

Computer Organization and Architecture

Computer Arithmetic

Er. Sujan Karki (IOE Purwanchal Campus)

May 22, 2025



Integer Representation

- An **8-bit word** can represent integers from **0 to 255**.
- Examples:
 - $00000000 = 0$
 - $00000001 = 1$
 -
 - $11111111 = 255$
- In general, for an **n-bit binary number** $a_{n-1}, a_{n-2}, \dots, a_1, a_0$:

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

- Each bit contributes its value multiplied by the corresponding power of 2.

Unsigned vs. Signed Numbers

Unsigned Number

- An **unsigned number** represents only non-negative values (zero and positive numbers).
- All available bits are used to represent the magnitude of the number.
- For an *n-bit unsigned number*, the range is:

$$\text{Range} = 0 \text{ to } (2^n - 1)$$

- The range of an 8-bit unsigned number is: 0 to 255.

Signed Number

- A **signed number** uses one bit (typically the most significant bit, MSB) to represent the sign of the number (positive or negative).
- The remaining bits represent the magnitude of the number. The range is split between positive and negative values.
- The range of an 8-bit signed number is: -128 to 127.
- For an *n-bit signed number*, the range is:

$$\text{Range} = -2^{n-1} \text{ to } (2^{n-1} - 1)$$

Sign magnitude representation:

- Represents both positive and negative integers in binary.
- The **most significant bit (MSB)** is the **sign bit**:
 - 0 \rightarrow positive
 - 1 \rightarrow negative
- Remaining $n-1$ bits represent the **magnitude**.
- $+18 \rightarrow 00010010$ and $+127 \rightarrow 01111111$
- $-18 \rightarrow 10010010$ and $-127 \rightarrow 11111111$

Drawbacks of Sign-Magnitude Representation

- ❶ **Complex arithmetic:** Signs and magnitudes must be handled separately.
- ❷ **Two representations of zero:**
 - $+0 \rightarrow 00000000$
 - $-0 \rightarrow 10000000$

This causes redundancy and inefficiency.

2's complement representation 2's complement representation is an alternative to sign-magnitude representation that also uses the most significant bit (MSB) as a sign bit. The key difference lies in how negative numbers are represented, making it more efficient for arithmetic operations such as addition and subtraction.

To find the 2's complement representation of a negative number:

- If the number is positive, the **sign bit** is 0, and the remaining bits represent the magnitude.
- For a positive integer A in n -bit representation:

$$A = \sum_{i=0}^{n-2} 2^i a_i \quad (\text{where } a_{n-1} = 0)$$

- Zero is represented as $000 \dots 0$ (all bits are zero).

To represent a negative number in 2's complement:

- Take the **1's complement** (invert all bits) of the binary representation of the positive number.
- Add 1 to the result.
- For an n -bit integer A :

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

Example

- Convert +18 to binary:

$$18_{10} = 00010010_2$$

(Using 8 bits)

- Convert -18 in 2's complement:

- ① Start with the binary representation of +18:

$$18_{10} = 00010010_2$$

- ② Take the 1's complement (invert each bit):

$$1's \text{ complement of } 00010010_2 = 11101101_2$$

- ③ Add 1 to the result:

$$11101101_2 + 1 = 11101110_2$$

Addition and Subtraction Algorithm

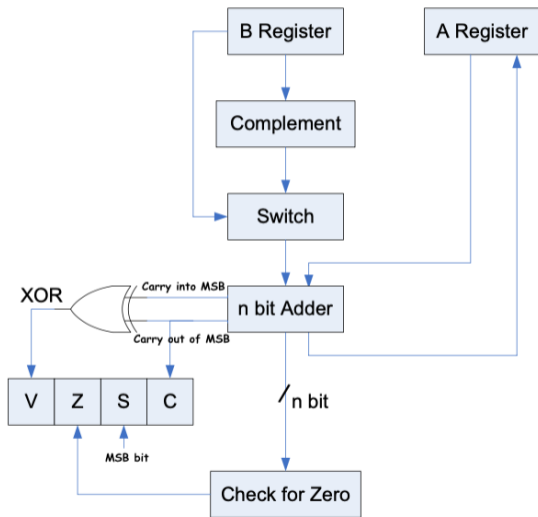


Fig: Block diagram of hardware for addition / subtraction

$\begin{array}{r} 1001 = -7 \\ + 0101 = 5 \\ \hline 1110 = -2 \end{array}$ <p>(a) $(-7) + (+5)$</p>	$\begin{array}{r} 1100 = -4 \\ + 0100 = 4 \\ \hline 10000 = 0 \end{array}$ <p>(b) $(-4) + (+4)$</p>
$\begin{array}{r} 0011 = 3 \\ + 0100 = 4 \\ \hline 0111 = 7 \end{array}$ <p>(c) $(+3) + (+4)$</p>	$\begin{array}{r} 1100 = -4 \\ + 1111 = -1 \\ \hline 11011 = -5 \end{array}$ <p>(d) $(-4) + (-1)$</p>
$\begin{array}{r} 0101 = 5 \\ + 0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$ <p>(e) $(+5) + (+4)$</p>	$\begin{array}{r} 1001 = -7 \\ + 1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$ <p>(f) $(-7) + (-6)$</p>

Figure 10.3 Addition of Numbers in Twos Complement Representation

- **V** = Overflow Flag = Carry into MSB \oplus Carry out of MSB.
- **Z** = Zero Flag = Set when all output bits are 0.
- **S** = Sign Flag = Set to the MSB of the result: 1 = negative, 0 = positive.
- **C** = Carry Flag = Set when there is a carry out from the MSB (in addition) or a borrow (in subtraction).

Subtraction

- In digital circuits, subtraction is performed using the 2's complement method.
- The subtrahend (B register) is passed through a **2's complement unit**.
- This transforms B into $-B$, so the adder performs:

$$A - B = A + (-B)$$

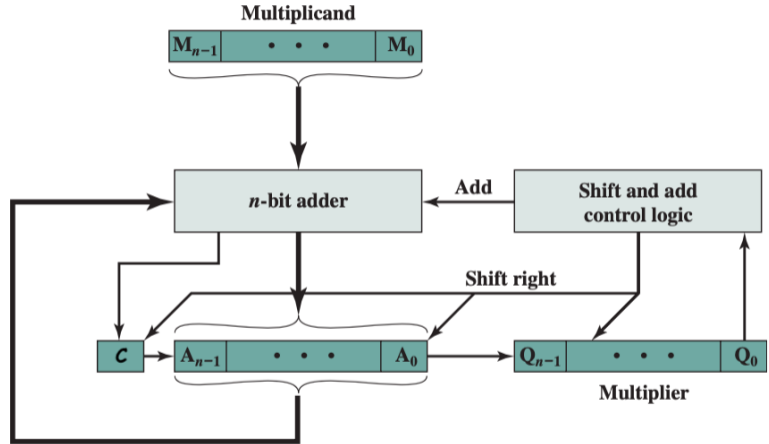
- This approach allows subtraction to be performed using the same hardware as addition.

Multiplication Algorithm(unsigned integers)

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

Figure 10.7 Multiplication of Unsigned Binary Integers

Paper and Pencil Approach



(a) Block diagram

Computerized Multiplication

- The multiplier and multiplicand bits are loaded into two registers: Q (multiplier) and M (multiplicand).
- A third register A is initially set to zero.
- C is a 1-bit register which holds the carry bit resulting from the addition.
- The control logic reads the bits of the multiplier one at a time.
- If $Q_0 = 1$, the multiplicand is added to register A , and the result is stored back in register A , with the C bit used for the carry.
- After addition, all bits of C, A, Q are shifted to the right by 1 bit:
 - The C bit goes to A_{n-1} ,
 - A_0 goes to Q_{n-1} ,
 - Q_0 is lost.
- If $Q_0 = 0$, no addition is performed, but the shift operation is still carried out.
- The process is repeated for each bit of the original multiplier.
- The resulting $2n$ -bit product is contained in the QA register.

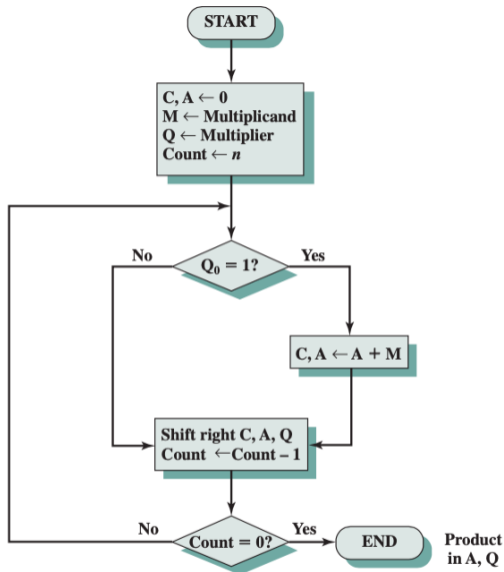


Figure 10.9 Flowchart for Unsigned Binary Multiplication

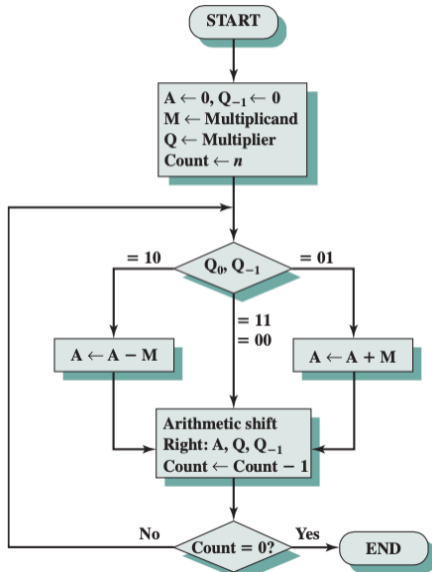
Example: Multiply 15 X 11 using unsigned binary method

- Accumulator (A) \leftarrow 0000
- Multiplier (Q) \leftarrow 1011
- Multiplicand (M) \leftarrow 1111
- Count \leftarrow 4 (number of bits of Q)

C	A	Q	M	Count	Remarks
0	0000	1011	1111	4	Initialization
0	1111	1011	-	3	Add ($A \leftarrow A + M$)
0	0111	1101	-		Logical Right Shift C, A, Q
1	0110	1101	-	2	Add ($A \leftarrow A + M$)
0	1011	0110	-		Logical Right Shift C, A, Q
0	0101	1011	-	1	Logical Right Shift C, A, Q
1	0100	1011	-	0	Add ($A \leftarrow A + M$)
0	1010	0101	-		Logical Right Shift C, A, Q

Result = 1010 0101 = $2^7 + 2^5 + 2^2 + 2^0 = 165$

Signed Multiplication (Booth Algorithm) – 2's Complement Multiplication



Multiply $9 \times -3 = -27$ using Booth Algorithm

- $+3 = 00011$, $-3 = 11101$ (2's complement of $+3$)
- Accumulator (A) $\leftarrow 0000$ and Multiplier (Q) $\leftarrow 11101$
- Multiplicand (M) $\leftarrow 01001$ and Count $\leftarrow 5$ (number of bits of Q)

A	Q	Q_{-1}	Add(M)	Sub($\overline{M} + 1$)	Count	Remarks
00000	11101	0	01001	10111	5	Initialization
10111	11101	0	-	-	-	Sub ($A \leftarrow A - M$)
11011	11110	1	-	-	4	Arithmetic Shift Right A, Q, Q_{-1}
00100	11110	1	-	-	-	Add ($A \leftarrow A + M$)
00010	01111	0	-	-	3	Arithmetic Shift Right A, Q, Q_{-1}
11001	01111	0	-	-	-	Sub ($A \leftarrow A - M$)
11110	10111	1	-	-	2	Arithmetic Shift Right A, Q, Q_{-1}
11110	01011	1	-	-	1	Arithmetic Shift Right A, Q, Q_{-1}
11111	00101	1	-	-	0	Arithmetic Shift Right A, Q, Q_{-1}

Result = 11111 00101 = $-2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^2 + 2^0 = -27$

Multiply $-9 \times -13 = ?$ using Booth Algorithm

- $-9 = 10111$, $-13 = 10011$ (5-bit 2's complement)
- Accumulator (A) $\leftarrow 00000$, Multiplier (Q) $\leftarrow 10011$
- Multiplicand (M) $\leftarrow 10111$, Count $\leftarrow 5$

A	Q	Q_{-1}	Add(M)	Sub($\overline{M} + 1$)	Count	Remarks
00000	10011	0	10111	01001	5	Initialization
01001	10011	0	-	-	-	Sub ($A \leftarrow A - M$)
00100	11001	1	-	-	4	Arithmetic Shift Right A, Q, Q_{-1}
00010	01100	1	-	-	3	Arithmetic Shift Right A, Q, Q_{-1}
11001	01100	1	-	-	-	Add ($A \leftarrow A + M$)
11100	10110	0	-	-	2	Arithmetic Shift Right A, Q, Q_{-1}
11110	01011	0	-	-	1	Arithmetic Shift Right A, Q, Q_{-1}
00111	01011	0	-	-	-	Add ($A \leftarrow A - M$)
00011	10101	1	-	-	0	Arithmetic Shift Right A, Q, Q_{-1}

Result = 00011 10101 = $2^6 + 2^4 + 2^4 + 2^2 + 2^0 = 117$

Division Algorithm(Restoring division)

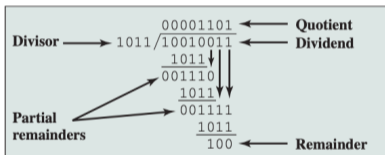


Figure 10.15 Example of Division of Unsigned Binary Integers

Division of Unsigned Binary Integers

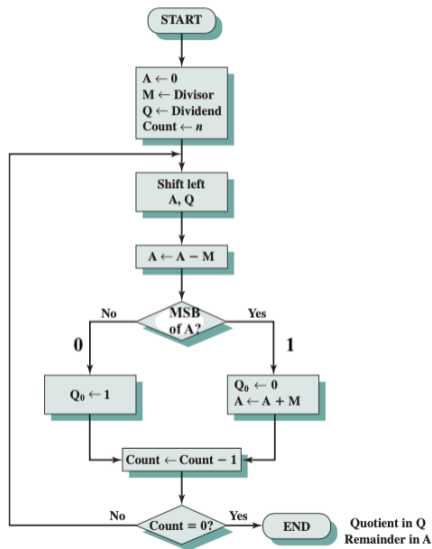


Figure 10.16 Flowchart for Unsigned Binary Division

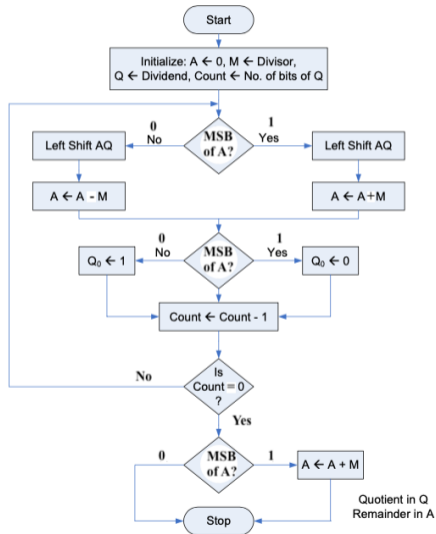
Divide 15 (1111) by 4 (0100) using Restoring Division

- Dividend (Q) = 1111 = 15, Divisor (M) = 0100 = 4, $\overline{M} + 1 = 11100$
- Initial Accumulator (A) = 00000, Count = 4

A	Q	M	$\overline{M} + 1$	Count	Remarks
00000	1111	00100	11100	4	Initialization
00001	111 □	-	-	-	Shift Left A, Q
11101	111 □	-	-	-	Sub ($A \leftarrow A - M$)
00001	111 0	-	-	3	$Q_0 \leftarrow 0$, Add ($A \leftarrow A + M$)
00011	110 □	-	-	-	Shift Left A, Q
11111	110 □	-	-	-	Sub ($A \leftarrow A - M$)
00011	110 0	-	-	2	$Q_0 \leftarrow 0$, Add ($A \leftarrow A + M$)
00111	100 □	-	-	-	Shift Left A, Q
00011	100 □	-	-	-	Sub ($A \leftarrow A - M$)
00011	100 1	-	-	1	$Q_0 \leftarrow 1$
00111	001 □	-	-	-	Shift Left A, Q
00011	001 □	-	-	-	$Q_0 \leftarrow 1$, Sub ($A \leftarrow A - M$)
00011	001 1	-	-	0	$Q_0 \leftarrow 1$

Quotient in Q = 0011 = 3 & Remainder in A = 00011 = 3

Non Restoring Division Algorithm



Divide 14 (1110) by 3 (0011) using Non-Restoring Division

- Dividend (Q) = 1110 = 14, Divisor (M) = 00011 = 3, $\overline{M} + 1 = 11101$
- Initial Accumulator (A) = 00000, Count = 4

A	Q	M	$\overline{M} + 1$	Count	Remarks
00000	1110	00011	11101	4	Initialization
00001	110 □	-	-	-	Shift Left A, Q
11110	110 □	-	-	-	Sub ($A \leftarrow A - M$)
11110	111 0	-	-	3	$Q_0 \leftarrow 0$
11101	100 □	-	-	-	Shift Left A, Q
00000	100 □	-	-	-	Add ($A \leftarrow A + M$)
00000	110 1	-	-	2	$Q_0 \leftarrow 1$
00001	001 □	-	-	-	Shift Left A, Q
11110	001 □	-	-	-	Sub ($A \leftarrow A - M$)
11110	001 0	-	-	1	$Q_0 \leftarrow 0$
11100	010 □	-	-	-	Shift Left A, Q
11111	010 □	-	-	-	Add ($A \leftarrow A + M$)
11111	010 0	-	-	0	$Q_0 \leftarrow 0$
00010	0100	-	-	-	Add ($A \leftarrow A + M$)

Quotient in Q = 0100 = 4 & Remainder in A = 00010 = 2

Floating Point Representation

With a fixed-point notation (e.g., two's complement), it is possible to represent a range of positive and negative integers centered on or near 0.

This approach has limitations. Very large numbers cannot be represented, nor can very small fractions. Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.

For decimal numbers, we get around this limitation by using scientific notation. Thus,

$$976,000,000,000,000 = 9.76 \times 10^{14} \quad \text{and} \quad 0.0000000000000976 = 9.76 \times 10^{-14}.$$

This same approach can be taken with binary numbers. We can represent a number in the form:

$$\pm M \times B^E$$

- This number can be stored in a binary word with three fields:
 - **Mantissa (M)**: The fractional part of the number
 - **Exponent (E)**: The power of the base B (usually 2 in binary)

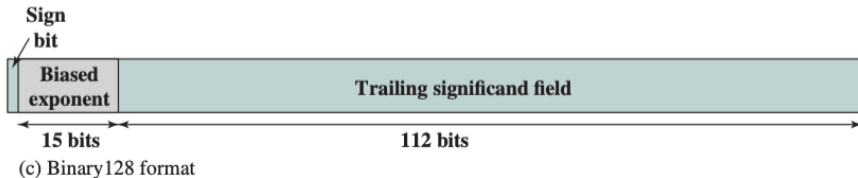
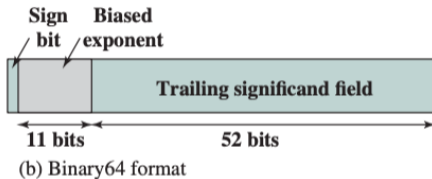
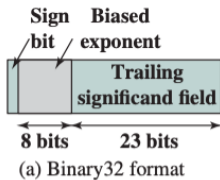


Figure 10.21 IEEE 754 Formats

Floating-Point Representation of 1001.11_2

Step 1: Convert to scientific notation:

We move the binary point 3 places to the left, giving:

$$1.00111_2 \times 2^3$$

Step 2: Determine mantissa and exponent:

- Mantissa: 1.00111_2
- Exponent: 3, with base 2

Step 3: Floating-point representation:

$$+1.00111_2 \times 2^3$$

- **Sign bit (S):** 0 (positive)
- **Exponent (E):** Exponent 3 with bias 127, so $3 + 127 = 130$, and $130_{10} = 10000010_2$
- **Mantissa (M):** Exclude the leading 1, so the mantissa is 001110

Final IEEE 754 single-precision (32-bit) representation:

0 10000010 001110000000000000000000

Floating-Point Representation of -1001.11_2

Step 1: Convert to scientific notation:

We move the binary point 3 places to the left, giving:

$$-1.00111_2 \times 2^3$$

Step 2: Determine mantissa and exponent:

Mantissa: 1.00111_2

Exponent: 3, with base 2

Step 3: Floating-point representation:

$$-1.00111_2 \times 2^3$$

- **Sign bit (S):** 1 (negative)
- **Exponent (E):** Exponent 3 with bias 127, so $3 + 127 = 130$, and $130_{10} = 10000010_2$.
- **Mantissa (M):** Exclude the leading 1, so the mantissa is 001110.

Final IEEE 754 single-precision (32-bit) representation:

1 10000010 001110000000000000000000

Why is Bias Needed in Floating-Point Representation?

Problem:

The exponent in floating-point numbers can be **negative**, but the exponent field stores only **unsigned binary values** (no sign bit).

Solution:

Use a **bias** to shift the range of exponents so they can be stored as unsigned numbers.

Bias formula:

For an exponent field of n bits:

$$\text{Bias} = 2^{n-1} - 1$$

Example (Single Precision):

- Exponent field = 8 bits \Rightarrow Bias = $2^7 - 1 = 127$
- Actual exponent: $E = 3$
- Stored exponent: $3 + 127 = 130 = 10000010_2$
- If $E = -3$, then stored value = $-3 + 127 = 124 = 01111100_2$

Floating Point Arithmetic

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

There are four basic operations for floating point arithmetic.

- **Exponent Overflow** : When the exponent becomes too large to be represented. Suppose your system supports exponents up to +127 (as in IEEE 754 single-precision).

$$1.5 \times 2^{127} \times 2^2 = 1.5 \times 2^{129}$$

You've exceeded the limit → Exponent Overflow occurs.

- **Exponent Underflow** : When the exponent is too negative to represent in normalized form. In IEEE 754 single-precision, the exponent is stored using 8 bits. This gives a range of exponents from -126 to +127 (after accounting for the bias).

- The "minimum exponent" is -126, and the "maximum exponent" is +127.

$$1.2 \times 2^{-126} \div 2^5 = 1.2 \times 2^{-131}$$

That's beyond the lower bound.

- **Significand Overflow:** When adding significands, the result carries out of the leftmost bit.

$$A = 0.9 \times 2^5 \quad \text{and} \quad B = 0.9 \times 2^5$$

Adding the two:

$$A + B = 1.8 \times 2^5$$

However, the value 1.8 exceeds the normalized range of the significand, which should be in the range: [0.5, 1). **Significand overflow occurs.**

- **Significand Underflow:** When aligning significands, bits fall off the right due to shifting.

$$A = 1.234 \times 2^5 \quad \text{and} \quad B = 1.002 \times 2^{-10}$$

To add them, we align the exponents:

$$B = 1.002 \times 2^{-10} = 0.000000000000976 \times 2^5$$

This is a very tiny number, and most of its bits will be lost when added to A. **Significand underflow (loss of precision).**

Floating Point Addition and Subtraction

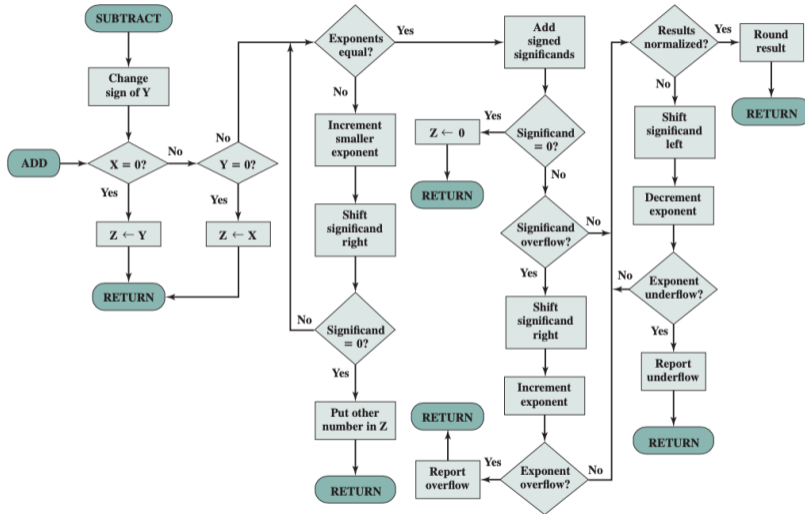


Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

There are four phases for the algorithm for floating point addition and subtraction.

① **Check for Zeros:**

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 subtraction operation. If either number is zero, the result is the other number.

② **Align the Significands:**

Alignment is achieved by shifting the smaller number to the right (increasing the exponent) or shifting the larger number to the left (decreasing the exponent).

③ **Addition or Subtraction of the Significands:**

Once the significands are aligned, they are operated on as required (addition or subtraction depending on the operation).

④ **Normalization of the Result:**

Normalization consists of shifting the significand digits left until the most significant bit is nonzero. This ensures that the result fits the normalized range for the floating-point number.

Example: Addition

$$X = 0.10001 \times 2^{110}, \quad Y = 0.101 \times 2^{100}$$

Since $E_Y < E_X$, adjust Y :

$$Y = 0.00101 \times 2^{100} \times 2^{10} = 0.00101 \times 2^{110}$$

Thus, $E_Z = E_X = E_Y = 110$.

Now, compute the mantissa sum:

$$M_Z = M_X + M_Y = 0.10001 + 0.00101 = 0.10110$$

Hence, the result is:

$$Z = M_Z \times 2^{E_Z} = 0.10110 \times 2^{110}$$

Example: Subtraction

$$X = 0.10001 \times 2^{110}, \quad Y = 0.101 \times 2^{100}$$

Since $E_Y < E_X$, adjust Y :

$$Y = 0.00101 \times 2^{100} \times 2^{10} = 0.00101 \times 2^{110}$$

Thus, $E_Z = E_X = E_Y = 110$.

Now, compute the mantissa difference:

$$M_Z = M_X - M_Y = 0.10001 - 0.00101 = 0.01100$$

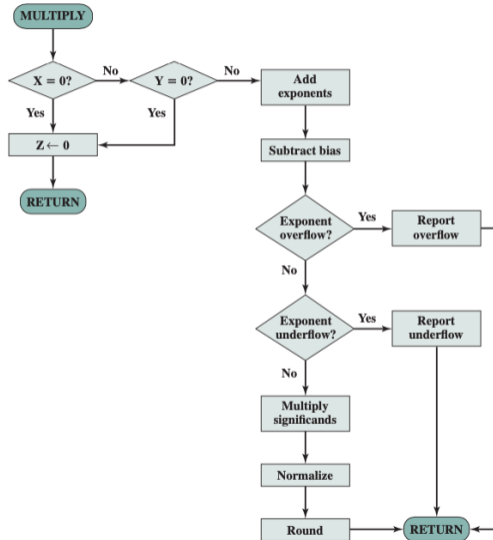
This gives the result:

$$Z = M_Z \times 2^{E_Z} = 0.01100 \times 2^{110}$$

However, this is unnormalized, so we must adjust:

$$Z = 0.1100 \times 2^{110} \times 2^{-1} = 0.1100 \times 2^{101}$$

Floating Point Multiplication



Example:

$$X = 0.101 \times 2^{110}, \quad Y = 0.1001 \times 2^{-010}$$

We know that:

$$Z = X \times Y = (M_X \times M_Y) \times 2^{E_X + E_Y}$$

$$Z = (0.101 \times 0.1001) \times 2^{(110 + (-010))} = 0.0101101 \times 2^{100}$$

Normalization:

$$Z = 0.101101 \times 2^{011}$$

$$\begin{array}{r} 1011.01 \\ \times \quad 110.1 \\ \hline 101101 \\ 0000000 \\ 10110100 \\ 101101000 \\ \hline 1001001.001 \end{array}$$

Why subtract bias in multiply in flowchart?

When multiplying two floating-point numbers:

- Each exponent is stored in **biased form**.
- Let E_x and E_y be the biased exponents of X and Y .
- Actual exponents: $e_x = E_x - \text{bias}$, $e_y = E_y - \text{bias}$
- When multiplying:

$$e_z = e_x + e_y$$

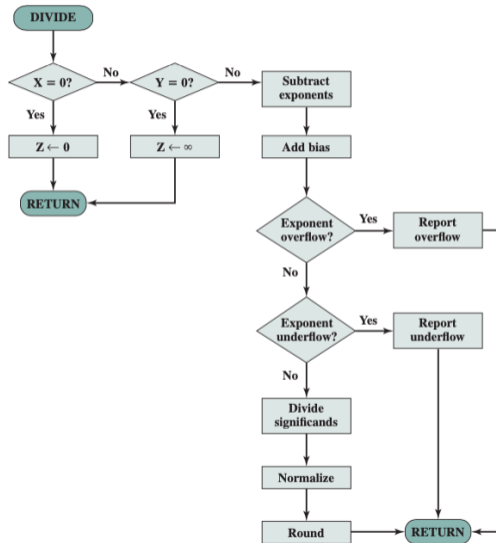
- But biased exponents are added directly:

$$E'_z = E_x + E_y = (e_x + \text{bias}) + (e_y + \text{bias}) = e_x + e_y + 2 \times \text{bias}$$

- To get the correctly biased result:

$$E_z = E_x + E_y - \text{bias}$$

Floating Point Division



Given:

$$X = 0.101 \times 2^{110}, \quad Y = 0.1001 \times 2^{-010}$$

We know:

$$Z = X \div Y = \left(\frac{M_X}{M_Y} \right) \times 2^{E_X - E_Y}$$

Compute the mantissa division:

$$\frac{0.101}{0.1001} = \frac{1/2 + 1/8}{1/2 + 1/16} = \frac{0.625}{0.5625} = 1.1111_{10} = 1.00011_2$$

Binary Fraction Expansion (detail):

$$0.1111 \times 2 = 0.2222 \Rightarrow 0 \quad (\text{keep } 0, \text{ frac: } 0.2222)$$

$$0.2222 \times 2 = 0.4444 \Rightarrow 0 \quad (\text{keep } 0, \text{ frac: } 0.4444)$$

$$0.4444 \times 2 = 0.8888 \Rightarrow 0 \quad (\text{keep } 0, \text{ frac: } 0.8888)$$

$$0.8888 \times 2 = 1.7776 \Rightarrow 1 \quad (\text{keep } 1, \text{ frac: } 0.7776)$$

$$0.7776 \times 2 = 1.5552 \Rightarrow 1 \quad (\text{fraction becomes } 0.00011\dots)$$

Exponent Difference:

$$E_X - E_Y = 110 - (-010) = 110 + 010 = 1000$$

So the result is:

$$Z = 1.00011 \times 2^{1000} = 0.100011 \times 2^{1001} \quad (\text{Normalized})$$

Why add bias in division in flowchart?

When dividing two floating-point numbers:

- Each exponent is stored in **biased form**.
- Let E_x and E_y be the biased exponents of X and Y .
- Actual exponents: $e_x = E_x - \text{bias}$, $e_y = E_y - \text{bias}$
- When dividing:

$$e_z = e_x - e_y$$

- Using biased exponents:

$$E'_z = E_x - E_y = (e_x + \text{bias}) - (e_y + \text{bias}) = e_x - e_y$$

- The result e_z is unbiased. To store it in IEEE 754 format, we must **add the bias back**:

$$E_z = e_z + \text{bias}$$

Assignments

- 1 Explain addition and subtraction algorithm for addition and subtraction of two numbers.[4]
- 2 Explain Booth's algorithm with the help of flowchart.[4]
- 3 Draw the flow chart for restoring and non restoring division [5+5]
- 4 Perform $19 * -14$ using Booth's algorithm.
- 5 Perform $19/14$ using restoring and non restoring division [5+5]
- 6 Explain the procedure of floating point addition and subtraction with the help of flowchart and example.[8]
- 7 Explain the procedure of floating point multiplication with the help of flowchart and example. [4]
- 8 Explain the procedure of floating point division with the help of flowchart and example. [4]
- 9 Why bias is subtracted in floating point multiplication and added in division? [4]