Unit III

Ms.Mayuri D. Kulkarni
Assistant Professor
Department of Computer Engineering
SVKM's IoT,Dhule

Content

- The arithmetic & logic unit
- Integer representation
- Integer arithmetic
- Floating point representation
- Floating point arithmetic
- Introduction of arithmetic co-processor
- Arithmetic processor

Introduction

- Computer arithmetic is commonly performed on two very different types of numbers: integer and floating point.

THE ARITHMETIC AND LOGIC UNIT

- The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out.
- Data 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.
- 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. The flag values are also stored in registers within the processor. The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

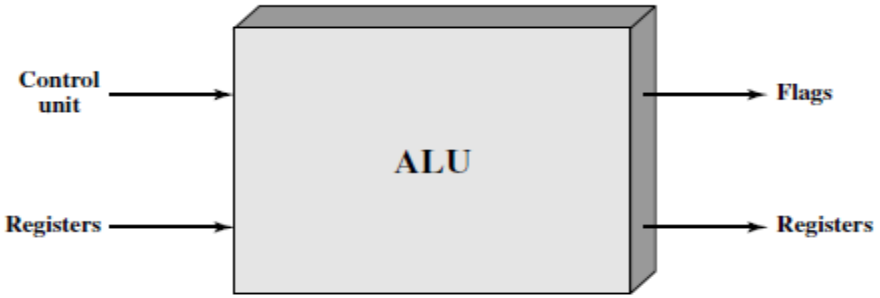


Figure 9.1 ALU Inputs and Outputs

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.**
- 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.
- In general, if an n-bit sequence of binary digits $a_{n-1},\ldots,a_0$ 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

The general case can be expressed as follows:

Sign Magnitude

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

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

Sign-Magnitude Representation

- 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.
- Another drawback is that there are two representations of 0, This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation. Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU.

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.
- Consider an n-bit integer, A, in twos complement representation. If A is positive, then the sign bit a_{n-1}, is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude

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

Twos Complement

- The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s. We can see that the range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1}-1$ (all of the magnitude bits are 1). Any larger number would require more bits
- Now, for a **negative number** A(A<0), the sign bit a_{n-1} , is one. The remaining bits can take on any one 2^{n-1} of values .Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1} .

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

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	—

Converting between Different Bit Lengths

−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 +2 +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 +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)

2’s complement of Integer

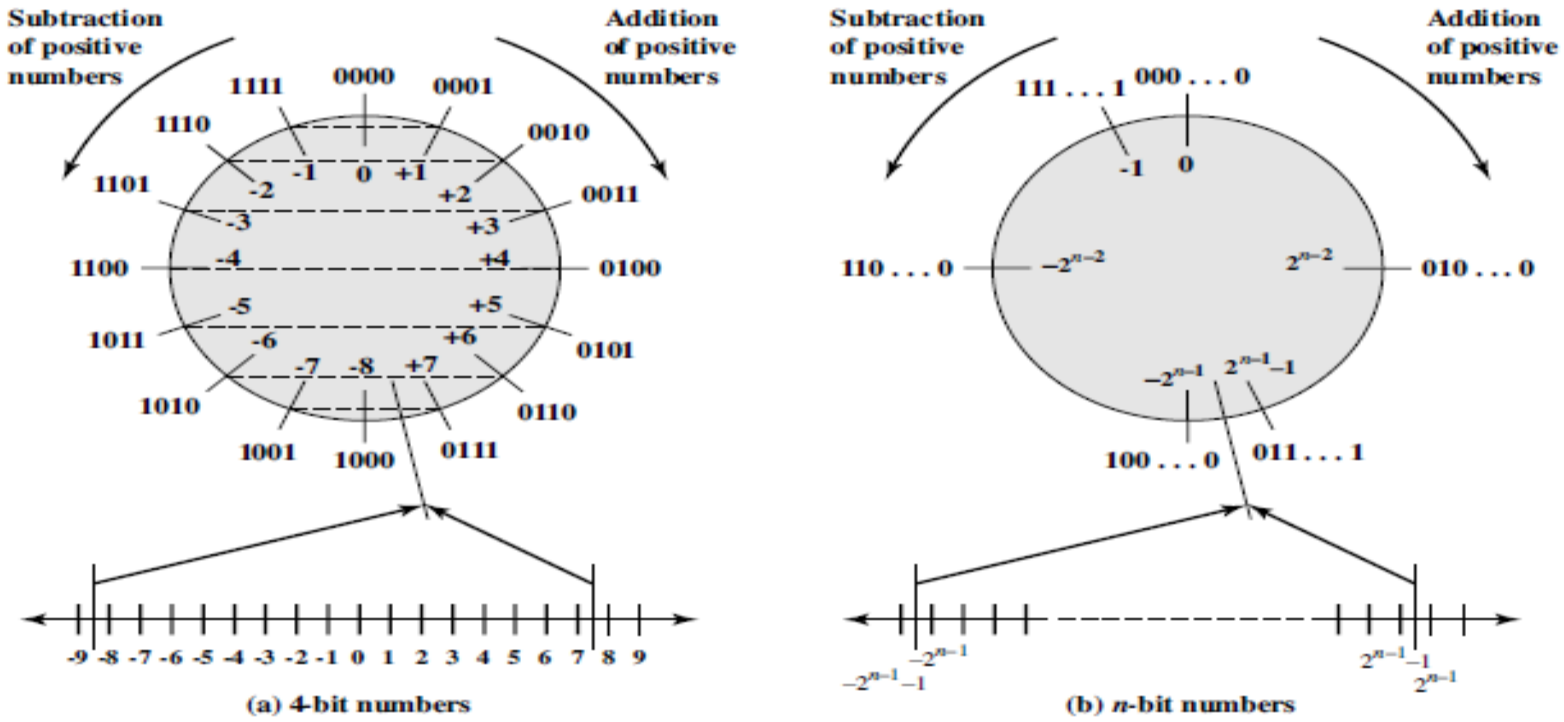


Figure 9.5 Geometric Depiction of Twos Complement Integers

INTEGER ARITHMETIC

- **Negation**
- In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. **Take the Boolean complement of each bit of the integer (including the sign** bit). That is, set each 1 to 0 and each 0 to 1.
2. **Treating the result as an unsigned binary integer, add 1.**

This two-step process is referred to as the **twos complement operation, or the t** mplement of an integer.

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

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 9.3 Addition of Numbers in Twos Complement Representation

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.

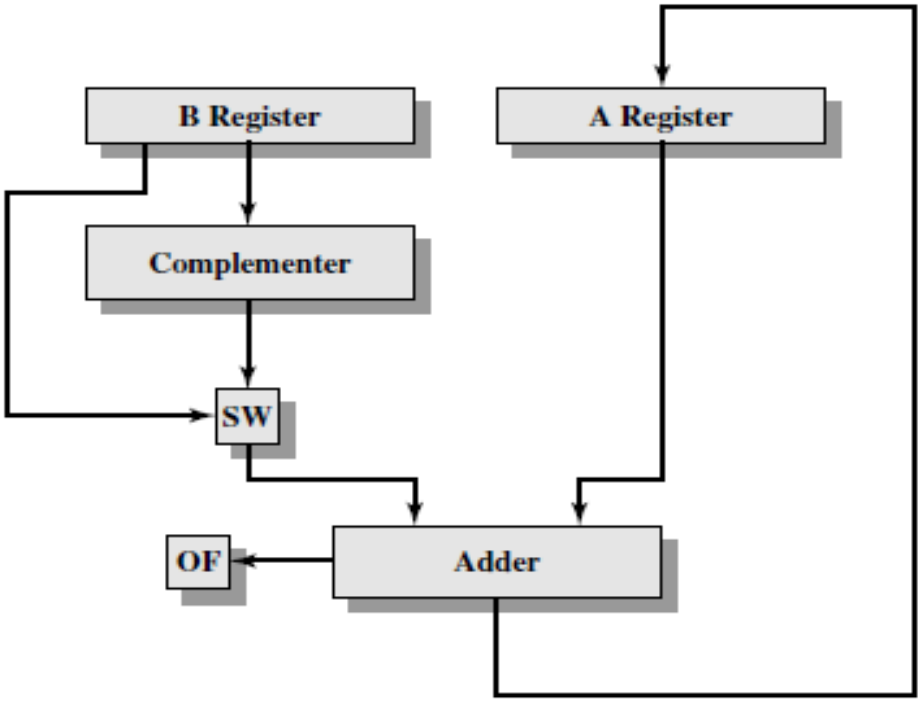
Subtraction

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)

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

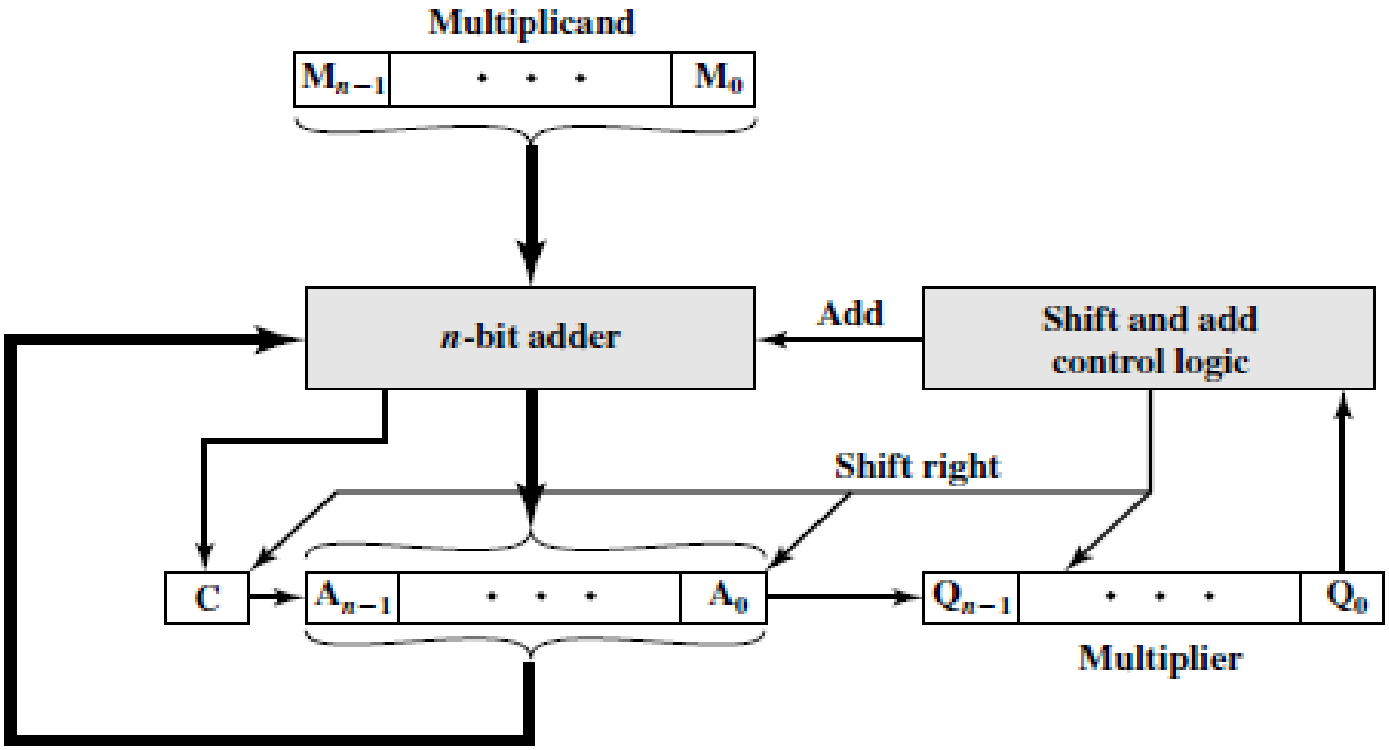
Addition & Subtraction



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

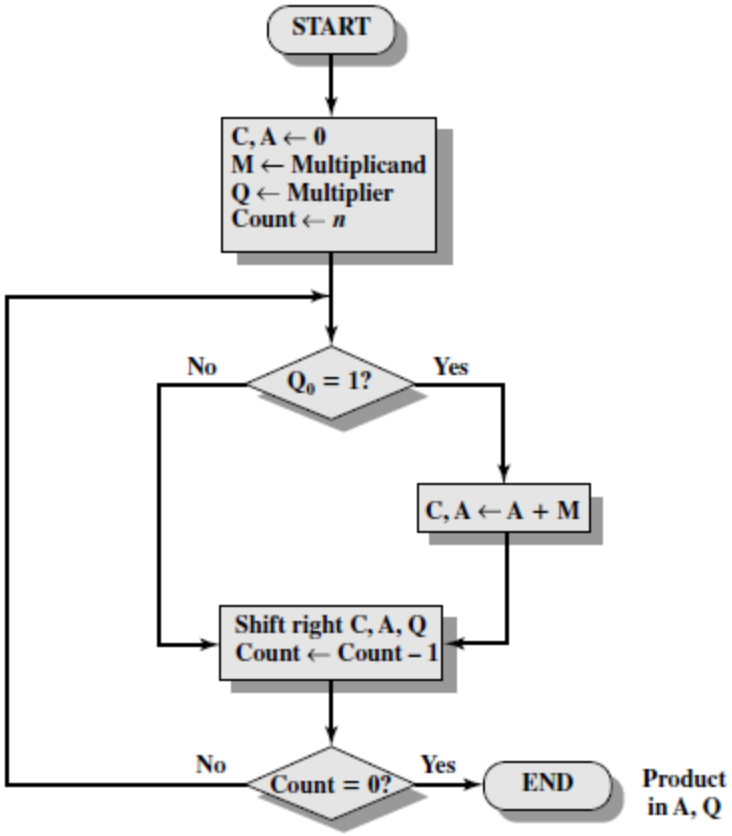
Figure 9.6 Block Diagram of Hardware for Addition and Subtraction

Multiplication



(a) Block diagram

Multiplication



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)

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

TWOS COMPLEMENT MULTIPLICATION

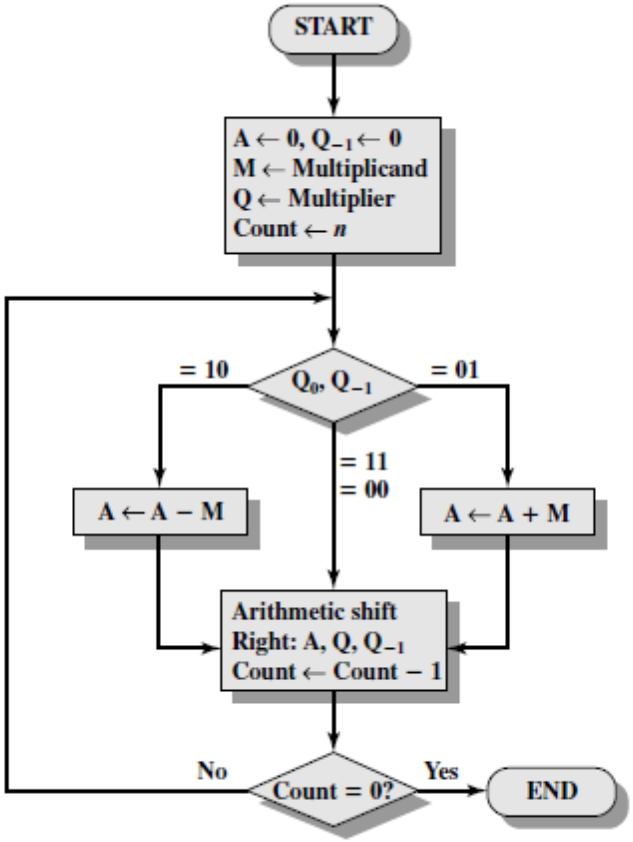


Figure 9.12 Booth's Algorithm for Twos Complement Multiplication

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)

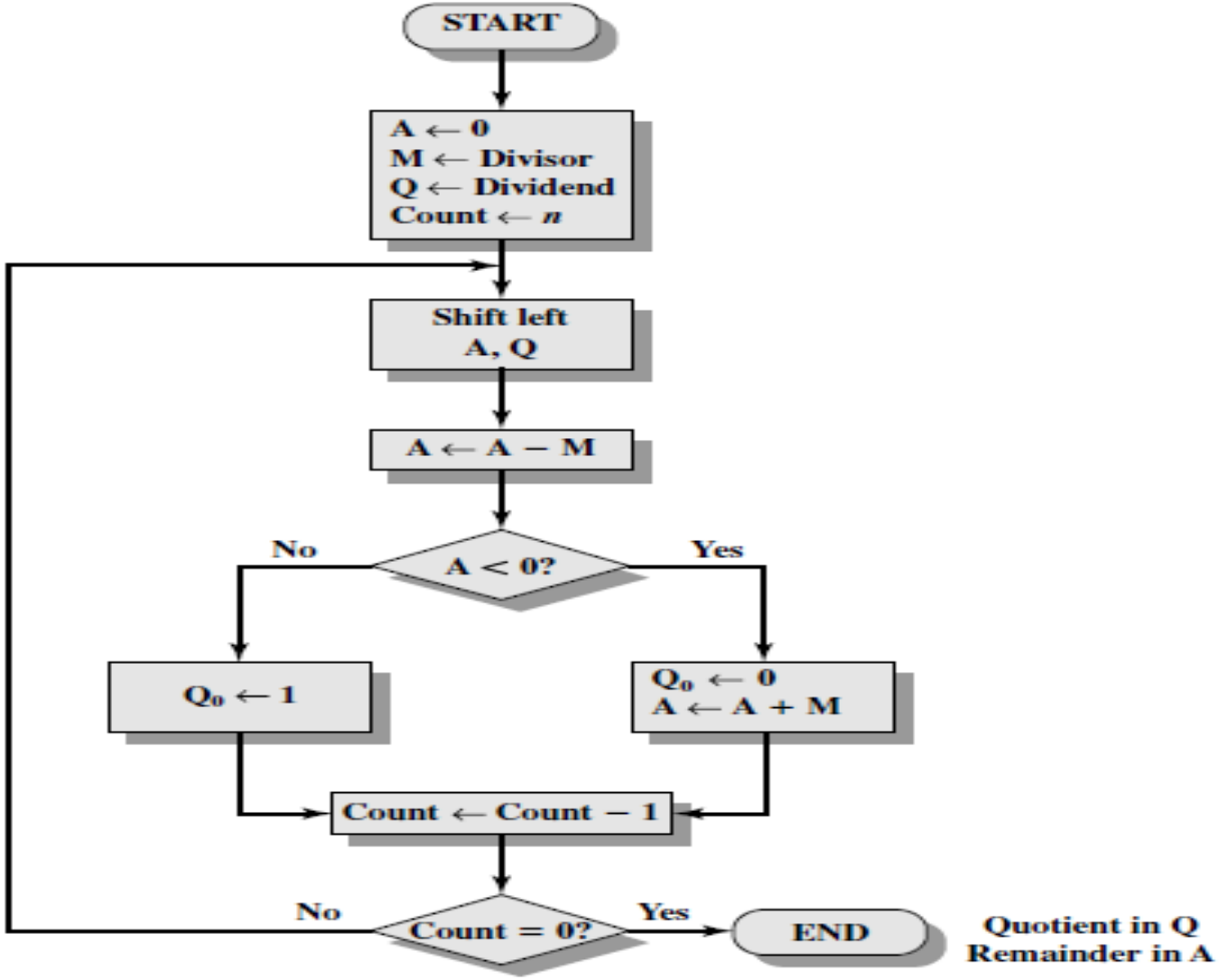


Figure 9.16 Flowchart for Unsigned Binary Division

Division

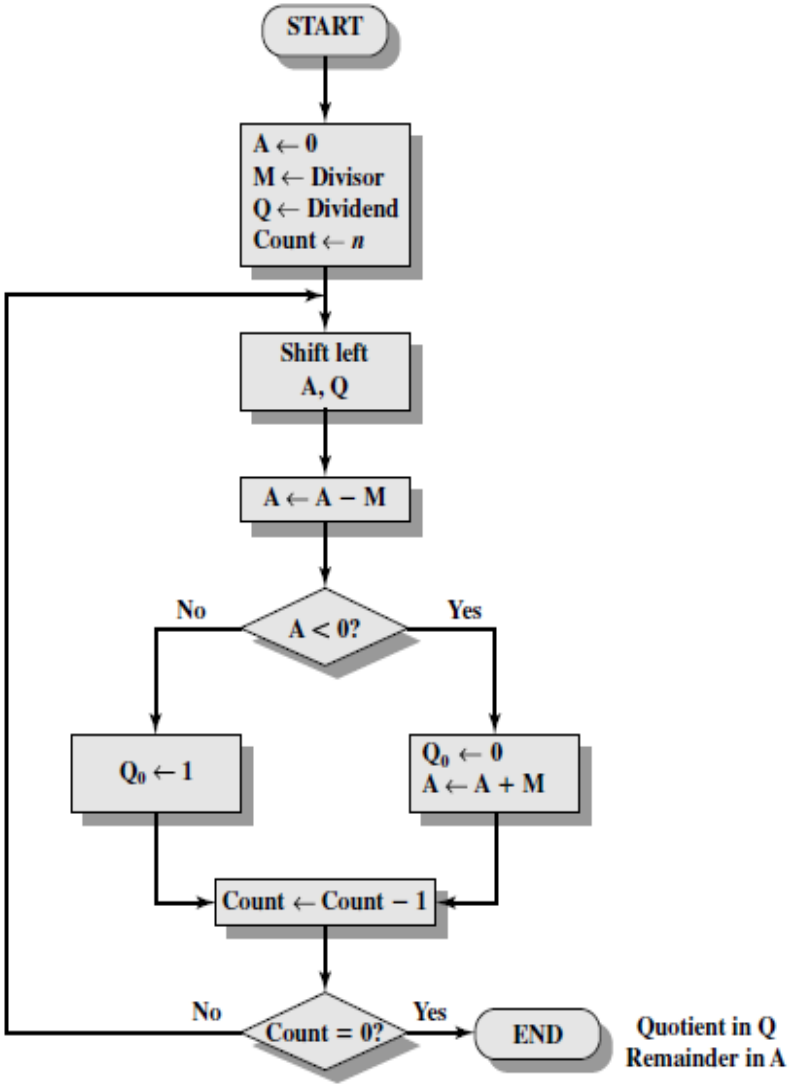
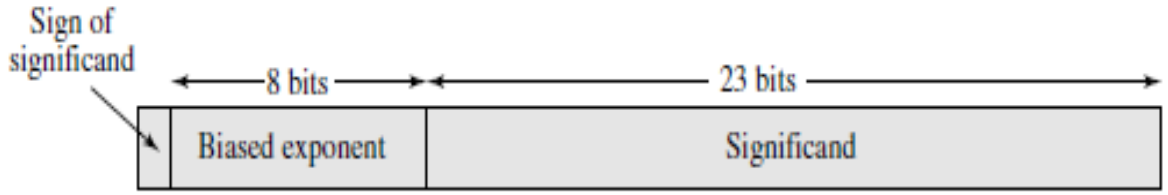


Figure 9.16 Flowchart for Unsigned Binary Division

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

Figure 9.17 Example of Restoring Twos Complement Division (7/3)

FLOATING-POINT REPRESENTATION



(a) Format

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$

$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$

$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$

$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

(b) Examples

Figure 9.18 Typical 32-Bit Floating-Point Format

FLOATING-POINT REPRESENTATION

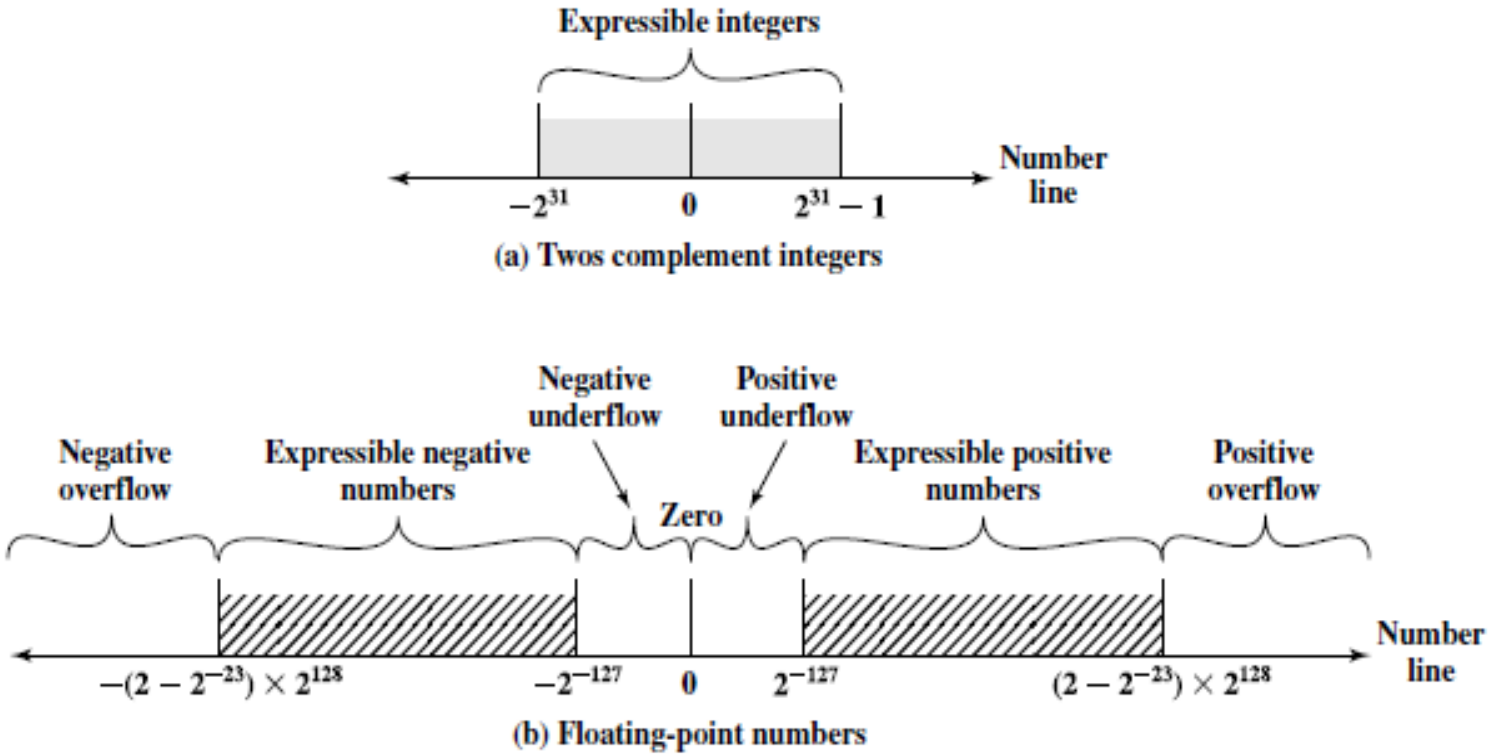


Figure 9.19 Expressible Numbers in Typical 32-Bit Formats

IEEE Standard for Binary Floating-Point Representation

- IEEE 754-2008 defines the following different types of floating-point formats:

■ Arithmetic format: All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.

■ Basic format: This format covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic. At least one of the basic formats is implemented in any conforming implementation.

■ Interchange format: A fully specified, fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage.

IEEE Standard for Binary Floating-Point Representation

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

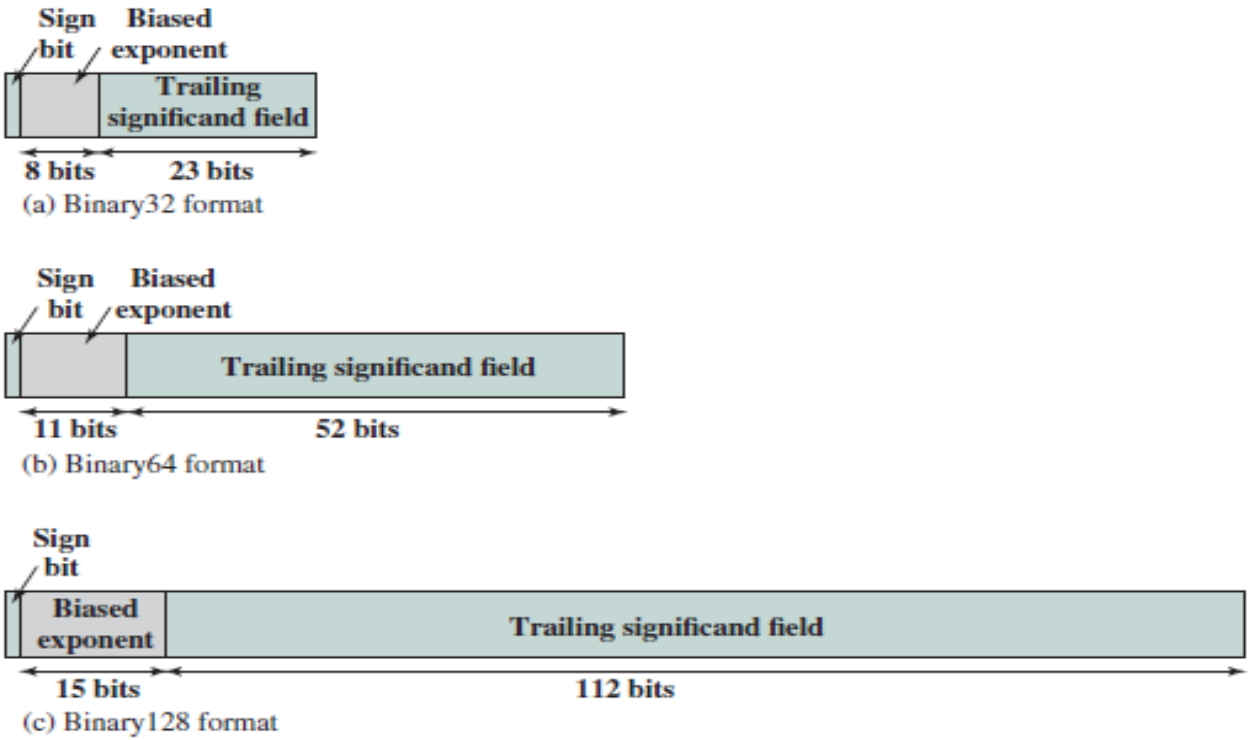


Figure 10.21 IEEE 754 Formats

IEEE 754 Format Parameters

Parameter	Format		
	Binary32	Binary64	Binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	−126	−1022	−16382
Approx normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16362}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Note: * Not including implied bit and not including sign bit.

IEEE 754

- IEEE 754-2008 defines an extendable precision format as a format with a precision and range that are defined under user control. Again, these formats may be used for intermediate calculations, but the standard places no constraint or format or length.

Classes of Numbers

- The following classes of numbers are represented:

■ For exponent values in the range of 1 through 254 for 32-bit format, 1 through 2046 for 64-bit format, and 1 through 16382, normal nonzero floating-point numbers are represented.

The exponent is biased, so that the range of exponents is -126 through +127 for 32-bit format, and so on. A normal number requires a 1 bit to the left of the binary point; this bit is implied, giving an effective 24-bit, 53-bit, or 113-bit significand. Because one of the bits is implied, the corresponding field in the binary format is referred to as the trailing significand field.

■ An exponent of zero together with a fraction of zero represents positive or negative zero, depending on the sign bit. As was mentioned, it is useful to have an exact value of 0 represented

Classes of Numbers

- An exponent of all ones together with a fraction of zero represents positive or negative infinity, depending on the sign bit. It is also useful to have a representation of infinity. This leaves it up to the user to decide whether to treat overflow as an error condition or to carry the value ∞ and proceed with whatever program is being executed.
- An exponent of zero together with a nonzero fraction represents a subnormal number. In this case, the bit to the left of the binary point is zero and the true exponent is -126 or -1022. The number is positive or negative depending on the sign bit.
- An exponent of all ones together with a nonzero fraction is given the value NaN, which means Not a Number, and is used to signal various exception conditions.

FLOATING-POINT ARITHMETIC

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 ∞
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value .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.

Addition and Subtraction

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

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

Floating-Point Addition & subtraction

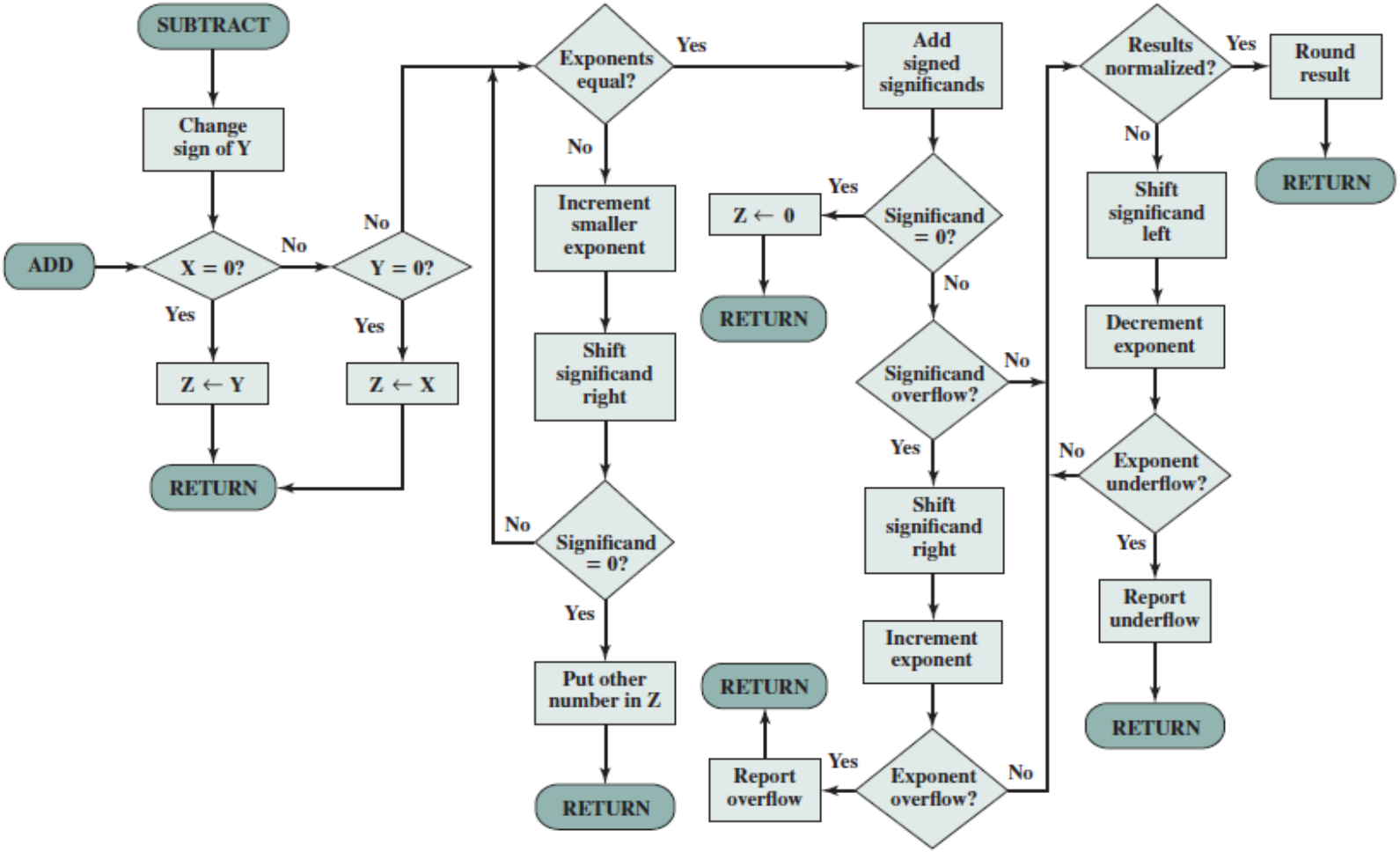


Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

Floating-Point Addition & subtraction

Phase 1.

Zero check: Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation. Next, if either operand is 0, the other is reported as the result.

Phase 2.

Significand alignment: The next phase is to manipulate the numbers so that the two exponents are equal.

Phase 3.

Addition: Next, the two significands are added together, taking into account their signs. Because the signs may differ, the result may be 0. There is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. An exponent overflow could occur as a result; this would be reported and the operation halted.

Phase 4.

Normalization: The final phase normalizes the result. Normalization consists of shifting significand digits left until the most significant digit (bit, or 4 bits for base-16 exponent) is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and then reported

Floating Point Multiplication

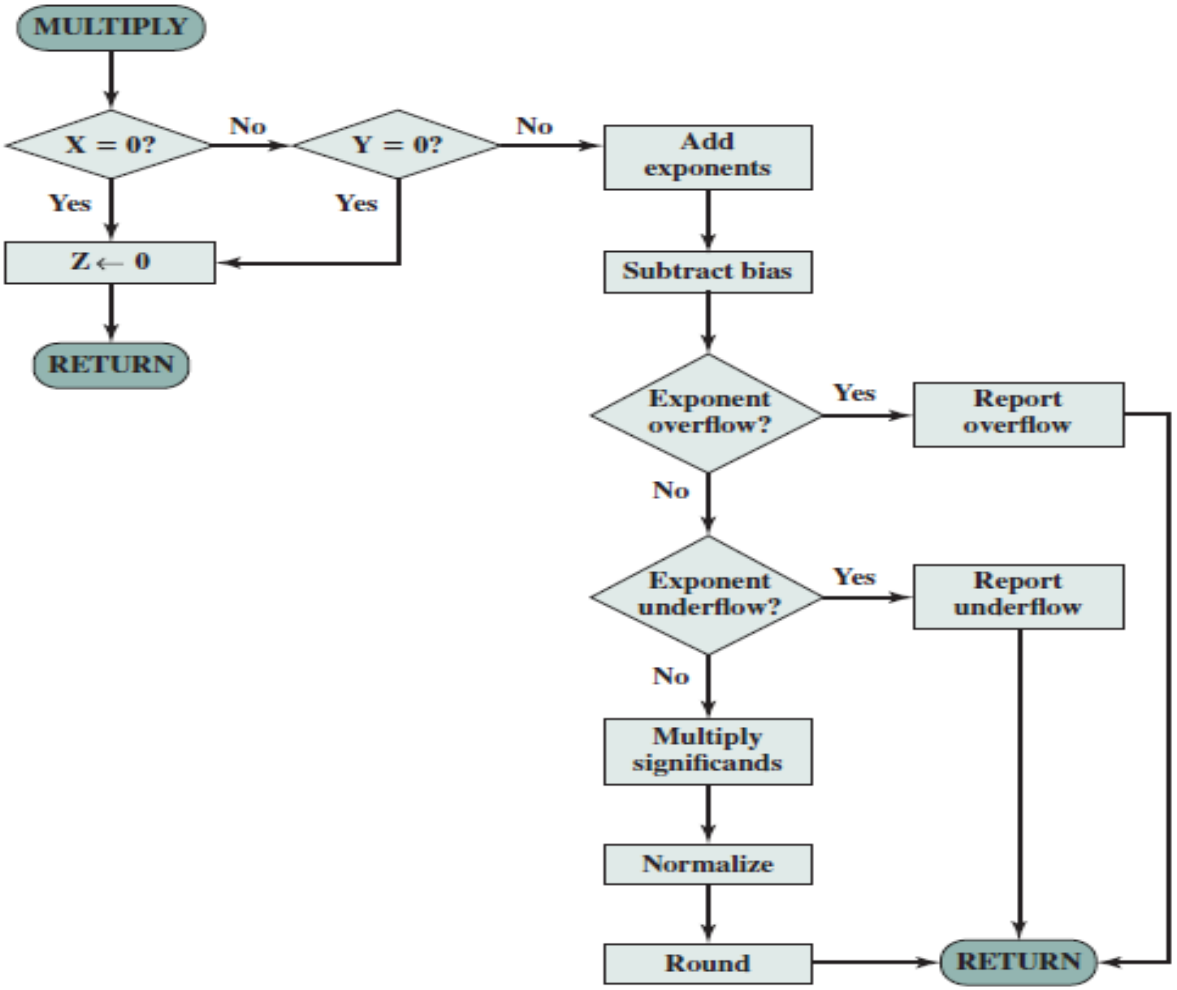


Figure 10.23 Floating-Point Multiplication ($Z \leftarrow X \pm Y$)

Floating Point Division

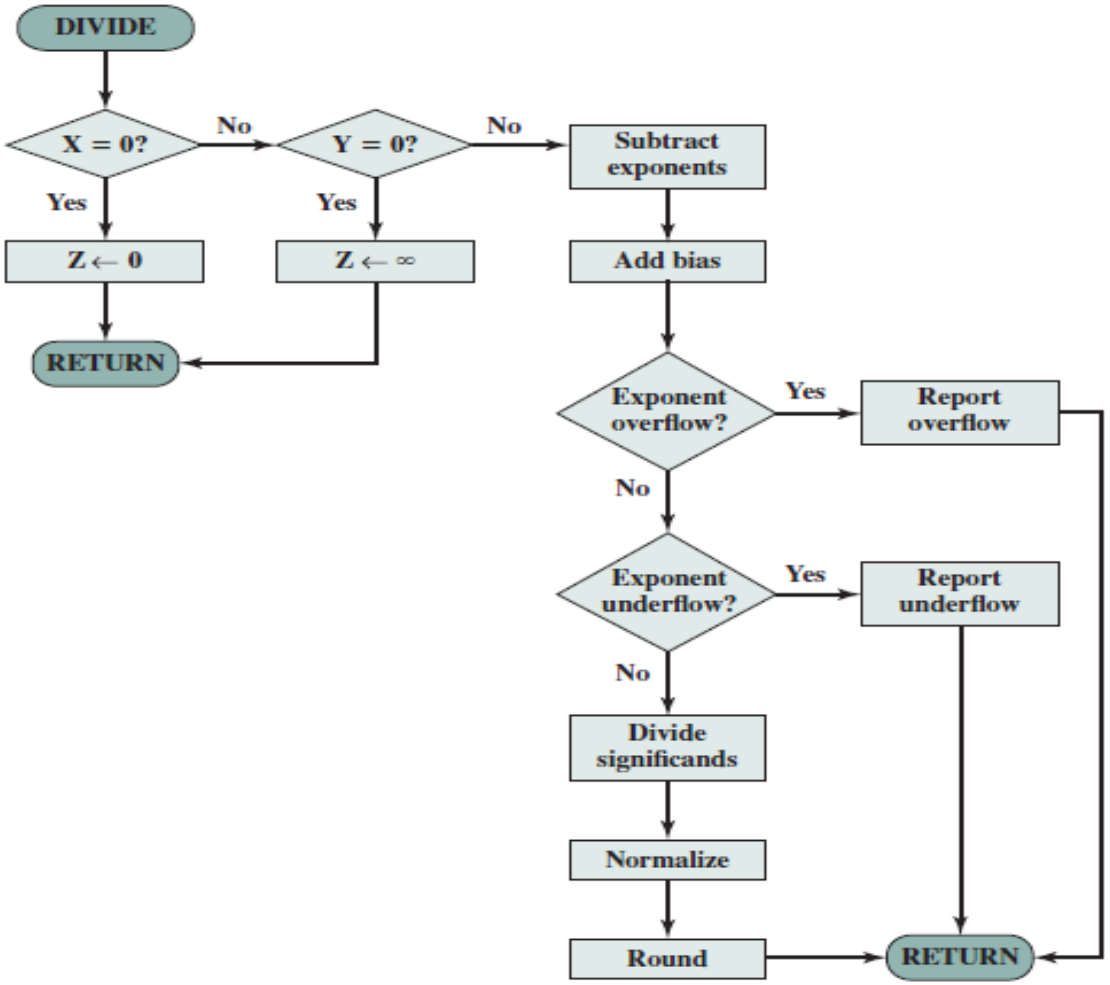


Figure 10.24 Floating-Point Division ($Z \leftarrow X/Y$)

The Use of Guard Bits.

$$
\begin{aligned}
x &= 1.000.....00 \times 2^{1} \\
-y &= 0.111.....11 \times 2^{1} \\
z &= 0.000.....01 \times 2^{1} \\
&= 1.000.....00 \times 2^{-22}
\end{aligned}
$$

(a) Binary example, without guard bits

$$
\begin{aligned}
x &= .100000 \times 16^{1} \\
-y &= .0FFFFF \times 16^{1} \\
z &= .000001 \times 16^{1} \\
&= .100000 \times 16^{-4}
\end{aligned}
$$

(c) Hexadecimal example, without guard bits

$$
\begin{aligned}
x &= 1.000.....00\ 0000 \times 2^{1} \\
-y &= 0.111.....11\ 1000 \times 2^{1} \\
z &= 0.000.....00\ 1000 \times 2^{1} \\
&= 1.000.....00\ 0000 \times 2^{-23}
\end{aligned}
$$

(b) Binary example, with guard bits

$$
\begin{aligned}
x &= .100000\ 00 \times 16^{1} \\
-y &= .0FFFFF\ F0 \times 16^{1} \\
z &= .000000\ 10 \times 16^{1} \\
&= .100000\ 00 \times 16^{-5}
\end{aligned}
$$

(d) Hexadecimal example, with guard bits

Figure 10.25 The Use of Guard Bits

Precision Considerations

- GUARD BITS:

The prior to a floating-point operation, the exponent and significand of each operand are loaded into ALU registers. In the case of the significand, the length of the register is almost always greater than the length of the significand plus an implied bit. The register contains additional bits, called guard bits, which are used to pad out the right end of the significand with 0s.

- ROUNDING

Another detail that affects the precision of the result is the rounding policy. The result of any operation on the significands is generally stored in a longer register. When the result is put back into the floating-point format, the extra bits must be eliminated in such a way as to produce a result that is close to the exact result. This process is called rounding.

Precision Considerations

- A number of techniques have been explored for performing rounding. In fact, the IEEE standard lists four alternative approaches:

■ Round to nearest: The result is rounded to the nearest representable number.

■ Round toward $+\infty$: The result is rounded up toward plus infinity.

■ Round toward $-\infty$: The result is rounded down toward negative infinity.

■ Round toward 0: The result is rounded toward zero.