PROCESSOR ARCHITECTURE

PRACTICAL 1

Division/Batch: B/B1
Branch: Computer Engineering

SAP ID	Name of Student	Date of Experiment	Date of Submission	Remarks
60004200107	Kartik Jolapara	20/9/21	3/10/21	

Aim

To implement Booth’s multiplication algorithm.

Theory

Booth's multiplication algorithm is an algorithm which multiplies 2 signed integers in 2's complement representation. This approach uses fewer additions and subtractions than more straightforward algorithms by processing in an efficient way, i.e., a smaller number of additions/subtractions required. It operates on the fact that strings of 0’s in the multiplier require no addition but just shifting and a string of 1’s in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^ (k+1) to 2^m.

The multiplicand and multiplier are placed in the m and Q registers respectively. A 1-bit register is placed logically to the right of the LSB (least significant bit) Q0 of Q register. This is denoted by Q-1. A and Q-1 are initially set to 0. Control logic checks the two bits Q0 and Q-1. If the two bits are same (00 or 11) then all the bits of A, Q, Q-1 are shifted 1 bit to the right. If they are not the same and if the combination is 10 then the multiplicand is subtracted from A and if the combination is 01 then the multiplicand is added with A. In both the cases results are stored in A, and after the addition or subtraction operation, A, Q, Q-1 are right shifted. The shifting is the arithmetic right shift operation where the left most bit namely, An-1 is not only shifted into An-2 but also remains in An-1. This is to preserve the sign of the number in A and Q. The result of the multiplication will appear in the A and Q.

The representations 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 2i 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.

Algorithm

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values A and Q to a product P, then performing a rightward arithmetic shift on P.

1. Set the Multiplicand and Multiplier binary bits as M and Q, respectively.
2. Initially, we set the AC and Q_{n+1} registers value to 0.
3. SC represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
4. A Qn represents the last bit of the Q, and the Q_{n+1} shows the incremented bit of Qn by 1.
5. On each cycle of the booth algorithm, Q_n and Q_{n+1} bits will be checked on the following parameters as follows:
   i. When two bits Q_n and Q_{n+1} are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product AC. And the bits of Qn and Q_{n+1} is incremented by 1 bit.
   ii. If the bits of Q_n and Q_{n+1} is shows to 01, the multiplicand bits (M) will be added to the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
   iii. If the bits of Q_n and Q_{n+1} is shows to 10, the multiplicand bits (M) will be subtracted from the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
6. The operation continuously works till we reached n - 1 bit in the booth algorithm.
7. Results of the Multiplication binary bits will be stored in the AC and QR registers.

Example

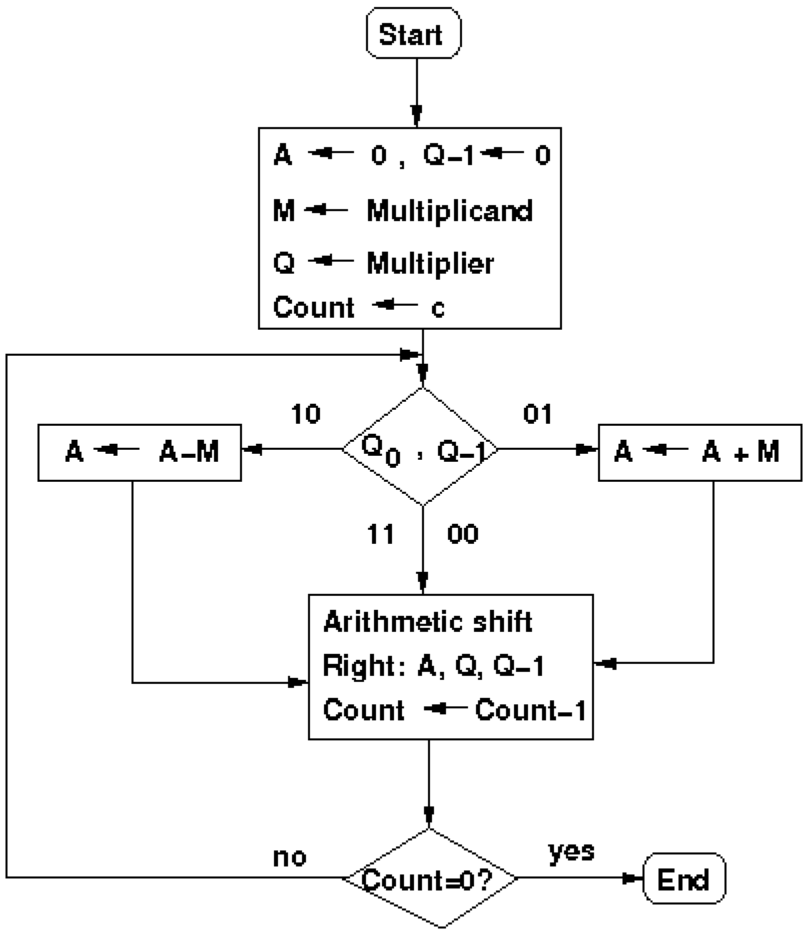
Problem: -5*-4

M=1011 Q=1100 -M=0101 N=4

N	A	Q	Q_{-1}	Operation
4	0000	1100	0	Initialization
	0000	0110	0	ARS, N=N-1
3	0000	0011	0	ARS, N=N-1
2	0101	0011	0	A=A-M
	0010	1001	1	ARS, N=N-1
1	0001	0100	1	ARS, N=N-1
0	0001	0100	1	Termination

Result : 00010100 : $(20)_{10}$

Flowchart



Code

```
// Documentation

// 1 - Aim

// 2 - Theory (What do you mean by boothe, Why, Flowchart, Algorithm)

// 3 - Code

// 4 - 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;
}

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(first number): ";
    cin >> mtemp;
    cout << "Enter q(second number): ";
    cin >> qtemp;

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

    int count = n;
```

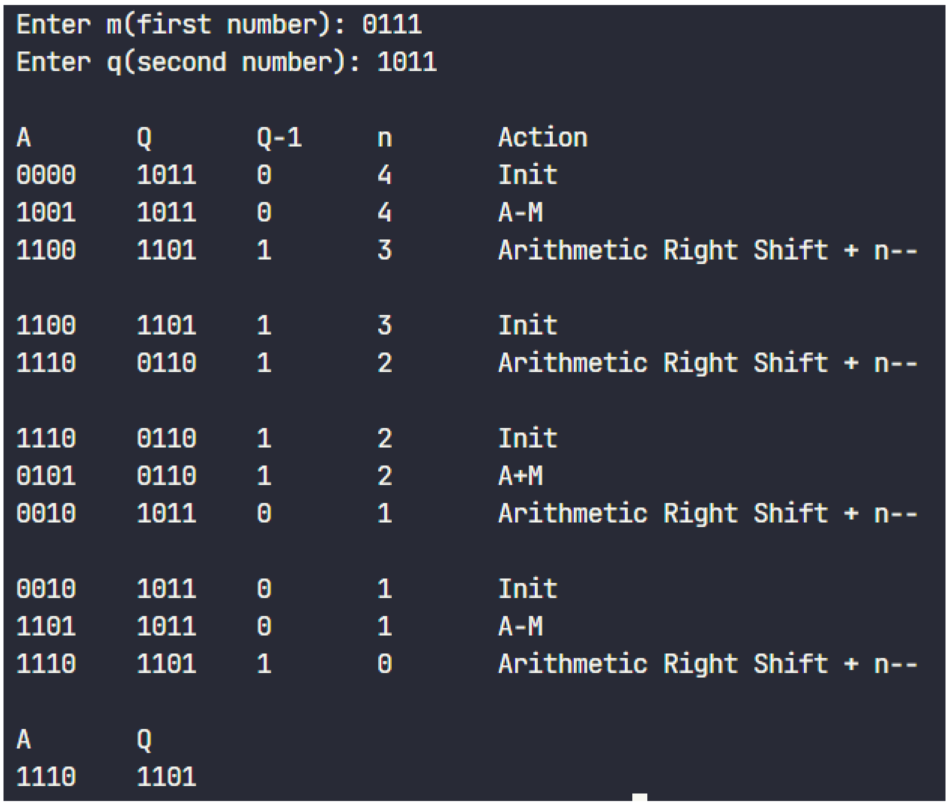
```
// Assigning the Accumulator with 0's
for (int i = 0; i < n; i++)
{
    a.push_back(0);
}

// 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\tQ-1\tn\tAction\n";

while (count)
{
```

```
// --- Init Printing format ---
for (auto x : a)
   cout << x;
cout << "\t";
for (auto x : q)
   cout << x;
cout << "\t" << qNeg << "\t" << count << "\t"
    << "Init"
    << "\n";
// --- Init Printing format ---

if (q[n - 1] == 1 && qNeg == 0)
{
   a = binaryAddition(a, negM, n);
   // --- A-M Printing format ---
   for (auto x : a)
      cout << x;
   cout << "\t";
   for (auto x : q)
      cout << x;
   cout << "\t" << qNeg << "\t" << count << "\t"
       << "A-M"
       << "\n";
   // --- A-M Printing format ---
}
else if (q[n - 1] == 0 && qNeg == 1)
{
   a = binaryAddition(a, m, n);
   // --- A+M Printing format ---
   for (auto x : a)
      cout << x;
```

```
            cout << "\t";
            for (auto x : q)
                cout << x;
            cout << "\t" << qNeg << "\t" << count << "\t"
                 << "A+M"
                 << "\n";
            // --- A+M Printing format ---
        }
        qNeg = arithmeticRightShift(a, q);
        count--;
        // --- Arithmetic Right Shift + n-- Printing format ---
        for (auto x : a)
            cout << x;
        cout << "\t";
        for (auto x : q)
            cout << x;
        cout << "\t" << qNeg << "\t" << count << "\t"
             << "Arithmetic Right Shift + n--"
             << "\n\n";
        // --- Arithmetic Right Shift + n-- Printing format ---
    };

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

    cout << "\t";
    for (auto x : q)
        cout << x;
    return 0;
}
```

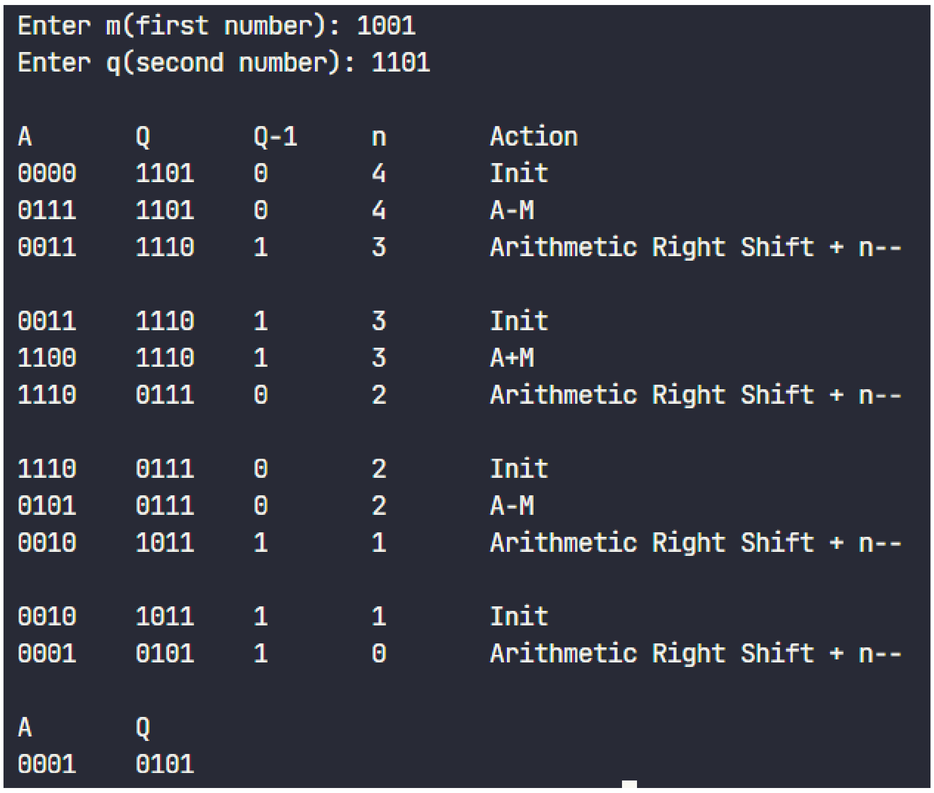
Output

1. 7(0111) * 11(1011)



```
Enter m(first number): 0111
Enter q(second number): 1011

A       Q       Q-1     n       Action
0000    1011    0       4       Init
1001    1011    0       4       A-M
1100    1101    1       3       Arithmetic Right Shift + n--

1100    1101    1       3       Init
1110    0110    1       2       Arithmetic Right Shift + n--

1110    0110    1       2       Init
0101    0110    1       2       A+M
0010    1011    0       1       Arithmetic Right Shift + n--

0010    1011    0       1       Init
1101    1011    0       1       A-M
1110    1101    1       0       Arithmetic Right Shift + n--

A       Q
1110    1101
```

2. -7(1001) * -3(1101)



```
Enter m(first number): 1001
Enter q(second number): 1101

A       Q       Q-1     n       Action
0000    1101    0       4       Init
0111    1101    0       4       A-M
0011    1110    1       3       Arithmetic Right Shift + n--

0011    1110    1       3       Init
1100    1110    1       3       A+M
1110    0111    0       2       Arithmetic Right Shift + n--

1110    0111    0       2       Init
0101    0111    0       2       A-M
0010    1011    1       1       Arithmetic Right Shift + n--

0010    1011    1       1       Init
0001    0101    1       0       Arithmetic Right Shift + n--

A       Q
0001    0101
```

Conclusion

The Booths algorithm is an efficient way to perform binary multiplication compared to traditional additive 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.