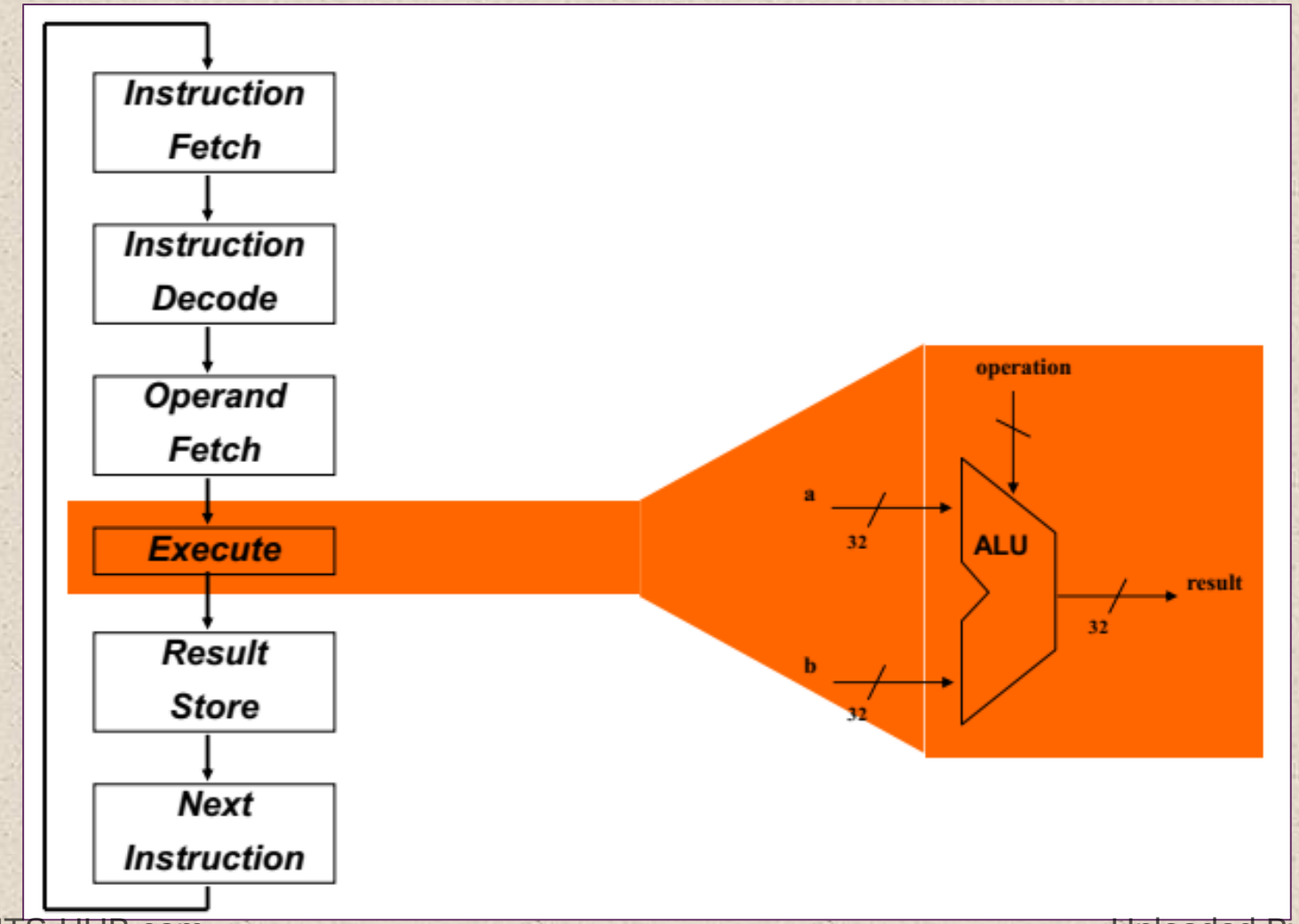
Chapter 10

Computer Arithmetic

Arithmetic - The heart of instruction execution



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
- Does the calculations
- Handles integers
- May handle floating point (real) numbers
- May be separate FPU (maths coprocessor)
- May be on chip separate FPU (486DX +)

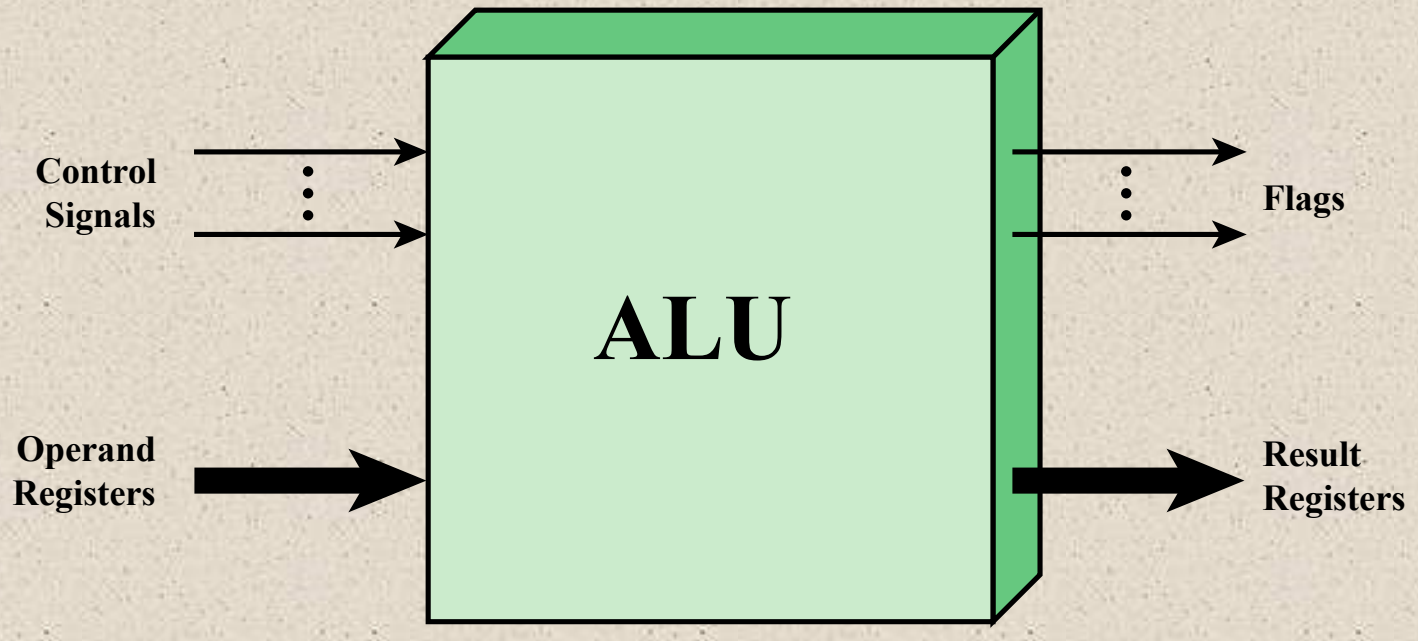


Figure 10.1 ALU Inputs and Outputs

Positional Number Systems

- Different Representations of Natural Numbers
  - XXVII Roman numerals (not positional)
  - 27 Radix-10 or decimal number (positional)
  - 11011_2 Radix-2 or binary number (also positional)
- Fixed-radix positional representation with k digits

Number N in radix $r = (d_{k-1}d_{k-2} \ldots d_1d_0)_r$

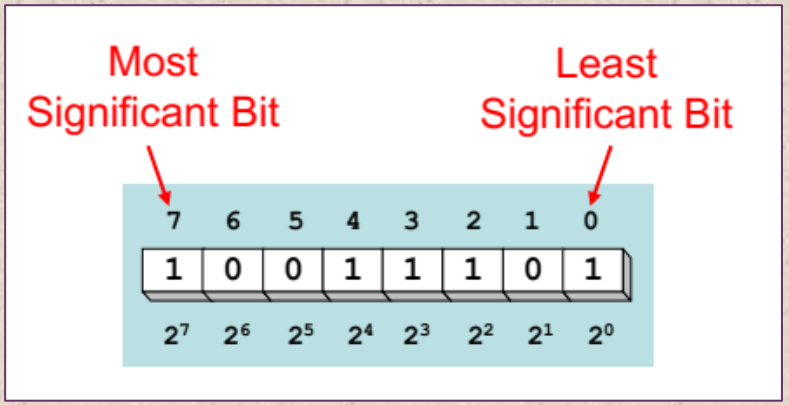
Value = $d_{k-1} \times r^{k-1} + d_{k-2} \times r^{k-2} + \ldots + d_1 \times r + d_0$

Examples: $(11011)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1 = 27$

$(2103)_4 = 2 \times 4^3 + 1 \times 4^2 + 0 \times 4 + 3 = 147$

Binary Numbers

- Each binary digit (called bit) is either 1 or 0
- Bits have no inherent meaning, can represent
  - Unsigned and signed integers
  - Characters
  - Floating-point numbers
  - Images, sound, etc.



- Bit Numbering
  - Least significant bit (LSB) is rightmost (bit 0)
  - Most significant bit (MSB) is leftmost (bit 7 in a 8-bit number)

Hexadecimal Integers

- 16 Hexadecimal Digits: 0 – 9, A – F
- More convenient to use than binary numbers

Binary, Decimal, and Hexadecimal Equivalents

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Converting Binary to Hexadecimal

- Each hexadecimal digit corresponds to 4 binary bits

Example:

Convert the 32-bit binary number to hexadecimal

1110 1011 0001 0110 1010 0111 1001 0100

Solution:

E	B	1	6	A	7	9	4
1110	1011	0001	0110	1010	0111	1001	0100

Integer Storage Sizes

Storage Type	Unsigned Range	Powers of 2
Byte	0 to 255	0 to $(2^8 - 1)$
Half Word	0 to 65,535	0 to $(2^{16} - 1)$
Word	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Double Word	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

What is the largest 20-bit unsigned integer?

Answer: $2^{20} - 1 = 1{,}048{,}575$

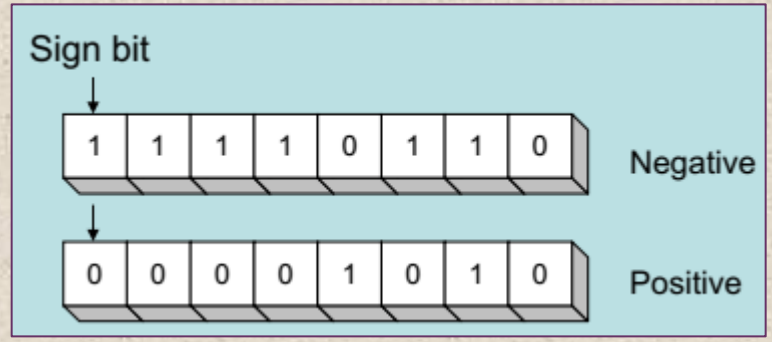
Signed Integers

- Several ways to represent a signed number
  - Sign-Magnitude
  - Biased
  - 1's complement
  - 2's complement
- Divide the range of values into 2 equal parts
  - First part corresponds to the positive numbers (≥ 0)
  - Second part correspond to the negative numbers (< 0)
- Focus will be on the 2's complement representation
  - Has many advantages over other representations
  - Used widely in processors to represent signed integers

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

Sign Bit

- Highest bit indicates the sign
- 1 = negative
- 0 = positive



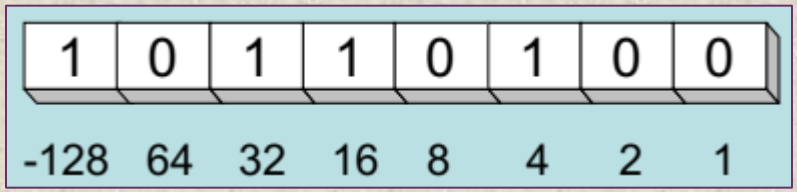
- For Hexadecimal Numbers, check most significant digit
  - If highest digit is > 7, then value is negative
- Examples: 8A and C5 are negative bytes
- B1C42A00 is a negative word (32-bit signed integer)
- Problems
  - Need to consider both sign and magnitude in arithmetic
  - Two representations of zero (+0 and -0) in sign-mag & 1's comp

Biased Representation

- Other type of binary number representations
- A fixed value called Bias is added for the binary value
- Typically, the bias equals $(2^{k-1}-1)$, where K is the number of bits in the binary number.
- e.g for 4 bit representation,
  - The bias value= $2^{4-1}-1= 7$
  - Representation of +8 => 8+7=15 => 1111
  - Representation of -7 => -7+7=0 => 0000

Two's Complement Representation

- Positive numbers
  - Signed value = Unsigned value
- Negative numbers
  - Signed value = Unsigned value – 2^n
  - n = number of bits
- Negative weight for MSB
- Another way to obtain the signed value is to assign a negative weight to most-significant bit



- = -128 + 32 + 16 + 4 = -76

8-bit Binary value	Unsigned value	Signed value
00000000	0	0
00000001	1	+1
00000010	2	+2
. . .	. . .	. . .
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
. . .	. . .	. . .
11111110	254	-2
11111111	255	-1

Forming the Two's Complement

starting value	00100100 = +36
step1: reverse the bits (1's complement)	11011011
step 2: add 1 to the value from step 1	+ 1
sum = 2's complement representation	11011100 = -36

Sum of an integer and its 2's complement must be zero:

00100100 + 11011100 = 00000000 (8-bit sum) ⇒ Ignore Carry

Another way to obtain the 2's complement:

Start at the least significant 1

Leave all the 0s to its right unchanged

Complement all the bits to its left

Binary Value

= 00100**1**00 (least significant 1)

2's Complement

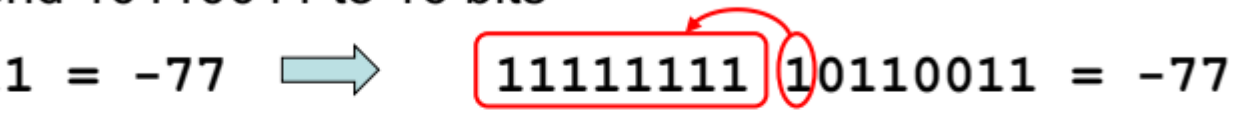
= 11011**1**00

Sign Extension

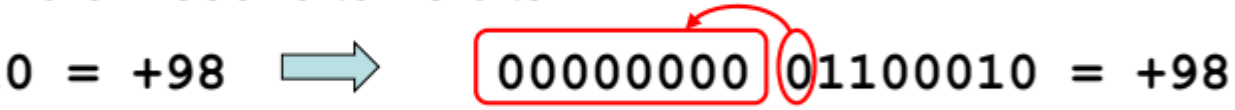
- **Step 1**: Move the number into the lower-significant bits
- **Step 2**: Fill all the remaining higher bits with the sign bit
- This will ensure that both magnitude and sign are correct

Examples

- Sign-Extend 10110011 to 16 bits



- Sign-Extend 01100010 to 16 bits



Infinite 0s can be added to the left of a positive number

Infinite 1s can be added to the left of a negative number

Ranges of Signed Integers

For n-bit signed integers: Range is -2^{n-1} to $(2^{n-1} - 1)$

Positive range: 0 to $2^{n-1} - 1$

Negative range: -2^{n-1} to -1

Storage Type	Signed Range	Powers of 2
Byte	–128 to +127	-2^7 to $(2^7 - 1)$
Half Word	–32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Word	–2,147,483,648 to +2,147,483,647	-2^{31} to $(2^{31} - 1)$
Double Word	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Practice: What is the range of signed values that may be stored in 20 bits?

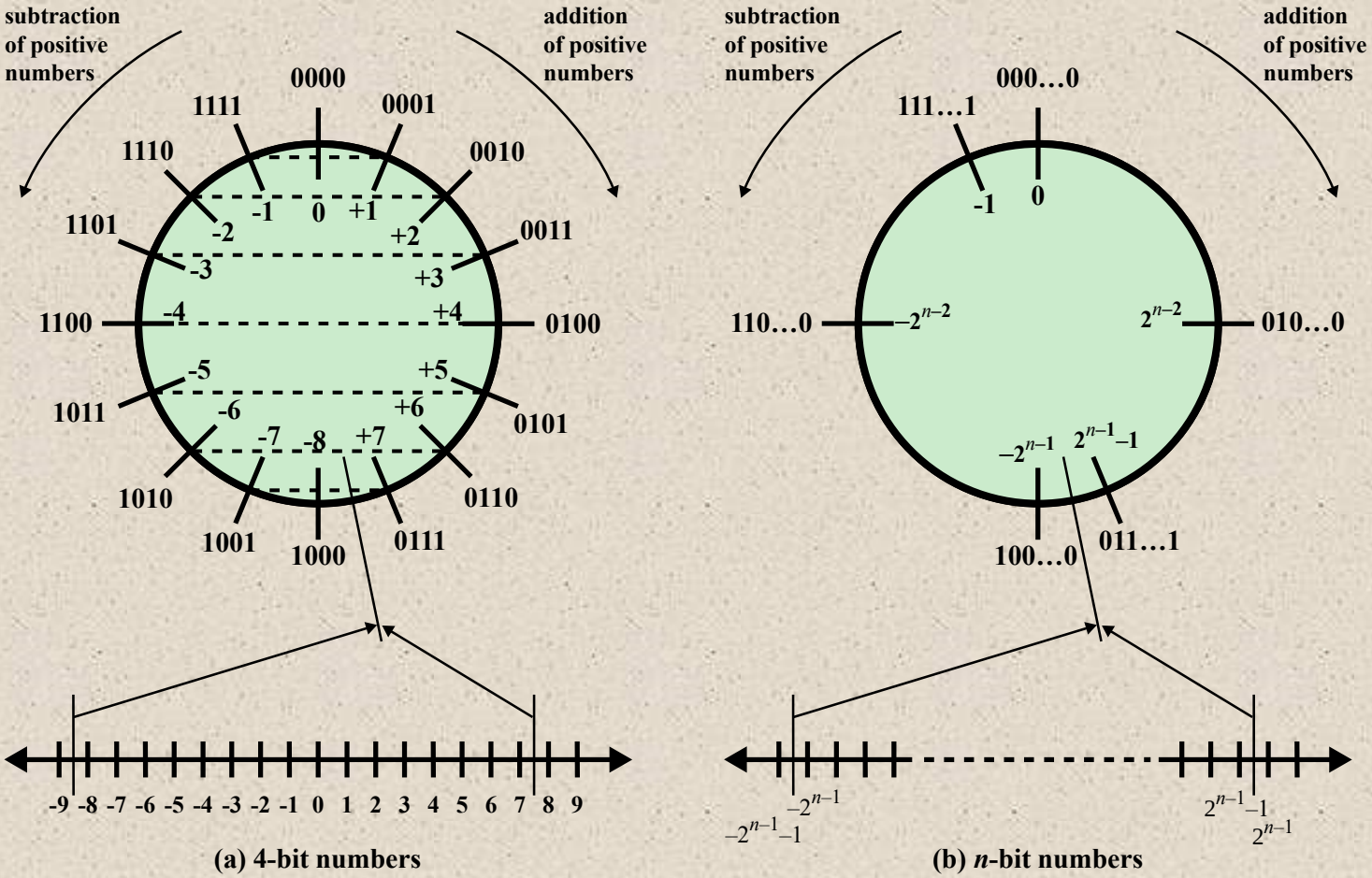


Figure 10.5 Geometric Depiction of Twos Complement Integers

Negation

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

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

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

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

Negation Special Case 1

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

Overflow is ignored, so:

- 0 = 0

Negation Special Case 2

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

So:

-(-128) = -128 X

Monitor MSB (sign bit)

It should change during negation

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

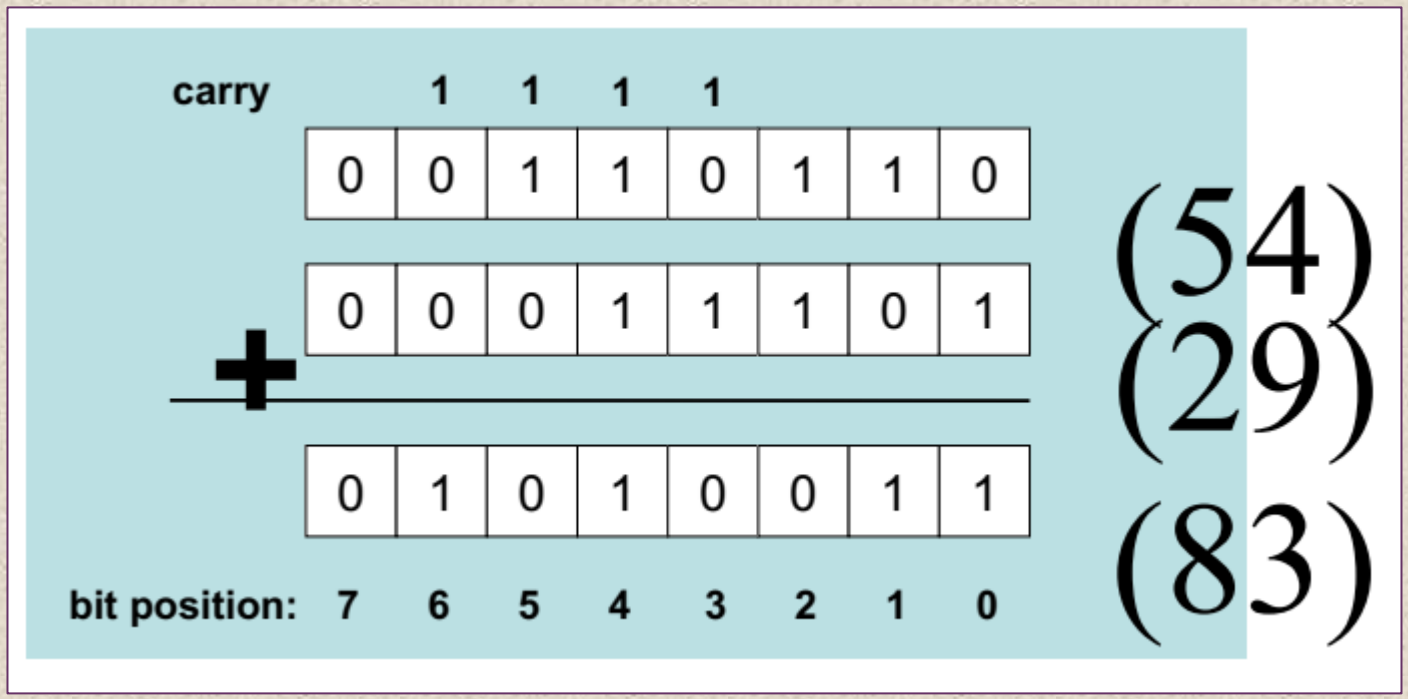
Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and 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.

Character Storage

- Character sets
  - Standard ASCII: 7-bit character codes (0 – 127)
  - Extended ASCII: 8-bit character codes (0 – 255)
  - Unicode: 16-bit character codes (0 – 65,535)
  - Unicode standard represents a universal character set
    - Defines codes for characters used in all major languages
    - Used in Windows-XP: each character is encoded as 16 bits
  - UTF-8: variable-length encoding used in HTML
    - Encodes all Unicode characters
    - Uses 1 byte for ASCII, but multiple bytes for other characters
- Null-terminated String
  - Array of characters followed by a NULL character

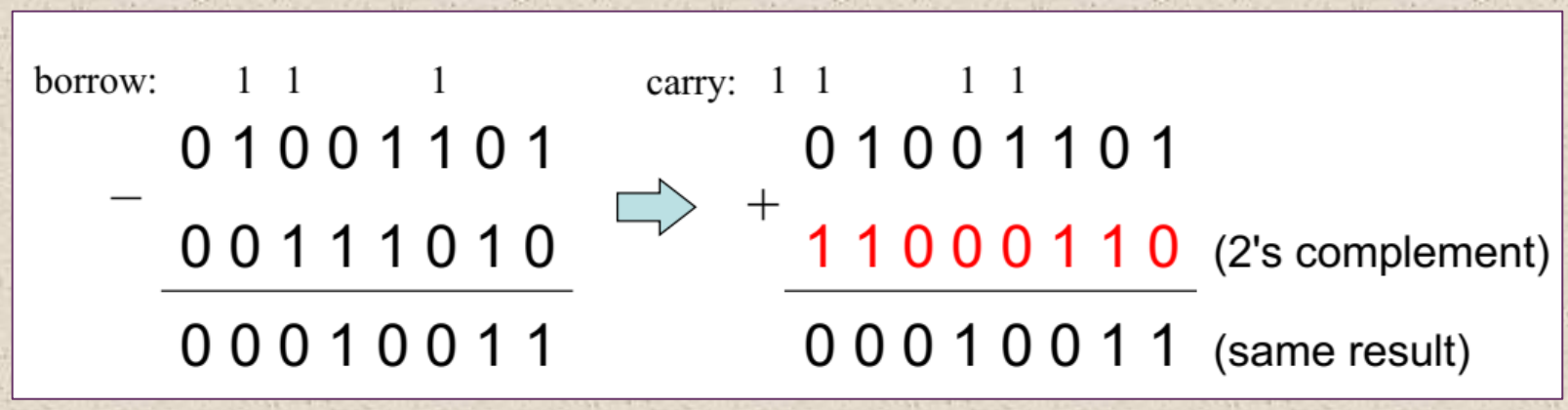
Binary Addition

- Start with the least significant bit (rightmost bit)
- Add each pair of bits
- Include the carry in the addition, if present



Binary Subtraction

- When subtracting A – B, convert B to its 2's complement
- Add A to (–B)



- Final carry is ignored, because
  - Negative number is sign-extended with 1's
  - You can imagine infinite 1's to the left of a negative number
  - Adding the carry to the extended 1's produces extended zeros

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.

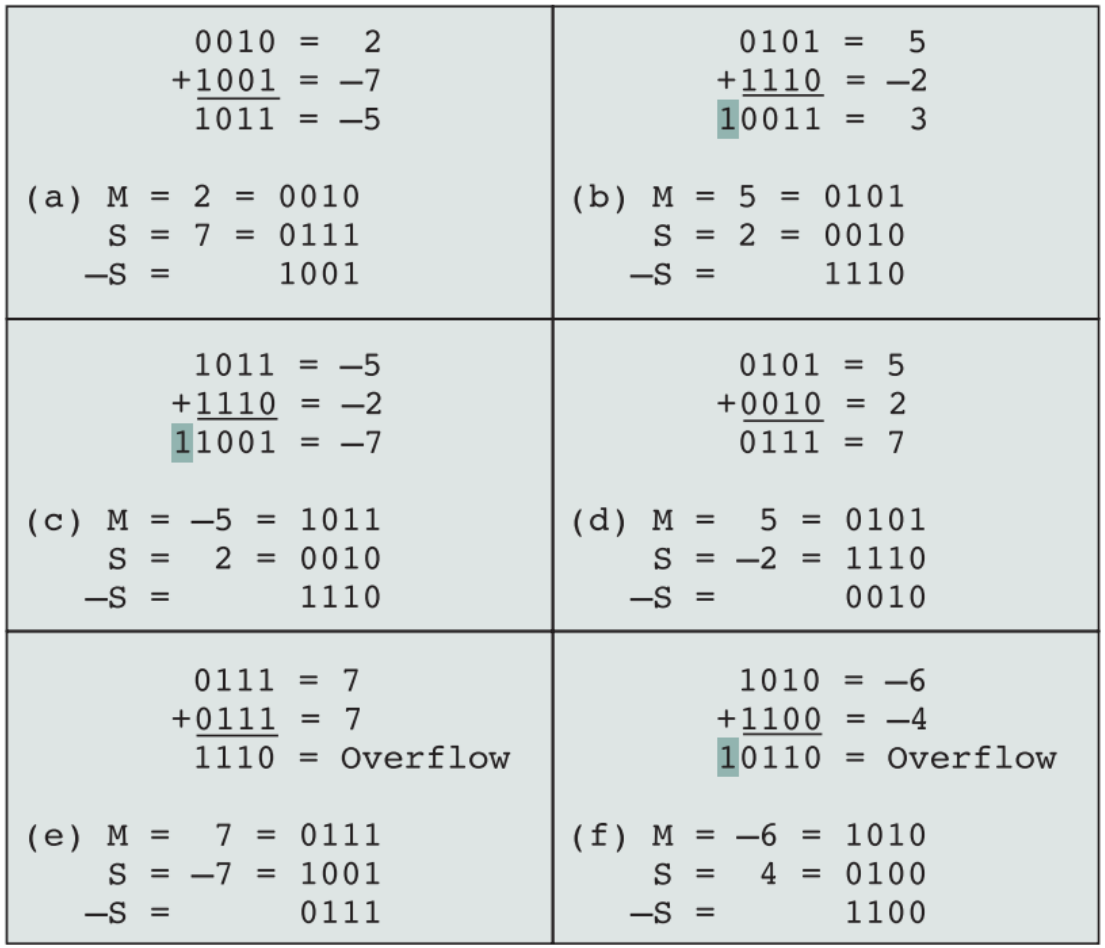
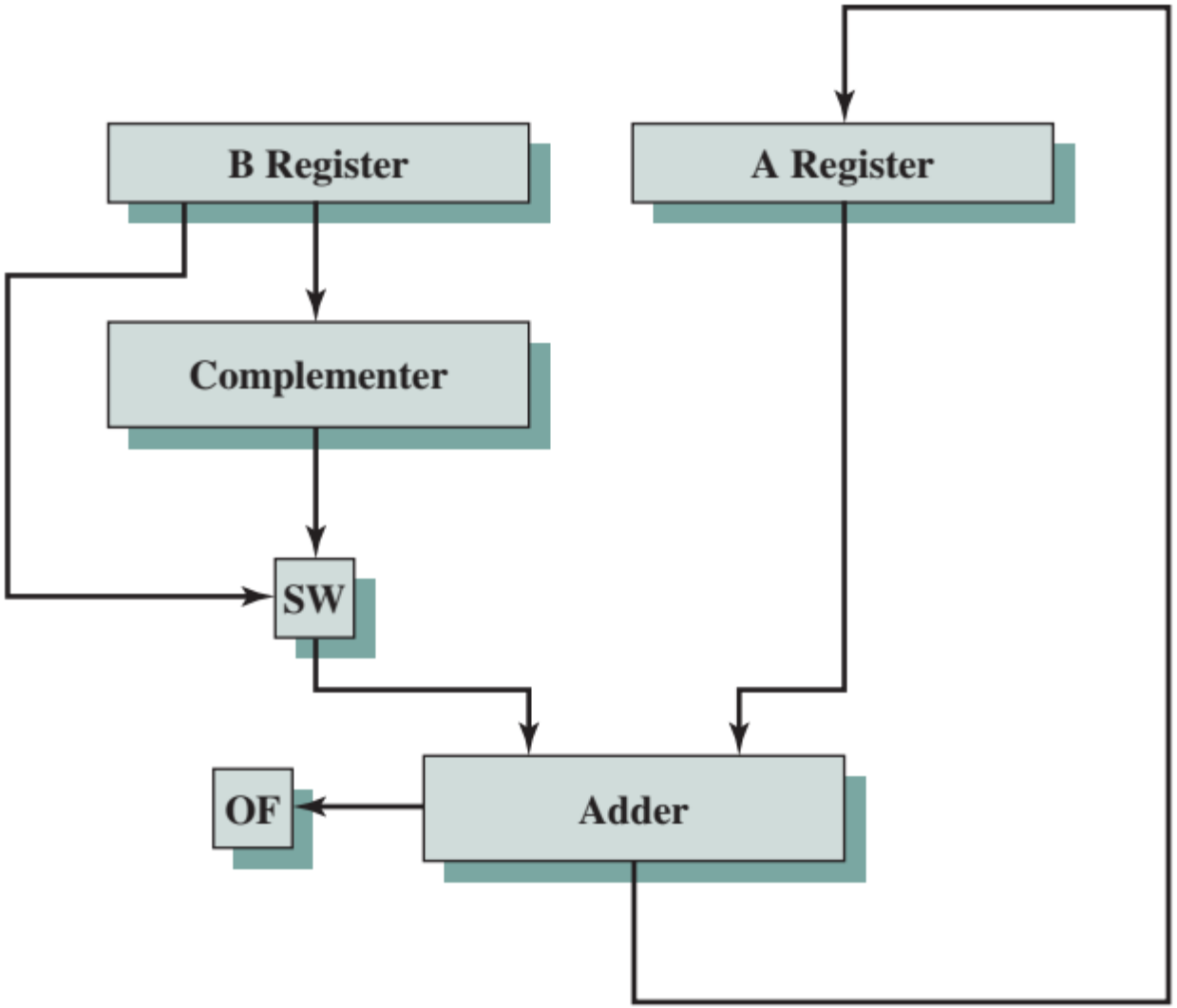


Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M − S)

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



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

Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

Unsigned Multiplication

Paper and Pencil Example:

Multiplicand	1100_2	= 12
Multiplier	× 1101_2	= 13
	1100	
	0000	
	1100	
	1100	
Product	10011100_2	= 156

Binary multiplication is easy
0 × multiplicand = 0
1 × multiplicand = multiplicand

- m-bit multiplicand × n-bit multiplier = (m+n)-bit product
- Accomplished via shifting and addition
- Consumes more time and more chip area

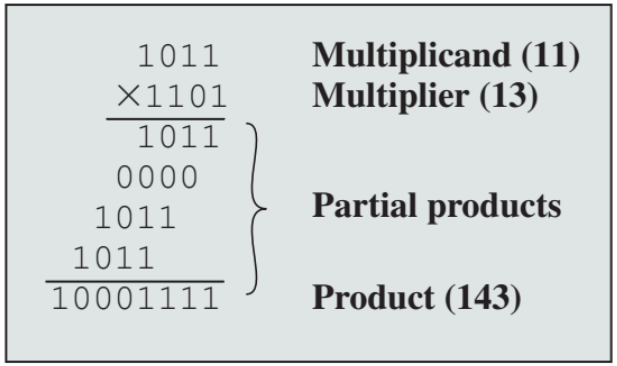


Figure 10.7 Multiplication of Unsigned Binary Integers

Uploaded By: anonymous

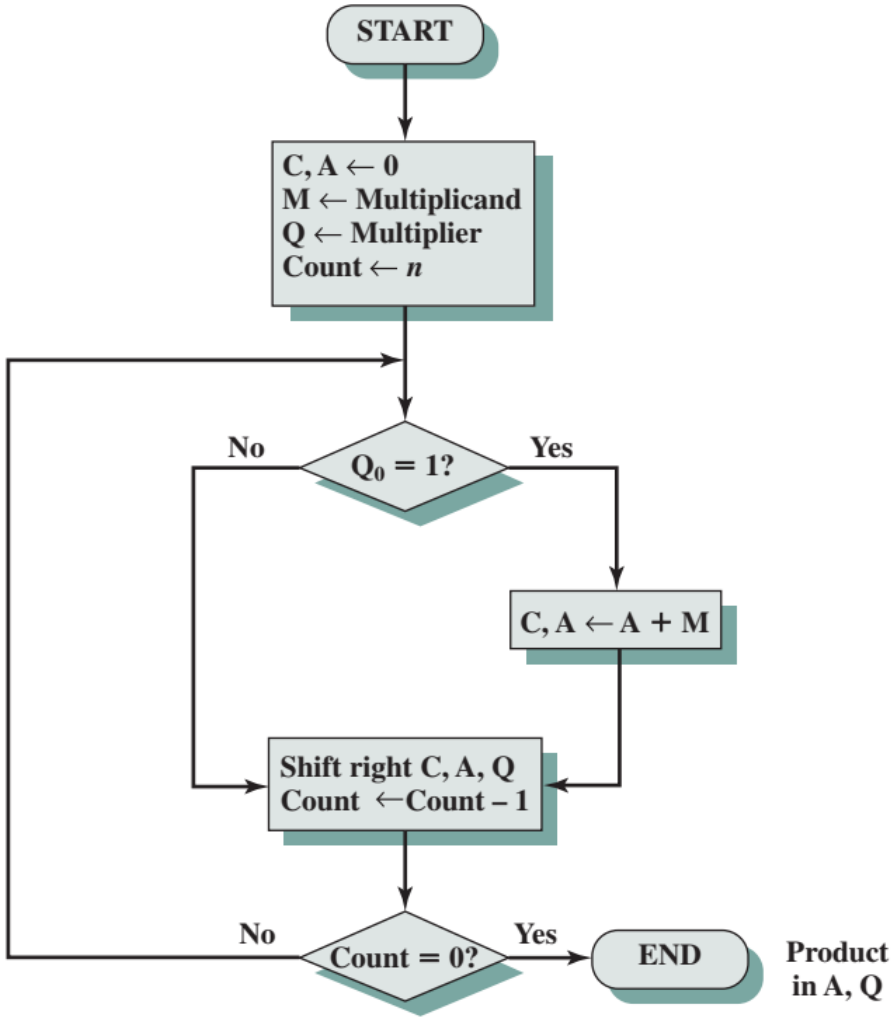
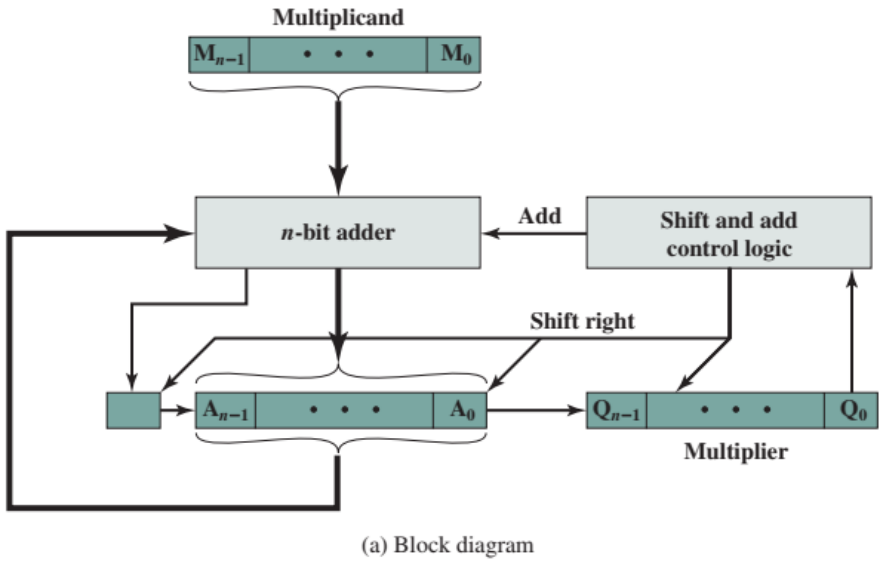


Figure 10.9 Flowchart for Unsigned Binary Multiplication



(a) Block diagram

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

(b) Example from Figure 10.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

Multiplying Negative Numbers

- This does not work!
- **Solution 1**
  - Convert to positive if required
  - Multiply as above
  - If signs were different, negate answer
- **Solution 2**
  - Booth's algorithm

Booth's algorithm

https://www.grahn.us/projects/booths-algorithm/

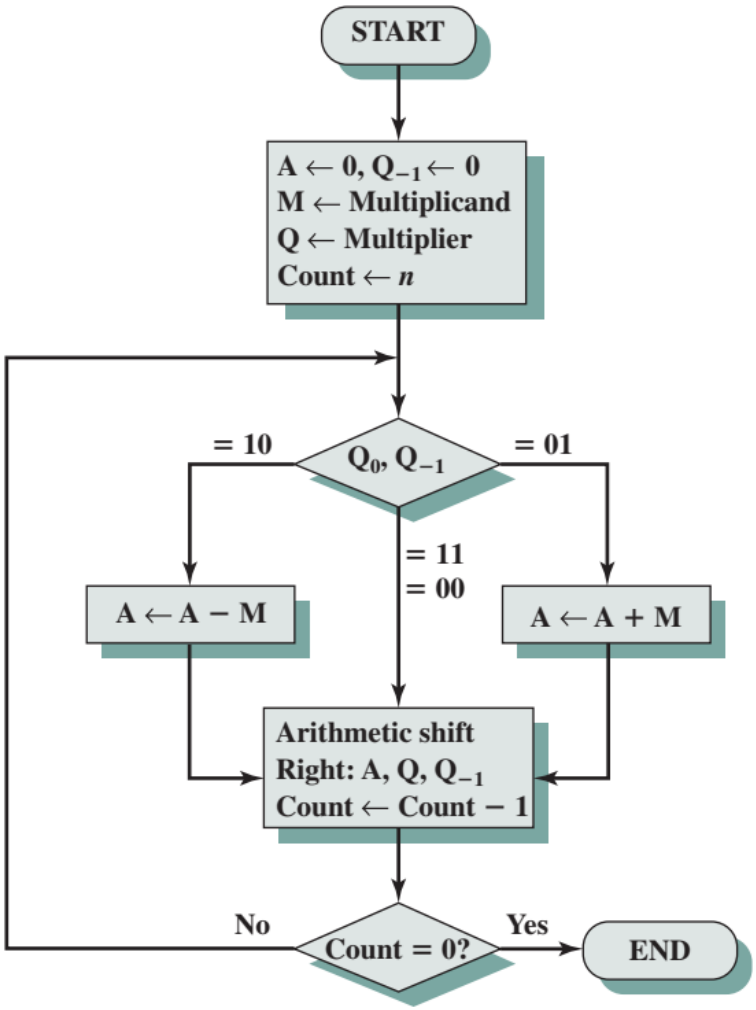


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

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

How it works

- Consider a positive multiplier consisting of a block of 1s surrounded by 0s. For example, 00111110. The product is given by :

$$M \times ''0\,0\,1\,1\,1\,1\,1\,0'' = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$$

- where M is the multiplicand.
- The number of operations can be reduced to two by rewriting the same as

$$M \times ''0\,1\,0\,0\,0\,0\,0\,0\text{-}1\,0'' = M \times (2^6 - 2^1) = M \times 62$$

- Note that:

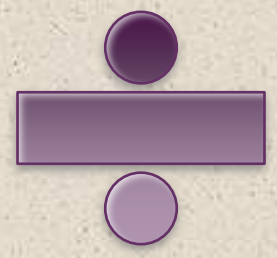
$$2^n + 2^{n-1} + \ldots + 2^{n-k} = 2^{n+1} - 2^{n-k}$$

How it works

- So, the product can be generated by one addition and one subtraction
- In Booth's algorithm
  - perform subtraction when the first 1 of the block is encountered (1 - 0)
  - perform addition when the last 1 of the block is encountered (0 - 1)
- (1 - 0) and (0 - 1) are observed from $Q_0 – Q_{-1}$ (see previous example)

Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division



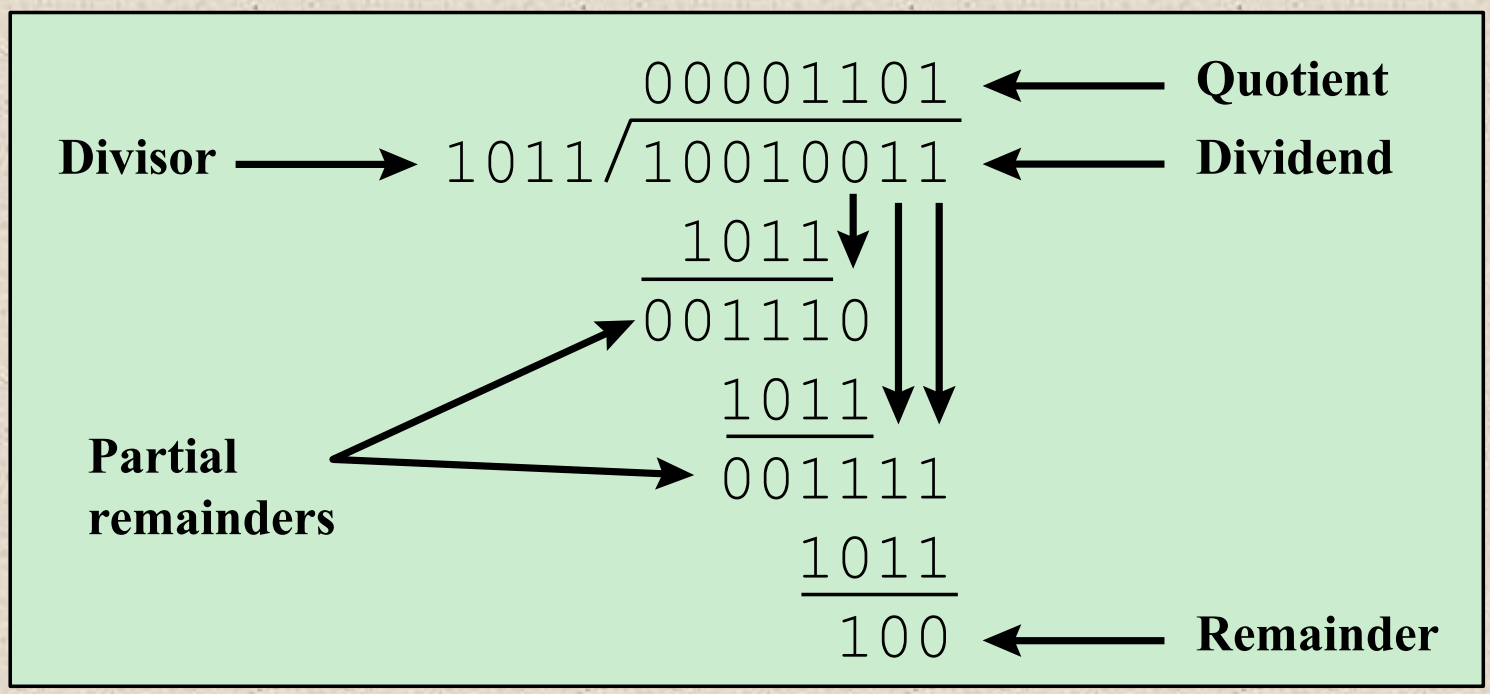


Figure 10.15 Example of Division of Unsigned Binary Integers

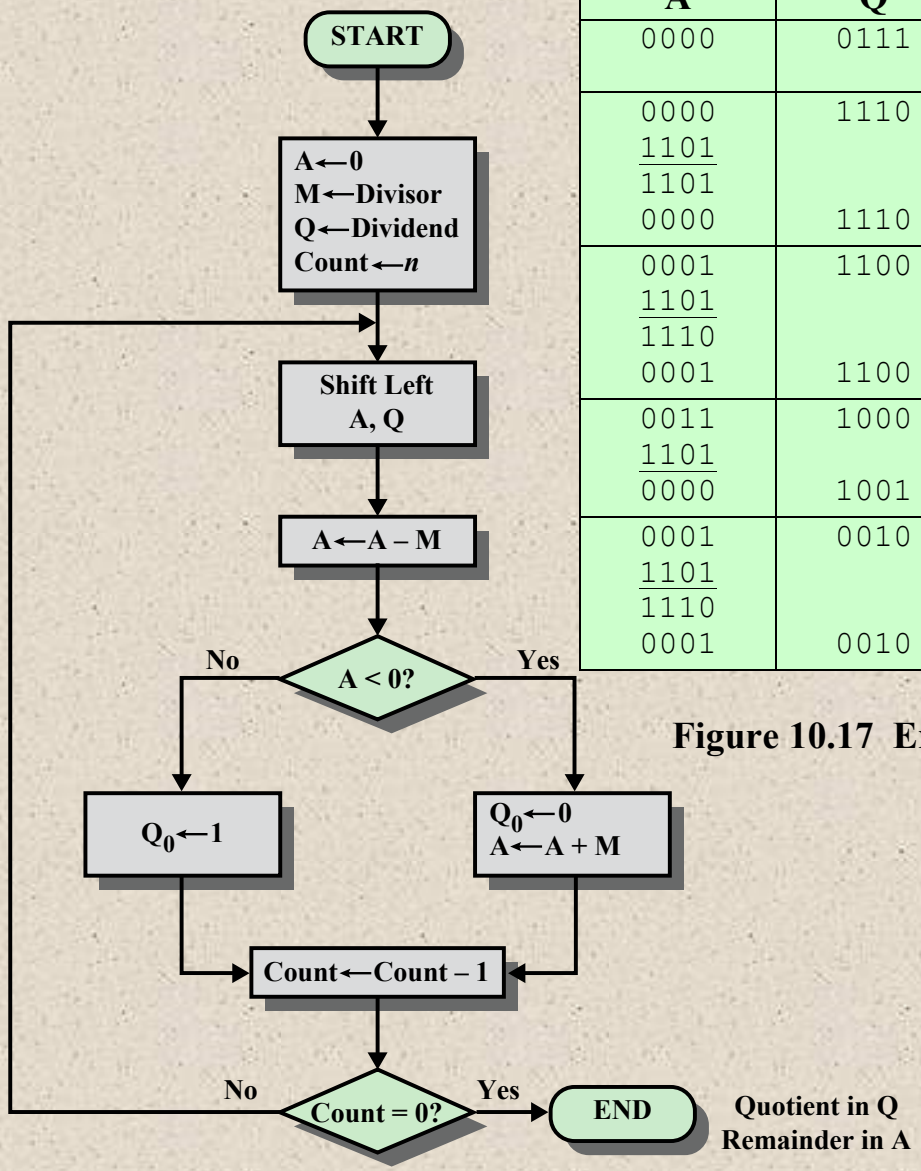


Figure 10.16 Flowchart for Unsigned Binary Division

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

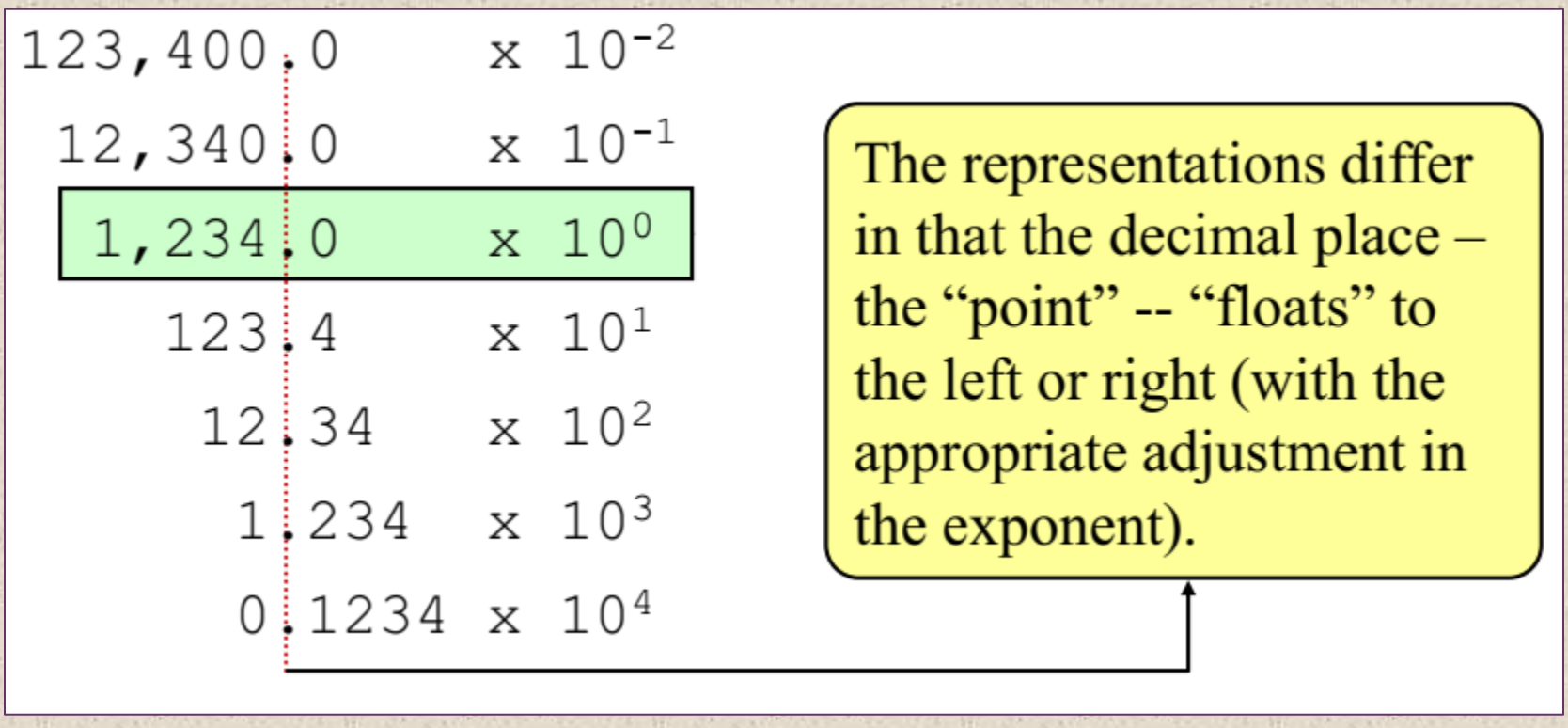
Figure 10.17 Example of Restoring Twos Complement Division (7/3)

Real Numbers

- Numbers with fractions
- Could be done in pure binary
  - $1001.1010 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625$
- Where is the binary point?
- Fixed?
  - Very limited
- Moving?
  - How do you show where it is?

Exponential Notation

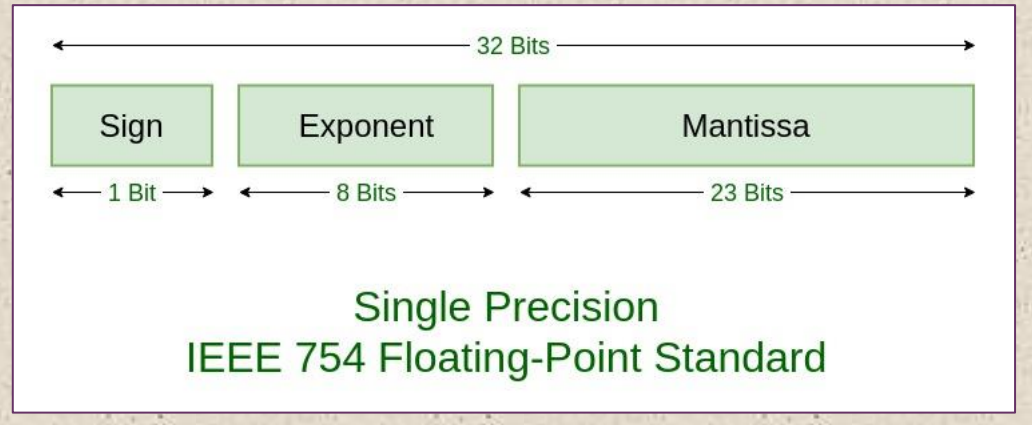
- The following are equivalent representations of 1,234

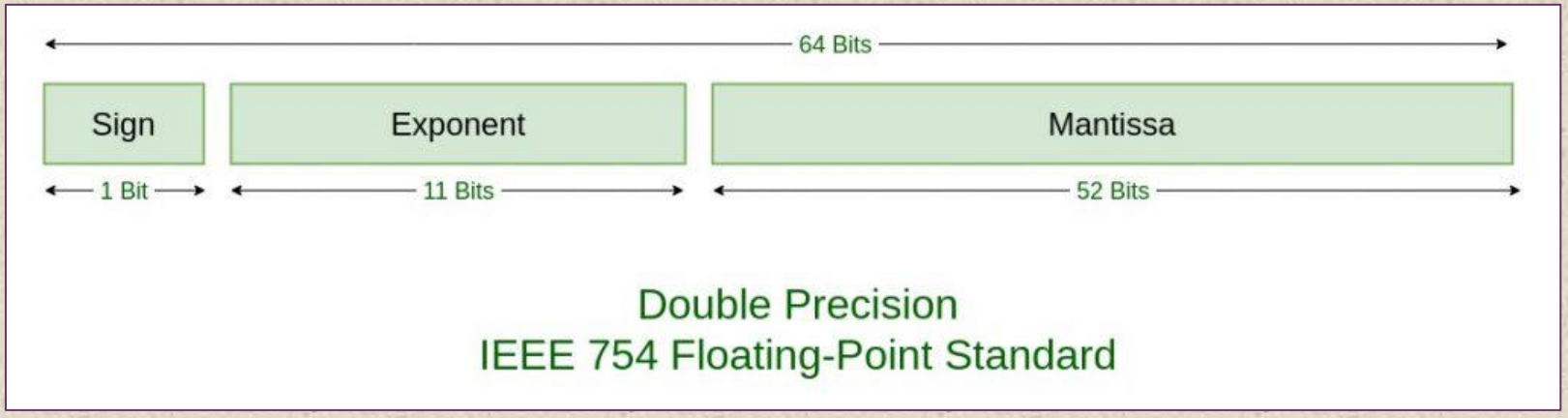


Floating Point

- An IEEE Std 754 floating point representation consists of
  - A Sign Bit (no surprise)
  - An Exponent ("times 2 to the what?")
  - Mantissa ("Significand"), which is assumed to be 1.xxxxx (thus, one bit of the mantissa is implied as 1)
  - This is called a normalized representation
- So a mantissa = 0 really is interpreted to be 1.0, and a mantissa of all 1111 is interpreted to be 1.1111
- Special cases are used to represent denormalized mantissas (true mantissa = 0), NaN, etc.

Representation Format





TYPES	SIGN	BIASED EXPONENT	NORMALISED MANTISA	BIAS
Single precision	1(31st bit)	8(30-23)	23(22-0)	127
Double precision	1(63rd bit)	11(62-52)	52(51-0)	1023

Example

85.125
85 = 1010101
0.125 = 001
85.125 = 1010101.001
=1.010101001 x 2^6
sign = 0

1. Single precision:
biased exponent 127+6=133
133 = 10000101
Normalised mantisa = 010101001
we will add 0's to complete the 23 bits

The IEEE 754 Single precision is:
= 0 10000101 01010100100000000000000
This can be written in hexadecimal form **42AA4000**

2. Double precision:
biased exponent 1023+6=1029
1029 = 10000000101
Normalised mantisa = 010101001
we will add 0's to complete the 52 bits

The IEEE 754 Double precision is:
= 0 10000000101 0101010010000000000000000000000000000000000000000000
This can be written in hexadecimal form **4055480000000000**

Example

- What number is represented by the single precision float 11000000101000…00

11000000101000…00

- S = 1
- Fraction = $01000…00_2$
- Fxponent = $10000001_2 = 129$

$x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

$= (-1) \times 1.25 \times 2^2$

$= -5.0$

Example

Represent –0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- S = 1
- Fraction = $1000...00_2$
- Exponent = –1 + Bias
  - Single: –1 + 127 = 126 = 01111110_2
  - Double: –1 + 1023 = 1022 = 01111111110_2

Single: 10111111010000...00

Double: 10111111111010000...00

Converting from Floating Point

- E.g., What decimal value is represented by the following 32-bit floating point number?
  - $C17B0000_{16}$

Step 1

—Express in binary and find S, E, and M

$C17B0000_{16}$ =

$1\ 10000010\ 11110110000000000000000_2$

S E M

1 = negative
0 = positive

Step 2

—Find "real" exponent, *n*

—n = E – 127

= 10000010_2 – 127

= 130 – 127

= 3

Converting from Floating Point

- E.g., What decimal value is represented by the following 32-bit floating point number?
  - $C17B0000_{16}$

Step 3

—Put S, M, and *n* together to form binary result

—(Don't forget the implied "1." on the left of the mantissa.)

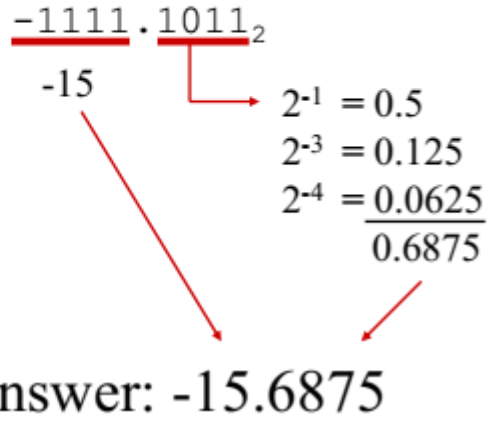
$-1.1111011_2 \times 2^n =$

$-1.1111011_2 \times 2^3 =$

-1111.1011_2

Step 4

—Express result in decimal



Converting to Floating Point

- E.g., Express 36.5625_{10} as a 32-bit floating point number (in hexadecimal)

Step 1

—Express original value in binary

$36.5625_{10} =$

100100.1001_2

Step 2

—Normalize

$100100.1001_2 =$

$1.001001001_2 \times 2^5$

Step 3

—Determine S, E, and M

$+1.001001001_2 \times 2^5$

S: M: n

$E = n + 127$
$= 5 + 127$
$= 132$
$= 10000100_2$

S = 0 (because the value is positive)

Step 4

—Put S, E, and M together to form 32-bit binary result

$0\ 10000100\ 00100100100000000000000_2$

S E M

Converting to Floating Point

- E.g., Express 36.5625_{10} as a 32-bit floating point number (in hexadecimal)

Step 5

—Express in hexadecimal

$0\ 10000100\ 00100100100000000000000_2 =$

$0100\ 0010\ 0001\ 0010\ 0100\ 0000\ 0000\ 0000_2 =$

$4\quad 2\quad 1\quad 2\quad 4\quad 0\quad 0\quad 0_{16}$

Answer: 42124000_{16}

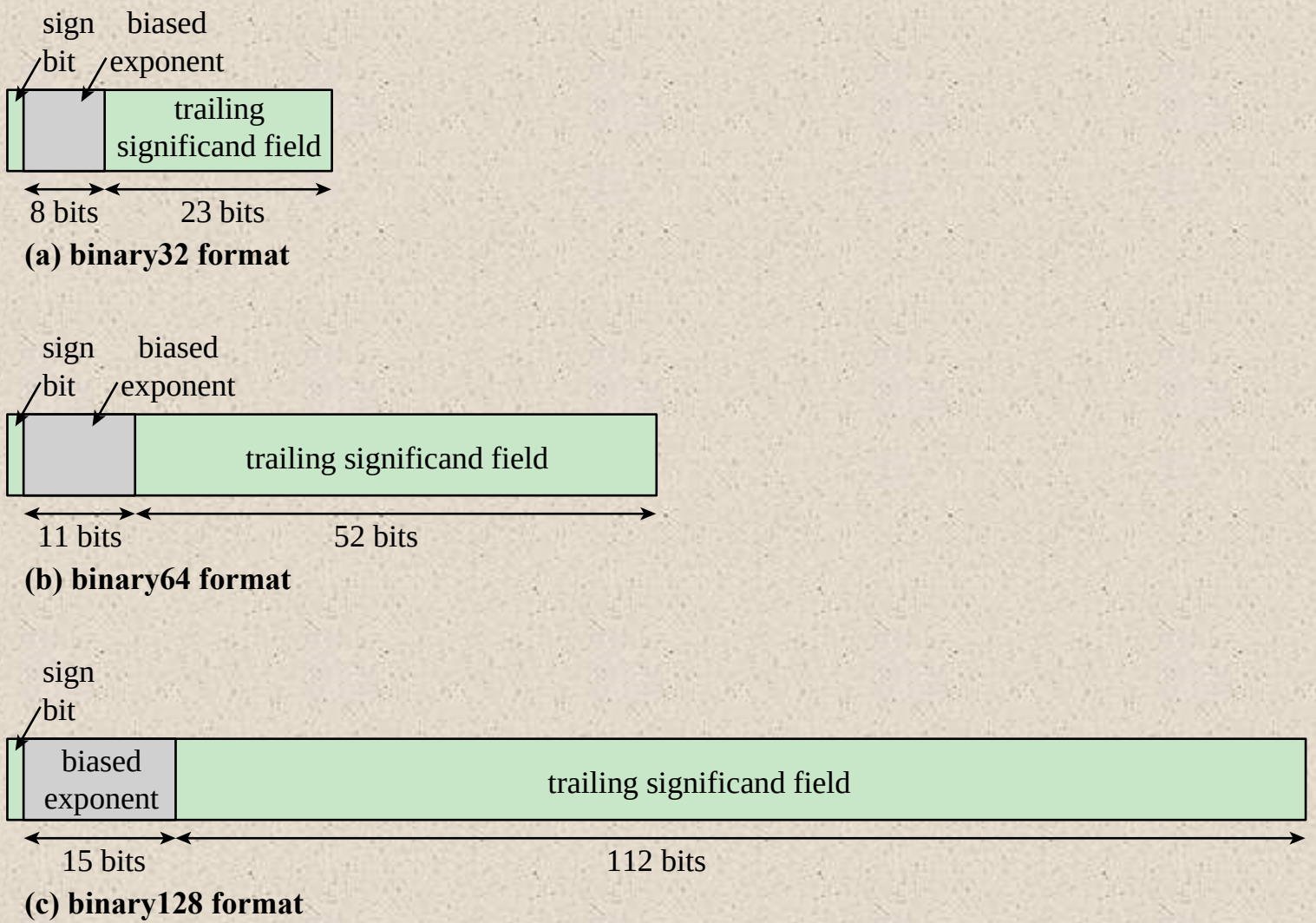


Figure 10.21 IEEE 754 Formats

FP Arithmetic +/-

- Check for zeros
- Align significands (adjusting exponents)
- Add or subtract significands
- Normalize result

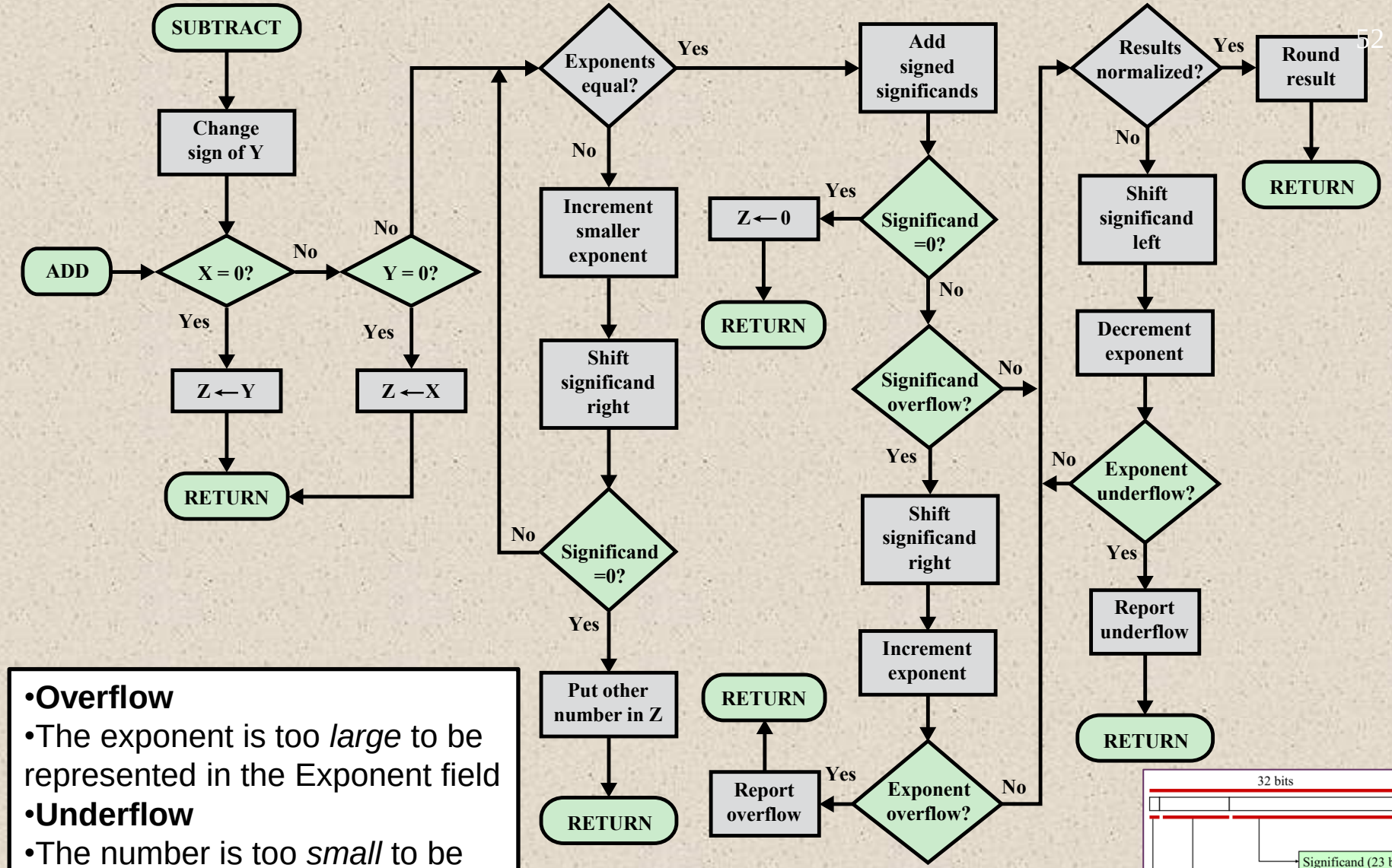


Figure 10.22 Floating-Point Addition and Subtraction (Z← X ± Y)

Example

Perform $0.5 + (-0.4375)$

$0.5 = 0.1 \times 2^{0} = 1.000 \times 2^{-1}$ (normalised)

$-0.4375 = -0.0111 \times 2^{0} = -1.110 \times 2^{-2}$ (normalised)

1. Rewrite the smaller number such that its exponent matches with the exponent of the larger number.

$-1.110 \times 2^{-2} = -0.1110 \times 2^{-1}$

2. Add the mantissas:

$1.000 \times 2^{-1} + -0.1110 \times 2^{-1} = 0.001 \times 2^{-1}$

3. Normalise the sum, checking for overflow/underflow:

$0.001 \times 2^{-1} = 1.000 \times 2^{-4}$

$-126 <= -4 <= 127$ ===> No overflow or underflow

4. Round the sum:

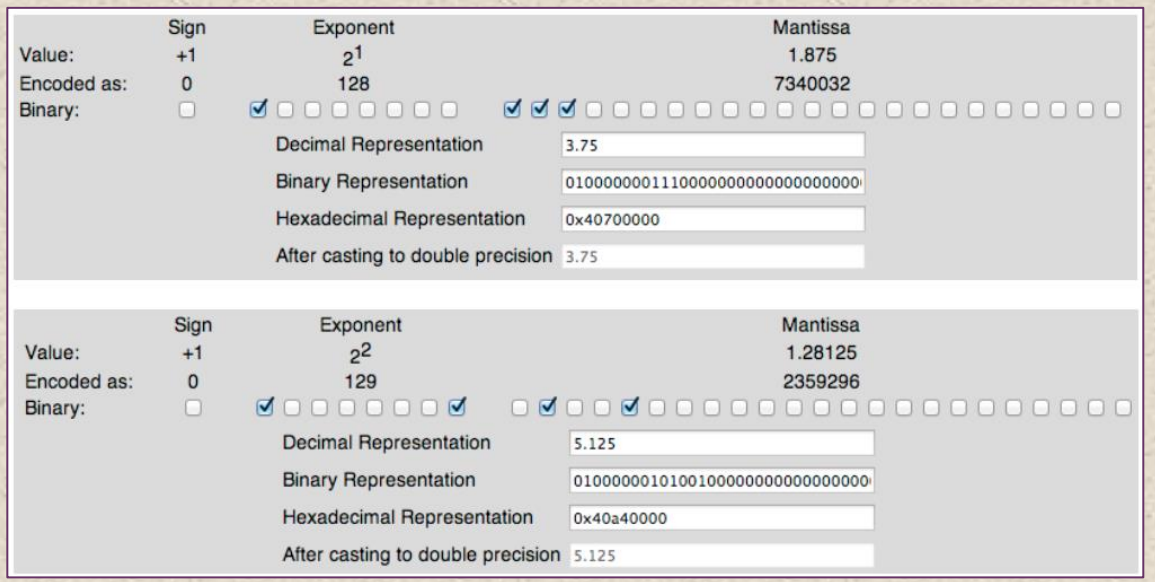
The sum fits in 4 bits so rounding is not required

Check: $1.000 \times 2^{-4} = 0.0625$ which is equal to $0.5 - 0.4375$

Example

Step 1: Decompose Operands

- Add the floating point numbers 3.75 and 5.125 to get 8.875 by directly manipulating the numbers in IEEE format.



- For 3.75, the sign bit is 0 (+), the exponent is 128 (1 unbiased), the mantissa (including the implicit 1 shown in bold) is: 0000 0000 **1**111 0000 0000 0000 0000 0000 = 0x00f00000

- For 5.125, the sign bit is 0 (+), the exponent is 129 (2 unbiased), the mantissa (including the implicit 1 shown in bold) is: 0000 0000 **1**010 0100 0000 0000 0000 0000 = 0x00a40000

Example

Step 2: Equalizing Operand Exponents

- If the first exponent is smaller than the second, we shift the first mantissa to the right and add the absolute difference in exponents to the first exponent. If vice versa, we do the same to the second mantissa and exponent.
- For this example the first exponent is 128, second exponent is 129, absolute difference is 1, so first exponent is smaller, so we must adjust the first mantissa and exponent, and leave the second mantissa and exponent unchanged.
  - Shift first mantissa right by 1: 0x00f00000 >> 1 = 0x00780000
  - Increase the first exponent by 1: 128 + 1 = 129

Example

Step 3: Convert operands from signed magnitude to 2's complement

- For each operand that is negative, convert the mantissa to 2's complement by inverting the bits and adding 1. Neither operand is negative in this example, so nothing needs to be done.

Step 4: Add Mantissas

- Both operands have an exponent of 129, so we can just add mantissas to get a positive result with the same exponent.
  - 0x00780000 + 0x00a40000 = 0x011c0000 = 0000 0001 0001 1100 0000 0000 0000 0000

Step 5: Convert result from 2's complement to signed magnitude

- If the result is negative, convert the mantissa back to signed magnitude by inverting the bits and adding 1. The result is positive in this example, so nothing needs to be done.

Example

Step 6: Normalize Result

- Because the leftmost 1 bit is not in the right place, we must shift the mantissa right or left to put it back into the IEEE format, and adjust the exponent accordingly.

- If the leftmost 1 bit is left of bit 23, we must shift the mantissa to the right and increase the exponent. If the leftmost 1 bit is at bit 23, there is no normalization required. If the leftmost 1 bit is right of bit 23, we must shift the mantissa to the left and decrease the exponent. In the example, we see the first case:

BEFORE NORMALIZATION

Sign of result = 0
Exponent of result = 129
Mantissa of result = 0000 0001 0001 1100 0000 0000 0000 0000 = 0x011c0000

The leftmost 1 bit is bit 24, so we must shift the mantissa right by 1 and add 1 to the exponent:

AFTER NORMALIZATION

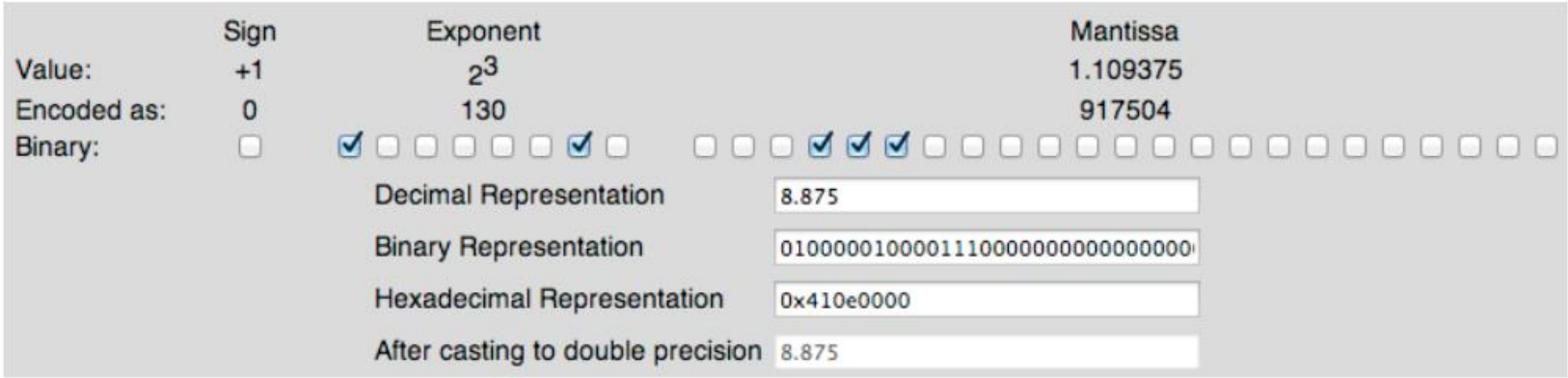
Sign of result = 0
Exponent of result = 129 + 1 = 130 = 100000010
Mantissa of result = 0x011c0000 >> 1 = 0x008e0000 = 0000 0000 **1**000 1110 0000 0000 0000 0000

Example

Step 7: Compose Result

Sign of result = 0
Exponent of result = 130 = 10000010
Mantissa of result (with implicit 1) = 0x008e0000 = 0000 0000 **1**000 1110 0000 0000 0000 0000
Mantissa of result (remove implicit 1) = 0x000e0000 = 0000 0000 0000 1110 0000 0000 0000 0000

So final result is: 0 10000010 000 1110 0000 0000 0000 0000 = 0x410e0000, as confirmed by the h-schmidt converter:



To summarize the result of the addition:

3.75 (0x40700000) = + 5.125 (0x40a40000) = 8.875 (0x410e0000)

FP Arithmetic x/÷

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

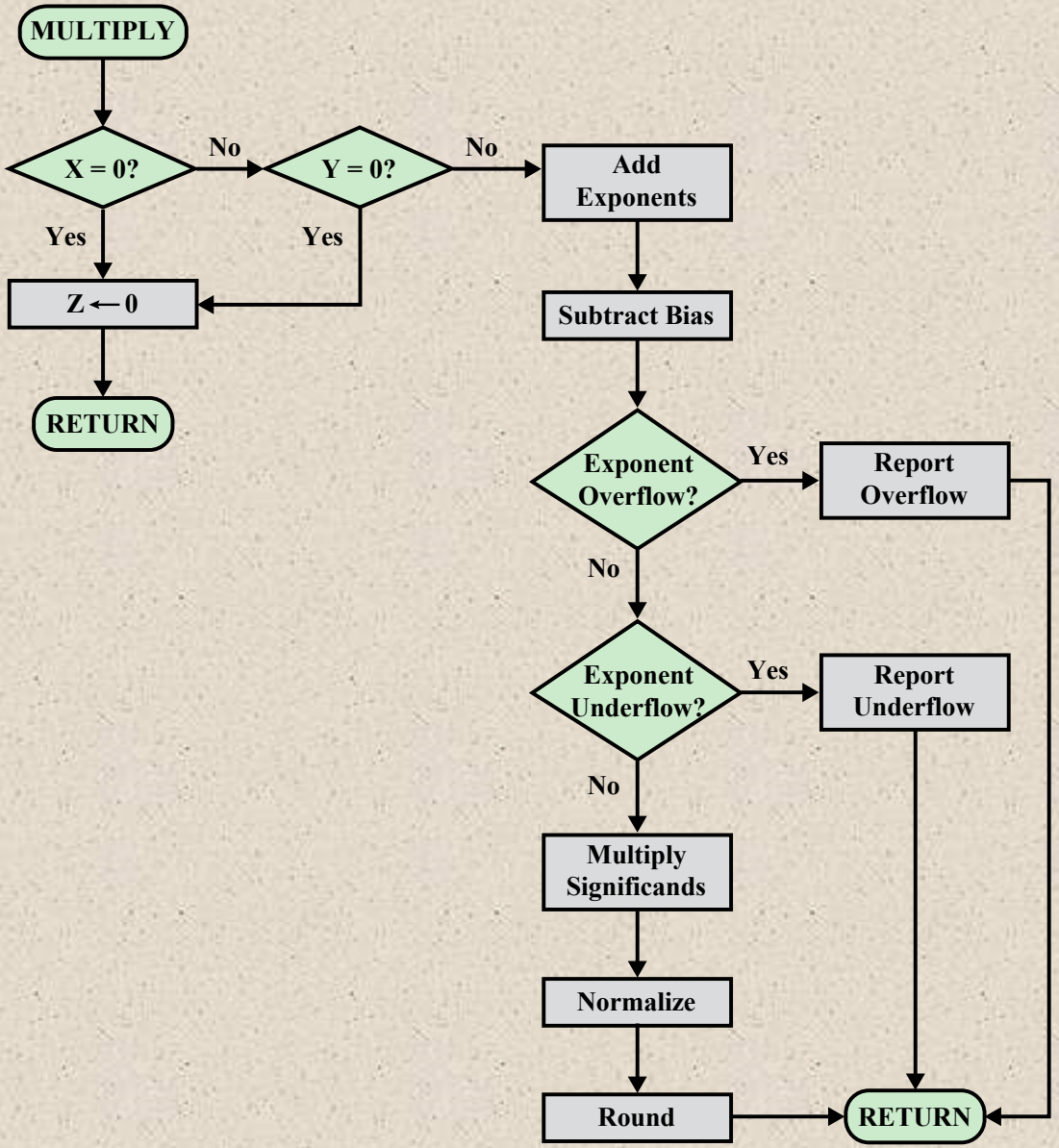


Figure 10.23 Floating-Point Multiplication (Z← X× Y)

Example

$1.000 \times 2^{-1} \times -1.110 \times 2^{-2}$

1. Add the biased exponents

$(-1 + 127) + (-2 + 127) - 127 = 124 \Rightarrow (-3 + 127)$

2. Multiply the mantissas

```
   1.000
x  1.110
-----------
       0000
      1000
     1000
+   1000
-----------
    1110000  ===> 1.110000
```

The product is 1.110000×2^{-3}

Need to keep it to 4 bits 1.110×2^{-3}

3. Normalise (already normalised)

   At this step check for overflow/underflow by making sure that

$-126 <= \text{Exponent} <= 127$

$1 <= \text{Biased Exponent} <= 254$

4. Round the result (no change)

5. Adjust the sign.

Since the original signs are different, the result will be negative

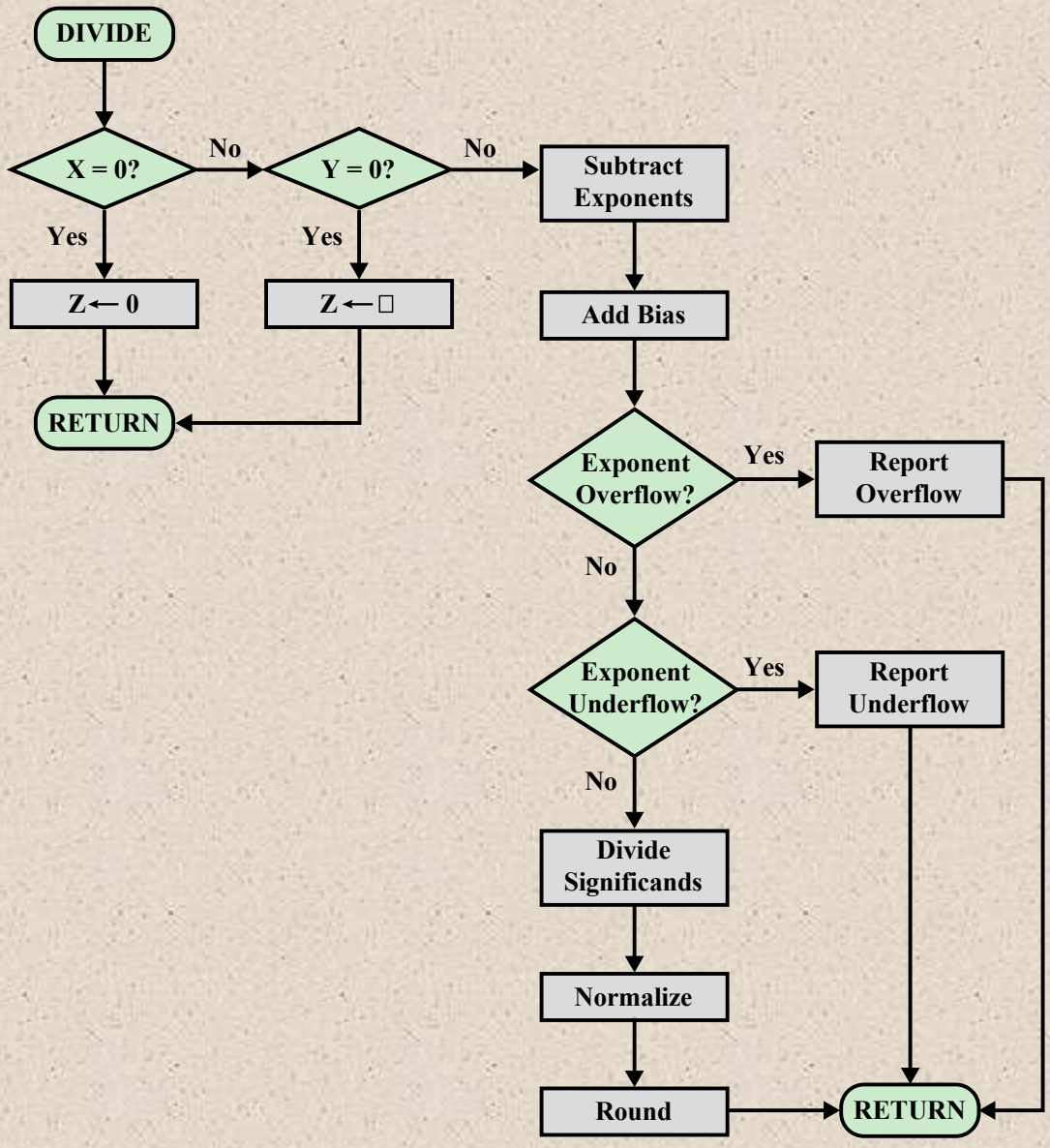


Figure 10.24 Floating-Point Division (Z← X/Y)