+

MODULE 2

William Stallings
Computer Organization and Architecture
10th Edition

Chapter 10

Computer Arithmetic

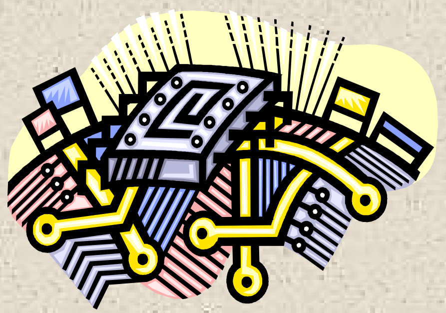
+

Objectives

- Understand the fundamentals of **numerical data** representation and manipulation in digital computers.
- Master the skill of converting between various **radix systems**.
- Understand how **errors** can occur in computations because of overflow and truncation.
- Understand the fundamental concepts of **floating-point** representation.

+ 10.1 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



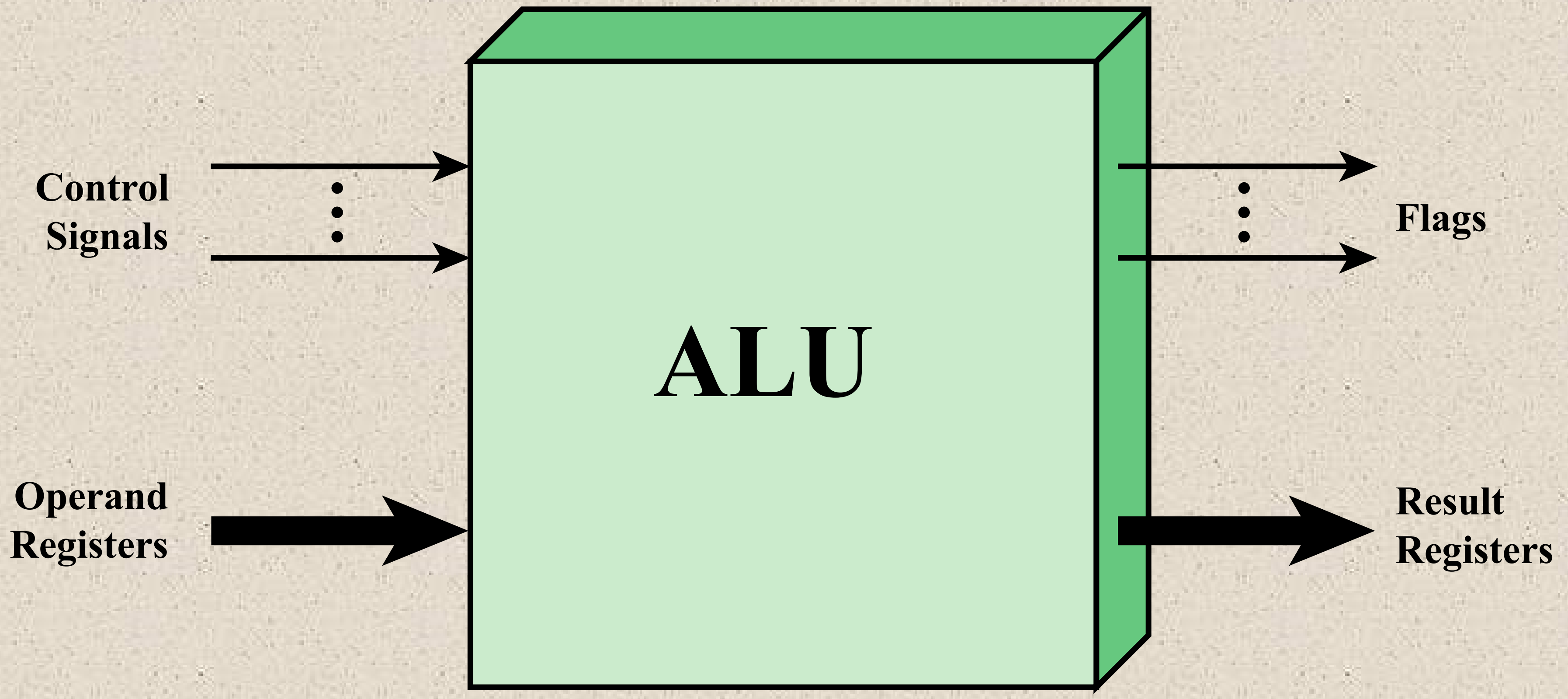
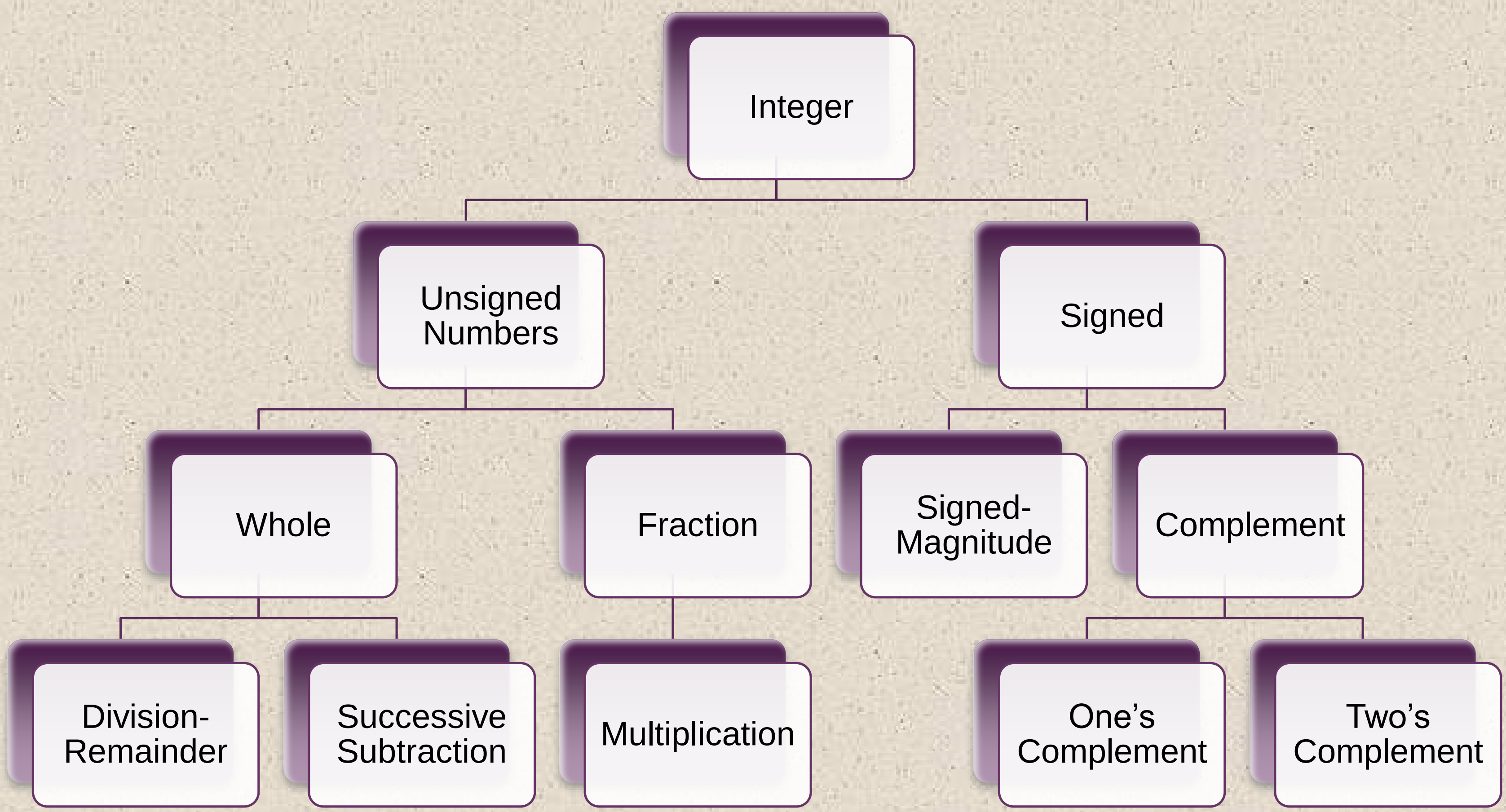


Figure 10.1 ALU Inputs and Outputs

10.2
Integer Representation

+ Sign-Magnitude Representation
+ Twos Complement Representation
+ Range Extension
+ Fixed-Point Representation

Numbers: overview



Unsigned Numbers

- For positive number only.
- Range number = 0 to 2^n-1
- *There is no negative values representation*.

If 32 bits long.
Range of numbers that can be represented
= 0 to (2^{32} – 1) = 0 to 4,294,967,295

+

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

Signed Numbers

- Integers as binary values can be positive or negative.
- Problem:
  - *How to represent and encode the actual sign of a number?*
- Solution:
  - Need code to **represent the signs**.
- Three signed representations covered:
  - Signed Magnitude
  - Ones Complement
  - Twos Complement

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

Table 10.1

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*.

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	—

–128	64	32	16	8	4	2	1

(a) An eight-position two's 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 10.2 Use of a Value Box for Conversion Between Twos Complement Binary and Decimal

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

00000010 **00000000 00000010**

- 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*

11111110 **11111111 11111110**

Sign bit

Sign bit extended

Sign bit

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

10.3
Integer Arithmetic

+ Negation
+ Addition and Subtraction
+ Multiplication
+ Division

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

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

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

Negation Special Case 1 (0)

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

Overflow is ignored, so:

- 0 = 0

Negation Special Case 2 (-128)

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

So:

-(-128) = -128 X

Monitor MSB (sign bit)

It should change during negation

```
 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:

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.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Overflow in Humankind History

The $7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed.

The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.

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

Rule

```
  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
```

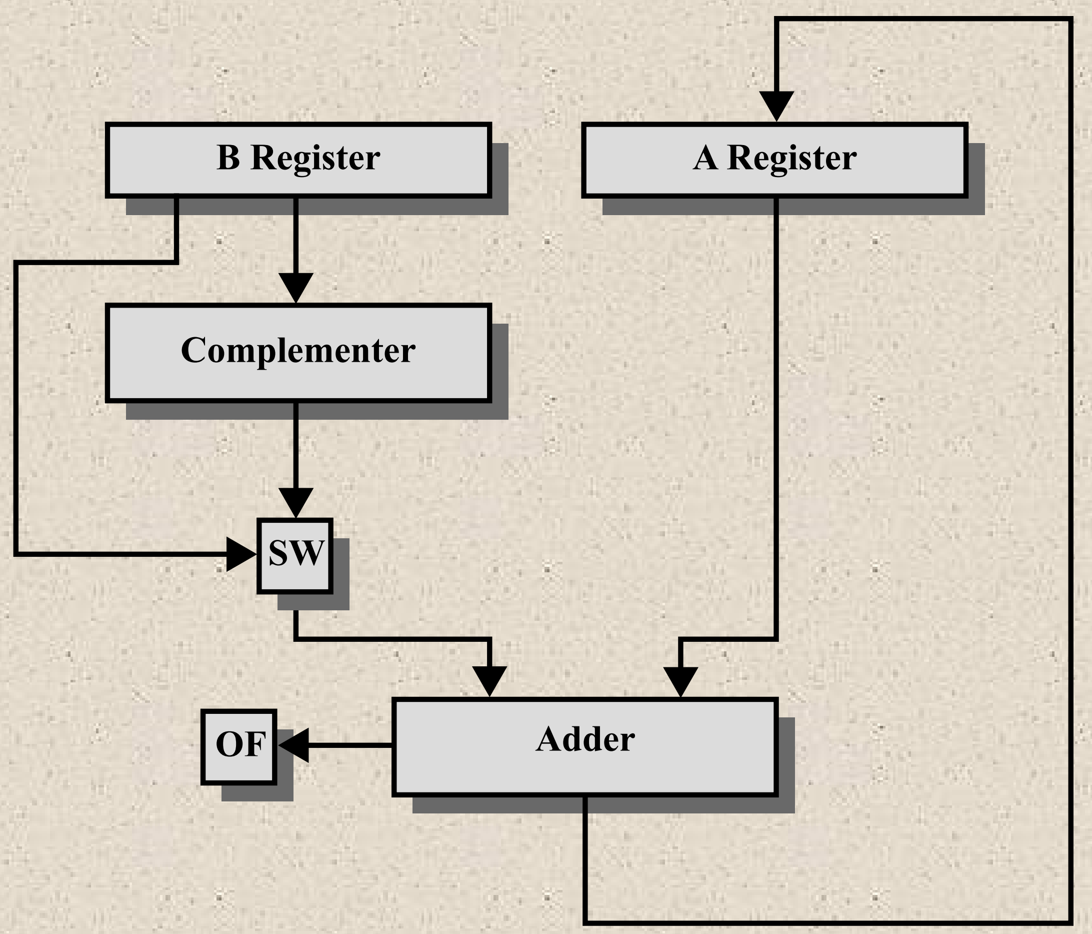
```
  0111 = 7
 +0111 = 7
  1110 = Overflow

(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)



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

Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

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

Figure 10.7 Multiplication of Unsigned Binary Integers

- Paper and pencil example (unsigned):

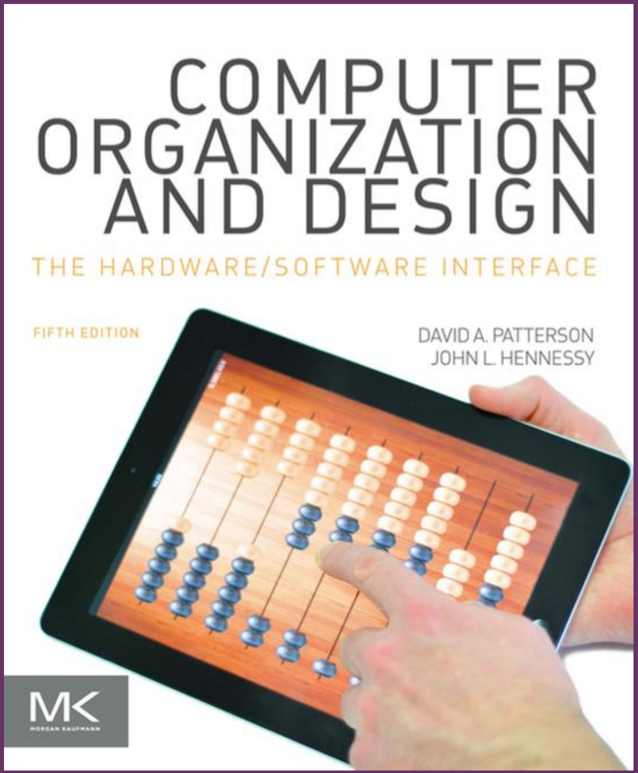
- 3 versions of multiply hardware & algorithm:

•successive refinement

Hardware implementation: multiplication & division

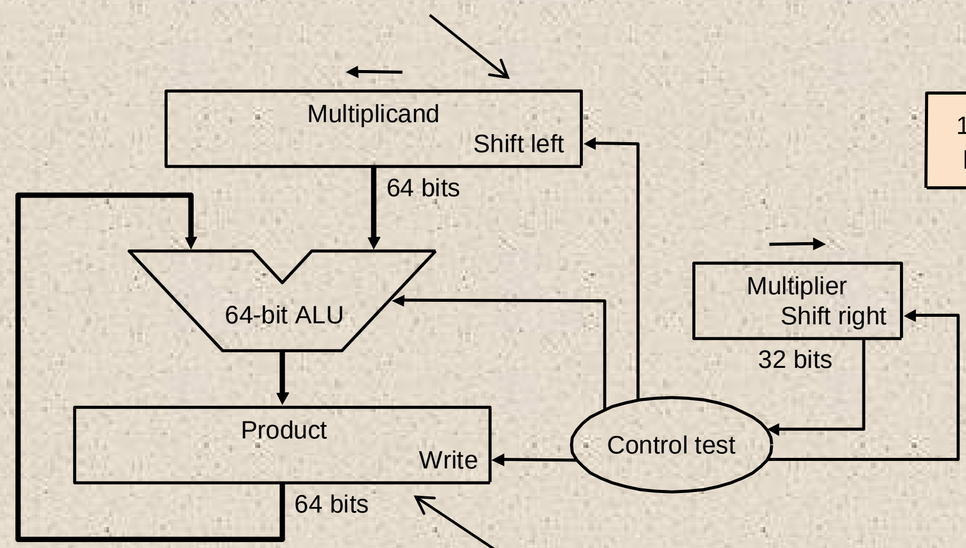
Refer to:

Chapter 3 , page 184



1st Version of Multiplication Hardware

32-bit multiplicand starts at right half of multiplicand register

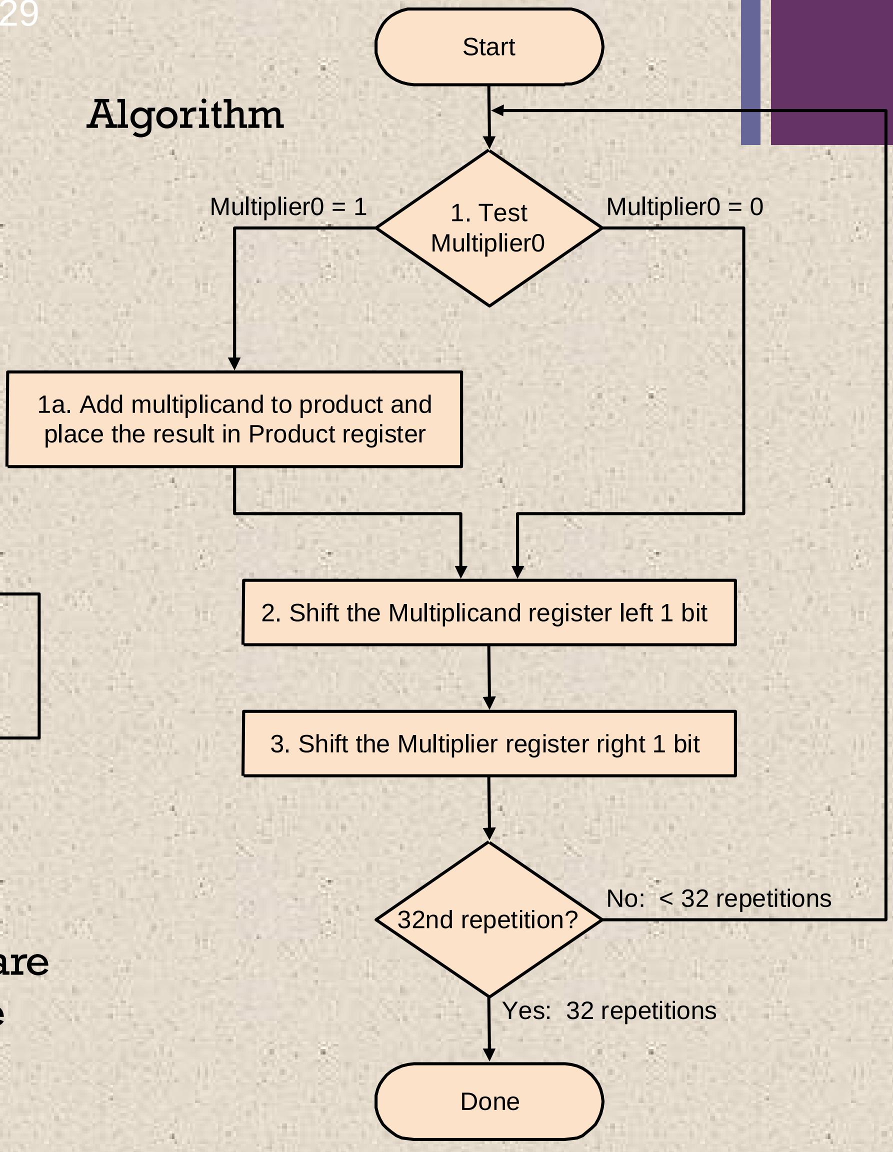


Product register is initialized at 0

Multiplicand register, product register, ALU are 64-bit wide; multiplier register is 32-bit wide

Flows of 1st Version Multiplication

Algorithm

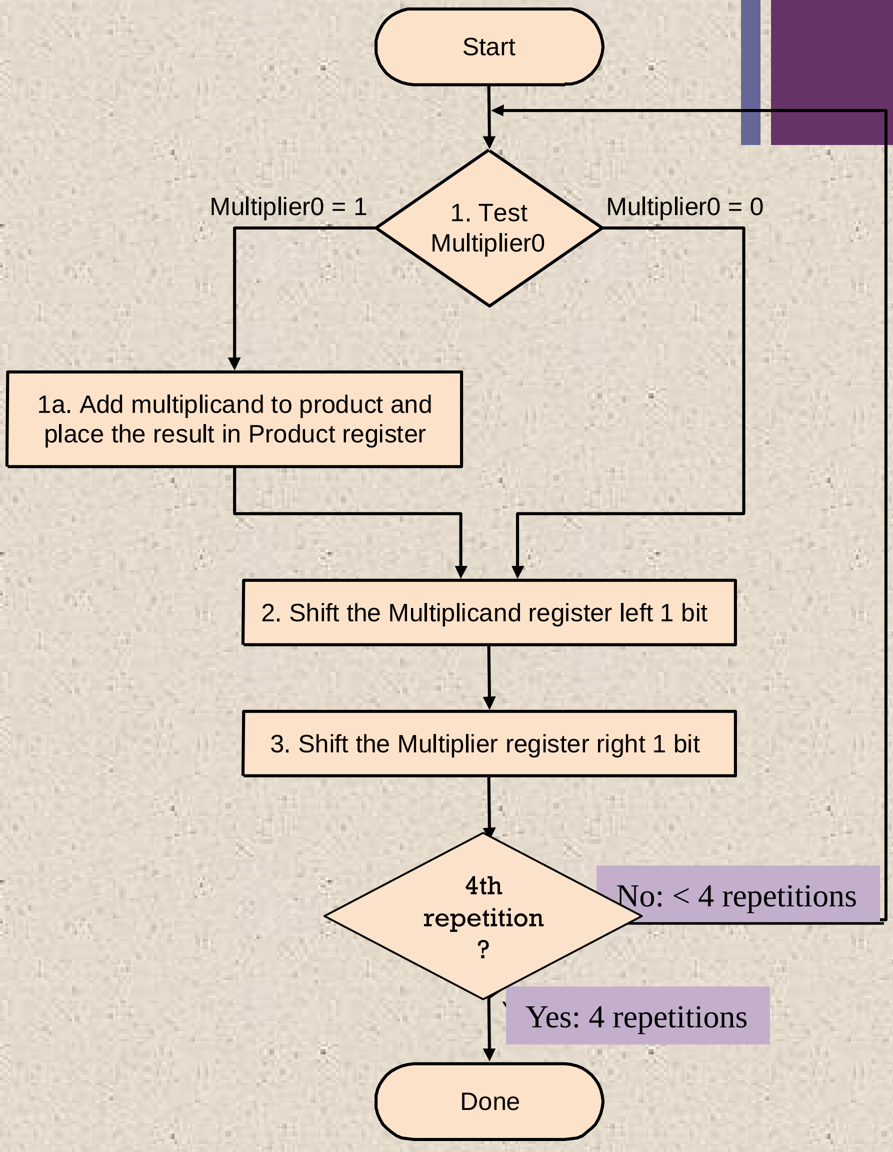


Example of Multiplication – 4 bits

- Example : 2 x 3 = ?

Multiplicand (MC) | Multiplier (MP) | Product (P)

- 2 x 3 ➔ 0010 x 0011
- Steps:
- 1a – test LSB for multiplier (0 or 1)
  - If 1 then P = P + MC
  - If 0 then no operation
- 2 – shift MC left
- 3 – shift MP right
- All bits done?
  - If still <max bit, repeat
  - If = max bit, stop



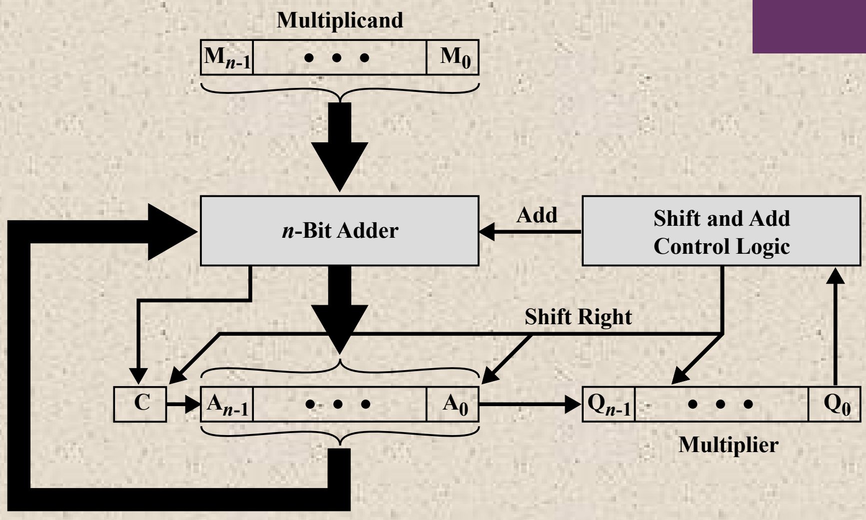
2 x 3 (0010 x 0011)

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
0	Initial value	0011	0000 0010	0000 0000
1	1a:1➔P = P + MC			0000 0010
	2: Shift MC left		0000 0100	
	3: Shift MP right	0001		
2	1a:1➔P = P + MC			0000 0110
	2: Shift MC left		0000 1000	
	3: Shift MP right	0000		
3	1a:0➔no operation			
	2: Shift MC left		0001 0000	
	3: Shift MP right	0000		
4	1a:0➔no operation			
	2: Shift MC left		0010 0000	
	3: Shift MP right	0000		

Exercise: Try with 5 x 4

Multiply Hardware - Version 2

1-bit carry register
32-bit Product register
32-bit Multiplier register
32-bit Multiplicand register



(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

Multiplication Version 2

	C	A	Q	MC	;C = carry
1.	0	0000	0011	0010	
2.	0	0010	0011	0010	;A = A+M
3.	0	0001	0001	0010	;SR
4.	0	0011	0001	0010	;A = A+M
5.	0	0001	1000	0010	;SR
6.	0	0000	1100	0010	;SR
7.	0	0000	0110	0010	;SR

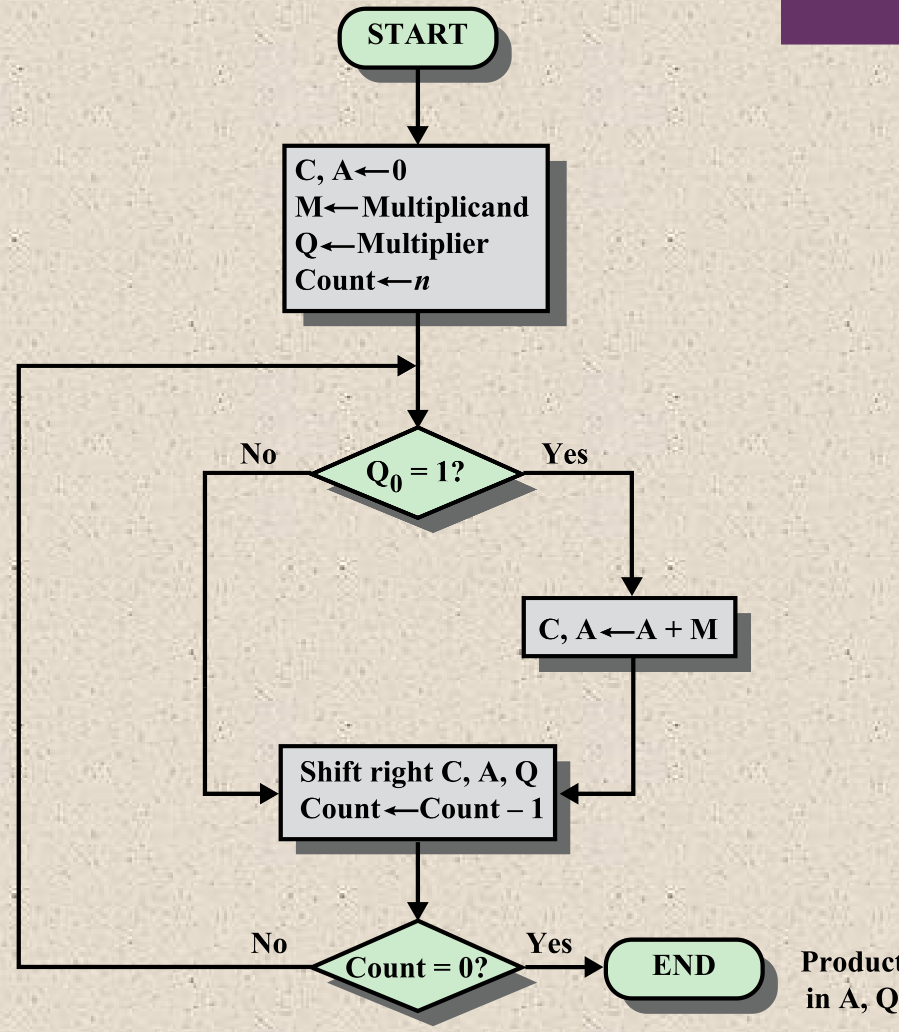


Figure 10.9 Flowchart for Unsigned Binary Multiplication

```
    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

(a) Unsigned integers		(b) Twos complement integers	
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

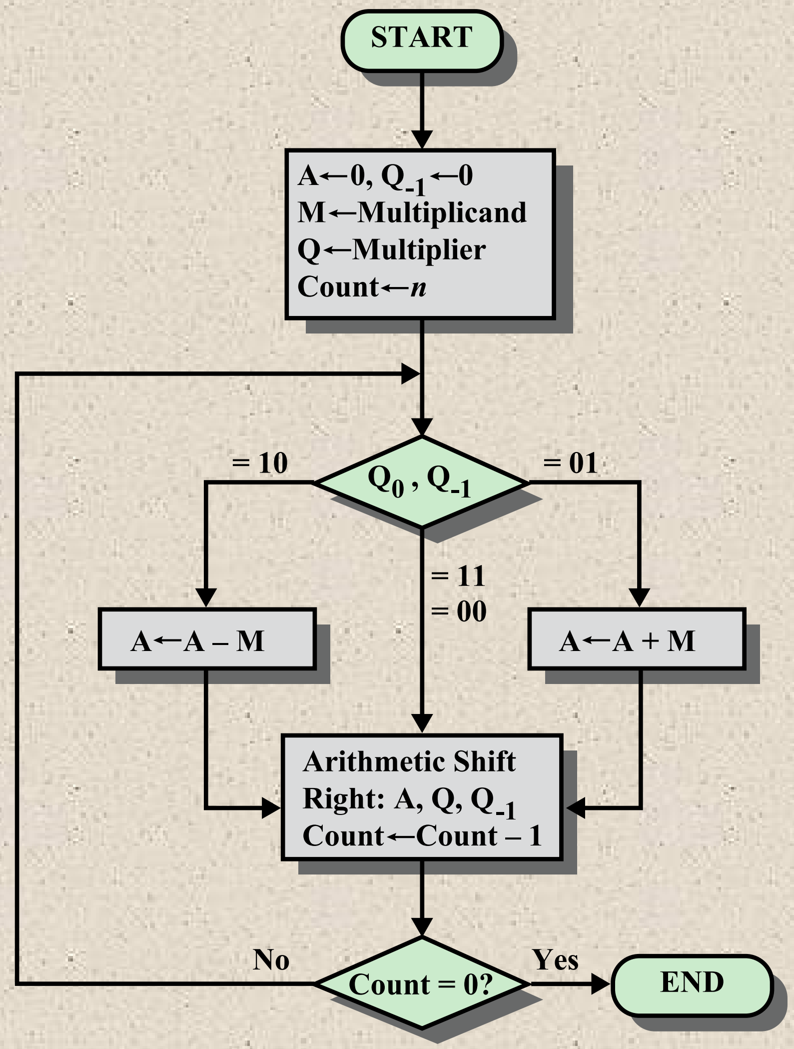


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

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

```
     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

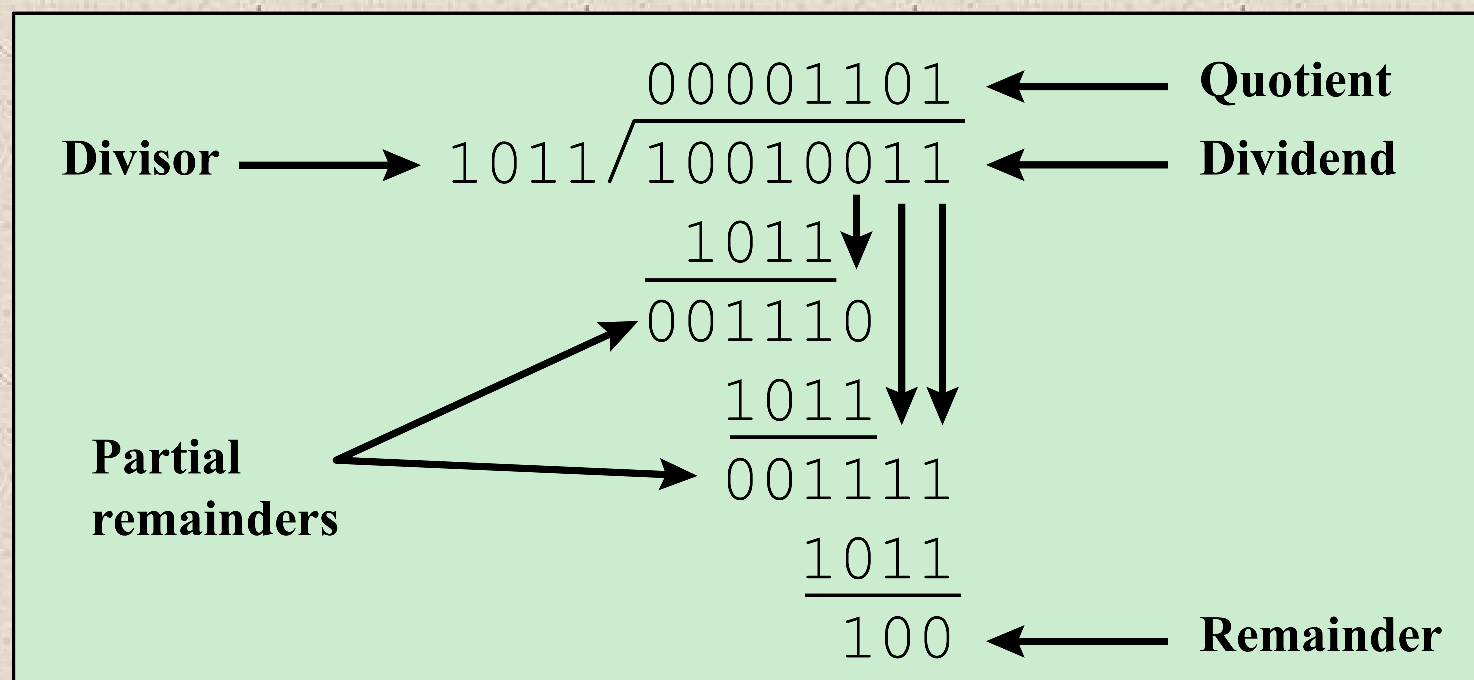


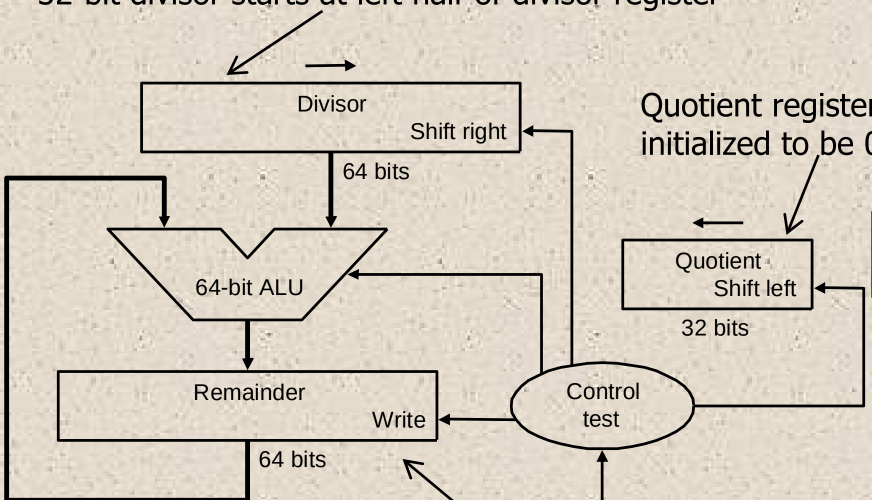
Figure 10.15 Example of Division of Unsigned Binary Integers

+ 1st Version of Division Hardware

Divisor starts at left half of divisor register

32-bit divisor starts at left half of divisor register

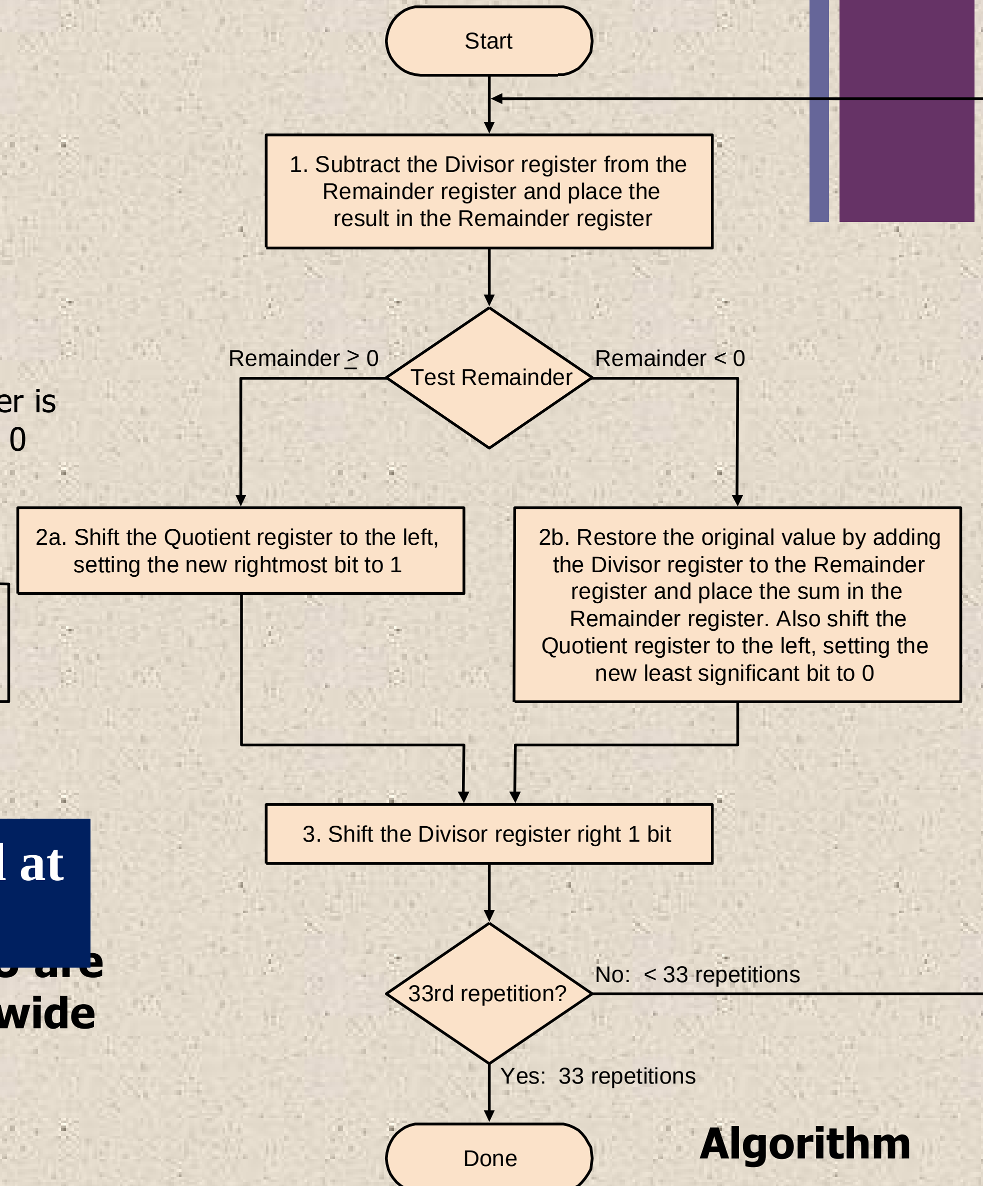
Quotient register is initialized to be 0



Remainder initialized with dividend at right

Divisor register, remainder register, ALU are 64-bit wide; quotient register is 32-bit wide

Flows of 1st Version Division



Algorithm

Example of Division – 4 bits

- Example : 7 / 2 = ?

Dividend (DD)

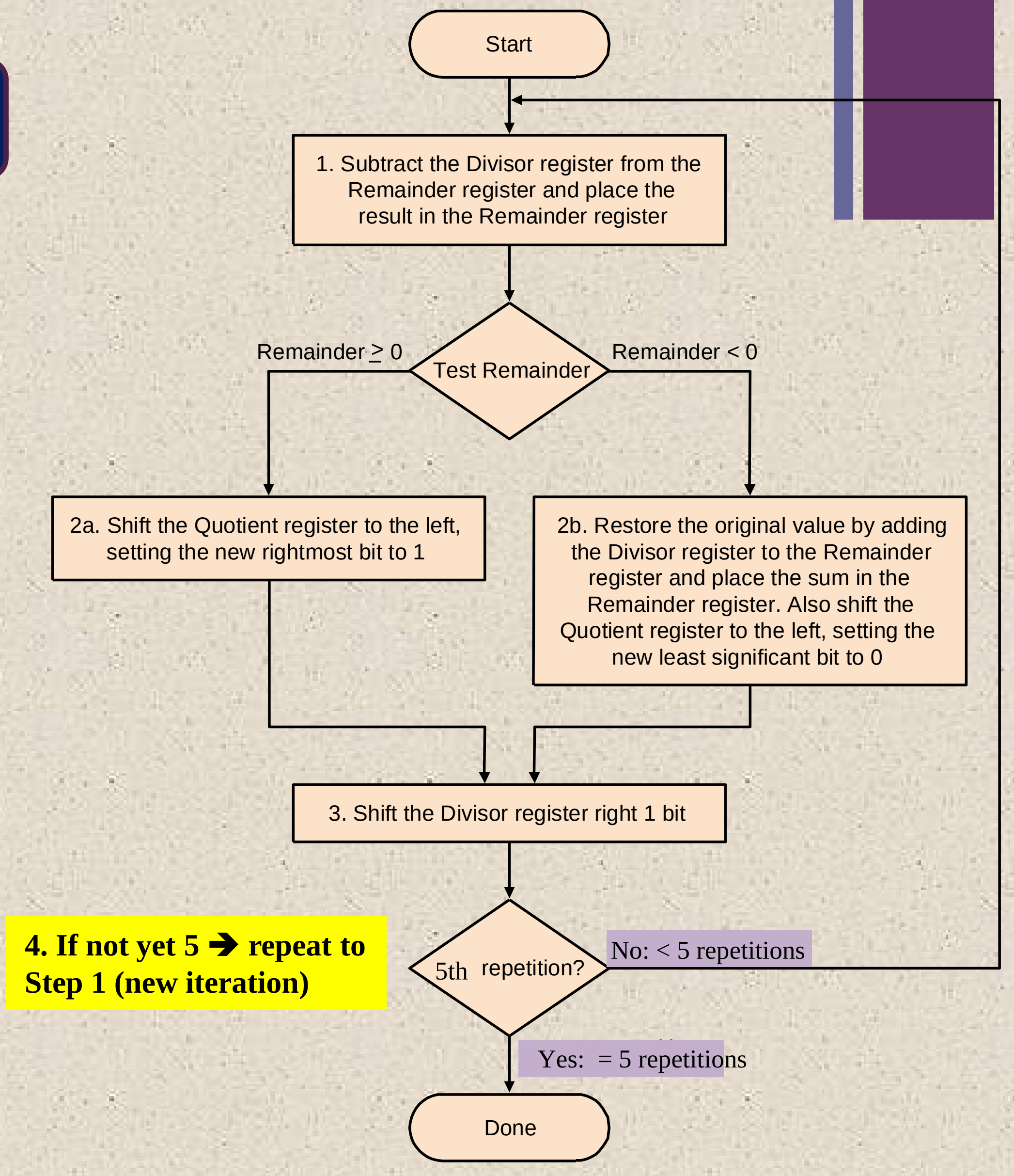
Divisor (D)

Quotient (Q)

- 7 / 2 ➔ 0111 / 0010

Steps:

- 1 – Remainder (R) = R – D
- 2 – test new **R** (**>=0 or <0**)
- 2a - If R**>=0** then
  - R = no operation;
  - Q = Shift left (add 1 at LSB)
- 2b - If R**<0** then
  - R = D + R
  - Q = Shift left (add 0 at LSB)
- 3 – shift D right
- All bits done?
  - If still **<(max bit + 1)**, repeat
  - If **= (max bit+1)**, stop



Iteration	Example: 7 ÷ 2 (0111 ÷ 0010) Step	Quotient (Q)	Divisor (D)	Remainder (R)
0	Initial value	0000	0010 0000	0000 0111
1	1. R = R - D			1110 0111
	2b. **R < 0**; R = D + R			0000 0111
	Q = Shift left (add 0 at LSB)	0000		
	3. D = Shift right		0001 0000	
2	1. R = R - D			1111 0111
	2b. **R < 0**; R = D + R			0000 0111
	Q = Shift left (add 0 at LSB)	0000		
	3. D = Shift right		0000 1000	

Iteration	Step	Quotient (Q)	Divisor (D)	Remainder (R)
		0000	0000 1000	
3	1. R = R – D			**1**111 1111
	2b. **R < 0**; R = D + R			0000 0111
	Q = Shift left (add 0 at LSB)	0000		
	3. D = Shift right		0000 0100	
4	1. R = R - D			**0**000 0011
	2a. **R >=0**; R = no operation			
	Q = Shift left (add 1 at LSB)	0001		
	3. D =		0000 0010	
5	1. R =			**0**000 0001
	2a. **R**			
	Q = Shift left (add 1 at LSB)	0011		
	3. D = Shift right		0000 0001	

7/2 = 3 remainder 1

3

1

Exercise: Try with 6 / 4

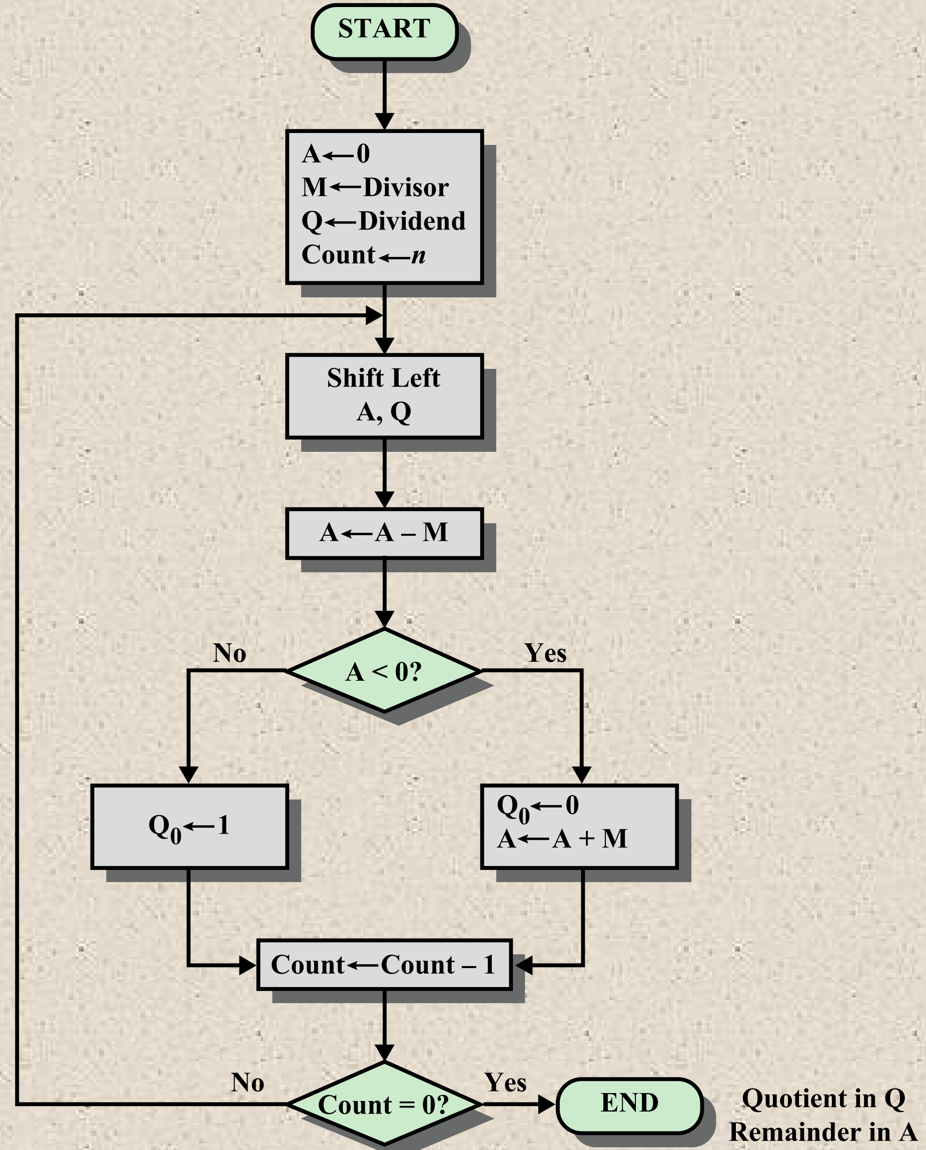


Figure 10.16 Flowchart for Unsigned Binary Division

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

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

10.4
Floating-Point Representation

Principles

+

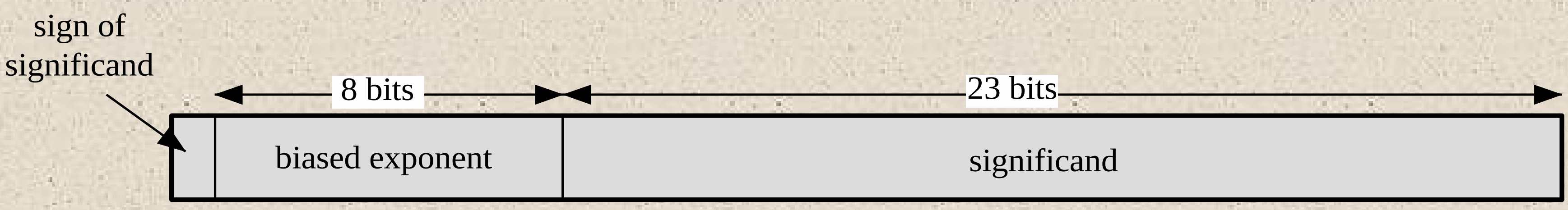
IEEE Standards for Binary-Floating Point Representation

+

10.4 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



(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 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:

$$0.110 * 2^5$$
$$110 * 2^2$$
$$0.0110 * 2^6 \text{ (unnormalized)}$$

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

\+

Normalization Process

- Normalization is the process of deleting the zeroes until a non-zero value is detected.
- Example :

➲Move radix point to the right (in this case 2 points)

0.00234×10^4

➔ $0.234 \times 10^{4-2}$ ➔ 0.234×10^2

⮌ Move radix point to the left (in this case 2 points)

12.0024×10^4

➔ $0.120024 \times 10^{4+2}$ ➔ 0.120×10^6

A rule of thumb:

moving the radix point **to the right** → **subtract exponent**

moving the radix point **to the left** → **add exponent**

Accurate Arithmetic: Overflow & Underflow

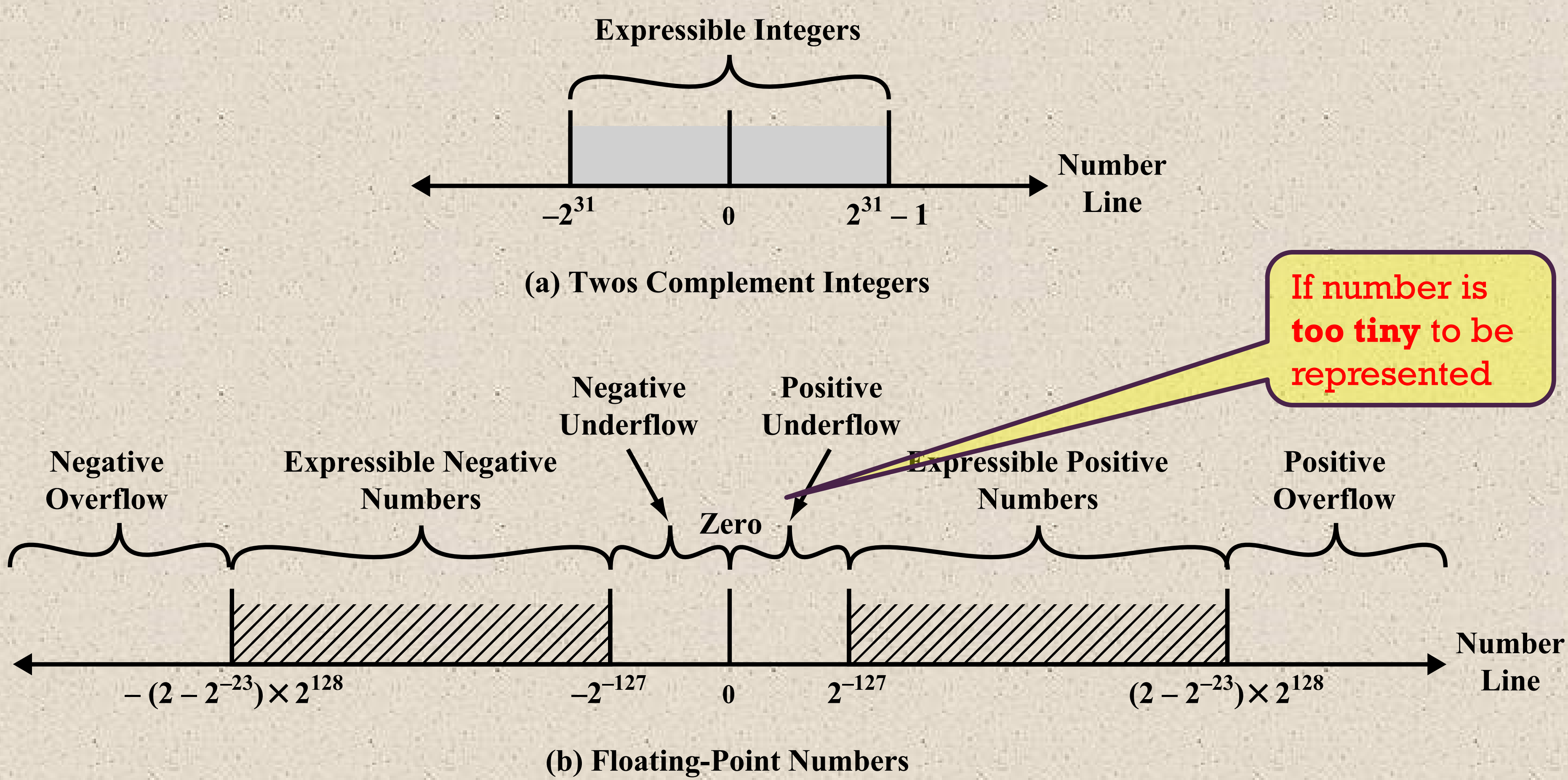


Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

IEEE Standard 754

Most important floating-point representation is defined

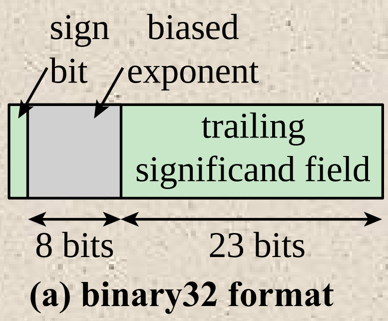
Standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs

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

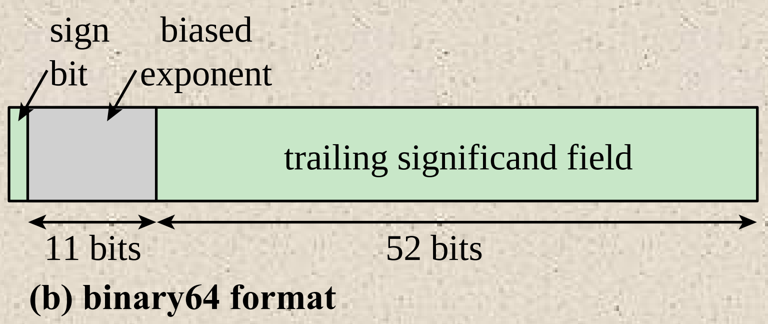
IEEE 754-2008 covers both binary and decimal floating-point representations

+
IEEE 754-2008

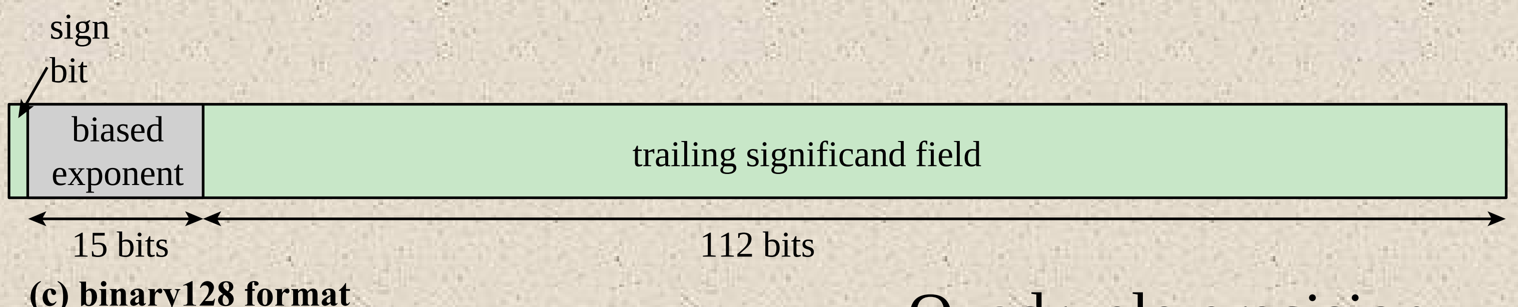
- Defines the following different types of floating-point formats:
- Arithmetic format
  - All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.
- Basic format
  - 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.



Single precision



Double precision



Quadruple precision

Figure 10.21 IEEE 754 Formats

Table 10.3 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}

* not including implied bit and not including sign bit

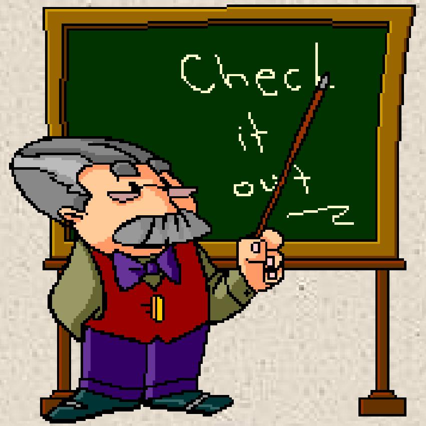
Additional Formats

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



+

IEEE 754 Floating-Point Standard

- Used in virtually every computer invented since 1980
- To pack more bits into the significand, the leading 1 bit of normalized binary number is made implicit
  - Original: $1.xxxxxxxx_{two} \times 2^{yyyy} \rightarrow (-1)^S \times F \times 2^E$
  - Modified: $(-1)^S \times (1+\text{Fraction}) \times 2^E$
  - Significand: 1 plus the fraction
    - Single precision: 24 bits
    - Double precision: 53 bits

The 1 in (1 + Fraction) is made implicit ➔ to pack more bits into the significand

Normalized Scientific Notation in IEEE 754

- In IEEE standard for normalization (used in computers), a floating point number is said to be normalized if there is only a single non-zero before the radix point.

- Example:

there is only a single non-zero before the radix point.

123.456	➔ normalized	1.23456×10^{2}
1010.1011_{B}	➔ normalized	1.0101011×2^{011}

Biased Notation in IEEE 754

- The desired notation must represent the most negative exponent as $00...00_{two}$ and the most positive as $11...11_{two}$
  - IEEE 754 uses a bias of 127 for single precision (and 1023 for double precision)
    - $-1 \rightarrow -1+127_{ten} = 0111\ 1110_{two}$
    - $+1 \rightarrow 1+127_{ten} = 1000\ 0000_{two}$

Bias
➔In single precision is **127**
➔In double precision **1023**

- A fixed value, called the bias, is subtracted from the field to get the true exponent value.
- An advantage of biased representation is that nonnegative floating-point numbers can be treated as integers for comparison purposes.

Single-Precision Range

- Exponents 00000000 and 11111111 are reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110 $= 254_{10}$
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

+DOUBLE-PRECISION RANGE

- Exponents 0000…00 and 1111…11 are reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

IEEE 754 Conversion

- To convert a decimal number to single (or double) precision floating point:
  - Step 1: Normalize
  - Step 2: Determine Sign Bit
  - Step 3: Determine exponent
  - Step 4: Determine Significand

IEEE 754 Conversion : Example 1

- Convert 10.4_d to ***single precision*** floating point.

- **Step 1: Normalize**

10 ➔ 00001010

0.4 x 2 = 0.8 ➔ 0
0.8 x 2 = 1.6 ➔ 1
0.6 x 2 = 1.2 ➔ 1
0.2 x 2 = 0.4 ➔ 0
0.4 x 2 = 0.8 ➔ 0
0.8 x 2 = 1.6 ➔ 1

For continuous results, take the 1st pattern before it repeats itself

0.4 = .0110

10.4 = 1010.0110 x 2^0

➔ 1.0100110 x 2^3

IEEE 754 Conversion : Example 1

- **Step 2: Determine Sign Bit (S)**
  - Because (10.4) is positive, S = 0

➔ 1.0100110×2^3

- **Step 3: Determine exponent**
  - Because its single precision ➔ bias = 127
  - Exponent = 3 + bias

    = 3 + 127

    = 130_d

    = **$1000\ 0010_b$**

IEEE 754 Conversion : Example 1

- **Step4: Determine Significand**
  - Drop the leading 1 of the significand

1.0100110 x 2^3 **→ 0100110**

- Then expand (padding) to 23 bits

01001100000000000000000

sign	Exponent	Significand
0	10000010	01001100000000000000000

Exponent = $1000\ 0010_b$

IEEE 754 Conversion : EXERCISE

- Convert -0.75_d to:
  - single precision floating point.
  - double precision floating point.

Converting Binary to Decimal Floating-Point

Remember: Biased notation ➔ (-1)sign x (1 + Fraction) x 2 $^{(exponent-bias)}$

- What decimal number is represented by this single precision float?

Sign (1 bit)	Exponent(8	Significand(23 bit)
1	10000001	01000000000000000000000

- Extract the values:

Sign = 1
Exponent = 10000001b = 129d

The Fraction = **-(1 + 0.25)**

Significand
= $(0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3})$
= ¼ = 0.25

The number
= - (1.25 x 2 $^{(exponent-bias)}$)
= - (1.25 x 2 $^{(129 - 127)}$)
= - (1.25 x 2^2)
= - (1.25 x 4) = -5.0

Math basic … fraction number!

- .154 = 1/10 + 5/100 + 4/1000

$$1.\frac{1}{10^1} + 5.\frac{1}{10^2} + 4.\frac{1}{10^3}$$

Decimal = base 10

Binary = base 2

- .1011 = 1/2 + 0/4 + 1/8 + 1/16

$$1.\frac{1}{2^1} + 0.\frac{1}{2^2} + 1.\frac{1}{2^3} + 1.\frac{1}{2^4}$$

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* ´ 32 for *n* > 4)	X		X
decimal64	X	X	X
decimal128	X	X	X
decimal{*k*} (*k* = *n* ´ 32 for *n* > 4)	X		X
extended precision	X		
extendable precision	X		

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

	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	–∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 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	$f \neq 0$	$-2_{e-126}(0.f)$

(a) binary32 format

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

	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	$f \neq 0$	$-2_{e-1022}(0.f)$

(a) binary64 format

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

	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	–∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	all 1s	f	$2_{e-16383}(1.f)$
negative normal nonzero	1	all 1s	f	$-2_{e-16383}(1.f)$
positive subnormal	0	0	f ≠ 0	$2_{e-16383}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2_{e-16383}(0.f)$

(a) binary128 format

10.5
Floating-Point Arithmetic

Addition and Subtraction
Multiplication and Division
Precision Considerations
IEEE Standard for Binary Floating-Point Arithmetic

Table 10.6 Floating-Point Numbers and Arithmetic Operations

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

Examples:

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

$X + Y = (0.3 \times 10_{2-3} + 0.2) \times 10_3 = 0.23 \times 10_3 = 230$
$X - Y = (0.3 \times 10_{2-3} - 0.2) \times 10_3 = (-0.17) \times 10_3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10_{2+3} = 0.06 \times 10_5 = 6000$
$X \div Y = (0.3 \div 0.2) \times 10_{2-3} = 1.5 \times 10_{-1} = 0.15$

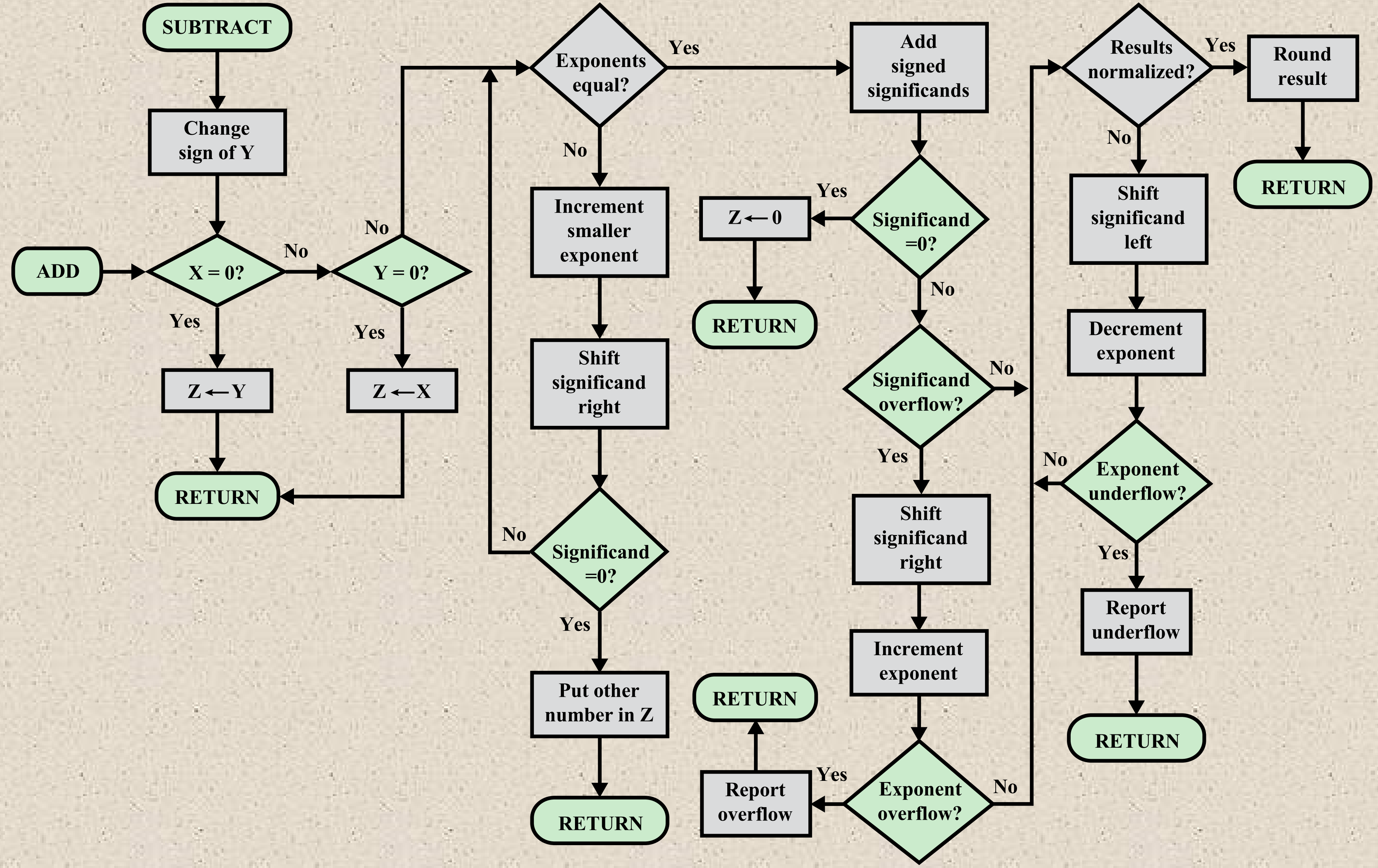
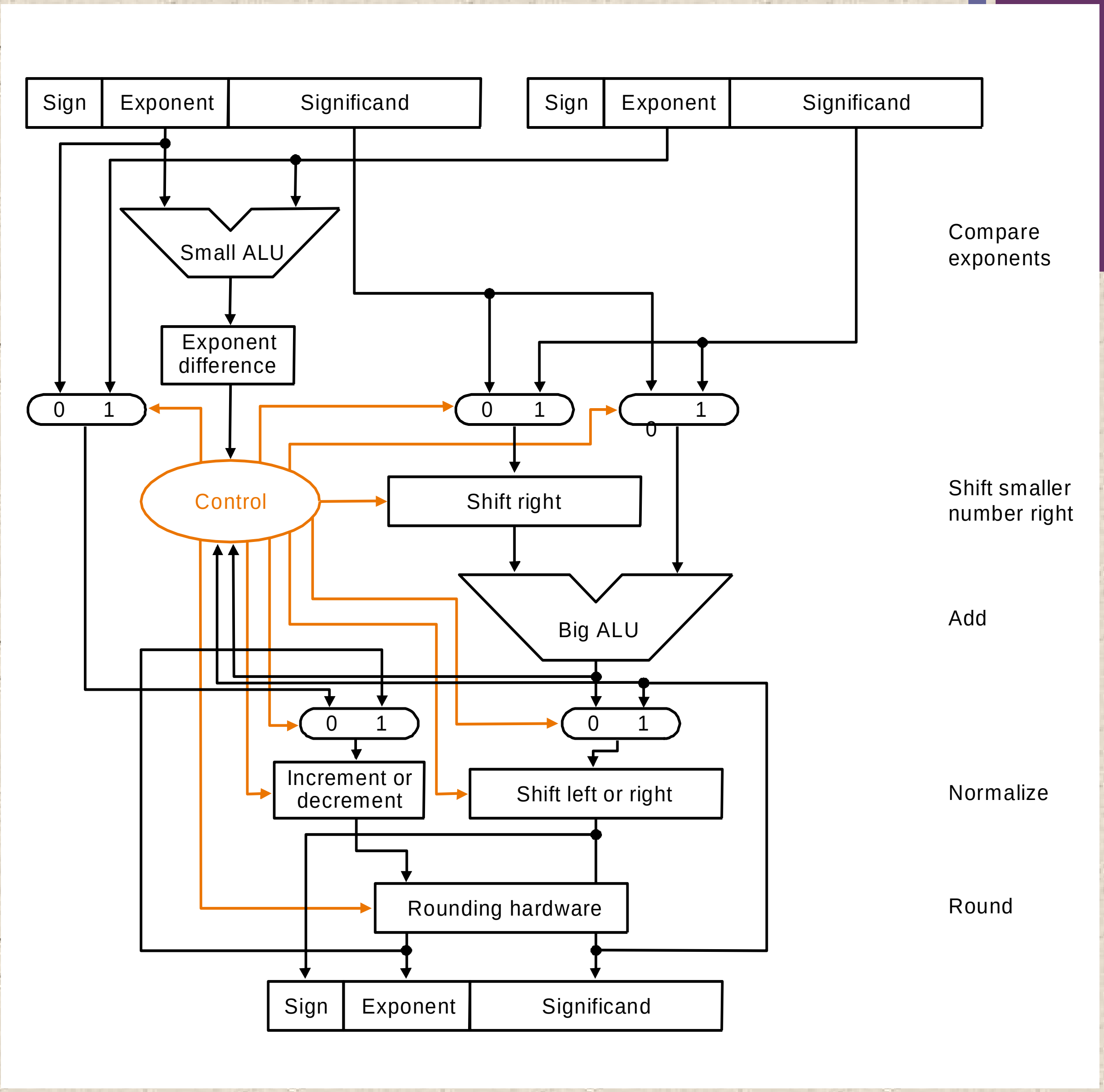


Figure 10.22 Floating-Point Addition and Subtraction (Z ← X ± Y)

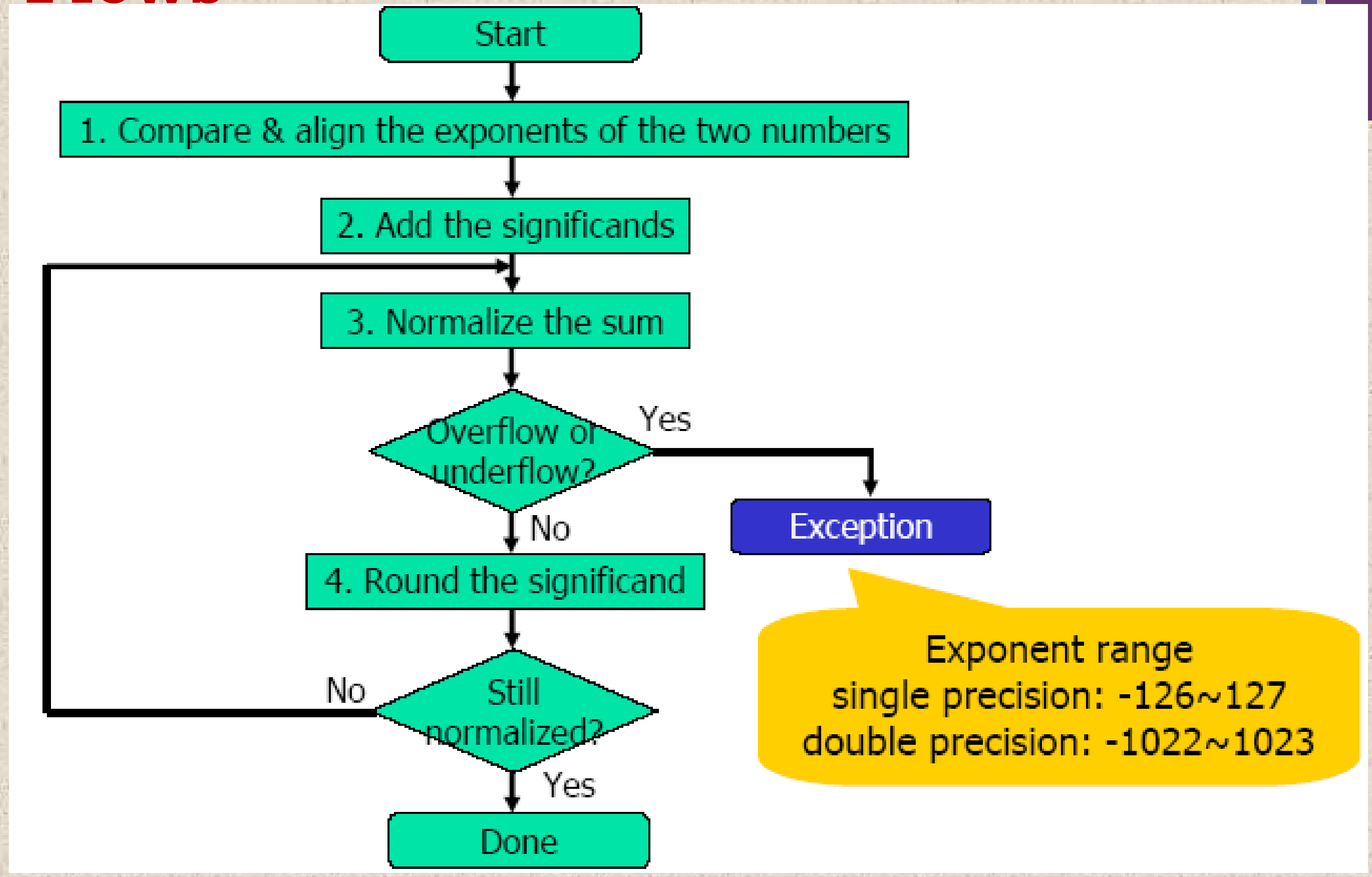
Block diagram of an arithmetic unit dedicated to floating-point addition.

+

Floating-Point ALU



Simplified Floating-Point Addition Flows



Decimal Floating-Point Addition

- Assume 4 decimal digits for significand and 2 decimal digits for exponent

Step 1: Align the decimal point of the number that has the smaller exponent

Step 2: Add the significand

- check for overflow/underflow of the significant

Step 3: Normalize the sum

- check for overflow/underflow of the exponent after normalisation

Step 4: Round the significand

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

Step 5: Normalize it (if need be)

A) Decimal Floating-Point Addition

Example: $9.999_d \times 10^1 + 1.610_d \times 10^{-1}$

- **Step 1**: Align the decimal point of the number that has the smaller exponent

Make $1.610_d \times \mathbf{10^{-1}}$ to $\mathbf{10^1}$

➔ $-1 + x = 1$ ➔ $x = 2$ → move 2 to left

➔ $0.0161_d \times 10^1$

- **Step 2**: add the significand

$$
\begin{array}{rl}
 9.9990 & \times 10^1 \\
+\ 0.0161_d & \times 10^1 \\
\hline
10.0151 & \times 10^1
\end{array}
$$

A) Decimal Floating-Point Addition

Example: $9.999_d \times 10^1 + 1.610_d \times 10^{-1}$

- **Step 3**: Normalize the sum

$10.0151 \times 10^1 \rightarrow 1.00151 \times 10^2$

- **Step 4**: Round the significand (to 4 decimal digits for significand)

$1.00151 \times 10^2 \rightarrow 1.0015 \times 10^2$

- **Step 5**: Normalize it (if need be)

No need as its normalized

+

B) Binary Floating-Point Addition

Example: $0.5_d + (-0.4375_d)$

- Convert the numbers to binary

$0.5 \rightarrow 0.10_b \times 2^0 \rightarrow 1.0_b \times 2^{-1}$

$-0.4375 \rightarrow -0.0111_b \times 2^0 \rightarrow -1.11_b \times 2^{-2}$

- **Step 1**: Align the decimal point of the number that has the smaller exponent

Make $1.11_b \times 2^{-2}$ to 2^{-1}

$\rightarrow -2 + x = -1 \rightarrow x = 1 \rightarrow$ move 1 to left

$\rightarrow - 0.111_b \times 2^{-1}$

B) Binary Floating-Point Addition

Example: $0.5_d + (-0.4375_d)$

- **Step 2:** add the significand

$$\begin{array}{r} 1.000 \times 2^{-1} \\ + \; -0.111 \times 2^{-1} \\ \hline \end{array}$$

$$\begin{array}{r} 1.000 \times 2^{-1} \\ - \; 0.111 \times 2^{-1} \\ \hline 0.001 \times 2^{-1} \end{array}$$

- **Step 3**: Normalize the sum

$0.001 \times 2^{-1} \rightarrow 1.0000 \times 2^{-4}$

- **Step 4:** Round the significand (to 4 decimal digits for significand)

Fits in the 4 decimal digits

- **Step 5**: Normalize it (if need be)

No need as its normalized

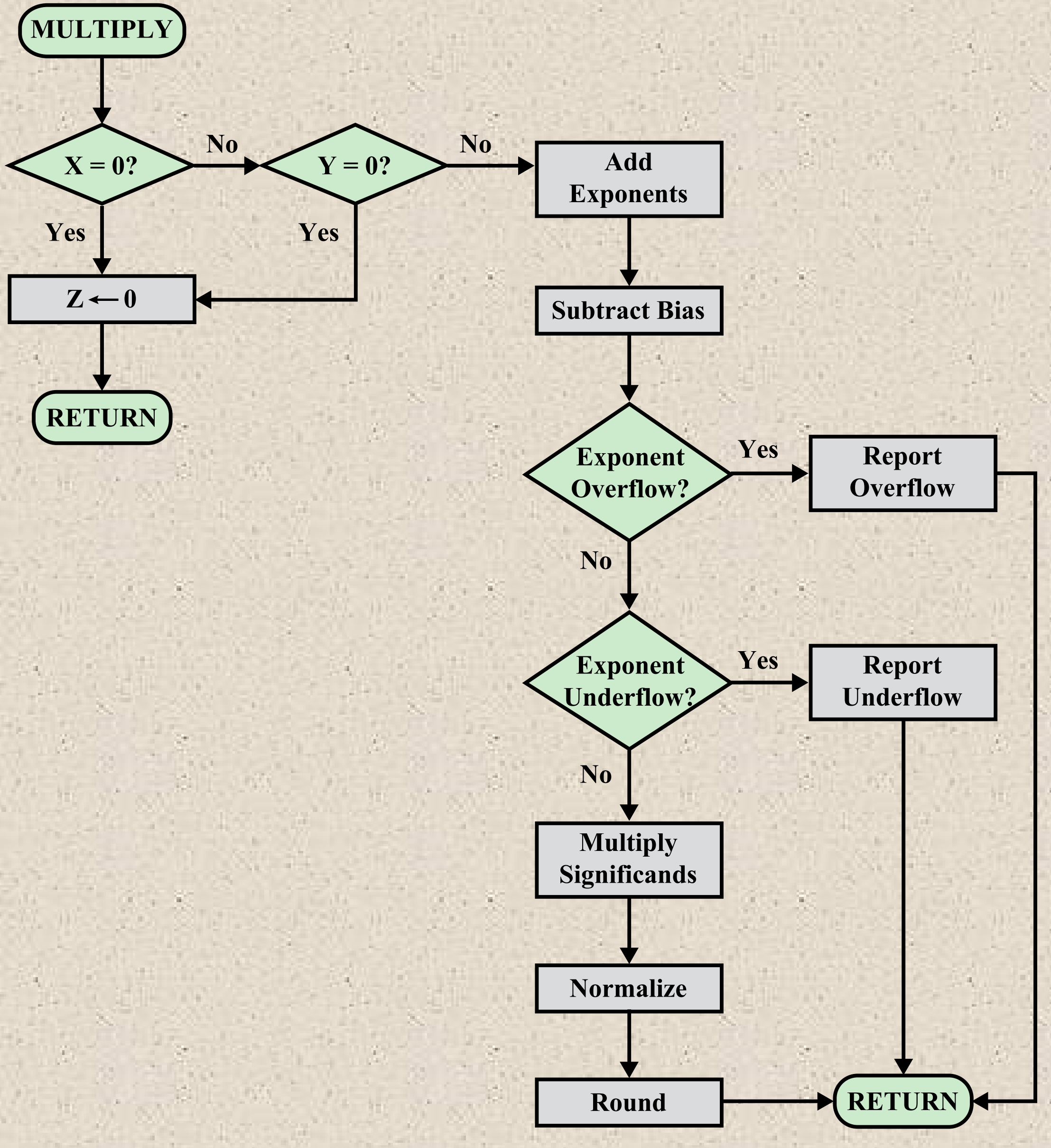


Figure 10.23 Floating-Point Multiplication (Z ← X × Y)

Floating-Point Multiplication

- Step 1: Add the exponent of the 2 numbers
- Step 2: Subtract bias
  - check for overflow/underflow of the exponent after normalization
- Step 3: Multiply the significands
- Step 4: Normalize the product
- Step 5: Round the significand
  - If the significand does not fit in the space reserved for it, it has to be rounded off
- Step 6: Set the sign of the product

A) Floating-Point Multiplication (decimal)

Example: $(1.110_d \times 10^{10}) \times (9.200_d \times 10^{-5})$

- Assume 4 decimal digits for significand and 2 decimal digits for exponent
- Step 1: Add the exponent of the 2 numbers

  $10 + (-5) = 5$

- Step 2: Subtract bias

$10 + (-5) + 127 = 132$

+

A) Floating-Point Multiplication (decimal)

Example: $(1.110_d \times 10^{10}) \times (9.200_d \times 10^{-5})$

- Assume 4 decimal digits for significand and 2 decimal digits for exponent

- Step 1: Add the exponent of the 2 numbers

$10 + (-5) = 5$

If biased is considered ➔ $10 + (-5) + 127 = 132$

- Step 2: Subtract bias

- Step 2: Multiply the significands

```
     9.200
 x   1.110
--------------
    92000
   9200
  9200
--------------
10.212000
```

➔ 10.212000 ➔ 10.2120×10^5

A) Floating-Point Multiplication (decimal)

Example: $(1.110_d \times 10^{10}) \times (9.200_d \times 10^{-5})$

- Step 3: Normalize the product

  $10.2120 \times 10^{5} \rightarrow 1.02120 \times 10^{6}$

- Step 4: Round the significand (4 decimal digits for significand)

  1.0212×10^{6}

- Step 5: Normalize it (if need be)

  Still normalized

- Step 6: Set the sign of the product

  $+1.0212 \times 10^{6}$

B) Floating-Point Multiplication (binary)

Example: $(1.000_b \times 2^{-1}) \times (-1.110_b \times 2^{-2})$

- Assume 4 binary digits for significand and 2 binary digits for exponent
- Step 1: Add the exponent of the 2 numbers

$-1 + (-2) = -3$

If biased is considered ➔ $-1 + (-2) + 127 = 124$

- Step 2: Multiply the significands

```
    1.110
 x  1.000
---------------------
 1110000

-> 1.110000
```

➔ 1.110000 ➔ 1.110000×2^{-3}

B) Floating-Point Multiplication (binar

Example: $(1.000_b \times 2^{-1}) \times (-1.110_b \times 2^{-2})$

- Step 3: Normalize the product

  1.110000×2^{-3} → already normalized

- Step 4: Round the significand (4 binary digits for significand)

  1.1100×2^{-3}

- Step 5: Normalize it (if need be)

  Still normalized

- Step 6: Set the sign of the product

  $-1.1100_b \times 2^{-3}$ → $-7/32_d$

Precision Considerations

Rounding

- IEEE standard approaches:
  - Round to nearest:
    - The result is rounded to the nearest representable number.
  - Round toward +∞ :
    - The result is rounded up toward plus infinity.
  - Round toward -∞:
    - The result is rounded down toward negative infinity.
  - Round toward 0:
    - The result is rounded toward zero.

Interval Arithmetic

- Provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result
- The two values correspond to the lower and upper endpoints of an interval that contains the true result
- The width of the interval indicates the accuracy of the result
- If the endpoints are not representable then the interval endpoints are rounded down and up respectively
- If the range between the upper and lower bounds is sufficiently narrow then a sufficiently accurate result has been obtained
- *Minus infinity* and *rounding to plus* are useful in implementing interval arithmetic

Truncation

- *Round toward zero*
- Extra bits are ignored
- Simplest technique
- A consistent bias toward zero in the operation
  - Serious bias because it affects every operation for which there are nonzero extra bits

+ Summary

Chapter 10

Computer Arithmetic

- 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

- 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