PROCESSOR ARCHITECTURE

PRACTICAL 2

Division/Batch: B/B1
Branch: Computer Engineering

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004200107	Kartik Jolapara	27/9/22	3/10/22	

Aim

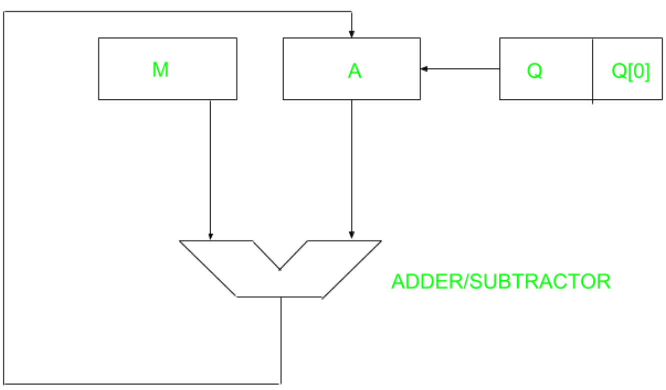
To study and implement restoring and non-restoring division algorithm.

Restoring Division

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.

Restoring Division Algorithm is used to divide two unsigned integers. This algorithm is used in Computer Organization and Architecture. This algorithm is called restoring because it restores the value of Accumulator(A) after each or some iterations. There is one more type i.e., Non-Restoring Division Algorithm in which value of A is not restored.



First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend). 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.

Algorithm

Restoring division operates on fixed-point fractional numbers and depends on the assumption $0 < D < N$. The basic algorithm for binary (radix 2) restoring division is:

```
R := N
D := D << n             -- R and D need twice the word width of N and Q
for i := n − 1 .. 0 do  -- For example 31..0 for 32 bits
  R := 2 * R − D          -- Trial subtraction from shifted value
  if R ≥ 0 then
    q(i) := 1           -- Result-bit 1
  else
    q(i) := 0           -- Result-bit 0
    R := R + D          -- New partial remainder is (restored) shifted value
  end
end
```

In simpler terms, let the dividend be Q and the divisor be M and the accumulator A = 0. Therefore:

1. At each step, left shift the dividend by 1 position.
2. Subtract the divisor from A (A – M).
3. If the result is positive, then the step is said to be successful. In this case, the quotient bit will be "1" and the restoration is not required.
4. If the result is negative, then the step is said to be unsuccessful. In this case, the quotient bit will be "0" and restoration is required.
5. Repeat the above steps for all the bits of the dividend.

Example

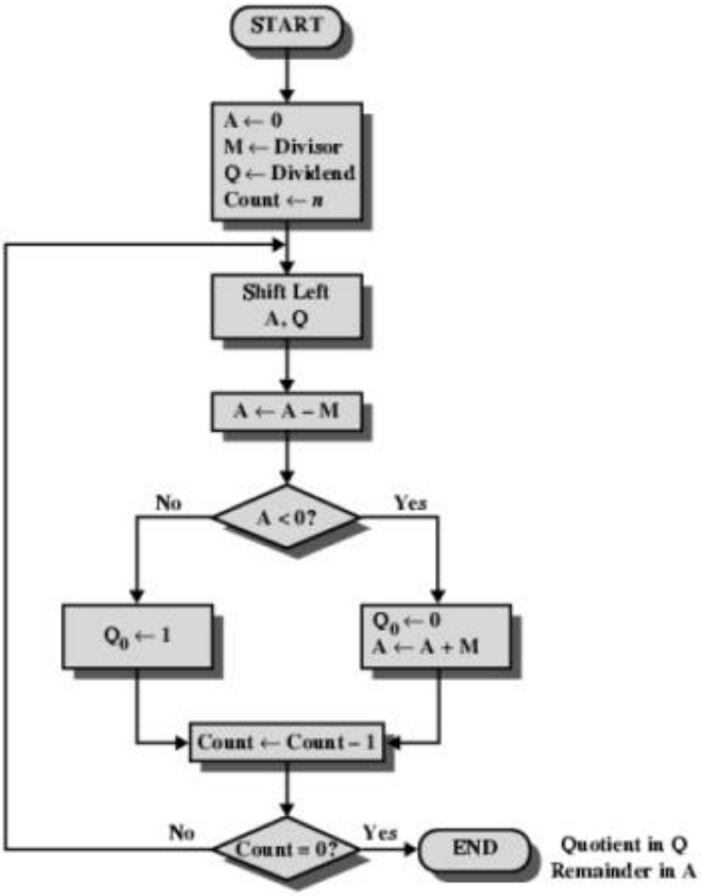
Problem: 13/4 M=0100 Q=1101 -M=1100 N=4

N	M	A	Q	Operation
4	0100	0000	1101	Initialization
	0100	0001	101-	LS, N=N-1
	0100	1101	101-	A=A-B
	0100	0001	1010	A=A+B
3	0100	0001	1010	
	0100	0011	010-	LS, N=N-1
	0100	1111	010-	A=A-B
	0100	0011	0100	A=A+B
2	0100	0011	0100	
	0100	0110	100-	LS, N=N-1
	0100	0010	100-	A=A-B
	0100	0010	1001	
1	0100	0010	1001	
	0100	0101	001-	LS, N=N-1

	0100	0001	001-	A=A-B
	0100	0001	0011	Termination

Result: Quotient: 0011 Remainder: 0001

Flowchart



Code

```
// Non-restoring division

#include <bits/stdc++.h>

using namespace std;

vector<bool> oneSComplement(vector<bool> num)
{
    for (int i = 0; i < num.size(); i++)
    {
        num[i] = !num[i];
    }
    return num;
```

```
}

vector<bool> twoSComplement(vector<bool> num)
{
    num = oneSComplement(num);
    if (num[num.size() - 1])
    {
        num[num.size() - 1] = 0;
        num[num.size() - 2] = 1;
    }
    else
    {
        num[num.size() - 1] = 1;
    }
    return num;
}

vector<bool> binaryAddition(vector<bool> a, vector<bool> b, int n)
{
    vector<bool> ans(n);
    bool carry = 0;
    for (int i = n - 1; i >= 0; i--)
    {
        if (a[i] == 1 && b[i] == 1 && carry)
        {
            ans[i] = 1;
            carry = 1;
        }
        else if ((a[i] == 1 && b[i] == 1) || (((a[i] == 1) || (b[i] ==
1)) && carry))
        {
            ans[i] = 0;
            carry = 1;
        }
        else
        {
            ans[i] = a[i] + b[i] + carry;
            carry = 0;
        }
    }
    return ans;
}

// Returns the right most shifted value from the q
bool arithmeticRightShift(vector<bool> &a, vector<bool> &q)
{
    bool kachra = q[q.size() - 1], prev = a[0], temp;
```

```
    for (int i = 1; i < a.size(); i++)
    {
        temp = a[i];
        a[i] = prev;
        prev = temp;
    }
    temp = q[0];
    q[0] = prev;
    prev = temp;
    for (int i = 1; i < q.size(); i++)
    {
        temp = q[i];
        q[i] = prev;
        prev = temp;
    }
    return kachra;
}

// Left Shifts the 2 numbers :)
void arithmeticLeftShift(vector<bool> &a, vector<bool> &q)
{
    bool kachra = q[q.size() - 1], prev = a[0], temp;
    for (int i = 0; i < a.size() - 1; i++)
    {
        a[i] = a[i + 1];
    }
    a[a.size() - 1] = q[0];
    for (int i = 0; i < q.size() - 1; i++)
    {
        q[i] = q[i + 1];
    }
}

int main()
{
    // Taking the input in string so that we don't need to ask for the
size of the number
    string qtemp, mtemp;
    // Then storing the number in a vector(Array) for easy access
    vector<bool> q, m, a, negM;
    bool qNeg = 0;

    // We'll be taking the input in binary format only
    cout << "Enter m(divisor): ";
    cin >> mtemp;
    cout << "Enter q(dividend): ";
    cin >> qtemp;
```

```
// Counter
int n;
// Assigning the counter with the max value
// if (mtemp.length() > qtemp.length())
//     n = mtemp.length();
// else
//     n = qtemp.length();

n = qtemp.length();

int count = n;

// Assigning the Accumulator with 0's
for (int i = 0; i < n + 1; i++)
{
    a.push_back(0);
}

int mtempNum = (n + 1) - mtemp.length();
while (mtempNum > 0)
{
    m.push_back(0);
    mtempNum--;
}

// Converting the string into vector(array)
for (int i = 0; i < qtemp.length(); i++)
{
    if (qtemp[i] == '1')
        q.push_back(1);
    else
        q.push_back(0);
}
for (int i = 0; i < mtemp.length(); i++)
{
    if (mtemp[i] == '1')
        m.push_back(1);
    else
        m.push_back(0);
}

// Calculating the -M
negM = twoSComplement(m);

vector<bool> ans = binaryAddition(m, q, n);
```

```cpp
    cout << "\nA\tQ\tn\tAction\n\n";

    while (count)
    {
        // --- Init Printing format ---
        for (auto x : a)
            cout << x;
        cout << "\t";
        for (auto x : q)
            cout << x;
        cout << "\t" << count << "\t"
             << "Init"
             << "\n";
        // --- Init Printing format ---
        arithmeticLeftShift(a, q);
        // --- Shift LEFT Printing format ---
        for (auto x : a)
            cout << x;
        cout << "\t";
        int tempCount = 0;
        for (auto x : q)
        {
            tempCount++;
            cout << ((tempCount == n) ? "_" : to_string(x));
        }
        cout << "\t" << count << "\t"
             << "Shift LEFT"
             << "\n";
        // --- Shift LEFT Printing format ---

        a = binaryAddition(a, negM, n + 1);

        // --- A-M Printing format ---
        for (auto x : a)
            cout << x;
        cout << "\t";
        tempCount = 0;
        for (auto x : q)
        {
            tempCount++;
            cout << ((tempCount == n) ? "_" : to_string(x));
        }
        cout << "\t" << count << "\t"
             << "A-M"
             << "\n";
        // --- A-M Printing format ---
```

```
        // a[0] = 1 means the number is -ve
        if (a[0])
        {
            q[q.size() - 1] = 0;
            a = binaryAddition(a, m, n + 1);
            // --- Q0<-0, A-M Printing format ---
            for (auto x : a)
                cout << x;
            cout << "\t";
            // int tempCount = 0;
            for (auto x : q)
            {
                // tempCount++;
                // cout << ((tempCount == n) ? "_" : to_string(x));
                cout << x;
            }
            cout << "\t" << count << "\t"
                 << "Q0<-0, A-M"
                 << "\n";
            // --- Q0<-0, A-M Printing format ---
        }
        else
        {
            q[q.size() - 1] = 1;
            // --- Q0<-1 Printing format ---
            for (auto x : a)
                cout << x;
            cout << "\t";
            // int tempCount = 0;
            for (auto x : q)
            {
                // tempCount++;
                // cout << ((tempCount == n) ? "_" : to_string(x));
                cout << x;
            }
            cout << "\t" << count << "\t"
                 << "Q0<-1"
                 << "\n";
            // --- Q0<-1 Printing format ---
        }
        cout << "\n";
        count--;
    };

    cout << "Quotient: ";
    for (auto x : q)
        cout << x;
```

```
    cout << "\n";
    cout << "Remainder: ";
    for (auto x : a)
        cout << x;

    return 0;
}
```

Output

```
Enter m(divisor): 11
Enter q(dividend): 1011

A       Q       n       Action

00000   1011    4       Init
00001   011_    4       Shift LEFT
11110   011_    4       A-M
00001   0110    4       Q0<-0, A-M

00001   0110    3       n--
00010   110_    3       Shift LEFT
11111   110_    3       A-M
00010   1100    3       Q0<-0, A-M

00010   1100    2       n--
00101   100_    2       Shift LEFT
00010   100_    2       A-M
00010   1001    2       Q0<-1

00010   1001    1       n--
00101   001_    1       Shift LEFT
00010   001_    1       A-M
00010   0011    1       Q0<-1

Quotient: 0011
Remainder: 00010
```

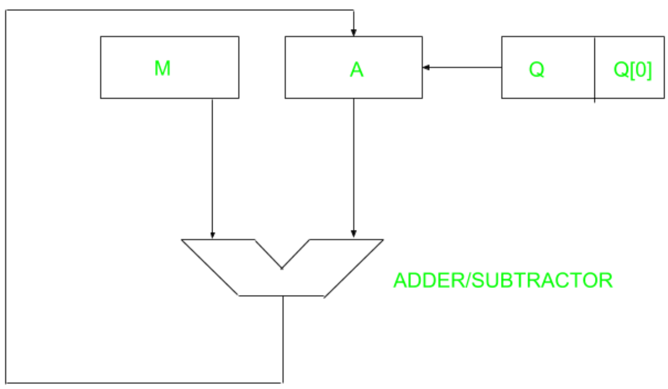
```
Enter m(divisor): 101
Enter q(dividend): 1101

A       Q       n       Action

00000   1101    4       Init
00001   101_    4       Shift LEFT
11100   101_    4       A-M
00001   1010    4       Q0<-0, A-M

00001   1010    3       n--
00011   010_    3       Shift LEFT
11110   010_    3       A-M
00011   0100    3       Q0<-0, A-M

00011   0100    2       n--
00110   100_    2       Shift LEFT
00001   100_    2       A-M
00001   1001    2       Q0<-1

00001   1001    1       n--
00011   001_    1       Shift LEFT
11110   001_    1       A-M
00011   0010    1       Q0<-0, A-M

Quotient: 0010
Remainder: 00011
```

Non-Restoring Division

Theory

Non-Restoring Division Algorithm is used to divide two unsigned integers. This algorithm is used in Computer Organization and Architecture The algorithm is more complex but has the advantage when implemented in hardware that there is only one decision and addition/subtraction per quotient bit; there is no restoring step after the subtraction, which potentially cuts down the numbers of operations by up to half and lets it be executed faster.

First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend). 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. Non-restoring division uses the digit set {−1, 1} for the quotient digits instead of {0, 1}.



Algorithm

The basic algorithm for binary (radix 2) non-restoring division of non-negative numbers is:

```
R := N
D := D << n                -- R and D need twice the word width of N
and Q
for i = n − 1 .. 0 do   -- for example 31..0 for 32 bits
  if R >= 0 then
    q[i] := +1
    R := 2 * R − D
  else
    q[i] := −1
    R := 2 * R + D
  end if
end
```

In simpler terms, let the dividend be Q and the divisor be M and the accumulator A = 0. Therefore:

1. First the registers are initialized with corresponding values
2. Check the sign bit of register A
3. If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M
4. If sign bit of register A is 1 Q[0] become 0 otherwise Q[0] become 1
5. Decrements value of N by 1, If N is not equal to zero go to Step 2
6. If sign bit of A is 1 then perform A = A+M

Example

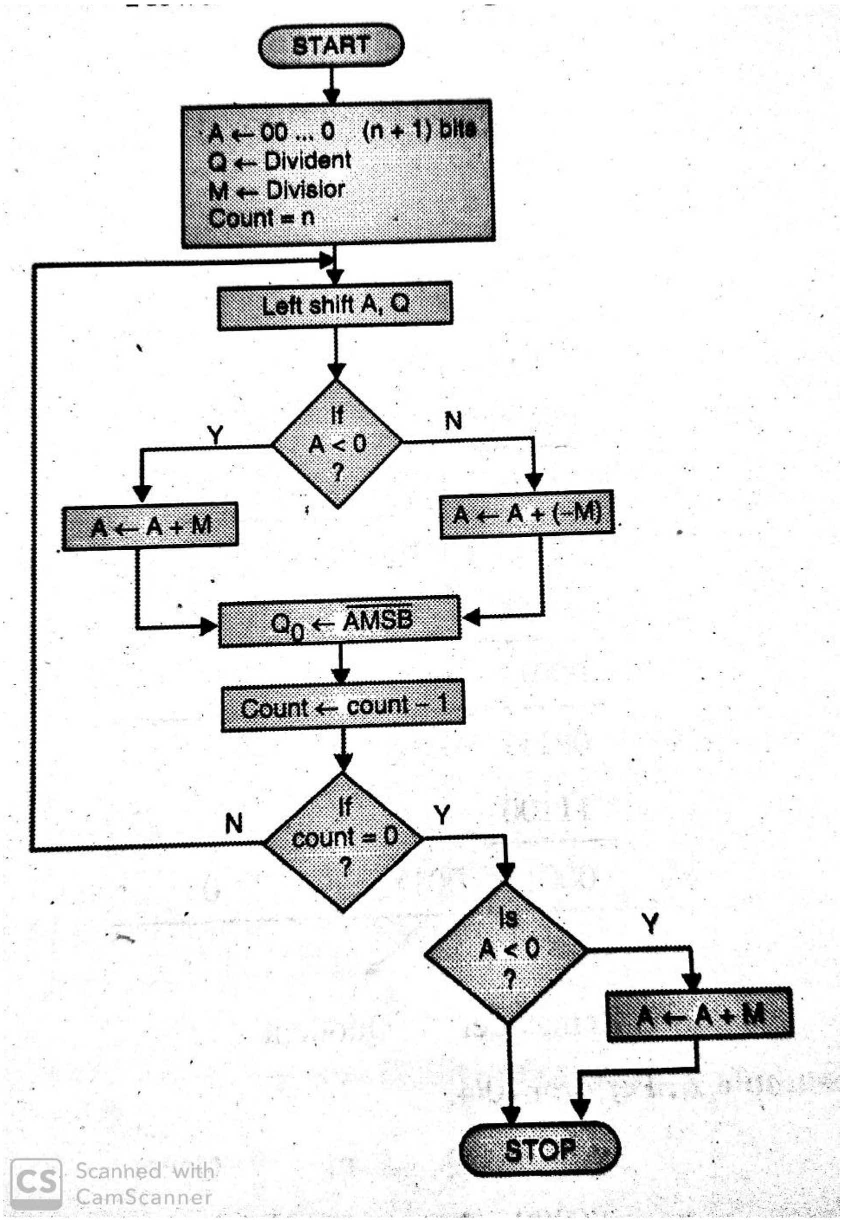
Problem: 13/4 M=0100 Q=1101 -M=1100 N=4

N	M	A	Q	Operation
4	0100	0000	1101	Initialization
	0100	0001	101-	LS, N=N-1
	0100	1101	101-	A=A-M
	0100	1101	1010	
3	0100	1101	1010	

	0100	1011	010-	LS, N=N-1
	0100	1111	010-	A=A+B
	0100	1111	0100	
2	0100	1111	0100	
	0100	1110	100-	LS, N=N-1
	0100	0010	100-	A=A+B
	0100	0010	1001	
1	0100	0010	1001	
	0100	0101	001-	LS, N=N-1
	0100	0001	001-	A=A-B
	0100	0001	0011	Termination

Result: **Quotient: 0011** **Remainder: 0001**

Flowchart



Code

```
// Non-restoring division
/*
Write up
1. Aim
FLowchart
2. Theory
3. Code (text)
4. Output
5. Conclusion
*/

#include <bits/stdc++.h>

using namespace std;

vector<bool> oneSComplement(vector<bool> num)
{
    for (int i = 0; i < num.size(); i++)
    {
        num[i] = !num[i];
    }
    return num;
}

vector<bool> twoSComplement(vector<bool> num)
{
    num = oneSComplement(num);
    if (num[num.size() - 1])
    {
        num[num.size() - 1] = 0;
        num[num.size() - 2] = 1;
    }
    else
    {
        num[num.size() - 1] = 1;
    }
    return num;
}

vector<bool> binaryAddition(vector<bool> a, vector<bool> b, int n)
{
    vector<bool> ans(n);
    bool carry = 0;
    for (int i = n - 1; i >= 0; i--)
    {
```

```
        if (a[i] == 1 && b[i] == 1 && carry)
        {
            ans[i] = 1;
            carry = 1;
        }
        else if ((a[i] == 1 && b[i] == 1) || (((a[i] == 1) || (b[i] ==
1)) && carry))
        {
            ans[i] = 0;
            carry = 1;
        }
        else
        {
            ans[i] = a[i] + b[i] + carry;
            carry = 0;
        }
    }
    return ans;
}

// Returns the right most shifted value from the q
bool arithmeticRightShift(vector<bool> &a, vector<bool> &q)
{
    bool kachra = q[q.size() - 1], prev = a[0], temp;
    for (int i = 1; i < a.size(); i++)
    {
        temp = a[i];
        a[i] = prev;
        prev = temp;
    }
    temp = q[0];
    q[0] = prev;
    prev = temp;
    for (int i = 1; i < q.size(); i++)
    {
        temp = q[i];
        q[i] = prev;
        prev = temp;
    }
    return kachra;
}

// Left Shifts the 2 numbers :)
void arithmeticLeftShift(vector<bool> &a, vector<bool> &q)
{
    bool kachra = q[q.size() - 1], prev = a[0], temp;
    for (int i = 0; i < a.size() - 1; i++)
```

```
    {
        a[i] = a[i + 1];
    }
    a[a.size() - 1] = q[0];
    for (int i = 0; i < q.size() - 1; i++)
    {
        q[i] = q[i + 1];
    }
}

int main()
{
    // Taking the input in string so that we don't need to ask for the
size of the number
    string qtemp, mtemp;
    // Then storing the number in a vector(Array) for easy access
    vector<bool> q, m, a, negM;
    bool qNeg = 0;

    // We'll be taking the input in binary format only
    cout << "Enter m: ";
    cin >> mtemp;
    cout << "Enter q: ";
    cin >> qtemp;

    // Counter
    int n;
    // Assigning the counter with the max value
    // if (mtemp.length() > qtemp.length())
    //     n = mtemp.length();
    // else
    //     n = qtemp.length();

    n = qtemp.length();

    int count = n;

    // Assigning the Accumulator with 0's
    for (int i = 0; i < n + 1; i++)
    {
        a.push_back(0);
    }

    int mtempNum = (n + 1) - mtemp.length();
    while (mtempNum > 0)
    {
        m.push_back(0);
```

```
        mtempNum--;
    }

    // Converting the string into vector(array)
    for (int i = 0; i < qtemp.length(); i++)
    {
        if (qtemp[i] == '1')
            q.push_back(1);
        else
            q.push_back(0);
    }
    for (int i = 0; i < mtemp.length(); i++)
    {
        if (mtemp[i] == '1')
            m.push_back(1);
        else
            m.push_back(0);
    }

    // Calculating the -M
    negM = twoSComplement(m);

    vector<bool> ans = binaryAddition(m, q, n);

    cout << "\nA\tQ\tn\tAction\n\n";

    while (count)
    {
        // --- Init Printing format ---
        for (auto x : a)
            cout << x;
        cout << "\t";
        for (auto x : q)
            cout << x;
        cout << "\t" << count << "\t"
             << (count == n ? "Init" : "n--")
             << "\n";
        // --- Init Printing format ---
        arithmeticLeftShift(a, q);
        // --- Shift LEFT Printing format ---
        for (auto x : a)
            cout << x;
        cout << "\t";
        int tempCount = 0;
        for (auto x : q)
        {
            tempCount++;
```

```
            cout << ((tempCount == n) ? "_" : to_string(x));
        }
        cout << "\t" << count << "\t"
             << "Shift LEFT"
             << "\n";
        // --- Shift LEFT Printing format ---

        // a[0] = 1 means the number is -ve
        if (a[0])
        {
            a = binaryAddition(a, m, n + 1);
            // --- A+M Printing format ---
            for (auto x : a)
                cout << x;
            cout << "\t";
            tempCount = 0;
            for (auto x : q)
            {
                tempCount++;
                cout << ((tempCount == n) ? "_" : to_string(x));
            }
            cout << "\t" << count << "\t"
                 << "A+M"
                 << "\n";
            // --- A+M Printing format ---
        }
        else
        {
            a = binaryAddition(a, negM, n + 1);
            // --- A-M Printing format ---
            for (auto x : a)
                cout << x;
            cout << "\t";
            tempCount = 0;
            for (auto x : q)
            {
                tempCount++;
                cout << ((tempCount == n) ? "_" : to_string(x));
            }
            cout << "\t" << count << "\t"
                 << "A-M"
                 << "\n";
            // --- A-M Printing format ---
        }
        q[q.size() - 1] = !a[0];
        // --- Q0<-!A(MSB) Printing format ---
        for (auto x : a)
```

```
            cout << x;
        cout << "\t";
        tempCount = 0;
        for (auto x : q)
        {
            tempCount++;
            cout << x;
        }
        cout << "\t" << count << "\t"
             << "Q0<-!A(MSB)"
             << "\n\n";
        // --- Q0<-!A(MSB) Printing format ---
        count--;
    };

    // a[0] = 1 means the number is -ve
    if (a[0])
    {
        a = binaryAddition(a, m, n + 1);
        // --- A<0, A+M Printing format ---
        for (auto x : a)
            cout << x;
        cout << "\t";
        int tempCount = 0;
        for (auto x : q)
        {
            tempCount++;
            cout << x;
        }
        cout << "\t" << count << "\t"
             << "A<0, A+M"
             << "\n\n";
        // --- A<0, A+M Printing format ---
    }

    cout << "Quotient: ";
    for (auto x : q)
        cout << x;

    cout << "\n";
    cout << "Remainder: ";
    for (auto x : a)
        cout << x;

    return 0;
}
```

Output

```
Enter m: 11
Enter q: 1011

A       Q       n       Action

00000   1011    4       Init
00001   011_    4       Shift LEFT
11110   011_    4       A-M
11110   0110    4       Q0<-!A(MSB)

11110   0110    3       n--
11100   110_    3       Shift LEFT
11111   110_    3       A+M
11111   1100    3       Q0<-!A(MSB)

11111   1100    2       n--
11111   100_    2       Shift LEFT
00010   100_    2       A+M
00010   1001    2       Q0<-!A(MSB)

00010   1001    1       n--
00101   001_    1       Shift LEFT
00010   001_    1       A-M
00010   0011    1       Q0<-!A(MSB)

Quotient: 0011
Remainder: 00010
```

```
Enter m: 11
Enter q: 111

A       Q       n       Action

0000    111     3       Init
0001    11_     3       Shift LEFT
1110    11_     3       A-M
1110    110     3       Q0<-!A(MSB)

1110    110     2       n--
1101    10_     2       Shift LEFT
0000    10_     2       A+M
0000    101     2       Q0<-!A(MSB)

0000    101     1       n--
0001    01_     1       Shift LEFT
1110    01_     1       A-M
1110    010     1       Q0<-!A(MSB)

0001    010     0       A<0, A+M

Quotient: 010
Remainder: 0001
```

Conclusion

The 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.

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 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.