Number System Arithmetic

Module 2

Introduction

- Algorithm – solution to any problem that is stated by a finite number of well-defined procedural steps.
- Algorithms for addition, subtraction, multiplication and division
  - Fixed point binary data in signed magnitude representation
  - Fixed point binary data in signed 2's complement representation
  - Floating point binary data

Signed Numbers

Need to be able to represent both *positive* and *negative* numbers

- Following 3 representations

Signed magnitude representation
Signed 1's complement representation
Signed 2's complement representation

Example: Represent +9 and -9 in 7 bit-binary number

Only one way to represent +9 ==> 0 001001
Three different ways to represent -9:

In signed-magnitude:	**1 001001**
In signed-1's complement:	**1 110110**
In signed-2's complement:	**1 110111**

In general, in computers, fixed point numbers are represented either integer part only or fractional part only.

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Pros and cons of integer representation

- Signed magnitude representation:
  - 2 representations for 0
  - Simple
  - 255 different numbers can be represented.
  - Need to consider both sign and magnitude in arithmetic
  - Different logic for addition and subtraction
- 1's complement representation:
  - 2 representations for 0
  - Complexity in performing addition and subtraction
  - 255 different numbers can be represented.
- 2's complement representation:
  - Only one representation for 0
  - 256 different numbers can be represented.
  - Arithmetic works easily
  - Negating is fairly easy

(a)	0010	(+2)		(b)	0100	(+4)
	+ 0011	(+3)			+ 1010	(−6)
	0101	(+5)			1110	(−2)
(c)	1011	(−5)		(d)	0111	(+7)
	+ 1110	(−2)			+ 1101	(−3)
	1001	(−7)			0100	(+4)
(e)	1101	(−3)			1101	
	− 1001	(−7)	⇒		+ 0111	
					0100	(+4)
(f)	0010	(+2)			0010	
	− 0100	(+4)	⇒		+ 1100	
					1110	(−2)
(g)	0110	(+6)			0110	
	− 0011	(+3)	⇒		+ 1101	
					0011	(+3)

Overflow

- Overflow in unsigned addition – result occupies more number of bits.
- Overflow in signed-magnitude / 2's complement addition – result occupies more number of bits / addition of two positive number resulting negative number and sum of two negative numbers resulting in positive number.
- There will no overflow during unsigned subtraction / signed – magnitude subtraction.
- Overflow bit in 2's complement subtraction is ignored.

2’s complement subtraction

- Represent negative numbers in 2’s complement form
- Subtract the given two numbers
  - Take 2’s complement of subtrahend (second number) and add to the minuend (first number)
- If carry
  - Discard it
  - Answer is positive
- Else
  - Answer is negative
  - So, answer is in 2’s complement form

Overflow / Underflow Problem

- Addition and bit-size restriction allow for possible overflow / underflow
- Overflow – when the addition of two binary numbers yields a result that is greater than the maximum possible value
- Underflow – when the addition/subtraction of two binary numbers yields a result that is less than the minimum possible value

Overflow Example

- Assume 4-bit restriction and 2's complement
- Maximum possible value: $2^{4-1} - 1 = 7$

$6_{10} + 3_{10} = 9_{10}$

$$\begin{array}{rr} 0110_2 & 6_{10} \\ \underline{+0011_2} & \underline{+3_{10}} \\ 1001_2 & -7_{10} \end{array}$$

← not good!

Underflow Example

- Assume 4-bit restriction and 2's complement
- Minimum possible value: $-(2^{4-1}) = -8$

$-5_{10} + -5_{10} = -10_{10}$

$$\begin{array}{rr} 1011_2 & -5_{10} \\ \underline{+1011}_2 & \underline{+-5}_{10} \\ 0110_2 & 6_{10} \end{array}$$

← not good!

Signed Overflow

```
  1101 (-3)
 +1011 (-5)
 ----------
  1000 (-8)
```

carry generated, but no overflow

```
  1
  1101 (-3)
 +1010 (-6)
 ----------
1 0111 (+7)
```

carry and overflow

```
  1101 (-3)
 +0010 (+2)
 ----------
0 1111 (-1)
```

no carry and no overflow

```
  0111 (+7)
 +0011 (+3)
 ----------
0 1010 (-6)
```

no carry and overflow

Addition/subtraction of signed numbers

x_i	y_i	Carry-in c_i	Sums$_i$	Carry-out c_{i+1}
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

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

At the ith stage:

Input:

c_i is the carry-in

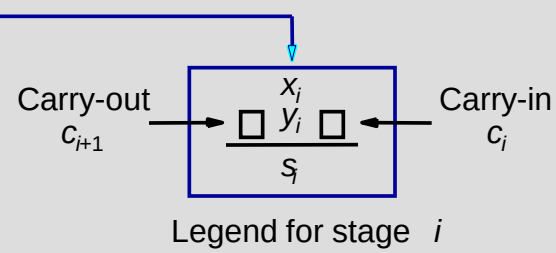
Output:

s_i is the sum

c_{i+1} carry-out to $(i+1)^{st}$ state

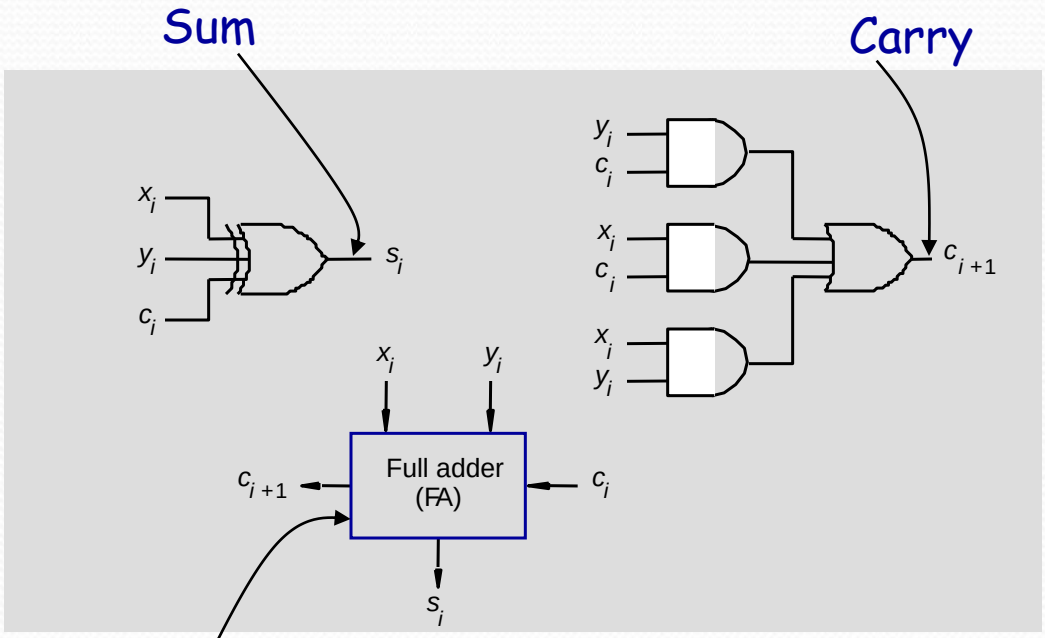
Example:

$$\begin{array}{r} X \\ +\ Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\ 6 \\ \hline 13 \end{array} = \begin{array}{r} 0\ \ 1\ \ \boxed{1}\ \ 1 \\ +\ {}_0 0\ {}_1 1\ \boxed{{}_1 1\ {}_0}\ 0\ {}_0 \\ \hline 1\ \ 1\ \ \boxed{0}\ \ 1 \end{array}$$



Legend for stage i

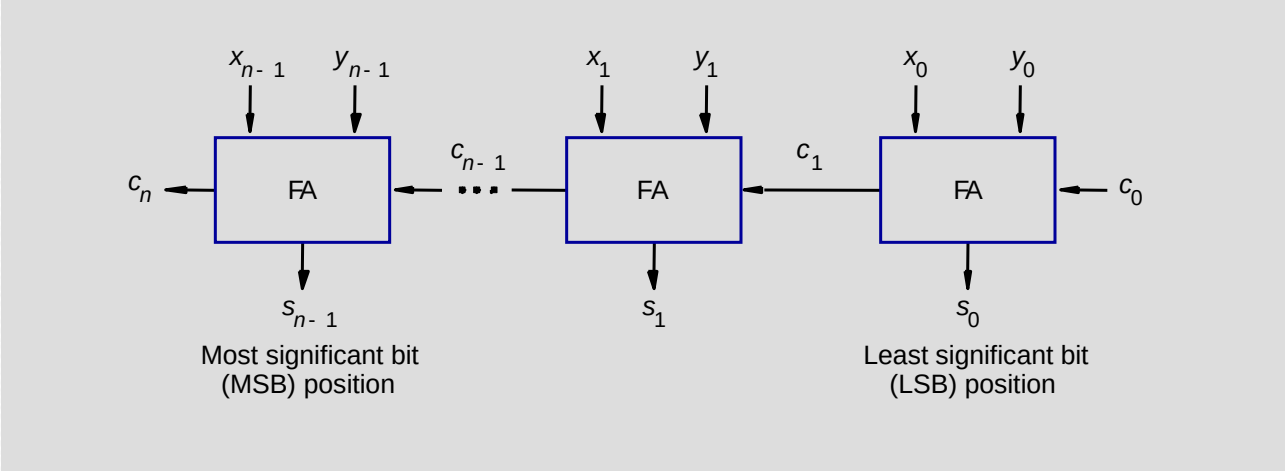
Addition logic for a single stage



Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

n-bit adder

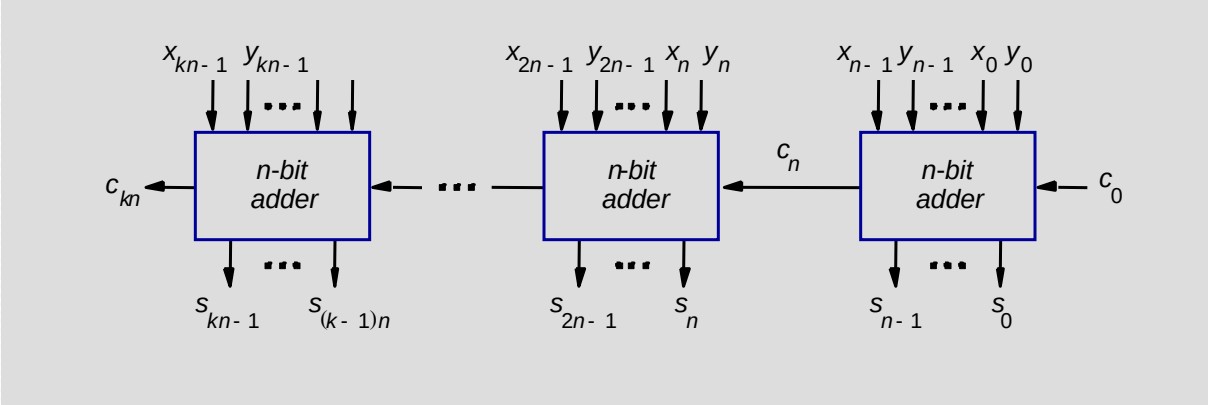
- Cascade *n* full adder (FA) blocks to form a *n*-bit adder.
- Carries propagate or ripple through this cascade, *n*-bit ripple carry adder.



Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

K n-bit adder

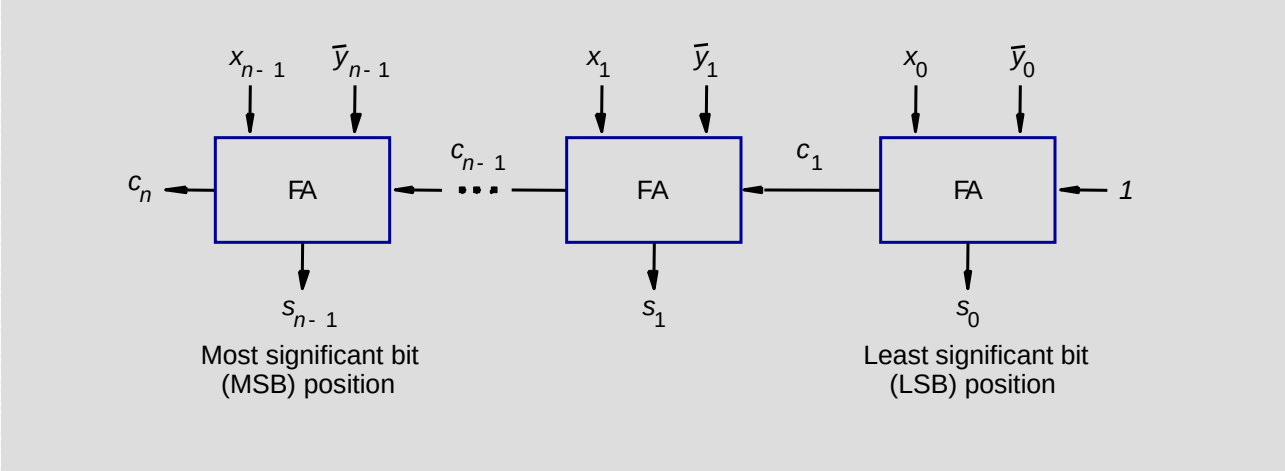
K n-bit numbers can be added by cascading k n-bit adders.



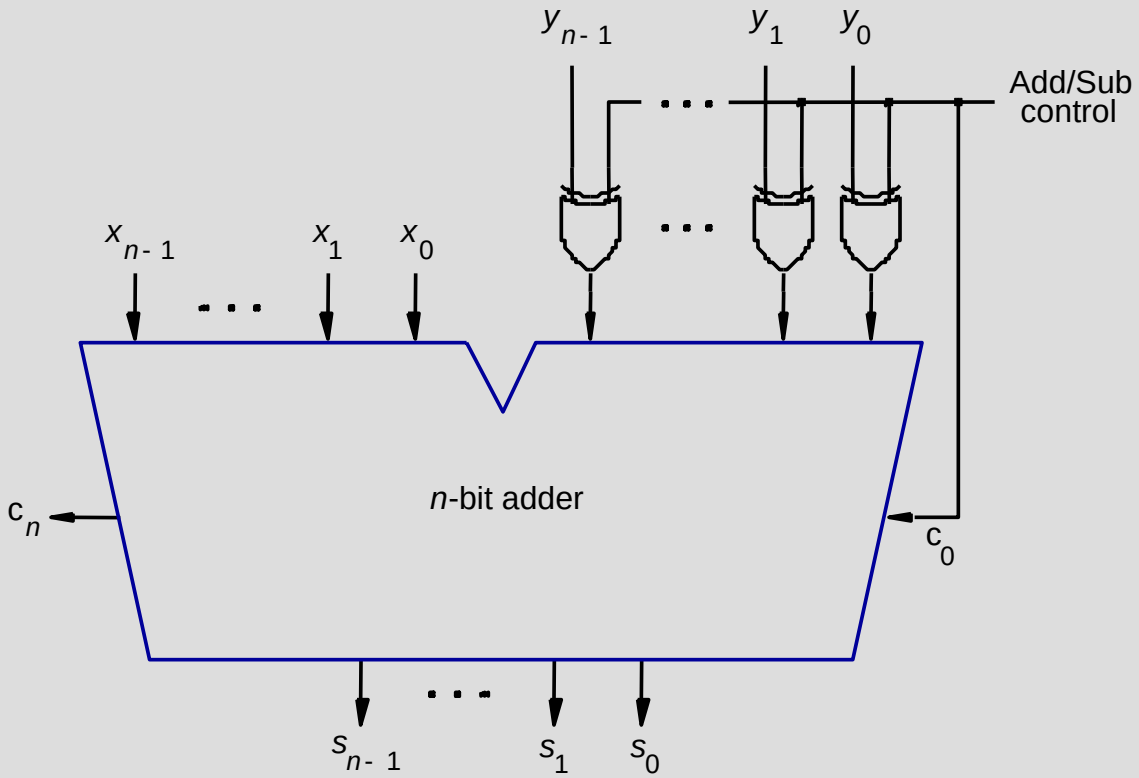
Each n-bit adder forms a block, so this is cascading of blocks.
Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

n-bit subtractor

- Recall $X - Y$ is equivalent to adding 2's complement of Y to X.
- 2's complement is equivalent to 1's complement + 1.
- $X - Y = X + \overline{Y} + 1$
- 2's complement of positive and negative numbers is computed similarly.



n-bit adder/subtractor (contd..)



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

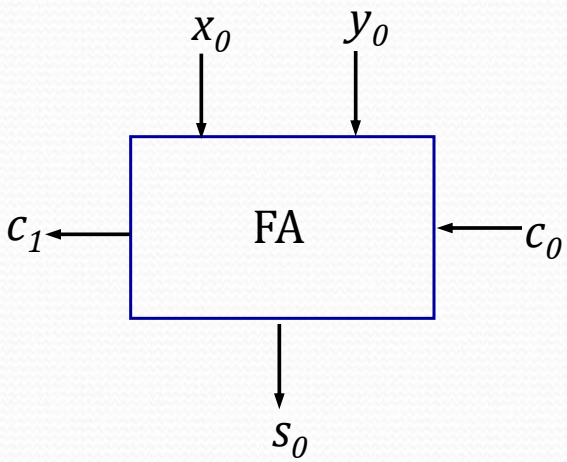
Detecting overflows

- Overflows can only occur when the sign of the two operands is the same.
- Overflow occurs if the sign of the result is different from the sign of the operands.
- Recall that the MSB represents the sign.
  - x_{n-1}, y_{n-1}, s_{n-1} represent the sign of operand *x*, operand *y* and result *s* respectively.
- Circuit to detect overflow can be implemented by the following logic expressions:

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

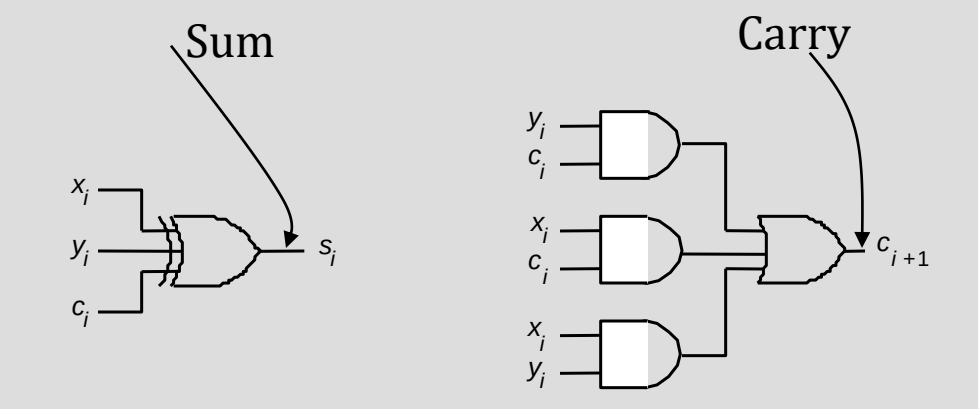
$$Overflow = c_n \oplus c_{n-1}$$

Computing the add time



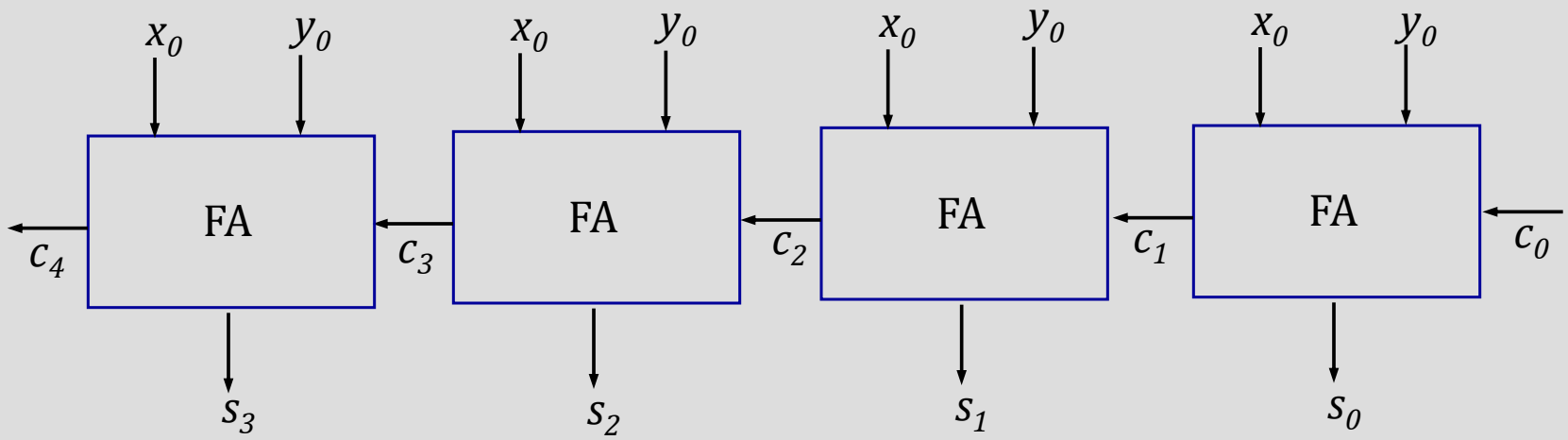
Consider 0th stage:

- c_1 is available after 2 gate delays.
- s_1 is available after 1 gate delay.



Computing the add time (contd..)

Cascade of 4 Full Adders, or a 4-bit adder



- s_0 available after 1 gate delays, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

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

Fast addition

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.

Carry lookahead

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

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

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

$$continuing$$

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

$$until$$

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

- 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.

Multiplication

Multiplication of unsigned numbers

				1	1	0	1	(13) Multiplicand M
				1	0	1	1	(11) Multiplier Q
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
	1	1	0	1				
1	0	0	0	1	1	1	1	(143) Product P

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

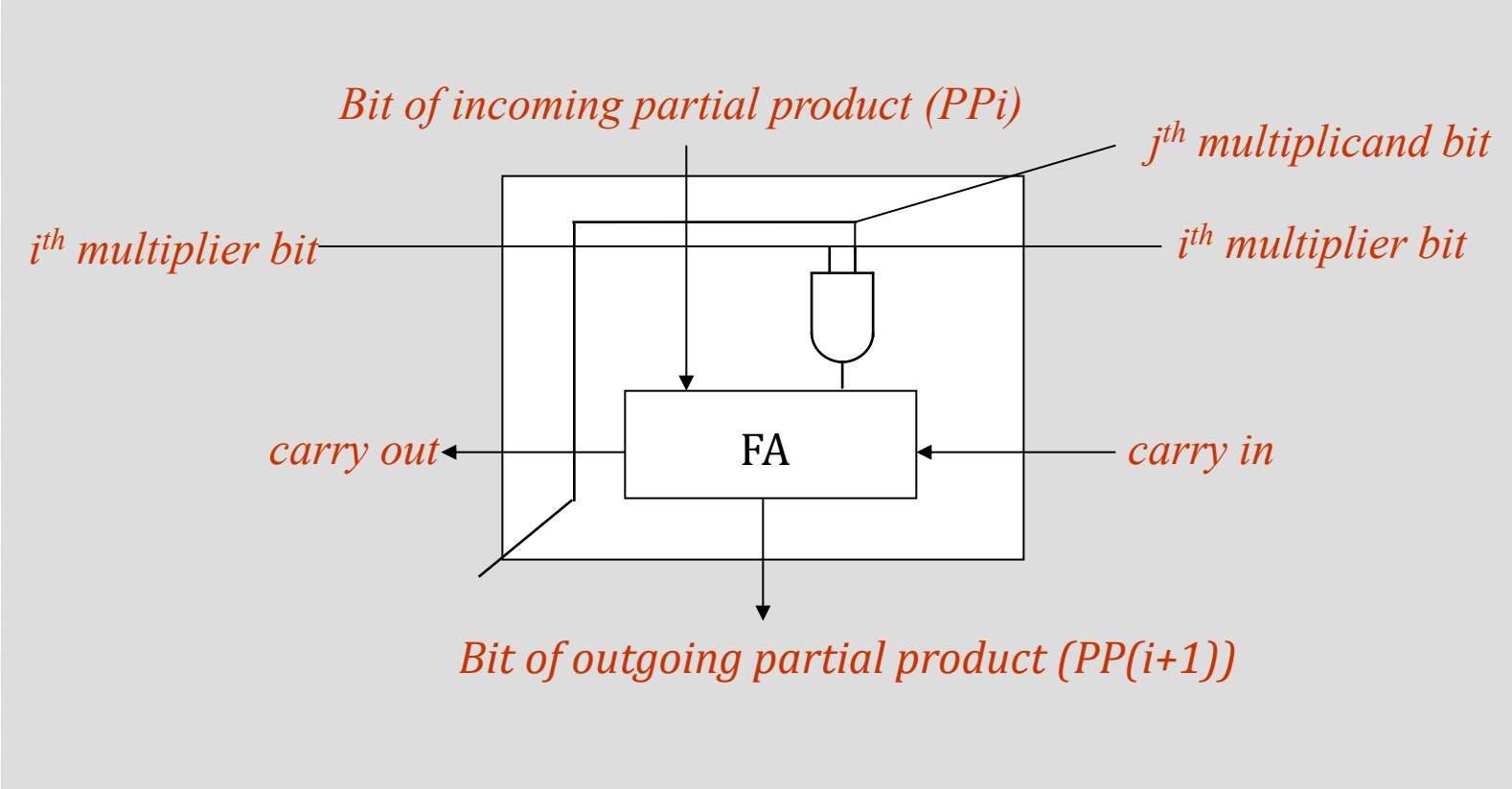
Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

Multiplication of unsigned numbers (contd..)

- We added the partial products at end.
  - Alternative would be to add the partial products at each stage.
- Rules to implement multiplication are:
  - If the ith bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product.
  - Hand over the partial product to the next stage
  - Value of the partial product at the start stage is 0.

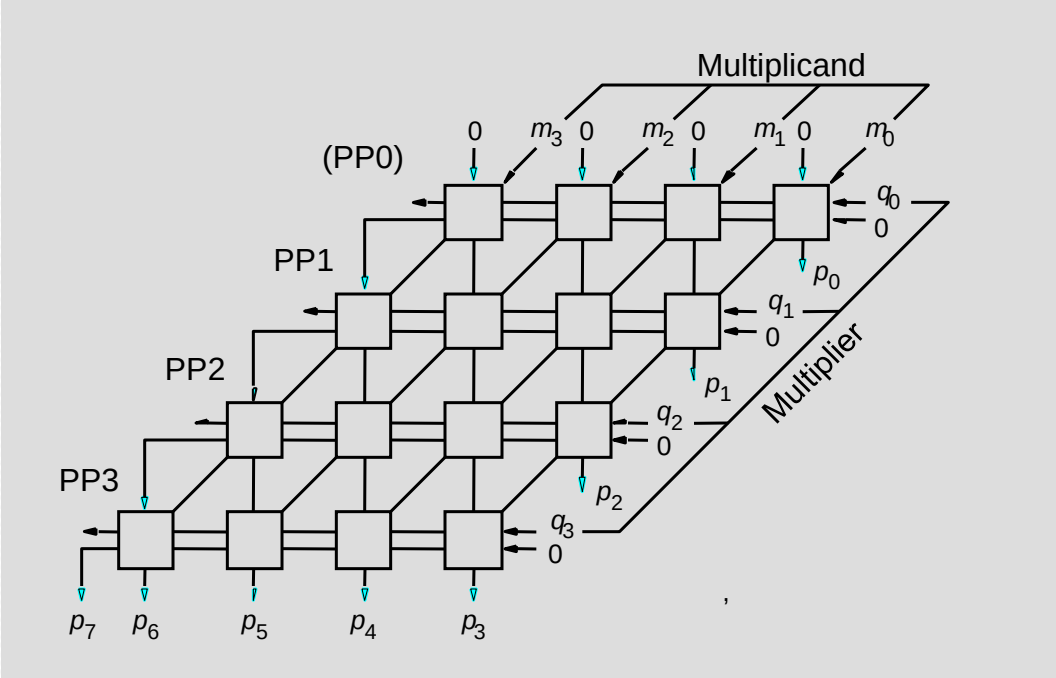
Multiplication of unsigned numbers

Typical multiplication cell



Combinatorial array multiplier

Combinatorial array multiplier



Product is: $p_7, p_6, .. p_0$

Multiplicand is shifted by displacing it through an array of adders.

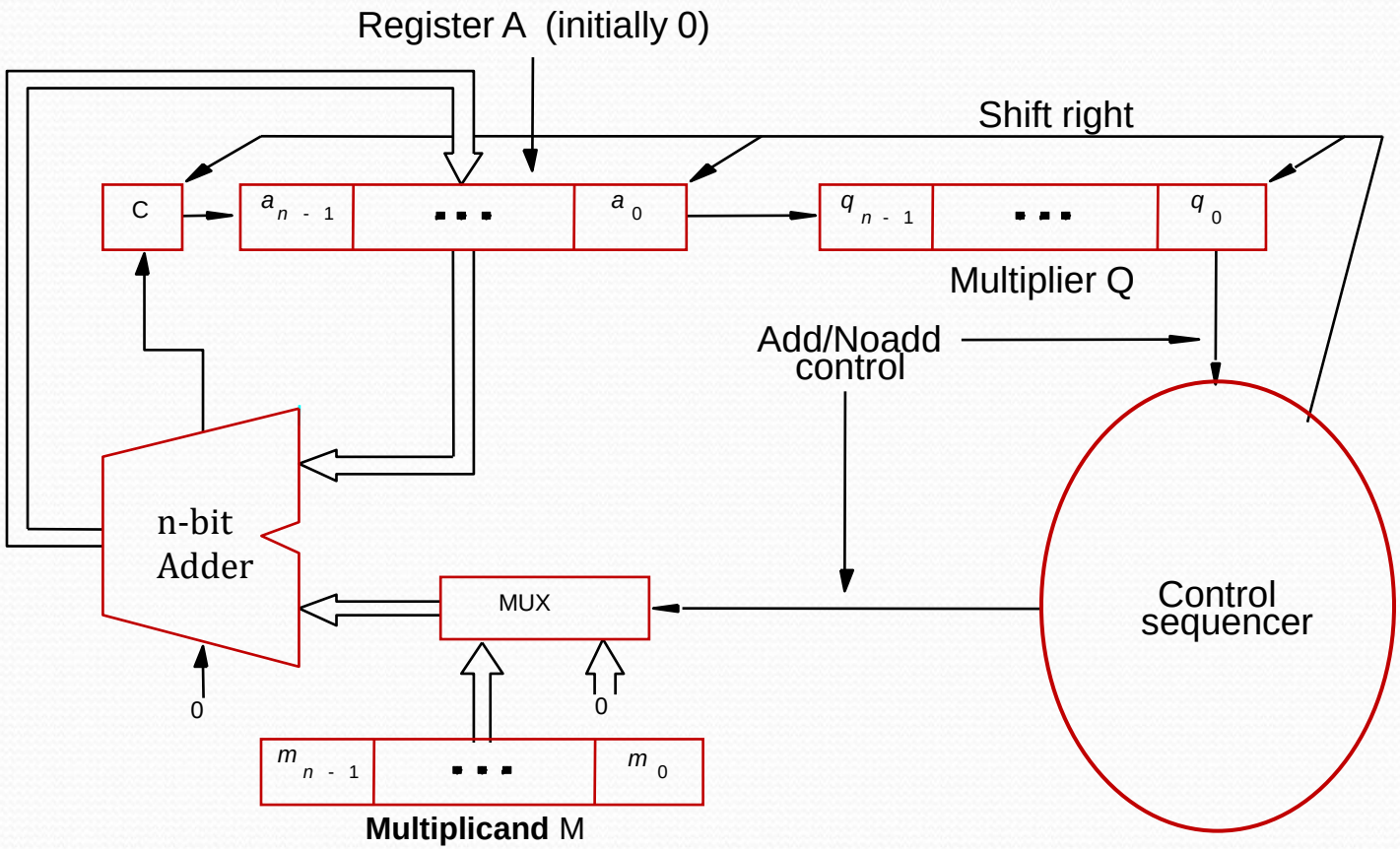
Combinatorial array multiplier (contd..)

- Combinatorial array multipliers are:
  - Extremely inefficient.
  - Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  - Perform only one function, namely, unsigned integer product.
- Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques requiring less combinational logic.

Sequential multiplication

- Recall the rule for generating partial products:
  - If the ith bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand has been shifted <u>left</u> when added to the partial product.
- However, adding a left-shifted multiplicand to an unshifted partial product is equivalent to adding an unshifted multiplicand to a right-shifted partial product.

Sequential Circuit Multiplier



Sequential multiplication (contd..)

C	A	Q		
	M: 1 1 0 1			Initial configuration
0	0 0 0 0	1 0 1 1		
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

Product

Signed Multiplication

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

Sign extension of negative multiplicand.

Signed Multiplication

- For a negative multiplier, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier.
- This is possible because complementation of both operands does not change the value or the sign of the product.
- A technique that works equally well for both negative and positive multipliers – Booth algorithm.

Booth Algorithm

- Consider in a multiplication, the multiplier is positive 0011110, how many appropriately shifted versions of the multiplicand are added in a standard procedure?

							0	1	0	1	1	0	1
							0	0	+1	+1	+1	+1	0
							0	0	0	0	0	0	0
						0	1	0	1	1	0	1	
					0	1	0	1	1	0	1		
				0	1	0	1	1	0	1			
			0	1	0	1	1	0	1				
		0	0	0	0	0	0	0					
	0	0	0	0	0	0	0						
0	0	0	1	0	1	0	1	0	0	0	1	1	0

Booth Algorithm

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit i-1	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Booth multiplier recoding table.

Booth Algorithm

- In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left. Ex:

0	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0
									⇓								
0	+1	-1	+1	0	-1	0	+1	0	0	-1	+1	-1	+1	0	-1	0	0

Booth recoding of a multiplier.

Booth Algorithm

	0	1	1	0	1	(+13)
x	1	1	0	1	0	(- 6)

$\Longrightarrow$

					0	1	1	0	1	
					0	-1	+1	-1	0	
0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	0	0	1	1		
0	0	0	0	1	1	0	1			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	1	0	1	1	0	0	1	0	(- 78)

Booth multiplication with a negative multiplier.

Booth Algorithm

- Best case – a long string of 1's (skipping over 1s)
- Worst case – 0's and 1's are alternating

Worst-case multiplier

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1

Ordinary multiplier

1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
0	-1	0	0	+1	-1	+1	0	-1	+1	0	0	0	-1	0	0

Good multiplier

0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1

- Refer to Booth algorithm and Modified booth algorithm illustration

Integer Division

Manual Division

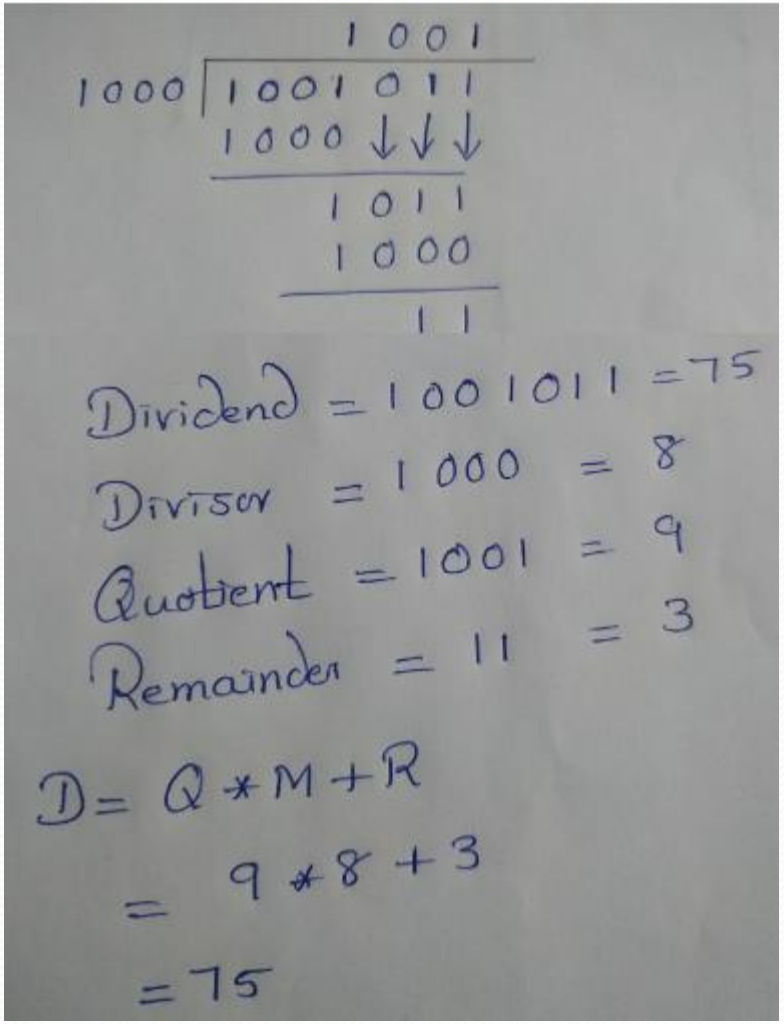
```
      21
13 ) 274
     26
     ---
      14
      13
     ---
       1
```

```
          10101
1101 ) 100010010
       1101
       ----
        10000
         1101
        -----
         1110
         1101
         ----
            1
```

Longhand division examples.

Simple Integer Division: Pencil-Paper Method

Let,
Dividend, D=1001011
Divisor, M=1000



Longhand Division Steps

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

Fixed Point Division

$C = a/b;$

- Restoring Division
- Non-Restoring Division

Circuit Arrangement

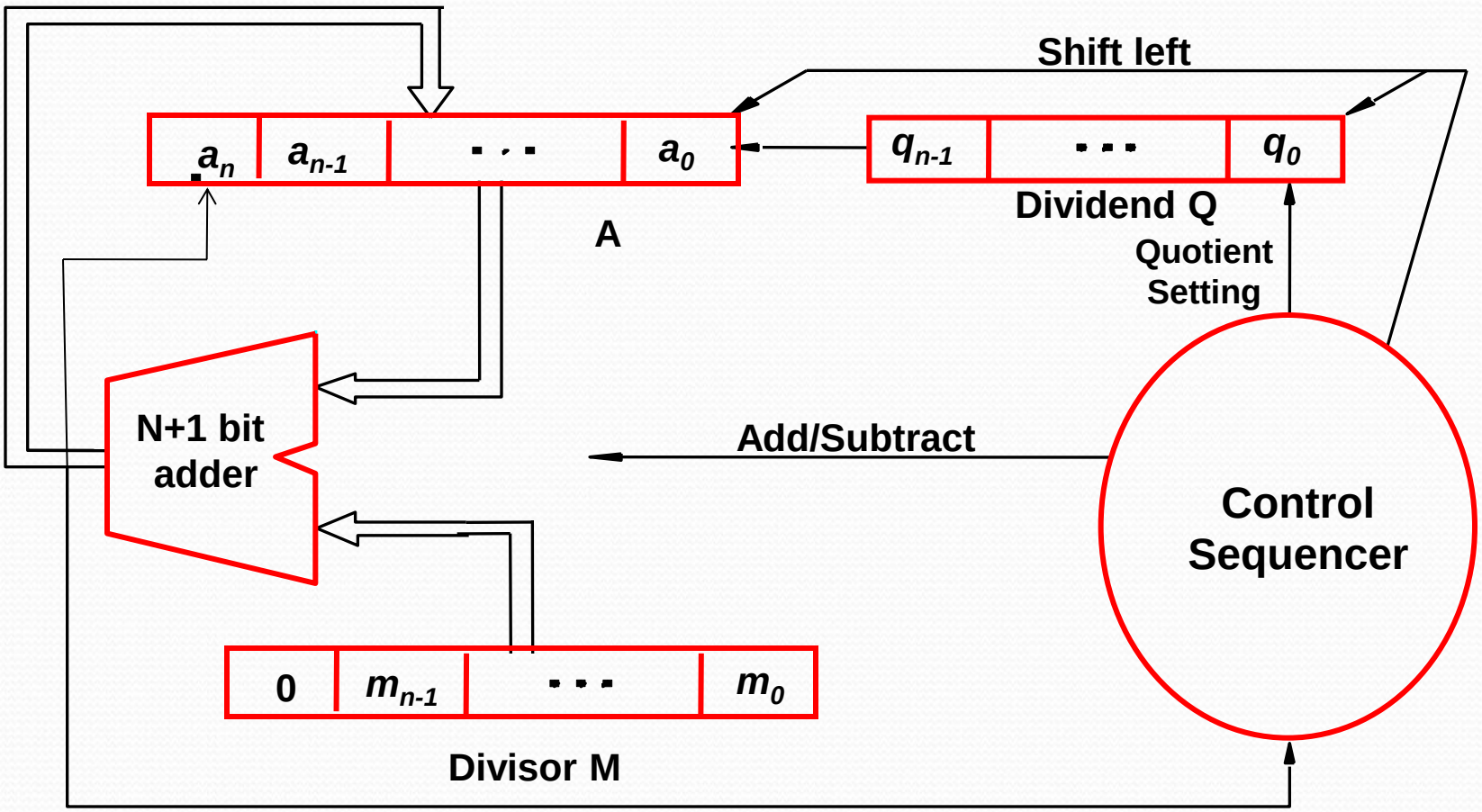
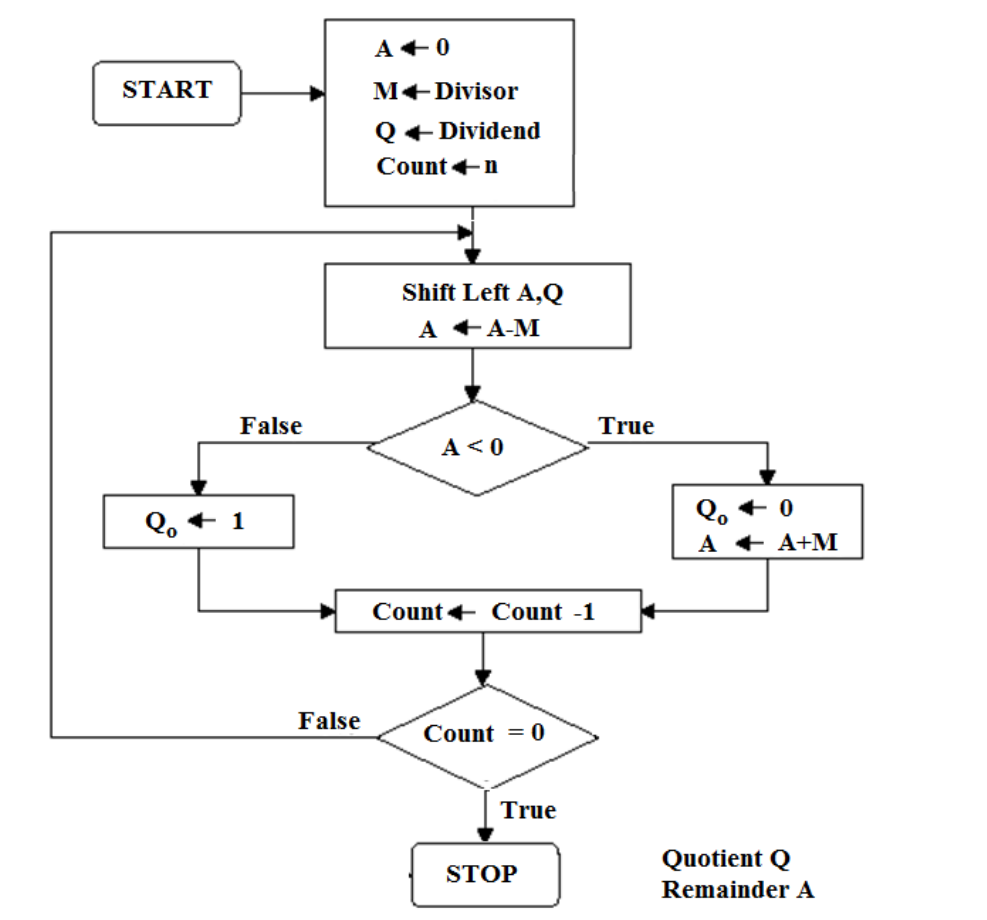


Figure 6.21. Circuit arrangement for binary division.

Restoring Division

- Shift A and Q left one binary position
- Subtract M from A, and place the answer back in A
- If the sign of A is 1, set q_0 to 0 and add M back to A (restore A); otherwise, set q_0 to 1
- Repeat these steps *n* times

Flow chart representation



Examples

```
          1 0
1 1 ) 1 0 0 0
        1 1
        -----
          1 0
```

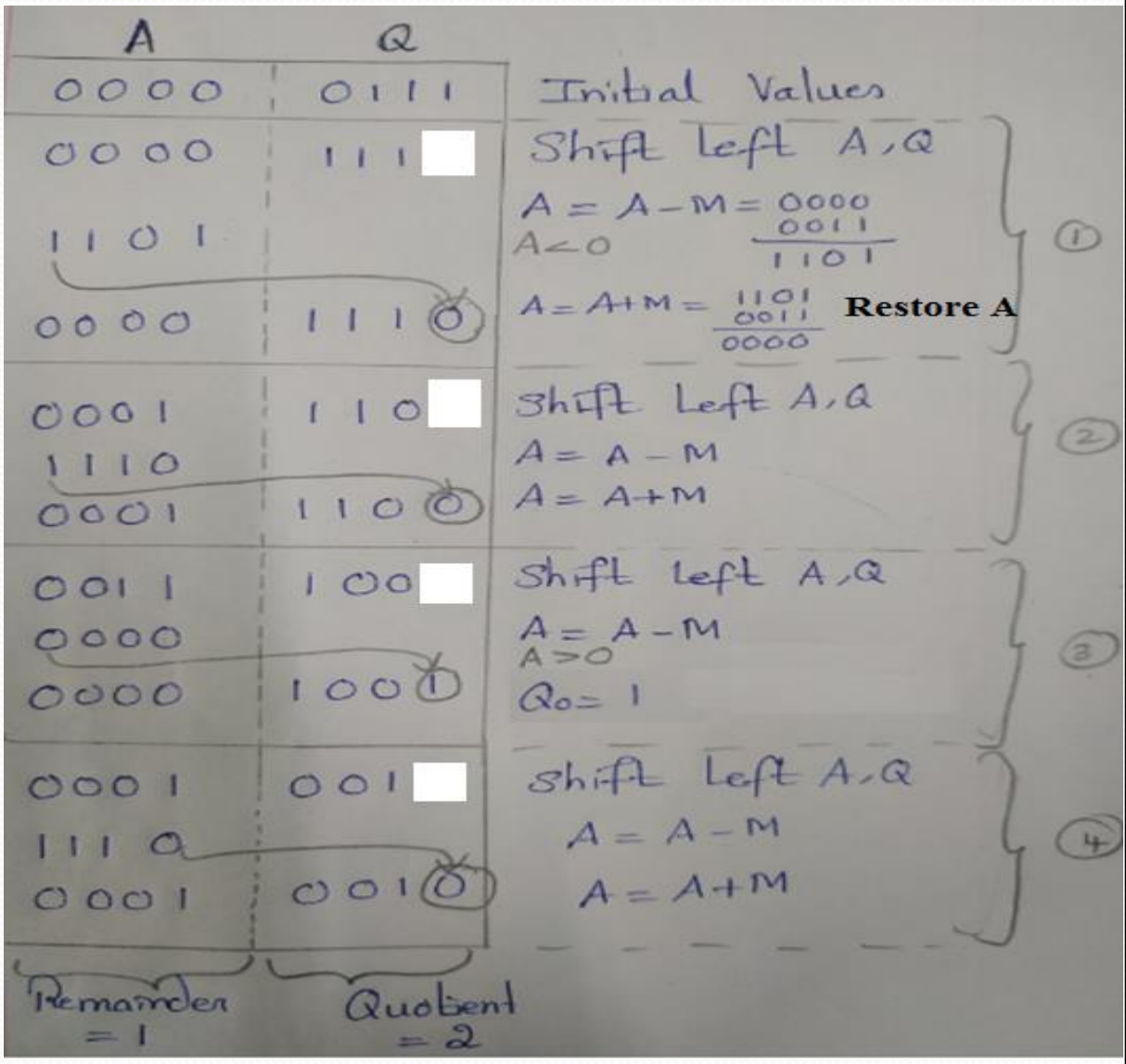
	Remainder	Quotient	
Initially	0 0 0 0 0	1 0 0 0	First cycle
	0 0 0 1 1		
Shift	0 0 0 0 1	0 0 0 □	
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 0		
Restore	1 1		
	0 0 0 0 1	0 0 0 0	
Shift	0 0 0 1 0	0 0 0 □	Second cycle
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 0 0	
Shift	0 0 1 0 0	0 0 0 □	Third cycle
Subtract	1 1 1 0 1		
Set q_0	0 0 0 0 1		
Shift	0 0 0 1 0	0 0 0 1	
Subtract	1 1 1 0 1	0 0 1 □	Fourth cycle
Set q_0	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 1 0	

Figure 6.22. A restoring-division example.

Example 7 ÷ 3

Q=7=0111
M=3=0011

Solution:
Quotient =0010=2
Remainder=0001=1



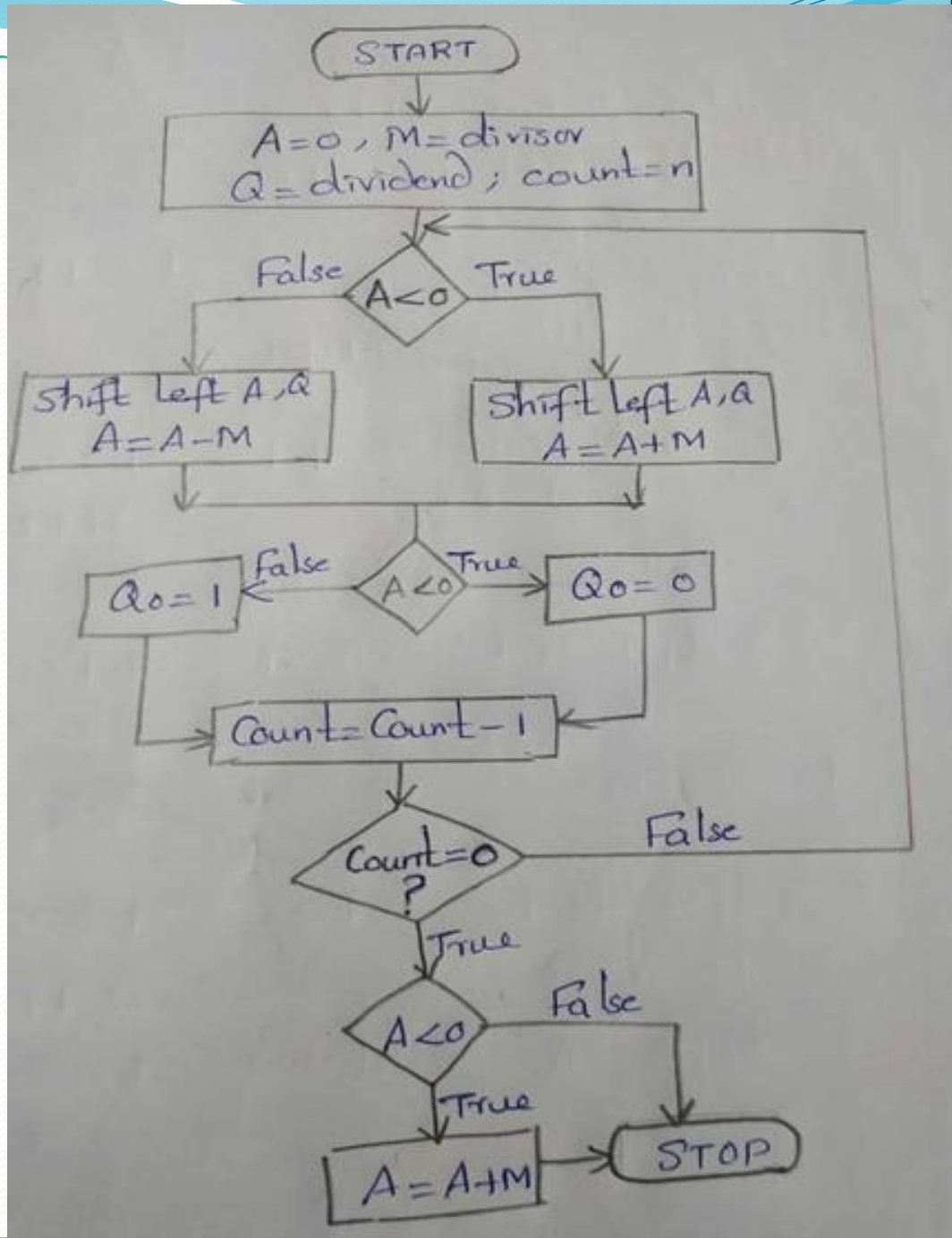
Non-restoring Division

Modifying the basic division algorithm by eliminating restoring step is non-restoring division

Algorithm

1. Start by initializing register A to 0 and repeat steps (2-4) n times
2. If A is positive,
   2.1 Shift A and Q left by one bit position
   2.2 Subtract M from A
3. If A is negative
   3.1 Shift A and Q left by one bit position
   3.2 Add M to A
4. If A is positive, set Q_0 to 1, else Q_0 to 0
5. If A is negative, add M to A as a final corrective step

Non-Restoring Division Flowchart



Examples

Initially	0	0	0	0	0	1	0	0	0	First cycle
	0	0	0	1	1					
Shift	0	0	0	0	1	0	0	0		
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	0	0	0	0	0	
Shift	1	1	1	0	0	0	0	0		Second cycle
Add	0	0	0	1	1					
Set q_0	1	1	1	1	1	0	0	0	0	
Shift	1	1	1	1	0	0	0	0		Third cycle
Add	0	0	0	1	1					
Set q_0	0	0	0	0	1	0	0	0	1	
Shift	0	0	0	1	0	0	0	1		Fourth cycle
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	1	0	0	1	0	

Quotient

	1	1	1	1	1
	0	0	0	1	1
Add	0	0	0	1	0

Remainder

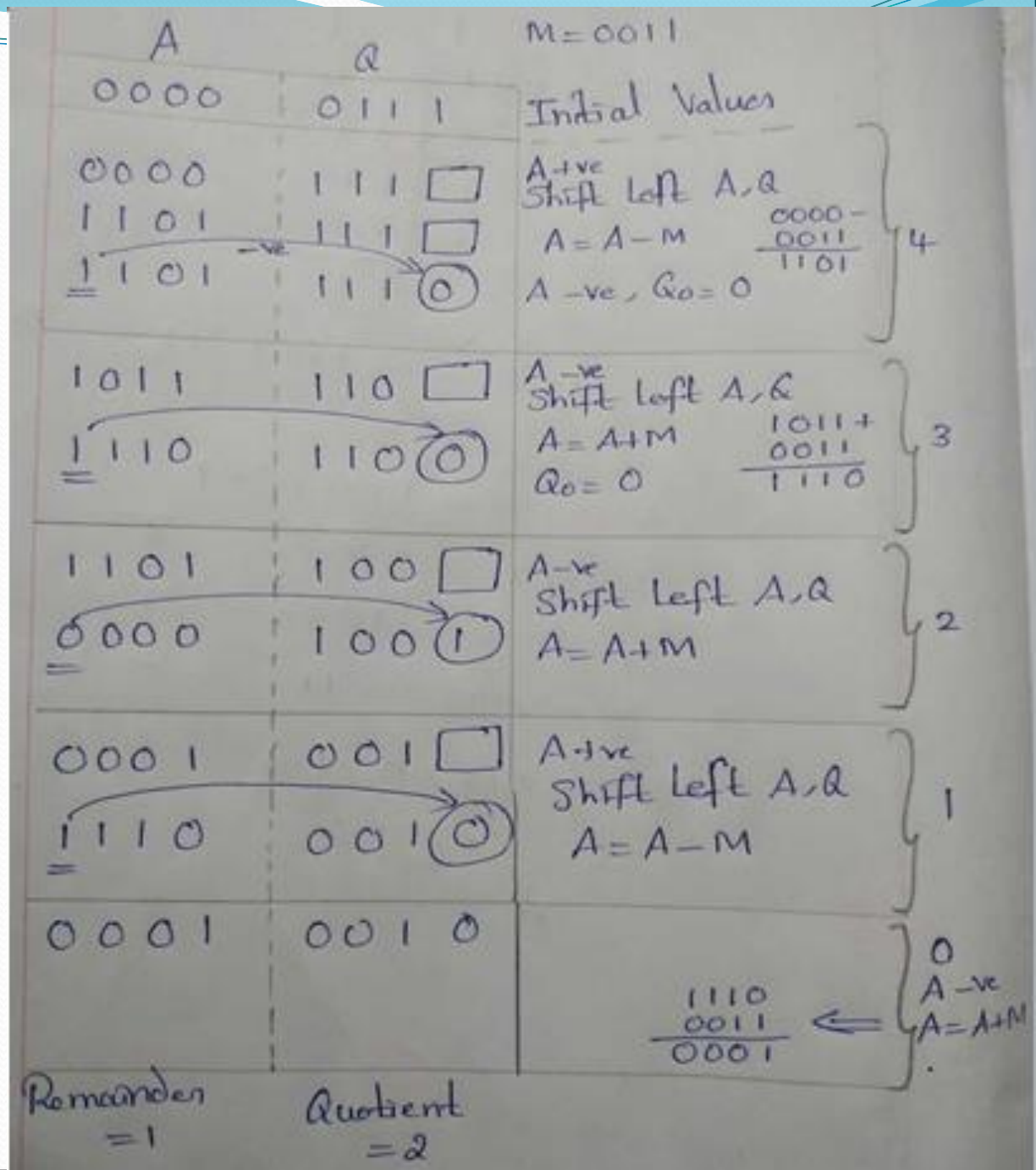
Restore remainder

A nonrestoring-division example.

Example 7 ÷ 3

Q=7=0111
M=3=0011

Solution:
Quotient =0010=2
Remainder=0001=1



A	Q		
		M=0011	
0000	0111	Initial Values	
0000	1111	A+ve Shift Left A,Q	4
1101	1111	A=A−M (0000 − 0011 = 1101)	
1101 (−ve)	1110	A −ve, Q_0=0	
1011	1110	A −ve Shift Left A,Q	3
1110	1110	A=A+M (1011 + 0011 = 1110)	
		Q_0=0	
1101	1001	A−ve Shift Left A,Q	2
0000	1001	A=A+M	
0001	0011	A+ve Shift Left A,Q	1
1110	0011	A=A−M	
0001	0010		0
		1110 + 0011 = 0001	A −ve A=A+M
Remainder =1	Quotient =2		

Floating-Point Numbers and Operations

Going Beyond Integers : Floating Point

Programming languages support numbers with real numbers i.e., fractions

These fractions are floating point numbers

Examples:
π=Pi value is 3.14159……
e= Euler Number is 2.71828…..
Nanoseconds in a day=86,400,000,000,000 ns
The last numbers is large integer that cannot fit in 32-bit integer.

Floating Point Numbers : Scientific Notation

We use *scientific notation* to represent fraction/floating point numbers. It has single digit to the left of decimal point.

A number 0.000000001_{ten} can be represented as $1.0_{ten} \times 10^{-9}$

Small number:
Mass of neutron=1.67×10^{-27} kg
Seconds in nanosecond=1.0×10^{-9} s

Large number:
Mass of earth=5.92×10^{24} kg
Seconds in a century=3.16×10^{9} s
Nanoseconds in a day=8.64×10^{13} ns

Scientific notation

Fixed point representation suffers from a drawback that the representation can only represent a finite range (and quite small) range of numbers.

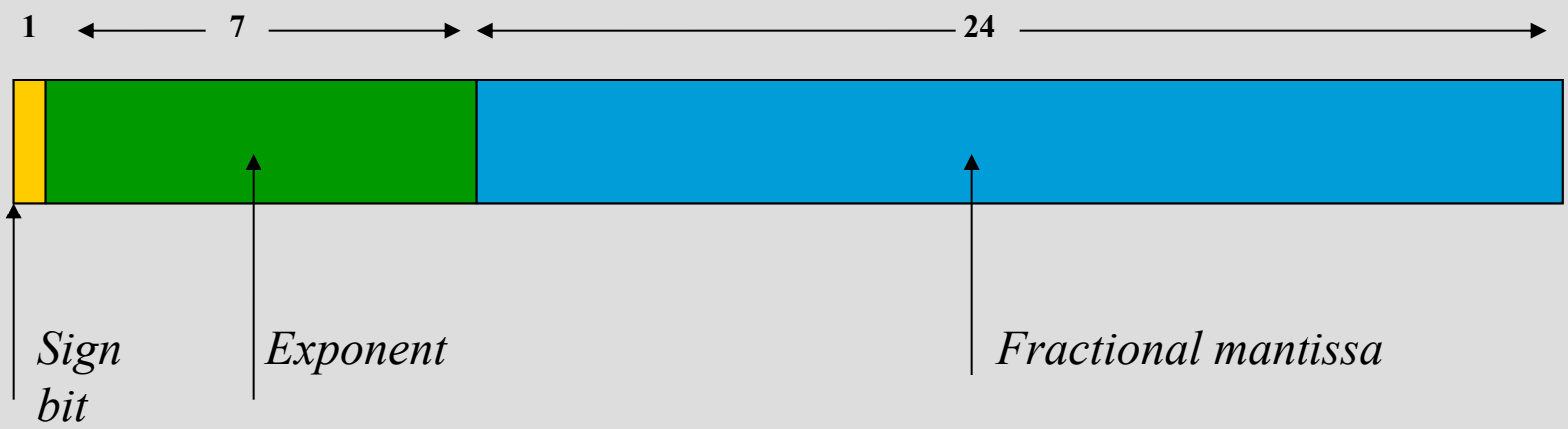
A more convenient representation is the scientific representation, where the numbers are represented in the form:

$$x = m_1.m_2m_3m_4 \times b^{\pm e}$$

Components of these numbers are:

Mantissa (m), implied base (b), and exponent (e)

A sample representation



- 24-bit mantissa with an implied binary point to the immediate left
- 7-bit exponent in 2's complement form, and implied base is 2.

Floating Point Representation: IEEE 754 Standard

A floating point number is represented using three fields: sign bit, exponent and fraction field

$$(-1)^S \times F \times 2^E$$

Sign	Exponent	Fraction
1 bit	8 bits	23 bits

The above is the IEEE 754 single precision floating point format

S for Sign: 1-bit
0 is positive and 1 is negative

E is exponent field: 8 bits
F is Fraction field : 23 bits

This exponent and fraction size give computer arithmetic an extraordinary range.

IEEE 754 Floating Point Standard Cntd…

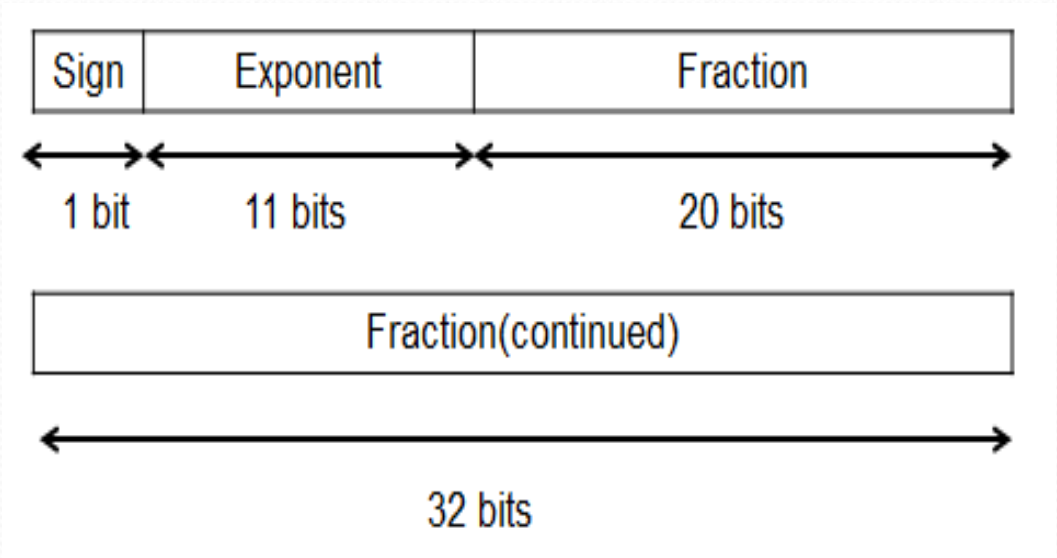
Possibility : *numbers(large)*

Overflow : Overflow interrupts can occur in floating-point arithmetic. Overflow means that the exponent is too large to be represented in the exponent field. In more specific if positive exponent becomes too large to fit in the exponent field.

Underflow: When a negative exponent becomes too large to fit in the exponent field.

IEEE 754 Floating Point Standard Cntd…

Underflow or overflow problems: solved using *exponent(larger)*
The following is IEEE 754 double precision floating point format.



This floating point number takes two 32-bit word. S is single bit sign, 11-bits exponent field, 52-bits fraction field.

Floating Point Numbers :Normalized Scientific Notation

Floating point numbers should be *normalized*. A number in scientific notation that has no leading 0s is called normalized number.

One non-zero digit should appear before the decimal point. In decimal, this digit should be from 1 to 9. In binary, this digit should be 1.

Example:

1.0×10^{-9} is normalized scientific notation

0.1×10^{-8} is not normalized scientific notation

10.0×10^{-10} is not normalized scientific notation

The procedure for normalizing a floating point number is:

```
Do (until MSB of mantissa = = 1)
    Shift the mantissa left (or right)
    Decrement (increment) the exponent by 1
end do
```

Biased Exponent Representation

IEEE 754 : exponent -biased representation

For single precision, IEEE 754 uses a bias of 127.

General representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

So -1 is represented as $-1+127=126_{10}= 0111\ 1110_2$(Exponent)
+1 is represented as $1+127=128_{10}= 1000\ 0000_2$

In general, excess-p coding is represented as:

$$E' = E_{true} + p$$

For double precision, IEEE 754 uses a bias of 1023.

Exercise 1:

Question: Show the IEEE 754 binary representation of the number -0.75_{10} in single and double precision.

Solution:

$\mathbf{-0.75_{10}}$ is represented in binary as $\mathbf{-.11_2}$

Scientific Notation value= $\mathbf{-.11_2 \times 2^0}$

Normalized Scientific Notation value= $\mathbf{-1.1_2 \times 2^{-1}}$

General representation for a single precision number is

$$\mathbf{(-1)^S \times (1 + Fraction) \times 2^{(Exponent - 127)}}$$

Similarly the given number in general representation is

$$\mathbf{(-1)^1 \times (1 + .1) \times 2^{(126-127)}}$$

The single precision binary representation of -0.75_{10} is then

1	01111110	10000000000000000000000

$0.75 * 2 = 1.50 \rightarrow (1)$
$0.5 * 2 = 1.00 \rightarrow (1)$

So, $-0.75_{10} \rightarrow (-0.11) * 2^0$

Exercise 2:

Question: Show the IEEE 754 binary representation of the number 9.75_{10} in single precision.

Solution:

$\mathbf{9.75_{10}}$ is represented in binary as $\mathbf{1001.11_2}$

Normalized Scientific Notation value = $\mathbf{1.00111_2 \times 2^3}$

General representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

Similarly the given number in general representation is

$$(-1)^0 \times (1 + .00111) \times 2^{(130-127)}$$

The single precision binary representation of 9.75_{10} is then

0	10000010	00111000000000000000000

Exercise 3: Find the decimal value for the given IEEE 754 binary representation

Question:

What number is represented by the single-precision float

11000000101000...00

Solution:

$S = 1$

Exponent $= 10000001_2 = 129$

Fraction $= 01000...00_2$

Decimal, $x = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$

$= (-1) \times 1.25 \times 2^2$

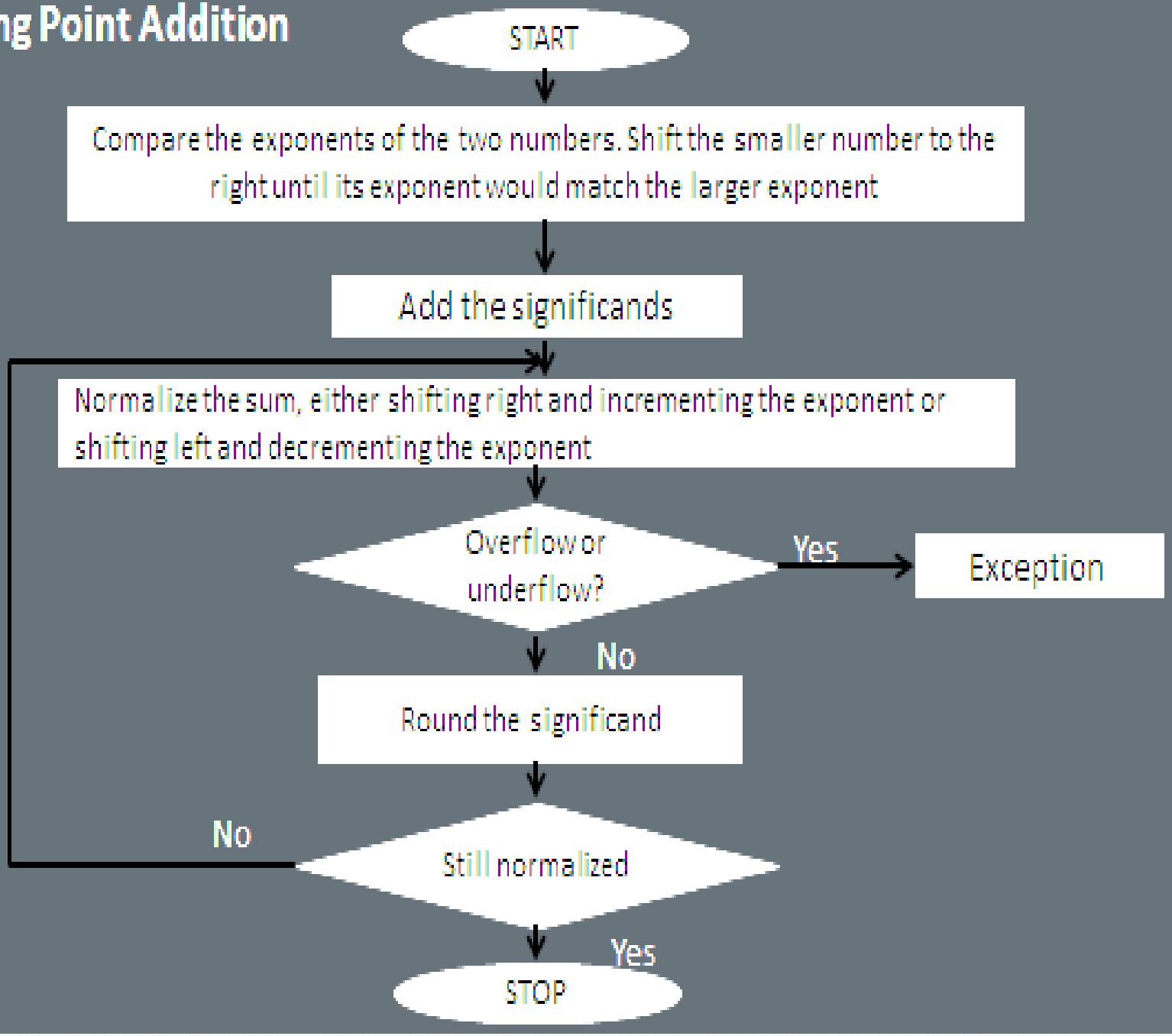
$= -5.0_{10}$

Floating Point Arithmetic: ADD/SUB rule

- Choose the number with the smaller exponent.
- Shift its mantissa until the exponents of both the numbers are equal.
- Add or subtract the mantissas.
- Determine the sign of the result.
- Normalize the result if necessary and truncate/round to the number of mantissa bits.

Note: This does not consider the possibility of overflow/underflow.

Floating Point Addition



Floating Point Addition

Question: Consider a 4-digit decimal: $9.999 \times 10^1 + 1.50 \times 10^{-1}$. Assume we can store only four decimal digits of the significand and two decimal digits of the exponent.

Solution:

- 1. Align decimal points to match the exponents. Shift number with smaller exponent

  $9.999_{10} \times 10^1 + 0.015_{10} \times 10^1$

- 2. Addition of significands

  $$\begin{array}{r} 9.999_{10} \\ +\ \ 0.015_{10} \\ \hline 10.014_{10} \end{array}$$

  Hence the sum is $10.014_{10} \times 10^1$

- 3. Adjust the sum to normalized scientific notation

  1.0014×10^2

- 4. Since we assumed that the significand can be only four digits long, we should round the answer.

Answer: 1.002×10^2

Floating Point Addition

Question: Add 0.5_{10} and -0.4375_{10} in binary.

Answer:

Convert decimal to normalized scientific notation of binary

0.5_{10} and $-0.4375_{10} = 1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

- 1. Align binary points to match the exponents

  Shift number with smaller exponent

  $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

- 2. Addition of significands

  $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

- 3. Normalize result, Adjust the sum to normalized scientific notation

  $1.000_2 \times 2^{-4}$, with no over/underflow

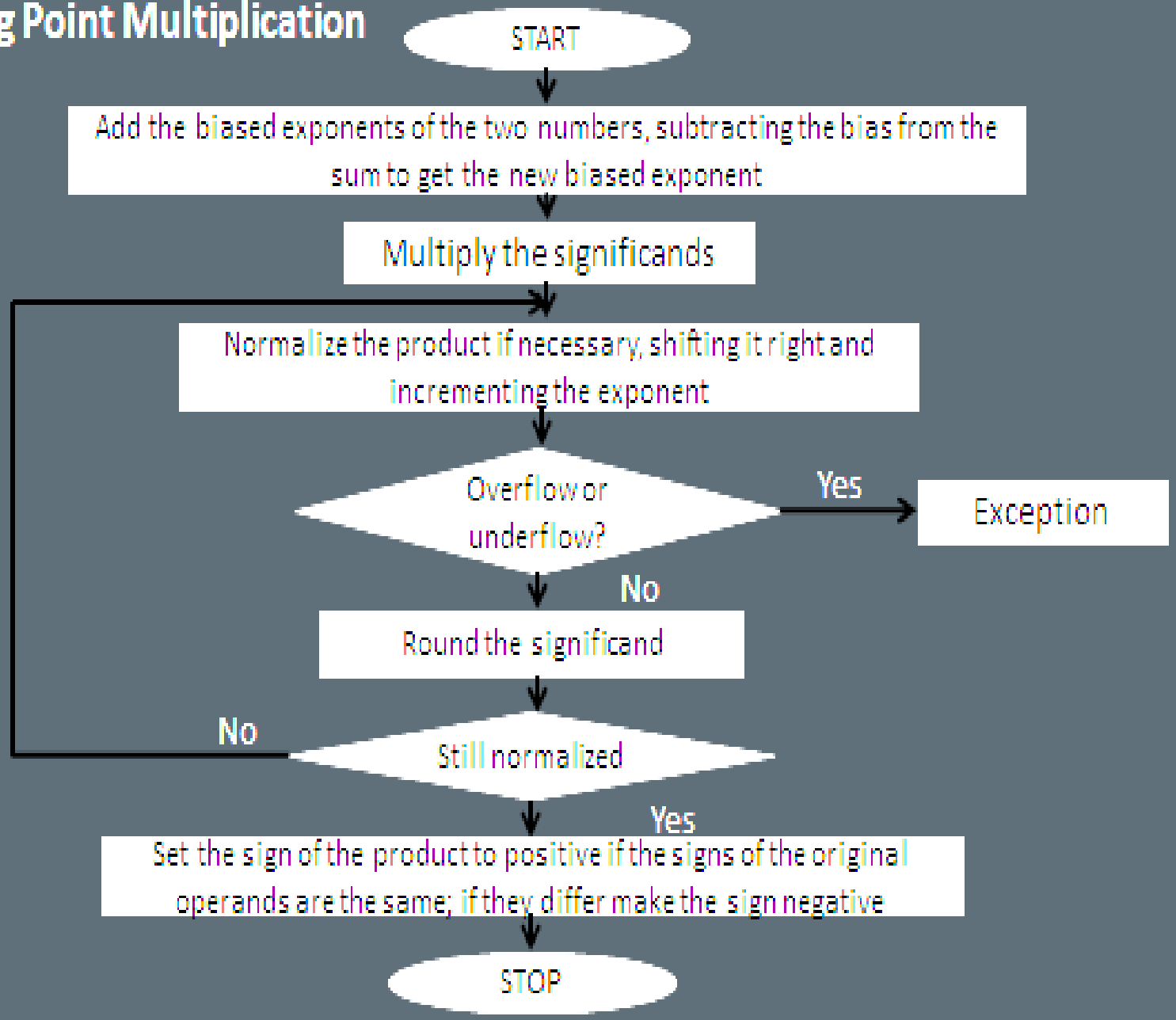
- 4. Round the sum if necessary

The sum is $\mathbf{1.000_2 \times 2^{-4}}$ (no change) $= 0.0625_{10}$

Floating Point Multiplication

- Add the exponents.
- Subtract the bias.
- Multiply the mantissas and determine the sign of the result.
- Normalize the result (if necessary).
- Truncate/round the mantissa of the result.

Floating Point Multiplication



Floating Point Multiplication

Question: Consider a 4-digit decimal $1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$

Solution:

- 1. Add exponents

  New exponent = 10 + –5 = 5

- 2. Multiply significands

  $1.110_{10} \times 9.200_{10} = 10.212_{10} \Rightarrow 10.212 \times 10^5$

- 3. Normalize the result

  1.0212×10^6

- 4. Round and renormalize if necessary

  1.021×10^6 (Significand only four digit long)

- 5. Determine sign of result from signs of operands

  $+1.021 \times 10^6$

Floating Point Multiplication

Question: Multiply in binary $0.5_{10} \times -0.4375_{10}$

Solution:

Convert decimal to normalized scientific notation of binary

0.5_{10} and $-0.4375_{10} = 1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

- 1. Add exponents

  Unbiased: $-1 + -2 = -3$

  Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

- 2. Multiply significands

  $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$

- 3. Normalize result & check for over/underflow

  $1.110_2 \times 2^{-3}$ (no change) with no over/underflow

- 4. Round and renormalize if necessary

  $1.110_2 \times 2^{-3}$ (no change)

- 5. Determine sign: +ve x –ve ⇒ –ve

  $\mathbf{-1.110_2 \times 2^{-3} = -0.21875}$

Floating point arithmetic: DIV rule

- Subtract the exponents
- Add the bias.
- Divide the mantissas and determine the sign of the result.
- Normalize the result if necessary.
- Truncate/round the mantissa of the result.

Note: Multiplication and division does not require alignment of the mantissas the way addition and subtraction does.

Guard bits

- While adding two floating point numbers with 24-bit mantissas, we shift the mantissa of the number with the smaller exponent to the right until the two exponents are equalized.
- This implies that mantissa bits may be lost during the right shift (that is, bits of precision may be shifted out of the mantissa being shifted).
- To prevent this, floating point operations are implemented by keeping guard bits, that is, extra bits of precision at the least significant end of the mantissa.
- The arithmetic on the mantissas is performed with these extra bits of precision.
- After an arithmetic operation, the guarded mantissas are:
  - Normalized (if necessary)
  - Converted back by a process called truncation/rounding to a 24-bit mantissa.

Truncation/rounding

- Straight chopping:
  - The guard bits (excess bits of precision) are dropped.
- Von Neumann rounding:
  - If the guard bits are all 0, they are dropped.
  - However, if any bit of the guard bit is a 1, then the LSB of the retained bit is set to 1.
- Rounding:
  - If there is a 1 in the MSB of the guard bit then a 1 is added to the LSB of the retained bits.

Rounding

- Rounding is evidently the most accurate truncation method.
- However,
  - Rounding requires an addition operation.
  - Rounding may require a renormalization, if the addition operation de-normalizes the truncated number.
- IEEE uses the rounding method.