Chapter – 5
Computer Arithmetic

Integer Representation: (Fixed-point representation):

An eight bit word can be represented the numbers from zero to 255 including
00000000 = 0
00000001 = 1
- - - - - - -
11111111 = 255

In general if an n-bit sequence of binary digits a_{n-1}, a_{n-2} …..a_1, a_0; is interpreted as unsigned integer A.

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Sign magnitude representation:

There are several alternative convention used to represent negative as well as positive integers, all of which involves treating the most significant (left most) bit in the word as sign bit. If the sign bit is 0, the number is +ve and if the sign bit is 1, the number is –ve. In n bit word the right most n-1 bit hold the magnitude of integer.

For an example,
+18 = 00010010
- 18 = 10010010 (sign magnitude)

The general case can be expressed as follows:

$$A = \sum_{i=0}^{n-2} 2^i a_i \quad \text{if } a_{n-1} = 0$$

$$= -\sum_{i=0}^{n-2} 2^i a_i \quad \text{if } a_{n-1} = 1$$

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad \text{(Both for +ve and –ve)}$$

There are several drawbacks to sign-magnitude representation. One is that addition or subtraction requires consideration of both signs of number and their relative magnitude to carry out the required operation. Another drawback is that there are two representation of zero. For an example:

$+0_{10}$ = 00000000
-0_{10} = 10000000 this is inconvenient.

2's complement representation:

Like sign magnitude representation, 2's complement representation uses the most significant bit as sign bit making it easy to test whether the integer is negative or positive. It differs from the use of sign magnitude representation in the way that the other bits are interpreted. For negation, take the Boolean complement (1's complement) of each bit of corresponding positive number, and then add one to the resulting bit pattern viewed as unsigned integer. Consider n bit integer A in 2's complement representation. If A is +ve then the sign bit a_{n-1} is zero. The remaining bits represent the magnitude of the number.

$A = \sum_{i=0}^{n-2} 2^i a_i \quad \text{for } A \geq 0$

The number zero is identified as +ve and therefore has zero sign bit and magnitude of all 0's. We can see that the range of +ve integer that may be represented is from 0 (all the magnitude bits are zero) through $2^{n-1}-1$ (all of the magnitude bits are 1).

Now for –ve number integer A, the sign bit a_{n-1} is 1. The range of –ve integer that can be represented is from -1 to -2^{n-1}

2's complement, $A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$

Defines the twos complement of representation of both positive and negative number.

For an example:

+18 = 00010010

1's complement = 11101101

2's complement = 11101110 = -18

5.1 Addition Algorithm

5.2 Subtraction Algorithm

1001 = -7	1100 = -4	0011 = 3
0101 = +5	0100 = +4	0100= 4
1110 =-2	10000 = 0	0111= 7
(a) (-7)+(+5)	(b) (-4)+(4)	(c) (+3)+(+4)
1100 = -4	0101 =5	1001 = -7
1111 = -1	0100 =4	1010 = -6
11011 = -5	1001=overflow	10011 = overflow
(d) (-4)+(-1)	(e) (+5)+(+4)	(f) (-7)+(-6)

The first four examples illustrate successful operation if the result of the operation is +ve then we get +ve number in ordinary binary notation. If the result of the operation is –ve we get negative number in twos complement form. Note that in some instants there is carry bit beyond the end of what which is ignored. On any addition the result may larger then can be held in word size being use. This condition is called over flow. When overflow occur ALU must signal this fact so that no attempt is made to use the result. To detect overflow the following rule observed. If two numbers are added, and they are both +ve or both –ve; then overflow occurs if and only if the result has the opposite sign.

The data path and hardware elements needed to accomplish addition and subtraction is shown in figure below. The central element is binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. For addition, the two numbers are presented to the adder from two registers A and B. The result may be stored in one of these registers or in the third. The overflow indication is stored in a 1-bit overflow flag V (where 1 = overflow and 0 = no overflow). For subtraction, the subtrahend (B register) is passed through a 2's complement unit so that its 2's complement is presented to the adder (a – b = a + (-b)).

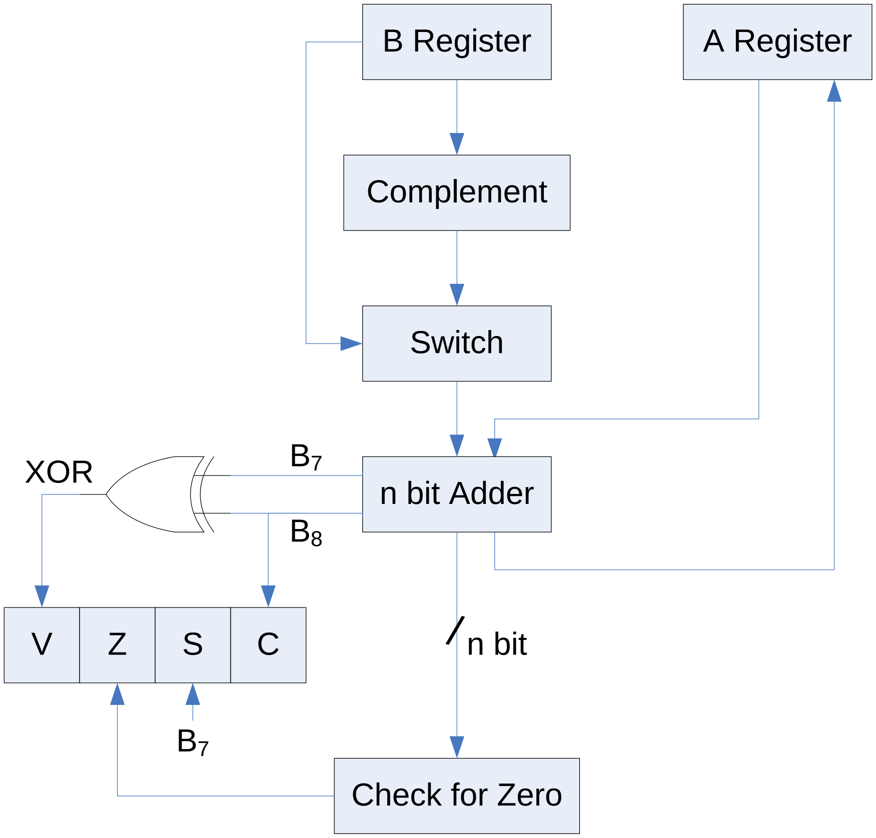


Fig: Block diagram of hardware for addition / subtraction

5.3 Multiplication Algorithm

The multiplier and multiplicand bits are loaded into two registers Q and M. A third register A is initially set to zero. C is the 1-bit register which holds the carry bit resulting from addition. Now, the control logic reads the bits of the multiplier one at a time. If Q_0 is 1, the multiplicand is added to the register A and is stored back in register A with C bit used for carry. Then all the bits of CAQ are shifted to the right 1 bit so that C bit goes to A_{n-1}, A_0 goes to Q_{n-1} and Q_0 is lost. If Q_0 is 0, no addition is performed just do the shift. The process is repeated for each bit of the original multiplier. The resulting 2n bit product is contained in the QA register.

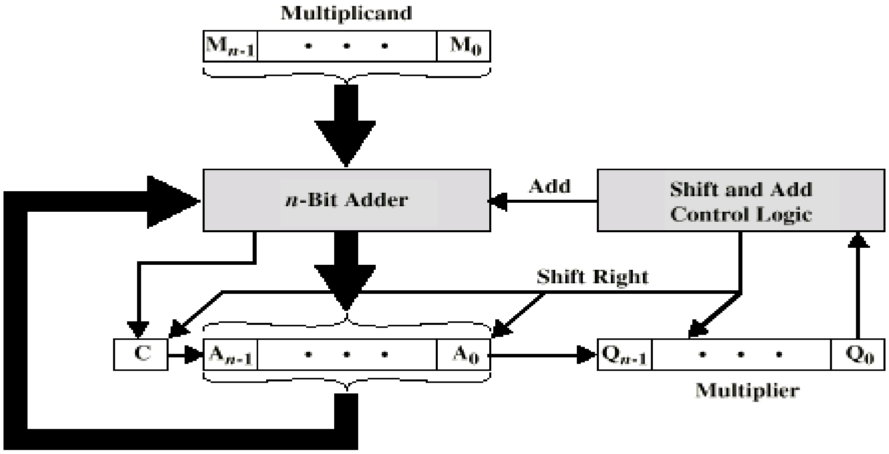


Fig: Block diagram of multiplication

There are three types of operation for multiplication.

- It should be determined whether a multiplier bit is 1 or 0 so that it can designate the partial product. If the multiplier bit is 0, the partial product is zero; if the multiplier bit is 1, the multiplicand is partial product.
- It should shift partial product.
- It should add partial product.

Unsigned Binary Multiplication

```
      1011    Multiplicand 11
    x 1101    Multiplier 13
      1011
     0000           Partial Product
    1011
  + 1011
  10001111    Product (143)
```

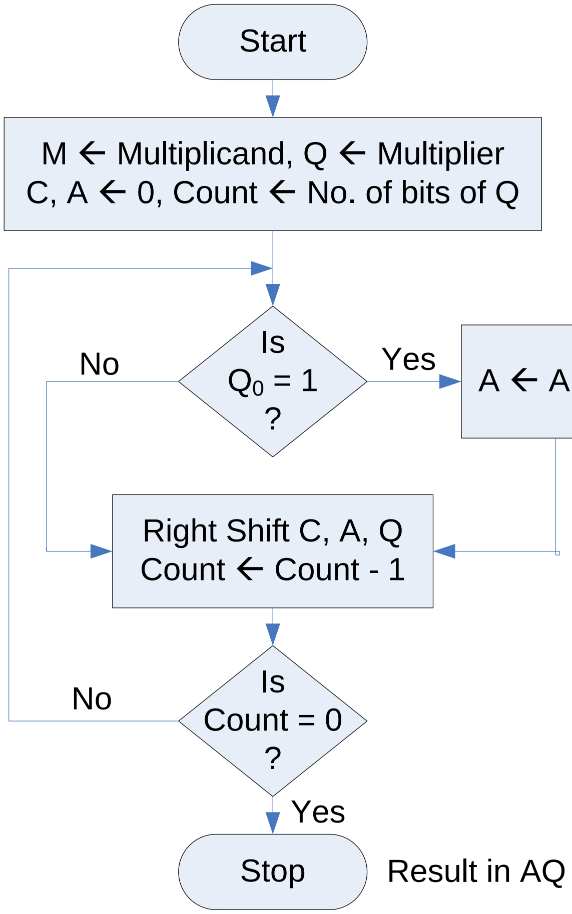


Fig. : Flowchart of Unsigned Binary Multiplication

Example: Multiply 15 x 11 using unsigned binary method

C	A	Q	M	Count	Remarks
0	0000	1011	1111	4	Initialization
0 0	1111 0111	1011 1101	- -	- 3	Add (A ← A + M) Logical Right Shift C, A, Q
1 0	0110 1011	1101 0110	- -	- 2	Add (A ← A + M) Logical Right Shift C, A, Q
0	0101	1011	-	1	Logical Right Shift C, A, Q
1 0	0100 1010	1011 0101	- -	- 0	Add (A ← A + M) Logical Right Shift C, A, Q

Result = 1010 0101 = $2^7 + 2^5 + 2^2 + 2^0 = 165$

Alternate Method of Unsigned Binary Multiplication

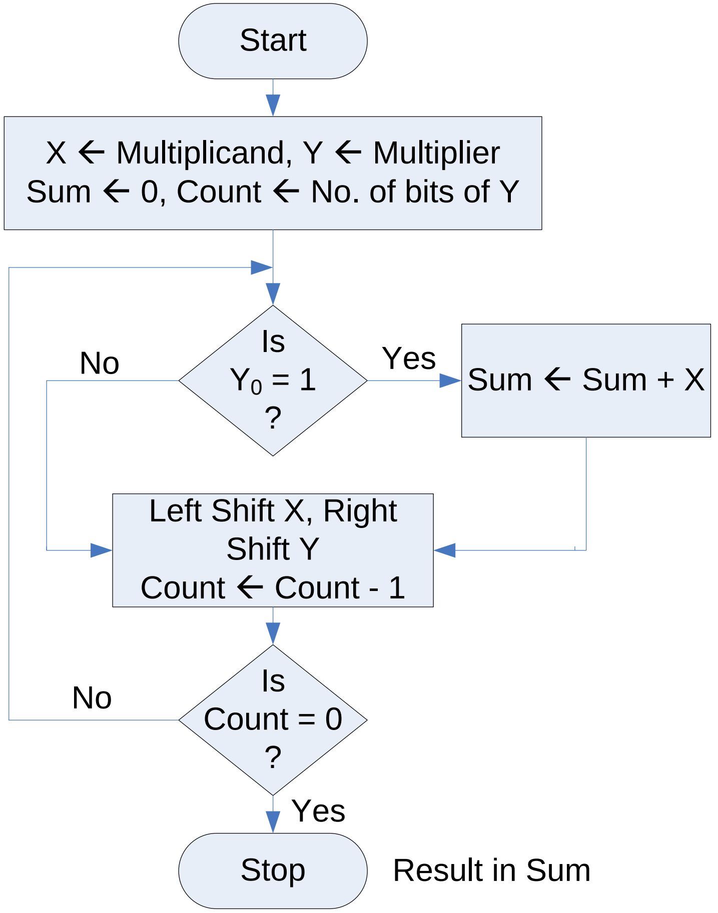

Start
X ← Multiplicand, Y ← Multiplier
Sum ← 0, Count ← No. of bits of Y
Is
Y₀ = 1
?
No
Yes
Sum ← Sum + X
Left Shift X, Right Shift Y
Count ← Count - 1
Is
Count = 0
?
No
Yes
Stop
Result in Sum

Fig: Unsigned Binary Multiplication Alternate method

Algorithm:

Step 1: Clear the sum (accumulator A). Place the multiplicand in X and multiplier in Y.
Step 2: Test $Y_{0;}$ if it is 1, add content of X to the accumulator A.
Step 3: Logical Shift the content of X left one position and content of Y right one position.
Step 4: Check for completion; if not completed, go to step 2.

Example: Multiply 7 X 6

Sum	X	Y	Count	Remarks
000000	000111	110	3	Initialization
000000	001110	011	2	Left shift X, Right Shift Y
001110	011100	001	1	Sum ← Sum + X, Left shift X, Right Shift Y
101010	111000	000	0	Sum ← Sum + X, Left shift X, Right Shift Y

Result = 101010 = $2^5 + 2^3 + 2^1 = 42$

Signed Multiplication (Booth Algorithm) – 2's Complement Multiplication

Multiplier and multiplicand are placed in Q and M register respectively. There is also one bit register placed logically to the right of the least significant bit Q_0 of the Q register and designated as Q_{-1}. The result of multiplication will appear in A and Q resister. A and Q_{-1} are initialized to zero if two bits (Q_0 and Q_{-1}) are the same (11 or 00) then all the bits of A, Q and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ then the multiplicand is added to or subtracted from the A register depending on weather the two bits are 01 or 10. Following the addition or subtraction the arithmetic right shift occurs. When count reaches to zero, result resides into AQ in the form of signed integer [$-2^{n-1}*a_{n-1} + 2^{n-2}*a_{n-2} +$ …………… $+ 2^1*a_1 + 2^0*a_0$].

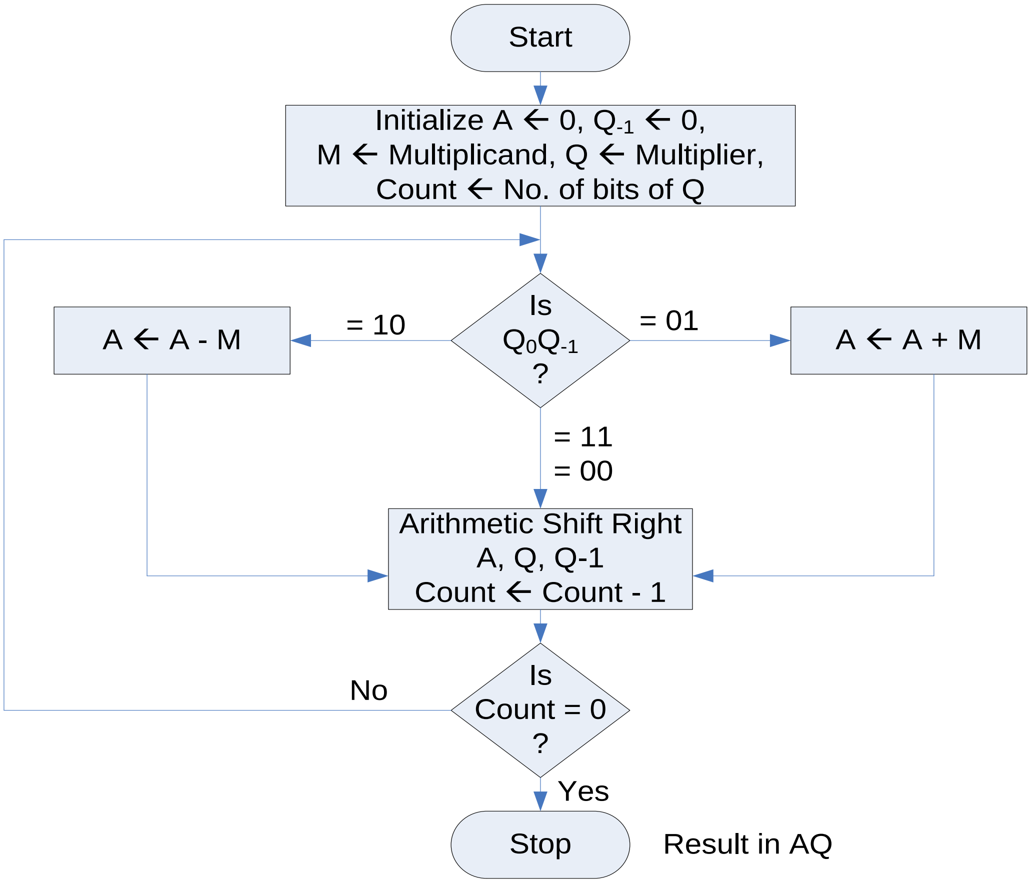


Fig.: Flowchart of Signed Binary Numbers (using 2's Complement, Booth Method)

Example: Multiply 9 x -3 = -27 using Booth Algorithm
+3 = 00011, -3 = 11101 (2's complement of +3)

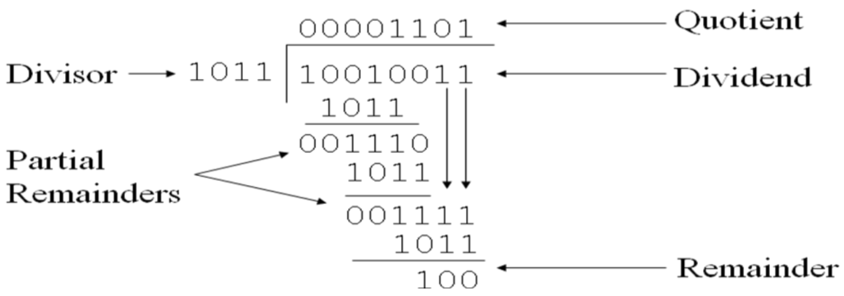
A	Q	Q_{-1}	Add (M)	Sub ($\overline{M}$ +1)	Count	Remarks
00000	11101	0	01001	10111	5	Initialization
10111 11011	11101 11110	0 1	- -	- -	- 4	Sub (A ← A - M) as Q_0Q_{-1} = 10 Arithmetic Shift Right A, Q, Q_{-1}
00100 00010	11110 01111	1 0	- -	- -	- 3	Add (A ← A + M) as Q_0Q_{-1} = 01 Arithmetic Shift Right A, Q, Q_{-1}
11001 11100	01111 10111	0 1	- -	- -	- 2	Sub (A ← A - M) as Q_0Q_{-1} = 10 Arithmetic Shift Right A, Q, Q_{-1}
11110	01011	1	-	-	1	Arithmetic Shift Right A, Q, Q_{-1} as Q_0Q_{-1} = 11
11111	00101	1	-	-	0	Arithmetic Shift Right A, Q, Q_{-1} as Q_0Q_{-1} = 11

Result in AQ = 11111 00101 = $-2^9+2^8+2^7+2^6+2^5+2^2+2^0$ = -512+256+128+64+32+4+1 = -27

5.4 Division Algorithm

Division is somewhat more than multiplication but is based on the same general principles. The operation involves repetitive shifting and addition or subtraction.

First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder*. The division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. The divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.



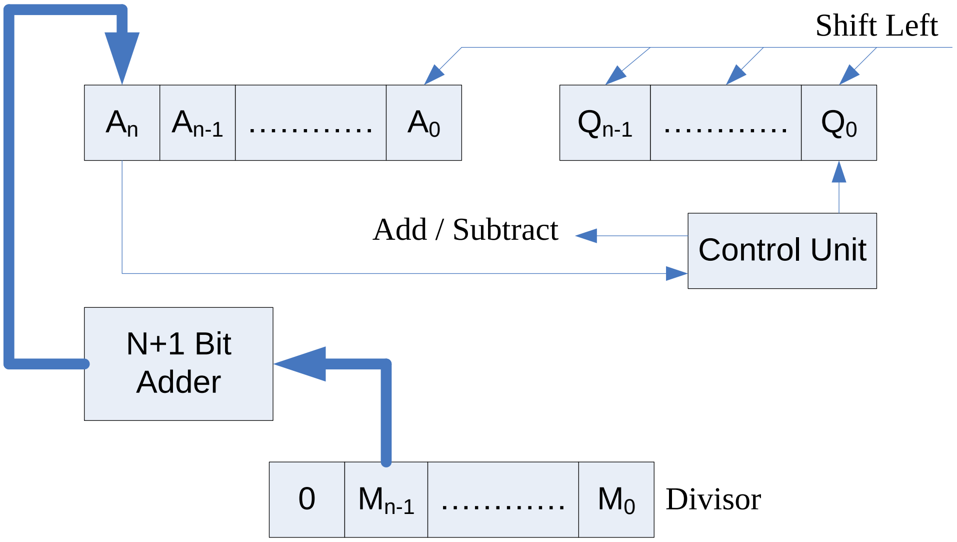
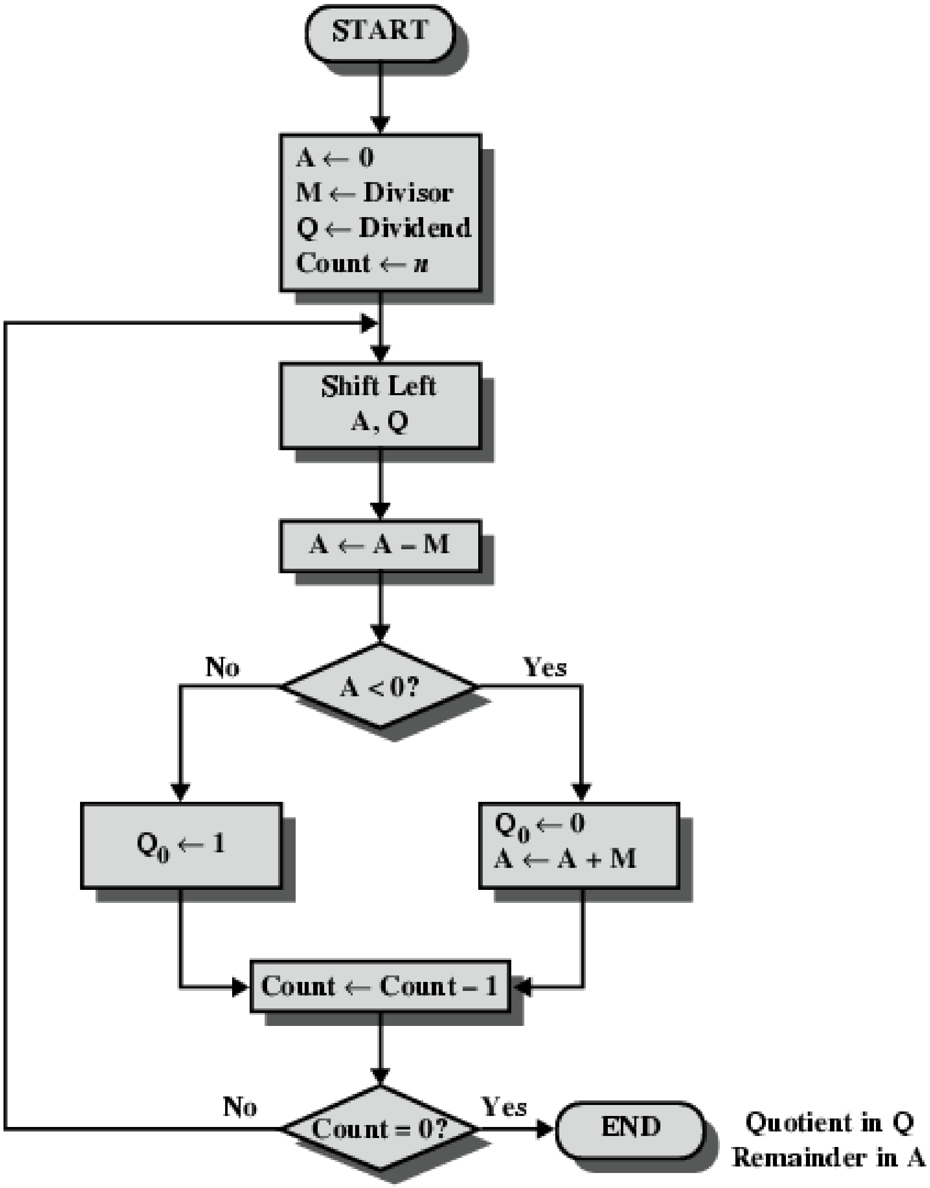


Fig.: Block Diagram of Division Operation

Restoring Division (Unsigned Binary Division)



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_0 to zero and add M back to A (restore A). If sign of A is 0, set Q_0 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.

Example: Divide 15 (1111) by 4 (0100)

A	Q	M	$\overline{M}$ +1	Count	Remarks
00000	1111	00100	11100	4	Initialization
00001 11101 00001	111□ 111□ 1110	- - -	- - -	- - 3	Shift Left A, Q Sub (A ← A – M) Q_0 ← 0, Add (A← A + M)
00011 11111 00011	110□ 110□ 1100	- - -	- - -	- - 2	Shift Left A, Q Sub (A ← A – M) Q_0 ← 0, Add (A← A + M)
00111 00011 00011	100□ 100□ 1001	- - -	- - -	- - 1	Shift Left A, Q Sub (A ← A – M) Set Q_0 ← 1
00111 00011 00011	001□ 001□ 0011	- - -	- - -	- - 0	Shift Left A, Q Sub (A ← A – M) Set Q_0 ← 1

Quotient in Q = 0011 = 3
Remainder in A = 00011 = 3

Non – Restoring Division (Signed Binary Division)
Algorithm

Step 1: Initialize A, Q and M registers to zero, dividend and divisor respectively and count to number of bits in dividend.

Step 2: Check sign of A;

If A < 0 i.e. b_{n-1} is 1

a. Shift A, Q left one binary position.
b. Add content of M to A and store back in A.

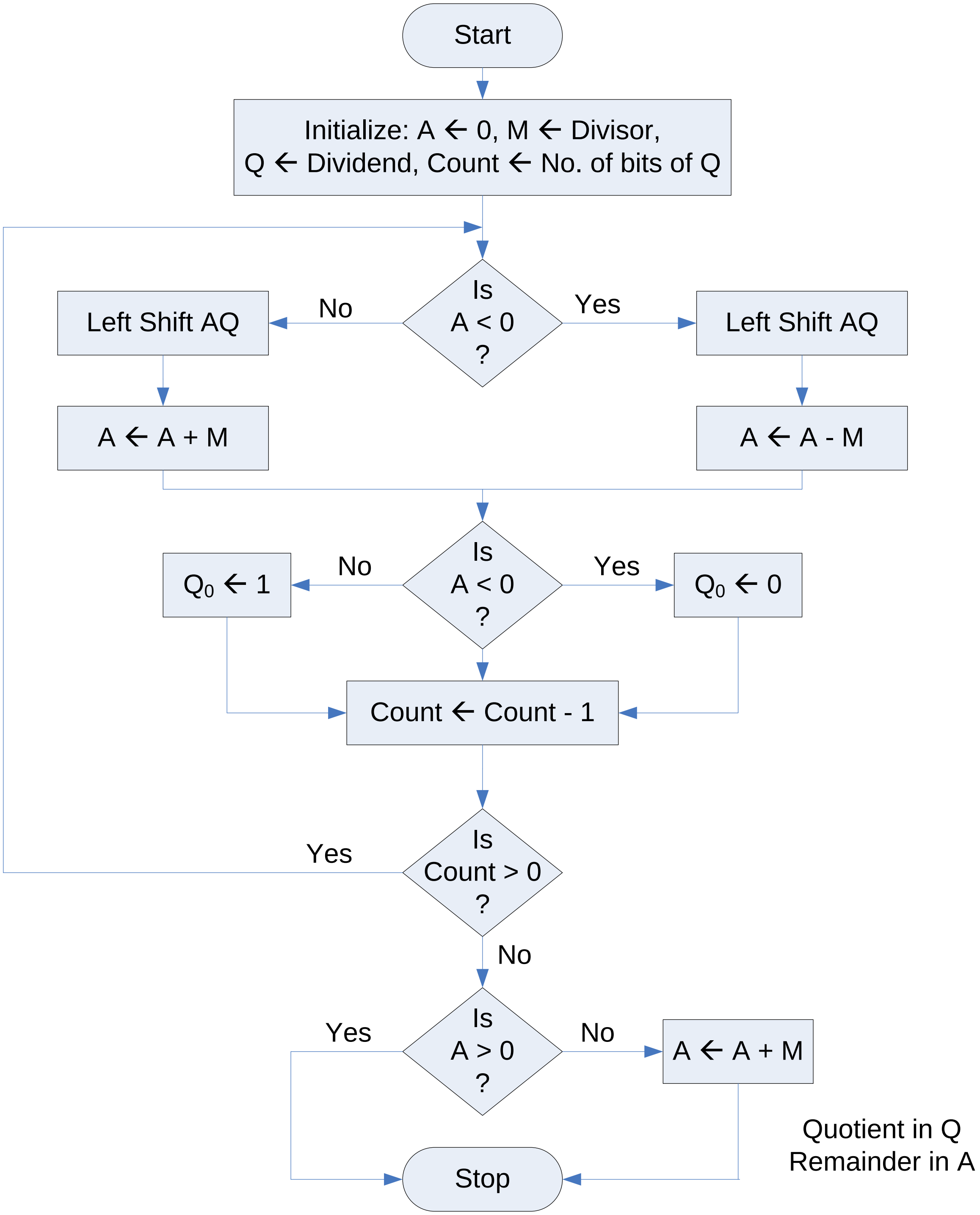
If A ≥ 0 i.e. b_{n-1} is 0

a. Shift A, Q left one binary position.
b. Subtract content of M to A and store back in A.

Step 3: If sign of A is 0, set Q_0 to 1 else set Q_0 to 0.

Step 4: Decrease counter. If counter > 0, repeat process from step 2 else go to step 5.

Step 5: If A ≥ 0 i.e. positive, content of A is remainder else add content of M to A to get the remainder. The quotient will be in Q.


Start
Initialize: A ← 0, M ← Divisor,
Q ← Dividend, Count ← No. of bits of Q
Is
A < 0
?
No
Yes
Left Shift AQ
Left Shift AQ
A ← A + M
A ← A - M
Is
A < 0
?
No
Yes
Q0 ← 1
Q0 ← 0
Count ← Count - 1
Is
Count > 0
?
Yes
No
Is
A > 0
?
Yes
No
A ← A + M
Stop
Quotient in Q
Remainder in A

Example: Divide 1110 (14) by 0011 (3) using non-restoring division.

A	Q	M	$\overline{M}$ +1	Count	Remarks
00000	1110	00011	11101	4	Initialization
00001	110□	-	-	-	Shift Left A, Q
11110	110□	-	-	-	Sub (A ← A – M)
11110	1100	-	-	3	Set Q_0 to 0
11101	100□	-	-	-	Shift Left A, Q
00000	100□	-	-	-	Add (A ← A + M)
00000	1001	-	-	2	Set Q_0 to 1
00001	001□	-	-	-	Shift Left A, Q
11110	001□	-	-	-	Sub (A ← A – M)
11110	0010	-	-	1	Set Q_0 to 0
11100	010□	-	-	-	Shift Left A, Q
11111	010□	-	-	-	Add (A ← A + M)
11111	0100	-	-	0	Set Q_0 to 0
00010	0100	-	-	-	Add (A ← A + M)

Quotient in Q = 0011 = 3
Remainder in A = 00010 = 2

Floating Point Representation

The floating point representation of the number has two parts. The first part represents a signed fixed point numbers called mantissa or significand. The second part designates the position of the decimal (or binary) point and is called exponent. For example, the decimal no + 6132.789 is represented in floating point with fraction and exponent as follows.

Fraction	Exponent
+0.6132789	+04

This representation is equivalent to the scientific notation $+0.6132789 \times 10^{+4}$

The floating point is always interpreted to represent a number in the following form $\pm M \times R^{\pm E}$. Only the mantissa M and the exponent E are physically represented in the register (including their sign). The radix R and the radix point position of the mantissa are always assumed.

A floating point binary no is represented in similar manner except that it uses base 2 for the exponent.

For example, the binary no +1001.11 is represented with 8 bit fraction and 0 bit exponent as follows.

0.1001110×2^{100}

Fraction	Exponent
01001110	000100

The fraction has zero in the leftmost position to denote positive. The floating point number is equivalent to $M \times 2^E = +(0.1001110)_2 \times 2^{+4}$

Floating Point Arithmetic

The basic operations for floating point arithmetic are

Floating point number	*Arithmetic Operations*
$X = Xs \times B^{XE}$	$X + Y = (Xs \times B^{XE-YE} + Ys) \times B^{YE}$
$Y = Ys \times B^{YE}$	$X - Y = (Xs \times B^{XE-YE} - Ys) \times B^{YE}$
	$X * Y = (Xs \times Ys) \times B^{XE+YE}$
	$X / Y = (Xs / Ys) \times B^{XE-YE}$

There are four basic operations for floating point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent values. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are straighter forward.

A floating point operation may produce one of these conditions:

- Exponent Overflow: A positive exponent exceeds the maximum possible exponent value.
- Exponent Underflow: A negative exponent which is less than the minimum possible value.
- Significand Overflow: The addition of two significands of the same sign may carry in a carry out of the most significant bit.
- Significand underflow: In the process of aligning significands, digits may flow off the right end of the significand.

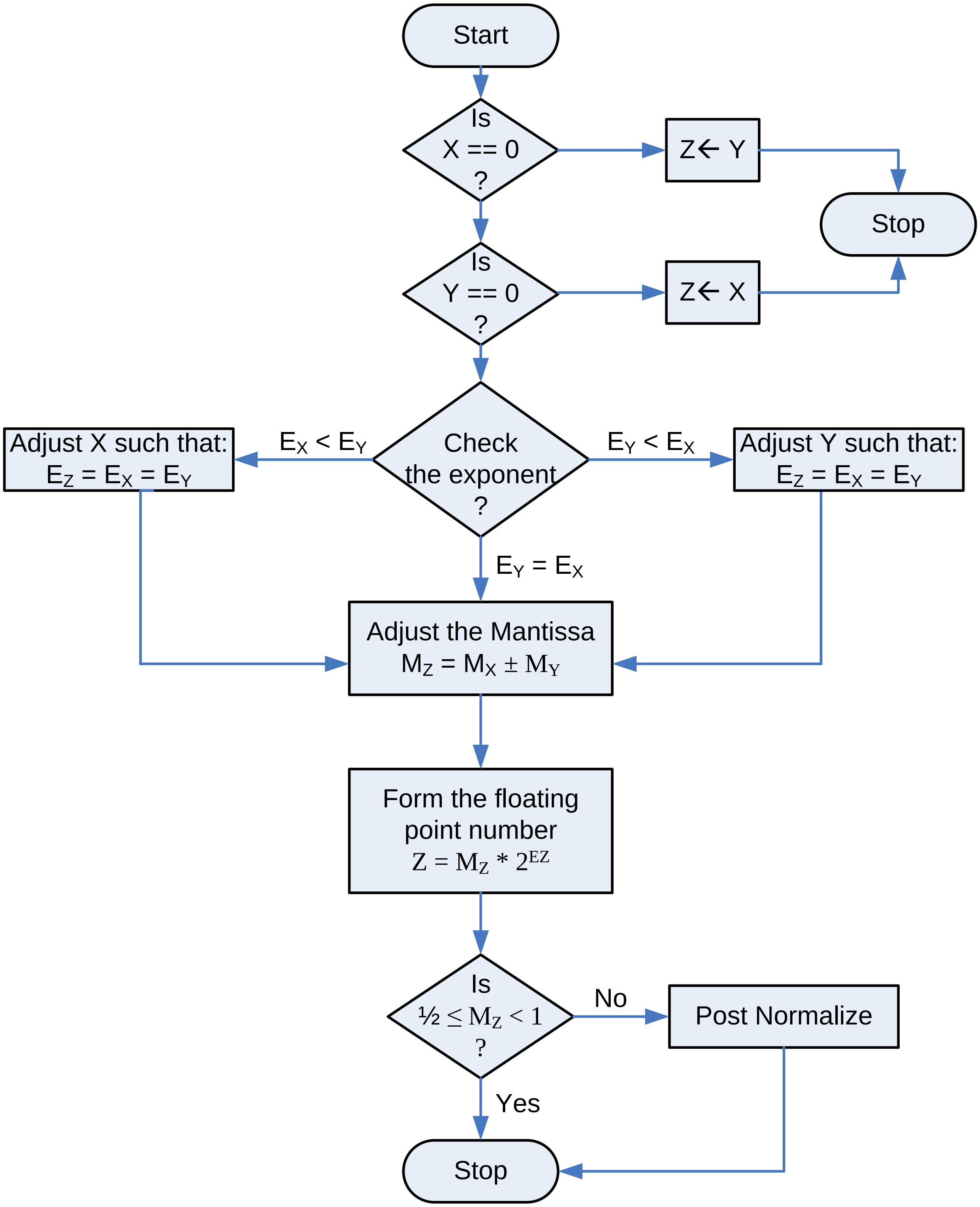
Floating Point Addition and Subtraction

In floating point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment. There are four phases for the algorithm for floating point addition and subtraction.

1. Check for zeros:
   Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtraction operation. Next; if one is zero, second is result.
2. Align the Significands:
   Alignment may be achieved by shifting either the smaller number to the right (increasing exponent) or shifting the large number to the left (decreasing exponent).
3. Addition or subtraction of the significands:
   The aligned significands are then operated as required.
4. Normalization of the result:
   Normalization consists of shifting significand digits left until the most significant bit is nonzero.

Example: Addition
$X = 0.10001 * 2^{110}$
$Y = 0.101 * 2^{100}$
Since $E_Y < E_X$, Adjust Y
$Y = 0.00101 * 2^{100} * 2^{010} = 0.00101 * 2^{110}$
So, $E_Z = E_X = E_Y = 110$
Now, $M_Z = M_X + M_Y = 0.10001 + 0.00101 = 0.10110$
Hence, $Z = M_Z * 2^{EZ} = 0.10110 * 2^{110}$

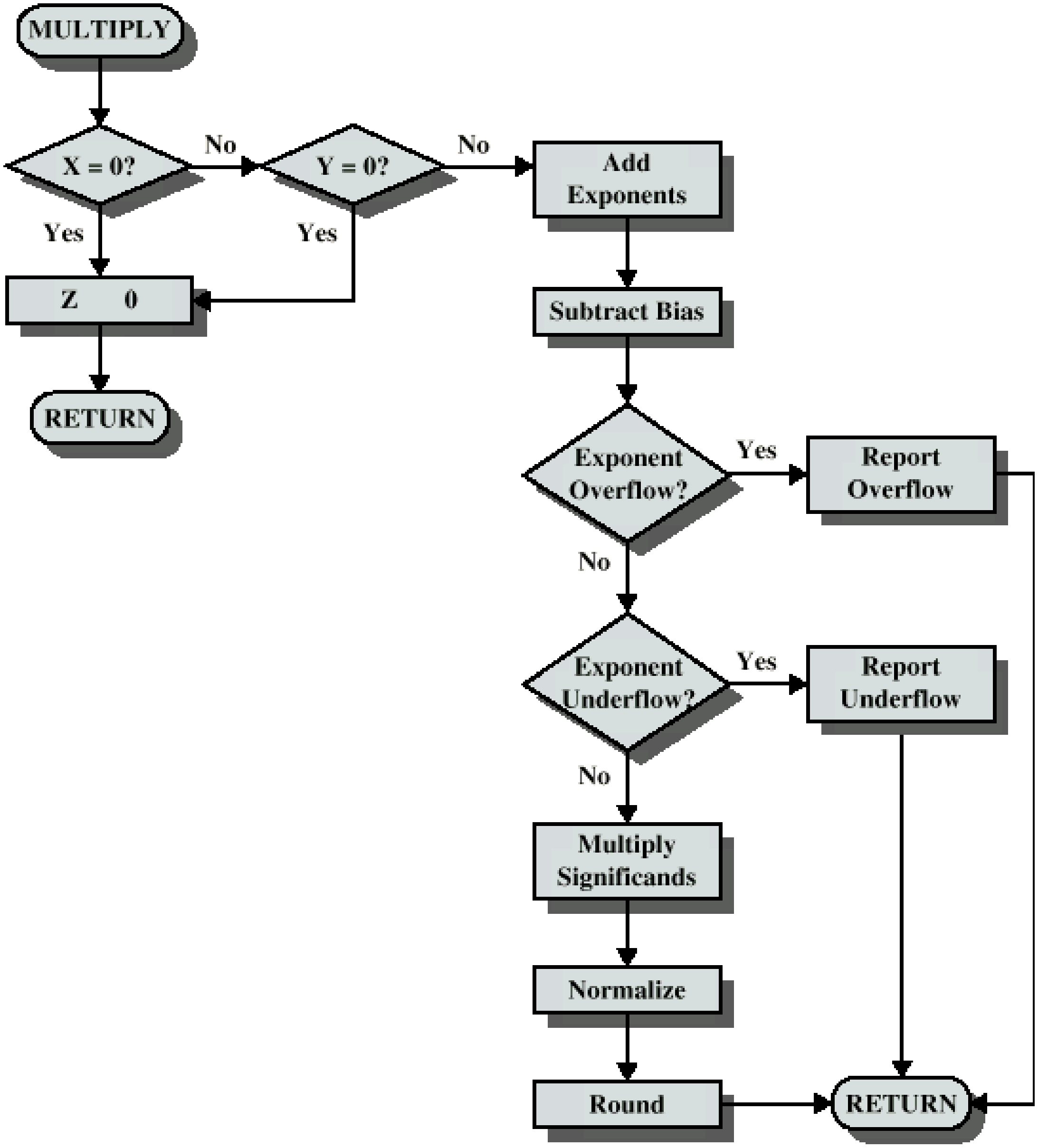
Example: Subtraction
$X = 0.10001 * 2^{110}$
$Y = 0.101 * 2^{100}$
Since $E_Y < E_X$, Adjust Y
$Y = 0.00101 * 2^{100} * 2^{010} = 0.00101 * 2^{110}$
So, $E_Z = E_X = E_Y = 110$
Now, $M_Z = M_X - M_Y = 0.10001 - 0.00101 = 0.01100$
$Z = M_Z * 2^{EZ} = 0.01100 * 2^{110}$ (Un-Normalized)
Hence, $Z = 0.1100 * 2^{110} * 2^{-001} = 0.1100 * 2^{101}$


Start
Is
X == 0
?
Z← Y
Stop
Is
Y == 0
?
Z← X
$E_X < E_Y$
Adjust X such that:
$E_Z = E_X = E_Y$
Check
the exponent
?
$E_Y < E_X$
Adjust Y such that:
$E_Z = E_X = E_Y$
$E_Y = E_X$
Adjust the Mantissa
$M_Z = M_X \pm M_Y$
Form the floating
point number
$Z = M_Z * 2^{EZ}$
Is
$\frac{1}{2} \leq M_Z < 1$
?
No
Post Normalize
Yes
Stop

Floating Point Multiplication

The multiplication can be subdivided into 4 parts.

1. Check for zeroes.
2. Add the exponents.
3. Multiply mantissa.
4. Normalize the product.



Example:

$X = 0.101 * 2^{110}$

$Y = 0.1001 * 2^{-010}$

As we know, $Z = X * Y = (M_X * M_Y) * 2^{(EX + EY)}$

$Z = (0.101 * 0.1001) * 2^{(110-010)}$

$= 0.0101101 * 2^{100}$

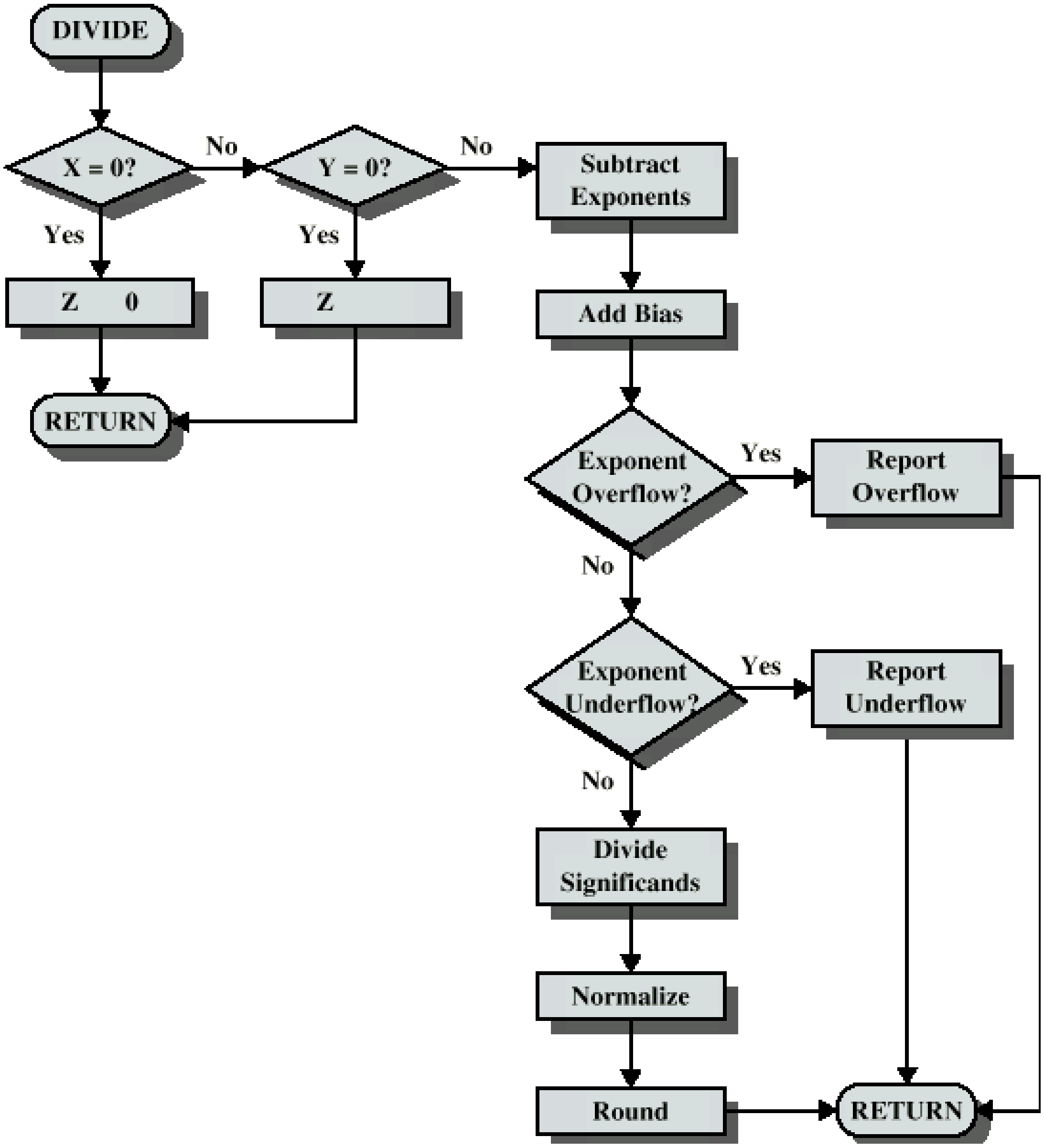
$= 0.101101 * 2^{011}$ (Normalized)

```
  0.1001
 *  0.101
     1001
    0000*
  +1001**
  101101 = 0.0101101
```

Floating Point Division

The division algorithm can be subdivided into 5 parts

1. Check for zeroes.
2. Initial registers and evaluates the sign.
3. Align the dividend.
4. Subtract the exponent.
5. Divide the mantissa.



Example:

$X = 0.101 * 2^{110}$

$Y = 0.1001 * 2^{-010}$

As we know, $Z = X / Y = (M_X / M_Y) * 2^{(EX – EY)}$

$M_X / M_Y = 0.101 / 0.1001 = (1/2 + 1/8) / (1/2 + 1/16) = 1.11 = 1.00011$

$0.11 * 2 = 0.22 \rightarrow 0$

$0.22 * 2 = 0.44 \rightarrow 0$

$0.44 * 2 = 0.88 \rightarrow 0$

$0.88 * 2 = 1.76 \rightarrow 1$

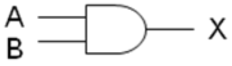
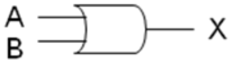
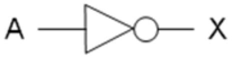
$0.76 * 2 = 1.52 \rightarrow 1$

$E_X – E_Y = 110 + 010 = 1000$

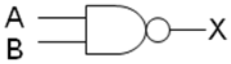
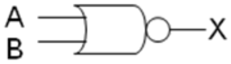
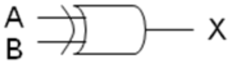
Now, $Z = M_Z * 2^{EZ} = 1.00011 * 2^{1000} = 0.100011 * 2^{1001}$

5.5 Logical Operation

Gate Level Logical Components

Name	Symbol	VHDL Equation	Truth Table
AND		X <= A and B	A B X 0 0 0 0 1 0 1 0 0 1 1 1
OR		X <= A or B	A B X 0 0 0 0 1 1 1 0 1 1 1 1
NOT		X <= not A	A X 0 1 1 0

Composite Logic Gates

Name	Symbol	VHDL Equation	Truth Table
NAND		X <= not (A and B)	A B X 0 0 1 0 1 1 1 0 1 1 1 0
NOR		X <= not (A or B)	A B X 0 0 1 0 1 0 1 0 0 1 1 0
XOR		X <= A xor B	A B X 0 0 0 0 1 1 1 0 1 1 1 0