Computer Organization

(Booth's Algorithm Project)

❑Objective

To implement Booth's algorithm for multiplication and division of binary numbers.

Multiplication

The following flowchart explains Booth's Algorithm for the multiplication of two numbers given as input in decimal numbers.

- ★ Multiplier and multiplicand are placed in registers M and B. There is also one register Q that is placed to the right of least significant bit of B which is assigned '0' initially.
- ★ Now the last two bits of B_0Q are compared and accordingly the action is taken as explained in flow chart.
- ★ If both the bits are the same, then directly Arithmetic Right Shift occurs, and if both are different then accordingly the multiplicand is subtracted and added in the accumulator and after that arithmetic right shift takes place.
- ★ This continues in a for loop running as many times as the length of the multiplicand.

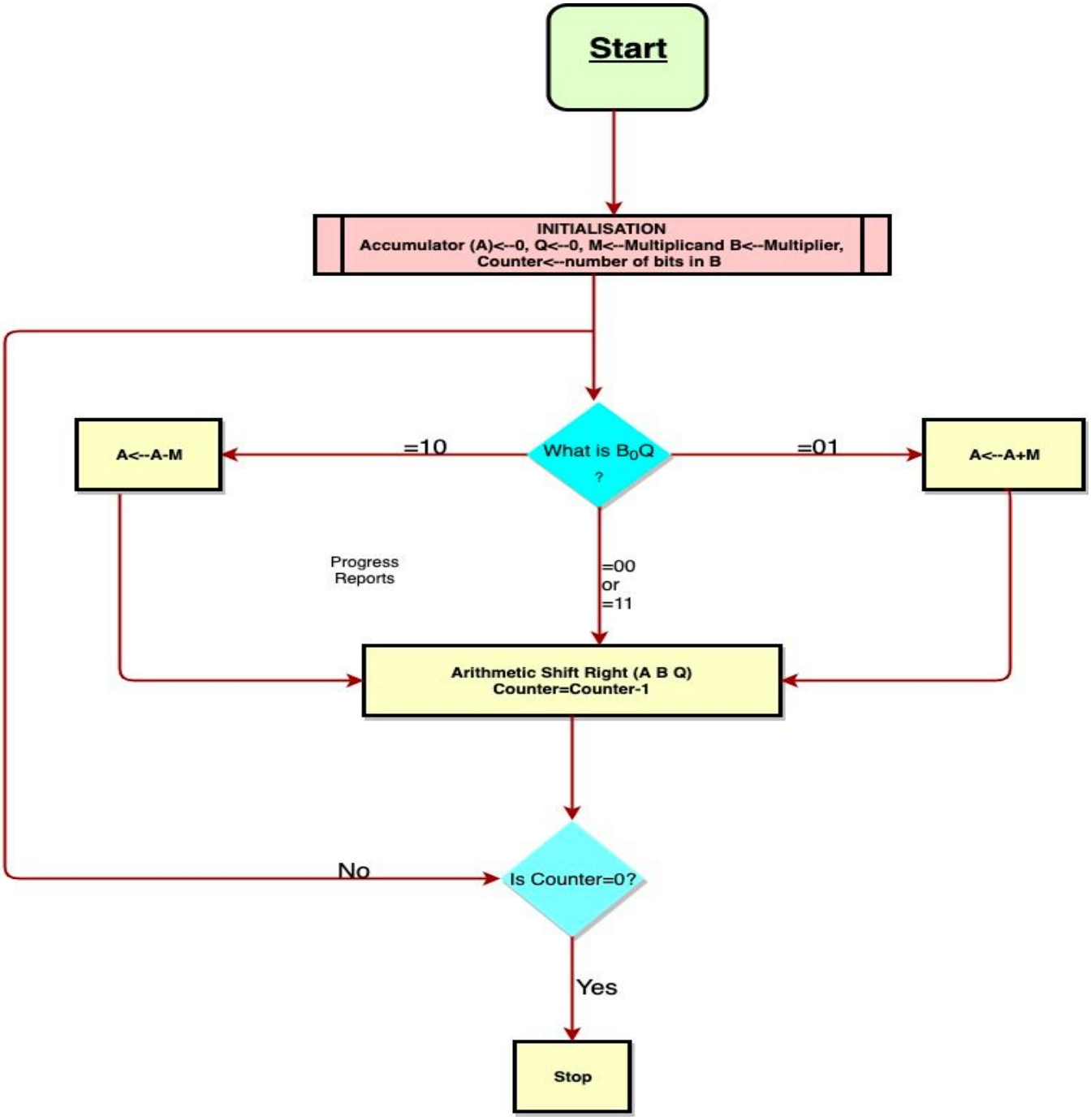
Complexity Analysis

If we consider n as the number of bits in the binary number of the multiplier, then since we have the loop of that length, the complexity becomes n*(time taken in addition or subtraction respectively). Now since the addition and subtraction is done 'n' number of times, that is every time we encounter different bits. Therefore the complexity is $O(n^2)$ if we consider n as the number of bits is binary.

Test Cases

Given 'a' as Multiplier and 'b' as Multiplicand

1. (a,b) = (-17,10)
2. (a,b) = (10,-17)
3. (a,b) = (7,19)
4. (a,b) = (-7,19)
5. (a,b) = (5,0)


Start
INITIALISATION
Accumulator (A)<--0, Q<--0, M<--Multiplicand B<--Multiplier,
Counter<--number of bits in B
What is B_0Q ?
=10
=01
A<--A-M
A<--A+M
Progress
Reports
=00
or
=11
Arithmetic Shift Right (A B Q)
Counter=Counter-1
Is Counter=0?
No
Yes
Stop

Division

The following Flowchart explains the Algorithm used since there are no Booth's Algorithm, the Division Algorithm used is a type of Non-Restoring Algorithm modified to work with signed numbers.

Complexity:

n is the number of bits required the larger of the two operands

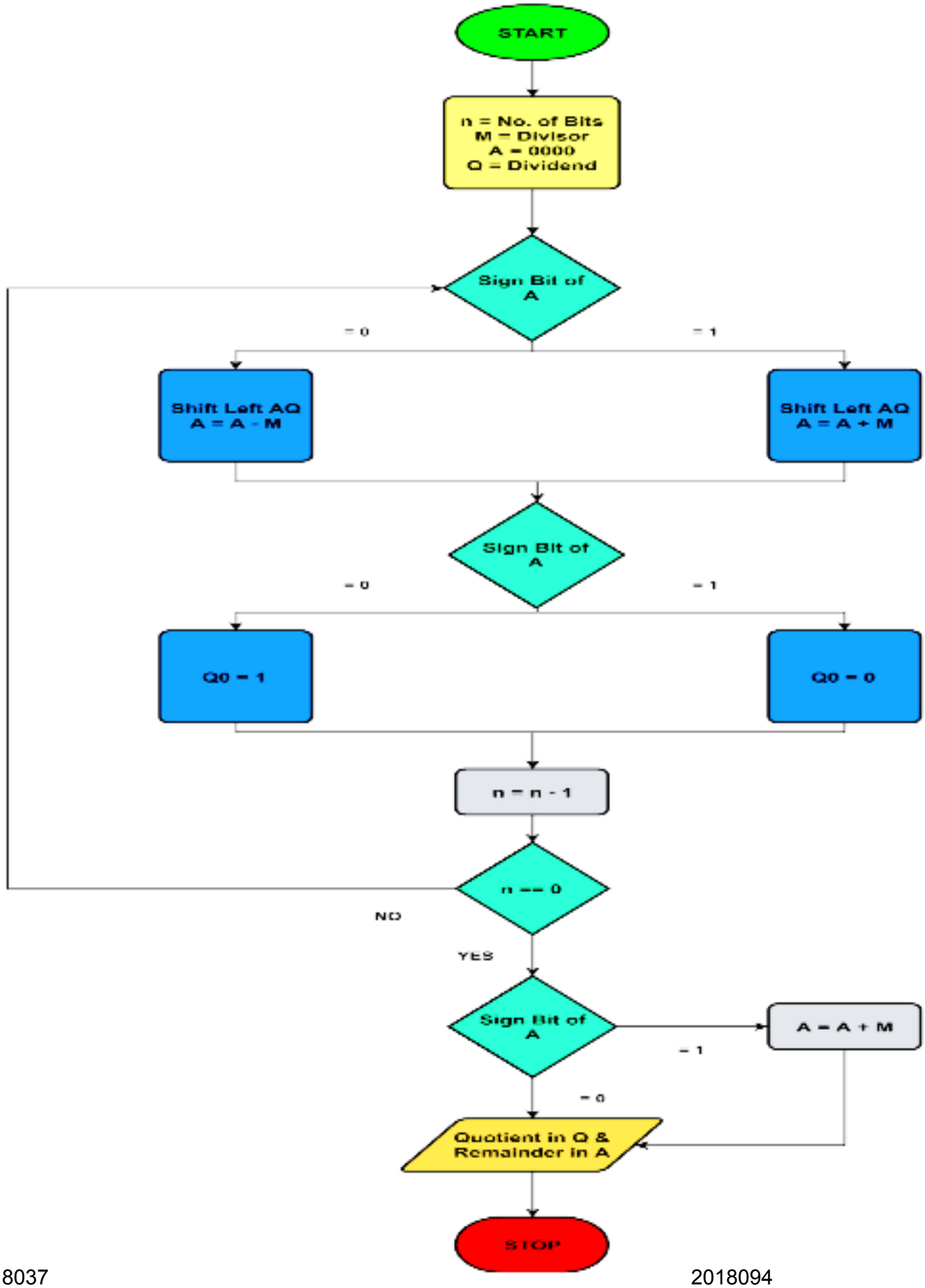
1. shiftL :- O(1) // Shifts Left the binary number
2. Comp2s :- O(n) // Calculates two's complement (n length loop)
3. _add(a,b) :- O(n) // Adds two binary number (n length loop)
4. _div(a, b) :- O(n^2) // Calculates the Quotient and Remainder
   // The _div method uses a '**n'** length loop in which every loop
   // instance calls the _add function thus the complexity

If we had used Internal function to **add two numbers in decimal** and then converted the number back to binary it would have taken O(1) time to do the task which would make the program more efficient making **_div(a,b)'s complexity O(n).**

Test Cases:

Given 'a' as Dividend and 'b' as Divisor

6. (a,b) = (-10,10)
7. (a,b) = (10,-10)
8. (a,b) = (27,19)
9. (a,b) = (-27,19)
10. (a,b) = (5,0) // Error
11. (a,b) = (0,5)
12. (a,b) = (-6,-4)


START
n = No. of Bits
M = Divisor
A = 0000
Q = Dividend
Sign Bit of A
= 0
= 1
Shift Left AQ
A = A - M
Shift Left AQ
A = A + M
Sign Bit of A
= 0
= 1
Q0 = 1
Q0 = 0
n = n - 1
n == 0
NO
YES
Sign Bit of A
= 1
A = A + M
= 0
Quotient in Q &
Remainder in A
STOP