UNIT-III

Data Representation:

Binary information in digital computers is stored in memory (or) processor registers

Registers can contain either data (or) control information.

→ Control information is a bit (or) group of bits used to specify the sequence of command signals needed for manipulation of the data in other registers

→ Data are numbers and other binary-coded information that are operated on to achieve the required computational results.

Number Systems

Decimal

Base or Radix is 10
(No of digits that are used in that Number System)

Digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ex

$(4562.92)_{10}$ ($10^3\ 10^2\ 10^1\ 10^0\ 10^{-1}\ 10^{-2}$)

Interpreted as

$4\times10^3 + 5\times10^2 + 6\times10^1 + 2\times10^0 + (9\times10^{-1}) + (2\times10^{-2})$

$\Rightarrow 4000 + 500 + 600 + 2 + 9\times\frac{1}{10} + 5\times\frac{1}{100}$

Octal

Base (or) Radix is 8

Digits : 0, 1, 2, 3, 4, 5, 6, 7.

Ex

$(162.4)_8$ is interpreted as

$1\times8^2 + 6\times8^1 + 2\times8^0 + 4\times8^{-1}$

$= 64 + 48 + 2 + \frac{1}{2}$

Binary

Base or Radix is 2

Digits 0, 1

Ex

$(110110)_2$ ($2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$)

$\Rightarrow 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$

$\Rightarrow 32 + 16 + 0 + 4 + 2 + 0$

$= (54)_{10}$

Hexadecimal

Base - 16

Digits

0 1 2 3 4 5 6 7 8 9
A B C D E F

$(2B)_{16}$

$(2\times16^1) + (B\times16^0)$

$\Rightarrow 32 + 11$

Number System Conversions

① Decimal to Other Base systems

Step 1: Divide the decimal Number to be Converted by the value of the New Base

$S_1 = \frac{29}{2} \Rightarrow$

Q	R
14	1
7	0
$\frac{14}{2}$	
⋮	
↓	

$(29)_{10} = (11101)_2$

* Other Base systems to Decimal

S_1: Determine Common value of Each digit

S_2: Multiply the Obtained Column values in (Step 1) by the digits in the corresponding Columns

S_3: Sum the products Calculated in Step-2

S_4 The sum total is equivalent to decimal.

$(11101)_2$

$1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 16 + 8 + 4 + 0 + 1$

$\Rightarrow (29)_{10}$

UNIT-II

DATA TYPES

* Binary information is stored in memory (or) Processor registers
* Register's Contain Either data (or) control information
* Data are Numbers and Other binary coded information
* Control information is a bit (or) a group of bits used to specify the sequence of Commond signals needed for manipulation of the data in other registers.

Data types found in the Register's of digital Computers may be classified as being one of the following:

1) Numbers used in Arithmatic Computations
2) Letters of the alphabet used in data processing
3) Other discrete symbols used for Specific purpose.

Number Systems

All types of data, Except binary numbers, are represented in computer registers in binary-Coded form.

* This is because registers are made up of flip-flops and flip-flops are two-state devices that can store only 1's and 0's

* Base (or) Radix : Uses distinct symbols for r digits

Most Common Number System:

1) Decimal. 2) Binary 3) Octal

4) Hexadecimal

* Binary coded - Decimal code

Each digit of a decimal Number is Represented by its binary Equivalent

874 (decimal)
1000 0111 0100 (BCD)

Only 4 bit Binary Numbers are used from 0000 – 1001

Comparision of BCD and Binary

137_{10} = $1000\,100\,1_2$ (Binary – 8 bits)

137_{10} = 0001 0011 0111 (BCD) (12 bits)

fixed point Representation

→ positive integers, including zero, can be represented as unsigned numbers.

→ However to represent negative integers we need a notation for Negative Values

→ In ordinary Arithmatic, a Negative Number is indicated by a minus sign and a positive Number by a plus sign

* Because of Hardware limitations Computers must Represent Everything with 1's and 0's including the sign of a Number.

→ As a Consequence, it is Customary to Represent the sign with a bit placed in the leftmost position of the Number

→ The convention is to make the sign bit equal to 0 for positive and 1 for negative

* In addition to the sign, a number May have a binary point (or) decimal point.

→ The position of the binary point is needed to Represent fractions, integers (or) mixed integer - fraction Numbers

→ The Representation of the binary point in a register is complecated by the fact that it is Characterized by the position in the Register.

* There are two ways of specifying the position of the binary point in a register
By giving it a
* (1) fixed position
(2) or by employing a Floating-point Representation.

* The fixed-point method assumes that the binary point is always fixed in one position.

The Two positions most widely used are:

(1) A Binary point in the Extreme left of the Register to make the Stored number a fraction, and

(2) A Binary point in the Extreme right of the Register to make the Stored number an integer.

In Either Case, the binary point is not actually present, but its presence is assumed from the fact that the Number Stored in the register is treated as a fraction (or) as an Integer

0000 0110

i's c 1111 1001

1

1111 1010

To find the Negative of the given Binary Number we use One's complement + 1.

* ~~Fixed~~ Integer ~~point~~ Representation

1) Signed Number's

* When an integer binary Number is positive, the sign is Represented by 0 and the magnitude by positive binary Number

* When the Number is Negative, the sign is Represented by 1, but rest of the Number may be represented in on of three possible ways:

1) Signed - magnitude Representation
2) Signed - 1's Complement representati
3) Signed - 2's Complement representatio

Ex

+14 is Represented by a sign bit of 0 in the left most position followed by the binary Equivalent of 14

(i.e) 0 0001110
↑ positive 14

(1) In Signed-magnitude representation = 1 0001110

(2) In Signed - 1's Complement " = 1 1110001

(3) In Signed - 2's Complement " = 1 1110010

→ In Signed-magnitude only sign is Complemented

→ In 1's Complement all bits are complemented

→ The 2's Complement is obtained by taking the 2's complement of the positive Number including its sign bit.

Arithmetic Addition

Ex The addition of two Numbers in the Signed-magnitude system follows the rules of ordinary arithmetic

→ This is a process that requires the Comparision of the sign's and the magnitude and then performing Either addition (or) Substraction.

① +6 0000 0110
+13 0000 1101
+19 0001 0011

② -6 1111 1010
+13 0000 1101
+7 0000 0111

③ +6 0000 0000
-13 1111 0011
-7 1111 1001

④ -6 1111 1010
-13 1111 0011
-19 1110 1101

In Each of the four Cases the operation performed is always addition including the sign bits. Any Carry out of the sign bit position is discarded, the Negative

results are automatically in 2's Complement form.

Arithmetic Subtraction

→ Subtraction of two signed binary numbers when negative numbers are in 2's Complement form is very simple and can be stated as:

1) Take the 2's complement of the subtrahend (including signed bit) and add it to the minuend (M)

2) A carryout of the sign bit position is discarded.

This is demonstrated by the following

(1) $(\pm A) - (+B) = (\pm A) + (-B)$

(2) $(\pm A) - (-B) = (\pm A) + (+B)$

Ex $(-6) - (-13) = \pm 7$ in Binary this is written as $11111010 - 11110011$

The sub is changed to addition by taking 2's Complement of (-13) as (+13)

* In binary this is 11111010 + 00001101

⇒ 100000111

Removing the End carry we obtain the Correct Answer 00000111 (+7)

Character Representation

→ An alphanumeric character set is a set of Elements that includes the decimal digits the 26 letters of the Alphabet and a Number of Special characters Such as (\$, +, and =.)

→ Such a set contains between 32 and 64 Elements (if only Upper case included) (or) between 64 and 128 (if both upper case and lower case letters are included

In the first case, the binary code will require six bits and in the second case seven bits

ASCII - American Standard Code for Information Interchange

Which uses seven bits to code 128 Charack

* Floating - point Representation

It has two parts.

(1) The first part represented a signed, fixed-point Number called the mantissa

(2) The second part designates the position of the decimal (or) Binary point and is called the Exponent.

(3) The fixed-point mantissa may be a fraction (or) an integer.

Ex +6132.789

The decimal Number is Represented in floating-point with a fraction and and Exponent as follow's:

Fraction	Exponent
+0.6132789	+04

→ Floating-point is always interpreted to Represent a Number in following form

$$m \times r^e$$

m = mantissa

r = radix point

e = Exponent.

---

* **floating point Representation in binary Number:**

→ It is same as the above but uses base 2 for the Exponent.

Ex +1001.11 is represented with an 8-bit fraction and 6-bit Exponent as follows

fraction	Exponent
01001110	000100

→ The fraction has a 0 in the leftmost position to denote positive.

→ The binary point of the fraction follows the sign bit but is not shown in the register

→ The Exponent has the Equivalent binary Number +4

Floating point Number $= m \times 2^e$

$m \times 2^e = +(.1001110)_2 \times 2^{+4}$

→ A floating point Number is said to be normalized if the most Significant Digit of the mantissa is non-zero.

Ex 350 is normalised // 000.35 is not

Regardless of radix point.

→ The 8-bit binary number 00011010 is not normalized because of three leading 0's

→ The Number Can be normalized by shifting it three positions to the left and discarding the leading 0's to obtain 11010000

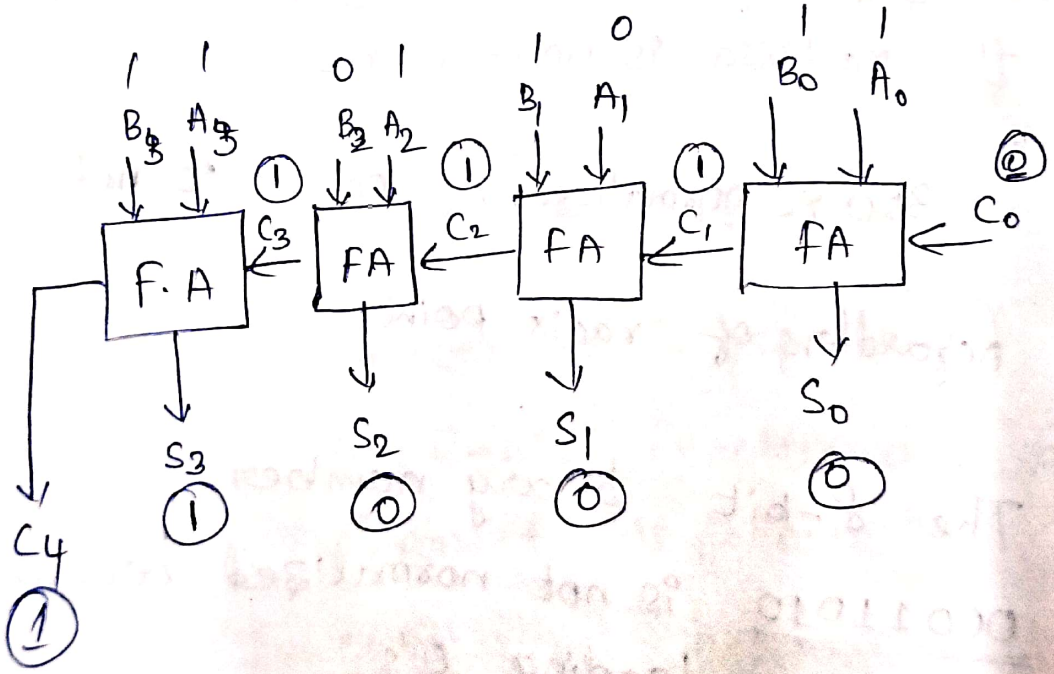
Ripple Carry Adder

(or)

Parallel Adder

Ripple Carry Adder is binary adder used to add two n-bits binary numbers.

	A_3	A_2	A_1	A_0	
A →	1	1	0	1	→ 13
B →	1	0	1	1	→ 11
	B_3	B_2	B_1	B_0	
	1	1	0	0	→ 24



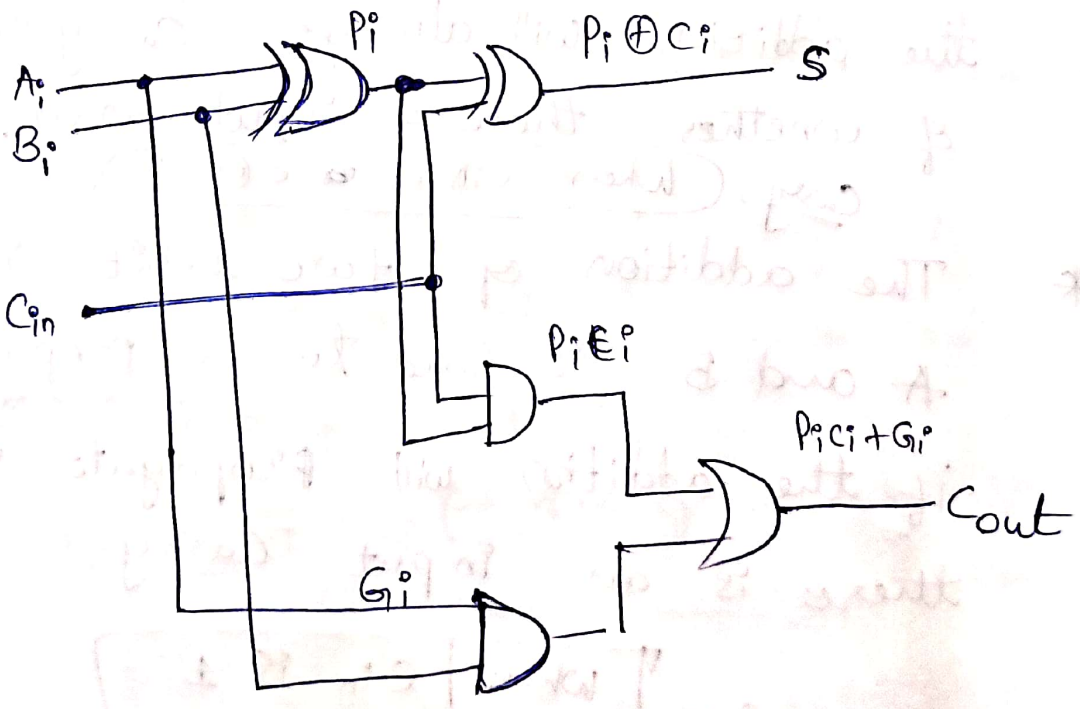
⇒ 11000

Propagation delay: If the full adder stage has a propagation delay of 20 Nanoseconds then it will take 60 ns

Carry Look Ahead Adder (CLA)

* A Carry look-ahead adder reduces the propagation delay by introducing more Complex hardware.

* In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bit of the adder is reduced to two level Logic.



$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

$$C_1 = G_0 + P_0 C_0 \;,\; C_2 = G_1 + P_1 C_1 \;,\; C_3 = G_2 + P_2 C_2$$

* The Carry look Ahead Adder Calculated one (or) more Carry bits before the Sum which reduces the wait time to Calculate the result of the larger value bits of the adder.

* CLA uses the Concept of Carry Propagator and Carry generator

* The addition of two 1-digit inputs A and B is said to generate if the addition will always Carry regardless of whether there is input Carry. (And Gate)
Carry (when Cin is 0 (or) 1)

* The addition of two 1-bit input A and B is said to be propagate if the addition will Propagate whenever there is an Input Carry. (XOR Gate)
Carry (when Cin is 1)

Ex =

A	B	cin	Cout	Type
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	P
1	0	0	0	
1	0	1	1	P
1	1	0	1	G
1	1	1	1	G

$$G_i = A_i B_i$$

$$P_i = C_i (A_i \oplus B_i)$$

$$C_{out} = G_i + P_i$$

$$\Rightarrow \boxed{A_i B_i + C_i (A_i \oplus B_i)}$$

$$C_{i+1} = A_i B_i + C_i (A_i \oplus B_i)$$

$$C_1 = A_0 B_0 + C_0 (A_0 \oplus B_0)$$

$$C_2 = A_1 B_1 + C_1 (A_1 \oplus B_1)$$

$$C_3 = A_2 B_2 + C_2 (A_2 \oplus B_2)$$

$$C_4 = A_3 B_3 + C_3 (A_3 \oplus B_3)$$

Booths Algorithm.

→ Booth's Algorithm is a multiplication Algorithm that allows us to multiply the two signed binary integers in 2's Complement representation.

→ It is also used to speed up the performance

→ It is Very Efficient Algorithm.

It operates on the fact that Strings of 0's in the multiplier require No addition but just shifting and the Strings of 1's in the multiplier from bit Weight 2^k to Weight 2^m Can be treated as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$

Ex: 14 ⇒ 01110 ⇒ (k=3) (m=1) ⇒ $2^{k+1} - 2^m$

$2^4 2^3 2^2 2^1 2^0$

$2^4 - 2^1$

$16 - 2 = 14$

∴ the multiplication M×14, where M is the multiplicand and 14 the multiplier

Can be done as $M \times 2^4 - M \times 2^1$

* The product can be obtained by shifting Binary multiplicand (i.e) M (4) times to left

And Substracting Binary multiplicand shifted Left once.

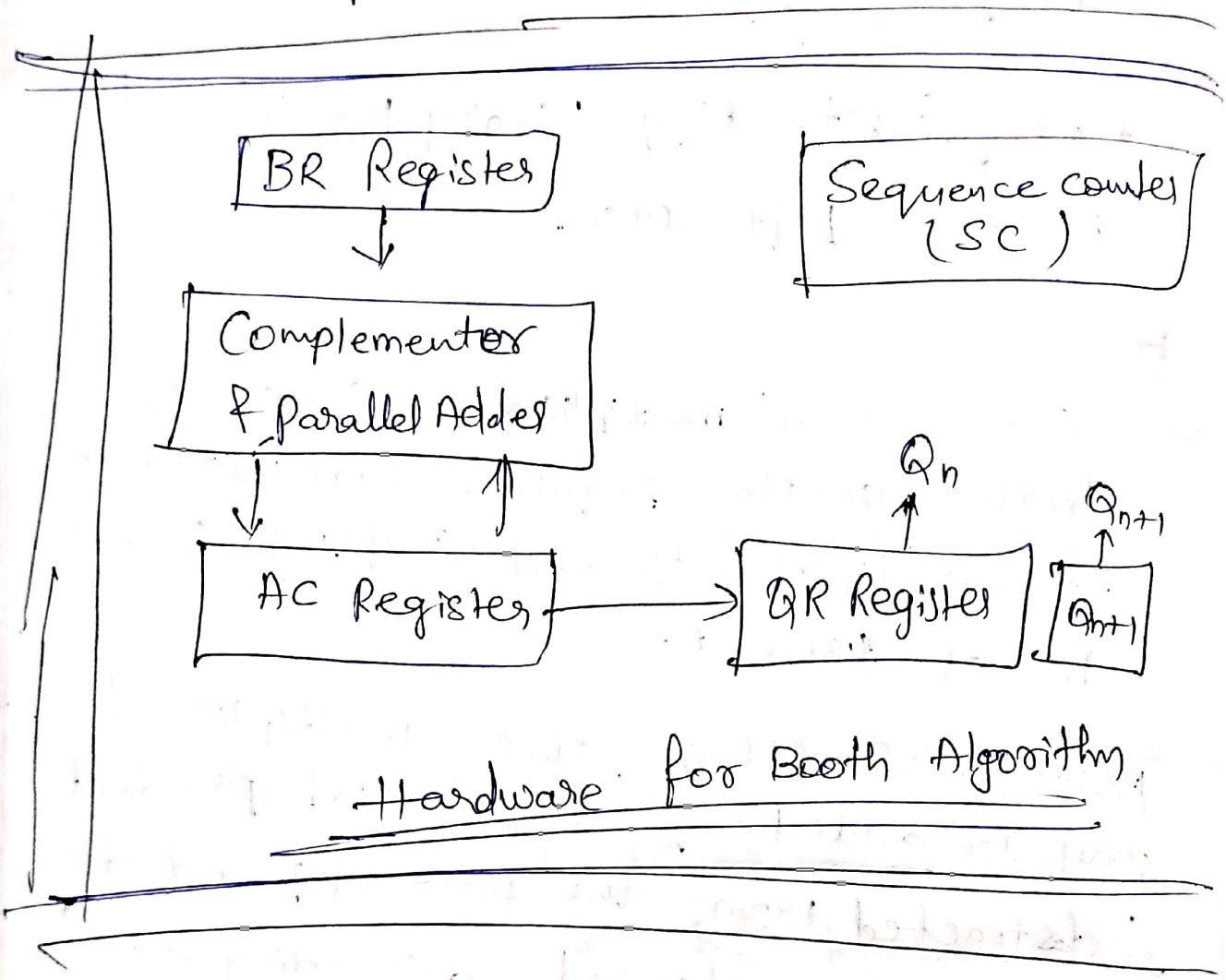
⇒ Same as all multiplication schemes, Booth Algorithm requires Examination of the multiplier bits and shifting of the Partial Product.

prior to shifting, the multiplicand may be added to the partial product, Substracted from the partial product, (or) left unchanged according to following rules.

(*) The multiplicand is substracted from the partial product upon Encountering the first least Significand 1 in a string of 1's in the multiplier.

(*) The multiplicand is added to the partial product upon Encountering the first 0 in the string of 0's in multiplier

(*) The partial product does not change when the multiplier bit is identical to the previous multiplier bit.



Hardware for Booth Algorithm

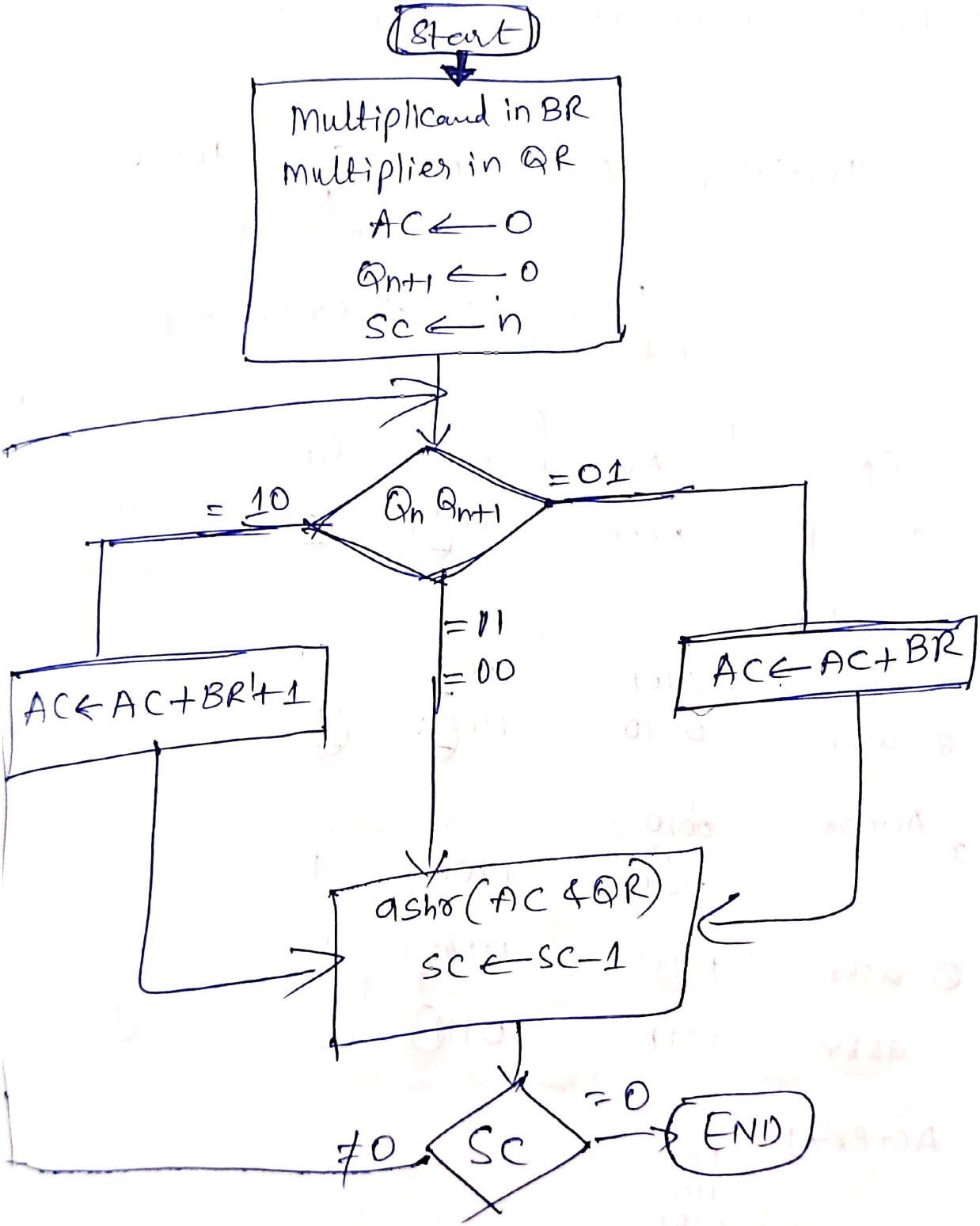
* Here, Sign bits are not seperated from rest of register.

* Register's A, B, and Q are renamed to AC, BR, QR Respectively

* Extra flipflop Q_{n+1} is appended to QR, it is needed to store almost lost right shifted bit to the multiplier.

Pair $Q_n Q_{n+1}$ inspect double bits of the multiplier.

Flowchart



Example

Compute -5×-7 using Booth's Algorithm

0101 => Multiplicand (-5) = 1011 (2's complement)

0111 => Multiplier (-7) = 1001 (11)

Initially AC = 0000, QR = 1001 (multiplier)

BR = 1011 (multiplicand) => 0101 (2's comp)

Operation	AC	QR	Q_{n+1}	SC
Initial	0000	1001	0	4
So ① AC+BR'+1				
↳	0101	1001	0	
② ashr	0010	1100	1	3
① AC+BR	0010 1011 1101	1100	1	
② ashr	1110	1110	0	2
ashr	1111	0111	0	1
AC+BR+1	1 1 1111 0101			
	0100	0111	0	
ashr	0010	0011	1	0

0010 0011 => 35 Ans

8 x 5 = 40

Multiplicand = 8 ⇒ BR
Multiplier = 5 ⇒ QR

0010
1000
0100

QR = 0101
BR = 1000

BR+1 = 0111
111
1000

1100
1000
0100

BR+1 = 1000

1101
1000
0101

Operation	AC	QR	Qn+1	SC
Initial	0000	0101	0	4
AC+BR+1	1000	0101	0	–
Ashr	1100	0010	1	3
AC+BR	0100	0010	1	
Ashr	0010	0001	0	2
AC+BR+1	1010	0001	0	
Ashr	1101	0000	1	1
AC+BR	0101	0000	1	
Ashr	0010	1000	0	0

0010 + 1000

32 + 8 ⇒ 40

Restoring Division Algorithm
(For unsigned integer)

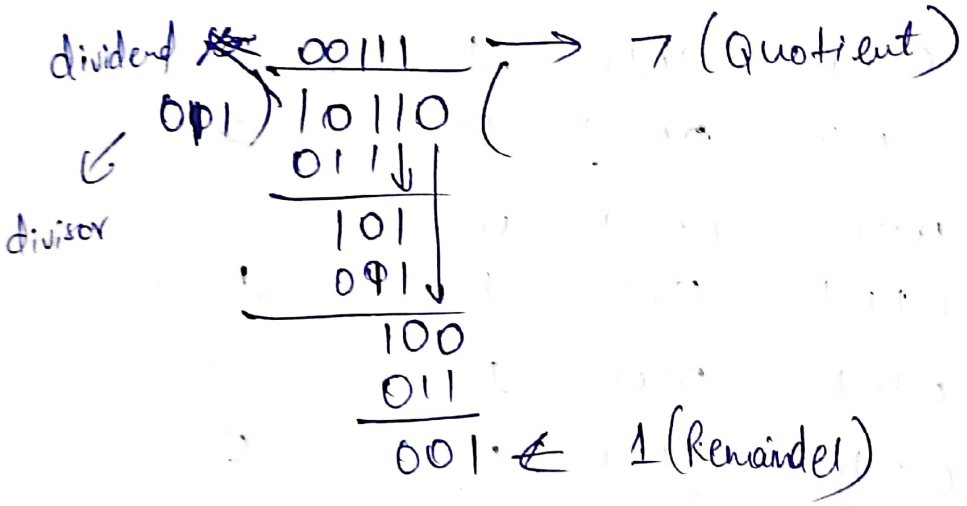
⇒ Example:

11 (Dividend) // 3 (Divisor) ⇒ 3 (Quotient) and 2 (Remainder)

⇒ Ex 22/3

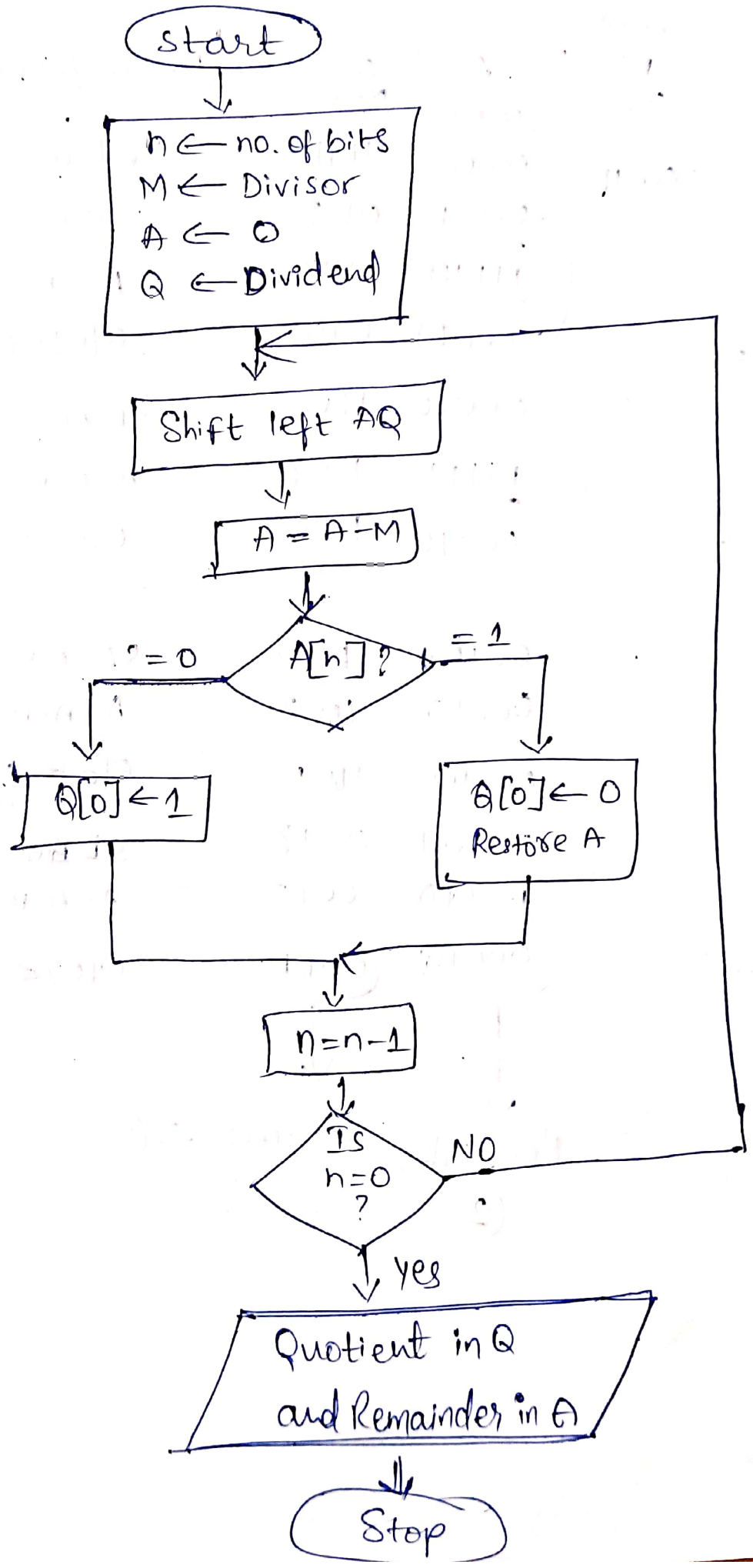
22 ⇒ 10110

3 ⇒ 011



General Technique

Restoring division Algorithm

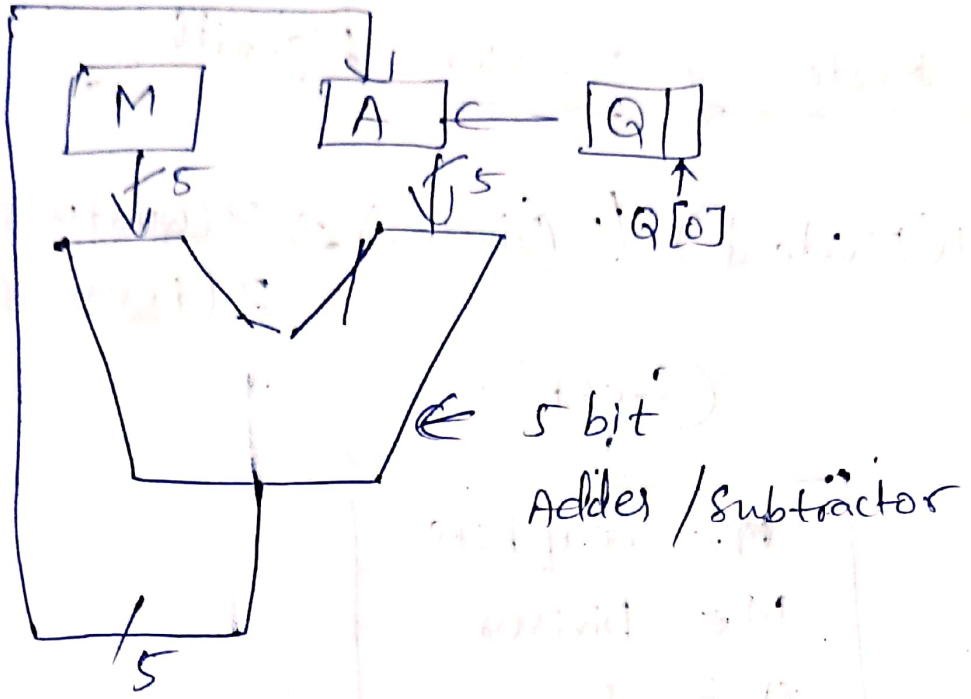


Tracing

11/3

n	M (Divisor) (n+1 bits)	A	Q (Dividend)	Action / operation
4 →	00011	00000	1011	Initialization
		00001	011?	Shift left AQ
		11110	011?	A = A − M
3 →		00001	0110	Q[0] ← 0 Restore A (Value of A is Restored)
		00010	110?	SL AQ
		11111	110?	A = A − M
2 →		00010	1100	Q[0] = 0
2		00100	100?	SL AQ
		00010	100?	A = A − M
1 →		00010	1001	Q[0] ← 1. No Restoring
		00101	001?	SL AQ
		00010	001?	A = A − M
0 →		00010	0011	Q[0] ← 1 No Restoring of 1

Remainder (2) — Quotient ⇒ (3)



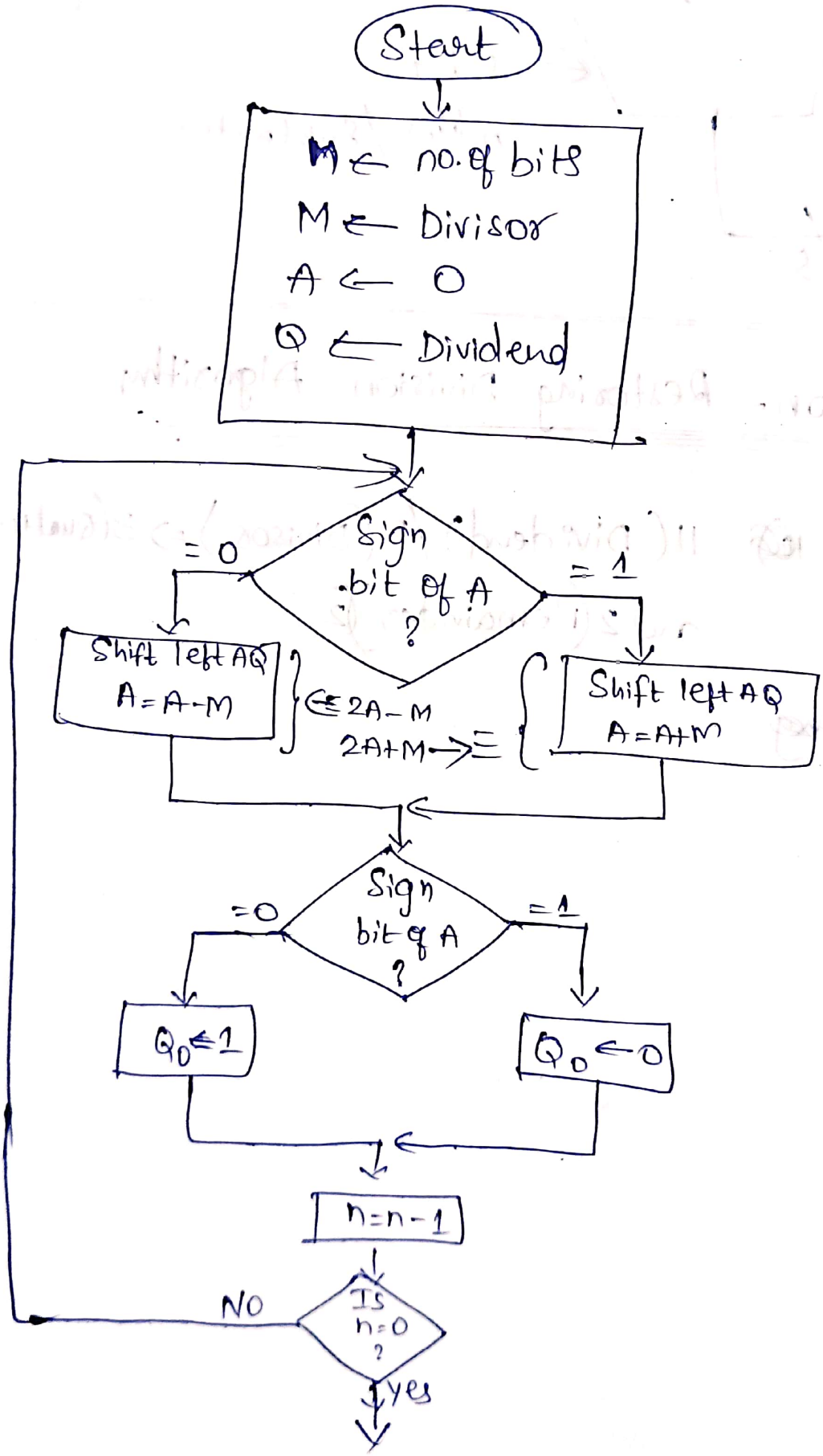
Non- Restoring Division Algorithm

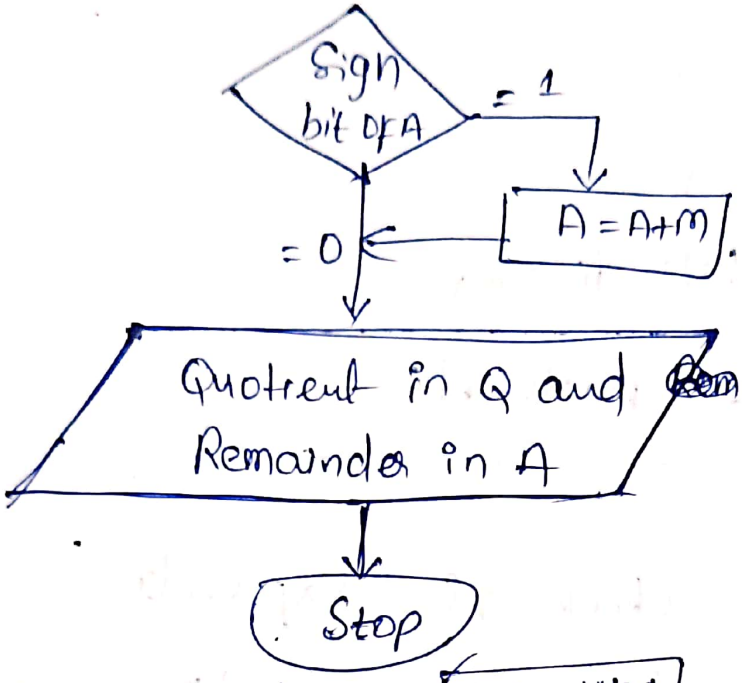
Ex (1) 11 (Dividend) / 3 (Divisor) ⇒ 3 (Quotient) and 2 (Remainder)2

Tracing

* Non-Restoring Division Algorithm

Ex 11 (Dividend) / 3 (Divisor) ⇒ 3 (Quotient)
2 (Remainder)





$$\begin{array}{r} 00011 \\ +\ 11101 \\ \hline 00000 \end{array}$$

$$\begin{array}{r} 00101 \\ +\ 11101 \\ \hline 00010 \end{array}$$

Tracing

$-M = 11101$

n	M (n+1) bits	A	Q	Action/operation
4	00011	00000	1011	Initialization
		00001	011?	Shift left AQ
		+ 11101		→ A = A−M
		11110	011?	
3		11110	0110	$Q_0 \leftarrow 0$
		11100	110?	SL AQ
		+ 11111	110?	A ← A+M
2		11111	1100	$Q_0 \leftarrow 0$
		11111	100?	SL AQ
		+ 00010	100?	A = A+M
1		00010	1001	$Q_0 \leftarrow 1$
		00101	001?	SL AQ
		+ 00010	001?	A = A−M
0		00010	0011	$Q_0 \leftarrow 1$

Remainder (2)

Quotient (3)

Q Non - Restoring

Restoring

Unsigned bits division

floating point Arithmetic

→ We have to check the Exponents

→ Have to allign the mantissa

→ Normalize the Numbers (if Not in Normalized form)

→ Calculate the Arithematic Operation