1} = 0$. For each bit y_i, for *i* running from 0 to *N*-1, the bits y_i and y_{i-1} are considered. Where these two bits are equal, the product accumulator *P* is left unchanged. Where $y_i = 0$ and $y_{i-1} = 1$, the multiplicand times 2^i is added to *P*; and where $y_{\mathrm{i}} = 1$ and $y_{\mathrm{i-1}} = 0$, the multiplicand times 2^i is subtracted from *P*. The final value of *P* is the signed product.

The representation of the multiplicand and product are not specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well. As stated here, the order of the steps is not determined. Typically, it proceeds from LSB to MSB, starting at $i = 0$; the multiplication by 2^i is then typically replaced by incremental shifting of the *P* accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest *N* bits of *P*. There are many variations and optimizations on these details.

The algorithm is often described as converting strings of 1's in the multiplier to a high-order +1 and a low-order –1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

A Typical Implementation

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**.

Source Code

```
/*
        Booth's Algorithm

         Authors: NAVIN THOMAS   -12CO60
                  R.M.SUMUKHA    -12CO67
                  SURAJ.K.RAJAN  -12CO94
                  MUKUND.Y.R     -12CO58
*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<windows.h>
#include<conio.h>

void booth(int*,int*,int*);
void  comp(int*,int,int);
void ncomp(int*,int);
char* print(int);
void shift(int*,int*,int);
int deci(int*);

void main()
{
   printf("COA Project:BOOTH's ALGORITHM\nDone By:\n1.NAVIN THOMAS:12CO60\n2.MUKUND
Y.R:12CO58\n3.R.M SUMUKHA:12CO67\n4.SURAJ KRISHNAN RAJAN:12CO94\nPress any key to
continue.");
   getch();
   int ch;
   do
   {
      int a[8]={0},b[9]={0},dec1,dec2,ab,i=0,j=0,arr[16],ans,c[8]={0};
      system("cls");
      printf("\nEnter two decimal numbers:");
      scanf("%d %d",&dec1,&dec2);
      if(dec1<0)
         comp(a,-dec1,8);

      else
         ncomp(a,dec1);

      if(dec2<0)
         comp(b,-dec2,8);
      else
         ncomp(b,dec2);

      booth(a,b,c);
      for(i=0;i<8;i++)
         arr[i]=c[i];
      for(j=0;j<8;j++)
         arr[i+j]=b[j];
      if(((dec1>0)&&(dec2>0))||((dec1<0)&&(dec2<0)))
         ans=deci(arr);
      else
      {
         comp(arr,-999,16);
         ans=deci(arr);
         ans=-ans;
      }
      printf("\n\nThe answer of %d*%d=%d",dec1,dec2,ans);
      printf("\n\nPress 1 to continue:");
      ch=getch();
   }while(ch==1+'0');
}

void ncomp(int ar[],int a)
{
   int p=a;
   int i=7;
      while(p!=0)
      {
         ar[i--]=p%2;
         p=p/2;
      }
}

void comp(int ar[],int a,int max)
{
   int i=7,j=1;
   int p=a;
   if(a!=-999)
   {
      while(p!=0)
      {
         ar[i--]=p%2;
         p=p/2;
      }
   }
   for(i=0;i<max;i++)
   {
      if(ar[i]==0)
         ar[i]=1;
        else
            ar[i]=0;
    }
    i--;
    while(((ar[i]+1)==2)&&(i!=0))
        ar[i--]=0;
    ar[i]=1;
}

void booth(int a[],int b[],int c[])
{
    int i=1,j,k;
    printf("\nStep\tAction\t\t\tAccumulator\t\tRegister Q\n");
    for(k=0;k<=40;k++)
        printf("--");

    printf("\n");
    printf("0\t%s\t",print(0));
    for(j=0;j<8;j++)
        printf("%d",c[j]);

    printf("\t\t");

    for(j=0;j<8;j++)
        printf("%d",b[j]);

    printf("\n\n");

    printf("\t%s\t\t",print(1));

    for(j=0;j<8;j++)
        printf("%d",c[j]);

    printf("\t\t");

    for(j=0;j<9;j++)
        printf("%d",b[j]);

    printf("\n");
    int u;
    for(u=0;u<=40;u++)
        printf("--");
    printf("\n");
    while(i<=8)
    {
        int k,l,n=0;
        printf("%d",i);
        if((b[7]==1)&&(b[8]==0))
        {
```

```
printf("\t\t\t\t");
for(k=0;k<8;k++)
   printf("%d",a[k]);
   printf("\n");
printf("\t%s\t",print(2));
int e[8],h;
for(h=0;h<8;h++)
   e[h]=a[h];
comp(e,-999,8);
for(k=7;k>=0;k--)
{
   if((n+c[k]+e[k])==0)
   {
      n=0;
      c[k]=0;
   }
   else if((n+c[k]+e[k])==1)
   {
      n=0;
      c[k]=1;
   }
   else if((n+c[k]+e[k])==2)
   {
      n=1;
      c[k]=0;
   }
   else
   {
      n=1;
      c[k]=1;
   }
}
for(k=0;k<8;k++)
   printf("%d",c[k]);
printf("\t\t");
for(k=0;k<9;k++)
   printf("%d",b[k]);
   printf("\n");

if(i==8)
   break;
shift(c,b,c[0]);
printf("\n\t%s\t\t",print(3));
for(k=0;k<8;k++)
   printf("%d",c[k]);
printf("\t\t");
for(k=0;k<9;k++)
   printf("%d",b[k]);
printf("\n");
```

```
}
else if((b[7]==0)&&(b[8]==1))
{
    n=0;
    printf("\t\t\t\t");
    for(k=0;k<8;k++)
        printf("%d",a[k]);
        printf("\n");
    for(k=7;k>=0;k--)
    {
        if((n+c[k]+a[k])==0)
        {
            n=0;
            c[k]=0;
        }
        else if((n+c[k]+a[k])==1)
        {
            n=0;
            c[k]=1;
        }
        else if((n+c[k]+a[k])==2)
        {
            n=1;
            c[k]=0;
        }
        else
        {
            n=1;
            c[k]=1;
        }
    }
    printf("\t%s\t\t",print(4));
    for(k=0;k<8;k++)
        printf("%d",c[k]);
    printf("\t\t");
    for(k=0;k<9;k++)
        printf("%d",b[k]);
        printf("\n\n");

        if(i==8)
            break;

    shift(c,b,c[0]);
    printf("\t%s\t\t",print(3));
    for(k=0;k<8;k++)
        printf("%d",c[k]);
    printf("\t\t");
    for(k=0;k<9;k++)
        printf("%d",b[k]);
```

```
            printf("\n");
        }
        else
        {
            printf("\t%s\t",print(5));
            for(k=0;k<8;k++)
                printf("%d",c[k]);
            printf("\t\t");
            for(k=0;k<9;k++)
                printf("%d",b[k]);
                printf("\n\n");
                if(i==8)
                    break;
            shift(c,b,c[0]);
            printf("\t%s\t\t",print(3));
            for(k=0;k<8;k++)
                printf("%d",c[k]);
            printf("\t\t");
            for(k=0;k<9;k++)
                printf("%d",b[k]);
            printf("\n");


        }
        i++;
        int z;
        for(z=0;z<=40;z++)
                printf("--");
                printf("\n");

    }
    b[7]=0;
    printf("\t%s\t\t",print(6));
    int m;
    for(m=0;m<8;m++)
        printf("%d",c[m]);
    printf("\t\t");
    for(m=0;m<9;m++)
        printf("%d",b[m]);

}
char* print(int a)
{
    if(a==0)
        return("Initialize Registers");

    else if(a==1)
        return("Set Q[-1] to 0");
    else if(a==2)
        return("Subtract M from A");

    else if(a==3)
        return("Right shift A.Q");

    else if(a==4)
        return("Add M to A");

    else if(a==5)
        return("Skip Add/Subtract");

    else if(a==6)
        return("Set Q[0] to 0");
}

void shift(int a[],int b[],int d)
{
    int i,j;
    for(i=7;i>=0;i--)
        b[i+1]=b[i];
    b[0]=a[7];
    for(j=6;j>=0;j--)
        a[j+1]=a[j];
    a[0]=d;
}



int deci(int a[])
{
    int abc=0,i,j;
    for(i=0,j=14;i<16,j>0;i++,j--)
        abc+=pow(2,i)*a[j];

    return abc;
}
```

Example

Find 3 × (−4), with **m** = 3 and **r** = −4, and x = 4 and y = 4:

- m = 0011, -m = 1101, r = 1100
- A = 0011 0000 0
- S = 1101 0000 0
- P = 0000 1100 0

- Perform the loop four times :
  1. P = 0000 110**0 0**. The last two bits are 00.
     - P = 0000 0110 0. Arithmetic right shift.
  2. P = 0000 011**0 0**. The last two bits are 00.
     - P = 0000 0011 0. Arithmetic right shift.
  3. P = 0000 001**1 0**. The last two bits are 10.
     - P = 1101 0011 0. P = P + S.
     - P = 1110 1001 1. Arithmetic right shift.
  4. P = 1110 100**1 1**. The last two bits are 11.
     - P = 1111 0100 1. Arithmetic right shift.

- The product is 1111 0100, which is −12.

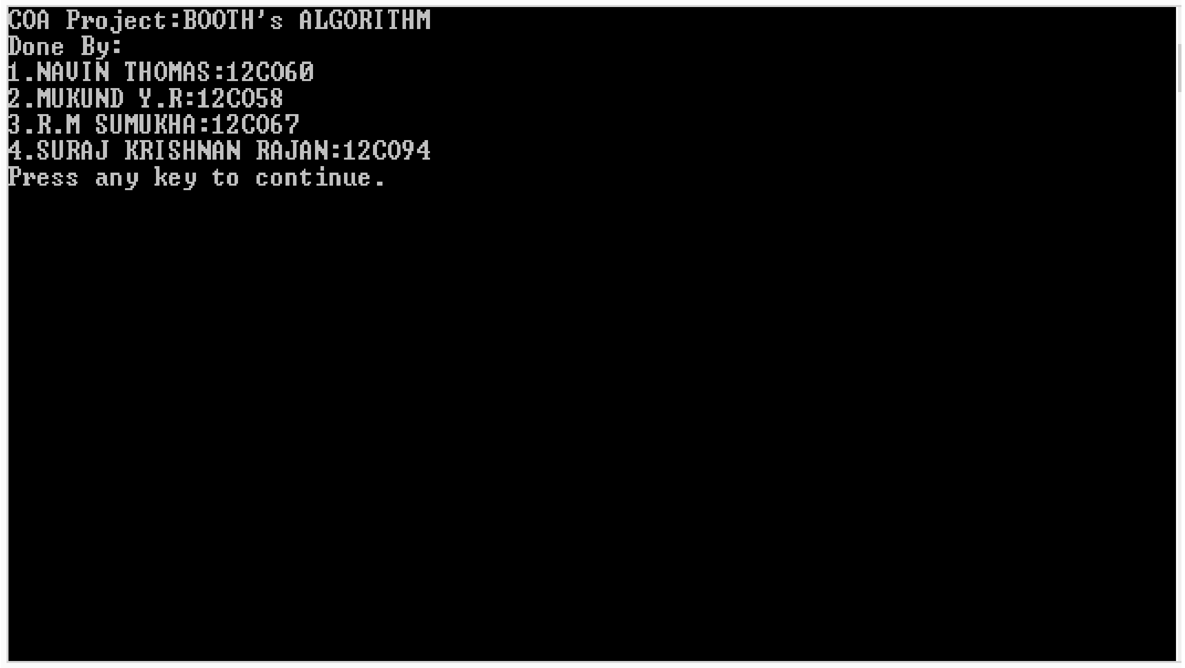
The above mentioned technique is inadequate when the multiplicand is the largest negative number that can be represented (e.g. if the multiplicand has 4 bits then this value is −8). One possible correction to this problem is to add one more bit to the left of A, S and P. Below, we demonstrate the improved technique by multiplying −8 by 2 using 4 bits for the multiplicand and the multiplier:

- A = 1 1000 0000 0
- S = 0 1000 0000 0
- P = 0 0000 0010 0

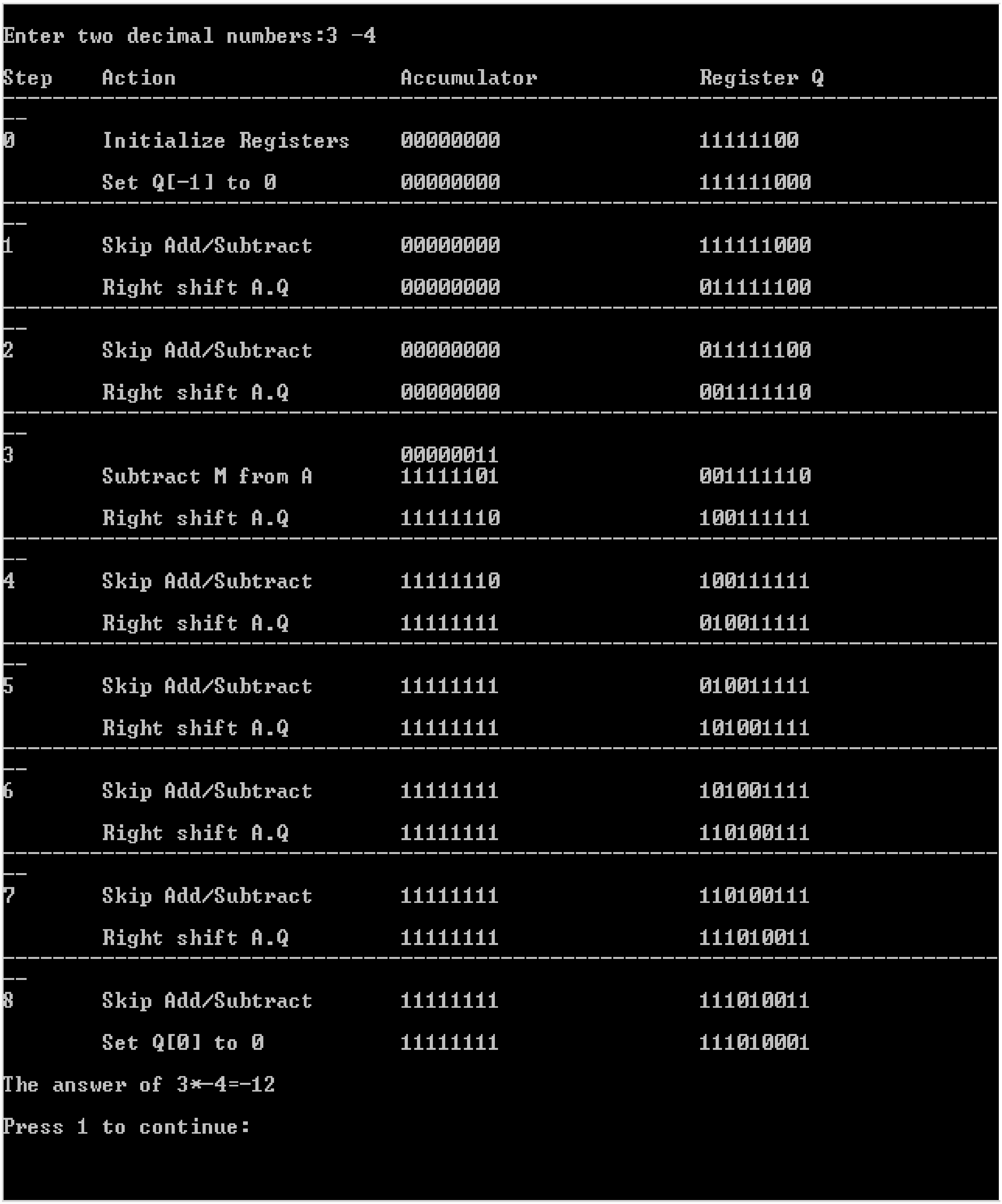
- Perform the loop four times :
  1. P = 0 0000 001**0 0**. The last two bits are 00.
     - P = 0 0000 0001 0. Right shift.
  2. P = 0 0000 000**1 0**. The last two bits are 10.
     - P = 0 1000 0001 0. P = P + S.
     - P = 0 0100 0000 1. Right shift.
  3. P = 0 0100 000**0 1**. The last two bits are 01.
     - P = 1 1100 0000 1. P = P + A.
     - P = 1 1110 0000 0. Right shift.
  4. P = 1 1110 000**0 0**. The last two bits are 00.
     - P = 1 1111 0000 0. Right shift.

- The product is 11110000 (after discarding the first and the last bit) which is −16.

Screen Shots



```
COA Project:BOOTH's ALGORITHM
Done By:
1.NAVIN THOMAS:12CO60
2.MUKUND Y.R:12CO58
3.R.M SUMUKHA:12CO67
4.SURAJ KRISHNAN RAJAN:12CO94
Press any key to continue.
```



```
Enter two decimal numbers:3 -4
Step    Action                  Accumulator             Register Q
--------------------------------------------------------------------------------
--
0       Initialize Registers    00000000                11111100
        Set Q[-1] to 0          00000000                111111000
--------------------------------------------------------------------------------
--
1       Skip Add/Subtract       00000000                111111000
        Right shift A.Q         00000000                011111100
--------------------------------------------------------------------------------
--
2       Skip Add/Subtract       00000000                011111100
        Right shift A.Q         00000000                001111110
--------------------------------------------------------------------------------
--
3                               00000011
        Subtract M from A       11111101                001111110
        Right shift A.Q         11111110                100111111
--------------------------------------------------------------------------------
--
4       Skip Add/Subtract       11111110                100111111
        Right shift A.Q         11111111                010011111
--------------------------------------------------------------------------------
--
5       Skip Add/Subtract       11111111                010011111
        Right shift A.Q         11111111                101001111
--------------------------------------------------------------------------------
--
6       Skip Add/Subtract       11111111                101001111
        Right shift A.Q         11111111                110100111
--------------------------------------------------------------------------------
--
7       Skip Add/Subtract       11111111                110100111
        Right shift A.Q         11111111                111010011
--------------------------------------------------------------------------------
--
8       Skip Add/Subtract       11111111                111010011
        Set Q[0] to 0           11111111                111010001
The answer of 3*-4=-12
Press 1 to continue:
```

How it works

- Consider a positive multiplier consisting of a block of 1s surrounded by 0s. For example, 00111110. The product is given by :
- $M \times "0\,0\,1\,1\,1\,1\,1\,0" = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$
- where M is the multiplicand. The number of operations can be reduced to two by rewriting the same as
- $M \times "0\,1\,0\,0\,0\,0\text{-}1\,0" = M \times (2^6 - 2^1) = M \times 62.$
- In fact, it can be shown that any sequence of 1's in a binary number can be broken into the difference of two binary numbers:
- $(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots)_2 \equiv (\ldots 1\overbrace{0\ldots 0}^{n}0\ldots)_2 - (\ldots 0\overbrace{0\ldots 1}^{n}0\ldots)_2.$
- Hence, we can actually replace the multiplication by the string of ones in the original number by simpler operations, adding the multiplier, shifting the partial product thus formed by appropriate places, and then finally subtracting the multiplier. It is making use of the fact that we do not have to do anything but shift while we are dealing with 0s in a binary multiplier, and is similar to using the mathematical property that 99 = 100 − 1 while multiplying by 99.
- This scheme can be extended to any number of blocks of 1s in a multiplier (including the case of single 1 in a block). Thus,
- $M \times "0\,0\,1\,1\,1\,0\,1\,0" = M \times (2^5 + 2^4 + 2^3 + 2^1) = M \times 58$
- $M \times "0\,1\,0\,0\text{-}1\,1\text{-}1\,0" = M \times (2^6 - 2^3 + 2^2 - 2^1) = M \times 58.$
- Booth's algorithm follows this scheme by performing an addition when it encounters the first digit of a block of ones (0 1) and a subtraction when it encounters the end of the block (1 0). This works for a negative multiplier as well. When the ones in a multiplier are grouped into long blocks, Booth's algorithm performs fewer additions and subtractions than the normal multiplication algorithm.

References

World Wide Web

[1] Wikipedia-Booth's Multiplication Algorithm
http://en.wikipedia.org/wiki/Booth's_multiplication_algorithm

Books

[1] John P. Hayes, 'Computer architecture and Organization, Tata McGraw-Hill, Third edition, 1998. 2. V.

2. DIVISION ALGORITHM

A **division algorithm** is an algorithm which, given two integers N and D, computes their quotient and/or remainder, the result of division. Some are applied by hand, while others are employed by digital circuit designs and software like the following.

The Algorithm

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 fall into this category.

Discussion will refer to the form where Q = N / D

Q = Quotient

N = Numerator (dividend)

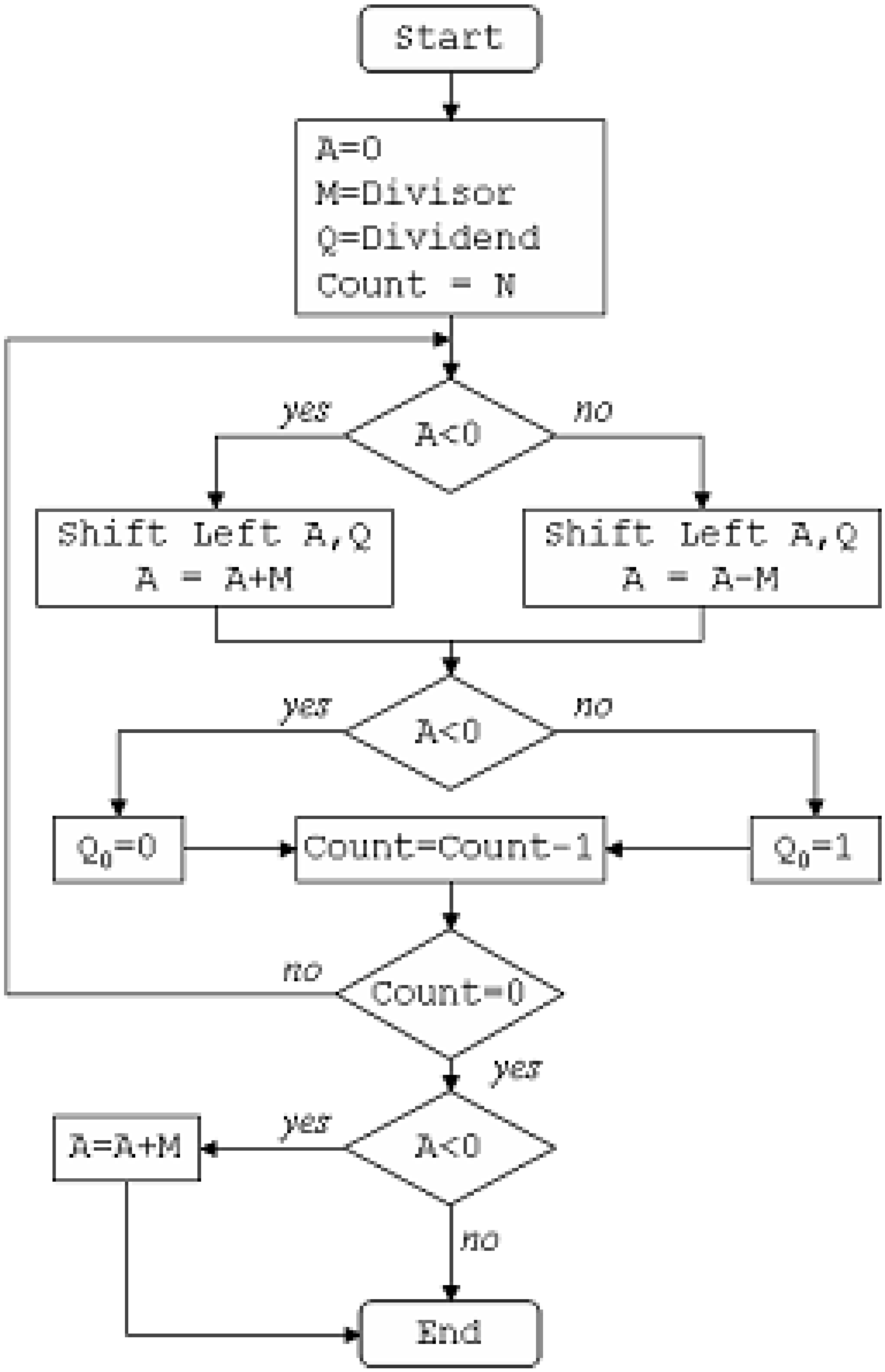
D = Denominator (divisor).

The following is based on the Non Restoring Division Algorithm.

In "Non-restoring Division Algorithm" the restoring step of the "Restoring Division" is eliminated by merging the addition and the subtraction step in the "Restoring Division" by a single step:

$R_{i+1} = 2R_i + V$

The flow chart in the following page demonstrates the algorithm that we have followed in the implementation


Start
A=0
M=Divisor
Q=Dividend
Count = N
A<0
yes
no
Shift Left A,Q
A = A+M
Shift Left A,Q
A = A-M
A<0
yes
no
Q_0=0
Count=Count-1
Q_0=1
Count=0
no
yes
A<0
yes
A=A+M
no
End

A Typical Implementation

According to our algorithm :

1. We set A to 0 intially , M to values of the divisor and Q to the value of the dividend , the count is set to the number of bits the dividend has .
2. The values in the registers A and Q are left shifted , and if the value in the register A is less than 0 we perform addition i.e A = A+M , else we perform a subtraction
   A = A-M.
3. Check the value of A and set the value of Q[0] accordingly i.e 0 if A<0 , 1 otherwise , decrement count by 1
4. Continue steps 2 and 3 till count is equal to 0, once count is equal to 0 perform the last addition step if the value in A is less than 0.
5. At the end of step 4 the value in the register A gives the value of the remainder and the value in the register Q gives the value of the quotient.

Source Code

```
/*
        Division Algorithm

  Authors:   R.M.SUMUKHA     -12CO67
             NAVIN THOMAS   -12CO60
             SURAJ.K.RAJAN   -12CO94
             MUKUND.Y.R       -12CO58
*/

#include<stdio.h>
#include<stdlib.h>
#define MAX 100

int binary_con(int quot[],int dividend)
{
  int ref=dividend,rem;
  int arr[MAX];
  int a=0,i=0,j;
  while(ref>0)
  {
    rem=ref%2;
    ref=ref/2;
    arr[i++]=rem;
  }
  for(j=i-1; j>=0; j--)
  quot[a++]=arr[j];
  return a;
}

void binary_con_1(int m_divs[],int divisor,int value)
{
  int num;
  int i,j;
  num=binary_con(m_divs,divisor);
  for(i=value-1,j=num-1; j>=0; i--,j--)
  m_divs[i]=m_divs[j];

  for(j=i; j>=0; j--)
  m_divs[j]=0;
}
```

```
void convert(char temp1[],char temp2[])
{
   int l,c;
   int i;
   l=strlen(temp1);
   for (i=l-1; i>=0; i--)
   {
      if (temp1[i]=='0')
      temp2[i]='1';

      else
      temp2[i]='0';
   }

   for(i=l-1; i>=0; i--)
   {
      if(i==l-1)
      {
         if(temp2[i]=='0')
         temp2[i]='1';

         else
         {
            temp2[i]='0';
            c=1;
         }
      }

      else
      {
         if(c==1 && temp2[i]=='0')
         {
            temp2[i]='1';
            c=0;
         }
         else if (c==1 &&temp2[i]=='1')
         {
            temp2[i]='0';
            c=1;
         }
      }
}

   temp2[l]='\0';
   /*printf("The 2's complement of divisor : %s",temp2);*/
}

void twos_com(int m_divs[],int m_divs_comp[],int value)
{
   char temp1[MAX],temp2[MAX];
```

```
    int i;
    for(i=0; i<value; i++)
    {
       if(m_divs[i]==0)
       temp1[i]='0';
       else if(m_divs[i]==1)
       temp1[i]='1';
    }
    temp1[i]='\0';
    /*printf("\n\ndivisor string :%s\n\n",temp1);*/
    convert(temp1,temp2);

    for(i=0; i<value; i++)
    {
       if(temp2[i]=='0')
       m_divs_comp[i]=0;
       else if(temp2[i]=='1')
       m_divs_comp[i]=1;
    }
}

void shift_left(int acc[],int quot[],int value)
{
    int i,j;
    for(i=0; i<value-1; i++)
    acc[i]=acc[i+1];
    acc[value-1]=quot[0];

    for(i=1; i<value; i++)
    quot[i-1]=quot[i];
}

void add(int temp1[],int temp2[],int value)
{
    int j;
    int c;
    /*char temp1[MAX],temp2[MAX];*/
    /*convert_string(arr1,temp1,value);
    convert_string(arr2,temp2,value);*/
    for(j=value-1; j>=0; j--)
    {
       if(j==value-1)
       {
          if((temp1[j]==0)&&(temp2[j]==0))
          {
             temp1[j]=0;
             c=0;
          }
          else if((temp1[j]==0)&&(temp2[j]==1))
          {
            temp1[j]=1;
            c=0;
        }
        else if((temp1[j]==1)&&(temp2[j]==0))
        {
            temp1[j]=1;
            c=0;
        }
        else if((temp1[j]==1)&&(temp2[j]==1))
        {
            temp1[j]=0;
            c=1;
        }
    }
    else
    {
        if((c==0)&&(temp1[j]==0)&&(temp2[j]==0))
        {
            temp1[j]=0;
            c=0;
        }
        else if((c==0)&&(temp1[j]==0)&&(temp2[j]==1))
        {
            temp1[j]=1;
            c=0;
        }
        else if((c==0)&&(temp1[j]==1)&&(temp2[j]==0))
        {
            temp1[j]=1;
            c=0;
        }
        else if((c==0)&&(temp1[j]==1)&&(temp2[j]==1))
        {
            temp1[j]=0;
            c=1;
        }
        else if((c==1)&&(temp1[j]==0)&&(temp2[j]==0))
        {
            temp1[j]=1;
            c=0;
        }
        else if((c==1)&&(temp1[j]==0)&&(temp2[j]==1))
        {
            temp1[j]=0;
            c=1;
        }
        else if((c==1)&&(temp1[j]==1)&&(temp2[j]==0))
        {
            temp1[j]=0;
            c=1;
        }
        else if((c==1)&&(temp1[j]==1)&&(temp2[j]==1))
        {
          temp1[j]=1;
          c=1;
        }
      }
    }
}

int main()
{
  /*char temp1[MAX],temp2[MAX];*/
  int acc[MAX],quot[MAX],m_divs[MAX],m_divs_comp[MAX];
  int dividend,divisor,value,flag1,flag2;
  int i,j;
  fflush(stdin);
  printf("enter the dividend and divisor\n");
  scanf("%d %d",&dividend,&divisor);
  value=binary_con(quot,dividend);

  /*printf("binary of dividend : ");
  for(i=0; i<value; i++)
  printf("%d",quot[i]);
  printf("\nvalue : %d",value);*/

  binary_con_1(m_divs,divisor,value);   //gives the integer array of that size

  /*printf("\n\ndivisor : ");
  for(i=0; i<value; i++)
  printf("%d",m_divs[i]);*/

  twos_com(m_divs,m_divs_comp,value);   //gives the integer array of 2's complement by the aid
of strings

  /*printf("two's complement of divisor : ");
  for(i=0; i<value; i++)
  printf("%d",m_divs_comp[i]);*/

  for(i=0; i<value; i++)
  acc[i]=0;
  printf("\nAcc=>Accumulator, M=>Memory Register\n");
  for(i=value; i>0; i--)
  {
    flag1=acc[0];
    shift_left(acc,quot,value);
    /*printf("acc : ");
    for(j=0; j<value; j++)
    printf("%d",acc[j]);
    printf("\nquotient : ");
    for(j=0; j<value; j++)
    printf("%d",quot[j]);
    exit(1);*/

    if(flag1==0)
    add(acc,m_divs_comp,value);
    else if(flag1==1)
    add(acc,m_divs,value);

    flag2=acc[0];
    if(flag2==0)
    quot[value-1]=1;
    else if(flag2==1)
    quot[value-1]=0;

    /*printf("acc :");
    for(i=0; i<value; i++)
    printf("%d",acc[i]);
    printf("\nquot :");
    for(i=0; i<value; i++)
    printf("%d",quot[i]);
    exit(1);*/

    if(flag1==0)
    {
      printf("\n\nstep : %d  here Acc>0\noperations performed are left shift and A=A-
M\n",value-i+1);
      printf("accumulator : ");
      for(j=0; j<value; j++)
      printf("%d",acc[j]);
      printf("\nquotient register : ");
      for(j=0; j<value; j++)
      printf("%d",quot[j]);
    }
    else if((flag1==1))
    {
      printf("\n\nstep : %d  here Acc<0\noperations performed are left shift and
A=A+M\n",value-i+1);
      printf("accumulator : ");
      for(j=0; j<value; j++)
      printf("%d",acc[j]);
      printf("\nquotient register : ");
      for(j=0; j<value; j++)
      printf("%d",quot[j]);
    }
  }
  if(acc[0]==1)
  {
    printf("\n\nHERE ACC < 0 , HENCE LAST STEP IS A=A+M\n");
    add(acc,m_divs,value);
  }

  printf("\n\nfinal answer : \n");
  printf("ACC - > remainder : ");
  for(i=0; i<value; i++)
  printf("%d",acc[i]);
  printf(" which is the decimal equivalent of %d",dividend%divisor);
  printf("\n\nQUOTIENT - > :");
  for(i=0; i<value; i++)
  printf("%d",quot[i]);
  printf(" which is the decimal equivalent of %d\n\n",dividend/divisor);

  return 0;
}
```

Example

Consider the example to demonstrate the working of the algorithm:

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

Initialization :

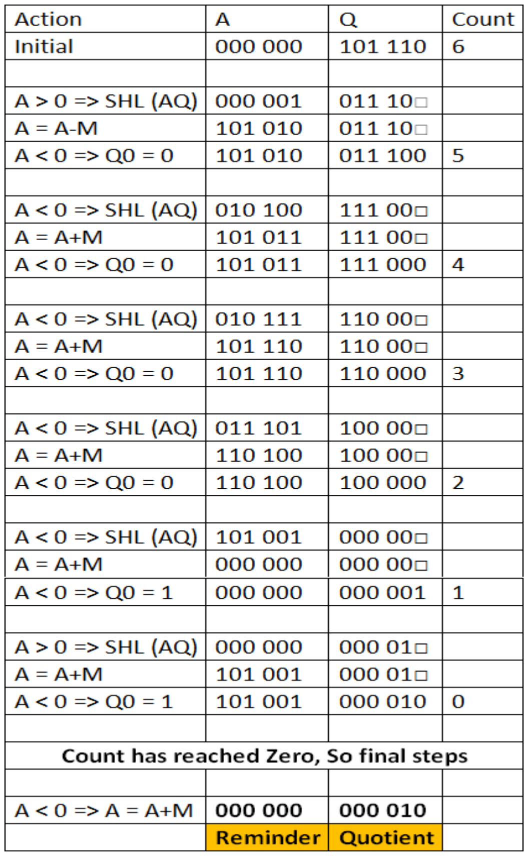
Set Register A = Dividend = 000000.

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.

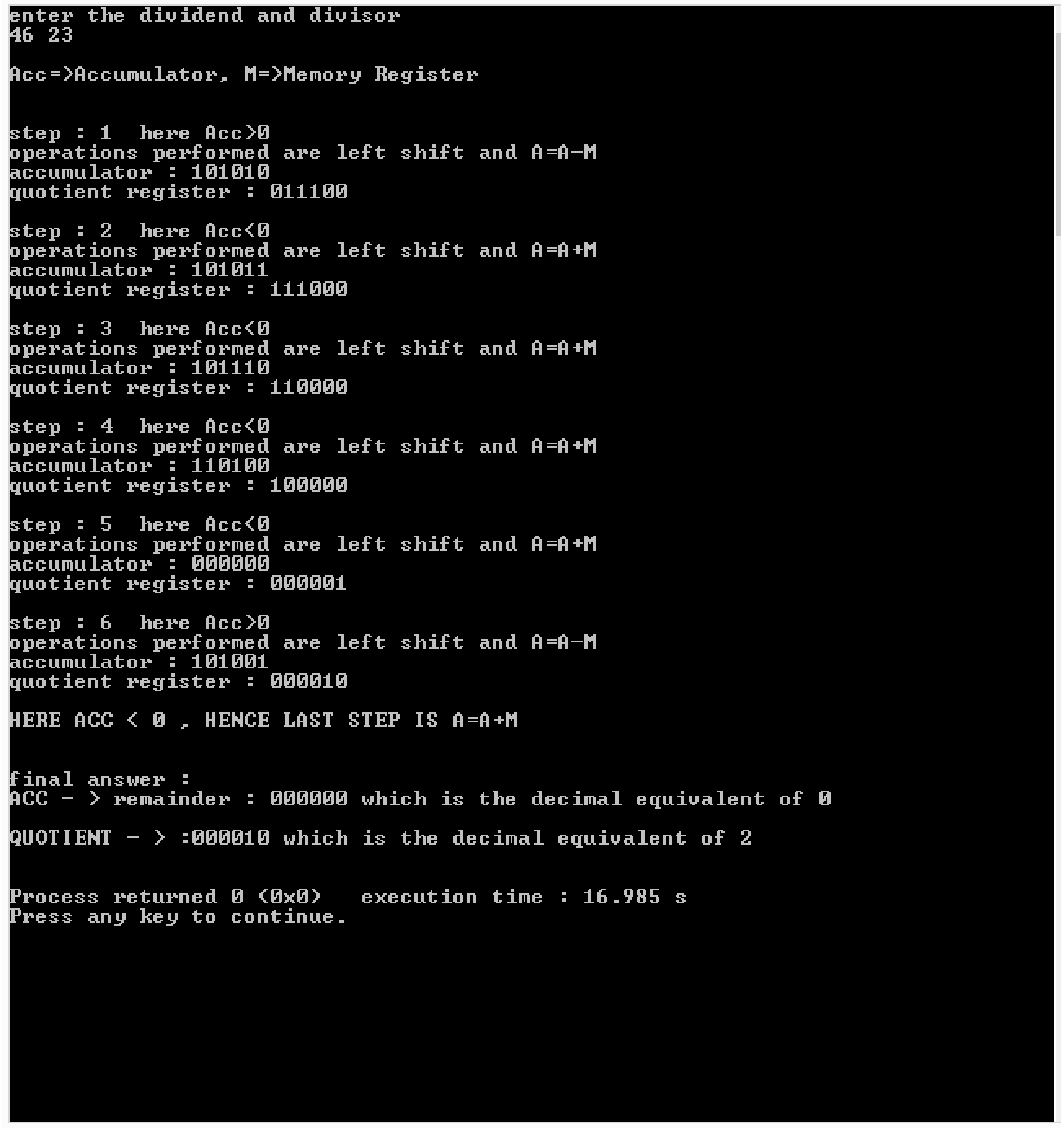
Set Count = 6, since 6 digits operation is being done here .

After this we start the algorithm, which I have shown in a table below :
In 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**	

Screen Shot



```
enter the dividend and divisor
46 23

Acc=>Accumulator, M=>Memory Register


step : 1  here Acc>0
operations performed are left shift and A=A-M
accumulator : 101010
quotient register : 011100

step : 2  here Acc<0
operations performed are left shift and A=A+M
accumulator : 101011
quotient register : 111000

step : 3  here Acc<0
operations performed are left shift and A=A+M
accumulator : 101110
quotient register : 110000

step : 4  here Acc<0
operations performed are left shift and A=A+M
accumulator : 110100
quotient register : 100000

step : 5  here Acc<0
operations performed are left shift and A=A+M
accumulator : 000000
quotient register : 000001

step : 6  here Acc>0
operations performed are left shift and A=A-M
accumulator : 101001
quotient register : 000010

HERE ACC < 0 , HENCE LAST STEP IS A=A+M


final answer :
ACC - > remainder : 000000 which is the decimal equivalent of 0

QUOTIENT - > :000010 which is the decimal equivalent of 2


Process returned 0 (0x0)   execution time : 16.985 s
Press any key to continue.
```

References

World Wide Web

[1] Wikipedia- Division Algorithm
http://en.wikipedia.org/wiki/Division_algorithm

Books

[1] John P. Hayes, 'Computer architecture and Organization, Tata McGraw-Hill, Third edition, 1998. 2. V.