- =Experiment 1

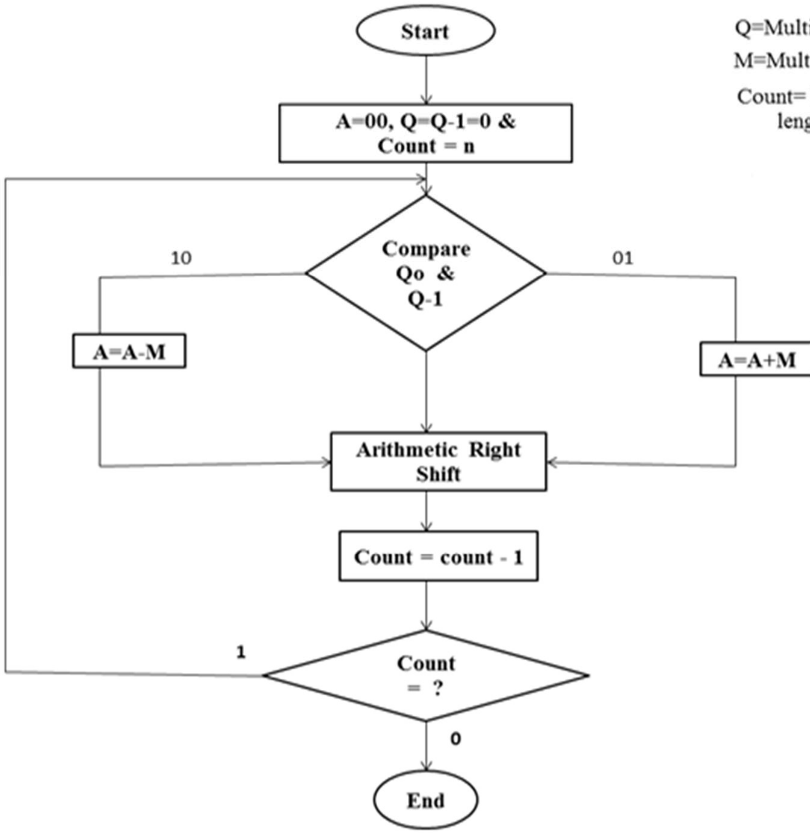
(Booth's multiplication)

Aim: Implement Booth's multiplication algorithm.

Theory:

- Booth algorithms gives a procedure for multiplying binary integers in signed 2's complement representation in 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.
- As in all multiplication schemes, booth algorithms require examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:
- The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier
- The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous '1') in a string of 0's in the multiplier.
- The partial product does not change when the multiplier bit is identical to the previous multiplier bit.
- **Example –** A numerical example of booth's algorithm is shown below for n = 4. It shows the step by step multiplication of 7 and 5.

Q=Multiplicand
M=Multiplier
Count= No. of bit length



Perform 7*5 using Booth's Algorithm

A	Q	Q-1	M		
0000	0101	0	0111	Initial value	
1001	0101	0	0111	A ← A–M	First cycle
1100	1010	1	0111	shift	
0011	1010	1	0111	A ← A+M	Second cycle
0001	1101	0	0111	shift	
1010	1101	0	0111	A ← A–M	Third cycle
1101	0110	1	0111	shift	
0100	0110	1	0111	A ← A+M	Fourth cycle
0010	0011	0	0111	shift	

CODE:

BOOTH'S MULTIPLICATION(SF1)

Divyesh khunt 60009210116

```
[2] import numpy as np
    import math
```

```
#decimal to binary
def d2b(decimal_num,size):

  if decimal_num>=0:
    binary_num = ""
    if decimal_num == 0:
        return "0".zfill(size)
    while decimal_num > 0:
        remainder = decimal_num % 2
        binary_num = str(remainder) + binary_num
        decimal_num //= 2
    return binary_num.zfill(size)


#working of function
print(d2b(3,4))
print(d2b(8,5))
```

```
0011
01000
```

```
def b2d(binary):
    decimal = 0
    power = len(str(binary)) - 1
    for digit in str(binary):
        decimal += int(digit) * 2**power
        power -= 1
    return decimal
#working
print('the deciaml of bianry 100 is',b2d('100'))
```

```
the deciaml of bianry 100 is 4
```

```python
def complement(num, bits):
    """Returns the two's complement of a binary number."""
    flipped = ''.join('1' if c == '0' else '0' for c in num)
    twos_comp = add_binary_nums(flipped, d2b(1, bits))
    return twos_comp
#working
print('The binary of 3 is',d2b(3,4),'\n2s complement',complement(d2b(3,4),4))

print('The binary of 6 is',d2b(6,5),'\n2scomplement ',complement(d2b(6,5),5))
```

```
The binary of 3 is 0011
2s complement 1101
The binary of 6 is 00110
2scomplement  11010
```

```python
def add(x, y):
    max_len = max(len(x), len(y))
    x = x.zfill(max_len)
    y = y.zfill(max_len)
    result = ''
    carry = 0
    for i in range(max_len-1, -1, -1):
        r = carry
        r += 1 if x[i] == '1' else 0
        r += 1 if y[i] == '1' else 0
        result = ('1' if r % 2 == 1 else '0') + result
        carry = 0 if r < 2 else 1
    if carry !=0 : result = '1' + result
    return result.zfill(max_len)

r=add(d2b(4,4),d2b(5,6))
print("The result in decimal:",b2d(r),"\nThe result in binary is:",r)
```

```
The result in decimal: 9
The result in binary is: 001001
```

```python
def d2b(dec):
    if int(dec)<0:
        bin = twos_complement(int(dec))
    else:
        bin = "{0:b}".format(int(dec))
        for i in range(4-len(bin)):
            bin = "0" + bin
    return bin

def twos_complement(dec):
    adjusted = abs(int(dec) + 1)
    binint = "{0:b}".format(adjusted)
    flipped = flip(binint)
    for i in range(4-len(flipped)):
        flipped = "1" + flipped
    return flipped

def flip(string):
    flipped_string = ""
    for bit in string:
        if bit == "1":
            flipped_string += "0"
        else:
            flipped_string += "1"
    return flipped_string
```

```python
def boothsTriumph(m, q):
    print("multipcand: " + m + " multiplier: " + q)
    prod = "00000000" + q + "0"
    print("Product: " + prod)
    print(buildLine(0,m,prod))
    for i in range(1,5):
        operation = prod[len(prod)-2:]
        prod = op(prod,m,operation)
        print(buildLine(i,m,prod))
    prod = shift(prod)
    prod = prod[5:9]
    print("Product: " + prod)
    return
```

```
def shift(prod):
    prod = "0"+prod[:len(prod)-1]
    return prod

def binAdd(num, num2):
    prod = ""
    carry = "0"
    for i in range(len(num)-1,-1,-1):
        if carry == "0":
            if num[i] == "0" and num2[i] == "0":
                prod = "0" + prod
            elif num[i] == "1" and num2[i] == "1": #case 1 and 1
                prod = "0" + prod
                carry = "1"
            else:
                prod = "1" + prod
        elif carry == "1":
            if num[i] == "0" and num2[i] == "0":
                prod = "1" + prod
                carry = "0"
            elif num[i] == "1" and num2[i] == "1": #case 1 and 1
                prod = "1" + prod
                carry = "1"
            else:
                prod = "0" + prod
                carry = "1"
    return prod
```

```
def buildLine(iteration, m, prod):
    line = "Step: " + str(iteration) + " | Multiplicand: " + m + " | Product: " \
    + prod[0:8] + " | " + prod[8:16] + " | " + prod[16]
    return line

M = int(input(" Enter Mutiplicand: "))
Q = int(input(" Enter Mutiplier: "))
size=int(input("Enter size of register"))
n1 = d2b(M)
n2 = d2b(Q)
boothsTriumph(n1,n2)
print("Decimal Result: " + str(int(M)*int(Q)))
```

```
#this fuction tell which operation to perfom
def op(prod,m,operation):
    if operation == "00":
        prod = shift(prod)
        print("No Op")
        return prod
    elif operation == "01":
        temp = binAdd(prod[0:4],m)
        prod = temp + prod[4:]
        prod = shift(prod)
        print("Add")
        return prod
    elif operation == "10":
        prod = subtraction(prod,m)
        prod = shift(prod)
        print("Sub")
        return prod
    elif operation == "11":
        prod = shift(prod)
        print("No Op")
        return prod
    else:
        return 0
```

```
def subtraction(prod,m):
    carry = 0
    prime_prod = prod[:8]
    final_prod = ""
    for i in range(len(prime_prod)-1,-1,-1):
        if (m[i] == "0" and prime_prod[i] == "0"):
            if (carry == 1):
                final_prod = "1" + final_prod
            else:
                final_prod = "0" + final_prod
        elif (m[i] == "1" and prime_prod[i] == "0"):
            if (carry == 1):
                final_prod = "0" + final_prod
            else:
                final_prod = "1" + final_prod
                carry = 1
        elif (m[i] == "0" and prime_prod[i] == "1"):
            if (carry == 1):
                final_prod = "0" + final_prod
                carry = 0
            else:
                final_prod = "1" + final_prod
        elif (m[i] == "1" and prime_prod[i] == "1"):
            if (carry == 1):
                final_prod = "1" + final_prod
                carry = 1
            else:
                final_prod = "0" + final_prod
        else:
            return 0
    return final_prod + prod[8:]
```

Lab Assignments to complete in this session

1. Perform binary multiplication of –7 and –3 using booths algorithm and register size=4 bits

```
 Enter Mutiplicand: -7
 Enter Mutiplier: -3
Enter size of register4
multipcand: 11111001 multiplier: 11111101
Product: 0000000111111010
Step: 0 | Multiplicand: 11111001 | Product: 00000000 | 11111101 | 0
Sub
Step: 1 | Multiplicand: 11111001 | Product: 00000011 | 11111110 | 1
Add
Step: 2 | Multiplicand: 11111001 | Product: 01111110 | 01111111 | 0
Sub
Step: 3 | Multiplicand: 11111001 | Product: 01000010 | 10111111 | 1
No Op
Step: 4 | Multiplicand: 11111001 | Product: 00100001 | 01011111 | 1
No Op
Step: 5 | Multiplicand: 11111001 | Product: 00010000 | 10101111 | 1
No Op
Step: 6 | Multiplicand: 11111001 | Product: 00001000 | 01010111 | 1
No Op
Step: 7 | Multiplicand: 11111001 | Product: 00000100 | 00101011 | 1
No Op
Step: 8 | Multiplicand: 11111001 | Product: 00000010 | 00010101 | 1
Product: 00010101
Decimal Result: 21
```

2. Perform binary multiplication of –9 and 7 using booths algorithm and register size=5 bits

```
 Enter Mutiplicand: -9
 Enter Mutiplier: 7
Enter size of register: 5
multipcand: 11110111 multiplier: 00000111
Product: 0000000000001110
Step: 0 | Multiplicand: 11110111 | Product: 00000000 | 00000111 | 0
Sub
Step: 1 | Multiplicand: 11110111 | Product: 00000100 | 10000011 | 1
No Op
Step: 2 | Multiplicand: 11110111 | Product: 00000010 | 01000001 | 1
No Op
Step: 3 | Multiplicand: 11110111 | Product: 00000001 | 00100000 | 1
Add
Step: 4 | Multiplicand: 11110111 | Product: 01111100 | 00010000 | 0
No Op
Step: 5 | Multiplicand: 11110111 | Product: 00111110 | 00001000 | 0
No Op
Step: 6 | Multiplicand: 11110111 | Product: 00011111 | 00000100 | 0
No Op
Step: 7 | Multiplicand: 11110111 | Product: 00001111 | 10000010 | 0
No Op
Step: 8 | Multiplicand: 11110111 | Product: 00000111 | 11000001 | 0
Product: 11000001
Decimal Result: -63
```

3. Perform binary multiplication of –13 and –6 using booths algorithm and register size=5 bits

```
 Enter Mutiplicand: -13
 Enter Mutiplier: -6
Enter size of register: 5
multipcand: 11110011 multiplier: 11111010
Product: 0000000011110100
Step: 0 | Multiplicand: 11110011 | Product: 00000000 | 11111010 | 0
No Op
Step: 1 | Multiplicand: 11110011 | Product: 00000000 | 01111101 | 0
Sub
Step: 2 | Multiplicand: 11110011 | Product: 00000110 | 10111110 | 1
Add
Step: 3 | Multiplicand: 11110011 | Product: 01111100 | 11011111 | 0
Sub
Step: 4 | Multiplicand: 11110011 | Product: 01000100 | 11101111 | 1
No Op
Step: 5 | Multiplicand: 11110011 | Product: 00100010 | 01110111 | 1
No Op
Step: 6 | Multiplicand: 11110011 | Product: 00010001 | 00111011 | 1
No Op
Step: 7 | Multiplicand: 11110011 | Product: 00001000 | 10011101 | 1
No Op
Step: 8 | Multiplicand: 11110011 | Product: 00000100 | 01001110 | 1
Product: 01001110
Decimal Result: 78
```