TSN1101
Computer Architecture and Organization

Section A (Digital Logic Design)

Lecture A-07

Data Representation and Arithmetic

TOPIC COVERAGE IN THE LECTURE (1)…

- Representation of integers/Fixed-Point Representation
  - Sign-magnitude
    - Examples, drawbacks
  - Ones complement
    - Examples, drawbacks
  - Twos complement
    - Examples, Characteristics, Advantages, Value-box for conversion
  - Conversion between different bit lengths

TOPIC COVERAGE IN THE LECTURE (2)…

- Addition and Subtraction of Twos complement numbers
  - Addition
    - Steps, Detection of Overflow, Examples
  - Subtraction
    - Steps, Examples
  - Hardware for Addition/Subtraction
- Multiplication of unsigned binary integers
  - Block diagram, Flow Chart, Examples
- Multiplication of twos complement numbers
  - Booth's algorithm – Flow Chart, Examples

TOPIC COVERAGE IN THE LECTURE (3)

- Fixed-point representation
  - Drawbacks
- Floating-point Representation
  - Principles
  - IEEE 754 standard format
    - Single Precision and Double Precision
    - Decimal to IEEE754 standard format conversion
    - IEEE754 standard format to decimal equivalent conversion
    - Range of floating-point numbers
- Arithmetic with Floating Point numbers
  - Floating-Point Addition/Subtraction
  - Floating-Point Multiplication/Division
  - Guard bits/Rounding

Data Representation and Arithmetic – Part 1 of 4

Representation of Integers

– Sign magnitude, Ones Complement, and Twos Complement

TOPIC COVERAGE - PART 1 of 4

- Representation of integers/Fixed-Point Representation
  - Sign-magnitude
    - Examples, drawbacks
  - Ones complement
    - Examples, drawbacks
  - Twos complement
    - Examples, Characteristics, Advantages, Value-box for conversion
  - Conversion between different bit lengths

Introduction

- Computers store bit sequences
- The bit sequences can be interpreted as representing integers or floating point numbers or letters.
- Arithmetic can be accomplished by the direct hardware implementation of the arithmetic algorithms.

Fixed point representation

- Representation for integers can be called as **fixed point representation**
  - Called so since the binary point is fixed and assumed to be the right of the right-most digit
- can use the same representation for fractions by scaling the numbers

Integer Representation

- **Representation of nonnegative integers:**
  - 8 bits word could be used to represent the nonnegative numbers from 0 to 255.
    - Example: 41=00101001
- **Representation of negative integers**:
  - No minus sign/No period for computer storage and processing
  - Only have 0 & 1 to represent everything
  - Methods:
    - Sign-Magnitude
    - One's complement
    - Two's complement

Sign-Magnitude Representation - Examples

- Left most bit is sign bit
  - 0 means positive
  - 1 means negative
- Examples: Assume 8-bit sign magnitude representation
  - +1 = **0**0000001
  - +2 = **0**0000010
  - +3 = **0**0000011
  - +45= **0**0101101
    - +0 = **0**0000000
    - -0 = **1**0000000
  - -1 = **1**0000001
  - -2 = **1**0000010
  - -3 = **1**0000011
  - -45 = **1**0101101

Sign Magnitude Representation - Drawbacks

- While performing addition/subtraction of numbers in sign-magnitude representation,
  - First Compare the sign of two operands
  - If the signs are different, then compare the magnitudes of the operands
  - Then perform the subtraction of smaller magnitude number from the larger magnitude number
  - Put the sign of larger magnitude number as the resultant sign.
- Sign Magnitude Rep. is rarely used for implementing the integer portion of ALU
  - Requirement of additional hardware circuitry needed to compare the sign as well as magnitude.
    - Time taken for performing addition/subtraction will also be more.
  - Two representations of zero (+0 and -0)
    - difficult to test for zero
    - always should convert from -0 to +0 .

Ones Complement Representation - Examples

- Assume 8-bit ones complement representation
  - +1 = 00000001
  - +2 = 00000010
  - +3 = 00000011
  - +45= 00101101
    - +0 = 00000000
    - -0 = 11111111
  - -1 = 11111110
  - -2 = 11111101
  - -3 = 11111100
  - -45 = 11010010

Ones Complement Representation (2) - Drawbacks

- Ones complement representation is rarely used for implementing the integer portion of ALU
  - Two representations of zero (+0 and -0)
    - difficult to test for zero
    - always should convert from -0 to +0
  - Complexity of adding carry out of sign position (end-around carry) to LSB of result, when adding two numbers
    - Time taken for performing addition/subtraction will also be more.

Twos Complement Representation - Examples

- Assume 8-bit twos complement representation
  - +1 = 00000001
  - +2 = 00000010
  - +3 = 00000011
  - +45= 00101101
    - 0 = 00000000
  - -1 = 11111111
  - -2 = 11111110
  - -3 = 11111101
  - -45 = 11010011

Twos Complement Representation - Characteristics (1)…

- **Range** : **$-(2^{n-1})$** through **$(2^{n-1})-1$**

  Example: 8 bit Twos complement

  - Range:
    - ✓ $-128 = 10000000 = (-2^7)$ through

      $+127 = 01111111 = (2^7 - 1)$

  16 bit Twos complement

  - Range:
  - ✓ $-32768 = 100000000\ 00000000 = (-2^{15})$ through

    $+32767 = 011111111\ 11111111 = (2^{15} - 1)$

- **Number of representations of zero** : One (There is no negative zero)

Twos Complement Representation - Characteristics (2)…

- **Negation** (conversion from positive to negative and vice versa)
  - **Procedure 1:** 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.
  - **Procedure 2:** Starting from LSB, write the bits as it is up to and including the first 1. Invert the remaining bits.

Twos Complement Representation - Characteristics (3)

- **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 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.

Twos Complement Representation - Advantages

- Only One representation of zero
- Arithmetic works easily - No special Hardware required
- Negation is fairly easy
  - Example:
    - 3 = 00000011
    - One way:
      - Boolean complement gives 11111100
      - Add 1 to LSB 11111101
    - Another way:
      - Starting from LSB, write the bits as it is up to and including the first 1, find Ones complement of remaining bits
      - **11111101**

Conversion between Twos Complement Binary and Decimal
- Value Box approach (1)…

An Eight-position Two's Complement Value Box

-128	**64**	**32**	**16**	**8**	**4**	**2**	**1**

Example 1: Convert 8-bit Twos complement binary 10000011 to Decimal

-128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	0	0	0	0	0	1	1
-128						+2	+1

= -128 +2 +1 = -125

Conversion between Twos Complement Binary and Decimal
- Value Box approach (2)

Example 2: Convert Decimal –120 to Twos complement Binary representation

-120 = -128 + 8

-128	64	32	16	8	4	2	1
-128				+8			
1	0	0	0	1	0	0	0

-120 = 10001000

Conversion Between different bit lengths - Sign Magnitude

- Sometimes desirable to take n-bit integer and store in m bits, m>n
- For **Sign-Magnitude**, move the sign bit to the new left-most position and fill in with zeros.
  - +18 = **0**001 0010 (Sign Magnitude, 8bits)
  - +18 = **0**000 0000 0001 0010 (Sign Magnitude,16 bits)
  - -18 = **1**001 0010 (Sign Magnitude,8 bits)
  - -18 = **1**000 0000 0001 0010 (Sign Magnitude,16 bits)

Conversion Between different bit lengths - Ones Complement

- For **Ones complement**,
- Positive numbers- pack with leading zeros
  - +18 = **0**0010010 (Ones complement - 8 bits)
  - +18 = 00000000 00010010 (Ones complement - 16 bits)
- Negative numbers- pack with leading ones
  - -18 = **1**1101101 (Ones complement-8 bits)
  - -18 = 11111111 11101101 (Ones complement-16 bits)
- i.e. Pack with MSB (Sign bit)
- called **Sign extension**

Conversion Between different bit lengths - Twos complement

- For **Twos complement**,
- Positive number pack with leading zeros
  - +18 = **0**0010010 (Twos complement - 8 bits)
  - +18 = 00000000 00010010 (Twos complement - 16 bits)
- Negative numbers pack with leading ones
  - -18 = **1**1101110 (Twos complement-8 bits)
  - -18 = 11111111 11101110 (Twos complement-16 bits)
- i.e. Pack with MSB (Sign bit)
- called **Sign extension**

Summary

– Example - Different Representations for 4-bit Integers

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

Summary

– Characteristics of Sign-magnitude and Ones complement numbers

	sign-magnitude	ones complement
Range	$-(2^{n-1}-1)$ to $(2^{n-1}-1)$	$-(2^{n-1}-1)$ to $(2^{n-1}-1)$
Number of representations of 0	2	2
Negation	Complement the sign bit	Complement each bit
Expansion of bit length	Move the sign bit to the new leftmost bit; fill in with zeros	Fill all new bit positions to the left with the sign bit
Subtract B from A	Complement the sign bit of B and add B to A using rules for addition of sign-magnitude numbers	Take the ones complement of B and add it to A

3-2
=0011 + 1010
= 0001

3-2
=0011 - 1101
=0000 + 1 = 0001

4-2
=0100-1101
=0001+1 = 0010

Summary

– Characteristics of Twos complement numbers

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*.

Data Representation and Arithmetic – Part 2 of 4

Integer Arithmetic with twos complement numbers

– Addition and Subtraction

TOPIC COVERAGE - PART 2 of 4

- Addition and Subtraction of Twos complement numbers
  - Addition
    - Steps, Detection of Overflow, Examples
  - Subtraction
    - Steps, Examples
  - Hardware for Addition/Subtraction

Integer Arithmetic with **twos complement numbers** - Addition (Steps)

- Perform normal binary addition as if two numbers were unsigned integers
- If the results of operation is positive, we get positive number in twos complement form (same as unsigned integer form)
- If the result of operation is negative, we get negative number in twos complement form.
- Discard the carry, if there is any.
- Monitor sign bit for overflow

Integer Arithmetic with **twos complement numbers** - Addition (Detection of Overflow)

- If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the results has the opposite sign.
- Detection of Overflow:
  - ❖Carry out of sign bit and carry into the sign bit will be given as two inputs to the exclusive-OR gate and the output of exclusive-OR gate is connected to overflow flip-flop.
  - ❖If carry out of sign bit and carry into sign bit are equal - NO overflow
  - ❖If carry out of sign bit and carry into sign bit are not equal - OVERFLOW occurs
- Note that overflow can occur whether or not there is a carry.

Integer Arithmetic with **twos complement numbers**
- Addition (Examples)

(-7) + (+5) 1001 +0101 1110 = -2	**(-4) + (+4)** 1100 +0100 **1** 0000 = 0
(+3) + (+4) 0011 +0100 0111 = 7	**(-4) + (-1)** 1100 +1111 **1** 1011 = -5
(+5) + (+4) 0101 +0100 1001 = overflow	**(-7) + (-6)** 1001 +1010 **1** 0011 =overflow

Integer Arithmetic with **twos complement numbers** - Subtraction (Steps)

- Subtraction is performed through addition
- To subtract one number (subtrahend) from another (minuend), take the twos complement of the subtrahend and add it to the minuend.

i.e. **(M – S) = M + (-S)**

- Apply addition rules including the overflow rule
- We need only addition and complement circuits.

Integer Arithmetic with **twos complement numbers** - Subtraction (Examples)

```
 0010
+1001
 1011   = -5
 M = 2 = 0010
 S = 7 = 0111
-S =     1001
```

```
  0101
 +1110
1 0011   = 3
 M = 5 = 0101
 S = 2 = 0010
-S =     1110
```

(M – S) = M + (-S)

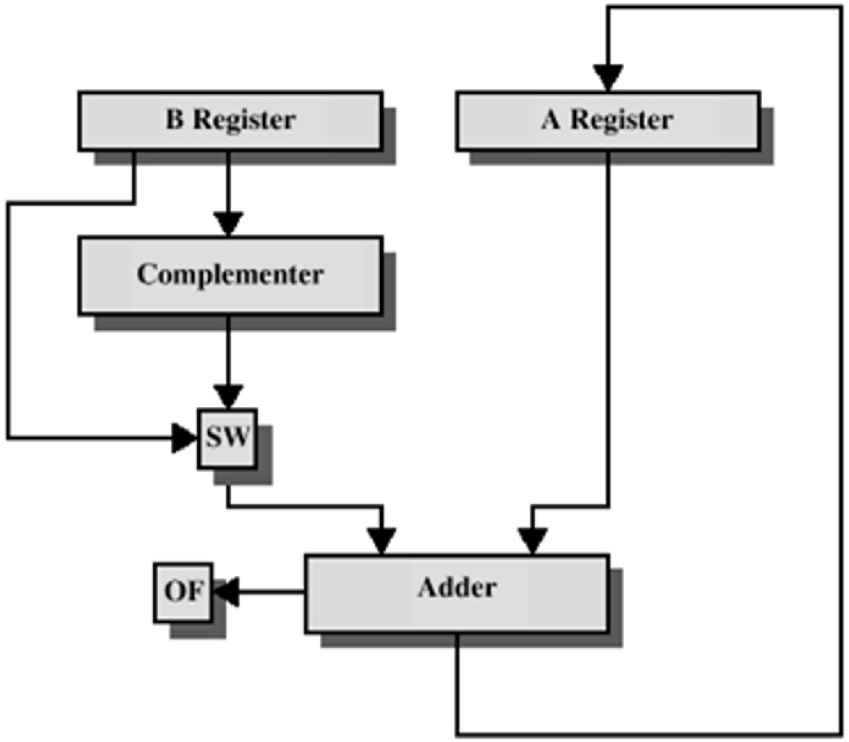
```
  1011
 +1110
1 1001   = -7
 M = -5 = 1011
 S = 2  = 0010
-S =      1110
```

```
 0101
+0010
 0111   = 7
 M = 5  = 0101
 S = -2 = 1110
-S =      0010
```

```
 0111
+0111
 1110   = overflow
 M = 7  = 0111
 S = -7 = 1001
-S =      0111
```

```
  1010
 +1100
1 0110   = overflow
 M = -6 = 1010
 S =  4 = 0100
-S =      1100
```

Integer Arithmetic with **twos complement numbers**
- Hardware for Addition and Subtraction



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

Integer Arithmetic with **twos complement numbers**

- Hardware for Addition and Subtraction

- For **binary addition**,
  - ✓ Two numbers are presented to the adder from two registers, designated as **A** and **B**.
  - ✓ The result may be stored in one of these registers or in a third register.
  - ✓ The overflow indication is stored in a 1-bit overflow flag
    - 0 indicates no overflow while 1 indicates overflow

- For **binary subtraction**,
  - ✓Subtrahend in B register is passed through a twos complementer circuit
  - A register contents and Twos complement of B register contents are fed to the adder

- Control signals are used to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

Problem (1)

- Find the following differences of the given **twos complement numbers.**
  - a. (111000) – (110011)
  - b. (11001100) – (101110)
  - c. (111100001111) – (110011110011)
  - d. (11000011) – (11101000)

Problem (2)

- Assume the following decimal numbers are to be represented in **8-bit twos complement** representation. Perform the following arithmetic. Check whether there is an overflow.
  - a. (85)+(65)
  - b. (-75)+(67)
  - c. (127) – (45)
  - d. (-128) – (-100)

Problem (3)

- What is the range of numbers that can be represented in **8-bit** and **16-bit ones complement** numbers
- Perform **addition/subtraction of 8 bit numbers using ones complement representation** using decimal numbers given below:

a. (6) + (13)

b. (-6) - (13)

c. (-94) + (65)

d. (120) – (89)

Data Representation and Arithmetic – Part 3 of 4

Multiplication algorithms for Integers

TOPIC COVERAGE - PART 3 OF 4

- Multiplication of unsigned binary integers
  - Block diagram, Flow Chart, Examples
- Multiplication of twos complement numbers
  - Booth's algorithm – Flow Chart, Examples

Multiplication of Unsigned binary integers (1)…

- Multiplication is a complex operation, whether performed in hardware or software
- Consider the multiplication of unsigned (nonnegative) binary integers using paper and pencil method
  - Partial products are generated, one for each digit in the multiplier.
    - When the multiplier bit is 0, the partial product is 0.
    - When the multiplier bit is 1, the partial product is multiplicand
  - Partial products are summed to produce the final product
    - Each successive partial product is shifted one position to the left
  - The multiplication of **two n-bit** binary integers results in a product of **up to 2n** bits in length.
    - Example: 4 bits multiplicand X 4 bits multiplier results in the product with maximum of 8 bits in length

Multiplication of Unsigned binary integers (2)…
- Example-paper and pencil approach

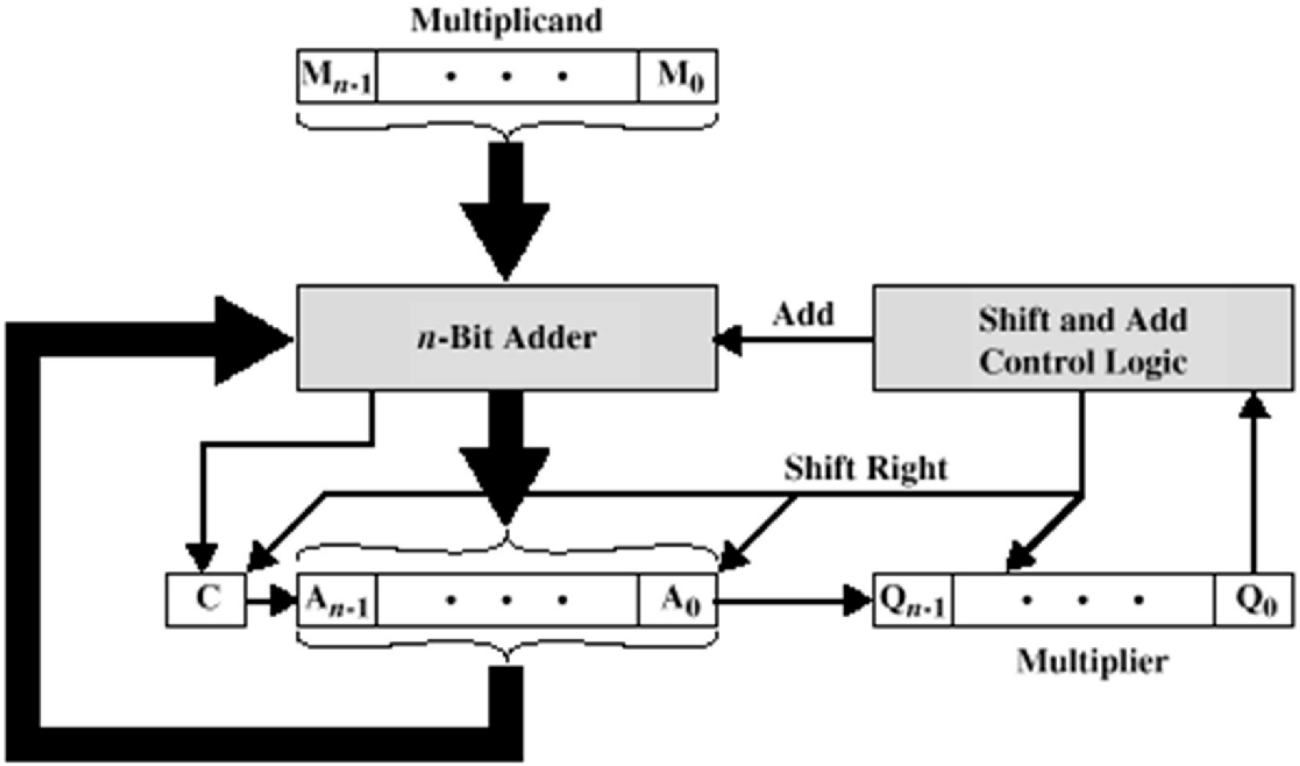
```
      1011   Multiplicand (= 11 Dec.)
    x 1101   Multiplier     (= 13 Dec.)
    ------
      1011   Partial products
     0000
    1011
   1011
   --------
  10001111   Product (143 in decimal)
  --------
```

- Note: If multiplier bit is 1, copy multiplicand (place value) otherwise zero
- Note: need double length result

Multiplication of Unsigned binary integers (3)… - Implementation

- To make the operation more efficient in computers,
  - Perform running addition on the partial products rather than waiting until the end
    - Eliminates the need for storage of all partial products.
    - Fewer registers are required.
  - For each 1 on the multiplier, perform add and shift operation. For each 0, perform only shift operation
    - Save time on the generation of partial products

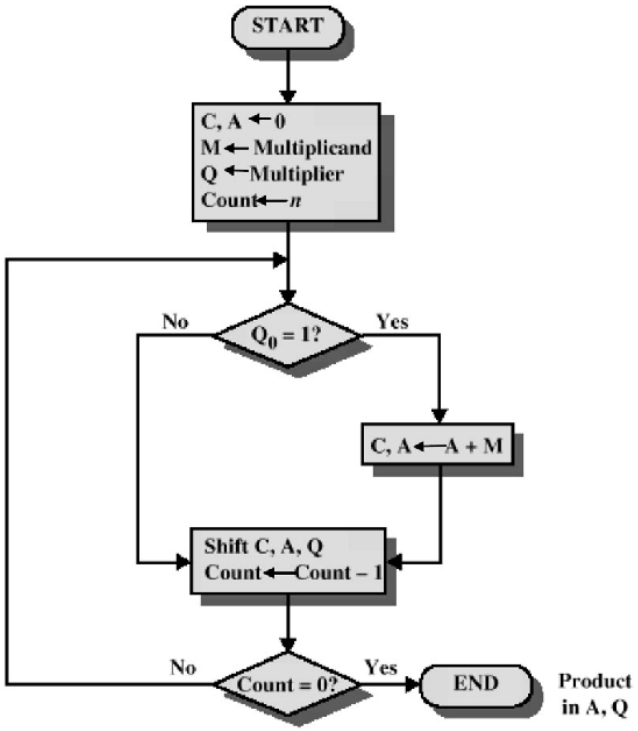
Multiplication of Unsigned binary integers (4)…
- Block Diagram



Multiplication of Unsigned binary integers (5)…
- Operation

- Control logic reads the bits of the multiplier one at a time
  - If $Q_0=1$,
    - Add the multiplicand to A register and store the result in A register and store the carry, if any in C register.
    - Shift the bits of C,A , and Q registers to the right one bit.
  - If $Q_0=0$,
    - Perform only shifting of the bits of C,A, and Q registers to the right one bit
- Repeat this process for each bit of the original multiplier
- The final 2n-bit product is stored in A and Q registers.

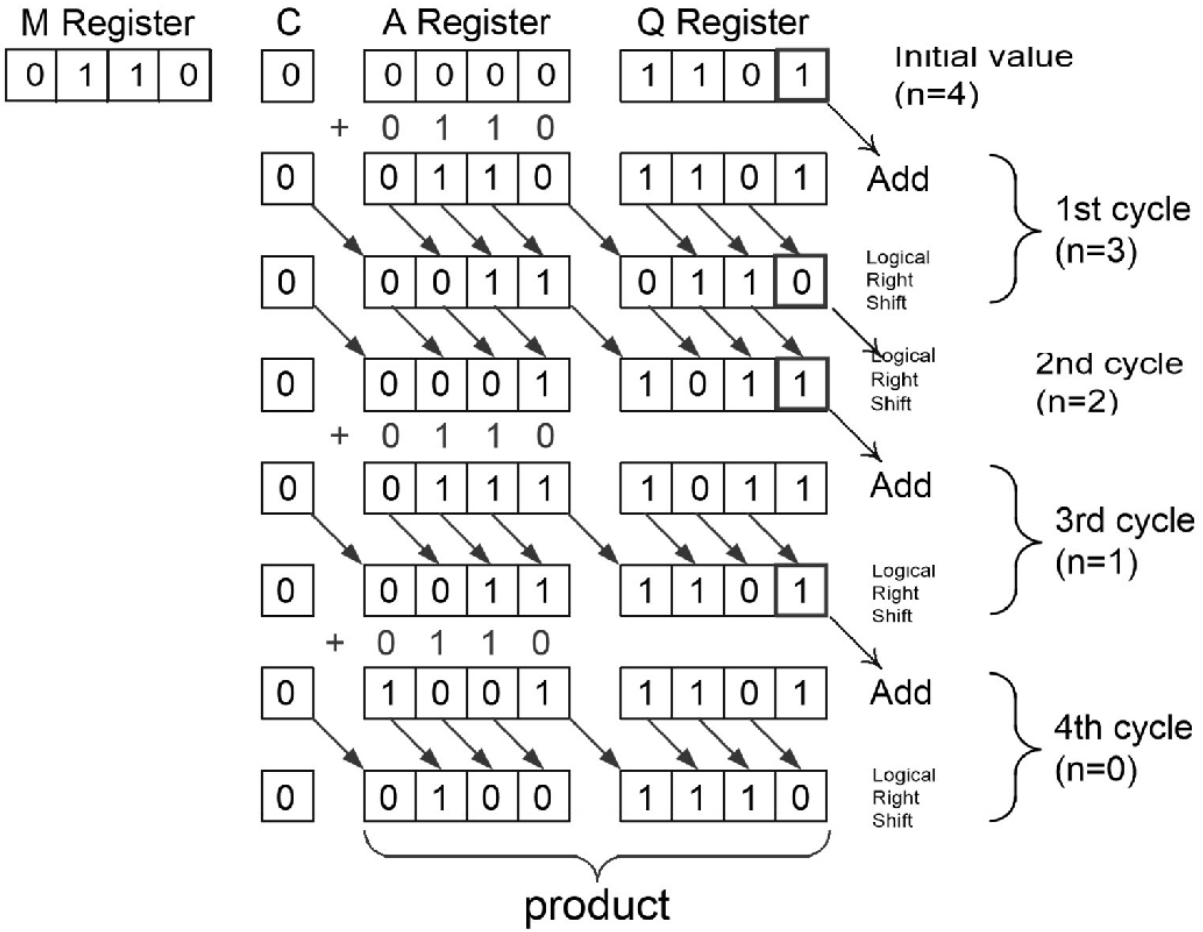
Multiplication of Unsigned binary integers (6)… - Flow Chart



- ✓ Multiplier in Q register
- ✓Multiplicand in M register
- ✓Product in A and Q registers
  - A register – Initially Zero
- ✓ 1 bit C register – To store carry out of addition
  - Initially zero

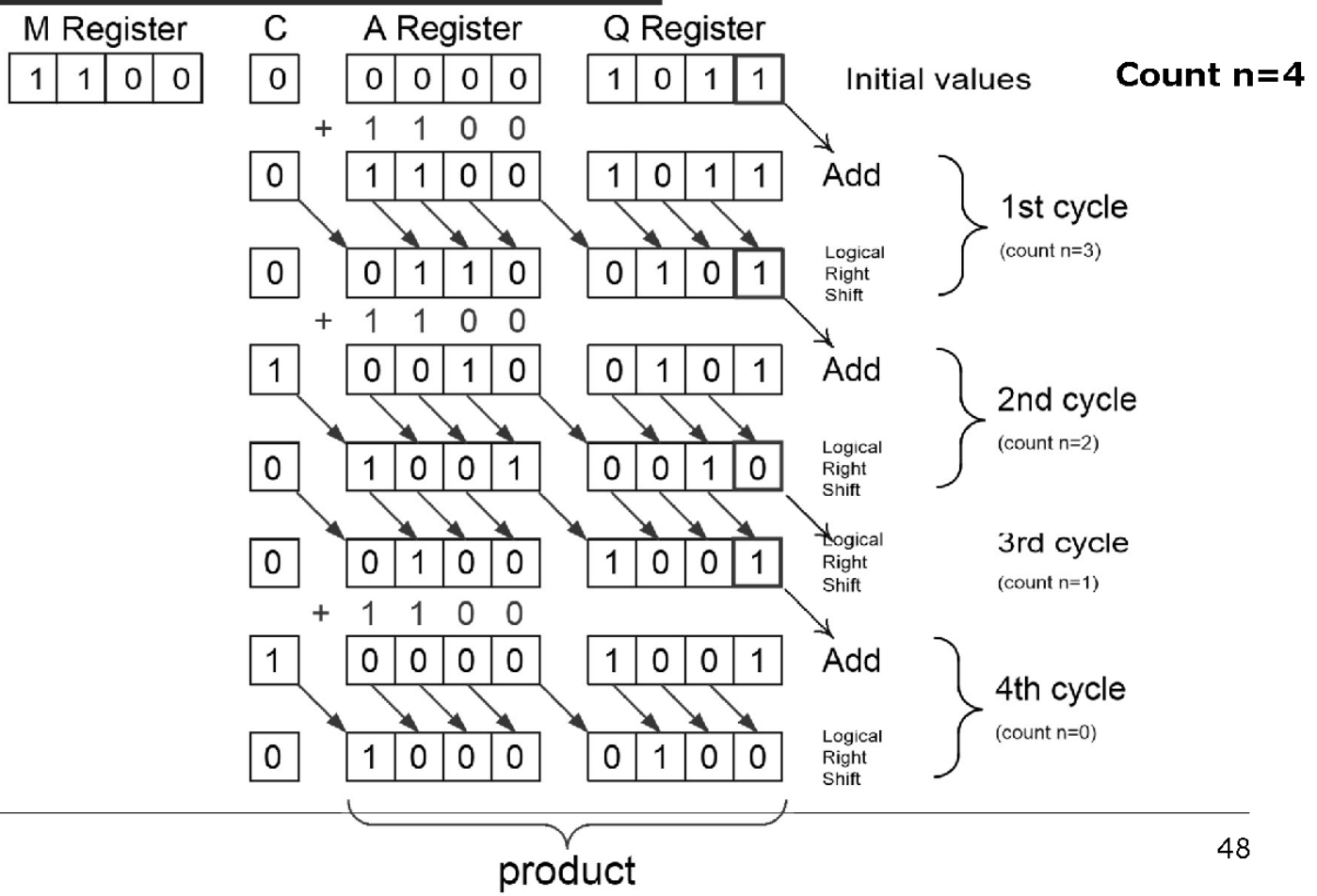
Multiplication of Unsigned binary integers (7)… - Example 1

Perform multiplication of unsigned binary integers, $(0110)_2 \times (1101)_2$



Multiplication of Unsigned binary integers (8) - Example 2

Perform multiplication of unsigned binary integers, $(1100)_2 \times (1011)_2$



Multiplication of Twos complement integers (1)…

- **Solution 1**
  - Convert both multiplier and multiplicand to positive numbers if required
  - Multiply as in unsigned binary
  - If signs of the operands are different, negate the answer (finding 2s complement of the result)
- **Solution 2**
  - Booth's algorithm
  - Performs fewer additions and subtractions than a more straightforward algorithm

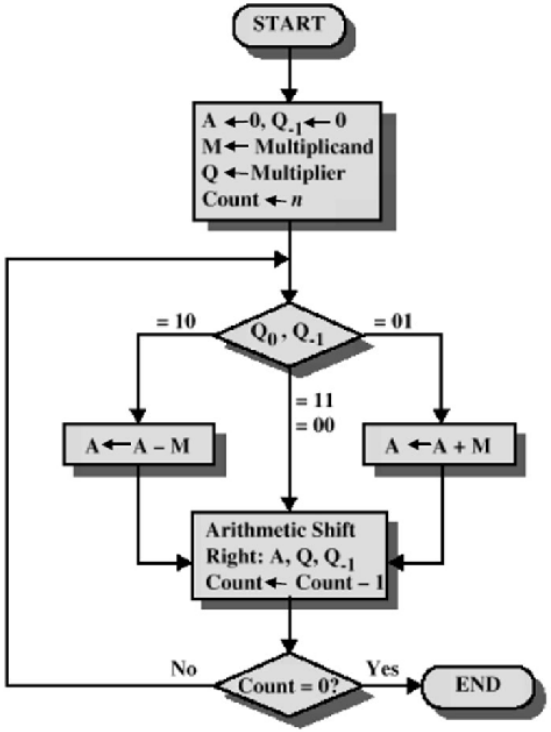
Multiplication of Twos complement integers (2)… - Solution 1

- Convert both multiplier and multiplicand to positive numbers, then perform multiplication and negate the product if the original numbers have different sign .
  Example: Perform multiplication with twos complement numbers **(7)× (-3)**

						0	1	1	1	(7)		M									
					X	0	0	1	1	(3)		Q									
		0		0	0	0	1	1	1			0111	X		1	X	2^0				
		0		0	0	1	1	1	0			0111	X		1	X	2^1				
		0		0	0	0	0	0	0			0111	X		0	X	2^2				
	+	0		0	0	0	0	0	0			0111	X		0	X	2^3				
		0		0	0	1	0	1	0	1	---->	negate	1	1	1	0	1	0	1	1	(-21)
													-128	64	32		8		2	1	

- This method is tedious as it involves checking the sign of the numbers and perform negation if necessary

Multiplication of Twos complement integers (3)… - Solution 2 (Booth Algorithm – Flow Chart)



- ✓Multiplicand in M register
- ✓Multiplier in Q register
- ✓Product in A and Q registers
- ✓1 bit register Q_{-1} placed logically to the right of LSB (Q_0) of Q register
- ✓ Initialize A and Q_{-1} to zero.

Multiplication of Twos complement integers (4)… - Solution 2 (Booth Algorithm - Operation)

- Scan the LSB of Q register (Q_0) and right of the bit (Q_{-1}) at the same time using control logic
- If two bits

(Q_0Q_{-1})=00 =11	Arithmetic right shift only	(A,Q,Q_{-1})
=01	A ←	A +M and Arithmetic right shift
=10	A ←	A - M and Arithmetic right shift

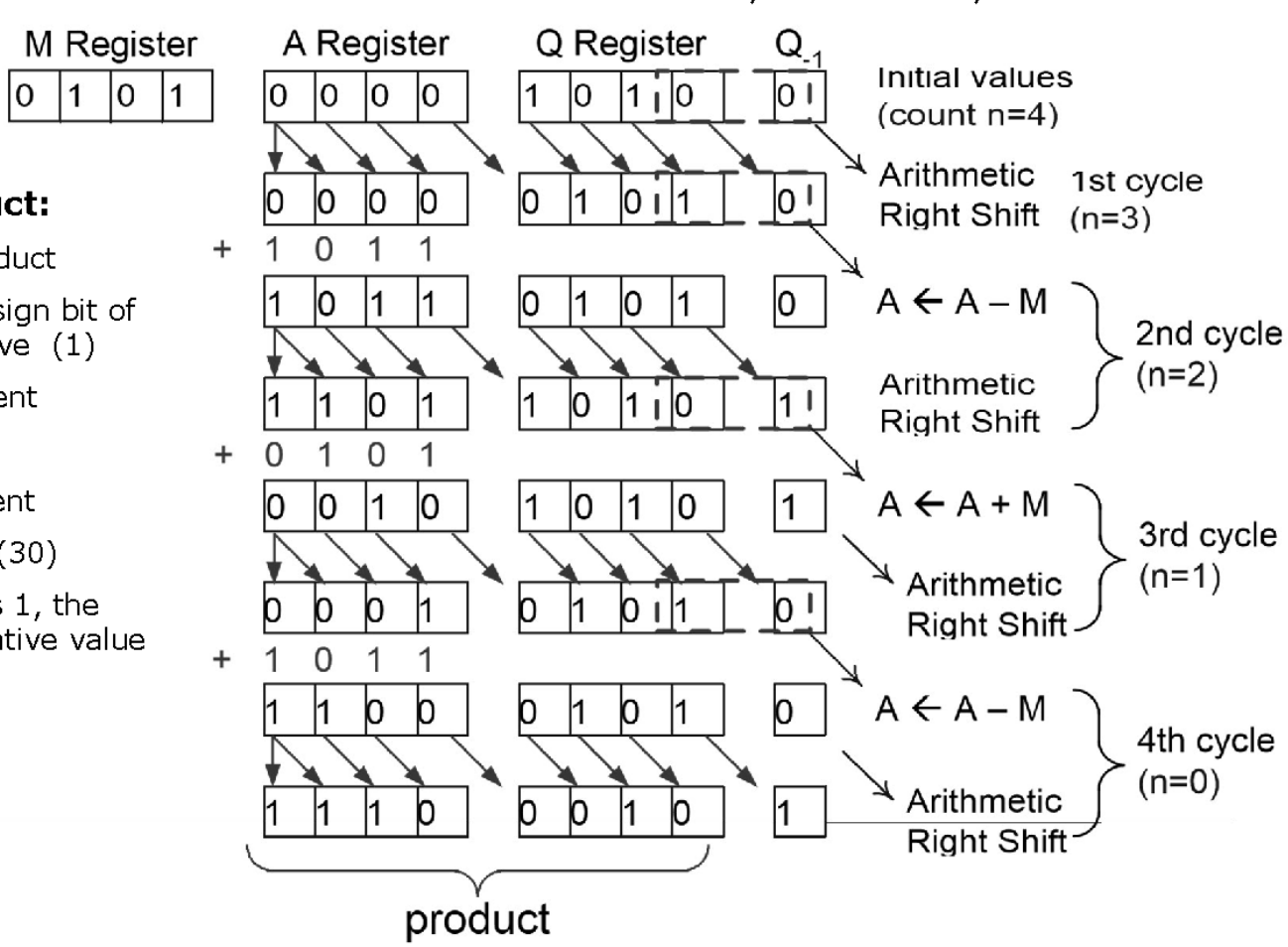
- To preserve the sign of the number in A and Q, arithmetic shift is done (A_{n-1} not only shifted into A_{n-2} but also remains in A_{n-1})

Multiplication of Twos complement integers (5)… - Solution 2 (Booth Algorithm – Example 1)

Consider the multiplication of **(5) x (-6)**, both represented in 4-bit twos complement notation, to produce an 8-bit product

Finding the Product:

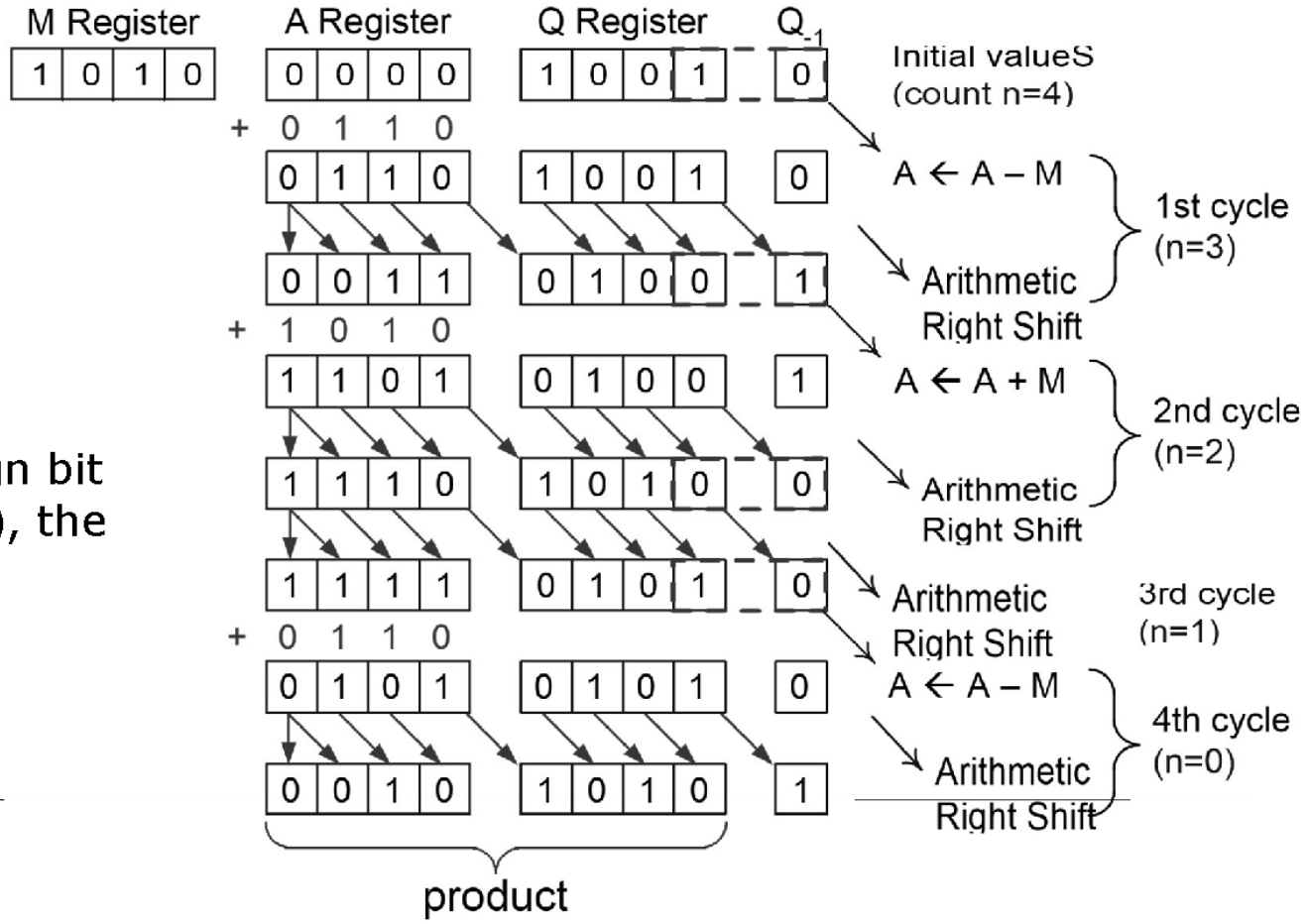
- ✓Negate the product (**1110 0010**) if sign bit of product is negative (1)
- ✓Ones complement = 0001 1101
- ✓Twos complement =00011110 (30)
- ✓Since sign bit is 1, the product is a negative value **(-30)**



Multiplication of Twos complement integers (6) - Solution 2 (Booth Algorithm – Example 2)

Consider the multiplication of **(-6) x (-7)**, both represented in 4-bit twos complement notation, to produce an 8-bit product

M=1010, Q=1001 , - M = 0110



✓Product = **00101010**

✓Since the sign bit is positive (0), the product is **42**

Problems

Problem 1:

Perform multiplication of unsigned binary integers,

$$(1011)_2 \times (1101)_2$$

using **unsigned multiplication** algorithm.

Problem 2:

Given **x=(0101)** and **y=(1010)** in twos complement notation,
(i.e., x = 5, y = -6),
compute the product **p=x×y** with **Booth's algorithm**

Data Representation and Arithmetic – Part 4 of 4

Floating Point Representation and Arithmetic

TOPIC COVERAGE - PART 4 OF 4

- Fixed-point representation
  - Drawbacks
- Floating-point Representation
  - Principles
  - IEEE 754 standard format
    - Single Precision and Double Precision
    - Decimal to IEEE754 standard format conversion
    - IEEE754 standard format to decimal equivalent conversion
    - Range of floating-point numbers
- Arithmetic with Floating Point numbers
  - Floating-Point Addition/Subtraction
  - Floating-Point Multiplication/Division
  - Guard bits/Rounding

Fixed-Point Representation - Drawbacks

- Using Fixed-point representation (ex: Twos complement),
  - Possible to represent a range of positive and negative integers centered on 0
  - Numbers with a fractional component can be represented by assuming a fixed radix point.
- Limitations of fixed-point representation
  - Very large numbers and very small fractions cannot be represented
  - Fractional part of the quotient in a division of two large numbers could be lost
- How to represent very large and very small numbers with only a few digits?
  - Dynamically slide the radix point to a convenient location and use the exponent part to keep track of the radix point – **Floating Point representation**

Floating-Point Representation - Principles

- **For decimal numbers**
  - Use scientific notation
  - Example 1: 976,000,000,000,000 can be represented as 9.76×10^{14}
  - Example 2: 0.0000 0000 0000 0976 can be represented as 9.76×10^{-14}
  - Dynamically slide the decimal point to a convenient location and use the exponent of 10 to keep track of the decimal point.

- **For binary numbers**
  - Represent the number in the form

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

    Store the number with three fields:
    - Sign: plus (0) or minus (1)
    - Mantissa M (or Significand or Fraction)
    - Exponent E
  - Base B is implicit and need not be stored
  - Assumption: Radix point is to the right of MSB of significand

IEEE Standard 754 Floating-Point Format (1)…

- Standard for floating point storage defined in IEEE 754, adopted in 1985
- To facilitate portability of programs from one processor to another
- Widely adopted and used on all processors and arithmetic coprocessors
- Defines 32-bit (single-precision) and 64-bit (double precision) standards
  - 8-bit and 11-bit exponents respectively

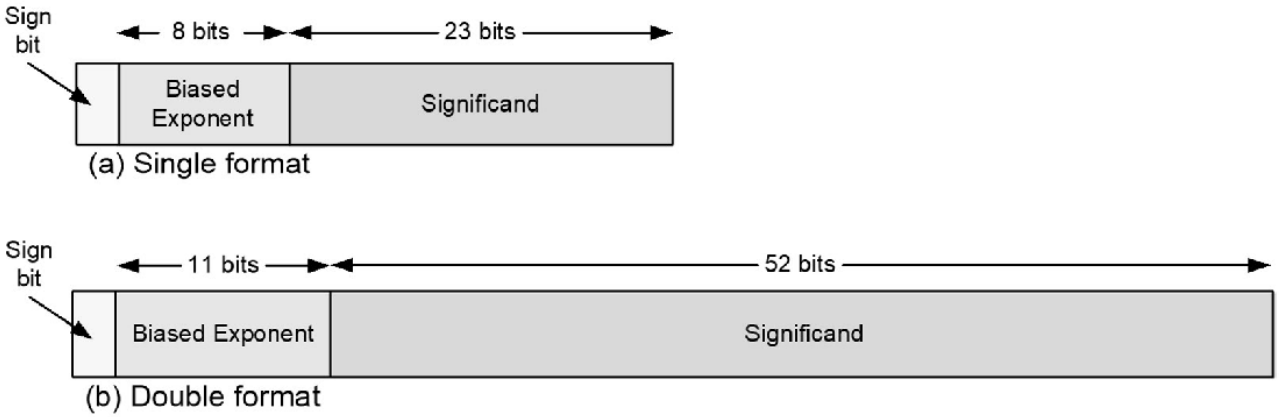
IEEE Standard 754 Floating-Point Format (2)…

- In this format, the numbers are normalized so that the significand lie in the range $1 \leq F < 2$
- An IEEE format floating-point number X is formally defined as:

$$X = -1^{S} \text{ x } 2^{E-B} \text{ x } 1.M$$

where S = sign bit [0→+, 1→–]
E = exponent biased by B
M = fractional Mantissa or significand

IEEE Standard 754 Floating-Point Format (3)…



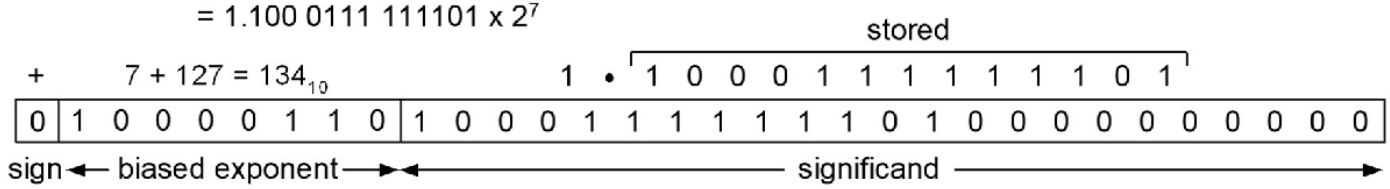
- A sign-magnitude representation has been adopted for the significand
- Significand is negative if S =1, and positive if S =0

IEEE Standard 754 Floating-Point Format (4)… - Example 1

Convert the given numbers to IEEE single precision format:

(a) $199.953125_{10} = 1100\ 0111.111101_2$

$= 1.100\ 0111\ 111101 \times 2^7$

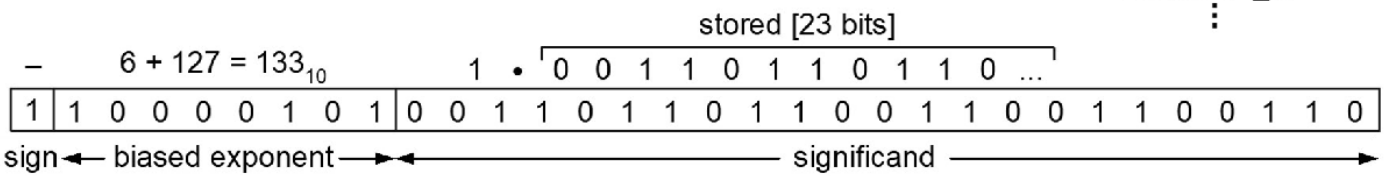


(b) $-77.7_{10} = -100\ 1101.10110\ 0110_2$ ...

$= -1.00\ 1101\ 101100110$ ... $\times 2^6$

$77_{10} = 100\ 1101_2$

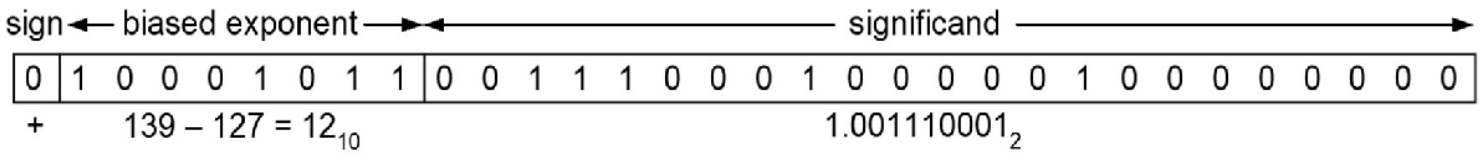
$0.7_{10} \Rightarrow$ 0.7 x 2 → 1.4
0.4 x 2 → 0.8
0.8 x 2 → 1.6
0.6 x 2 → 1.2
0.2 x 2 → 0.4
0.4 x 2 → 0.8
0.8 x 2 → 1.6
0.6 x 2 → 1.2
0.2 x 2 → 0.4
⋮



IEEE Standard 754 Floating-Point Format (5) - Example 2

Convert the given IEEE single precision floating-point numbers to their decimal equivalent:

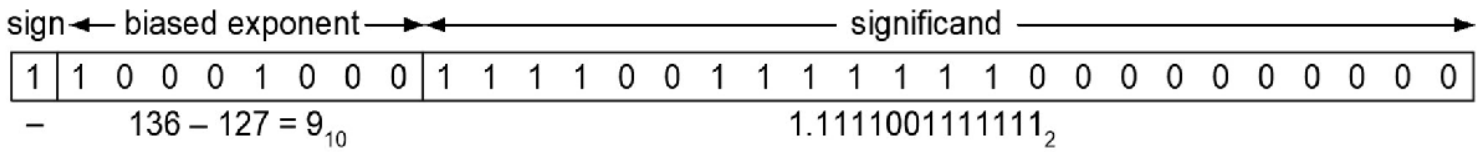
(a) $0100\ 0101\ 1001\ 1100\ 0100\ 0001\ 0000\ 0000_2$



$1.001110001000001_2 \times 2^{12} = 1001110001000.001_2$

$= 5000.125_{10}$

(b) $1100\ 0100\ 0111\ 1001\ 1111\ 1100\ 0000\ 0000_2$



$-1.1111001111111_2 \times 2^9 = -1111100111.1111_2$

$= -999.9375_{10}$

PROBLEM

- Express the number $(-640.5)_{10}$ in IEEE 32-bit and 64-bit floating point format

SOLUTION (1)…

- **IEEE 32 BIT FLOATING POINT FORMAT**

MSB 8 bits 23 bits

sign	Biased Exponent	Mantissa/Significand (Normalized)

Step 1: Express the given number in binary form

$(640.5) = 1010000000.1 * 2^0$

Step 2: Normalize the number into the form 1.bbbbbbb

$1010000000.1 * 2^0 = 1.0100000001 * 2^9$

Once Normalized, every number will have 1 at the leftmost bit. So IEEE notation is saying that there is no need to store this bit. Therefore significand to be stored is 0100 0000 0100 0000 0000 000 in the allotted 23 bits

SOLUTION (2)…

- **Step 3:** For the 8 bit biased exponent field, the bias used is
  $2^{k-1}-1 = 2^{8-1}-1 = 127$
  Add the bias 127 to the exponent 9 and convert it into binary in order to store for 8-bit biased exponent.
  127 + 9 =136 (1000 1000)
- **Step 4:** Since the given number is negative, put MSB as 1
- **Step 5:** Pack the result into proper format (IEEE 32 bit)

1	1000 1000	0100 0000 0100 0000 0000 000

SOLUTION (3)…

- **IEEE 64 BIT FLOATING POINT FORMAT**

MSB 11 bits 52 bits

sign	Biased Exponent	Mantissa/Significand (Normalized)

Step 1: Express the given number in binary form

(640.5) = $1010000000.1 * 2^0$

Step 2: Normalize the number into the form 1.bbbbbbb

$1010000000.1 * 2^0 = 1.0100000001 * 2^9$

Once Normalized, every number will have 1 at the leftmost bit. So IEEE notation is saying that there is no need to store this bit. Therefore significand to be stored is 0100 0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 in the allotted 52 bits

SOLUTION (4)

- **Step 3:** For the 11 bit biased exponent field, the bias used is
  $2^{k-1}-1 = 2^{11-1}-1 = 1023$
  Add the bias 1023 to the exponent 9 and convert it into binary in order to store for 11-bit biased exponent. 1023 + 9 =1032 (1000 0001 000)
- **Step 4:** Since the given number is negative, put MSB as 1
- **Step 5:** Pack the result into proper format(IEEE 64 bit)

1	1000 0001 000	0100 0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Reference

Slides adopted from

Thomas L.Floyd, “Digital Fundamentals,”
11th Edition, Prentice Hall, 2014
(ISBN10:0132737965/ISBN13:9780132737968)