Computer Organization

Computer Arithmetic

Chapter 9

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 $r = (d_{k-1}d_{k-2} \ldots d_1 d_0)_r$

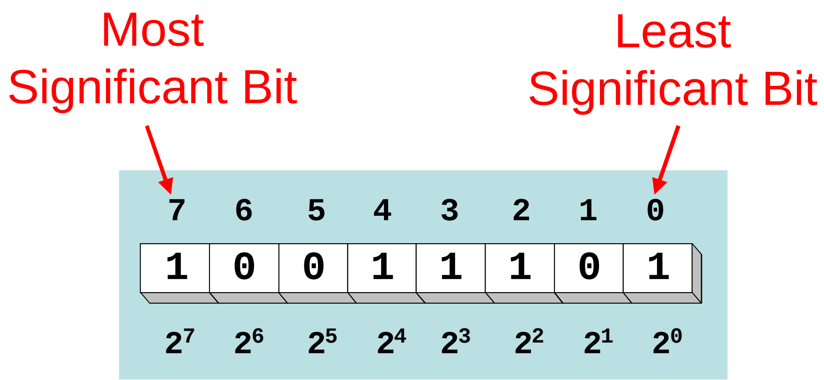
Value $= d_{k-1} \times r^{k-1} + d_{k-2} \times r^{k-2} + \ldots + d_1 \times r + d_0$

Examples: $(11011)_2 = 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2 + 1 = 27$

$(2103)_4 = 2\times4^3 + 1\times4^2 + 0\times4 + 3 = 147$

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 an 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:

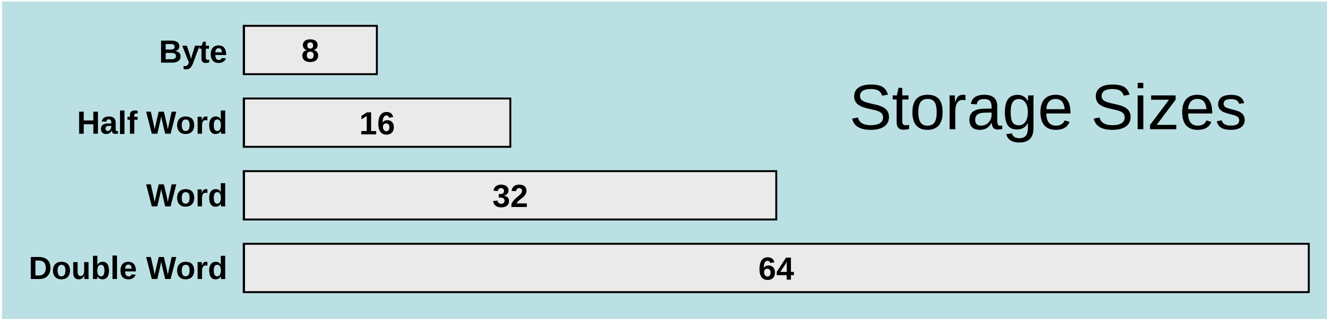
  Convert the 32-bit binary number to hexadecimal

  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?

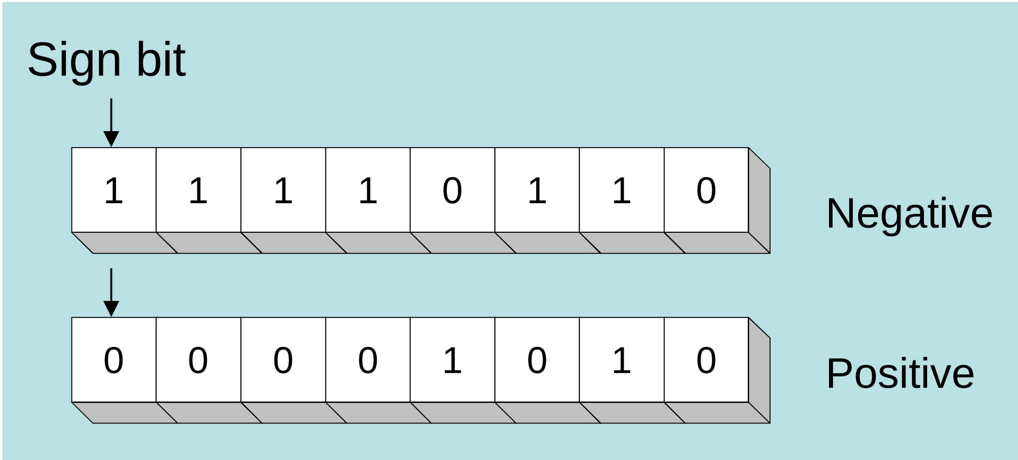
Answer: $2^{20} - 1 = 1{,}048{,}575$

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

Sign Bit

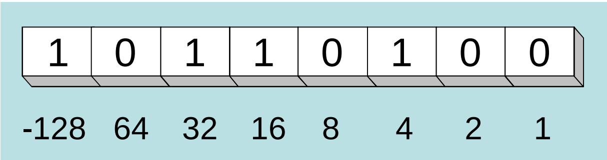
- Highest bit indicates the sign
- 1 = negative
- 0 = positive



- 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)

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

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

    10110011 = -77 ⇒ 11111111 10110011 = -77

  - Sign-Extend 01100010 to 16 bits

    01100010 = +98 ⇒ 00000000 01100010 = +98
- 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	Unsigned 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?

Binary Addition

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

carry		1	1	1	1				
	0	0	1	1	0	1	1	0	(54)
+	0	0	0	1	1	1	0	1	(29)
	0	1	0	1	0	0	1	1	(83)
bit position:	7	6	5	4	3	2	1	0	

Binary Subtraction

❖ When subtracting A – B, convert B to its 2's complement

❖ Add A to (–B)

borrow:			1	1			1		
		0	1	0	0	1	1	0	1
–		0	0	1	1	1	0	1	0
		0	0	0	1	0	0	1	1

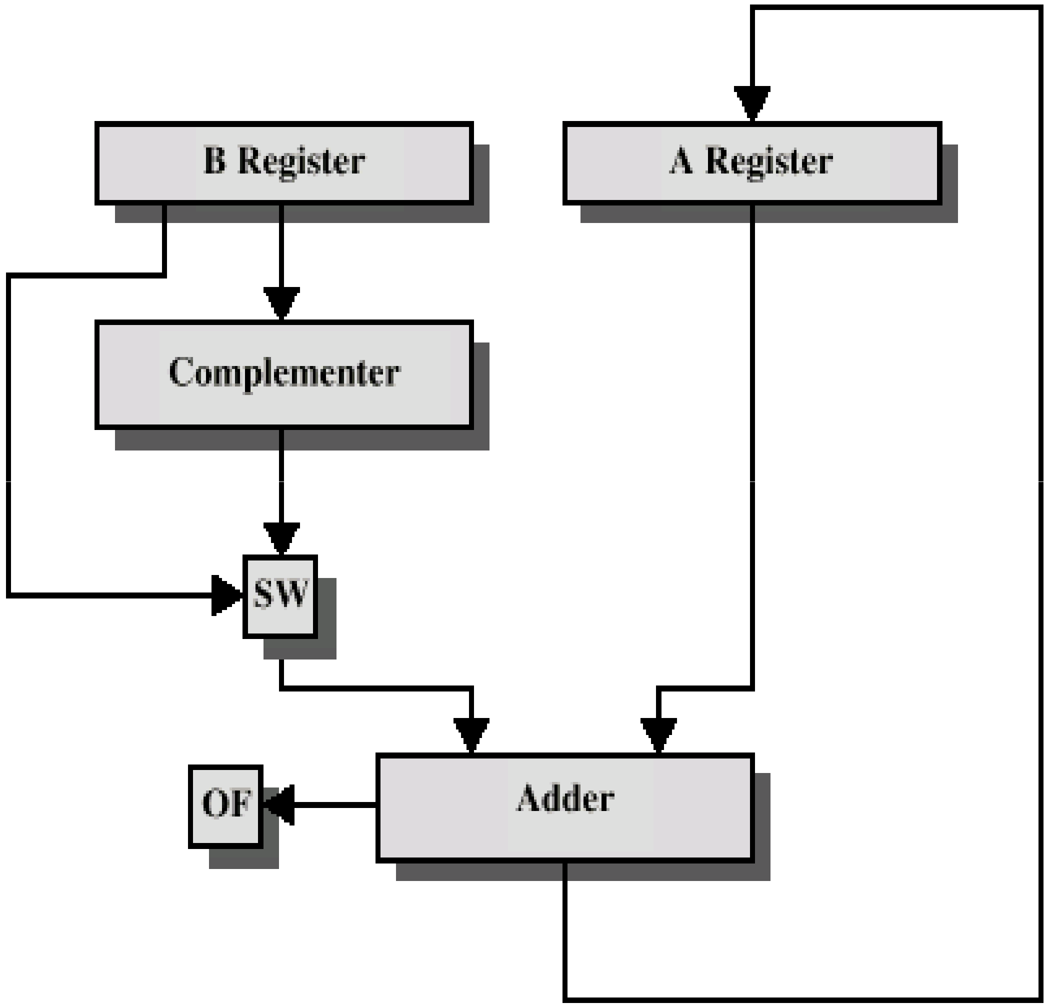
⇨

carry:	1	1			1	1				
		0	1	0	0	1	1	0	1	
+		1	1	0	0	0	1	1	0	(2's complement)
		0	0	0	1	0	0	1	1	(same result)

❖ 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

Hardware for Addition and Subtraction



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

Carry and Overflow

- Carry is important when …
  - Adding or subtracting unsigned integers
  - Indicates that the unsigned sum is out of range
  - Either < 0 or >maximum unsigned *n*-bit value
- Overflow is important when …
  - Adding or subtracting signed integers
  - Indicates that the signed sum is out of range
- Overflow occurs when
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
  - Can happen because of the fixed number of sum bits

Carry and Overflow Examples

- We can have carry without overflow and vice-versa
- Four cases are possible (Examples are 8-bit numbers)

				1					
	0	0	0	0	1	1	1	1	15
+	0	0	0	0	1	0	0	0	8
	0	0	0	1	0	1	1	1	23

Carry = 0 Overflow = 0

1	1	1	1	1					
	0	0	0	0	1	1	1	1	15
+	1	1	1	1	1	0	0	0	248 (-8)
	0	0	0	0	0	1	1	1	7

Carry = 1 Overflow = 0

	1								
	0	1	0	0	1	1	1	1	79
+	0	1	0	0	0	0	0	0	64
	1	0	0	0	1	1	1	1	143 (-113)

Carry = 0 Overflow = 1

1			1	1					
	1	1	0	1	1	0	1	0	218 (-38)
+	1	0	0	1	1	1	0	1	157 (-99)
	0	1	1	1	0	1	1	1	119

Carry = 1 Overflow = 1

Addition of Numbers in Twos Complement Representation

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)

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

```
   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
```

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

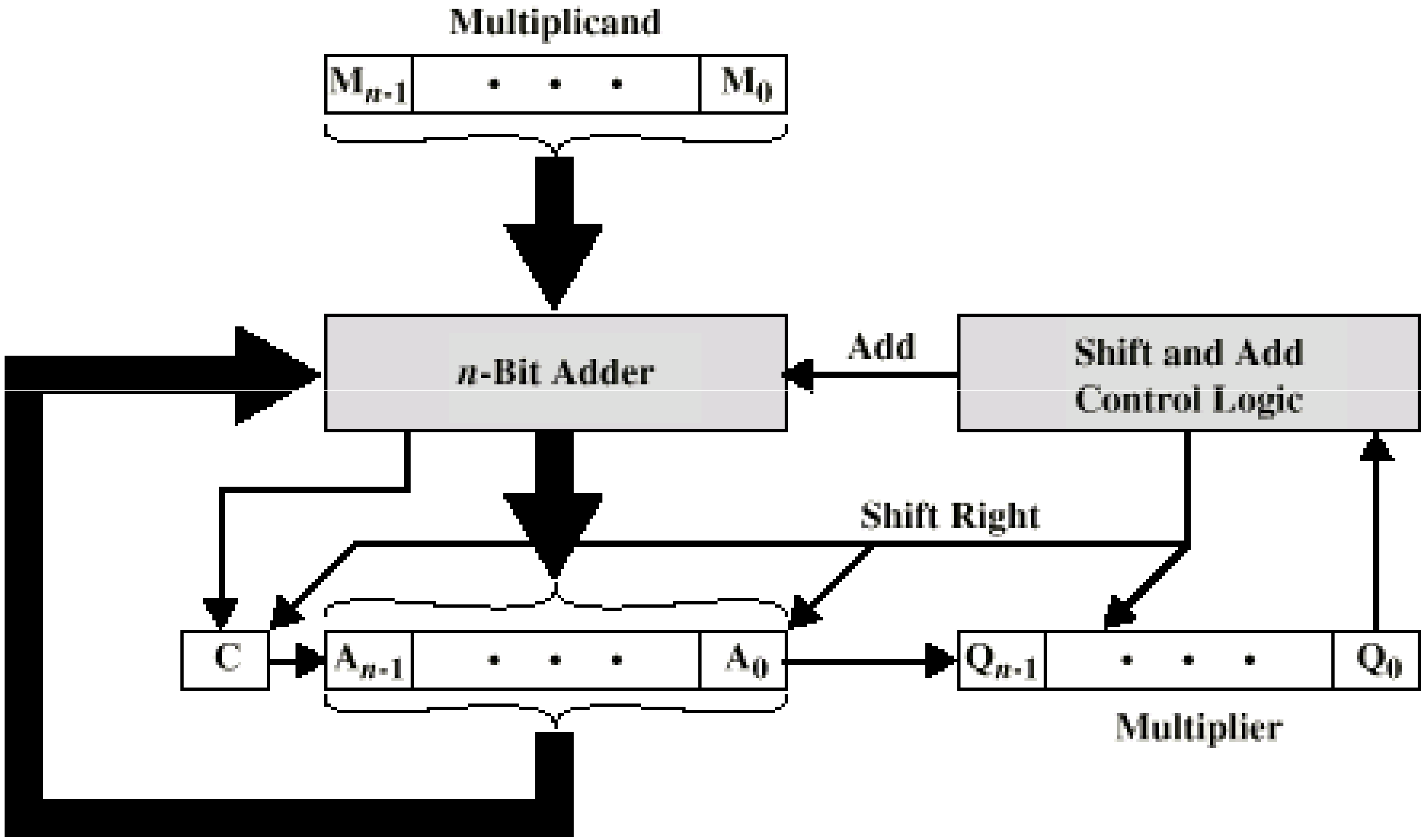
1 × multiplicand = multiplicand

❖ m-bit multiplicand × n-bit multiplier = (m+n)-bit product

❖ Accomplished via shifting and addition

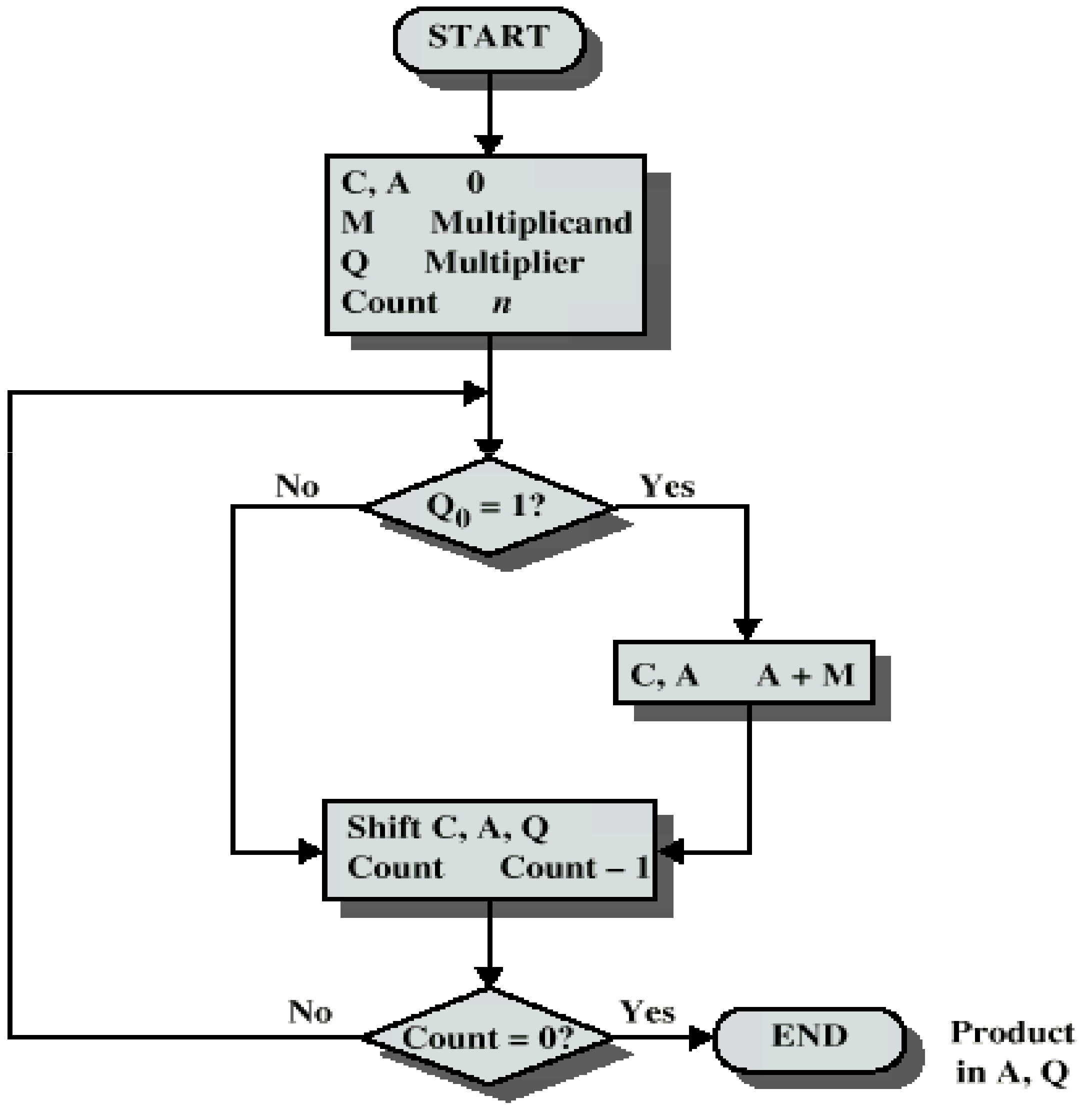
❖ Consumes more time and more chip area

Unsigned Binary Multiplication



(a) Block Diagram

Flowchart for Unsigned Binary Multiplication



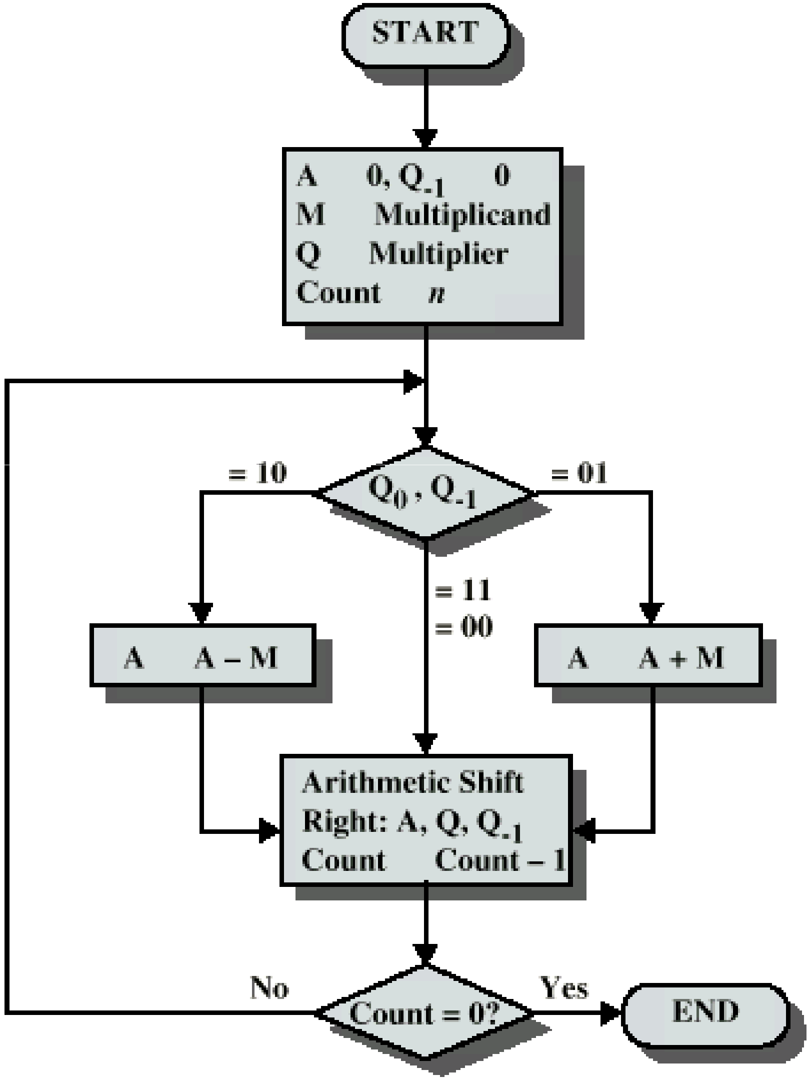
Execution of Example

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	

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



Example of Booth's Algorithm

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

Examples Using Booth's Algorithm

```
    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)

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:

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

How it works

- So, the product can be generated by one addition and one subtraction

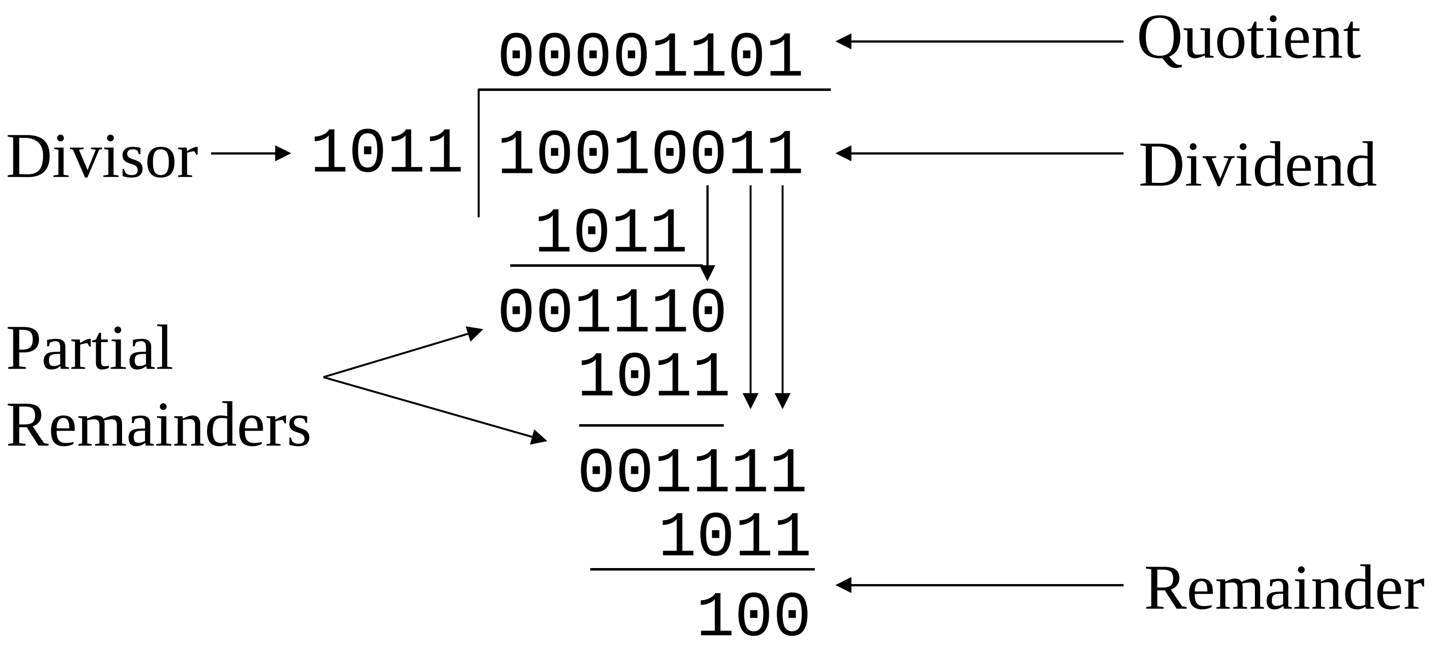
- In Booth's algorithm
  - perform subtraction when the first 1 of the block is encountered (1 - 0)
  - perform addition when the last 1 of the block is encountered (0 - 1)

- (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

Division of Unsigned Binary Integers



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

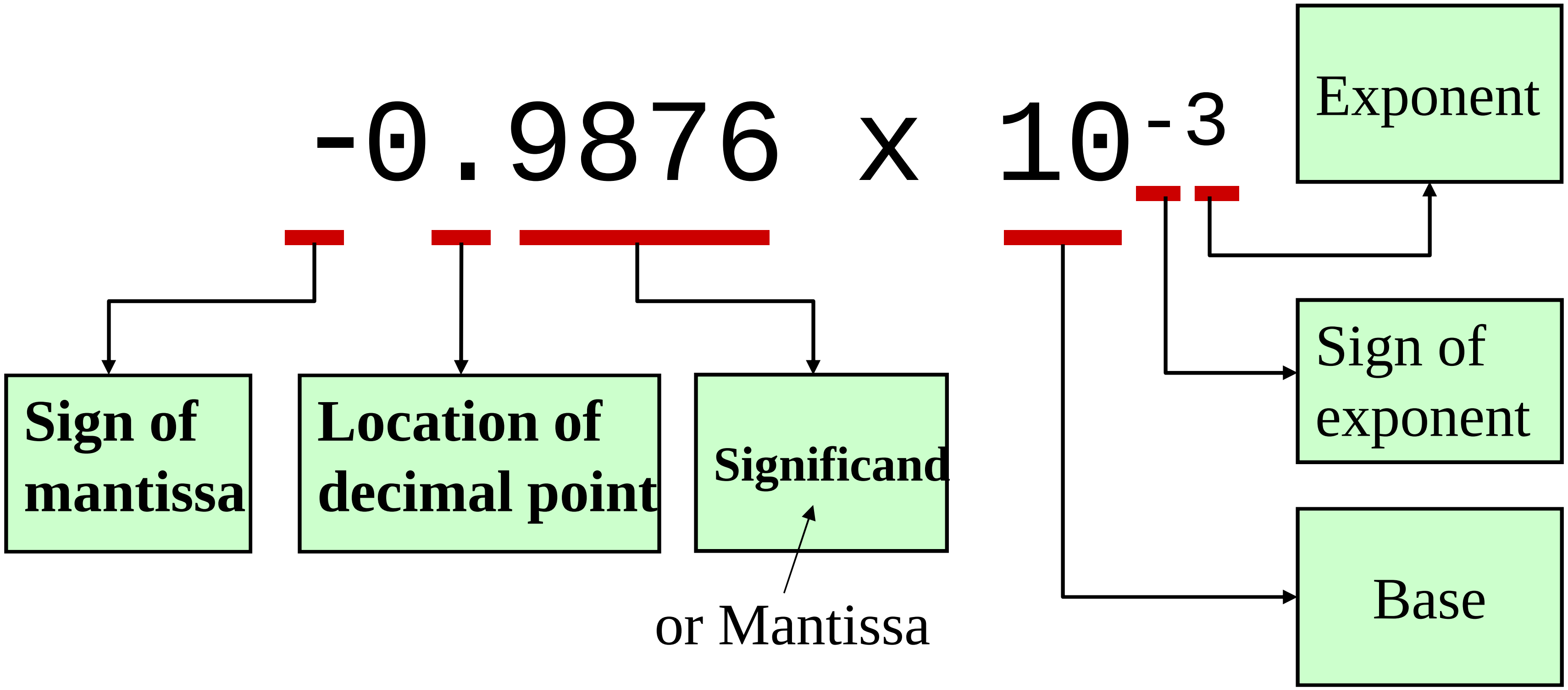
- The following are equivalent representations of 1,234

123,400.0	$\times 10^{-2}$
12,340.0	$\times 10^{-1}$
1,234.0	$\times 10^{0}$
123.4	$\times 10^{1}$
12.34	$\times 10^{2}$
1.234	$\times 10^{3}$
0.1234	$\times 10^{4}$

The representations differ in that the decimal place – the "point" -- "floats" to the left or right (with the appropriate adjustment in the exponent).

p. 122

Parts of a Floating Point Number



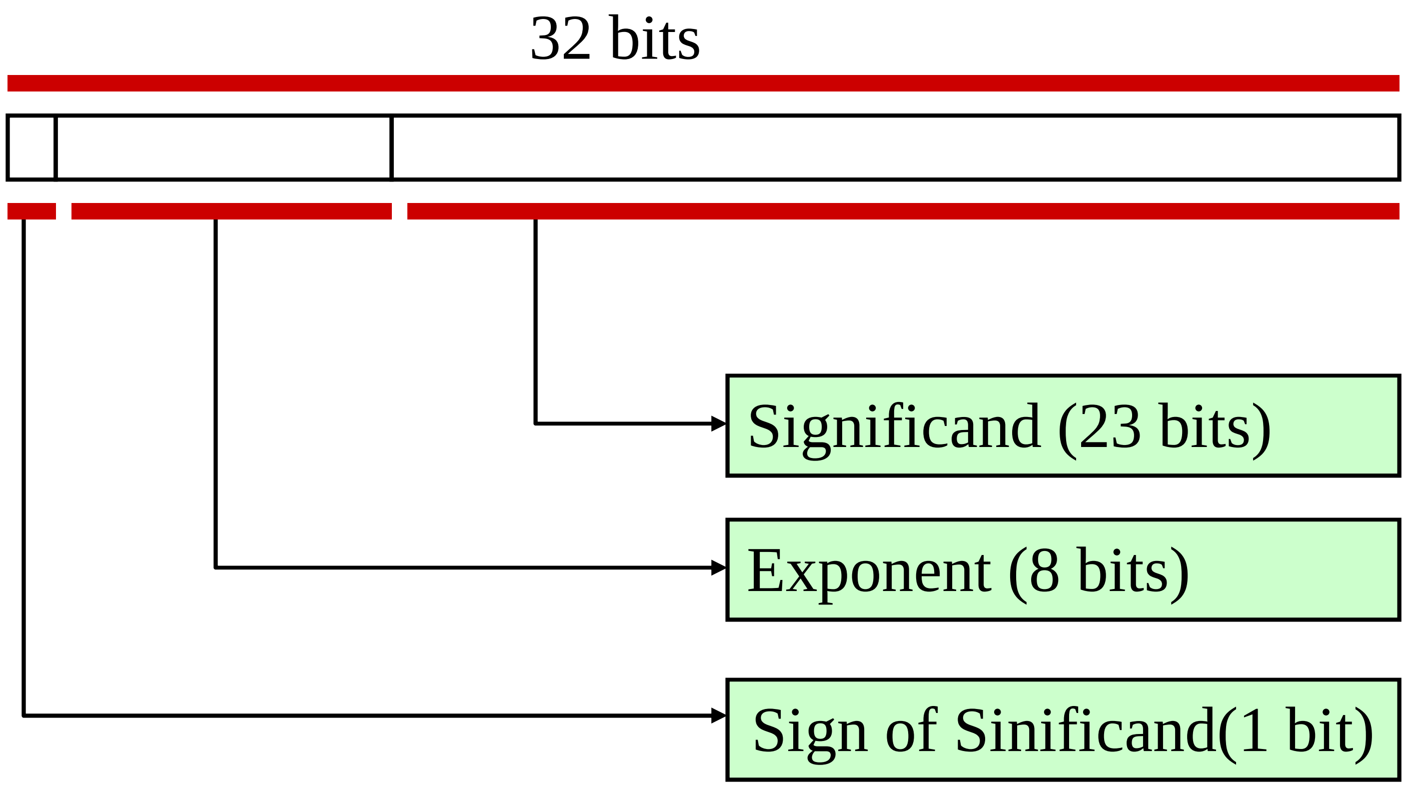
In binary, the significand is represented by 1s and 0’s, and the

Base = 2. E.g. -1.1111011 x 2^3

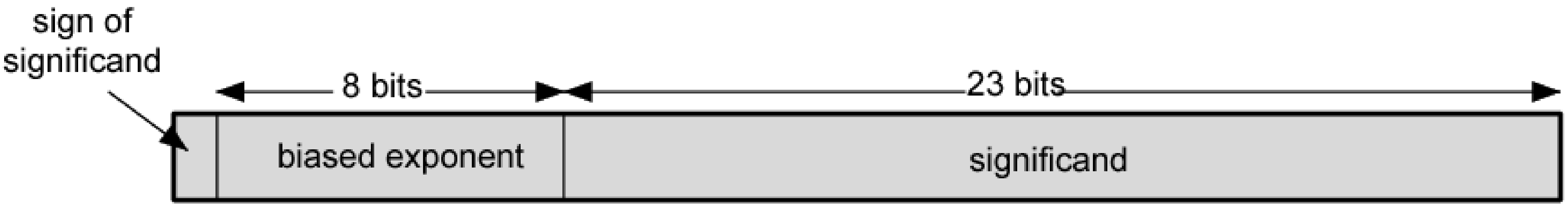
Biased Representation

- 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 = 1111
  - Representation of -7 = 0000

Representation Format



Floating Point



(a) Format

- +/- .significand x $2^{exponent}$
- Misnomer
- Point is actually fixed between sign bit and body of mantissa
- Exponent indicates place value (point position)

Floating Point Examples

(a) Format

$$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20}$$

$$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20}$$

$$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20}$$

$$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}$$

(b) Examples

Signs for Floating Point

- Mantissa is stored in 2s compliment
- Exponent is in excess or **biased notation**
  - e.g. Excess (bias) 128 means
  - 8 bit exponent field
  - Pure value range 0-255 (8-bit)
  - Subtract 127 to get correct value
    - **Bias= 2^{8-1}-1= 127**
  - Range of exponent values: -127 to +128
    - For representation: bias must be added for any value
    - Exponent value -127 is represented as -127+127 = 0 (00000000:Min value)
    - Exponent value +128 is represented as 128+127 = 255 (11111111:Max value)

Normalization

- FP numbers are usually normalized
- i.e. exponent is adjusted so that leading **bit (MSB) of Significand is 1**
- Since it is **always 1** there is no need to store it
- (c.f. Scientific notation where numbers are normalized to give a single digit before the decimal point
- E.g.,
  - Significand → **10100000000000000000000**
  - Represents… $\mathbf{1.101_2 = 1.625_{10}}$

Converting <u>from</u> 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

$C17B0000_{16}$ =

$\underline{1}\ \underline{10000010}\ \underline{11110110000000000000000}_2$

S E M

1 = negative
0 = positive

- Step 2
  - Find "real" exponent, n
  - $n = E - 127$

    $= 10000010_2 - 127$

    $= 130 - 127$

    $= 3$

- 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

-1111.1011_2

-15

$2^{-1} = 0.5$

$2^{-3} = 0.125$

$2^{-4} = 0.0625$

0.6875

Answer: -15.6875

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 = + (sign), M = 001001001, n = 5

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

S = 0 (because the value is positive)

- Step 4
  - Put S, E, and M together to form 32-bit binary result

$$\underbrace{0}_{S}\ \underbrace{10000100}_{E}\ \underbrace{00100100100000000000000}_{M}{}_2$$

- Step 5
  - Express in hexadecimal

$$0\ 10000100\ 00100100100000000000000_2 =$$

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

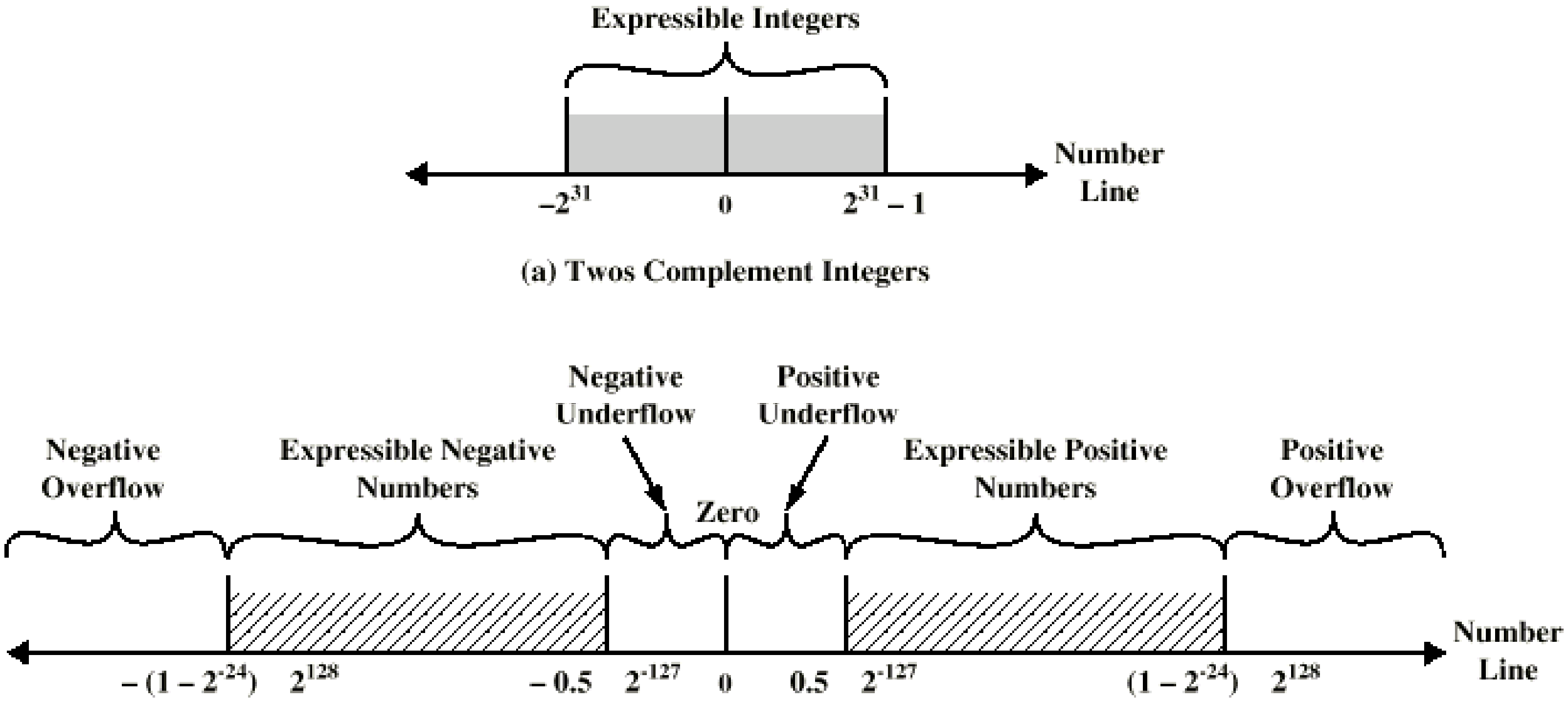
$$4\quad 2\quad 1\quad 2\quad 4\quad 0\quad 0\quad 0_{16}$$

Answer: 42124000_{16}

FP Ranges

- For a 32 bit number
  - 8 bit exponent
  - +/- $2^{256} \approx 1.5 \times 10^{77}$
- Accuracy
  - The effect of changing lsb of mantissa
  - 23 bit mantissa $2^{-23} \approx 1.2 \times 10^{-7}$
  - About 6 decimal places

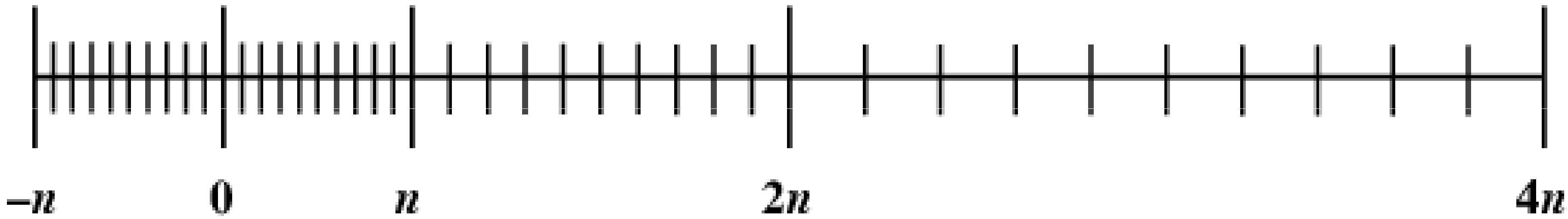
Expressible Numbers



(a) Twos Complement Integers

(b) Floating-Point Numbers

Density of Floating Point Numbers

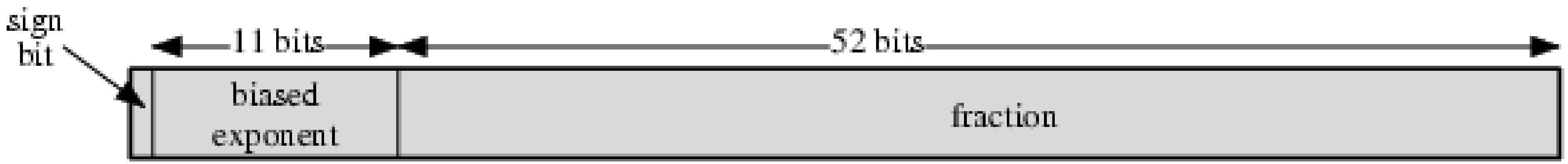


IEEE 754

- Standard for floating point storage
- 32 and 64 bit standards
- 8 and 11 bit exponent respectively
- Extended formats (both mantissa and exponent) for intermediate results

IEEE 754 Formats

(a) Single format

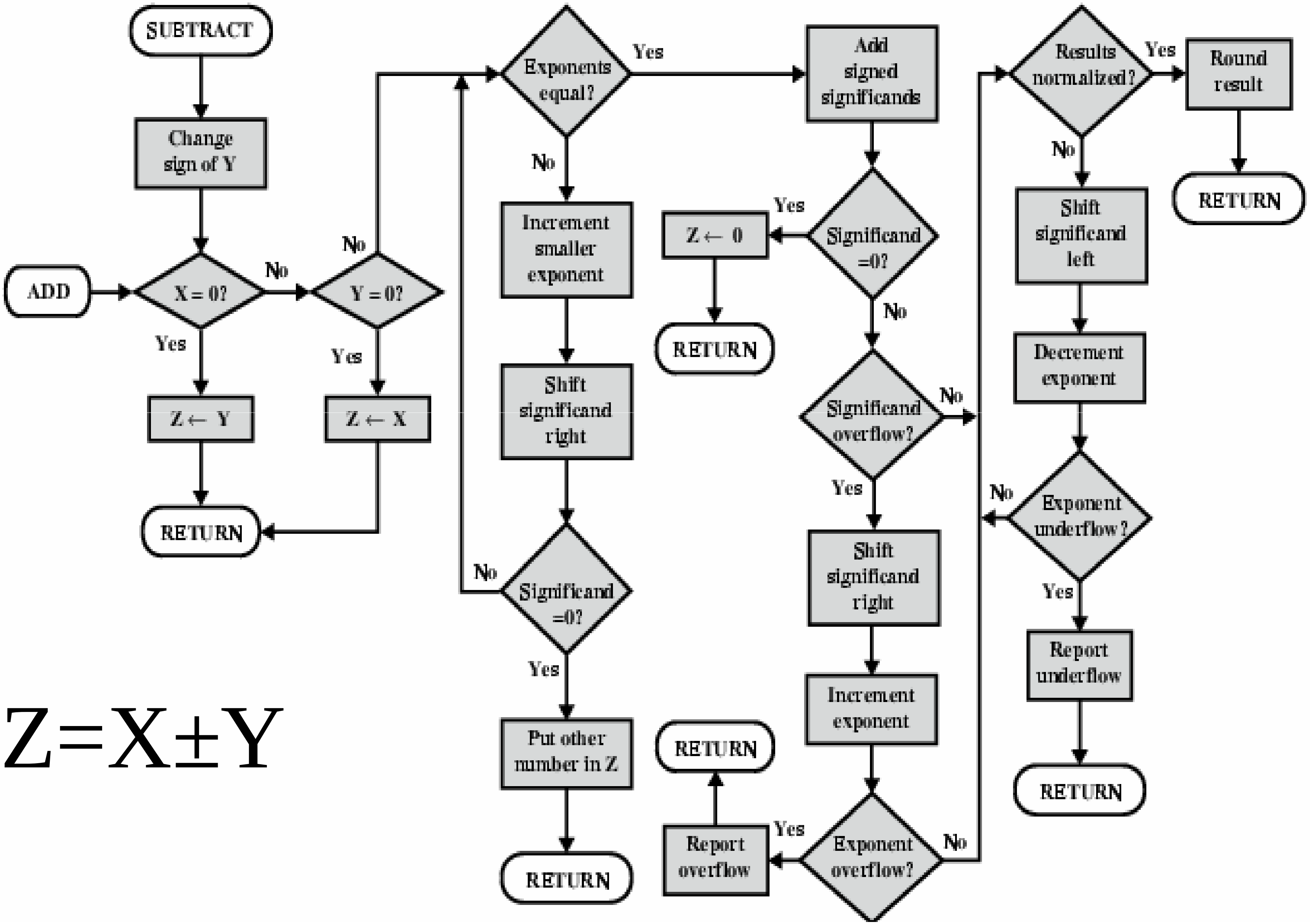


(b) Double format

FP Arithmetic +/-

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

FP Addition & Subtraction Flowchart

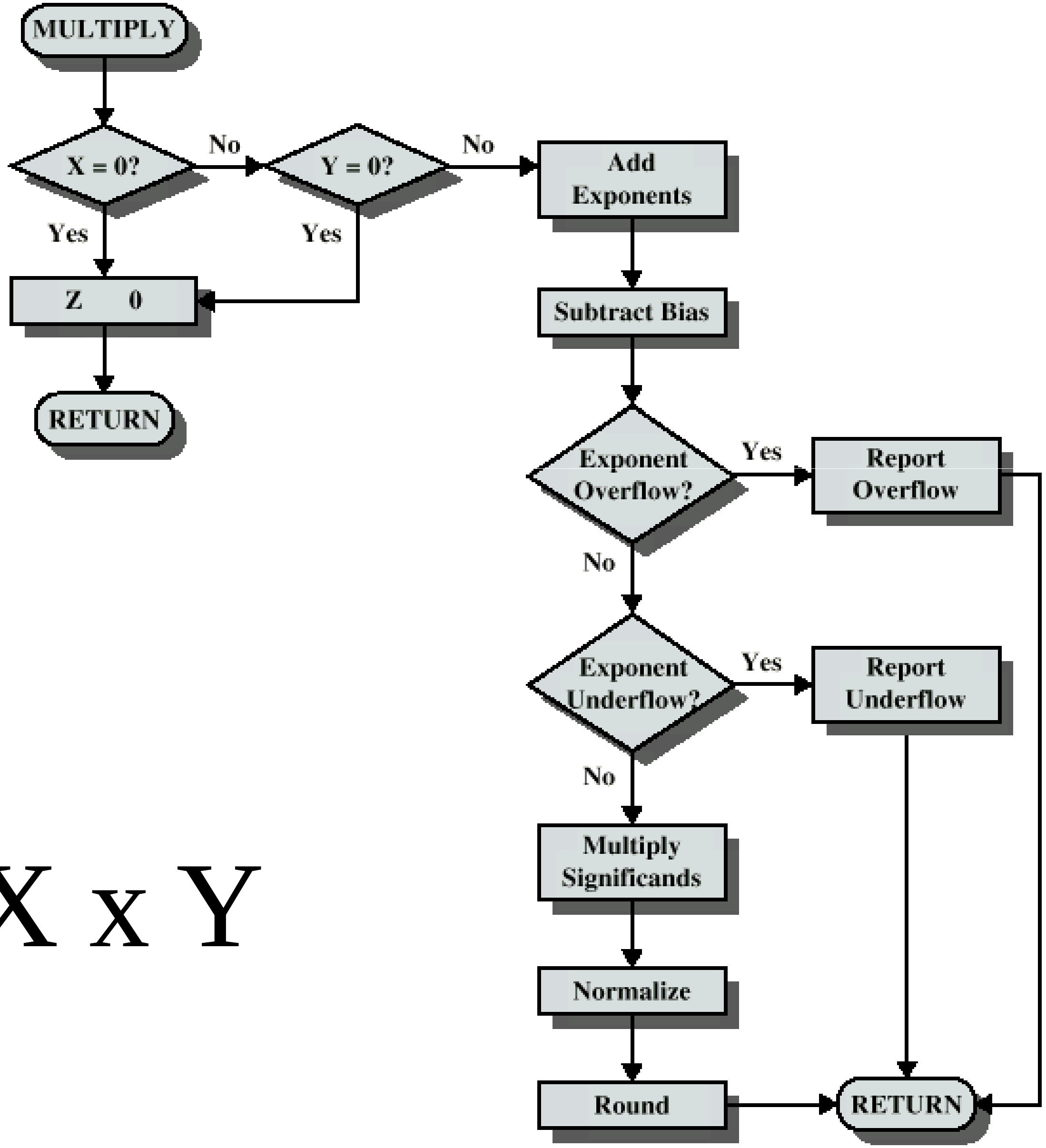


$$Z = X \pm Y$$

FP Arithmetic x/÷

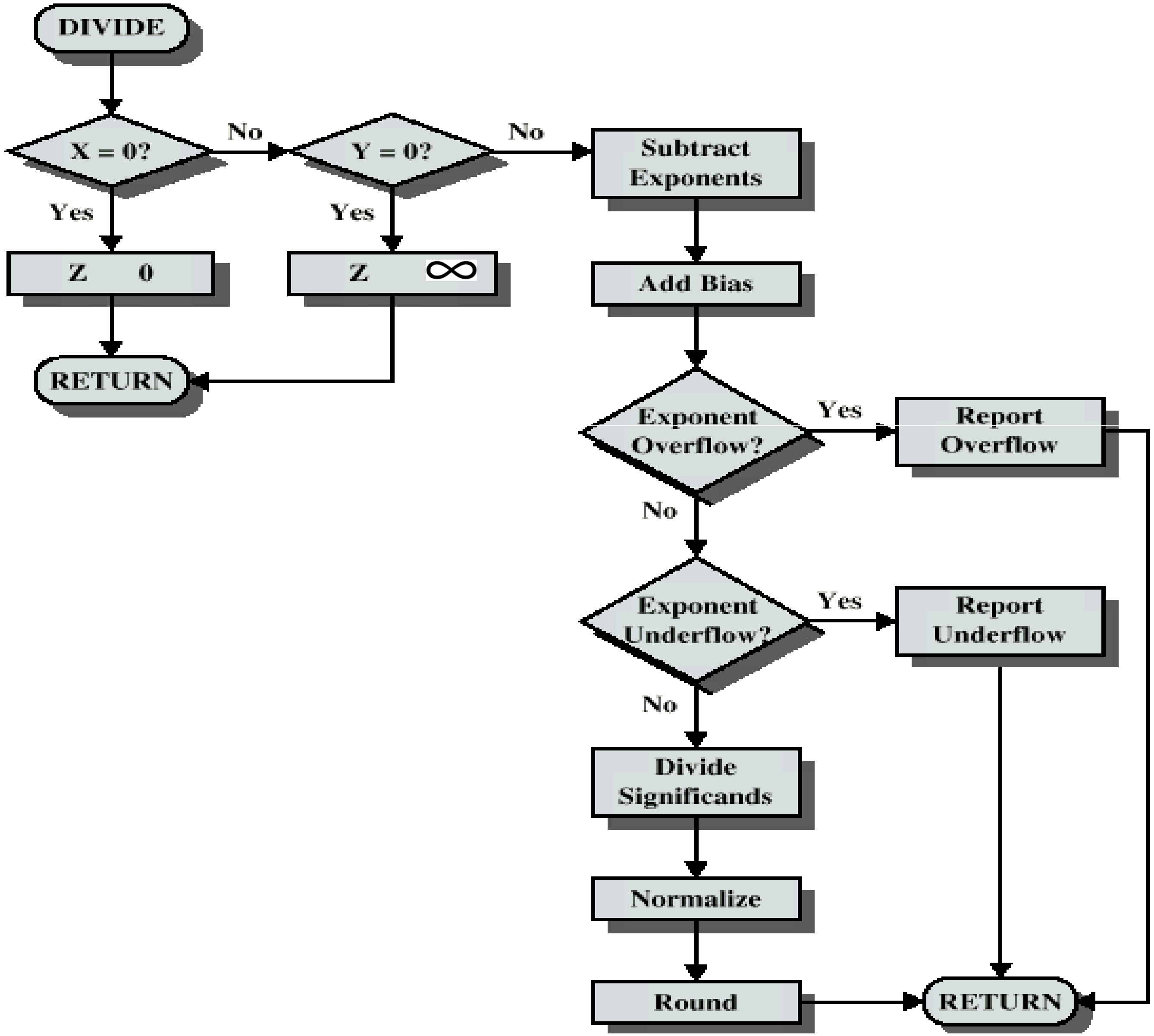
- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

Floating Point Multiplication



Z=X x Y

Floating Point Division



Z=X/Y