Computer Organisation and Architecture

Course Code: CO206

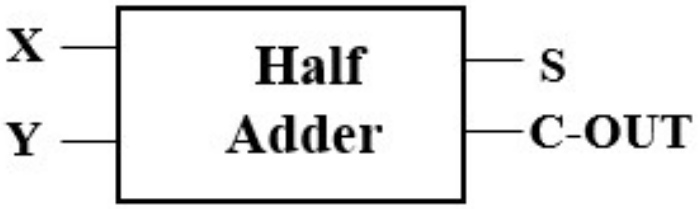
Module-3- Central Processing Unit

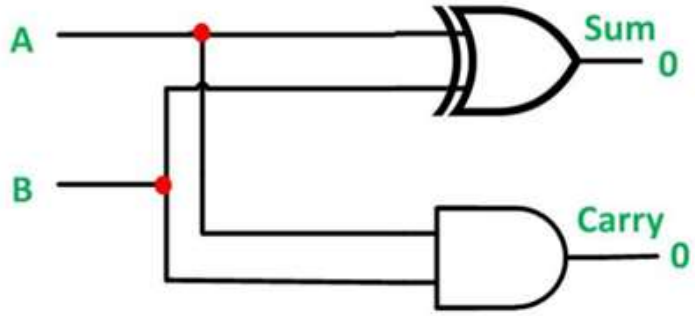
Contents of Module 3

- ❑ Addition and Subtraction of Signed Numbers
- ❑ Look Ahead Carry Adders
- ❑ Array Multiplier
- ❑ Signed Operand Multiplication: Booths Algorithm
- ❑ Division and logic operations
- ❑ Floating point arithmetic operation
- ❑ Processor Organisation, general register & Stack organization
- ❑ Addressing Modes

Half Adder

- Adding two single-bit binary values, X, Y and produces a sum bit S and a carry out C-out bit.

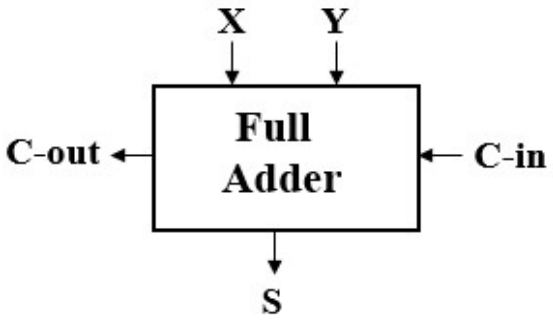


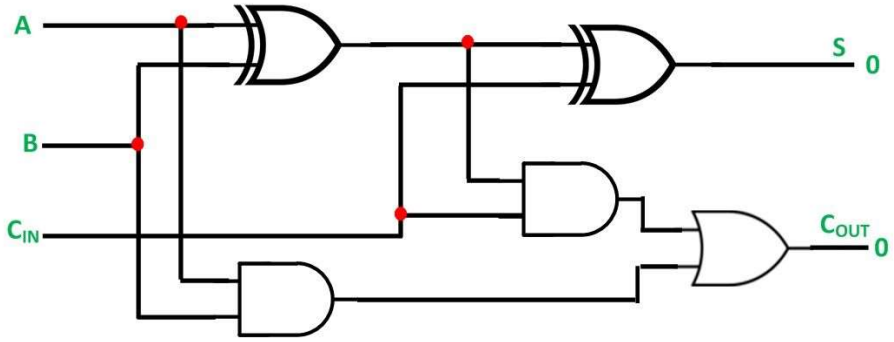


Inputs		Outputs	
X	Y	S	C-out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

- Adding two single-bit binary values, X, Y along with a carry input bit C-in and produces a sum bit S and a carry out C-out bit.



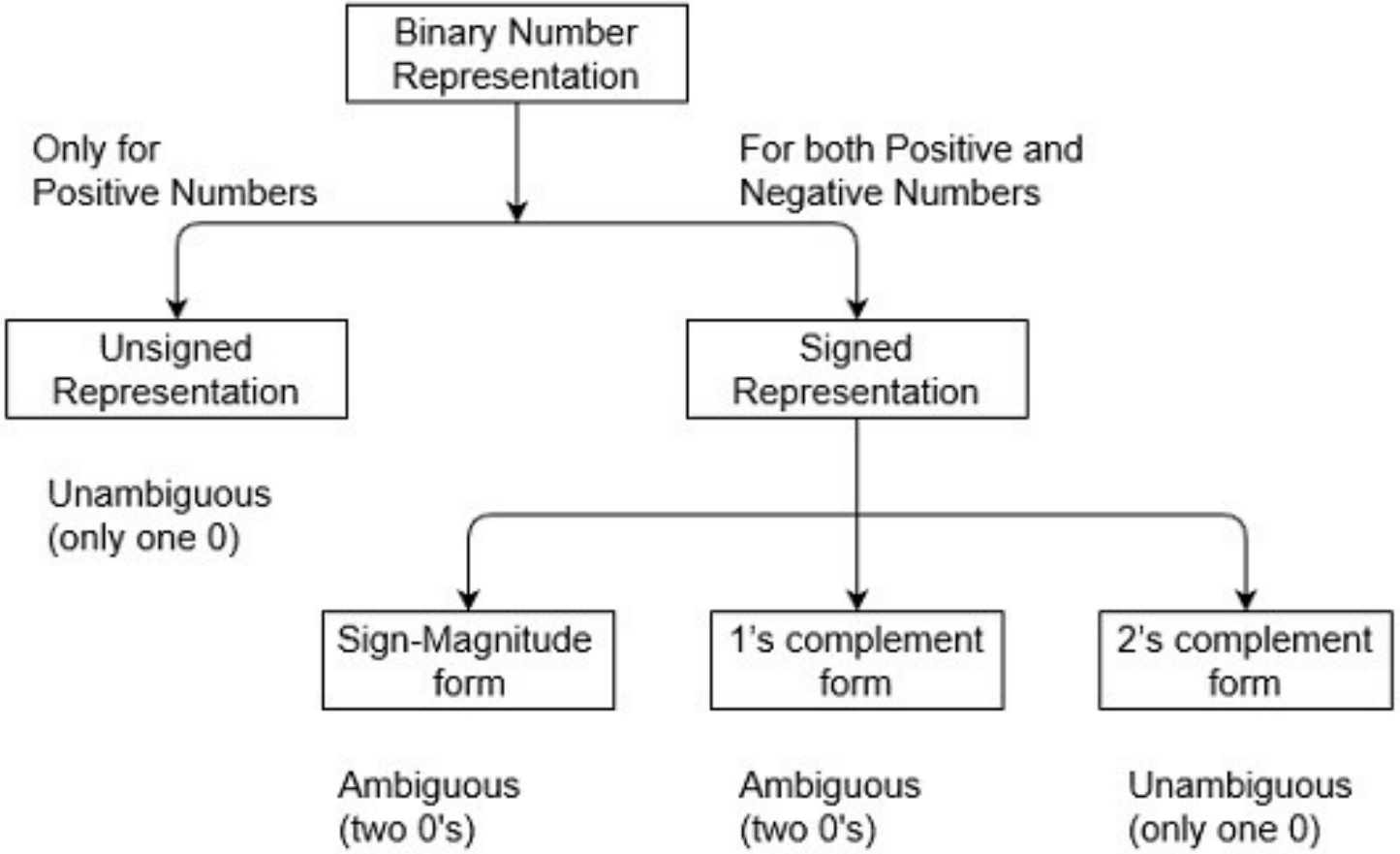


Inputs			Outputs	
X	Y	C-in	S	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Binary Number Representation



Signed Magnitude Representation

- For **n bit binary number**, **1 bit is reserved for sign symbol**.
- If the value of sign bit is 0, then the given number will be positive, else if the value of sign bit is 1, then the given number will be negative.
- Remaining (n-1) bits represent magnitude of the number.
- Since magnitude of number zero (0) is always 0, so there can be two representation of number zero (0), positive (+0) and negative (-0), which depends on value of sign bit.
- Hence these representations are **ambiguous** generally **because of two representation of number zero (0)**. Generally **sign bit is a most significant bit (MSB)** of representation.
- The **range** of Sign-Magnitude form is from **$(-2^{(n-1)}-1)$ to $(2^{(n-1)}-1)$**.

1's Complement Representation

- **1's complement** of a number is obtained by **inverting each bit** of given number.
- So, we **represent positive numbers in binary form** and **negative numbers in 1's complement form**.
- There is extra bit for sign representation. **If value of sign bit is 0, then number is positive** and you can directly represent it in simple binary form, but **if value of sign bit 1, then number is negative** and you have to take 1's complement of given binary number.
- You can get **negative number by 1's complement of a positive number** and **positive number by using 1's complement of a negative number**.
- Therefore, in this representation, **zero (0) can have two representation**, that's why **1's complement** form is also **ambiguous** form.
- The **range** of 1's complement form is from **$(-2^{(n-1)}-1)$ to $(2^{(n-1)}-1)$** .

2's Complement Representation

- **2's complement of a number is obtained by inverting each bit of given number plus 1 to least significant bit (LSB).**
- So, we represent positive numbers in binary form and negative numbers in 2's complement form.
- There is extra bit for sign representation. If value of sign bit is 0, then number is positive and you can directly represent it in simple binary form, but if value of sign bit 1, then number is negative and you have to take 2's complement of given binary number.
- You can **get negative number by 2's complement of a positive number** and positive number by directly using simple binary representation.
- If value of most significant bit (MSB) is 1, then take 2's complement from, else not.
- Therefore, in this representation, **zero (0) has only one (unique) representation** which is always positive.
- **The range of 2's complement form is from (-2^{n-1}) to $(2^{n-1}-1)$.**

Addition of Signed Binary Number

- Let us perform the addition of two decimal numbers +2 and +4 using 2's complement method using 4 bits.

- The 2's complement representations of +2 and +4 with 4 bits each are shown below.

  +2 = 0010
  +4 = 0100

The addition of these two numbers is
(+2) + (+4) = 0010 +0100 = 0110= +6

Addition of Signed Binary Number

Let us perform the addition of two decimal numbers -2 and -4 using 2's complement method using 4 bits.

The 2's complement representations of -2 and -4 with 4 bits each are shown below.

+2 = 0010 , -2= 1110
+4 = 0100, -4= 1100

The addition of these two numbers is

(-2) + (-4) = 1110 +1100 = 11010 (Discard carry) = -6

Addition of Signed Binary Number

- Unsigned addition in 4-bit arithmetic

```
( carry)   11_
          1011
        + 0011
        ------
          1110
```

- 11 + 3 = 14

- Signed addition in 4-bit arithmetic

```
( carry)  11_
          1011
        + 0011
        ------
          1110
```

- (-5)+ 3 = -2

Same rules apply even though bit strings represent ***different values.***

Addition of Signed Binary Number

- No overflow in signed arithmetic

```
( carry)111_
    1110
  + 0011
  ------
    0001
```

- `-2 + 3 = 1 (correct)`

- Signed addition in 4-bit arithmetic

```
( carry)  11__
    0110
  + 0011
  ------
    1001
```

- `6 + 3 ≠ -7 (false)`

Same rules apply even though bit strings represent ***different values.*** Sole difference is ***overflow handling***

4 bit Binary Numbers in 2's Complement form

The range of 2's complement form is from (-2^{n-1}) to $(2^{n-1}-1)$ i.e. -8 to 7

0001	1	1110	+1 = 1111	-1
0010	2	1101	+1 = 1110	-2
0011	3	1100	+1 = 1101	-3
0100	4	1011	+1 = 1100	-4
0101	5	1010	+1 = 1011	-5
0110	6	1001	+1 = 1010	-6
0111	7	1000	+1 = 1001	-7
1000	8	0111	+1 = 1000	-8

1001	-7	0110	+1 = 0111	7
1010	-6	0101	+1 = 0110	6
1011	-5	0100	+1 = 0101	5
1100	-4	0011	+1 = 0100	4
1101	-3	0010	+1 = 0011	3
1110	-2	0001	+1 = 0010	2
1111	-1	0000	+1 = 0001	1
0000	0	1111	+1 = 0000	0

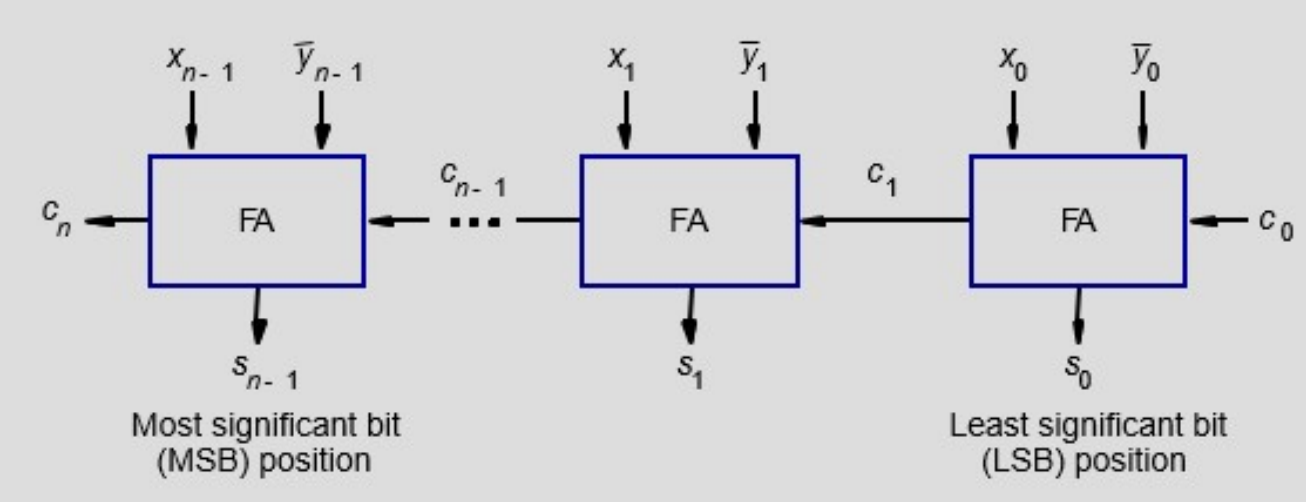
Overflow in Signed Arithmetic

- In signed arithmetic an overflow happens when
  - The sum of two positive numbers exceeds the maximum positive value that can be represented using n bits: $2^{n-1} - 1$
  - The sum of two negative numbers falls below the minimum negative value that can be represented using n bits: -2^{n-1}

Example of Overflow

- Four-bit arithmetic:
  - 16 possible values
  - Positive overflow happens when result > 7
  - Negative overflow happens when result < -8
- Eight-bit arithmetic:
  - 256 possible values
  - Positive overflow happens when result > 127
  - Negative overflow happens when result < -128

Detection of Overflow



$$Overflow = x_{n-1}y_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1}$$

$$Overflow = c_n \oplus c_{n-1}$$

(carry) 1___ 1100 + 1011 0111	(carry) 11__ 0110 + 0011 1001

Subtraction of Signed Binary Number(2's Complement)

- Whenever we have to subtract a number B from number A, then take 2's complement of B and add it to A. So, mathematically we can write it as

  A - B = A+(-B)=A + 2's complement of B

- **Rule for Subtraction**
  - Suppose we have to perform A-B. It can also be written as A + (-B)
  - Take 2's complement of B
  - Sum= A + 2's complement of B
  - Case1: If carry is generated, ignore carry, Sum is the final result.
  - Case2: If carry is not generated, sum is negative(i.e. in 2's complement form). Take 2's complement of the sum to get the result.

Subtraction of Signed Binary Number(2's Complement)

❑ **Example 1:** Subtract (+4)- (+7)

- ❑ The given expression can be written as (+4)+(−7).

- ❑ The 2's complement representation of +4 and -7 with 4 bits each are shown below.

  +4 = 0100

  −7 = 1001

  ⇒+4 + −7 = 0100 + 1001 = 1101

- ❑ Here, carry is not obtained.
- ❑ The sign bit '1' indicates that the resultant sum is negative.
- ❑ So, by taking 2's complement of it we will get the magnitude of resultant sum as 3 in decimal number system.
- ❑ Therefore, subtraction of two decimal numbers +4 and +7 is -3.

Subtraction of Signed Binary Number(2's Complement)

❑ **Example 2:** Subtract (+7) – (+4)

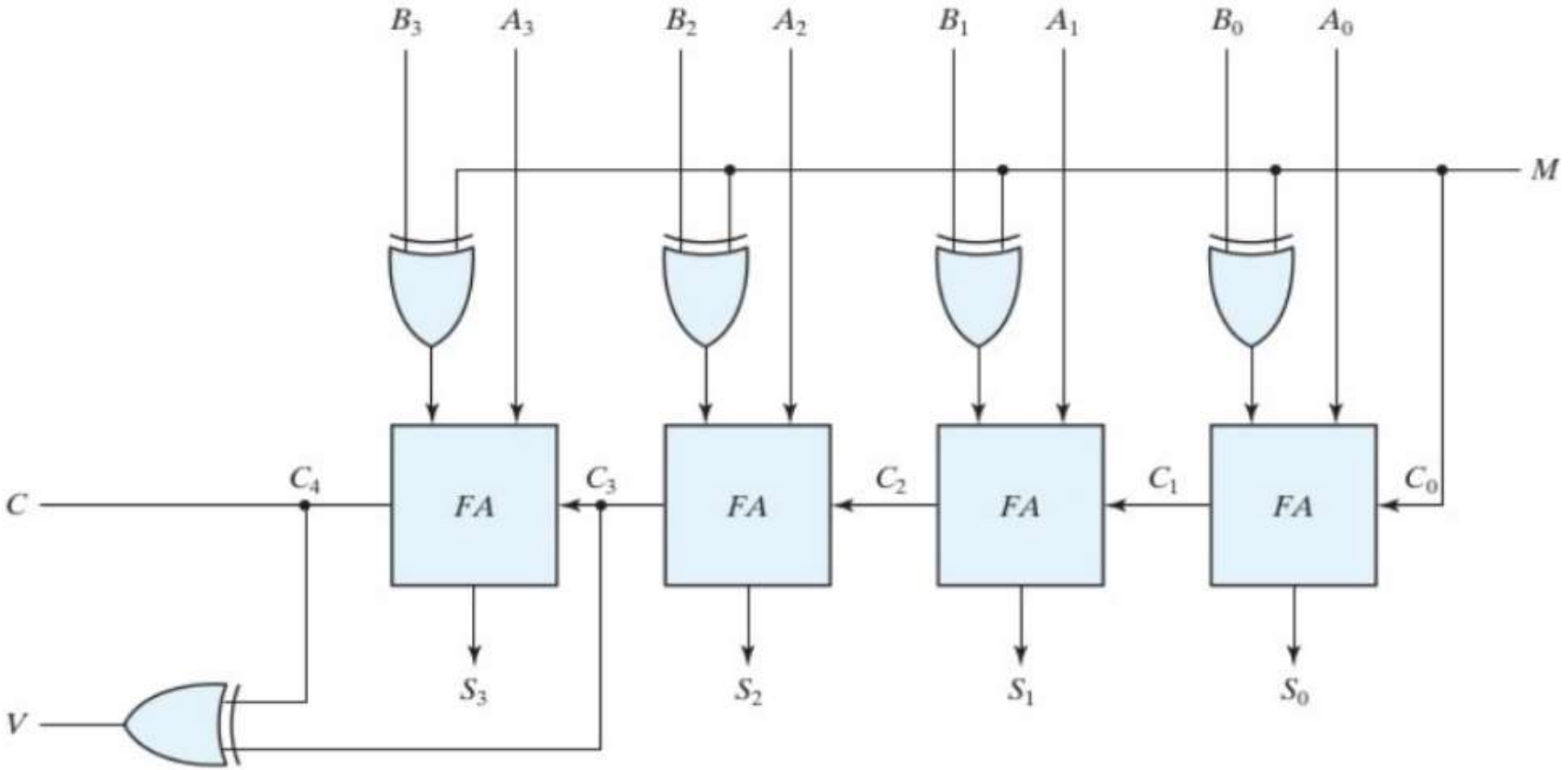
- ❑ The 2's complement representation of +7 and -4 with 4 bits each are shown below.

  +7 = 0111

  -4 = 1100

  ⇒+7 + -4 = 0111 + 1100 = 1 0011
- ❑ Here, the carry is obtained. So, we can discard it.
- ❑ The resultant sum after removing carry is = 0011
- ❑ The sign bit '0' indicates that the resultant sum is positive. So, the magnitude of it is 3 in decimal number system.
- ❑ Therefore, subtraction of two decimal numbers +7 and +4 is +3.

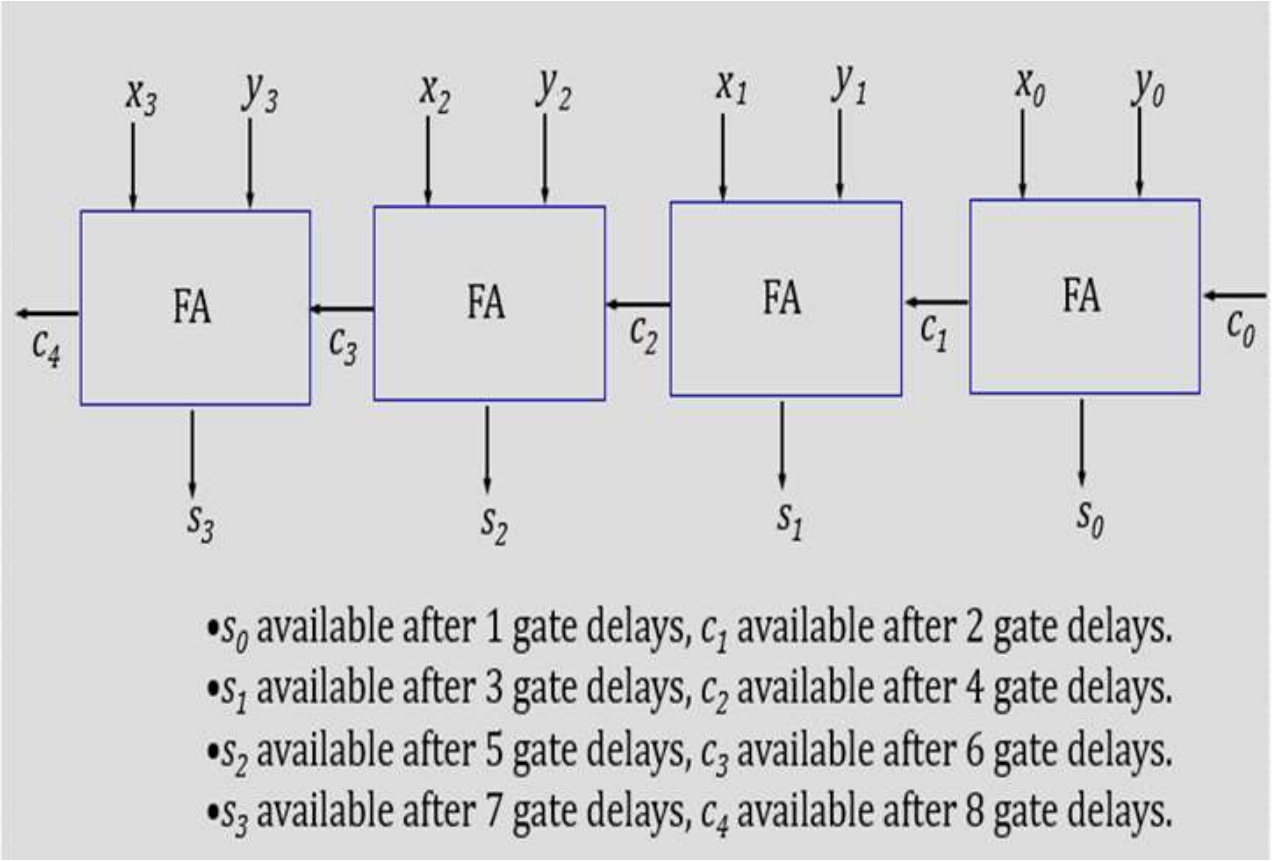
4 Bit Binary Adder/Subtractor

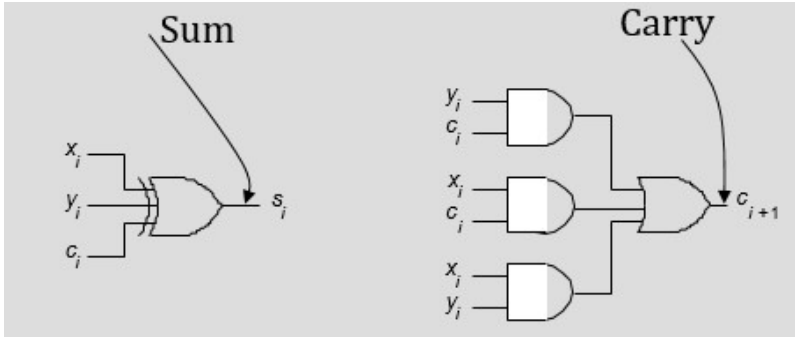


Here M is mode bit

M=0 : Addition
M=1: Subtraction

Look Ahead Carry Adder





Inputs			Outputs	
X	**Y**	**C-in**	**S**	**C-out**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

For an n-bit adder, s_{n-1} is available after $2n-1$ gate delays c_n is available after $2n$ gate delays.

Look Ahead Carry Adder

Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as:

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

We can write:

$$c_{i+1} = G_i + P_i c_i$$

$$where\ G_i = x_i y_i\ and\ P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i are computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.

Look Ahead Carry Adder

$$c_{i+1} = G_i + P_i c_i$$

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1} c_{i-1})$$

continuing

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} c_{i-2}))$$

until

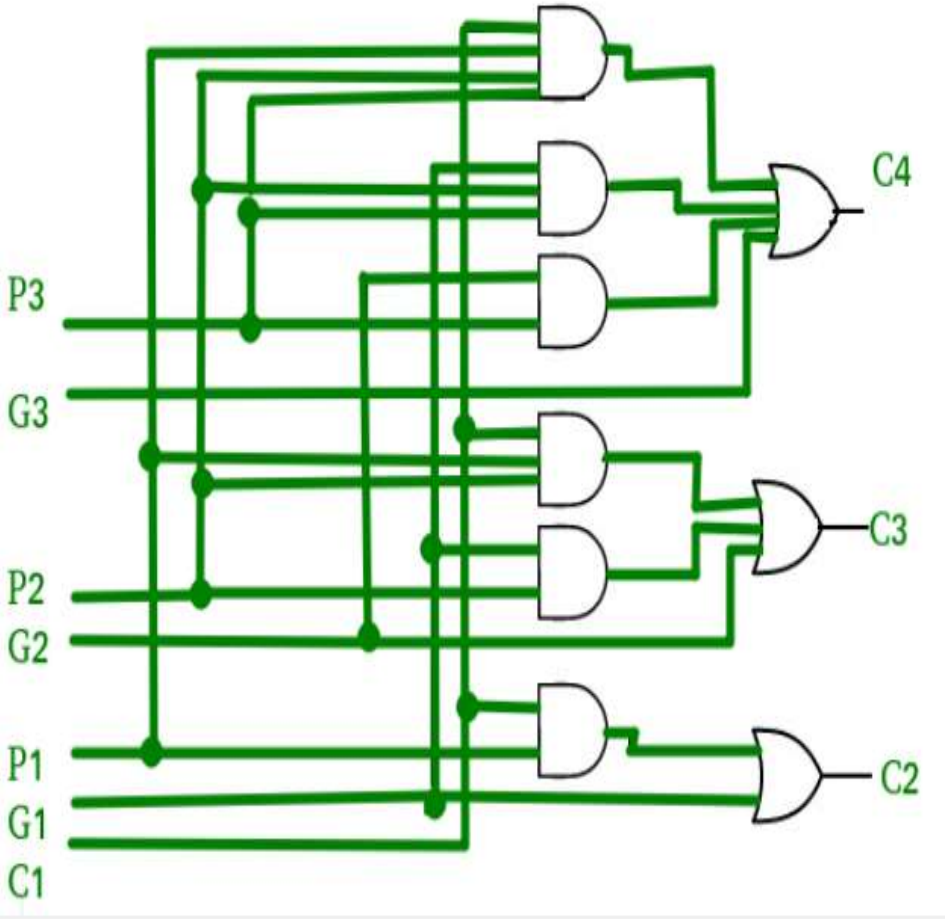
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

$$C_1 = G_0 + P_0 C_0$$

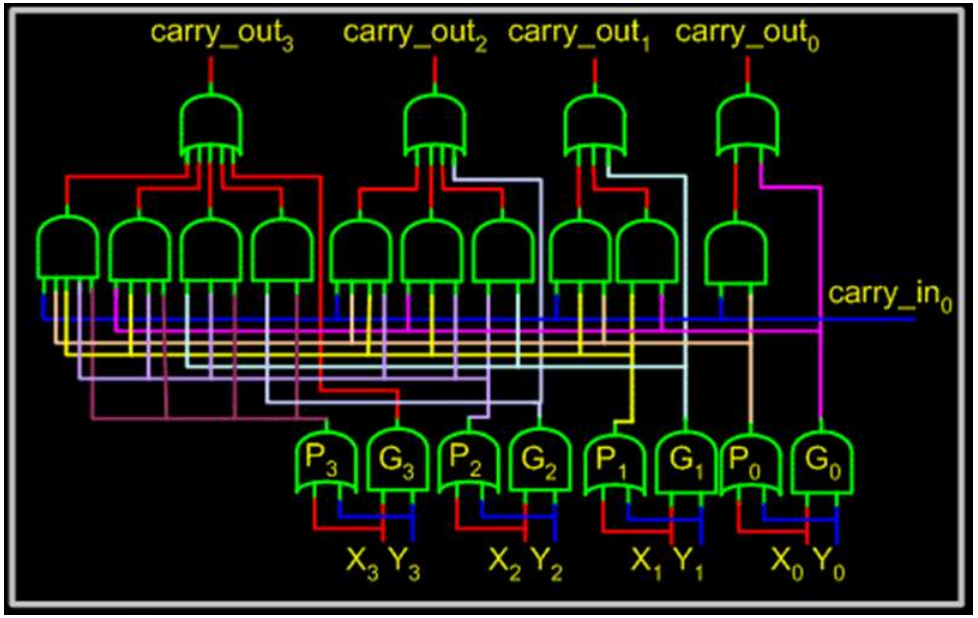
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

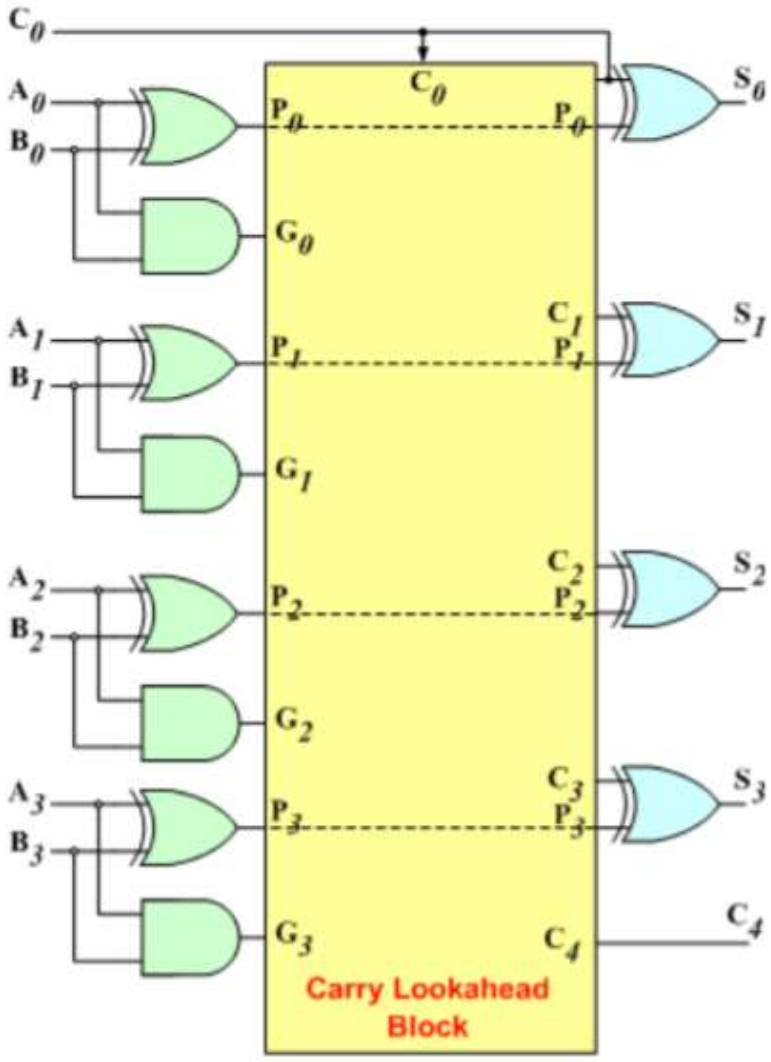


Look Ahead Carry Adder



- All carries can be obtained 3 gate delays after X, Y and c_0 are applied.
  - -One gate delay for P_i and G_i
  - -Two gate delays in the AND-OR circuit for c_{i+1}
- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of n, n-bit addition requires only 4 gate delays.
- This is called Carry Lookahead adder.

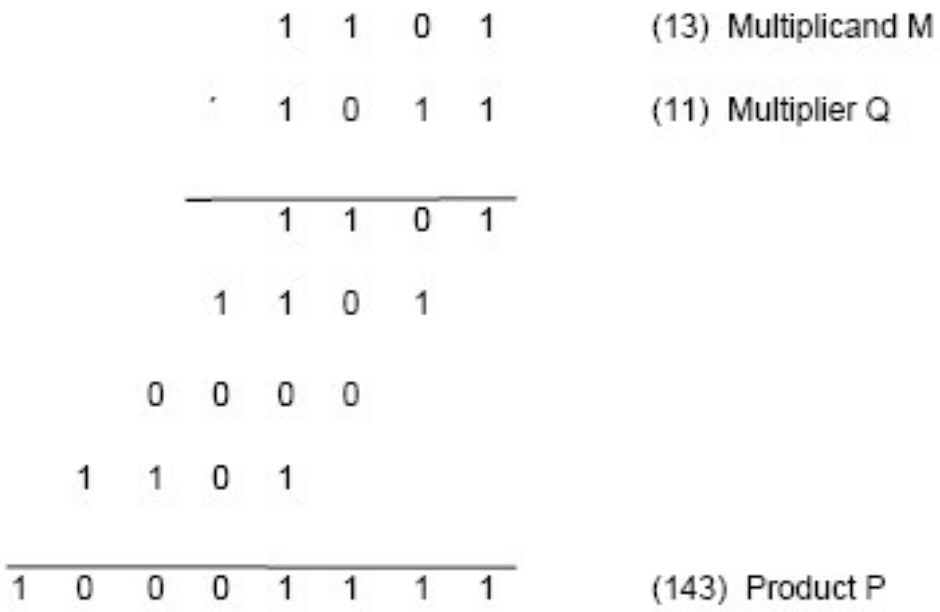
Look Ahead Carry Adder



Array Multiplier

- Multiplication of binary numbers is performed in the same way as with decimal numbers.
- The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit.
- The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left.
- The product is obtained by adding these shifted partial products.

Array Multiplier



Product of two *n*-bit numbers is at most a *2n*-bit number.

Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

Array Multiplier

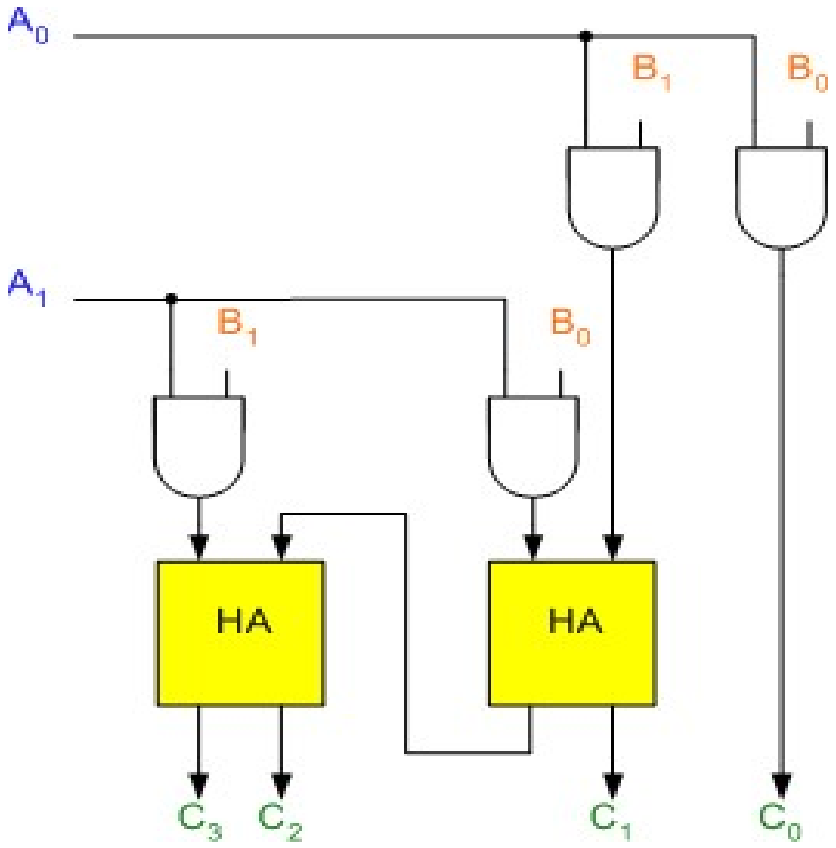
Example 1: Consider an example of multiplication of two numbers, say A and B (2 bits each), ***C = A*** x ***B*.**

- The first partial product is formed by multiplying the B_1B_0 by A_0**.** The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.
- The second partial product is formed by multiplying the B_1B_0 by A_1 and is shifted one position to the left.

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & \times & A_1 & A_0 \\
\hline
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}
$$

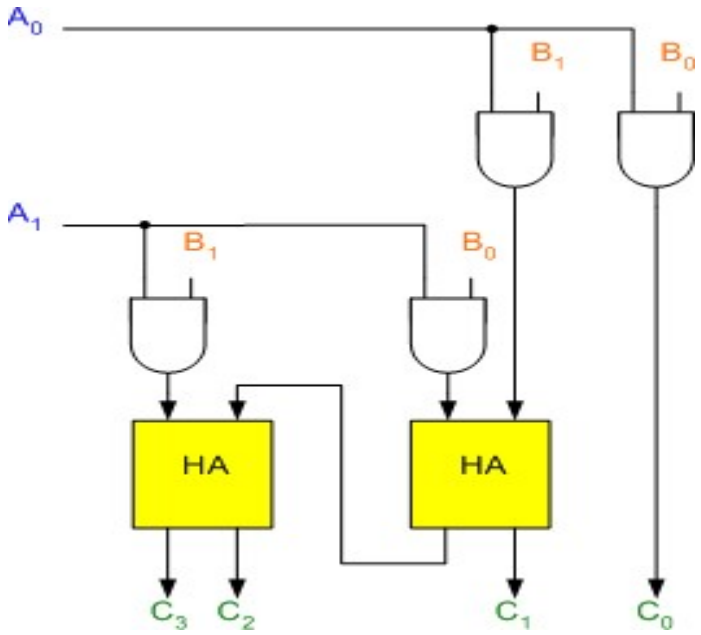
Array Multiplier

- The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use FAs.



Array Multiplier

- The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use FAs.

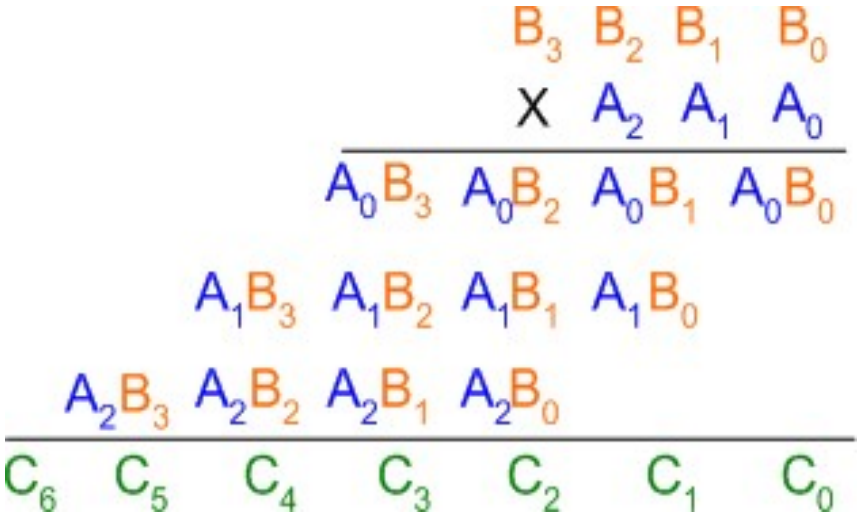


- The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate
- A binary multiplier with more bits can be constructed in a similar manner.

Array Multiplier

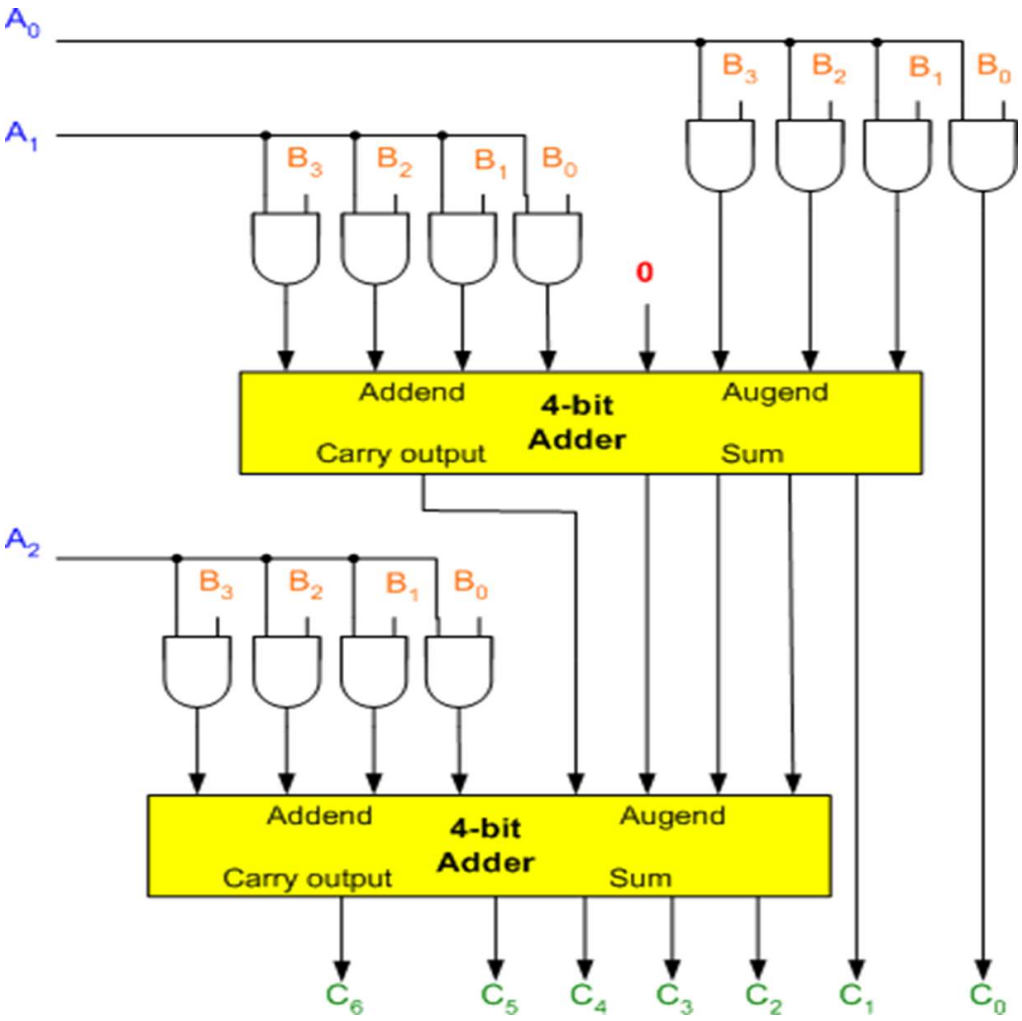
Example 2: Consider the example of multiplying two numbers, say A (3-bit number) and B (4-bit number).

- Each bit of A (the multiplier) is ANDed with each bit of B (the multipcant) as shown in the Figure.



- The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the final product.

Array Multiplier



Signed Multiplication

Considering 2's-complement signed operands, what will happen to (-13)×(+11) if following the same method of unsigned multiplication?

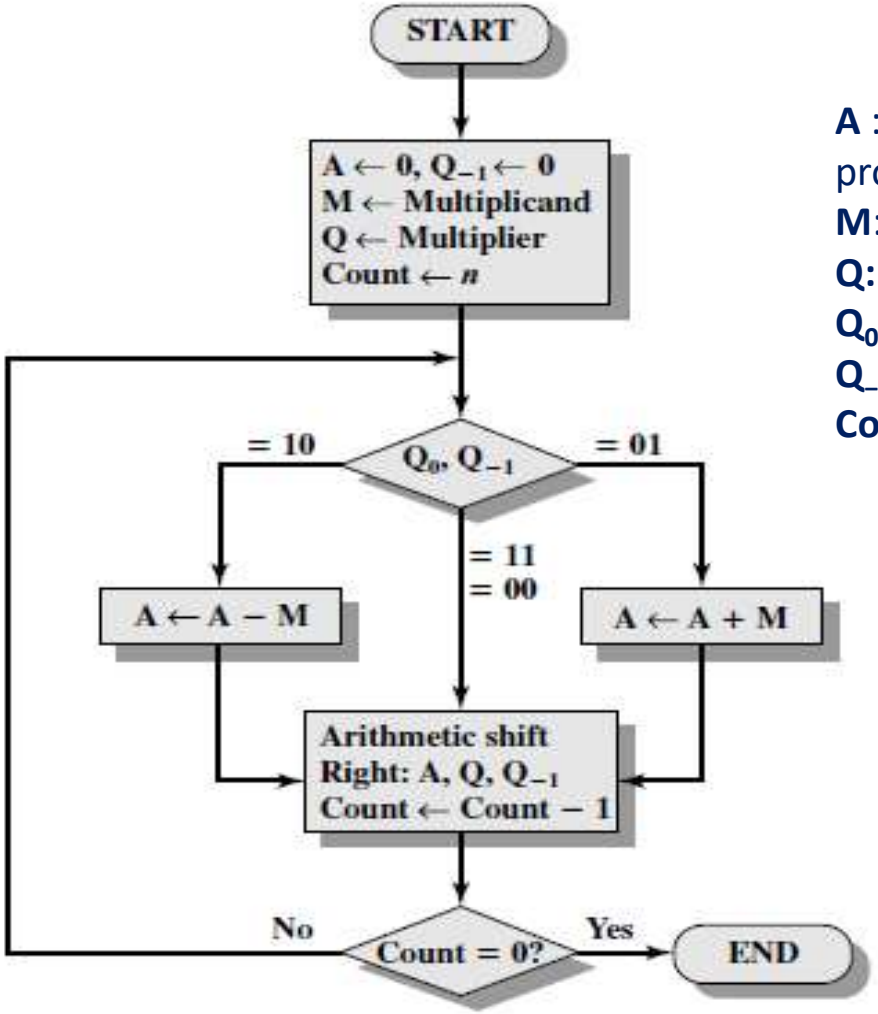
					1	0	0	1	1	(-13)
					0	1	0	1	1	(+11)
1	1	1	1	1	1	0	0	1	1	
1	1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0	0			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	0	1	1	1	0	0	0	1	(-143)

Sign extension is shown in blue

Rule:

1. Extend the sign upto 2n bits (here n represents the bits used in representing the number)
2. Discard the carry if generated.

Booths Algorithm for Signed Multiplication



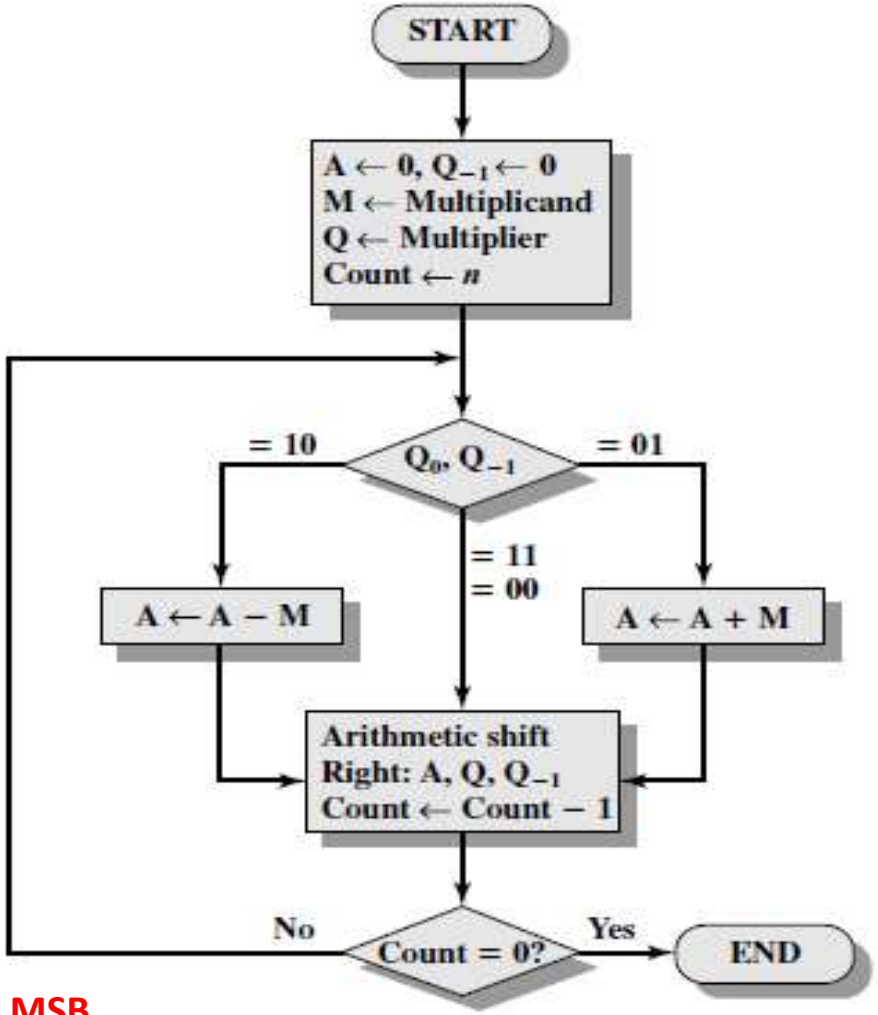
A : It represents the **Accumulator** which stores the partial product, It is initialized with Zero (0)
M: It represents the **multiplicand**
Q: It represents the multiplier,
Q_0: it represents the LSB of Q
Q_{-1} : It represents a Flip Flop which is initialized with Zero(0)
Count: It represents the counter(number of bits in **M or Q**)

Note: In Arithmetic Shift Right, We copy the sign bit of the number in MSB

Example: Booths Algorithm for Signed Multiplication

Perform 7 x 3
7: 0111
3: 0011

A	Q	Q_{-1}	M	
	3		7	
0000	0011	0	0111	
1001	0011	0	0111	A <- (A - M) 1st cycle
1100	1001	1	0111	Shift
1110	0100	1	0111	Shift
0101	0100	1	0111	A <- (A + M) 2nd cycle
0010	1010	0	0111	Shift
0001	0101	0	0111	Shift

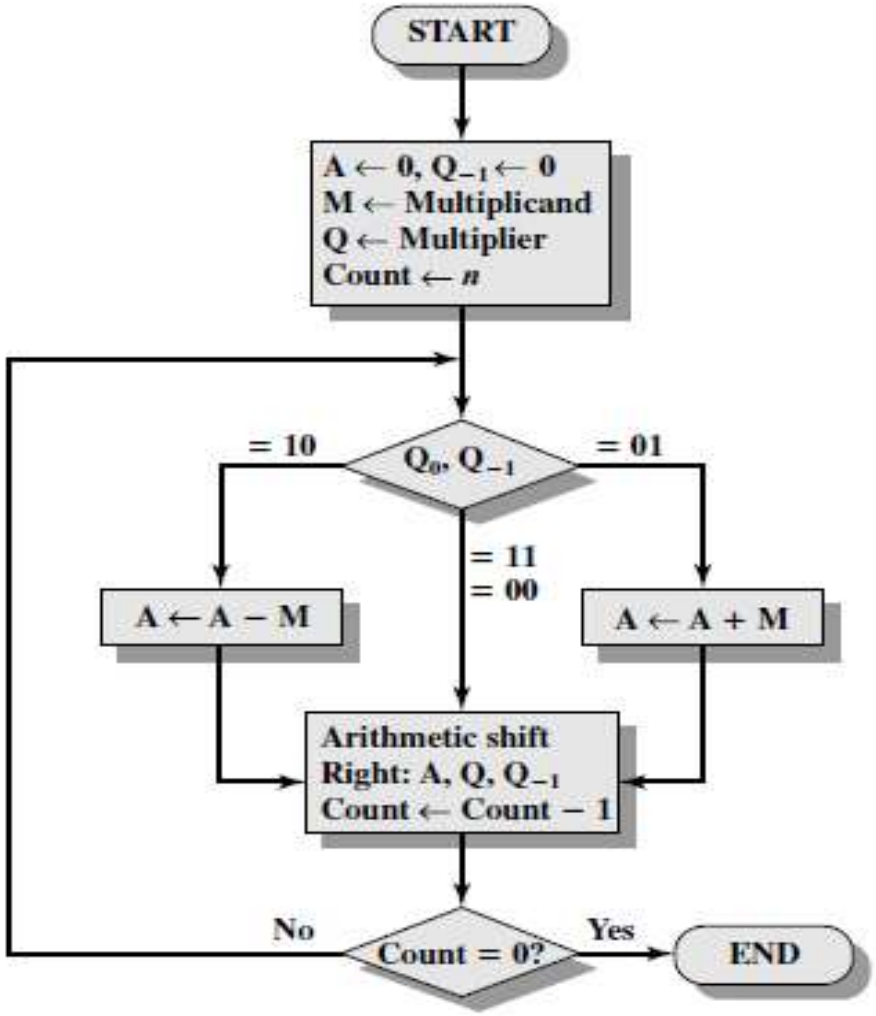


Note: In Arithmetic Shift Right, We copy the sign bit of the number in MSB

Example: Booths Algorithm for Signed Multiplication

Perform 7 x (-3)
7: 0111
3: 0011
-3: 1101

A	Q	Q_{-1}	M	
	(-3)		7	
0000	1101	0	0111	
1001	1101	0	0111	A <- (A - M) 1st cycle
1100	1110	1	0111	Shift
0011	1110	1	0111	A <- (A + M) 2nd cycle
0001	1111	0	0111	Shift
1010	1111	0	0111	A <- (A - M) 3rd cycle
1101	0111	1	0111	Shift
1110	1011	1	0111	Shift



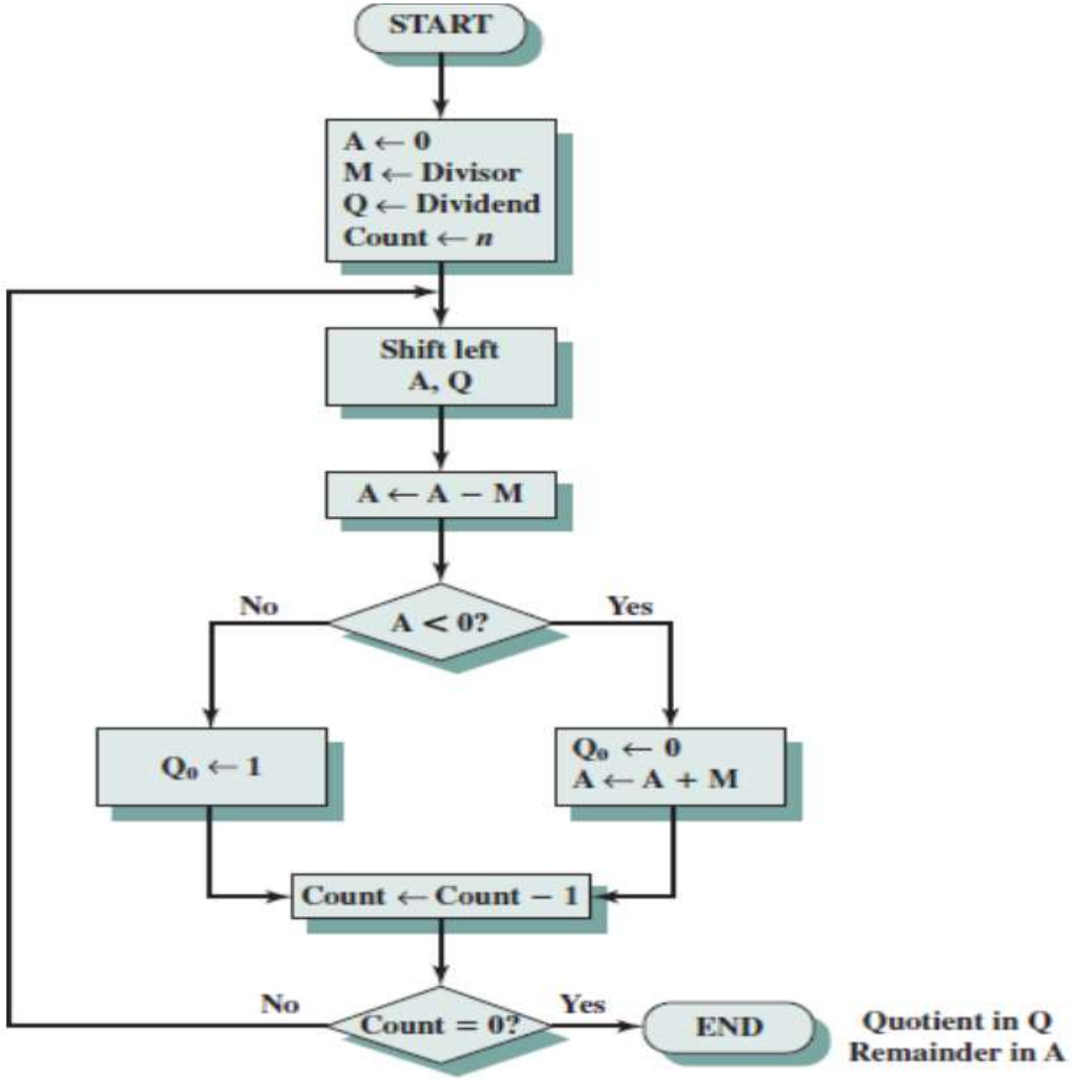
Binary Division

```
      21
13 ) 274
     26
     --
     14
     13
     --
      1
```

```
Divisor:                11010          Quotient = Q
B = 10001 )0111000000                  Dividend = A
           01110                       5 bits of A < B, quotient has 5 bits
           011100                      6 bits of A ⩾ B
           - 10001                     Shift right B and subtract; enter 1 in Q
           -010110                     7 bits of remainder ⩾ B
           --10001                     Shift right B and subtract; enter 1 in Q
           --001010                    Remainder < B; enter 0 in Q; shift right B
           ---010100                   Remainder ⩾ B
           ----10001                   Shift right B and subtract; enter 1 in Q
           ----000110                  Remainder < B; enter 0 in Q
           -----00110                  Final remainder
```

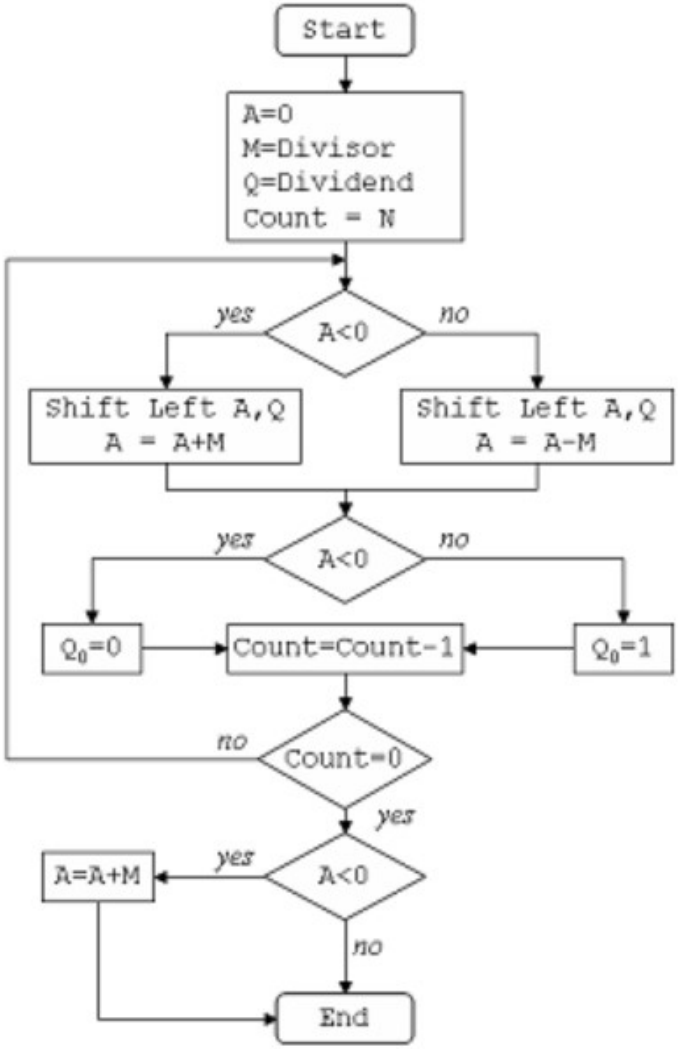
Division using Restoring Algorithm

Operation	n	M	A	Q= $Q_3Q_2Q_1Q_0$
Initialization	4	00011	00000	1011
Shift Left AQ			00001	011?
A=A-M			00001 +11101 11110	011?
Q_0=0 (Restore A)	3		00001	0110
Shift Left AQ			00010	110?
A=A-M			00010 +11101 11111	110?
Q_0=0 (Restore A)	2		00010	1100
Shift Left AQ			00101	100?
A=A-M			00101 +11101 00010	100?
Q_0=1	1		00010	1001
Shift Left AQ			00101	001?
A=A-M			00101 +11101 00010	001?
Q_0=1	0		00010	0011



Division using Non-Restoring Algorithm

Operation	n	M	A	Q= $Q_3Q_2Q_1Q_0$
Initialization	4	00011	00000	1011
Shift Left AQ			00001	011?
A=A-M			00001 +11101 11110	011?
Q_0=0	3		11110	0110
Shift Left AQ			11100	110?
A=A+M			11100 +00011 11111	110?
Q_0=0	2		11111	1100
Shift Left AQ			11111	100?
A=A+M			11111 +00011 00010	100?
Q_0=1	1		00010	1001
Shift Left AQ			00101	001?
A=A-M			00101 +11101 00010	001?
Q_0=1	0		00010	0011



Signed Binary Division

- To deal with negative numbers, we recognize that the remainder is defined by

  D = Q * V + R

- Consider the following examples of integer division with all possible combinations of signs of D and V:
  - **D = 7 V = 3 Q = 2 R = 1**
  - **D = 7 V = -3 Q = -2 R = 1**
  - **D = -7 V = 3 Q = -2 R = -1**
  - **D = -7 V = -3 Q = 2 R = -1**
- We see that the magnitudes of Q and R are unaffected by the input signs and that the signs of Q and R are easily derivable from the signs of D and V.
- Specifically, sign(R) = sign(D) and sign(Q) = sign(D) * sign(V).
- Hence, one way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed.

IEEE Standard for Floating Point Numbers

- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**.
- IEEE 754 has 3 basic components:

- ***The Sign of Mantissa –***
  0 represents a positive number while 1 represents a negative number.
- ***The Biased exponent –***
  A bias is added to the actual exponent in order to get the stored exponent.
- ***The Normalized Mantissa –***
  A normalized mantissa is one with only one 1 to the left of the decimal.

IEEE Standard for Floating Point Numbers

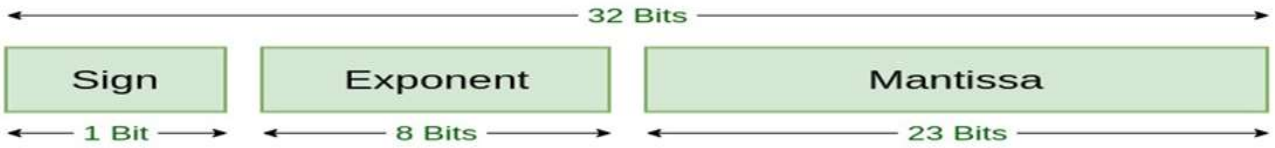
Floating Point Representation has 3 parts:

1. Mantissa
2. Base
3. Exponent

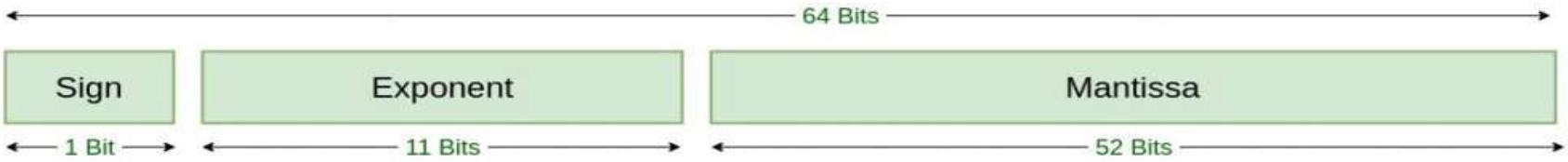
Number	Mantissa	base	exponent
$3 X10^6$	**3**	**10**	**6**
$110 x 2^8$	**110**	**2**	**8**
6132.784	**6132784**	**10**	**-3**

IEEE Standard for Floating Point Numbers

IEEE 754 numbers are divided into two based on the above three components: single precision and double precision.



Single Precision
IEEE 754 Floating-Point Standard



Double Precision
IEEE 754 Floating-Point Standard

IEEE Standard for Floating Point Numbers

Floating Point Examples

(a) Format

$$
\begin{aligned}
1.1010001 \times 2^{10100} &= 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20} \\
-1.1010001 \times 2^{10100} &= 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20} \\
1.1010001 \times 2^{-10100} &= 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20} \\
-1.1010001 \times 2^{-10100} &= 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}
\end{aligned}
$$

(b) Examples

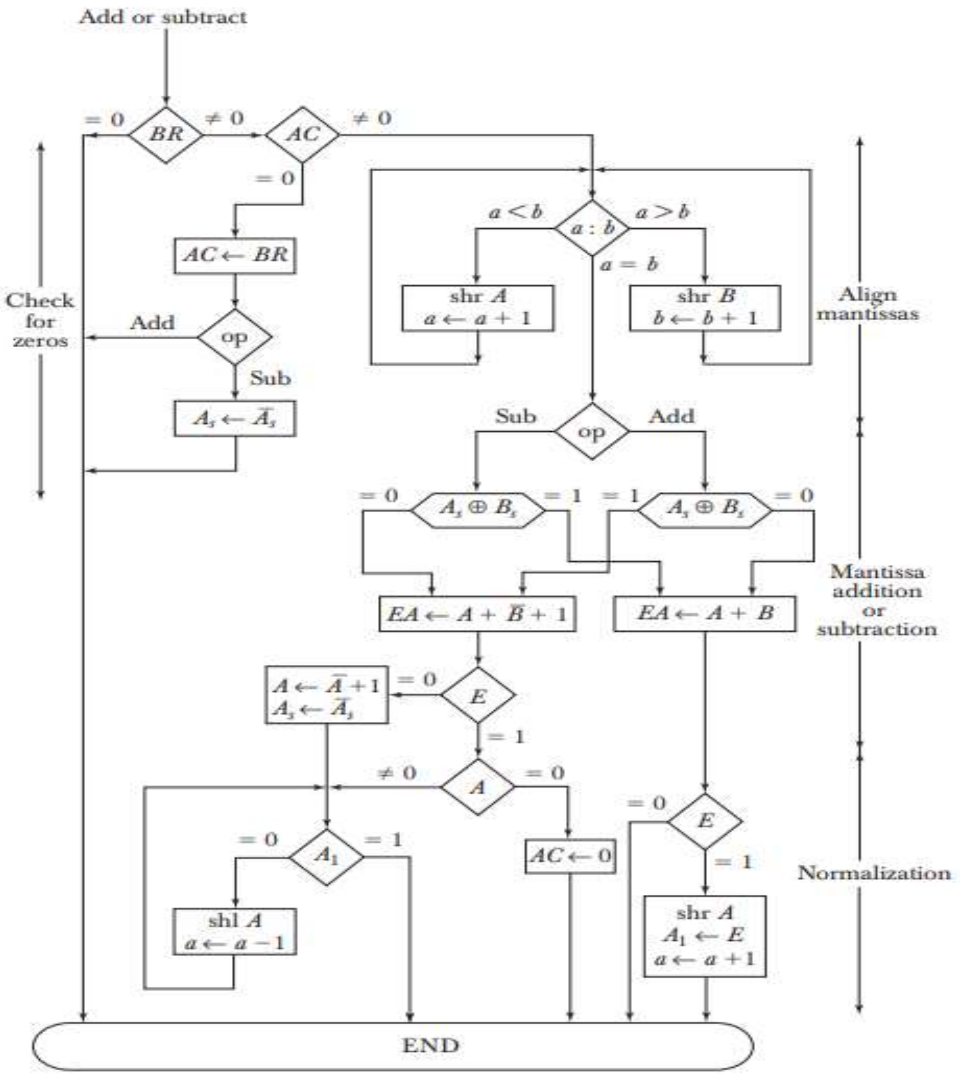
IEEE Standard for Floating Point Numbers

EXPONENT	MANTISA	VALUE
0	0	exact 0
255	0	Infinity
0	not 0	denormalised
255	not 0	Not a number (NAN)

Floating Point Addition and Subtraction

The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.



Example: Floating Point Addition

- Say **101 + 11** or **$1.01\times2^2 + 1.1\times2^1$**
- Denormalize number with smaller exponent:
  $1.01\times2^2 + 0.11\times2^2$
- Add the numbers:
  $1.01\times2^2 + 0.11\times2^2 = 10.00\times2^2$
- Normalize the results
  $10.00\times2^2 = 1.000\times2^3$

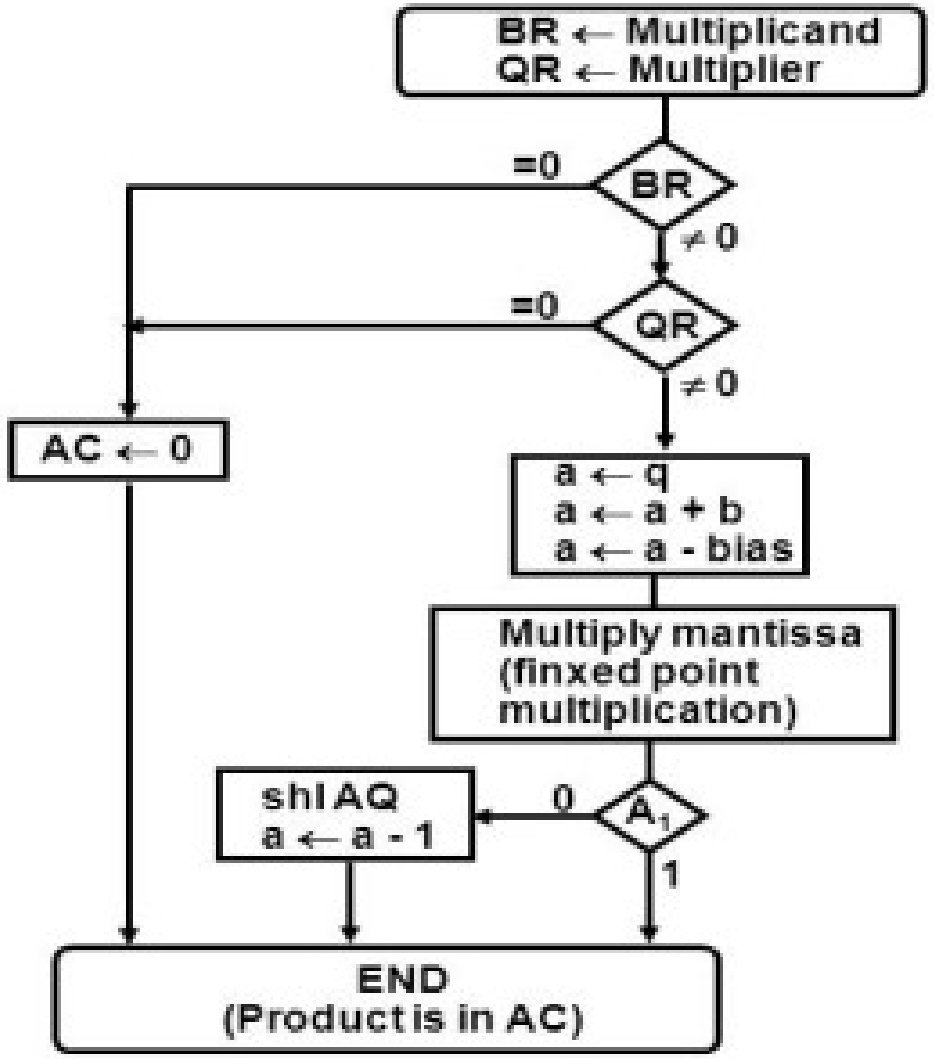
Example: Floating Point Subtraction

- Say **101 – 11** or **$1.01\times2^2 - 1.1\times2^1$**
- Denormalize number with smaller exponent:
  $1.01\times2^2 - 0.11\times2^2$
- Perform the subtraction:
  $1.01\times2^2 - 0.11\times2^2 = 0.10\times2^2$
- Normalize the results
  $0.10\times2^2 = 1.0\times2^1$

Floating Point Multiplication

The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.



Floating Point Multiplication

- Say **110 ×11** or **$1.1\times2^2 \times 1.1\times2^1$**
- Exponent of product is:

  2 + 1 = 3
- Multiply the coefficients:

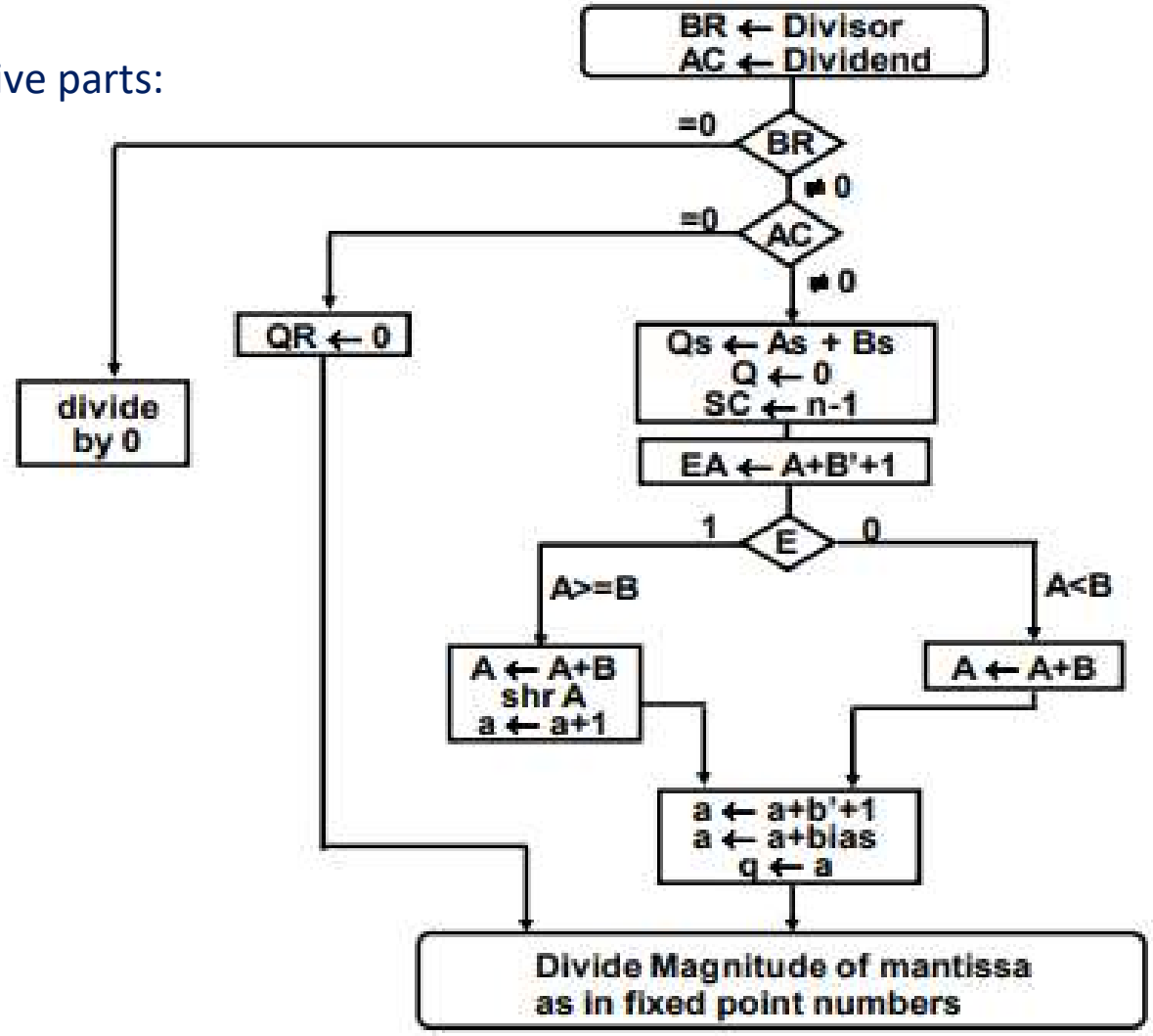
  1.1 × 1.1 = 10.01
- Result will be positive
- Normalize the result:

  $10.01\times2^3 = 1.001\times2^4$

Floating Point Division

The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Initialise registers and evaluate the sign.
3. Align the dividend
4. Subtract the exponents.
5. Divide the mantissa.



Floating Point Division

- Say **1100 /110** or **1.100×2^3 / 1.10×2^2**
- Subtract the exponents:

  3 -2 = 1
- Divide the mantissas:

  1.100 / 1.10= 1.0
- Normalize the result:

  $1.0\times2^1 = 1.0\times2^1$