

Experiment 4: Booth's Multiplication Algorithm

Objective:

1. To implement Booth's Multiplication Algorithm.

Theory:

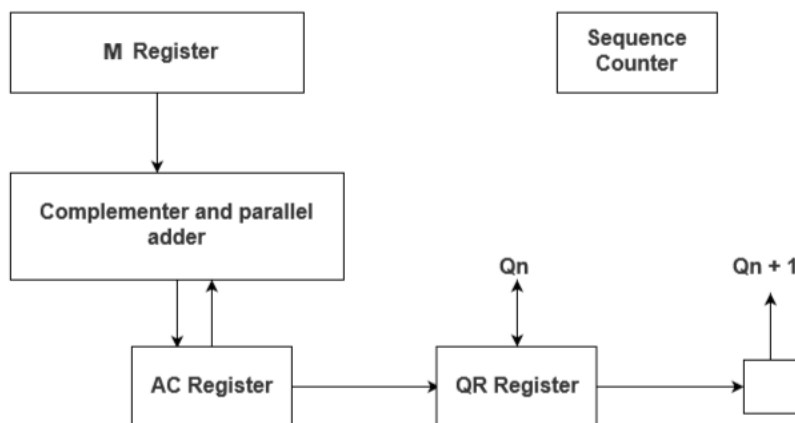
Booth's algorithm is a technique used for multiplying binary numbers. It reduces the number of operations required for multiplication by encoding the multiplier in a specific way. When compared to traditional multiplication algorithms, it treats both positive and negative numbers in a consistent manner, which makes it highly efficient for computer systems where signed numbers are commonly used.

Hardware Implementation

In Booth's Algorithm, the multiplication process involves:

- Inspecting two consecutive bits of the multiplier at a time.
- Adding, subtracting, or leaving unchanged the current product based on these bits.
- Shifting the product right after each operation gradually forms the final result.

Implementation of Booth's Algorithm



Algorithm:

1. Initially, $BR \leftarrow \text{Multiplicand}$ and $QR \leftarrow \text{Multiplier}$.
2. $AC \leftarrow 0$, $Q_{n+1} \leftarrow 0$ and $SC \leftarrow n$ (no of bits in multiplier).
3. Inspect Q_n, Q_{n+1} ,
 - i. If ($Q_n Q_{n+1} = 10$), first 1 in a string of 1's has been encountered, subtraction required.
i.e., $AC \leftarrow AC + \overline{BR} + 1$
 - ii. If ($Q_n Q_{n+1} = 01$), first 0 in a string of 0's has been encountered, addition required.
i.e., $AC \leftarrow AC + BR$
 - iii. If ($Q_n Q_{n+1} = 00$ or 11 (equal)), do nothing
4. Shift right the partial product and the multiplier (including bit Q_{n+1}).
This Arithmetic Shift Right (ashr) operation shifts AC and QR to the right and leaves the sign bit in AC unchanged.
5. $SC \leftarrow SC - 1$. If ($SC = 0$) stop the process else continue and go to step 3.
6. Result in AC and QR

Flowchart:

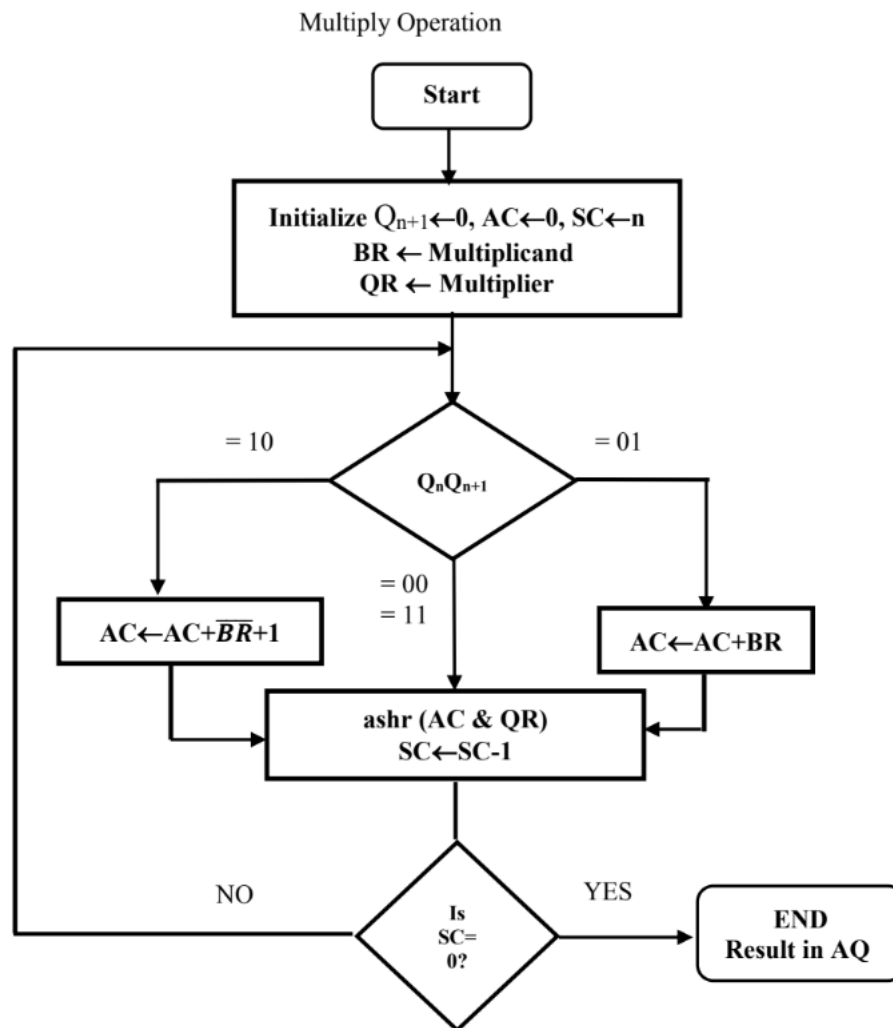


Fig: Booths Algorithm for multiplication of Signed-2's Complement numbers

Source code:

```
#include <stdio.h>
void add(int a[], int x[], int q);
void complement(int a[], int n)
{
    int i;
    int x[8] = {0};
    x[0] = 1;
    for (i = 0; i < n; i++)
    {
        a[i] = (a[i] + 1) % 2;
    }
    add(a, x, n);
}
void add(int ac[], int x[], int q)
{
    int i, c = 0;
    for (i = 0; i < q; i++)
    {
        ac[i] = ac[i] + x[i] + c;
        if (ac[i] > 1)
        {
            ac[i] = ac[i] % 2;
        }
    }
}
```

```
c = 1;
}
else
    c = 0;
}
}

void ashr(int ac[], int qr[], int *qn, int q)
{
    int temp, i;
    temp = ac[0];
    *qn = qr[0];
    printf("\t\tashr\t\t");
    for (i = 0; i < q - 1; i++)
    {
        ac[i] = ac[i + 1];
        qr[i] = qr[i + 1];
    }
    qr[q - 1] = temp;
}

void display(int ac[], int qr[], int qrn)
{
    int i;
    for (i = qrn - 1; i >= 0; i--)
        printf("%d", ac[i]);
    printf(" ");
    for (i = qrn - 1; i >= 0; i--)
        printf("%d", qr[i]);
}

int main()
{
    int mt[10], br[10], qr[10], sc, ac[10] = {0};
    int brn, qrn, i, qn, temp;
    printf("\t=== Compiled by Jonash Chataut ===\n");
    printf("\t===== Booth's Algorithm =====\n");
    printf("\n Number of multiplicand bit= ");
    scanf("%d", &brn);
    printf("\nmultiplicand= ");

    for (i = brn - 1; i >= 0; i--)
        scanf("%d", &br[i]); // multiplicand
    for (i = brn - 1; i >= 0; i--)
        mt[i] = br[i];
    complement(mt, brn);
    printf("\nNo. of multiplier bit= ");
    scanf("%d", &qrn);
    sc = qrn;
    printf("Multiplier= ");

    for (i = qrn - 1; i >= 0; i--)
        scanf("%d", &qr[i]);
    qn = 0;
    temp = 0;
    printf("qn\tq[n+1]\t\tBR\tAC\tQR\t\tsc\n");
    printf("\t\t\tinitial\t\t");
```

```

display(ac, qr, qrn);
printf("\t\t%d\n", sc);
while (sc != 0)
{
    printf("%d\t%d", qr[0], qn);
    if ((qn + qr[0]) == 1)
    {
        if (temp == 0)
        {
            add(ac, mt, qrn);
            printf("\t\tsubtracting BR\t");
            for (i = qrn - 1; i >= 0; i--)
                printf("%d", ac[i]);
            temp = 1;
        }
        else if (temp == 1)
        {
            add(ac, br, qrn);
            printf("\t\tadding BR\t");
            for (i = qrn - 1; i >= 0; i--)
                printf("%d", ac[i]);
            temp = 0;
        }
        printf("\n\t");
        ashr(ac, qr, &qn, qrn);
    }
    else if (qn - qr[0] == 0)
        ashr(ac, qr, &qn, qrn);
    display(ac, qr, qrn);
    printf("\t");
    sc--;
    printf("\t\t%d\n", sc);
}
printf("Result= ");
display(ac, qr, qrn);
}

```

Output:

```
C:\csit\third sem Jonash\CA\C  X  +  v

=== Compiled by Jonash Chataut ===
===== Booth's Algorithm =====

Number of multiplicand bit= 4

multiplicand= 0 1 1 1

No. of multiplier bit= 4
Multiplier= 0 0 1 0
qn      q[n+1]      BR      AC      QR      sc
initial      0000 0010      4
0      0      ashr      0000 0001      3
1      0      subtracting BR 1001
      ashr      1100 1000      2
0      1      adding BR 0011
      ashr      0001 1100      1
0      0      ashr      0000 1110      0
Result= 0000 1110
-----
Process exited after 17.13 seconds with return value 0
Press any key to continue . . . |
```

Conclusion:

Booth's Algorithm is an efficient way to multiply signed binary numbers using shifts and additions. It handles both positive and negative inputs with ease and mimics how real hardware performs multiplication.

Experiment 5: Restoring Division Algorithm

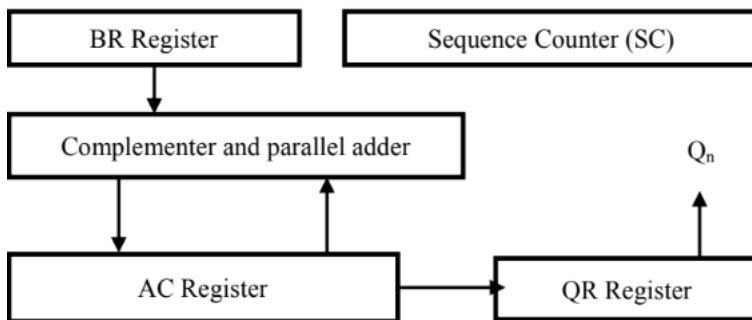
Objective:

1. To implement Restoring Division Algorithm.

Theory:

The restoring division algorithm is a method used for binary division in computer architecture. It involves repeatedly subtracting the divisor from the dividend and restoring the partial remainder if the result is negative. The algorithm uses a sign bit to track the sign of the partial remainder and adjusts accordingly. The process continues until the division is complete, and the quotient and remainder are obtained.

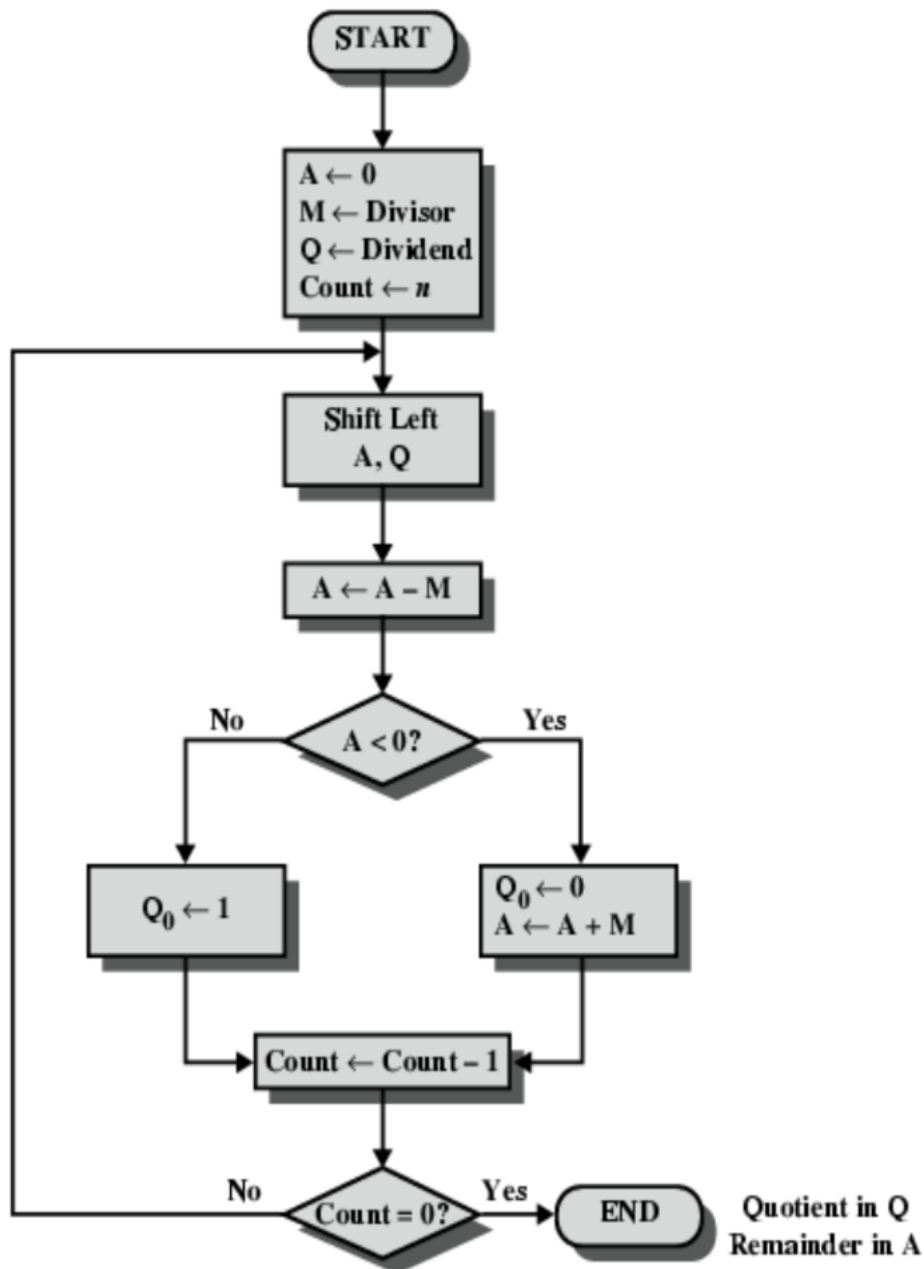
Hardware Implementation



Algorithm:

- Step 1:** Initialize A, Q and M registers to zero, dividend and divisor respectively and counter to n where n is the number of bits in the dividend.
- Step 2:** Shift A, Q left one binary position.
- Step 3:** Subtract M from A placing answer back in A. If sign of A is 1, set Q to zero and add M back to A (restore A). If sign of A is 0, set Q to 1.
- Step 4:** Decrease counter; if counter > 0, repeat process from step 2 else stop the process. The final remainder will be in A and quotient will be in Q.

Flowchart:



Source code:

```
#include <stdio.h>
void print_binary(int num, int bits)
{
    int i;
    for (i = bits - 1; i >= 0; i--)
    {
        printf("%d", (num >> i) & 1);
    }
}
void print_step(int step, const char *op, int A, int Q, int bits)
{
    printf("| %-12s | ", op);
    print_binary(A, bits);
    printf(" | ");
    print_binary(Q, bits);
    if (op == "Initialize")
        printf(" | %2d |\n", step);
    else if (op == "Restore A" || op == "Set Q0=1")
```

```

    printf(" | %2d |\n", step - 1);
else
    printf(" |  |\n");
}

int main()
{
    int dividend, divisor;
    int bits = 4;
    int A, Q, M;
    int sc;
    printf("\t=== Compiled by Jonash Chataut ===\n");
    printf("==== Restoring Division Algorithm (4-bit) =====\n");
    printf("Enter Dividend Q (0 - 15): ");
    scanf("%d", &dividend);
    printf("Enter Divisor M (0 - 15): ");
    scanf("%d", &divisor);
    A = 0;
    Q = dividend;
    M = divisor;
    sc = bits;
    printf("\n+-----+-----+-----+-----+\n");
    printf("| Operation | A | Q | SC |\n");
    printf("+-----+-----+-----+-----+\n");
    print_step(sc, "Initialize", A, Q, bits);

    for (sc = bits; sc > 0; sc--)
    {
        // Shift AQ left
        A = (A << 1) | ((Q >> (bits - 1)) & 1);
        Q <<= 1;
        print_step(sc, "Shift AQ", A, Q, bits);

        // Subtract M from A
        A -= M;
        print_step(sc, "A = A - M", A, Q, bits);

        if (A < 0)
        {
            A += M; // Restore
            print_step(sc, "Restore A", A, Q, bits);
        }
        else
        {
            Q |= 1; // Set LSB
            print_step(sc, "Set Q0=1", A, Q, bits);
        }
    }
    printf("+-----+-----+-----+-----+\n");
    printf("\nResult: Quotient = %d, Remainder = %d\n", Q & 0xF, A & 0xF); // Mask to 4 bits
}

```

Output:

```
C:\csit\third sem Jonash\CA\C  x  +  v

=== Compiled by Jonash Chataut ===
===== Restoring Division Algorithm (4-bit) =====
Enter Dividend Q (0 - 15): 14
Enter Divisor M (0 - 15): 2

+-----+-----+-----+-----+
| Operation | A | Q | SC |
+-----+-----+-----+-----+
| Initialize | 0000 | 1110 | 4 |
| Shift AQ | 0001 | 1100 | |
| A = A - M | 1111 | 1100 | |
| Restore A | 0001 | 1100 | 3 |
| Shift AQ | 0011 | 1000 | |
| A = A - M | 0001 | 1000 | |
| Set Q0=1 | 0001 | 1001 | 2 |
| Shift AQ | 0011 | 0010 | |
| A = A - M | 0001 | 0010 | |
| Set Q0=1 | 0001 | 0011 | 1 |
| Shift AQ | 0010 | 0110 | |
| A = A - M | 0000 | 0110 | |
| Set Q0=1 | 0000 | 0111 | 0 |
+-----+-----+-----+-----+

Result: Quotient = 7, Remainder = 0

-----
Process exited after 12.42 seconds with return value 0
Press any key to continue . . .
```

Conclusion:

The Restoring Division Algorithm provides a systematic method to divide binary numbers using shifting and subtraction. It is simple to implement in C and effectively handles unsigned binary division, giving accurate quotient and remainder.