AAiT

Computer Architecture & Organization

Chapter 4

Computer Arithmetic

AAiT

Arithmetic & Logic Unit (ALU)

- Part of the computer that actually performs arithmetic and logical operations on data
- All of the other elements of the computer system are there mainly to bring data into the ALU for it to process and then to take the results back out
- ALU is based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations

ALU Inputs and Outputs

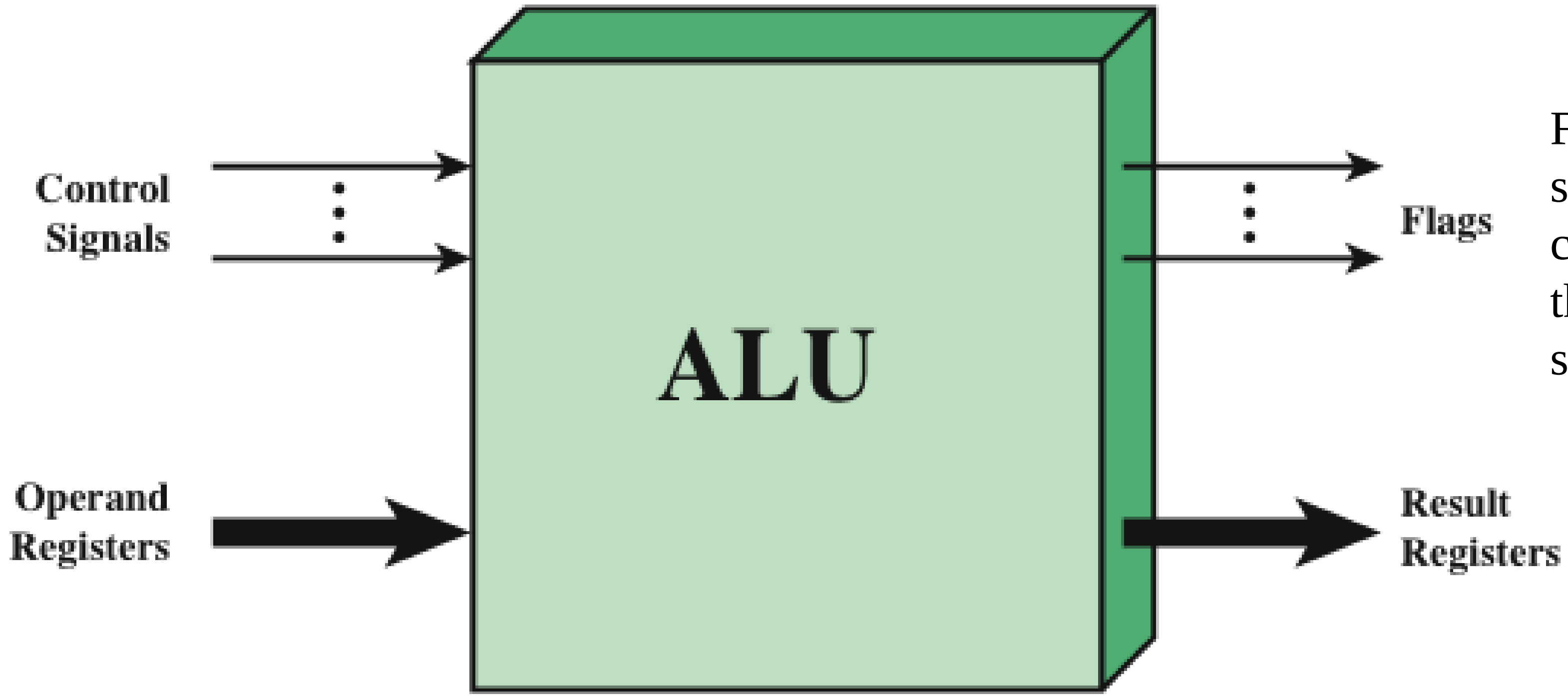


Figure 10.1 ALU Inputs and Outputs

For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

Integer Representation

- In the binary number system arbitrary numbers can be represented with:
  - The digits zero and one
  - The minus sign (for negative numbers)
  - The period, or ***radix point*** (for numbers with a fractional component)
- For purposes of computer storage and processing we do not have the benefit of special symbols for the minus sign and radix point
- Only binary digits (0,1) may be used to represent numbers
- If we are limited to nonnegative integers, the representation is straightforward.

AAiT

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers

- All of these alternatives 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

Sign-magnitude representation is the simplest form that employs a sign bit

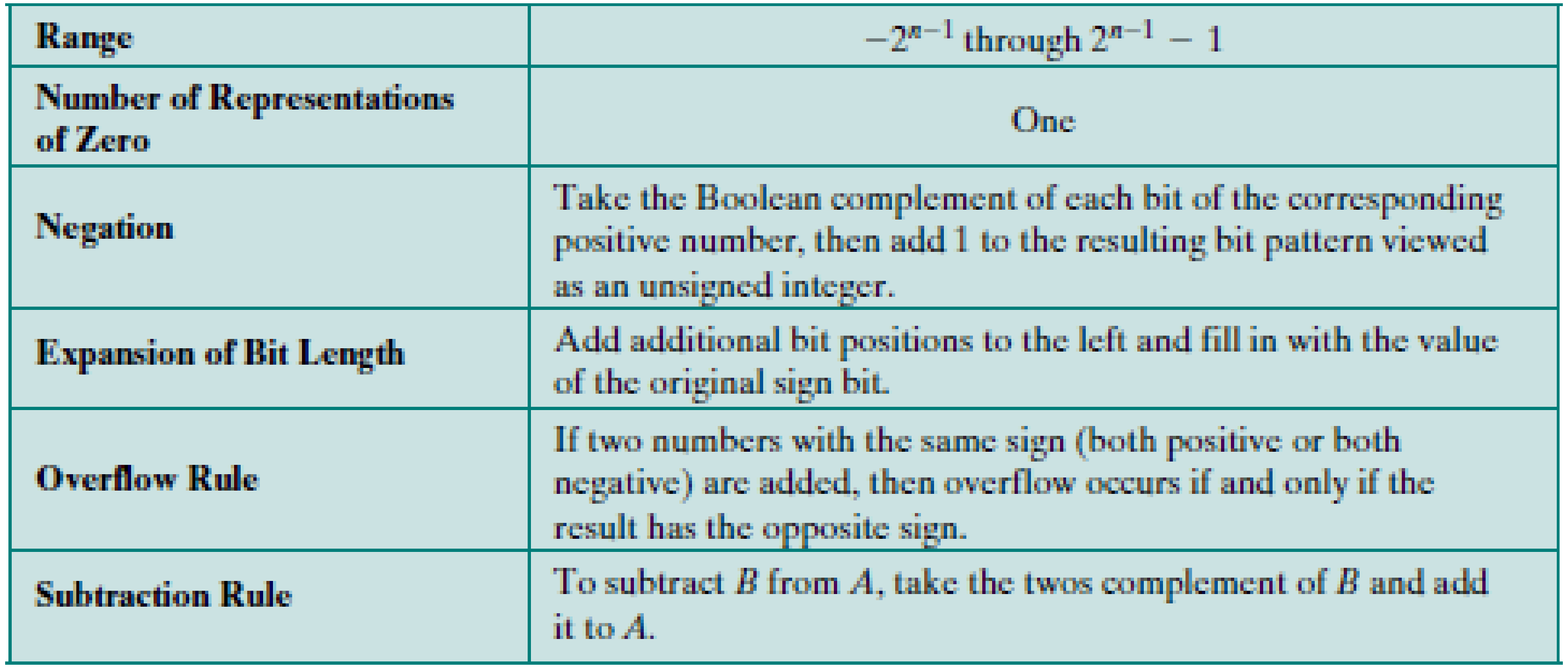
Drawbacks:

- Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation
- There are two representations of 0

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU

Twos Complement Representation

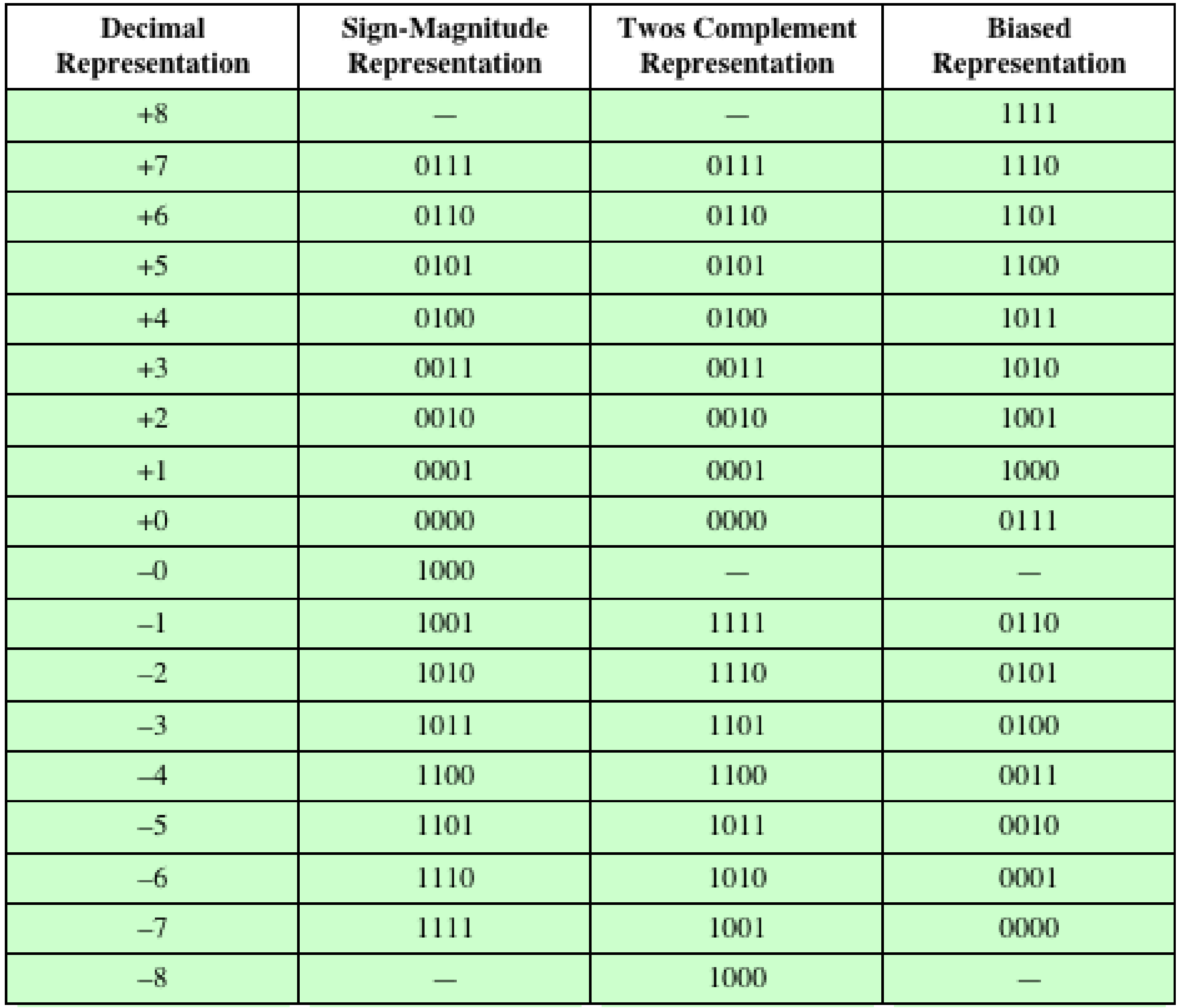
- Uses the most significant bit as a sign bit
- Differs from sign-magnitude representation in the way that the other bits are interpreted



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.

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

Table 10.2
Alternative Representations for 4-Bit Integers



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

Range Extension

- Range of numbers that can be expressed is extended by increasing the bit length
- In sign-magnitude notation this is accomplished by moving the sign bit to the new leftmost position and fill in with zeros
- This procedure will not work for twos complement negative integers
  - Rule 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*

Fixed-Point Representation

The radix point (binary point) is fixed and assumed to be to the right of the rightmost digit

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

Negation

- Twos complement operation
  - Take the Boolean complement of each bit of the integer (including the sign bit)
  - Treating the result as an unsigned binary integer, add 1

```
                +18 = 00010010 (twos complement)
bitwise complement = 11101101
                     +       1
                      11101110 = -18
```

- The negative of the negative of that number is itself:

```
                -18 =  11101110 (twos complement)
bitwise complement =  00010001
                      +       1
                       00010010 = +18
```

Negation Special Case 1

0 =		00000000	(twos complement)
Bitwise complement =		11111111	
Add 1 to LSB		+ 1	
Result	100000000		

Overflow is ignored, so:

- 0 = 0

Negation Special Case 2

```
                 -128     =       10000000    (twos complement)
Bitwise complement   =        01111111
Add 1 to LSB                 +          1
                             ------------
Result                            10000000
```

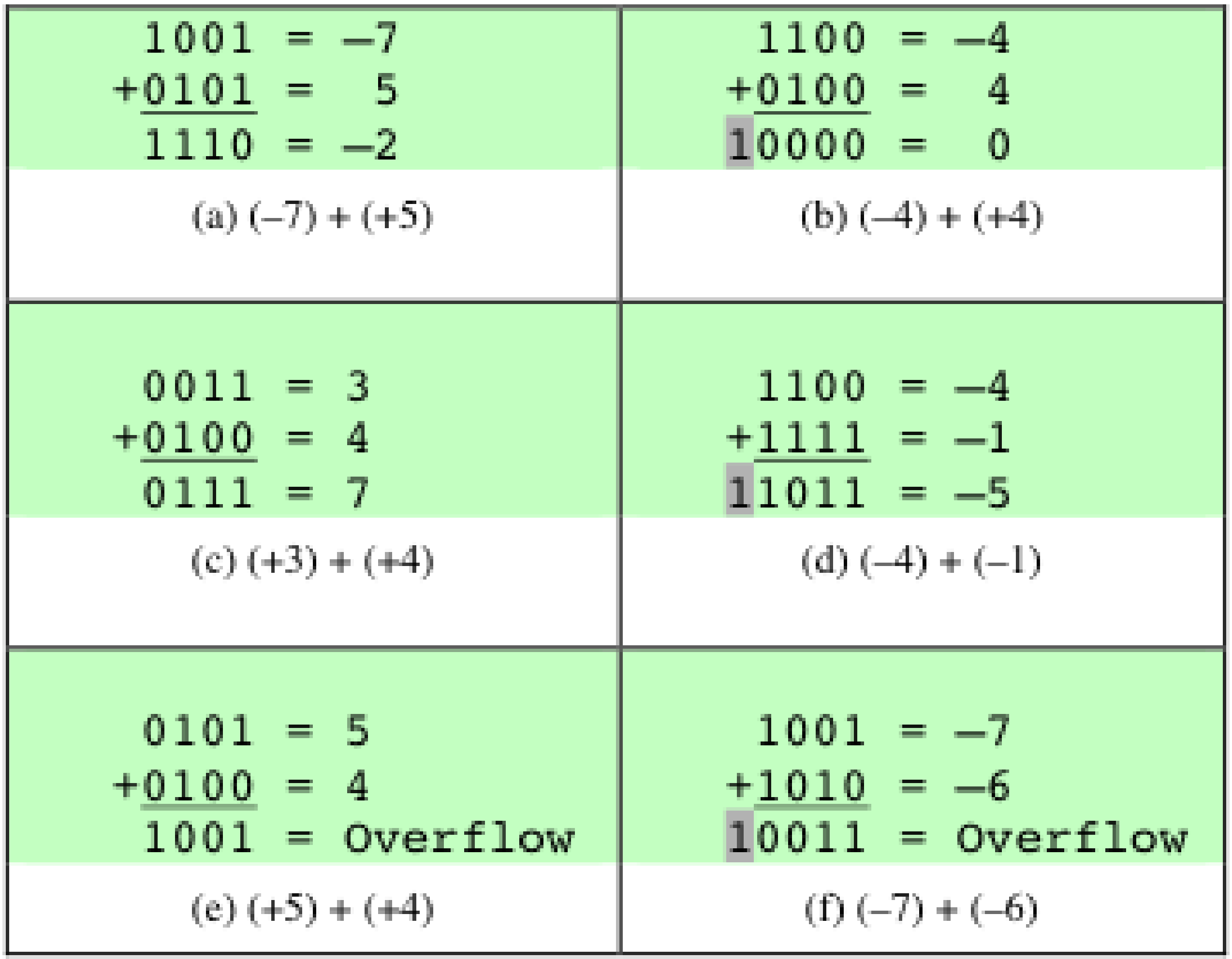
So:

-(-128) = -128   X

Monitor MSB (sign bit)

It should change during negation

Addition



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 10.3 Addition of Numbers in Twos Complement Representation

AAiT

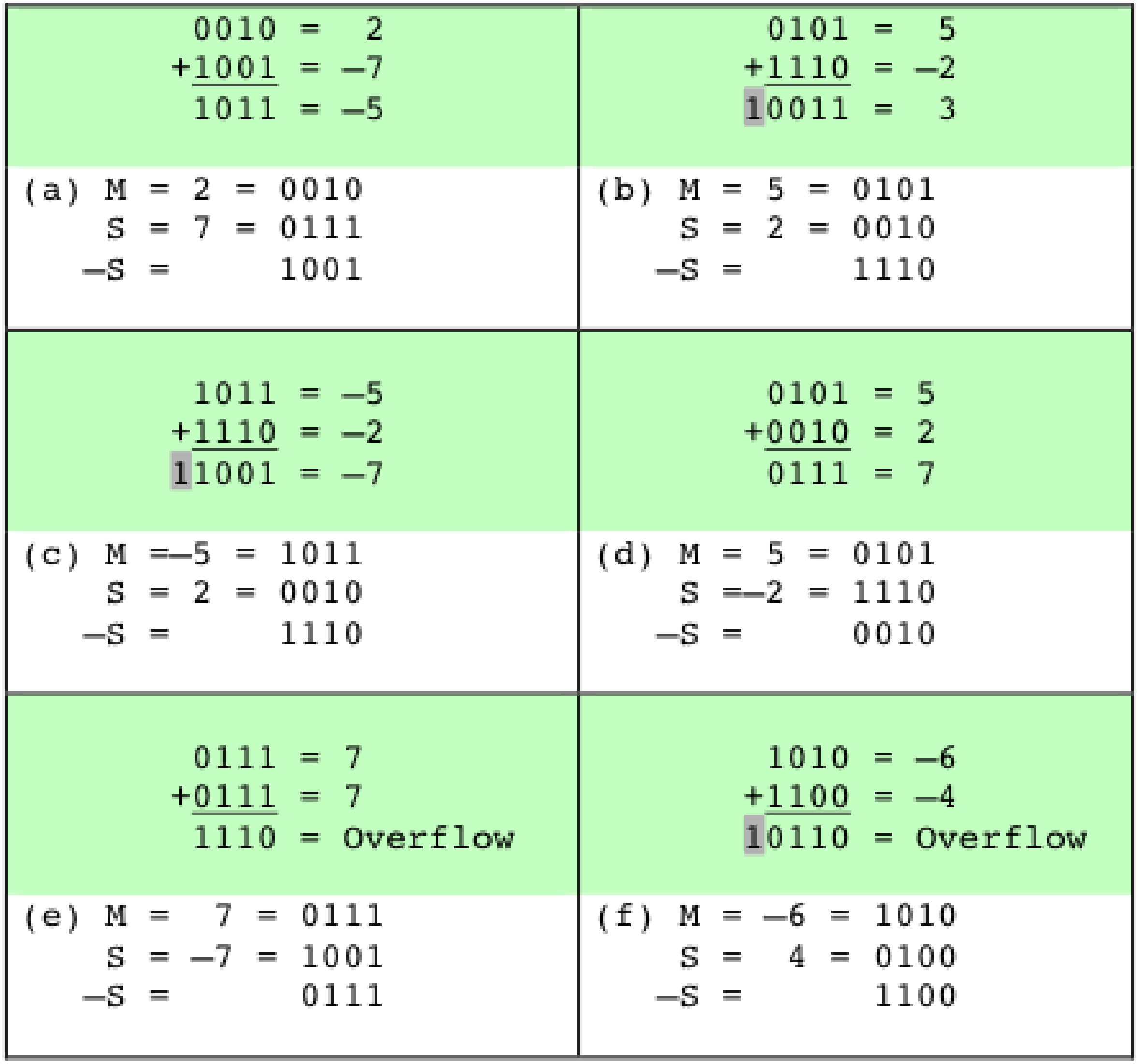
OVERFLOW RULE:

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.

Subtraction



```
   0010 =  2                 0101 =  5
  +1001 = –7                +1110 = –2
   1011 = –5                10011 =  3

(a) M = 2 = 0010          (b) M = 5 = 0101
    S = 7 = 0111              S = 2 = 0010
   –S =     1001             –S =     1110

   1011 = –5                 0101 = 5
  +1110 = –2                +0010 = 2
  11001 = –7                 0111 = 7

(c) M =–5 = 1011          (d) M = 5 = 0101
    S = 2 = 0010              S =–2 = 1110
   –S =     1110             –S =     0010

   0111 = 7                  1010 = –6
  +0111 = 7                 +1100 = –4
   1110 = Overflow          10110 = Overflow

(e) M =  7 = 0111         (f) M = –6 = 1010
    S = –7 = 1001             S =  4 = 0100
   –S =      0111            –S =      1100
```

Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M – S)

AAiT

Geometric Depiction of Twos Complement Integers

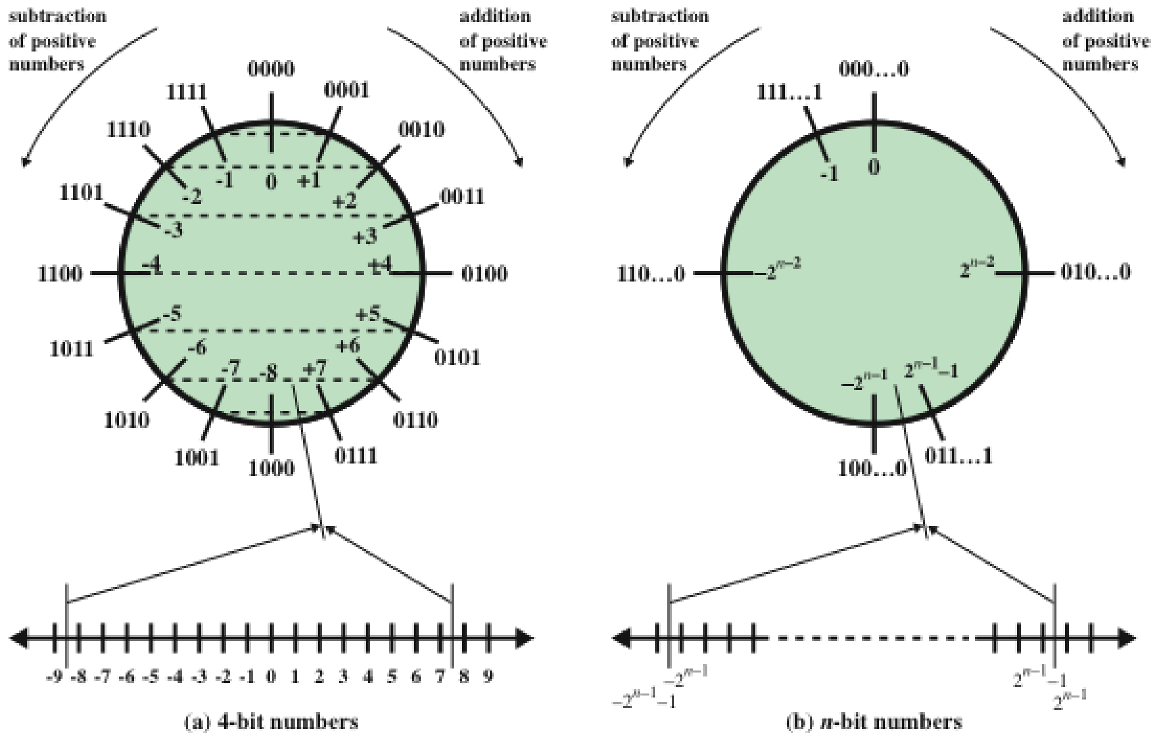
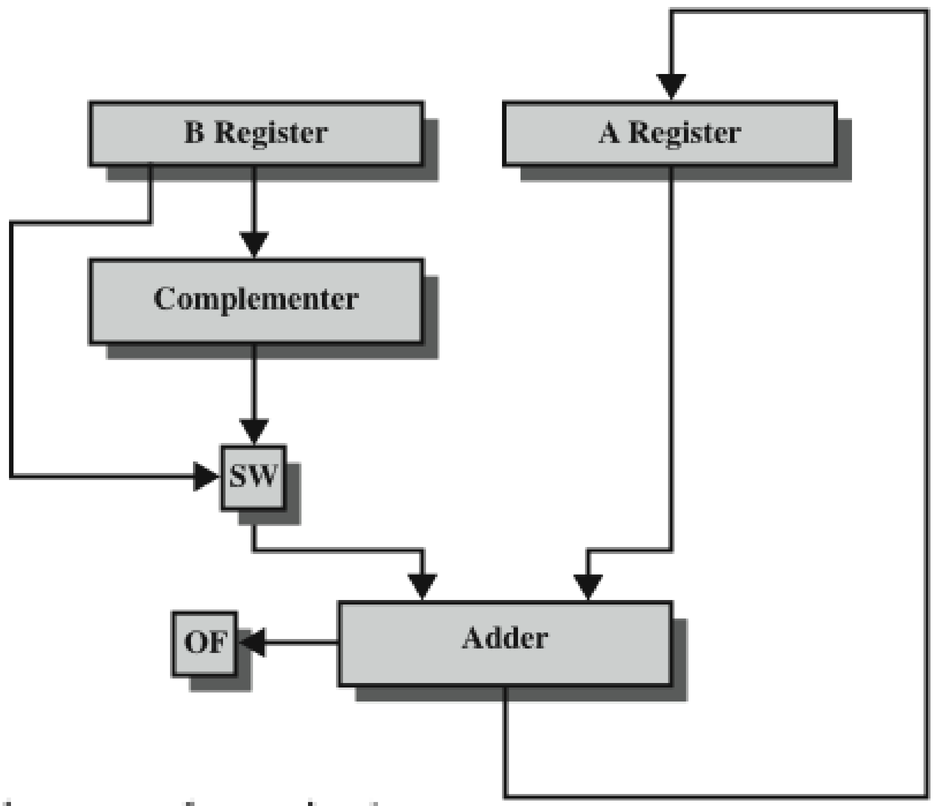


Figure 10.5 Geometric Depiction of Twos Complement Integers

Hardware for Addition and Subtraction



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

Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

Multiplication

Most common techniques for multiplication, as might be carried out using paper and pencil

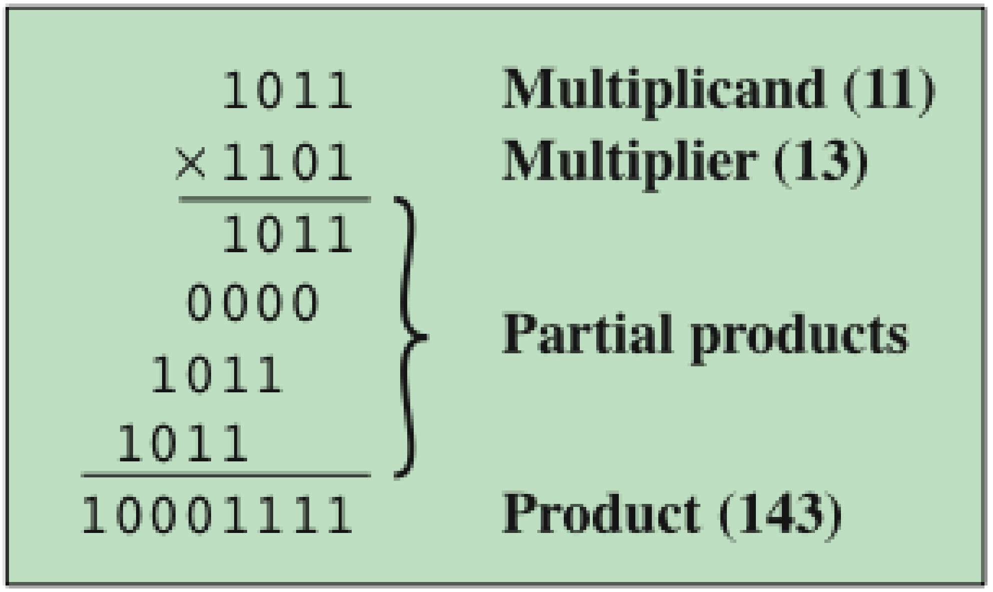
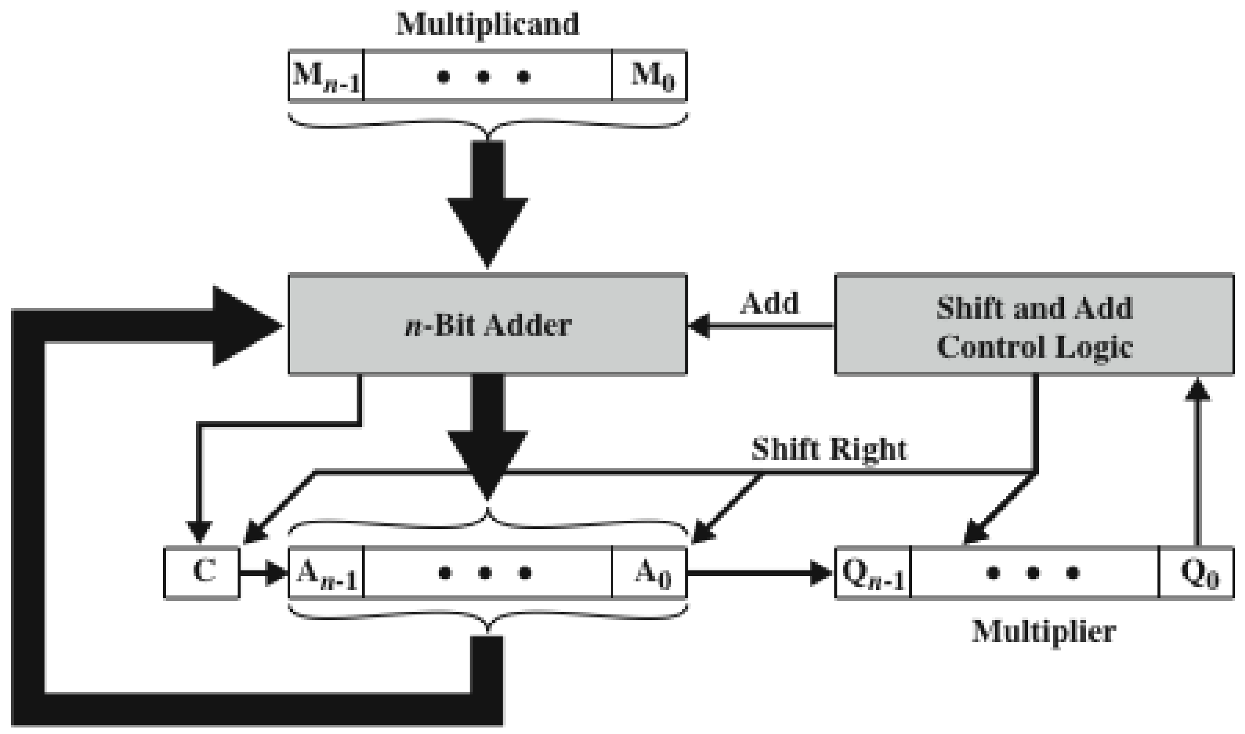
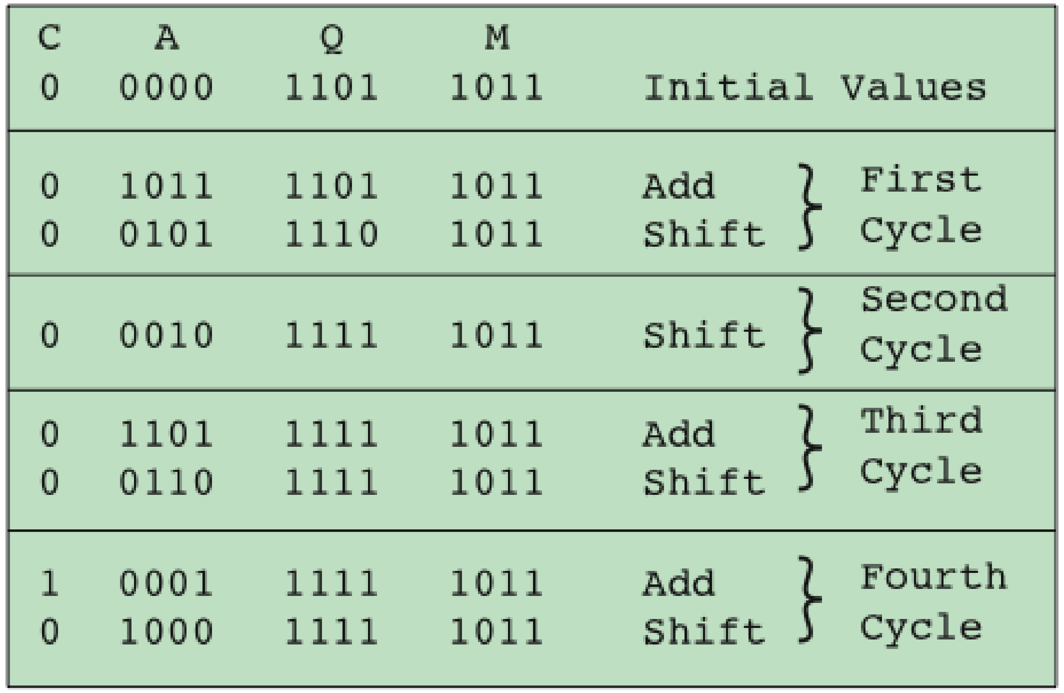

```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
   -----
    1011  ⎫
   0000   ⎬  Partial products
  1011    ⎪
 1011     ⎭
--------
10001111     Product (143)
```

Figure 10.7 Multiplication of Unsigned Binary Integers

Hardware Implementation of Unsigned Binary Multiplication



(a) Block Diagram



C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

Flowchart for Unsigned Binary Multiplication

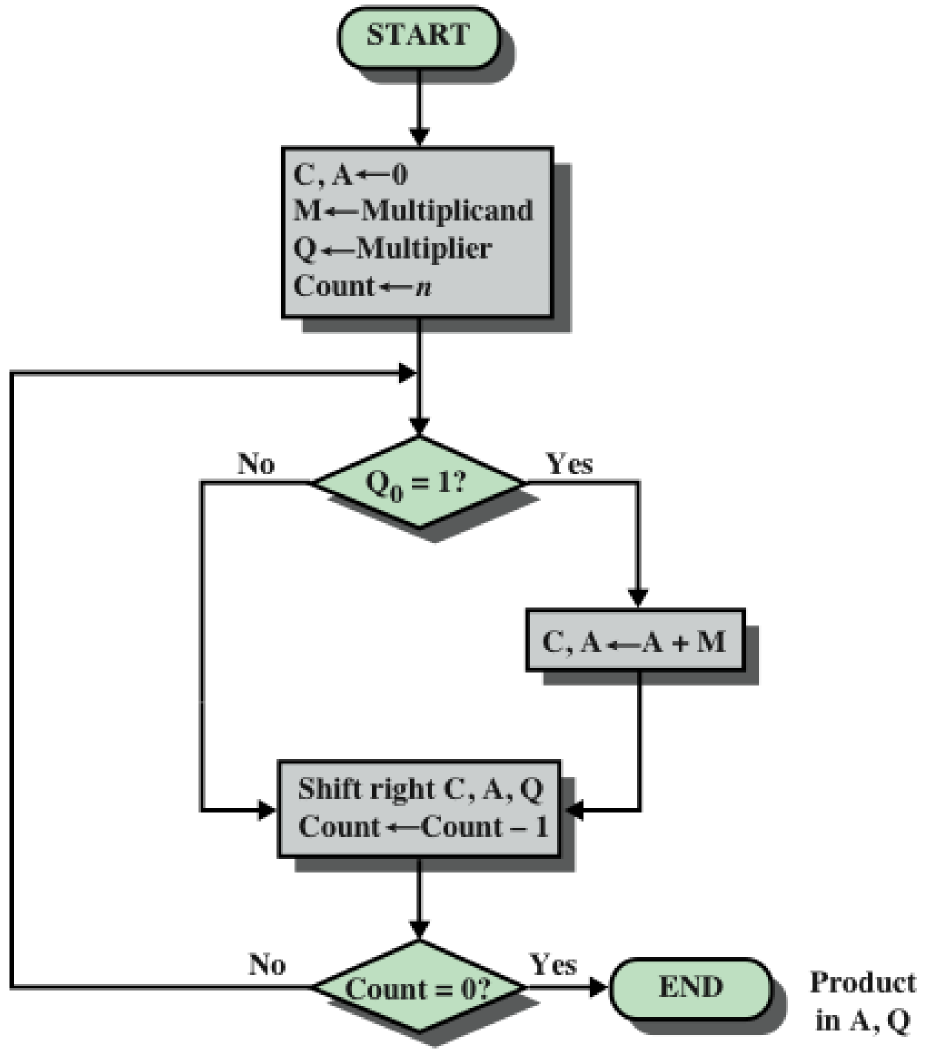
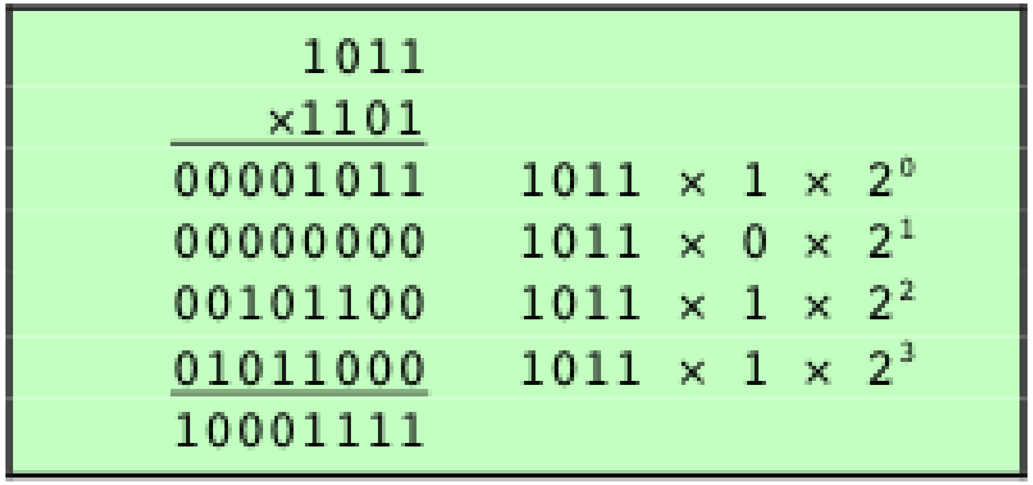


Figure 10.9 Flowchart for Unsigned Binary Multiplication

Twos Complement Multiplication

- We multiplied 11 (1011) by 13 (1101) to get 143 (10001111).
- If we interpret these as twos complement numbers, we have -5 (1011) times -3 (1101) equals -113 (10001111) which is wrong.
- This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative.

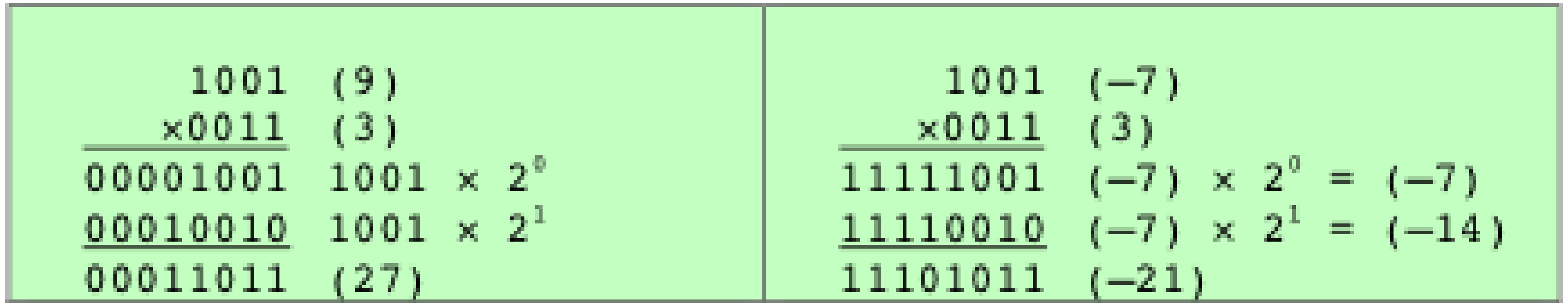


```
    1011
   ×1101
--------
00001011    1011 × 1 × 2^0
00000000    1011 × 0 × 2^1
00101100    1011 × 1 × 2^2
01011000    1011 × 1 × 2^3
--------
10001111
```

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Comparison

If 1001 is interpreted as the complement value -7, then each partial product must be a negative twos complement number of 2n (8) bits, as shown in Figure 10.11b. Note that this is accomplished by padding out each partial product to the left with binary 1s.



```
    1001  (9)                       1001  (-7)
   ×0011  (3)                      ×0011  (3)
00001001  1001 × 2^0            11111001  (-7) × 2^0 = (-7)
00010010  1001 × 2^1            11110010  (-7) × 2^1 = (-14)
00011011  (27)                  11101011  (-21)
```

(a) Unsigned integers

(b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

AAiT

Exercise

- Represent -4 and -17 in twos complement. Re write the answer in hexadecimal.

- Multiply 1100 by 1001:
  a. If the numbers are unsigned integers
  b. If the numbers are signed twos complement
- Multiply -8 by 3 in binary

Booth’s Algorithm

AAiT

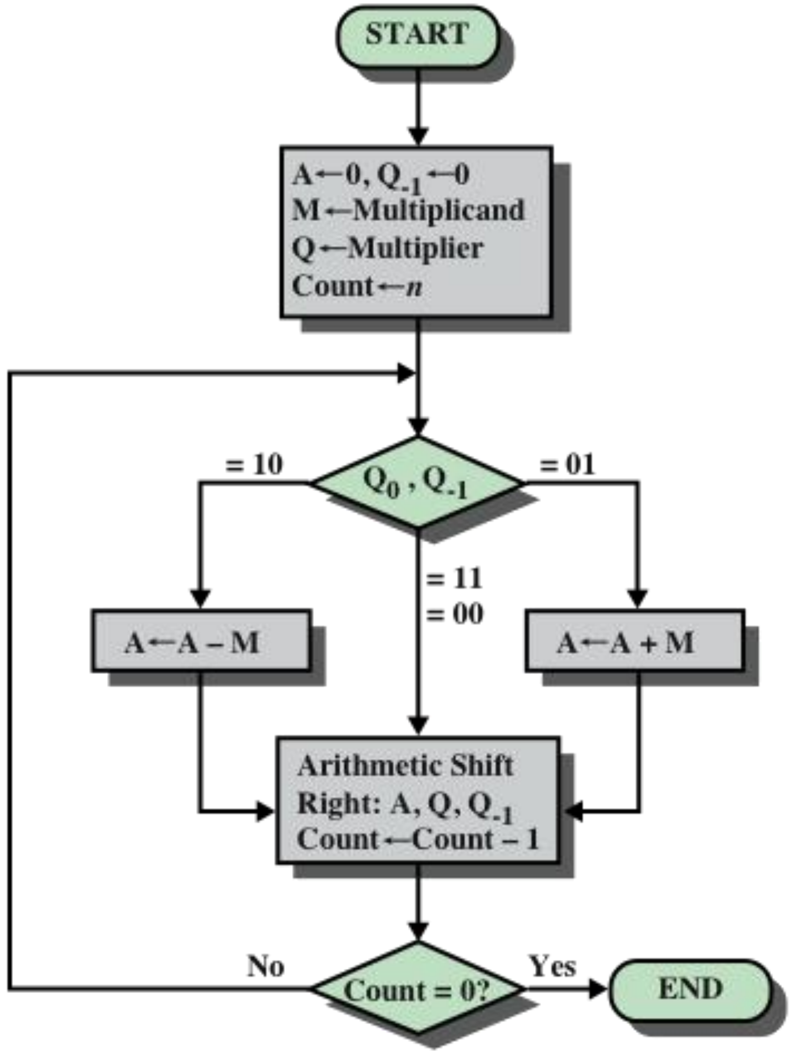
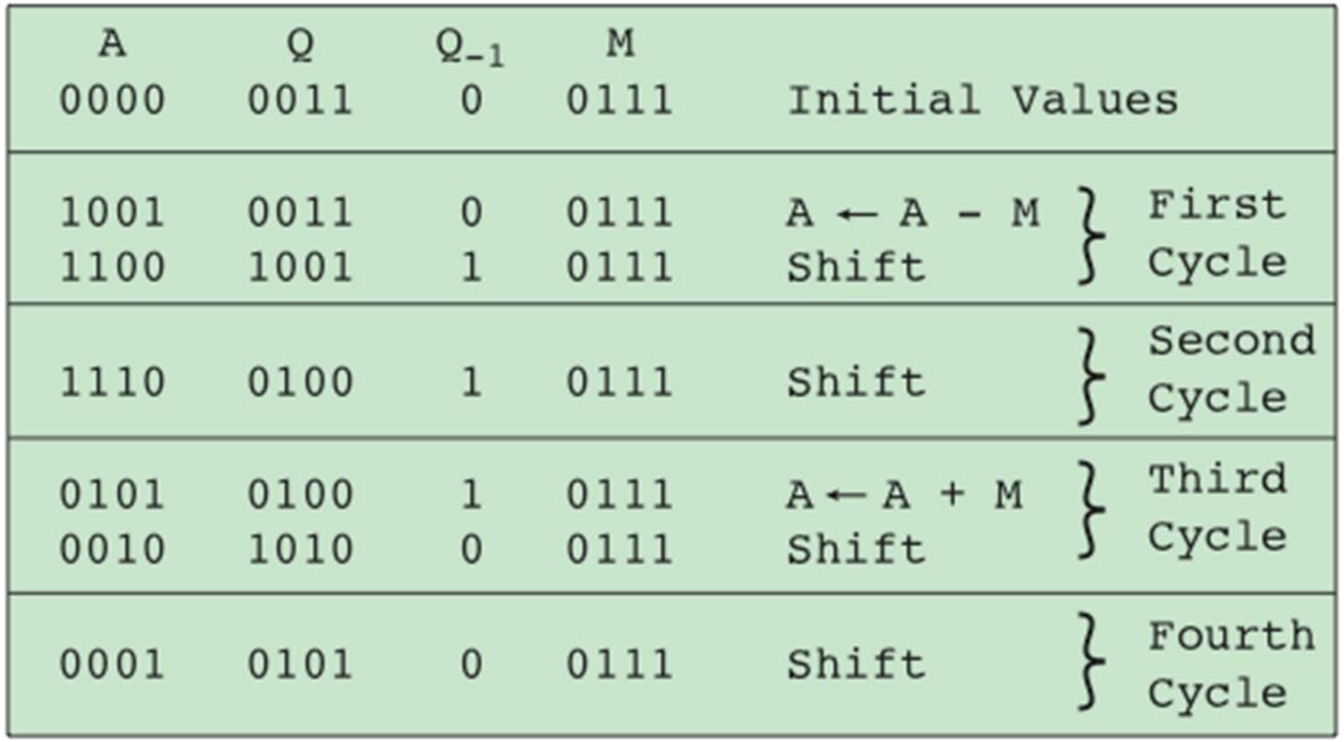


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

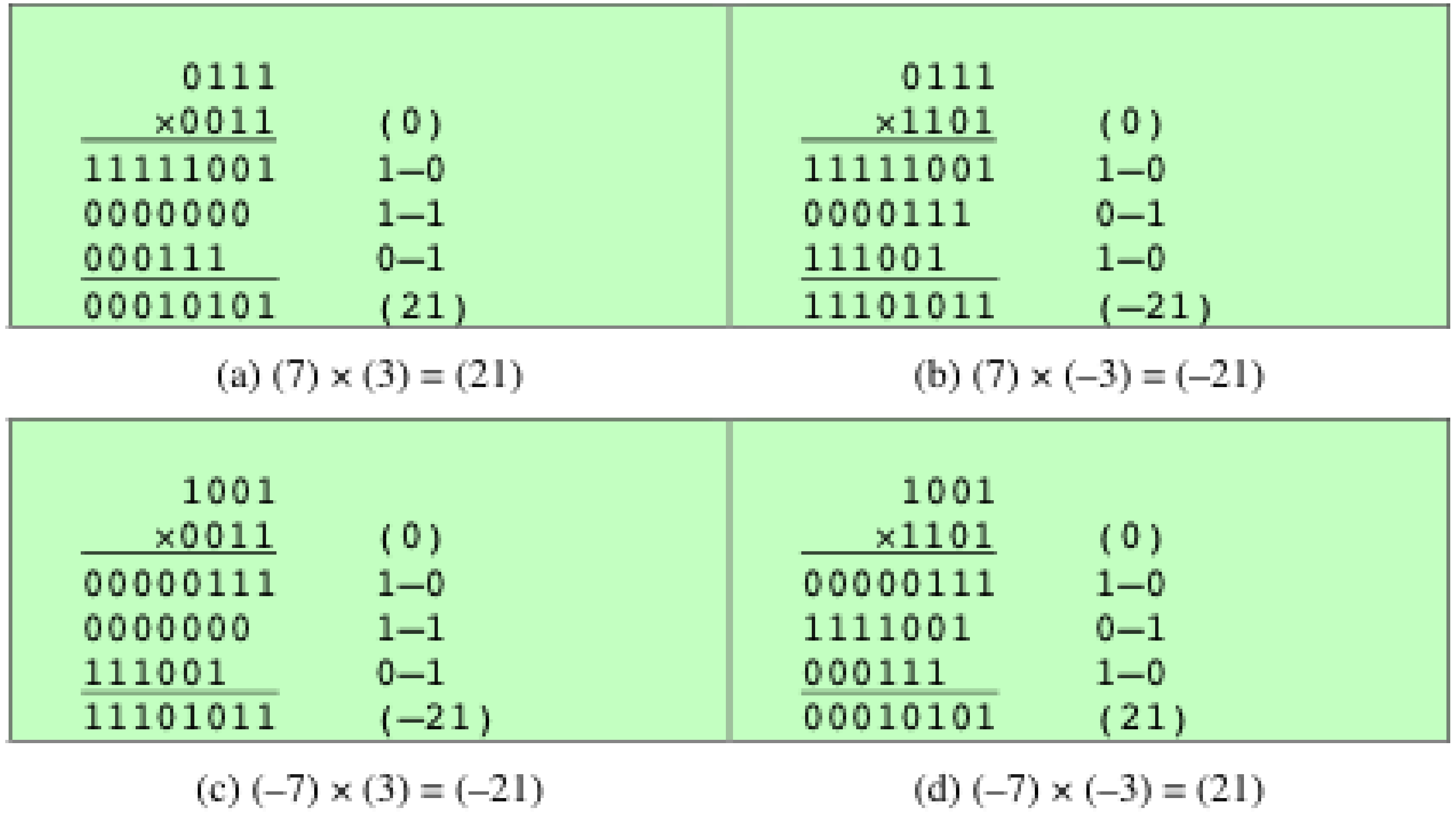
Note that the shift is arithmetic shift: A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}



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 10.13 Example of Booth's Algorithm (7× 3)

Examples Using Booth’s Algorithm



```
    0111
   ×0011      (0)
11111001      1–0
0000000       1–1
000111        0–1
00010101      (21)
```

(a) (7) × (3) = (21)

```
    0111
   ×1101      (0)
11111001      1–0
0000111       0–1
111001        1–0
11101011      (–21)
```

(b) (7) × (–3) = (–21)

```
    1001
   ×0011      (0)
00000111      1–0
0000000       1–1
111001        0–1
11101011      (–21)
```

(c) (–7) × (3) = (–21)

```
    1001
   ×1101      (0)
00000111      1–0
1111001       0–1
000111        1–0
00010101      (21)
```

(d) (–7) × (–3) = (21)

Figure 10.14 Examples Using Booth's Algorithm

- Figure 10.14 gives other examples of the algorithm.
- As can be seen, it works with any combination of positive and negative numbers.
- Note also the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of only one addition or subtraction per block.

AAiT

Division

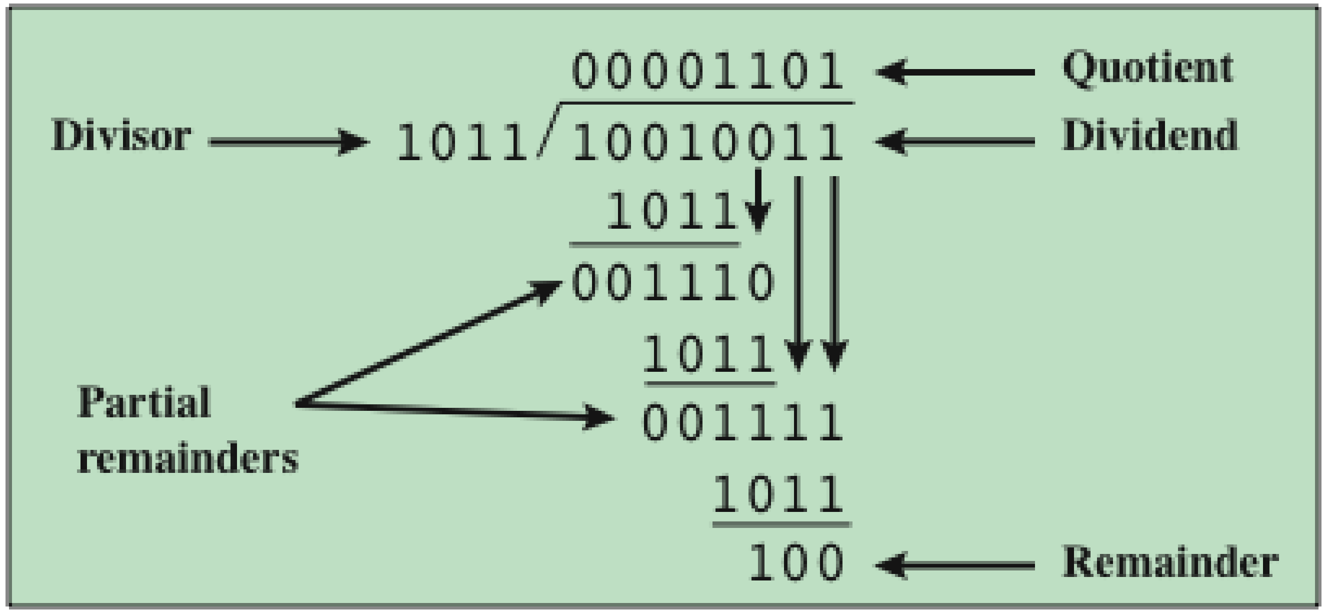


Figure 10.15 Example of Division of Unsigned Binary Integers

Flowchart for Unsigned Binary Division

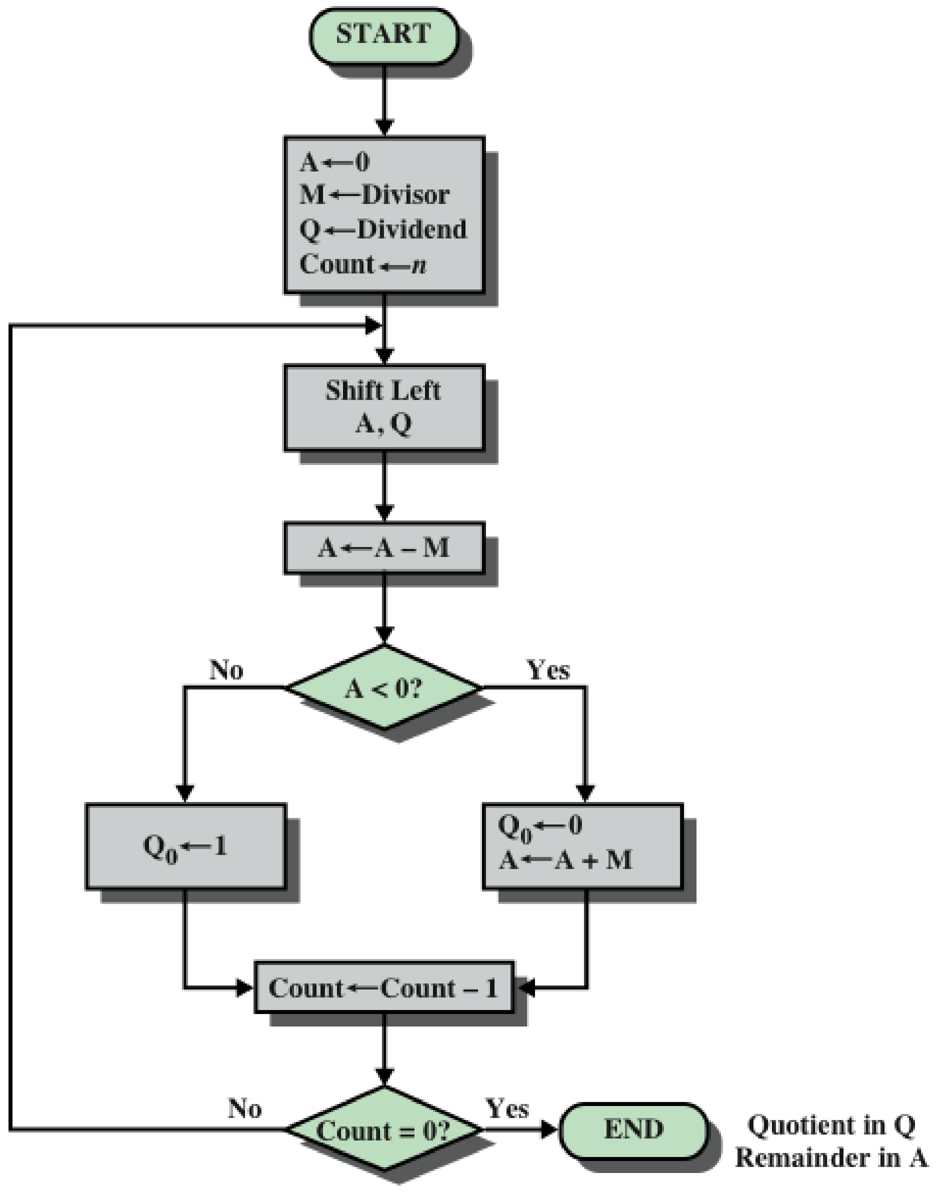
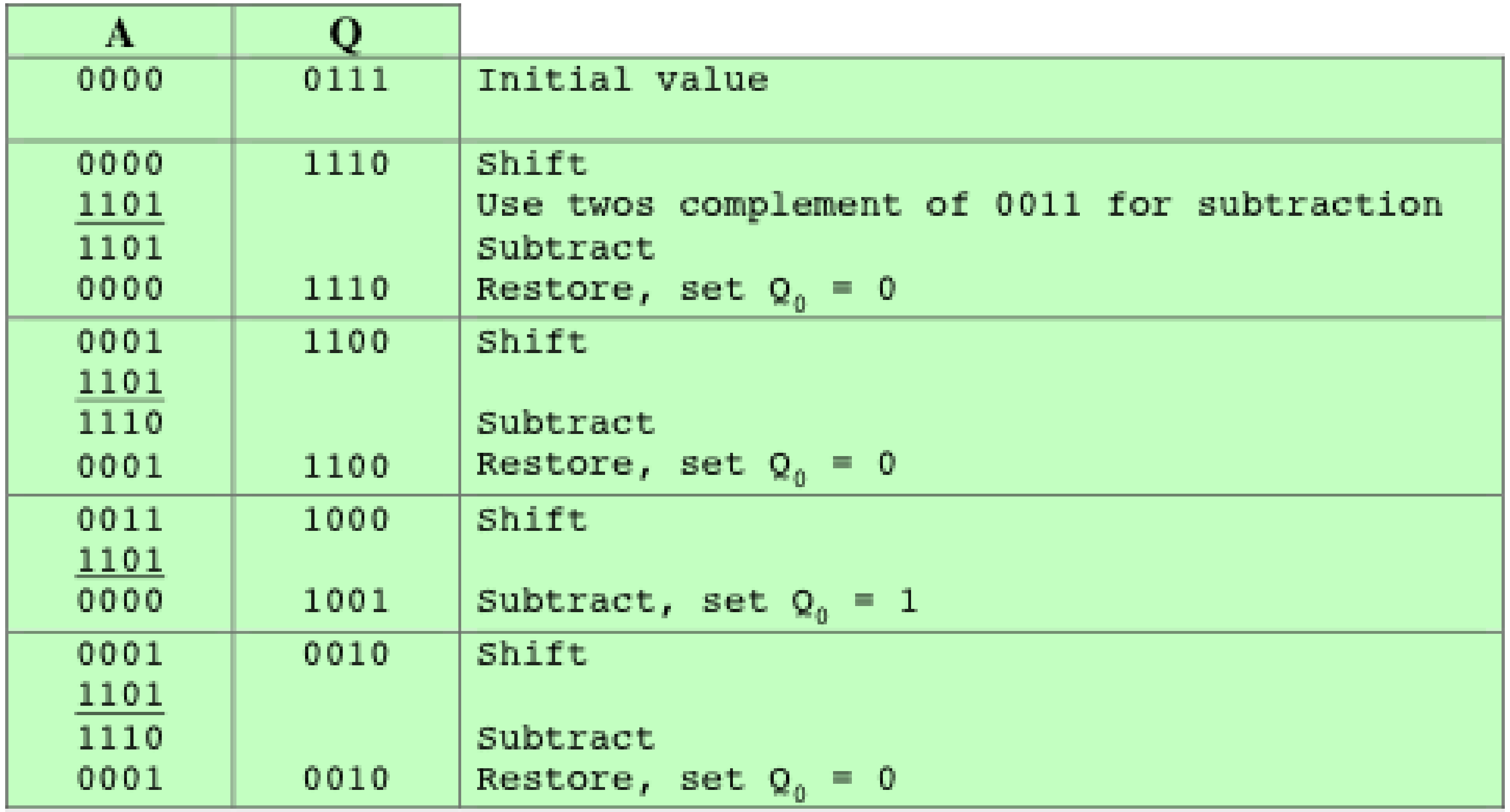


Figure 10.16 Flowchart for Unsigned Binary Division

Example of Restoring Twos Complement Division



A	Q	
0000	0111	Initial value
0000	1110	Shift
$\underline{1101}$		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
$\underline{1101}$		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
$\underline{1101}$		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
$\underline{1101}$		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 10.17 Example of Restoring Twos Complement Division (7/3)

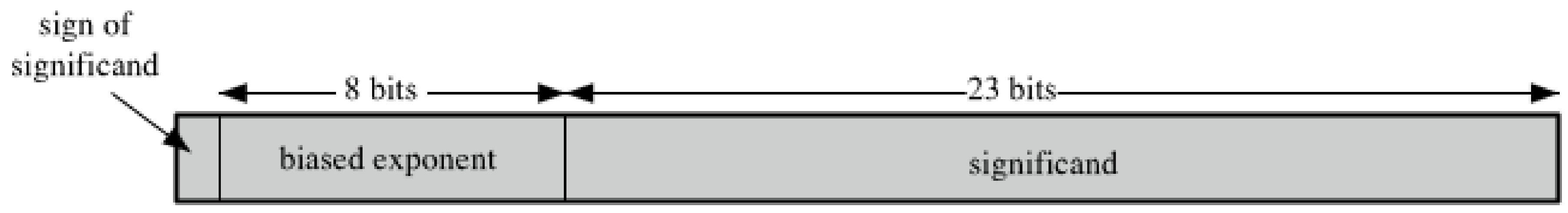
+

Floating-Point Representation

Principles

- With a fixed-point notation it is possible to represent a range of positive and negative integers centered on or near 0
- By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well
- Limitations:
  - Very large numbers cannot be represented nor can very small fractions
  - The fractional part of the quotient in a division of two large numbers could be lost

Typical 32-Bit Floating-Point Format



(a) Format

Typically, the bias equals (2^{k-1} - 1), 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), the true exponent values are in the range -127 to +128.

$$
\begin{aligned}
1.1010001 \times 2^{10100} &= 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}\\
-1.1010001 \times 2^{10100} &= 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}\\
1.1010001 \times 2^{-10100} &= 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}\\
-1.1010001 \times 2^{-10100} &= 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}
\end{aligned}
$$

(b) Examples

Figure 10.18 Typical 32-Bit Floating-Point Format

Floating-Point

Significand

- The final portion of the word
- Any floating-point number can be expressed in many ways

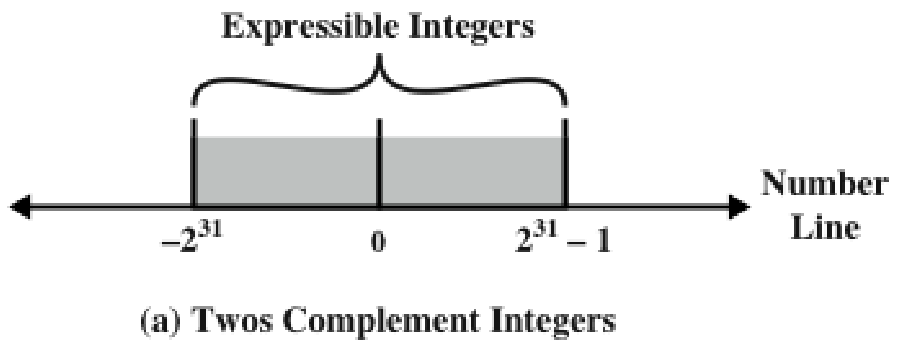
The following are equivalent, where the significand is expressed in binary form:

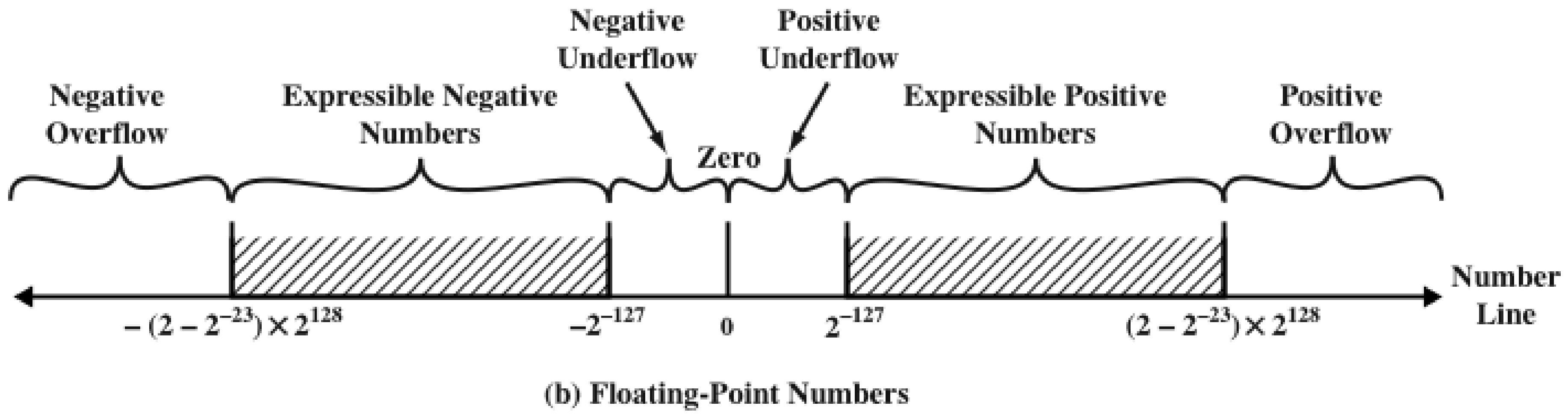
$$0.110 * 2^5$$
$$110 * 2^2$$
$$0.0110 * 2^6$$

- *Normal number*
  - The most significant digit of the significand is nonzero

AAiT

Expressible Numbers





Negative Overflow
Expressible Negative Numbers
Negative Underflow
Zero
Positive Underflow
Expressible Positive Numbers
Positive Overflow
Number Line
$-(2 - 2^{-23}) \times 2^{128}$
-2^{-127}
0
2^{-127}
$(2 - 2^{-23}) \times 2^{128}$
(b) Floating-Point Numbers

Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

AAiT

Density of Floating-Point Numbers

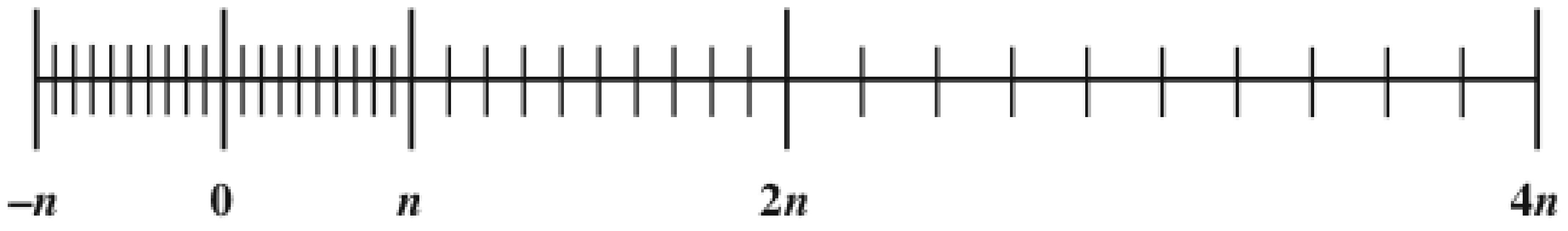


Figure 10.20 Density of Floating-Point Numbers

- Also, note that the numbers represented in floating-point notation are not spaced evenly along the number line, as are fixed-point numbers. The possible values get closer together near the origin and farther apart as you move away, as shown in Figure 10.20.

IEEE Standard 754

Most important floating-point representation is defined

Standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs

Standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors

IEEE 754-2008 covers both binary and decimal floating-point representations

IEEE 754 Formats

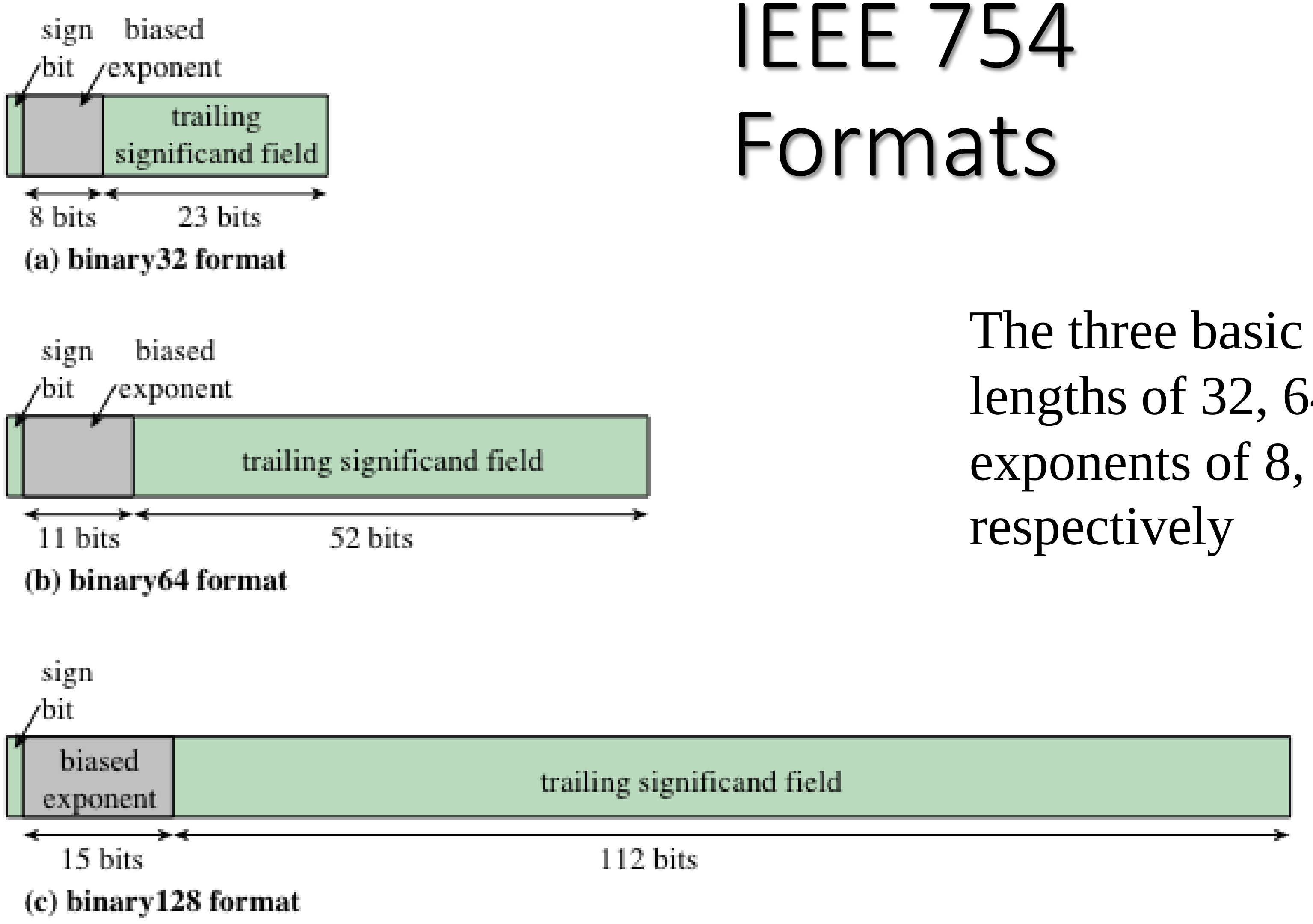


Figure 10.21 IEEE 754 Formats

The three basic binary formats have bit lengths of 32, 64, and 128 bits, with exponents of 8, 11, and 15 bits, respectively

Additional Formats

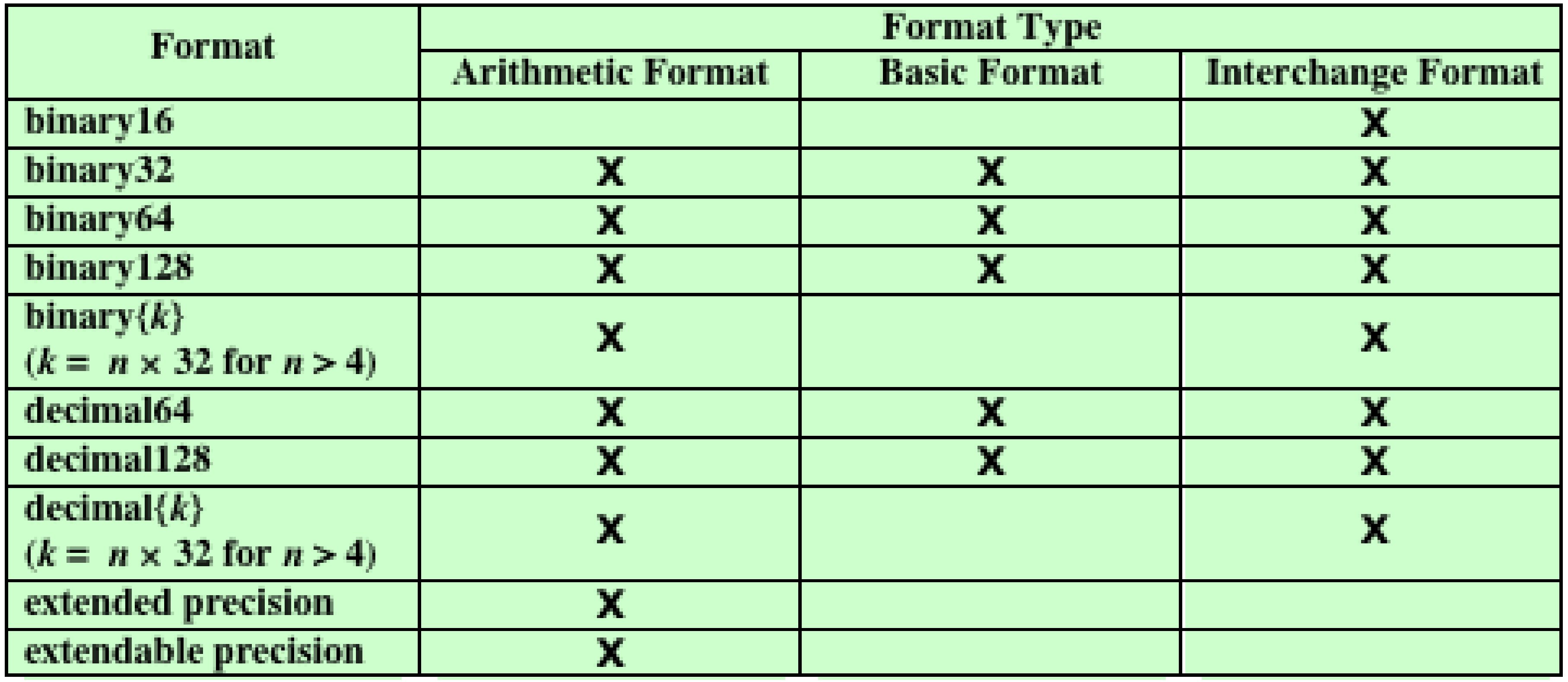
Extended Precision Formats

- Provide additional bits in the exponent (extended range) and in the significand (extended precision)
- Lessens the chance of a final result that has been contaminated by excessive roundoff error
- Lessens the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format
- Affords some of the benefits of a larger basic format without incurring the time penalty usually associated with higher precision

Extendable Precision Format

- Precision and range are defined under user control
- May be used for intermediate calculations but the standard places no constraint or format or length

Table 10.4 IEEE Formats

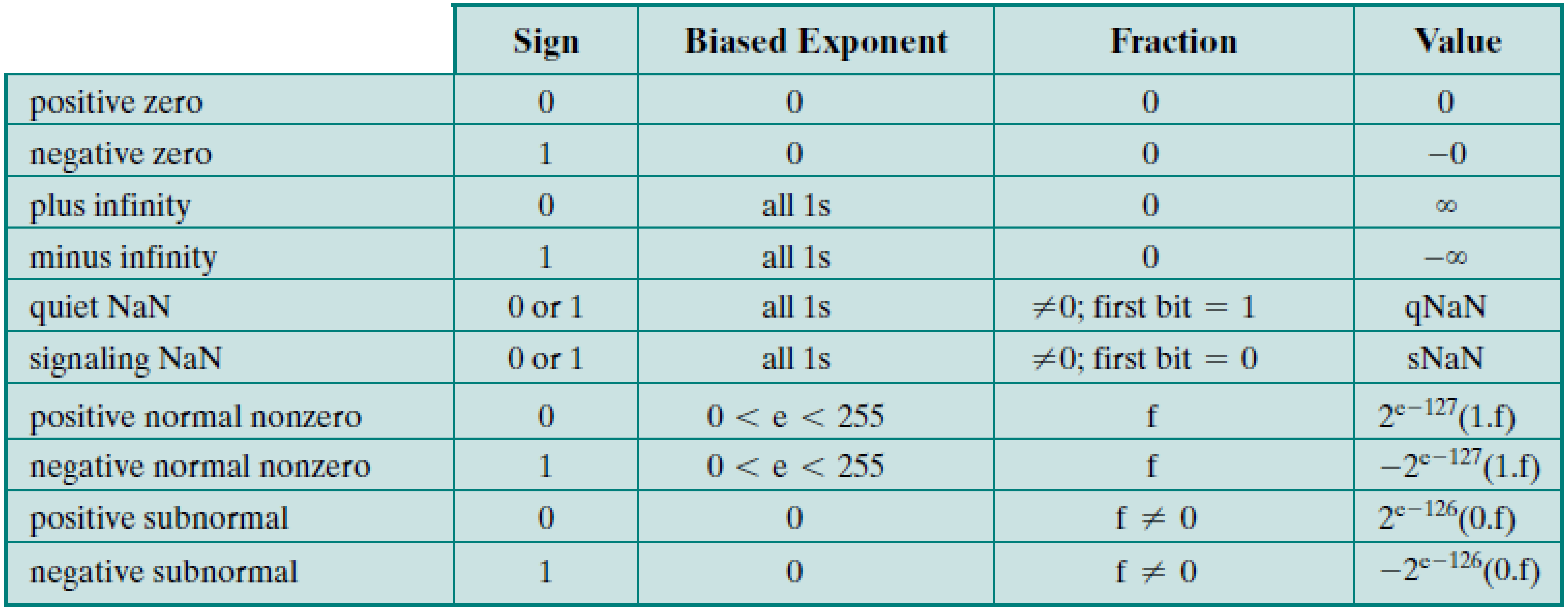


Format	Format Type		
	Arithmetic Format	Basic Format	Interchange Format
binary16			X
binary32	X	X	X
binary64	X	X	X
binary128	X	X	X
binary{k} ($k = n \times 32$ for $n > 4$)	X		X
decimal64	X	X	X
decimal128	X	X	X
decimal{k} ($k = n \times 32$ for $n > 4$)	X		X
extended precision	X		
extendable precision	X		

Table 10.4 IEEE Formats

Interpretation of IEEE 754 Floating-Point Numbers

(a) binary 32 format

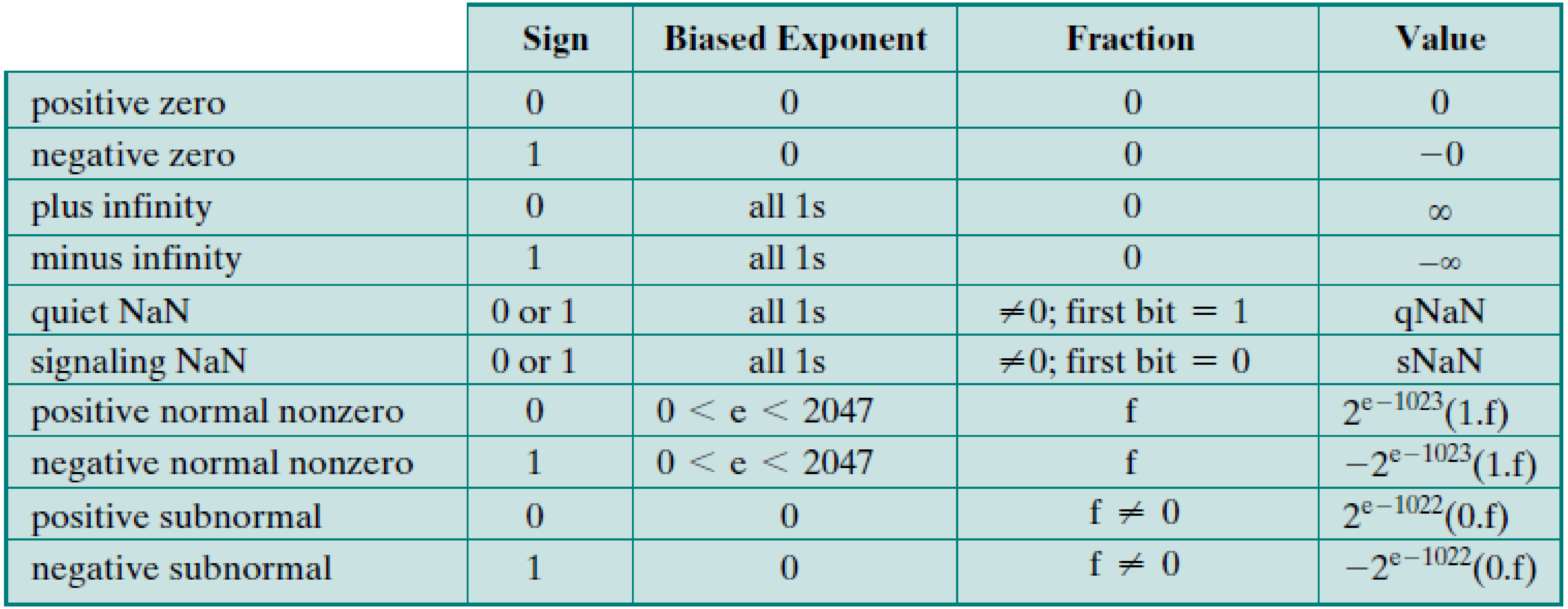


	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	-0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$; first bit $= 1$	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$; first bit $= 0$	sNaN
positive normal nonzero	0	$0 < e < 255$	f	$2^{e-127}(1.f)$
negative normal nonzero	1	$0 < e < 255$	f	$-2^{e-127}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2^{e-126}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2^{e-126}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 1 of 3)

Interpretation of IEEE 754 Floating-Point Numbers

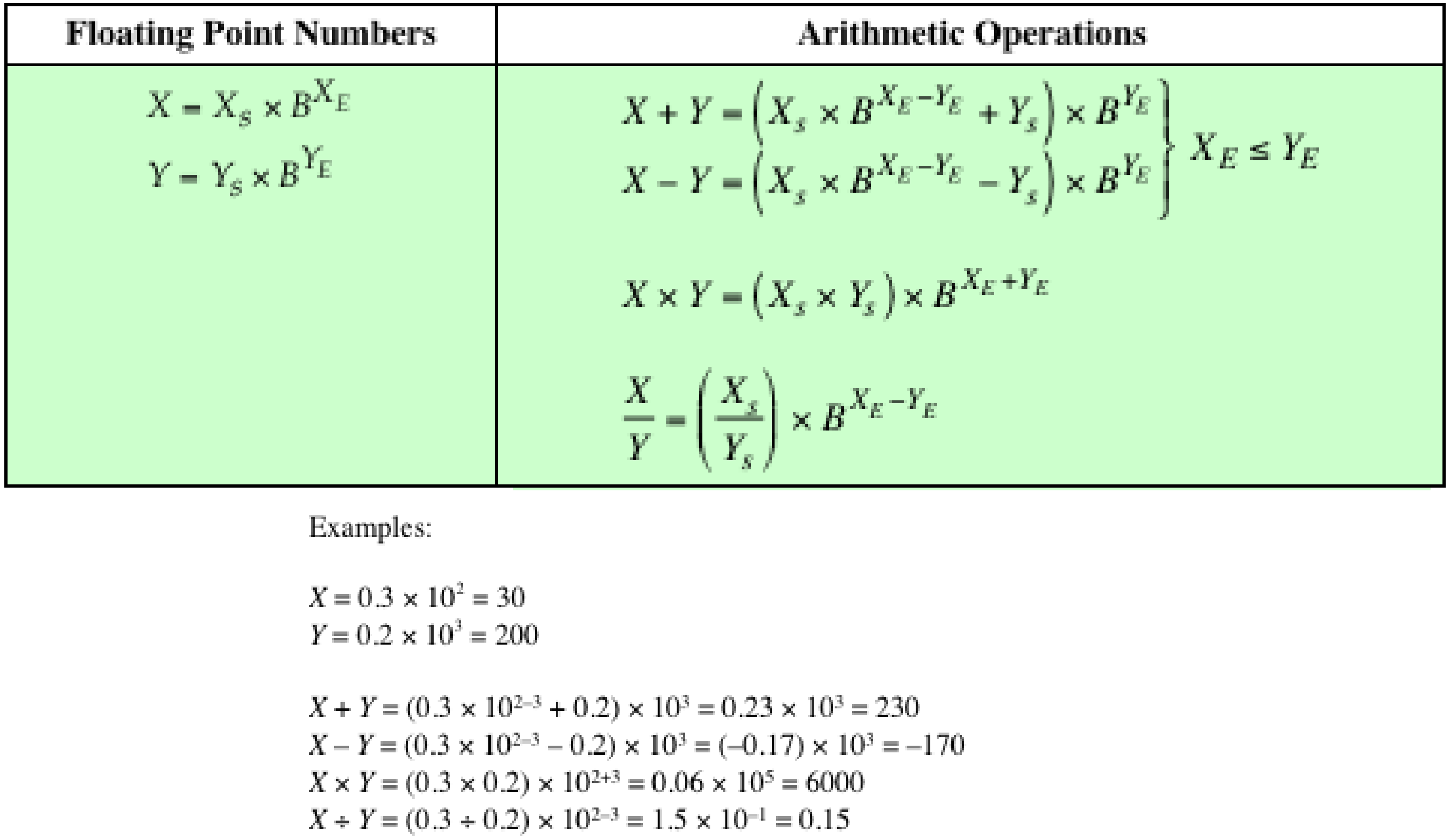
(b) binary 64 format



	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	-0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$; first bit $= 1$	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$; first bit $= 0$	sNaN
positive normal nonzero	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
negative normal nonzero	1	$0 < e < 2047$	f	$-2^{e-1023}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2^{e-1022}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2^{e-1022}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 2 of 3)

Table 10.6 Floating-Point Numbers and Arithmetic Operations



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

Examples:

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

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

Floating-Point Addition and Subtraction

AAiT

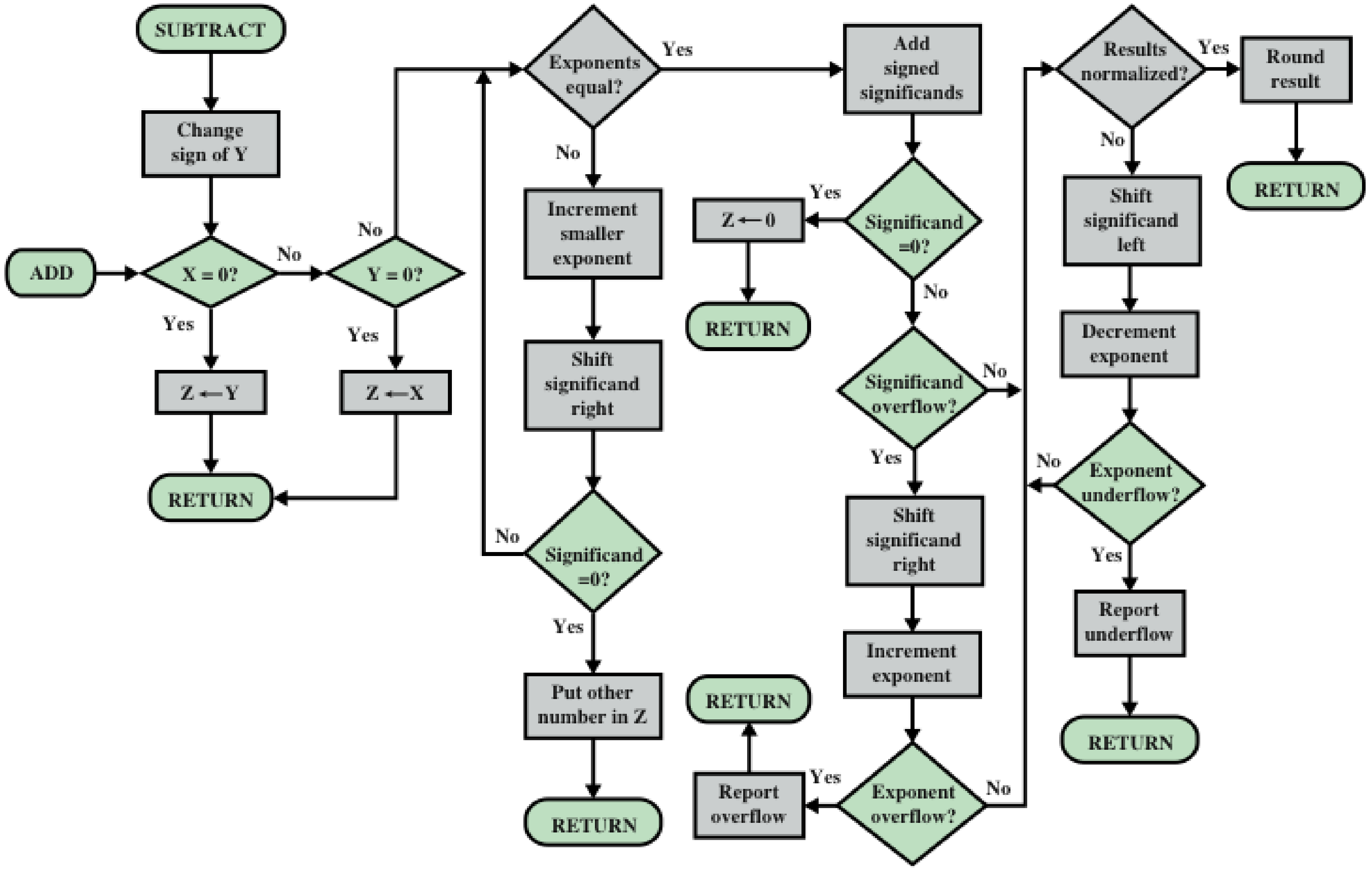


Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

AAiT

Floating-Point Multiplication

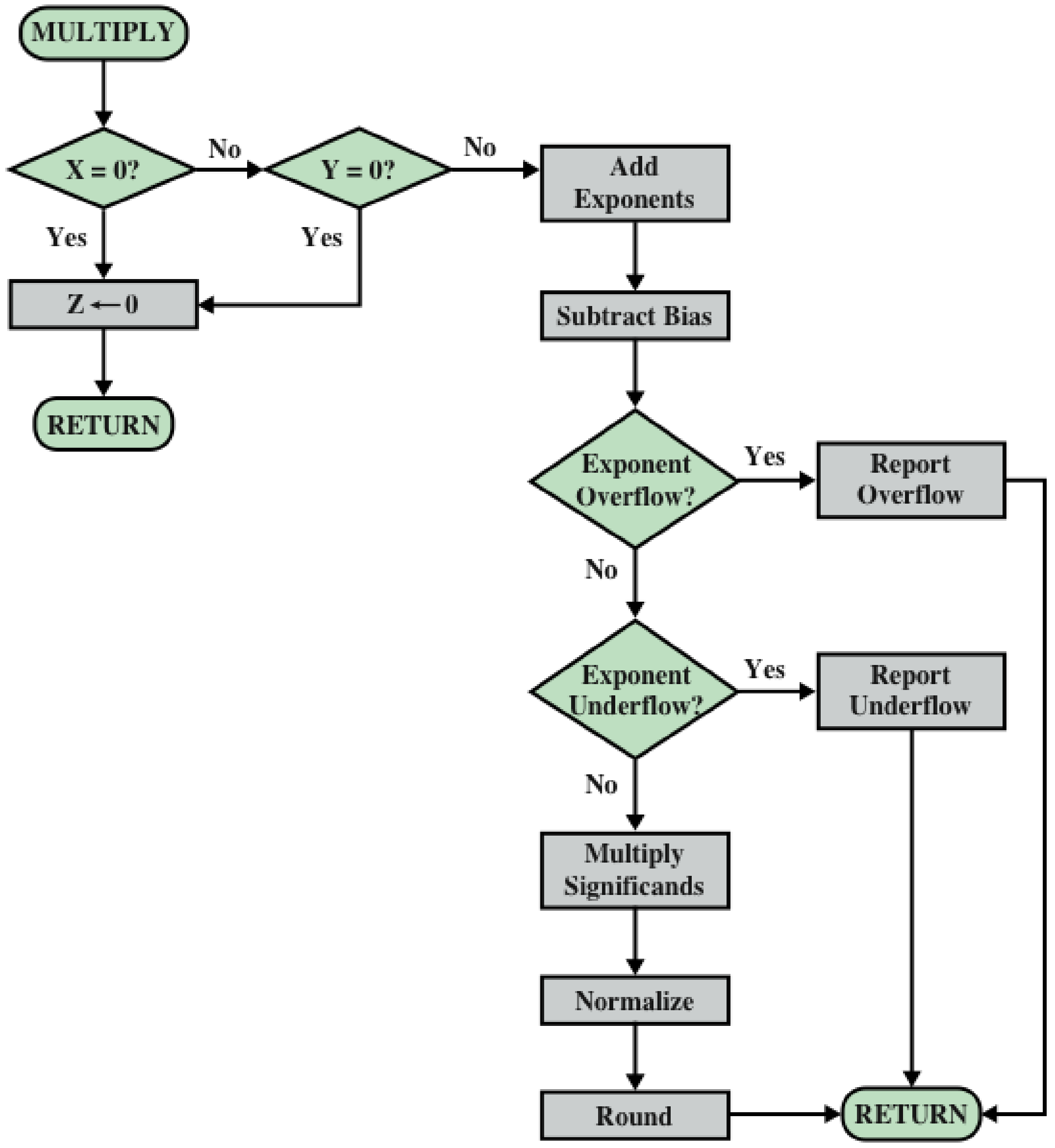


Figure 10.23 Floating-Point Multiplication (Z← X× Y)

AAiT

Floating-Point Division

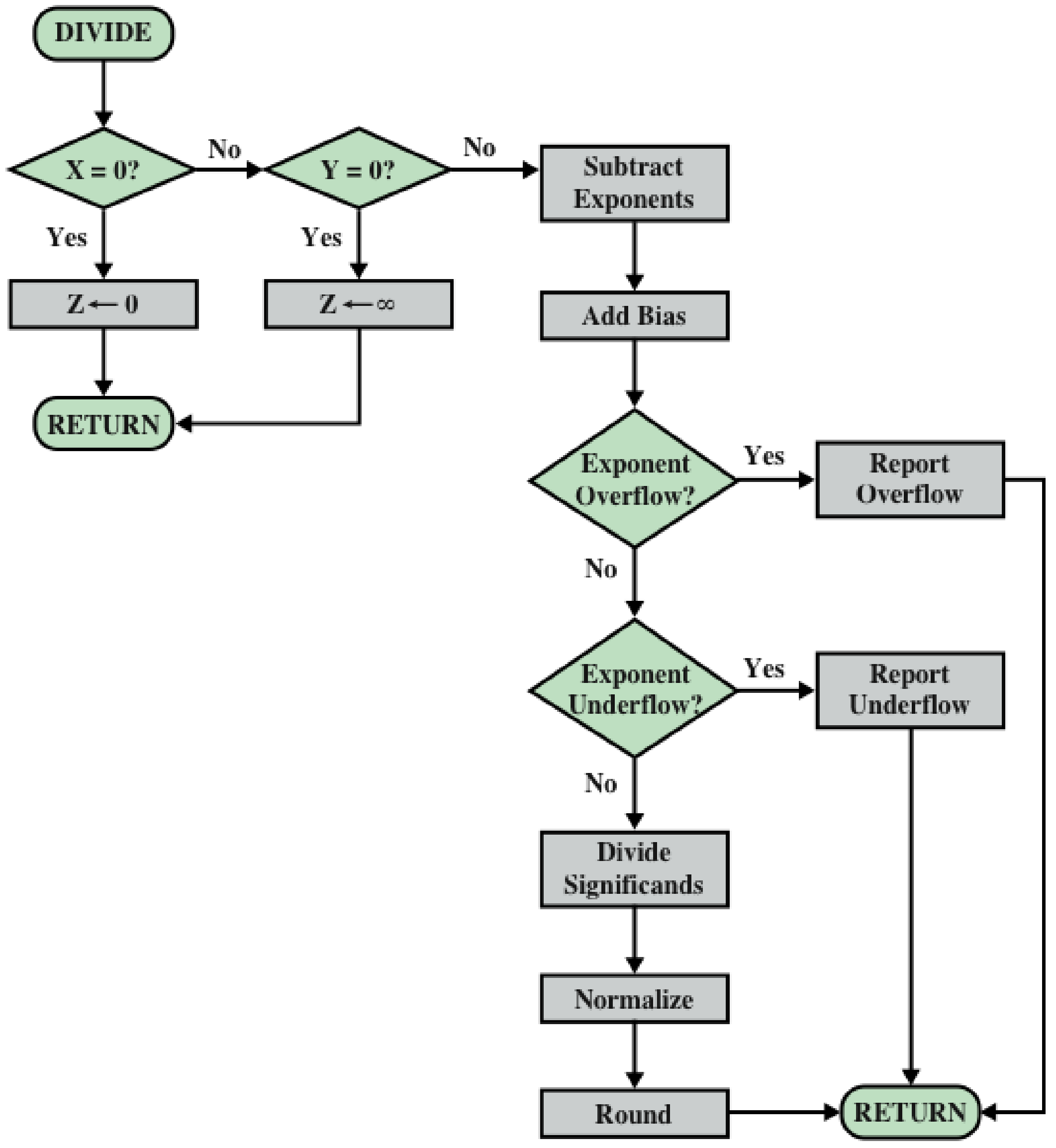


Figure 10.24 Floating-Point Division (Z← X/Y)

Summary

Chapter 10

- ALU
- Integer representation
  - Sign-magnitude representation
  - Twos complement representation
  - Range extension
  - Fixed-point representation
- Floating-point representation
  - Principles
  - IEEE standard for binary floating-point representation

Computer Arithmetic

- Integer arithmetic
  - Negation
  - Addition and subtraction
  - Multiplication
  - Division
- Floating-point arithmetic
  - Addition and subtraction
  - Multiplication and division
  - Precision consideration
  - IEEE standard for binary floating-point arithmetic