Akdeniz University
Computer Engineering Department

CSE206 Computer Organization

Week09: Number Systems and Computer Arithmetic

Assoc.Prof.Dr. Taner Danışman
tdanisman@akdeniz.edu.tr

Course program (Textbook: Stalling 10th Edt.)

Week 1	10-Feb-25	Introduction	Ch1
Week 2	17-Feb-25	Computer Evolution	Ch2
Week 3	24-Feb-25	Computer Systems	Ch3
Week 4	3-Mar-25	Cache Memory, Direct Cache Mapping	Ch4
Week 5	10-Mar-25	Associative and Set Associative Mapping	Ch4
Week 6	17-Mar-25	Internal Memory, External Memory, I/O	Ch5-Ch6-Ch7
Week 7	24-Mar-25	Number Systems, Computer Arithmetic	Ch9-Ch10
Week 8	31-Mar-25	Midterm (Expected date, may change)	Ch1-…-Ch10
Week 9	7-Apr-25	Digital Logic	Ch11
Week 10	14-Apr-25	Instruction Sets	Ch12
Week 11	21-Apr-25	Addressing Modes	Ch13
Week 12	28-Apr-25	Processor Structure and Function	Ch14
Week 13	5-May-25	RISC, Inctruction Level Parallelism	Ch15-Ch16
Week 14	12-May-25	Assembly Language (TextBook : Assembly Language for x86 Processors)	Kip Irvine
Week 15	19-May-25	Assembly Language (TextBook : Assembly Language for x86 Processors)	Kip Irvine

The Decimal System

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers
- For example the number 83 means eight tens plus three:

$$83 = (8 * 10) + 3$$

- The number 4728 means four thousands, seven hundreds, two tens, plus eight:

$$4728 = (4 * 1000) + (7 * 100) + (2 * 10) + 8$$

- The decimal system is said to have a ***base***, or ***radix***, of 10. This means that each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position:

$$83 = (8 * 10^1) + (3 * 10^0)$$

$$4728 = (4 * 10^3) + (7 * 10^2) + (2 * 10^1) + (8 * 10^0)$$

Decimal Fractions

- The same principle holds for decimal fractions, but negative powers of 10 are used. Thus, the decimal fraction 0.256 stands for 2 tenths plus 5 hundredths plus 6 thousandths:

$$0.256 = (2 * 10^{-1}) + (5 * 10^{-2}) + (6 * 10^{-3})$$

- A number with both an integer and fractional part has digits raised to both positive and negative powers of 10:

$$442.256 = (4 * 10^{2}) + (4 + 10^{1}) + (2 * 10^{0}) + (2 * 10^{-1}) + (5 * 10^{-2}) + (6 * 10^{-3})$$

- ***Most significant digit***
  - The leftmost digit (carries the highest value)
- ***Least significant digit***
  - The rightmost digit

Positional Interpretation of a Decimal Number

4	7	2	2	5	6
100s 10^2 position 2	10s 10^1 position 1	1s 10^9 position 0	tenths 10^{-1} position −1	hundredths 10^{-2} position −2	thousandths 10^{-3} position −3

Table 9.1 Positional Interpretation of a Decimal Number

Positional Number Systems

- Each number is represented by a string of digits in which each digit position *i* has an associated weight r^i, where *r* is the *radix, or base,* of the number system.
- The general form of a number in such a system with radix *r* is

$$(\ldots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3} \ldots)_r$$

where the value of any digit a_i *is an integer in the range* $0 \le a_i < r$. The dot between a_0 *and* a_{-1} is called the ***radix point.***

Positional Interpretation of a Number in Base 7

Position	4	3	2	2	0	–1
Value in exponential form	7^4	7^3	7^2	7^1	7^0	7^{-1}
Decimal value	2401	343	49	7	1	1/7

Table 9.2 Positional Interpretation of a Number in Base 7

The Binary System

- Only two digits, 1 and 0
- Represented to the base 2
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_2 = 0_{10}$$
$$1_2 = 1_{10}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

$$10_2 = (1 * 2^1) + (0 * 2^0) = 2_{10}$$
$$11_2 = (1 * 2^1) + (1 * 2^0) = 3_{10}$$
$$100_2 = (1 * 2^2) + (0 * 2^1) + (0 * 2^0) = 4_{10}$$

and so on. Again, fractional values are represented with negative powers of the radix:

$$1001.101 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625_{10}$$

Converting Between Binary and Decimal

- Binary notation to decimal notation:
  - Multiply each binary digit by the appropriate power of 2 and add the results
- Decimal notation to binary notation:
  - Integer and fractional parts are handled separately

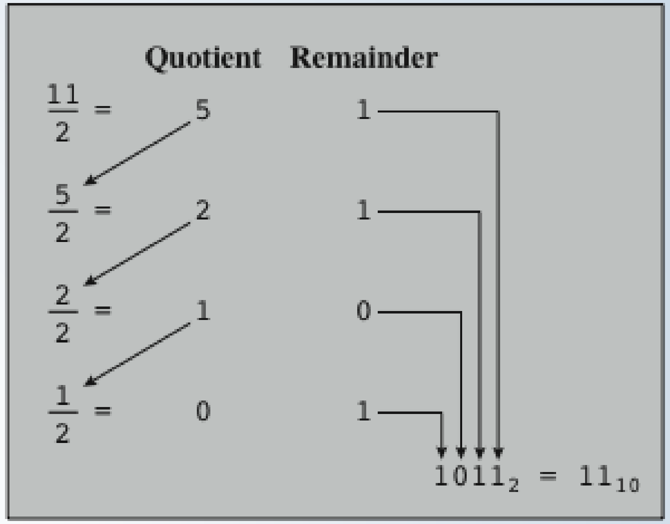
Integers

- For the integer part, recall that in binary notation, an integer represented by
  - $b_{m-1}b_{m-2} \ldots b_2b_1b_0 \quad b_i = 0 \text{ or } 1$
- has the value
  - $(b_{m-1} * 2^{m-1}) + (b_{m-2} * 2^{m-2}) + \ldots + (b_1 * 2^1) + b_0$

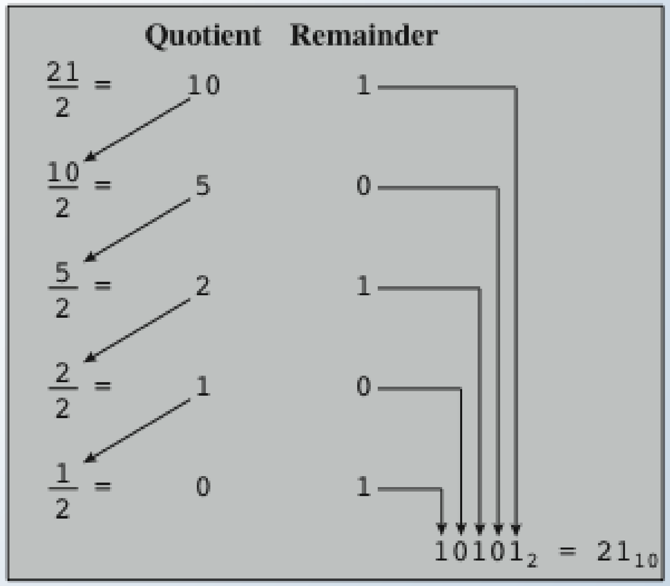
Continued . . .

- Suppose it is required to convert a decimal integer N into binary form. If we divide N by 2, in the decimal system, and obtain a quotient N_1 and a remainder R_0, we may write
  - $N = 2 * N_1 + R_0 \qquad R_0 = 0 \text{ or } 1$
- Next, we divide the quotient N_1 by 2. Assume that the new quotient is N_2 and the new remainder R_1. Then
  - $N_1 = 2 * N_2 + R_1 \qquad R_1 = 0 \text{ or } 1$
- so that
  - $N = 2(2N_2 + R_1) + R_0 = (N_2 * 2^2) + (R_1 * 2^1) + R_0$
- If next
  - $N_2 = 2N_3 + R_2$
- we have
  - $N = (N_3 * 2^3) + (R_2 * 2^2) + (R_1 * 2^1) + R_0$

Figure 9.1 Converting from Decimal Notation to Binary Notation for Integers

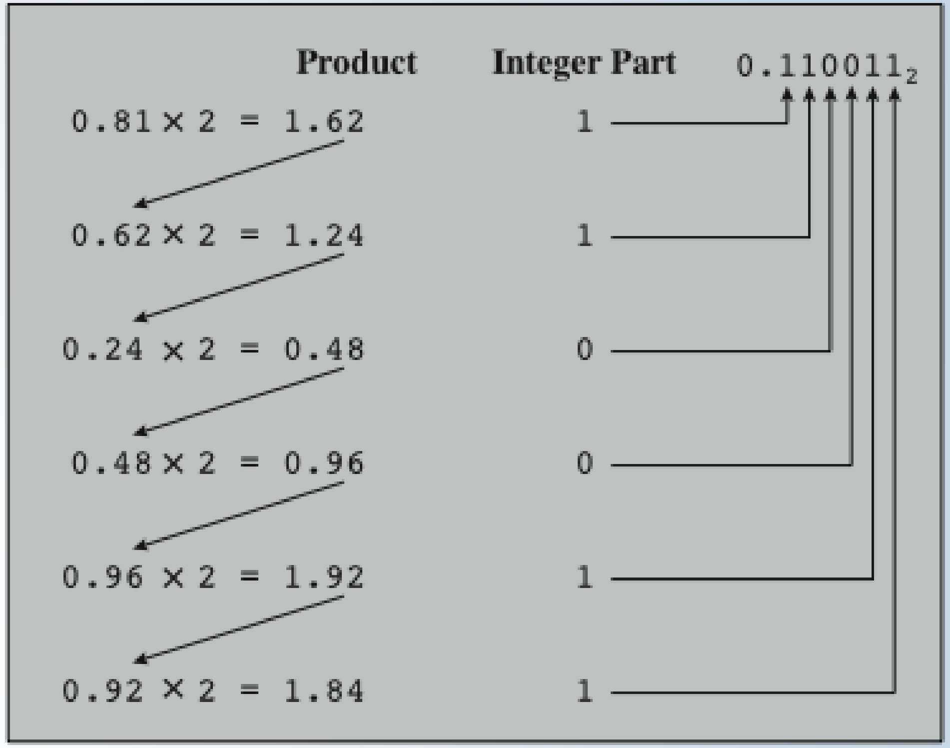


(a) 11_{10}

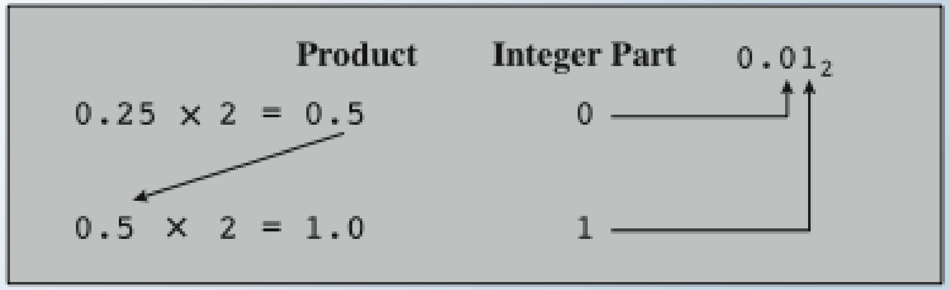


(b) 21_{10}

Figure 9.2 Converting from Decimal Notation to Binary Notation for Fractions



(a) $0.81_{10} = 0.110011_2$ (approximately)



(b) $0.25_{10} = 0.01_2$ (exactly)

Hexadecimal Notation

- Binary digits are grouped into sets of four bits, called a ***nibble***
- Each possible combination of four binary digits is given a symbol, as follows:

0000 = 0	0100 = 4	1000 = 8	1100 = C
0001 = 1	0101 = 5	1001 = 9	1101 = D
0010 = 2	0110 = 6	1010 = A	1110 = E
0011 = 3	0111 = 7	1011 = B	1111 = F

- Because 16 symbols are used, the notation is called *hexadecimal* and the 16 symbols are the *hexadecimal digits*
- Thus

$$2C_{16} = (2_{16} * 16^1) + (C_{16} * 16^0)$$
$$= (2_{10} * 16^1) + (12_{10} * 16^0) = 44$$

Table 9.3 Decimal, Binary, and Hexadecimal

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
18	0001 0010	12
31	0001 1111	1F
100	0110 0100	64
255	1111 1111	FF
256	0001 0000 0000	100

Hexadecimal Notation

Not only used for representing integers but also as a concise notation for representing any sequence of binary digits

Reasons for using hexadecimal notation are:

- It is more compact than binary notation
- In most computers, binary data occupy some multiple of 4 bits, and hence some multiple of a single hexadecimal digit
- It is extremely easy to convert between binary and hexadecimal notation

Summary

- Number Systems
  - The decimal system
  - Positional number systems
  - The binary system
- Converting between binary and decimal
  - Integers
  - Fractions
- Hexadecimal notation

Computer Arithmetic

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
- Based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations

ALU Inputs and Outputs

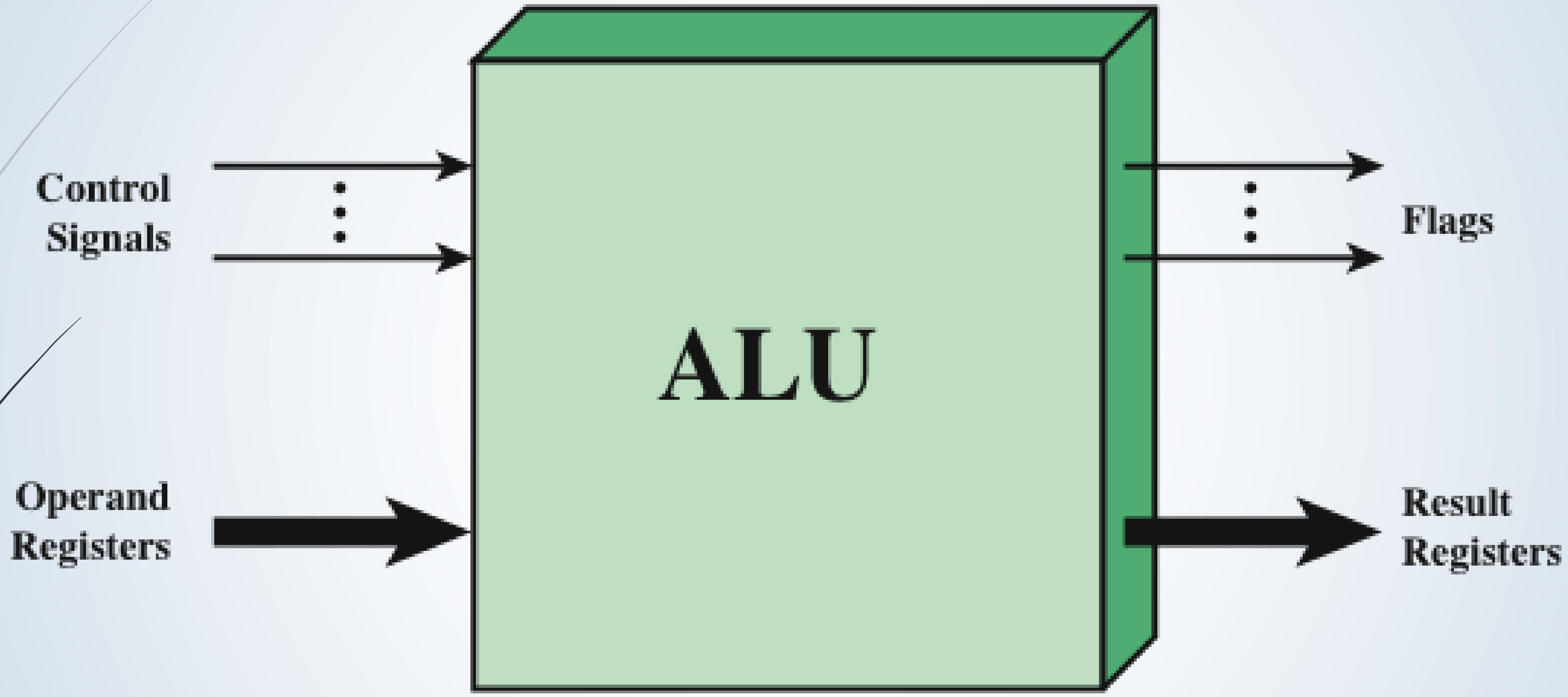


Figure 10.1 ALU Inputs and Outputs

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

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
+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

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*

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

Addition Examples

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

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 Example

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

Geometric Depiction of Twos Complement Integers

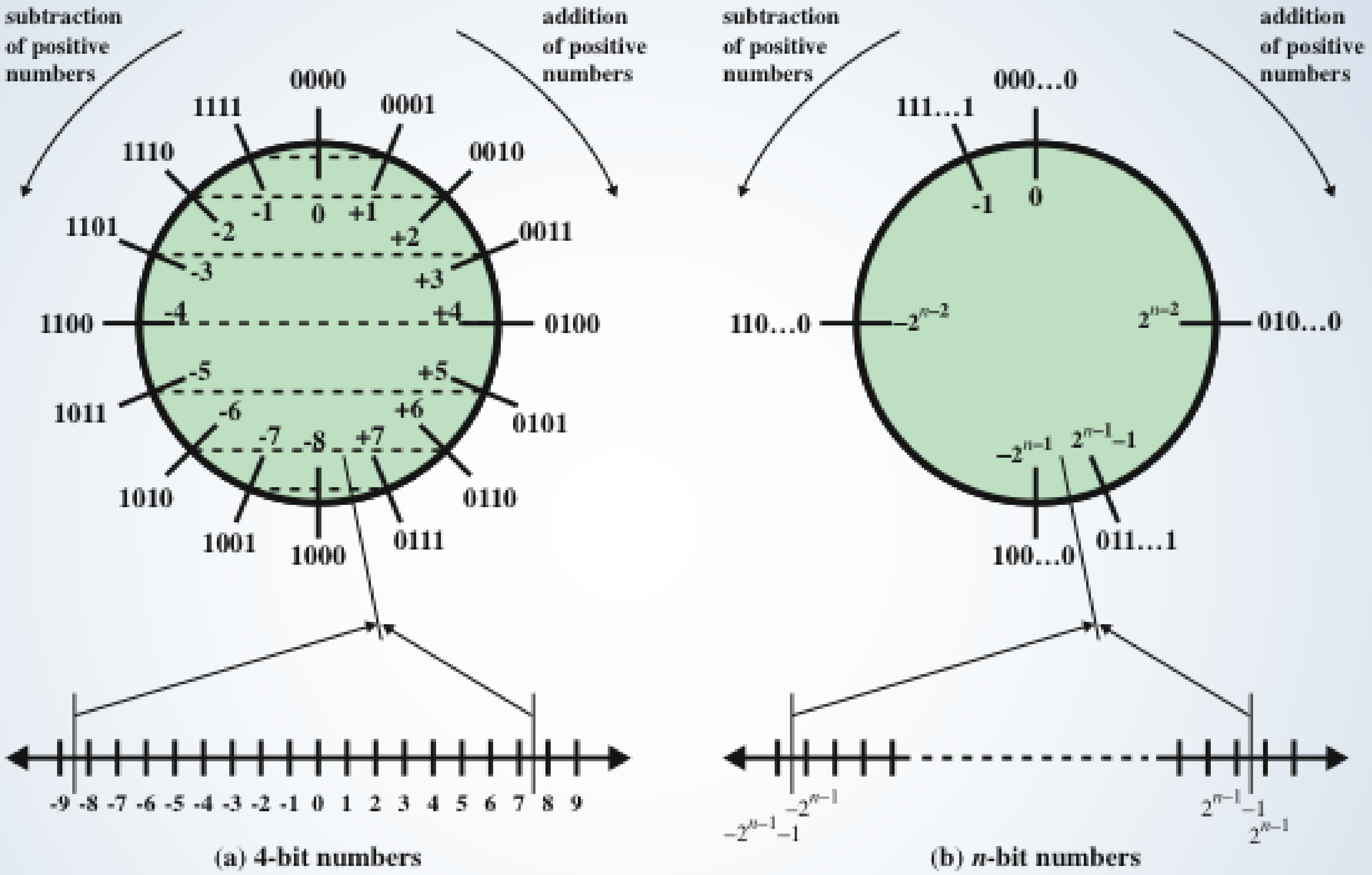
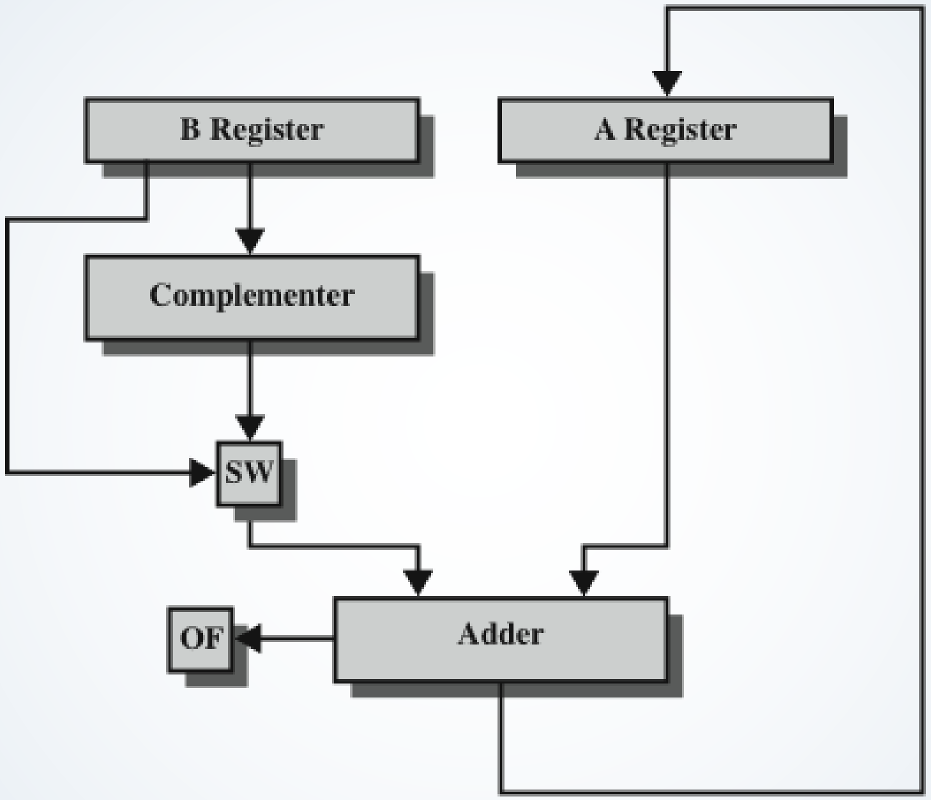


Figure 10.5 Geometric Depiction of Twos Complement Integers

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

Floating-Point Representation

- 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

Floating-Point Representation

- For decimal numbers, we use scientific notation.
- Thus, 976,000,000,000,000 can be represented as
  - $9.76 * 10^{14}$ and
- 0.0000000000000976 can be represented as
  - $9.76 * 10^{-14}$
- We dynamically slide the decimal point to a convenient location and use the exponent of 10 to keep track of that dec
  - **Sign**: plus or minus
  - **Significand** S
  - **Exponent** E

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

The **base** B is implicit and need not be stored because it is the same for all numbers.

Floating-Point

- 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:

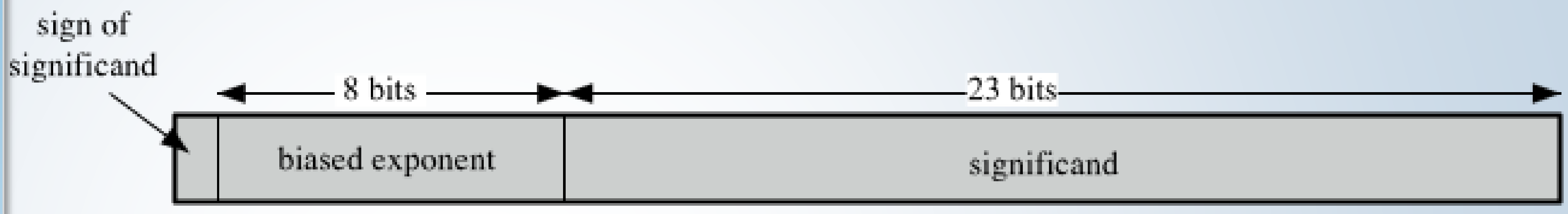
$$0.110 * 2^5$$
$$110 * 2^2$$
$$0.0110 * 2^6$$

- *Normal number*
  - The most significant digit of the significand is **nonzero**
  - A normal number is one in which the most significant digit of the significand is nonzero.
  - For base 2 representation, a normal number is therefore one in which the most significant bit of the significand is one.

Typical 32-Bit Floating-Point Format

The leftmost bit stores the **sign** of the number (0 = positive, 1 = negative).
The **exponent** value is stored in the next 8 bits. The representation used is known as a **biased representation.**

Typically, the bias equals $(2^{k-1} - 1)$, where k is the number of bits in the binary exponent



(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

the true exponent values are in the range -127 to +128

Figure 10.18 Typical 32-Bit Floating-Point Format

To see the need for aligning exponents, consider the following decimal addition:

$$(123 \times 10^0) + (456 \times 10^{-2})$$

Clearly, we cannot just add the significands. The digits must first be set into equivalent positions, that is, the 4 of the second number must be aligned with the 3 of the first. Under these conditions, the two exponents will be equal, which is the mathematical condition under which two numbers in this form can be added. Thus,

$$(123 \times 10^0) + (456 \times 10^{-2}) = (123 \times 10^0) + (4.56 \times 10^0) = 127.56 \times 10^0$$

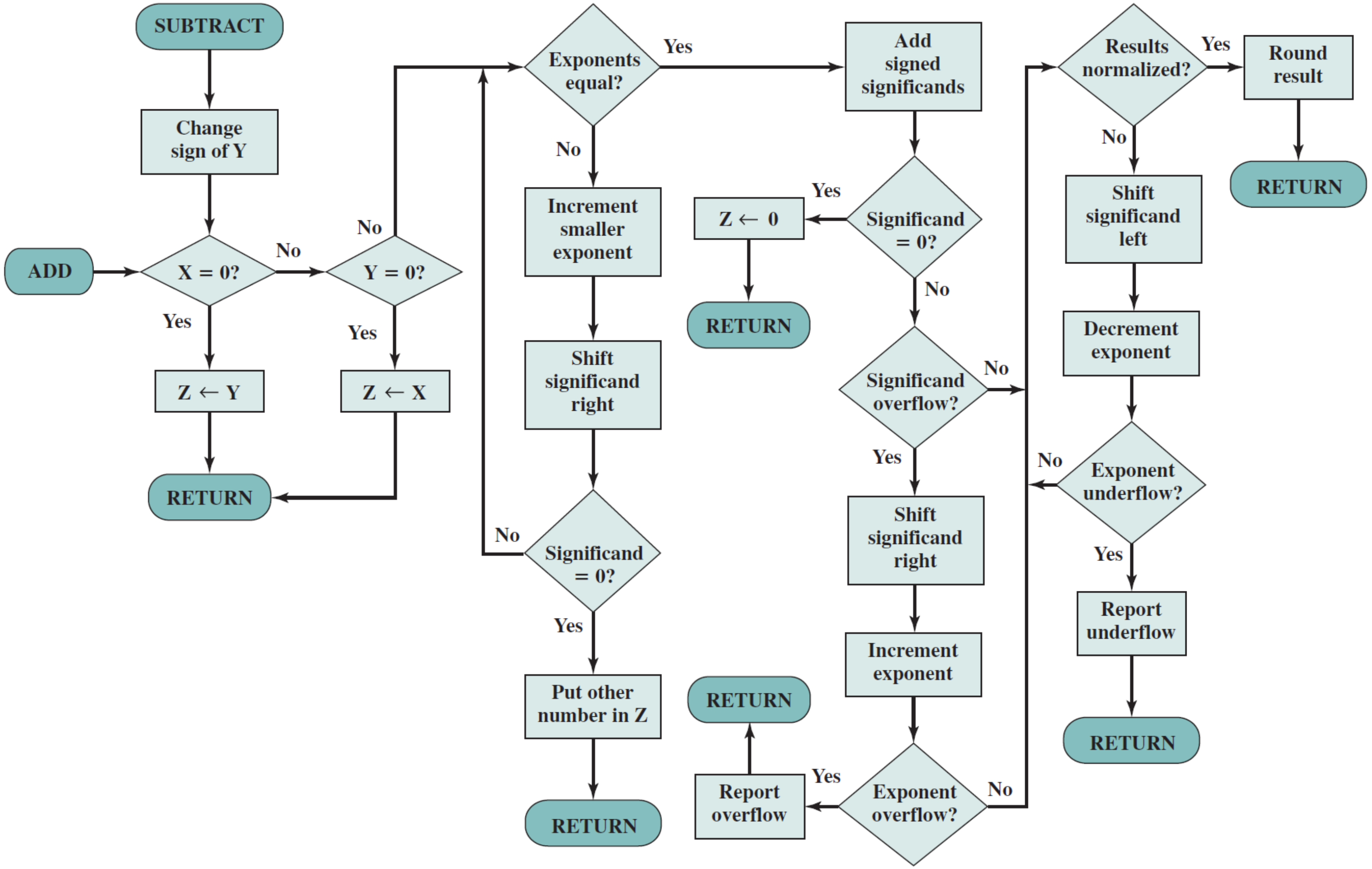
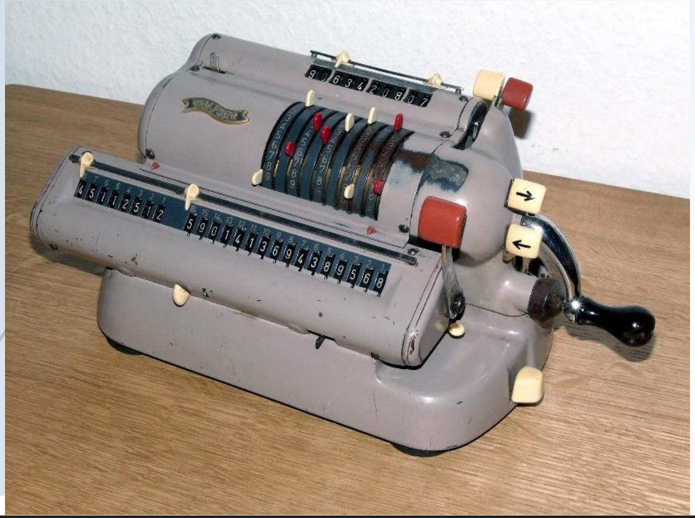


Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

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

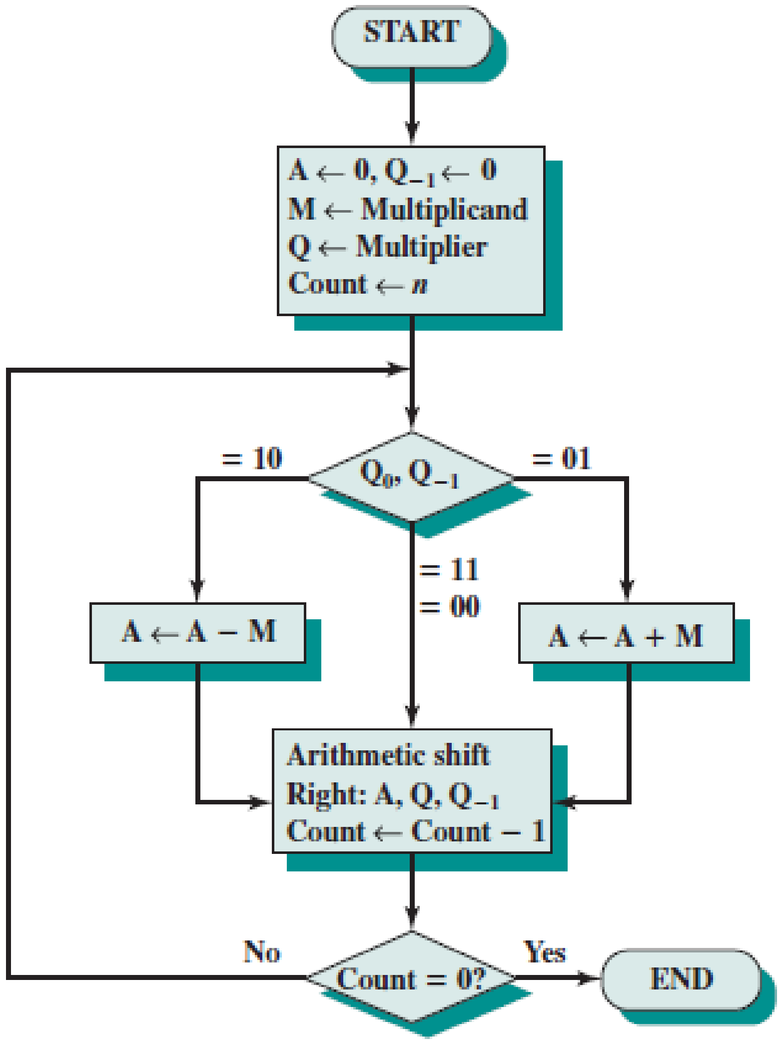


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

A	Q	Q_{-1}	M
0000	0101	0	1100

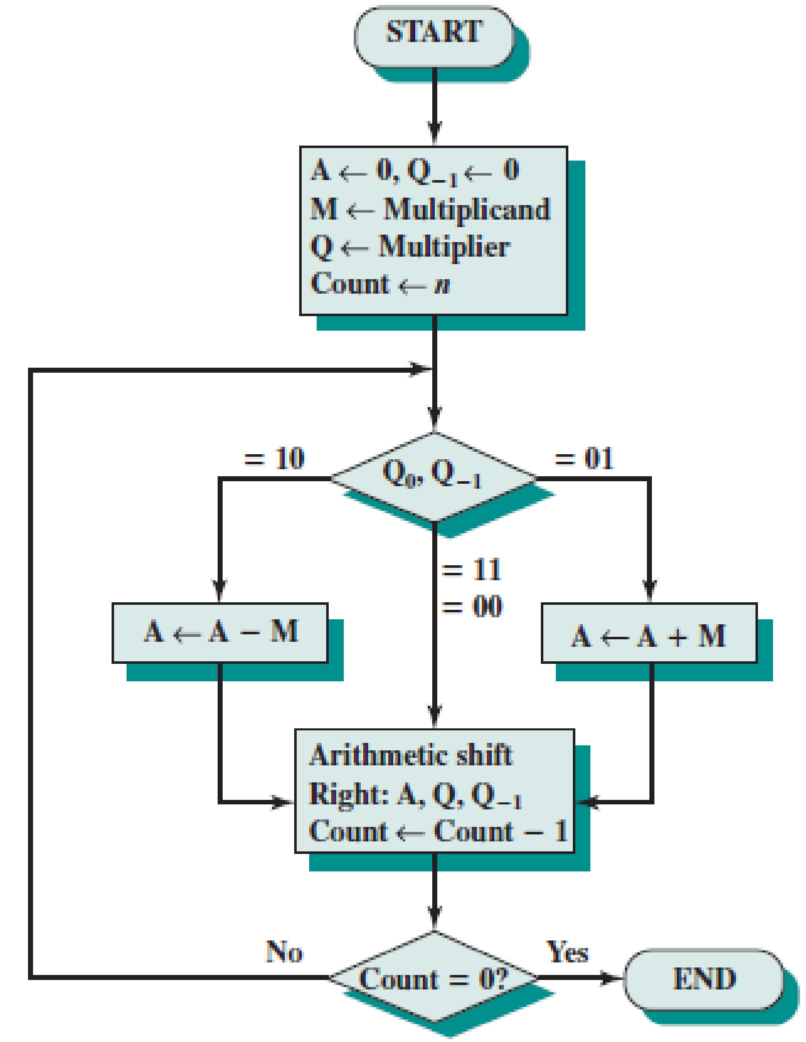


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication