William Stallings
Computer Organization and Architecture
6th Edition

Chapter 9

Computer Arithmetic

2.1 **Introduction to Arithmetic and Logical unit, Computer Arithmetic: Fixed and Floating point numbers, Signed numbers, Integer Arithmetic, 2’s Complement arithmetic**

2.2 **Booth’s Recoding and Booth’s algorithm for signed multiplication, Restoring division and non-restoring division algorithms**

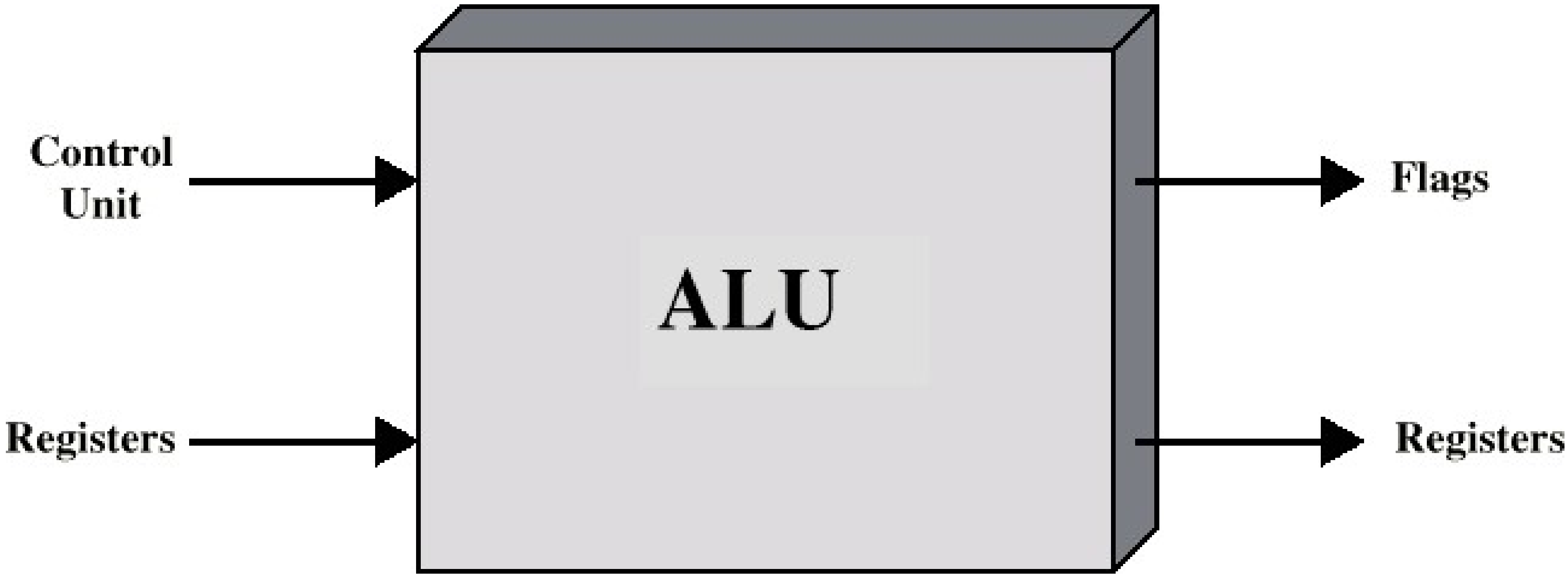
2.3 **IEEE floating point number representation and operations: Addition. Subtraction, Multiplication and Division. IEEE standards for Floating point representations :Single Precision and Double precision Format**

Arithmetic & Logic Unit

- Does the calculations
- Everything else in the computer is there to

  service this unit
- Handles integers
- May handle floating point (real) numbers
- May be separate FPU-floating-point unit (FPU), which operates on floating point numbers. (maths co-processor)

ALU Inputs and Outputs



- Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored in registers.
- These registers are temporary storage locations within the processor that are connected by signal paths to the ALU.
- The ALU may also set flags as the result of an operation.
- The flag values are also stored in registers within the processor.
- The processor provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

Integer representation

In the binary number system,[1] arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period, or **radix point**.

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, including

$$\begin{aligned}00000000 &= 0\\ 00000001 &= 1\\ 00101001 &= 41\\ 10000000 &= 128\\ 11111111 &= 255\end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2}\ldots a_1a_0$ is interpreted as an unsigned integer A, its value is

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

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n - 1$ bits hold the magnitude of the integer.

$+ 18$	$= 00010010$	
$- 18$	$= 10010010$	(sign magnitude)

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases} \quad \textbf{(9.1)}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. This should become clear in the discussion in Section 9.3. Another drawback is that there are two representations of 0:

$+0_{10}$	$= 00000000$	
-0_{10}	$= 10000000$	(sign magnitude)

Twos Complement Representation

Like sign magnitude, **twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative.**
It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted.
it facilitates the most important arithmetic operations, addition and subtraction. For this reason, it is almost universally used as the processor representation for integers

Table 9.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

Twos Complement $$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Table 9.2 Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation
+8	—	—
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
−0	1000	—
−1	1001	1111
−2	1010	1110
−3	1011	1101
−4	1100	1100
−5	1101	1011
−6	1110	1010
−7	1111	1001
−8	—	1000

-128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

$-128 \quad +2 \quad +1 = -125$

(b) Convert binary 10000011 to decimal

-128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

$-120 = -128 \quad +8$

(c) Convert decimal -120 to binary

Figure 9.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−32,658	=	1000000001101110	(twos complement, 16 bits)

The next to last line is easily seen using the value box of Figure 9.2. The last line can be verified using Equation (9.2) or a 16-bit value box.

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

-18	$=$	11101110	(twos complement, 8 bits)
-18	$=$	1111111111101110	(twos complement, 16 bits)

Fixed point representation

the radix point (binary point) is fixed and assumed to be to the right of the rightmost digit.
The programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location.

Addition and Subtraction

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

Figure 9.3 Addition of Numbers in Twos Complement Representation

- On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

SUBTRACTION RULE: To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Thus, subtraction is achieved using addition, as illustrated in Figure 9.4. The last two examples demonstrate that the overflow rule still applies.

```
         0010 =  2
        +1001 = –7
         1011 = –5

(a) M = 2 = 0010
    S = 7 = 0111
   –S =     1001
```

```
         0101 =  5
        +1110 = –2
        10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   –S =     1110
```

```
         1011 = –5
        +1110 = –2
        11001 = –7

(c) M = –5 = 1011
    S =  2 = 0010
   –S =      1110
```

```
         0101 = 5
        +0010 = 2
         0111 = 7

(d) M =  5 = 0101
    S = –2 = 1110
   –S =      0010
```

```
         0111 = 7
        +0111 = 7
         1110 = Overflow

(e) M =  7 = 0111
    S = –7 = 1001
   –S =      0111
```

```
         1010 = –6
        +1100 = –4
        10110 = Overflow

(f) M = –6 = 1010
    S =  4 = 0100
   –S =      1100
```

Figure 9.4 Subtraction of Numbers in Twos Complement Representation (M − S)

Addition and Subtraction

- Normal binary addition
- Monitor sign bit for overflow
- Take twos compliment of substahend and add to minuend
  - i.e. a - b = a + (-b)

- So we only need addition and complement circuits

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	0, Carry 1
1	1	1,Carry 1

Example of 2's Compliment

```
  5 = 00000101
      ↓↓↓↓↓↓↓↓      Complement Digits
      11111010
            +1      Add 1
      --------
 -5 = 11111011
```

```
0 1 1 0  1 1 1 0   ← Original binary value

1 0 0 1  0 0 0 1   ← 1's complement

  1 0 0 1  0 0 0 1
+                1
  ---------------
  1 0 0 1  0 0 1 0 ← 2's complement
```

```
-13 = 11110011
      ↓↓↓↓↓↓↓↓      Complement Digits
      00001100
            +1      Add 1
      --------
 13 = 00001101
```

Find 2's compliment

1000

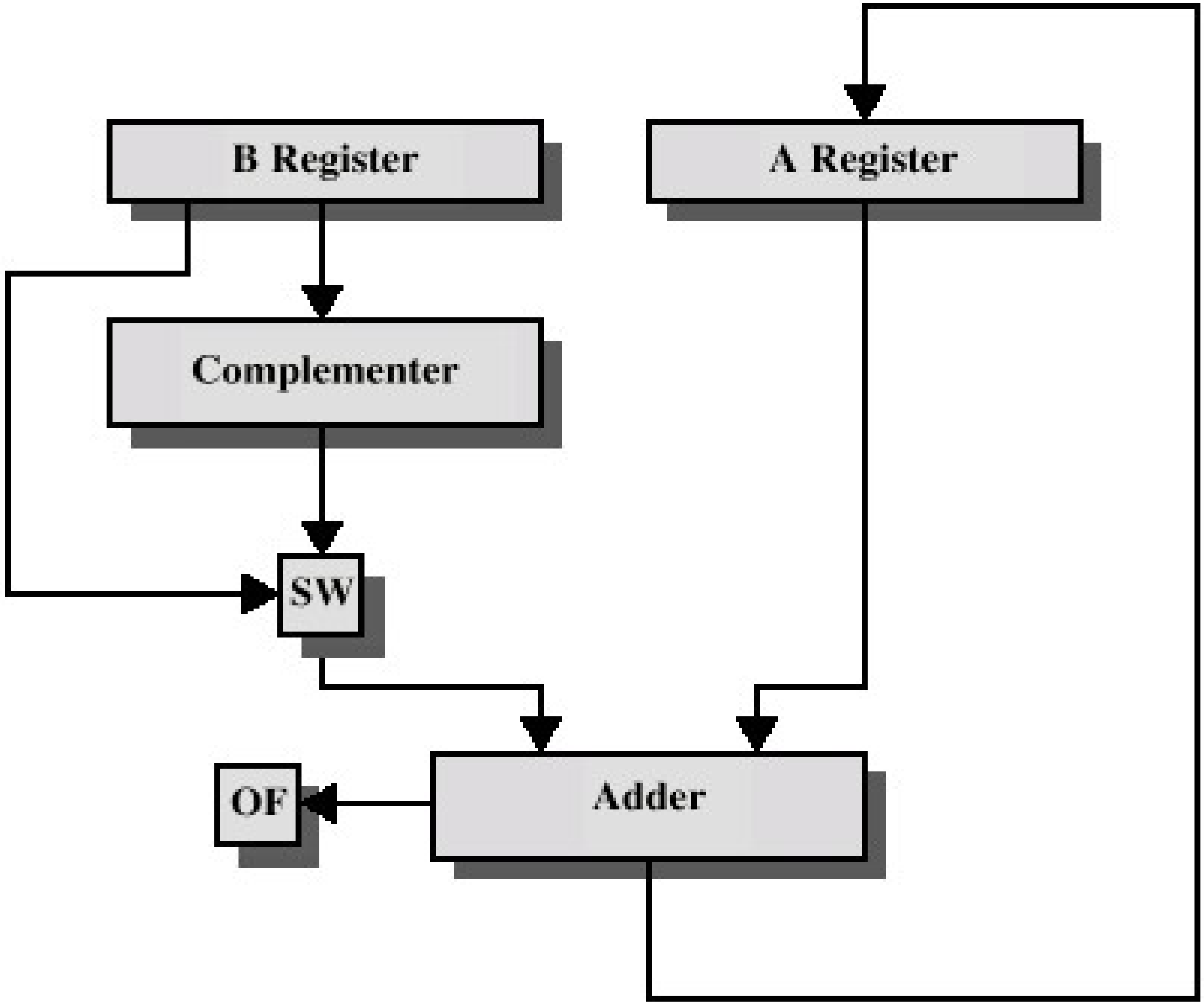
0100

111001

101010

100000

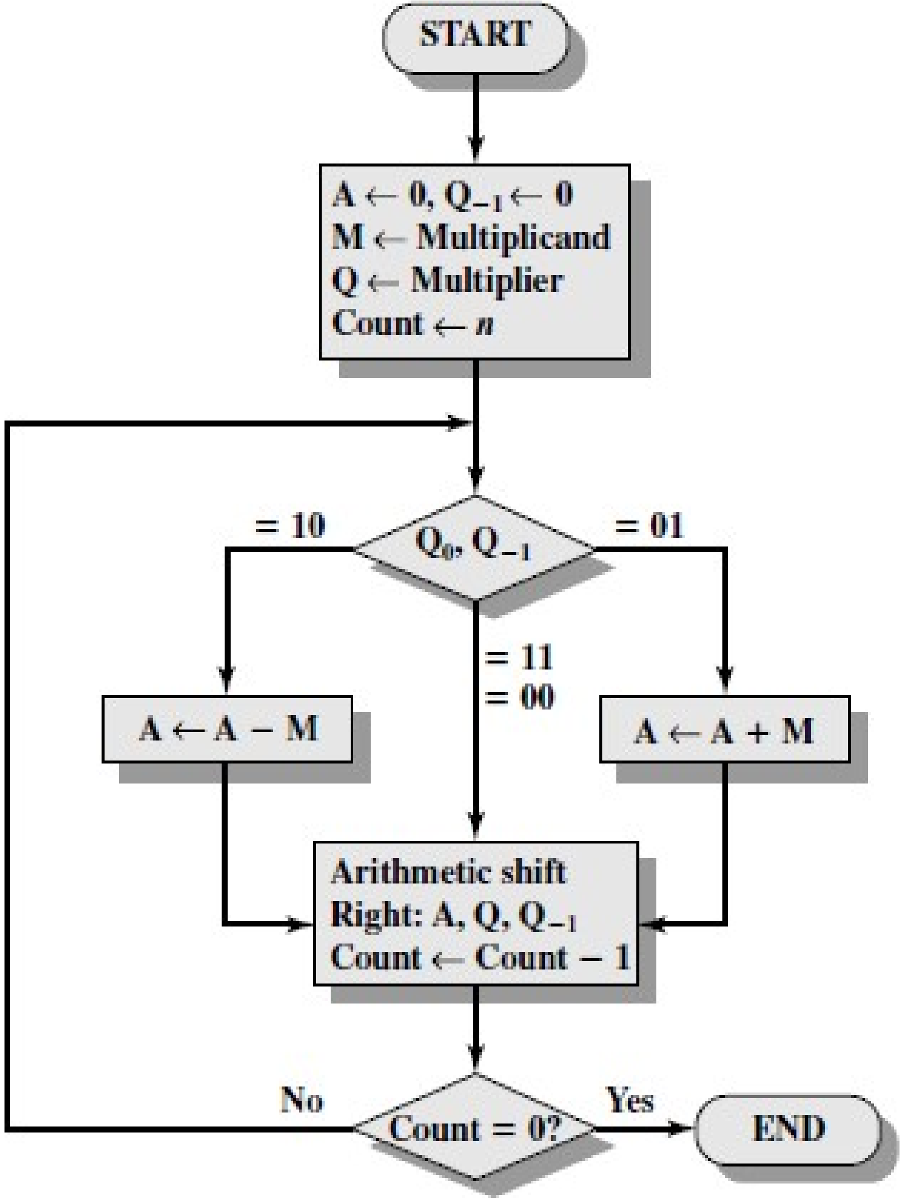
Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

- data paths and hardware elements needed to accomplish addition and subtraction.
- The central element is a 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, designated in this case as A and B registers.
- The result may be stored in one of these registers or in a third.
- The overflow indication is stored in a 1-bit overflow flag.
- For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder
- Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

Booth's Algorithm



Q0	Q-1	Result
0	0	Only shift
1	1	
0	1	A=A + M ,then shift
1	0	A= A – M , then shift

M =7

Q =3

M = 0 1 1 1

Q = 0 0 1 1

- M = 1 0 0 1

Example of Booth's Algorithm:7(M)*3(Q)

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A = A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A = A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Answer is in A and Q→0001 0101 =21

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

Figure 9.13 Example of Booth's Algorithm (7 × 3)

Examples-size of n determines answer

Solve using Booths Algorithm

A. M = 5 ,   Q = 5

B. M = 12,   Q = 11

C. M = 9,   Q = - 3

D. **M = -13 (0011)** , Q = 6

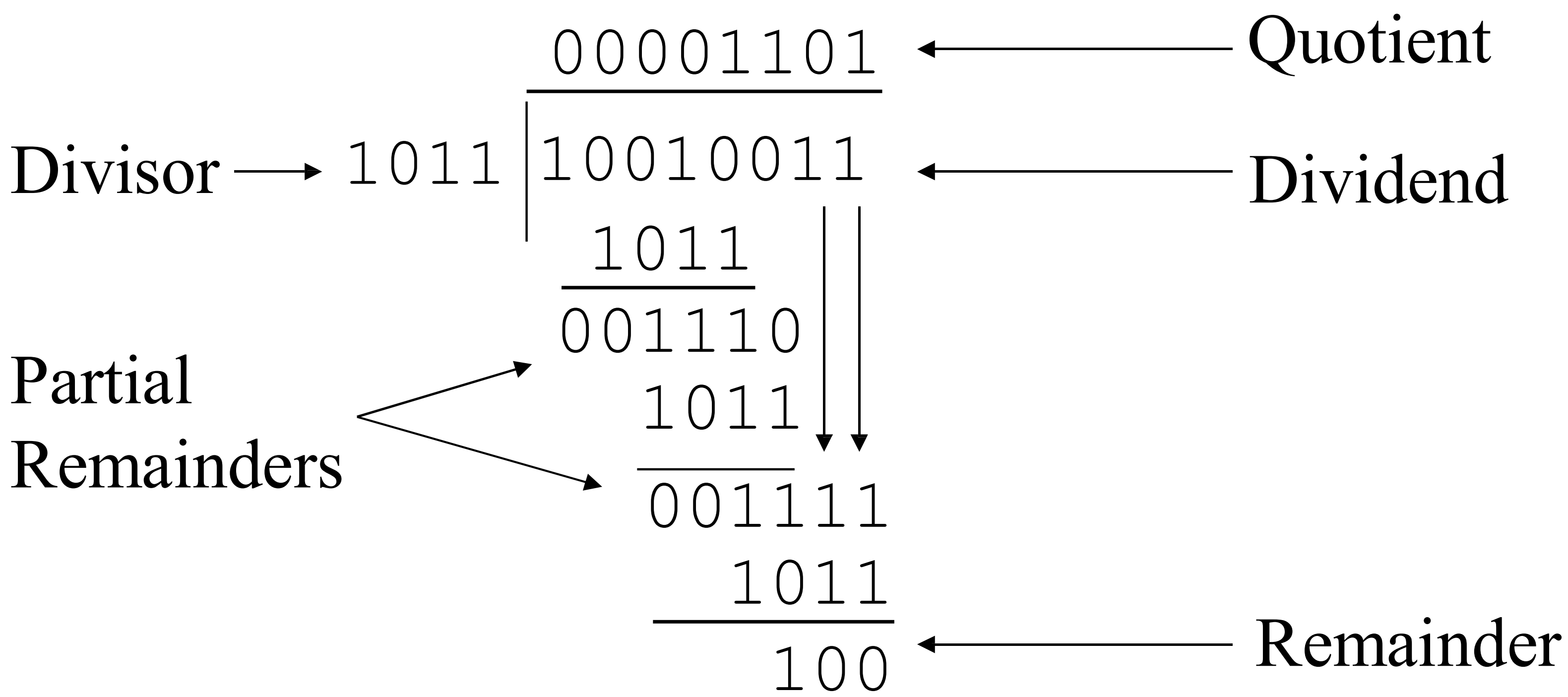
-M=13 (1101)

A. M = -19 , Q = -20

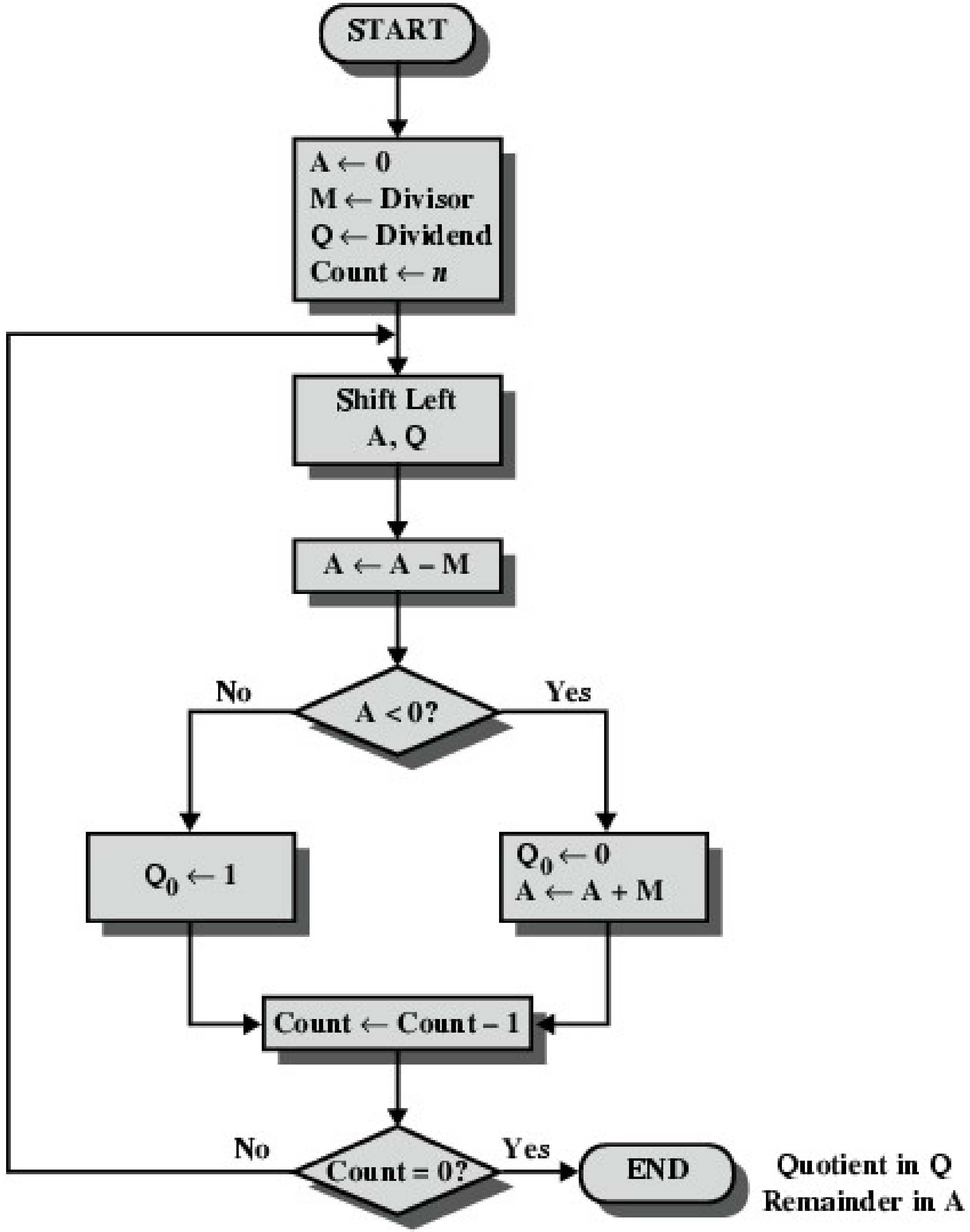
Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division

Division of Unsigned Binary Integers



Flowchart for Restoring Division



$M \leftarrow 3\overline{)7} \rightarrow Q$

DATE / /

$M = 0011$

$-M = 1101$

$Q = 0111$

$$\begin{array}{r|l|l} 2 & 7 & \\ 2 & 3 & 1 \\ & 1 & 1 \end{array} \qquad \begin{array}{r|l|l} 2 & 3 & \\ & 1 & 1 \end{array}$$

1 1 0 0

1

0000

	A	Q	
	0 0 0 0	0 1 1 1	
	0 0 0 0	1 1 1 $\square$	Shift left A, Q
	1 1 0 1	1 1 1 $\square$	$A = A - M$
	0 0 0 0	1 1 1 $\boxed{0}$	set $q_0 = 0$; $A = A + M$ → Restore

A = 1101

M = 0011

0000

cycle 1

	A	Q	
0001	0001	110 $\square$	Shift left
1101	Restore		A = A − M
1110	1110	110 $\boxed{0}$	
			set $Q_0 = 0$
1110			A = A + M
0011	0001	1100	
0001			

0001

0 0 1 1   1 0 0 $\boxed{}$ Shift left

$$\begin{array}{r} 0011 \\ \cancel{1}1101 \\ \hline 0000 \end{array}$$

0 0 0 0   1 0 0 $\boxed{1}$ $A = A - M$

set $q_0 = 1$

0 0 0 0   1 0 0 1

0 0 0 1   0 0 1 $\boxed{}$ shift left

$$\begin{array}{r} 0001 \\ 1101 \\ \hline 1110 \end{array}$$

1 1 1 0   0 0 1 $\boxed{0}$ $A = A - M$

set $q_0 = 0$

$A = A + M$

$$\underbrace{0\ 0\ 0\ 1}_{\text{Remainder} = 1} \qquad \underbrace{0\ 0\ 1\ 0}_{\text{Quotient} = 2}$$

M = 27 ; Q = 55

M = 011011

−M = 100101

Q = 110111

	A	Q	
cle 1	000000	110111	
	000001	10111□	shift left A,
	100110	10111$\boxed{0}$	A = A − M
	000001	101110	Set $Q_0 = 0$; A = A

cycle2	000011	01110☐	shift left A,Q
	101000	01110$\boxed{0}$	A=A-M
			set $q_0=0$
	000011	011100	A=A+M
cycle3	000110	11100☐	shift left A,Q
	101011	11100$\boxed{0}$	A=A-M
	000110	111000	Set $q_0=0$; A=A+M
cycle4	001101	11000☐	shift left A,Q
	110010	11000$\boxed{0}$	A=A-M

cycle 4	0011101	11000☐	shift left A, Q
	1110010	11000$\boxed{0}$	A = A − M
	0011101	110000	set $Q_0 = 0$, A = A +
cycle 5	0111011	10000☐	shift left A, Q,
	0000000	10000$\boxed{1}$	set $Q_0 = 1$
cycle 6	0000001	00001☐	shift left A, Q
	1001110	00001$\boxed{0}$	A = A − M
			set $Q_0 = 0$,
	0000001	000010	A = A + M

R = 1 Q = 2

M = 27 ; Q = 55

M = 011011

−M = 100101

Q = 110111

	A	Q	
cycle1	000000	110111	
	000001	10111□	shift left A,Q
	100110	10111$\boxed{0}$	A = A − M
	000001	101110	Set Q_0=0; A=A+M
cycle2	000011	01110□	shift left A,Q
	101000	01110$\boxed{0}$	A = A − M
			set Q_0 = 0
	000011	011100	A = A + M
cycle3	000110	11100□	shift left A,Q
	101011	11100$\boxed{0}$	A = A − M
	000110	111000	Set Q_0=0; A=A+M
cycle4	001101	11000□	shift left A,Q
	110010	11000$\boxed{0}$	A = A − M
	001101	110000	set Q_0=0, A=A+M
cycle5	011011	10000□	shift left A,Q,
	000000	10000$\boxed{1}$	set Q_0 = 1
cycle6	000001	00001□	shift left A,Q
	100110	00001$\boxed{0}$	A = A − M
			Set Q_0 = 0
	000001	000010	A = A + M

R = 1 Q = 2

Solve using Restoring Division

A. M = 5 , Q = 5 , A=0000 , Q=0010

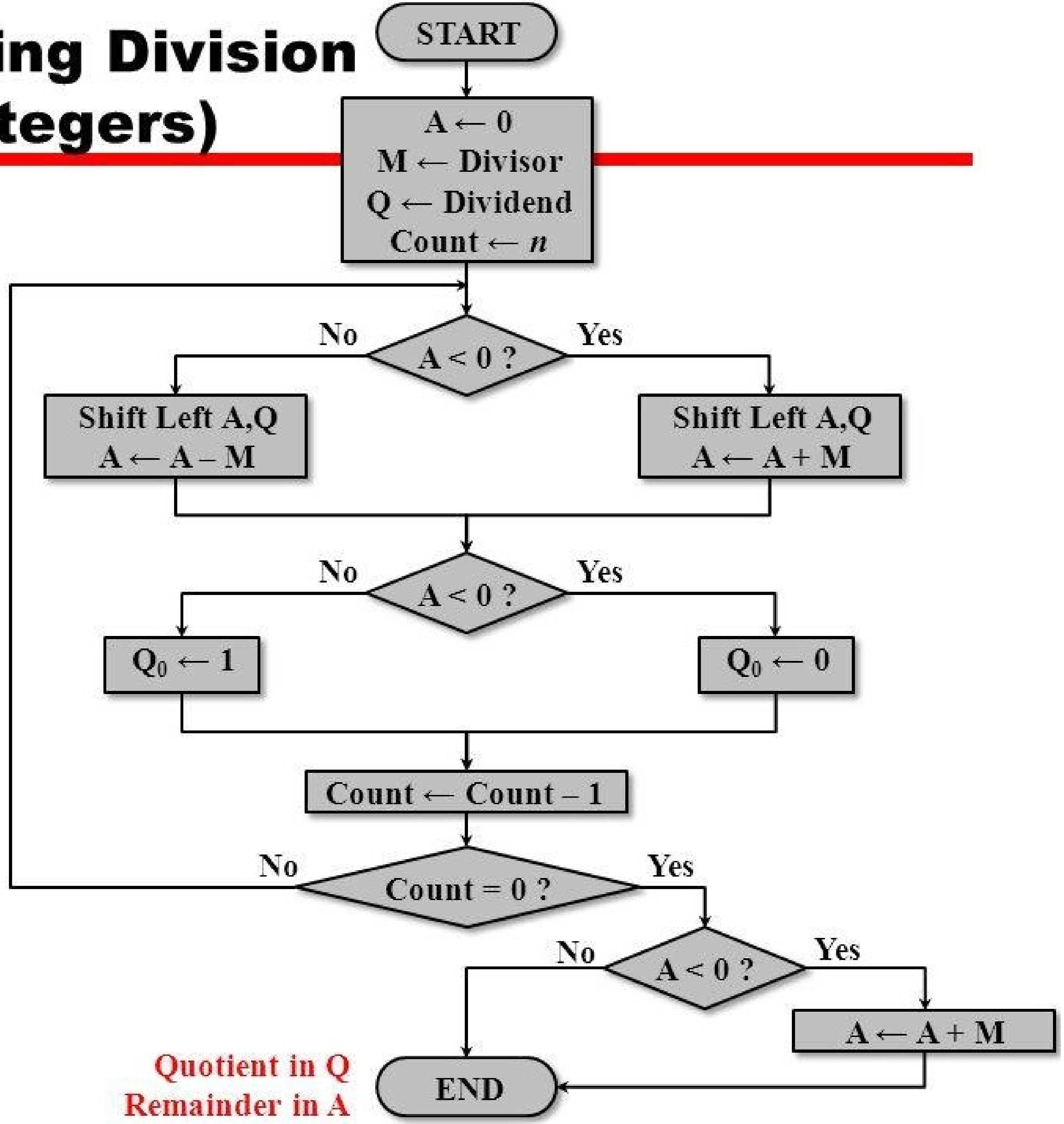
B. M = 12, Q = 26, A=00010 , Q= 00010

C. M = 9, Q = 19 , A=00001 , Q =00010

D. M = 32 , Q = 59, A=011011, Q=000001

E. M = 17 , Q = 42, A=001000 , Q=000010

Non-Restoring Division (Positive Integers)



$M = 2$; $Q = 4$

$M = 0010$

$-M = 1110$

$Q = 0100$

	A	Q	
	0000	0100	
Shift left A,Q	0000	100□	①
A=A−M, set $q_0=0$	1110	100$\boxed{0}$	①
shift left A,Q	1101	000□	②
A=A+M, set $q_0=0$	1111	000$\boxed{0}$	②

shift left A,Q	1 1 1 0	0 0 0 $\square$	(3)
A = A + M	0 0 0 0	0 0 0 $\boxed{1}$	
set $q_0 = 1$			
shift left A,Q	0 0 0 0	0 0 1 $\square$	(4)
A = A − M	1 1 1 0	0 0 1 $\boxed{0}$	
set $q_0 = 0$			

count = 0

A = A + M

$$
\begin{array}{rl}
 & 1110 \ (A) \\
 & 0010 \ (M) \\
\hline
\cancel{1} & 0000
\end{array}
$$

$\underbrace{0000}_{}$ A = Remainder

$\underbrace{0010}_{}$ Q = Quotient

Solve using Non Restoring

A. M = 5 , Q = 5,A=0000,Q=0001.

B. M = 12, Q = 26,A=000010, Q=000010.

C. M = 9, Q = 19,A=00001,Q=00010.

D. M = 32 , Q = 59,A=011011,Q=000001.

E. M = 17 , Q = 42,A=001000,Q=000010

Booths Recoding / Bit pair recording

STEPS

Booth's Recoding algorithm

5 × 3

M Q

M 0101 Q = 0011

−M 1011

Step 1 : Table for M

Operation	Value
0	0000 0000
+1	0000 0101
−1	1111 1011
+2	0000 1010
−2	1111 0110

left shift +1

left shift −1

Step 2 : Value of Q

Q_{-1}

0 0 1 1 0

0 +1 0 −1

2(1st nos) + 2nd nos

$2(0) + 1$ $\quad$ $2(0) + (-1)$

1 $\quad$ −1

Step 3: M * Q

0 1 0 1

1 -1

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

Discard carry $\boxed{1}$

0 0 0 0 1 1 1 1

$2^3\ 2^2\ 2^1\ 2^0$

$8+4+2+1=15$

Solve using Booths Recoding

1. M= 5 , Q = 4 (4 bits)= 00010100 (20)
2. M=9 , Q = -6 (5 bits)=11110 01010 (-54)
3. M=15 , Q=-10 (5 bits)=11011 01010(-150)
4. M= -13 ,Q = -20 (6 bits)=000100000100(260)

Sample mix problems-Kindly refrain referring to flowchart.

1. Booth's Algorithm = 000 100 000 100(260)

A= 110011 (Multiplicand)

B= 101100 (Multiplier)

2. Booth's Recoding = 0110 1010/11011 01010

M= (15)

Q= (-10)

3. Non Restoring Division

M=11 , Q= 21 , A= 01010 , Q= 00001

4. Restoring Division

M=14 , Q= 15, A=00001 , Q = 00001

Floating Point

Sign bit	Biased Exponent	Significand or Mantissa

- A floating point number, is a positive or negative whole number with a decimal point. For example, 5.5, 0.25, and -103.342 are all floating point numbers, while 91, and 0 are not

$$\pm .\text{significand} \times 2^{\text{exponent}}$$

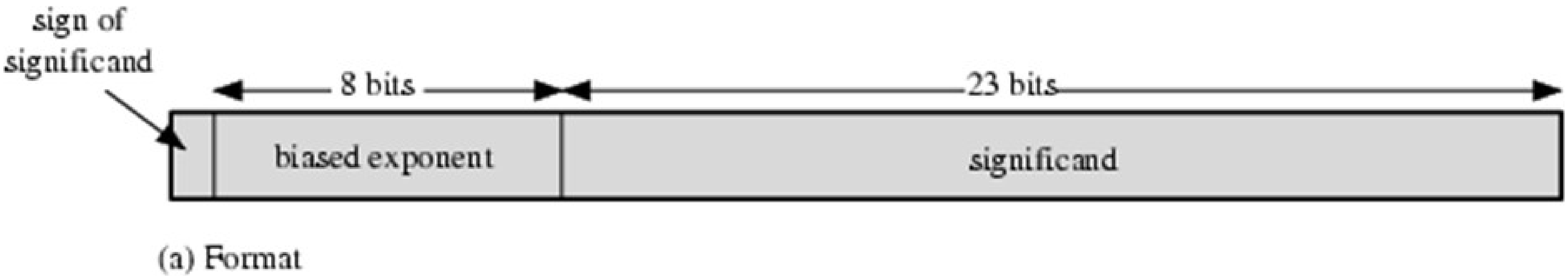
- Misnomer
- Point is actually fixed between sign bit and body of mantissa
- Exponent indicates place value (point position)

$$\pm S \times B^{\pm E}$$

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E

Floating Point Examples



Typical 32-Bit Floating-Point Format

The leftmost bit stores the **sign** of the number
The **exponent** value is stored in the next 8 bits.
The representation used is known as a **biased representation**.
A fixed value, called the bias, is subtracted from the field to get the true exponent value.

- A fixed value, called the bias, is subtracted from the field to get the true exponent value. Typically, the bias equals where k is the number of bits in the binary exponent.
- In this case, the 8-bit field yields the numbers 0 through 255.
- With a bias of 127 (2^7-1), true exponent values are in the range -127 to 128. In this example, the base is assumed to be 2.
- **A normalized number is one in which the most significant digit of the significand is nonzero. For base 2 representation, a normalized number is therefore one in which the most significant bit of the significand is one**

Thus, a normalized nonzero number is one in the form

$$\pm 1.bbb\ldots b \times 2^{\pm E}$$

Signs for Floating Point

- Mantissa is stored in 2s compliment
- Exponent is in excess or biased notation
  - e.g. Excess (bias) 128 means
  - 8 bit exponent field
  - Pure value range 0-255
  - Subtract 128 to get correct value
  - Range -128 to +127

Expressible Numbers

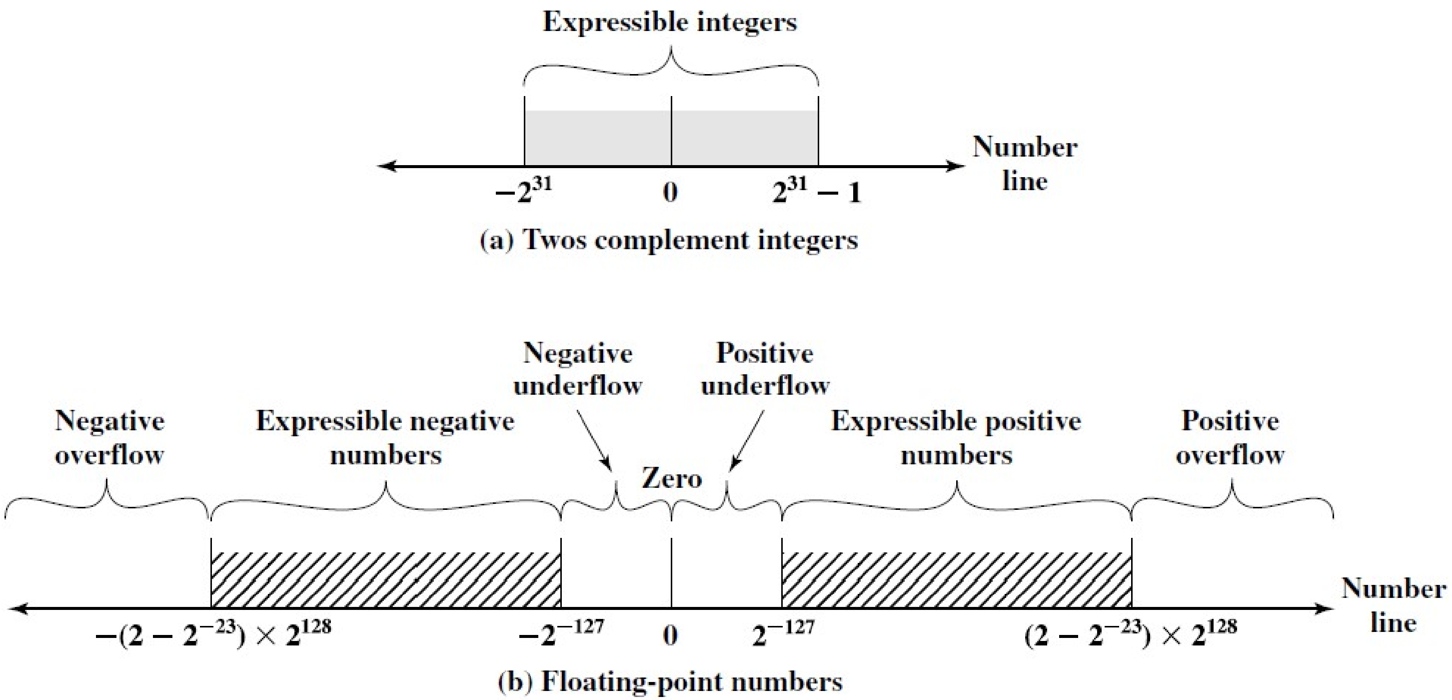


Figure 9.19 Expressible Numbers in Typical 32-Bit Formats

- Negative numbers between $-(2 - 2^{-23}) \times 2^{128}$ and -2^{-127}
- Positive numbers between 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(2 - 2^{-23}) \times 2^{128}$, called **negative overflow**
- Negative numbers greater than 2^{-127}, called **negative underflow**
- Zero
- Positive numbers less than 2^{-127}, called **positive underflow**
- Positive numbers greater than $(2 - 2^{-23}) \times 2^{128}$, called **positive overflow**

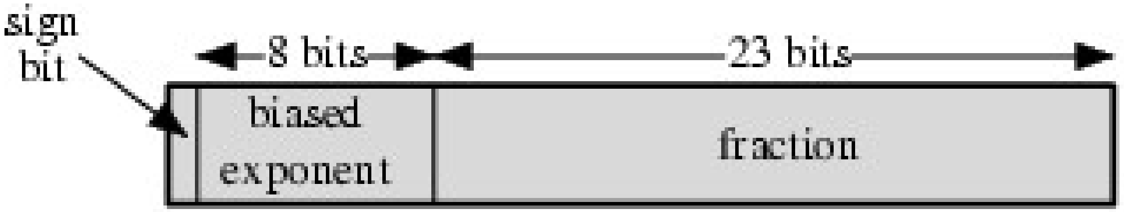
Overflow occurs when an arithmetic operation results in a magnitude greater than can be expressed with an exponent of 128.
Underflow occurs when the fractional magnitude is too small

IEEE 754

- Standard for floating point storage
- developed to facilitate the portability of programs from one processor to another
- Defines 32 and 64 bit standards with 8 and 11 bit exponent respectively
- the standard defines two extended formats, single and double, whose exact format is implementation dependent.
- The extended formats include additional bits in the exponent (extended range) and in the significand (extended precision).
- The extended formats are to be used for intermediate calculations.
- Extended formats (both mantissa and exponent) for intermediate results

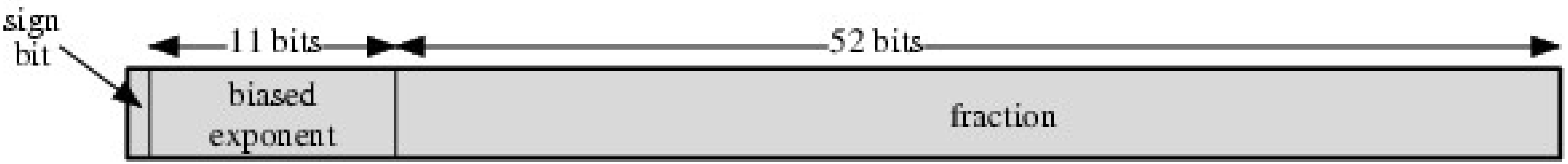
IEEE 754 Formats

$(1.N)2^{E-127}$



(a) Single format

32 BIT



(b) Double format

64 BIT

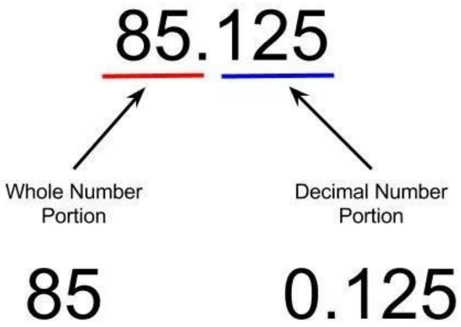
$(1.N)2^{E-1023}$

Steps

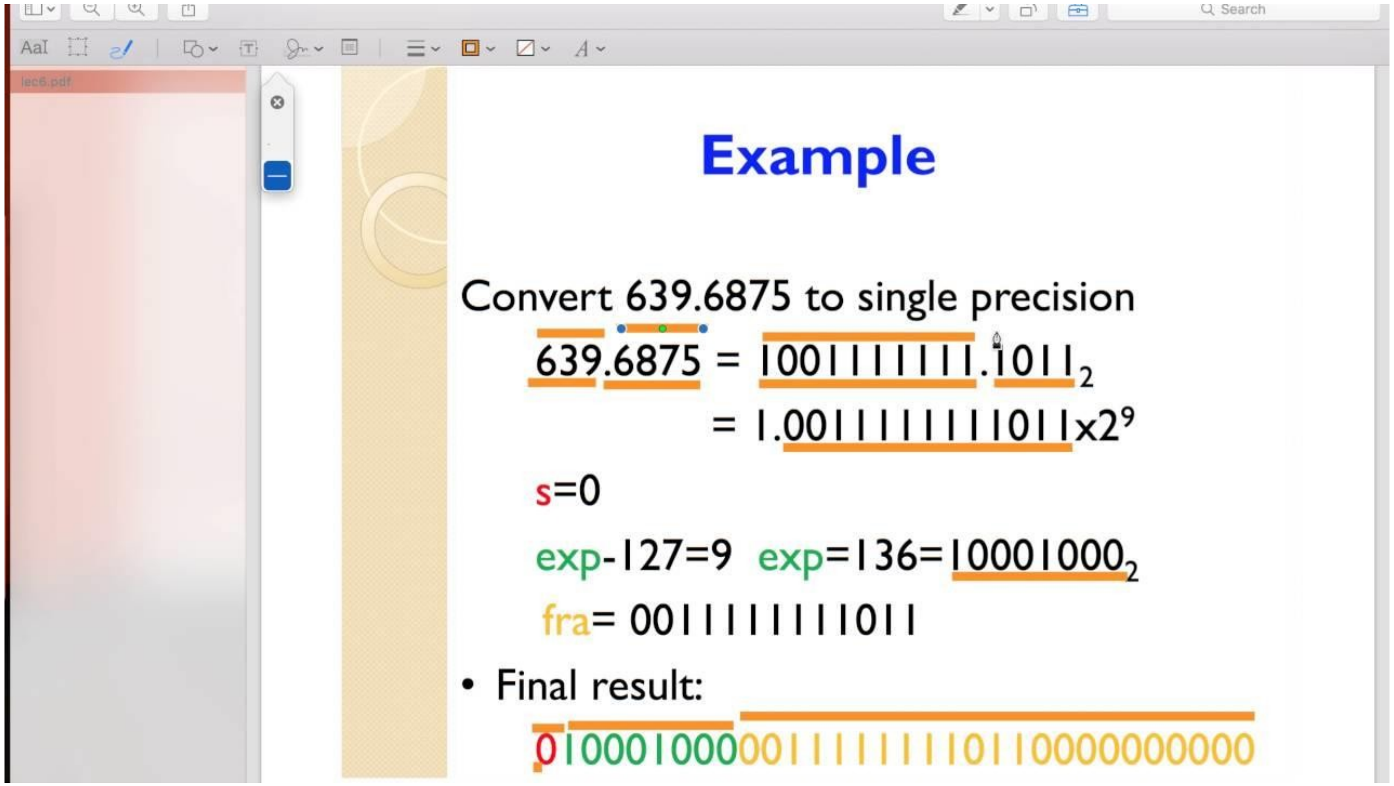
- 1. Convert Decimal to Binary
- 2. Normalization
  - Rewriting Step 1 into (1.N) form
  - Ex: $111.011 = \mathbf{1.}11011 \times 2^{\mathbf{2}}$
  - Ex: $0.00010 = 0000\mathbf{1.}0 \times 2^{\mathbf{-4}}$

Exponent

- 3.Biasing
  - Applying Single Precision (E – 1 2 7) & Double Precision (E – 1 0 2 3) on exponent from Step 2
- 4. Representation in Single (32 bit)and Double Precision (64 bit) Format



85.125
Whole Number Portion
Decimal Number Portion
85
0.125



Search
Example
Convert 639.6875 to single precision
639.6875 = 1001111111.1011₂
= 1.0011111111011x2⁹
s=0
exp-127=9 exp=136=10001000₂
fra= 0011111111011
• Final result:
01000100000111111110110000000000

Solved Example

Eg 12.25

Step 1: Converting Dec to Bin

2	12	
2	6	0
2	3	0
	1	1
		1

$$
\begin{array}{r}
.25 \\
\times\ 2 \\
\hline
0.50 \\
\times\ 2 \\
\hline
1.00
\end{array}
\rightarrow \text{stop}
$$

$$\underline{12} \, . \, \underline{25}$$

$$1100.01$$

Step 2: Normalization (1. N)

1.10001×2^{3} → Exponent

Step3: Biasing

Single Precision	Double precision
$E-127$	$E-1023$
$3 = E-127$	$3 = E-1023$
$E = 127+3$	$E = 1023+3$
$= 130$	$= 1026$

2	130	
2	65	0
2	32	1
2	16	0
2	8	0
2	4	0
2	2	0
	1	0
		1

2	1026	
2	513	0
2	256	1
2	128	0
2	64	0
2	32	0
2	16	0
2	8	0
2	4	0
2	2	0
	1	0

Single Precision (32 bits)

Sign bit	Biased Exponent	Mantissa/Significand
0	10000010	10001
1 bit	8 bits	23 bits

Double Precision (64 bits)

Signbit		
0	10000000010	10001
1 bit	11 bits	52 bits

Solve

25.44	SP- 0|100000|1001 0111 0000 1010 0011 110
	DP- 0|10000000011|1001 0111 0000 1010 0011 110
0.00635	SP- 0|1110111|00000001101000...
	DP- 0|1111110111|00000001101000...
-125.10	SP- 1 | 10000101| 1111 010001
	DP- 1 | 10000000101| 1111 010001
-13.54	SP- 1|10000010|10110001010
	DP- 1|10000000010|10110001010

Sample Problems to Solve

1) 178.1875

SP 0|10000110|01100100011

DP 0|10000000110|

1) 309.175

SP 0|10000111|01011101001011

DP 0|10000000111|

1) 1259.125

SP 0|10001001|0011101011001000…(9 zeroes)

DP 0|10000001001|**010100111100**

1) 0.0625

SP 0|01111011|0000000….

DP 0|01111111|00000…..

Division of signed numbers

1. Load the divisor into the M register and the dividend into the A, Q registers. The dividend must be expressed as a $2n$-bit twos complement number. Thus, for example, the 4-bit 0111 becomes 00000111, and 1001 becomes 11111001.
2. Shift A, Q left 1 bit position.
3. If M and A have the same signs, perform $A \leftarrow A - M$; otherwise, $A \leftarrow A + M$.
4. The preceding operation is successful if the sign of A is the same before and after the operation.
   a. If the operation is successful or $A = 0$, then set $Q_0 \leftarrow 1$.
   b. If the operation is unsuccessful and $A \neq 0$, then set $Q_0 \leftarrow 0$ and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A. If the signs of the divisor and dividend were the same, then the quotient is in Q; otherwise, the correct quotient is the twos complement of Q.

The reader will note from Figure 9.17 that $(-7) \div (3)$ and $(7) \div (-3)$ produce different remainders. This is because the remainder is defined by

$$D = Q \times V + R$$

where

$$D = \text{dividend}$$
$$Q = \text{quotient}$$
$$V = \text{divisor}$$
$$R = \text{remainder}$$

The results of Figure 9.17 are consistent with this formula.

A	Q	M = 0011
0000	0111	Initial value
0000	1110	shift
1101		subtract
0000	1110	restore
0001	1100	shift
1110		subtract
0001	1100	restore
0011	1000	shift
0000		subtract
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		subtract
0001	0010	restore

(a) (7)/(3)

Solve

a) 7 / -3

b) -7 (Q) / 3 (M)

c) -7 (Q) / -3 (M)

A	Q	M = 1101
0000	0111	Initial value
0000	1110	shift
1101		add
0000	1110	restore
0001	1100	shift
1110		add
0001	1100	restore
0011	1000	shift
0000		add
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		add
0001	0010	restore

(b) (7)/(−3)

A	Q	M = 0011
1111	1001	Initial value
1111	0010	shift
0010		add
1111	0010	restore
1110	0100	shift
0001		add
1110	0100	restore
1100	1000	shift
1111		add
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		add
1111	0010	restor

(c) (–7)/(3)

A	Q	M = 1101
1111	1001	Initial value
1111	0010	shift
0010		subtract
1111	0010	restore
1110	0100	shift
0001		subtract
1110	0100	restore
1100	1000	shift
1111		subtract
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		subtract
1111	0010	restore

(d) (–7)/(–3)

- Dividend negative → Remainder –ve

Table 9.5 Floating-Point Numbers and Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$$X = 0.3 \times 10^2 = 30$$
$$Y = 0.2 \times 10^3 = 200$$

$$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$$
$$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$$
$$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$$
$$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$$

A floating-point operation may produce one of these conditions:

- **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as $+\infty$ or $-\infty$.
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value (e.g., -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.
- **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand. As we shall discuss, some form of rounding is required.
- **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we shall explain.

4 phases of FP Arithmetic +/-

- Check for zeros
- Align significands (adjusting exponents)
- Add or subtract significands
- Normalize result

Floating Point Addition

Add the following two decimal numbers in scientific notation:

8.70×10^{-1} with 9.95×10^{1}

Rewrite the smaller number such that its exponent matches with the exponent of the larger number.

$8.70 \times 10^{-1} = 0.087$ (Note !) $\times 10^{1}$

Add the mantissas

$9.95 + 0.087 = 10.037$ and

write the sum 10.037×10^1

Put the result in Normalised Form

$10.037 \times 10^1 = 1.0037 \times 10^2$

(shift mantissa, adjust exponent)

Check for overflow/underflow of the exponent after normalisation

- **Overflow**

  The exponent is too large to be represented in the Exponent field

- **Underflow**

  The number is too small to be represented in the Exponent field

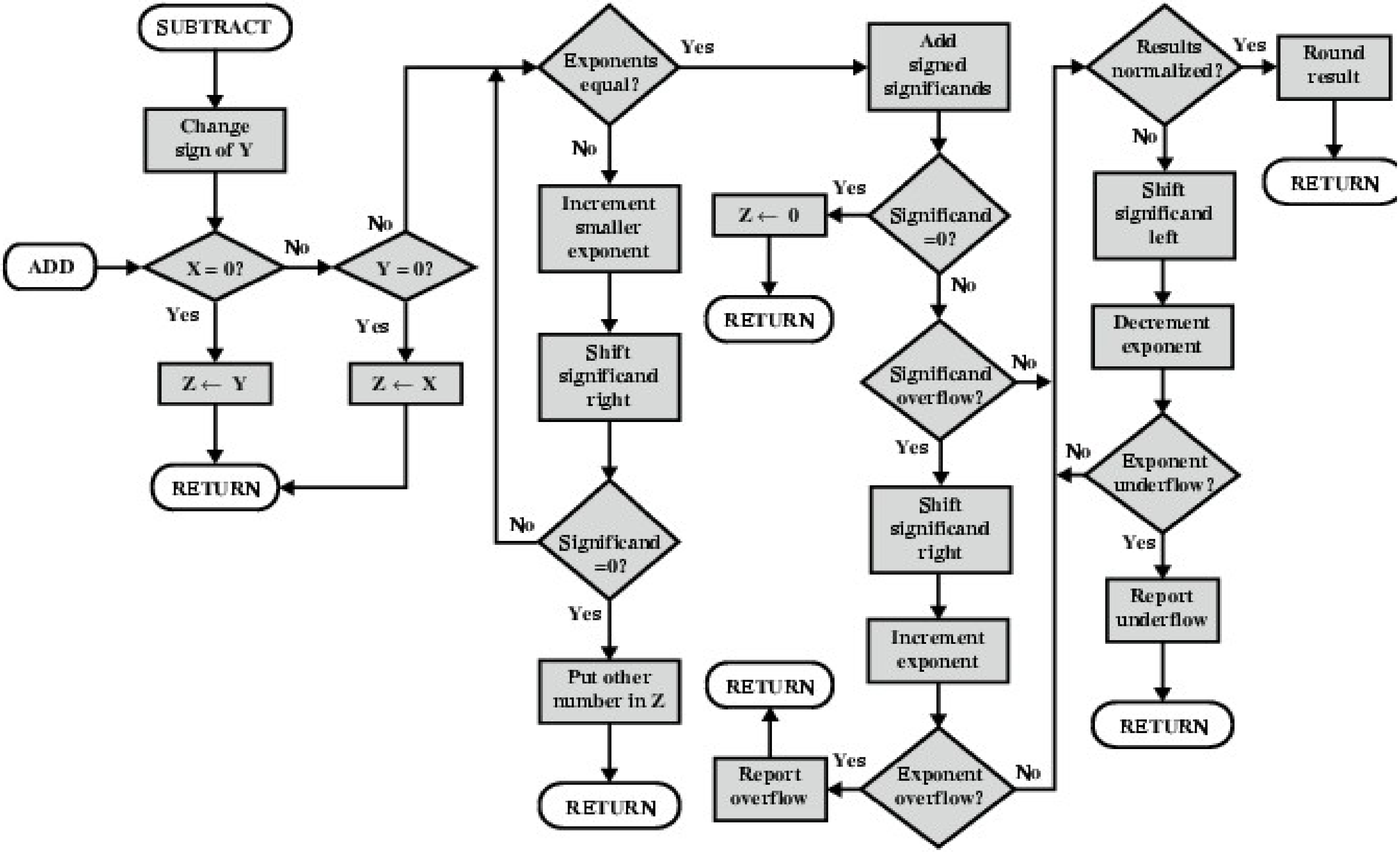
Round the result

If the mantissa does not fit in the space reserved for it, it has to be rounded off.

For Example: If only 4 digits are allowed for mantissa

1.0037×10^2 ===> 1.004×10^2

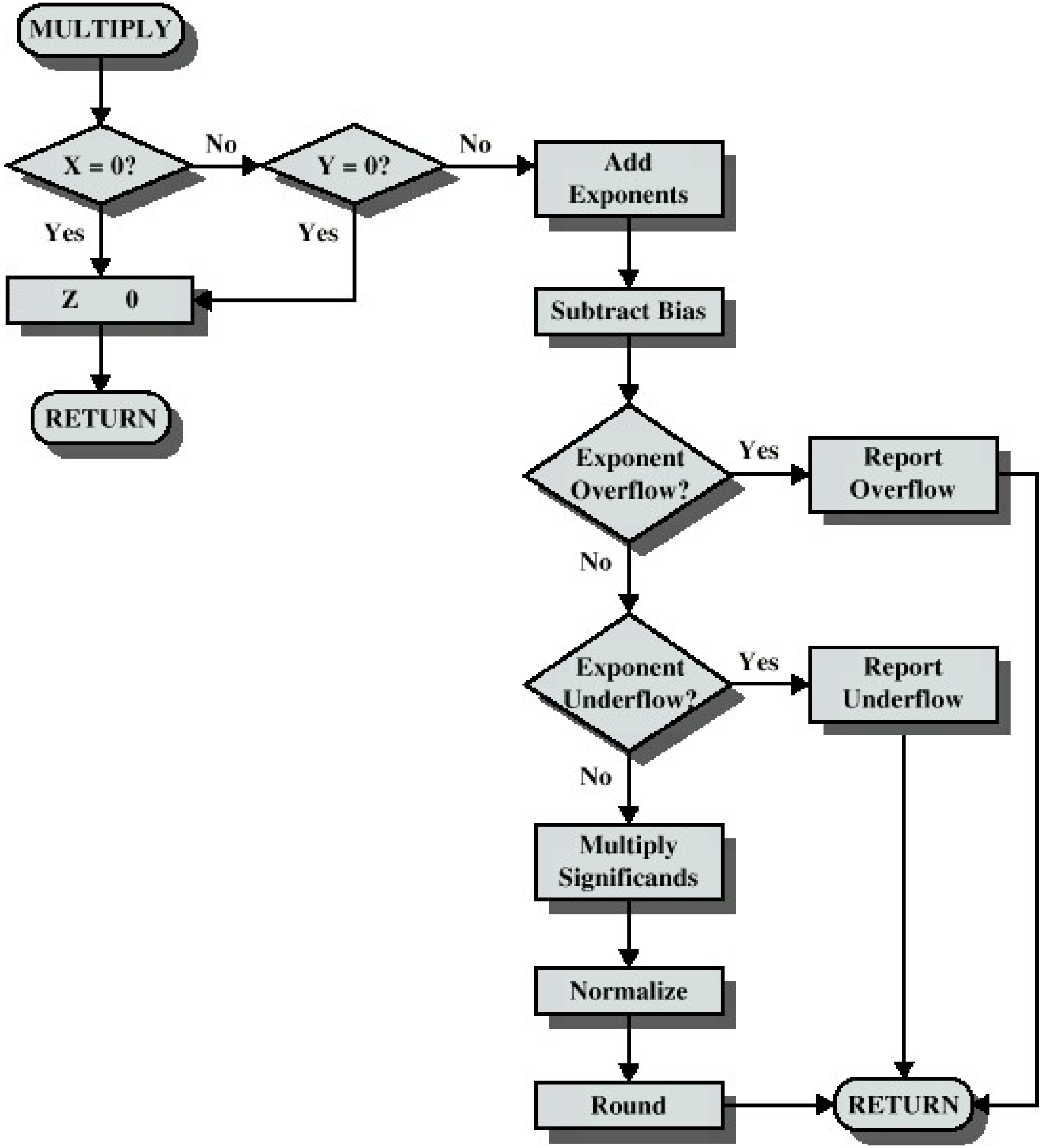
FP Addition & Subtraction Flowchart



FP Arithmetic x/÷

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

Floating Point Multiplication



Floating Point Division