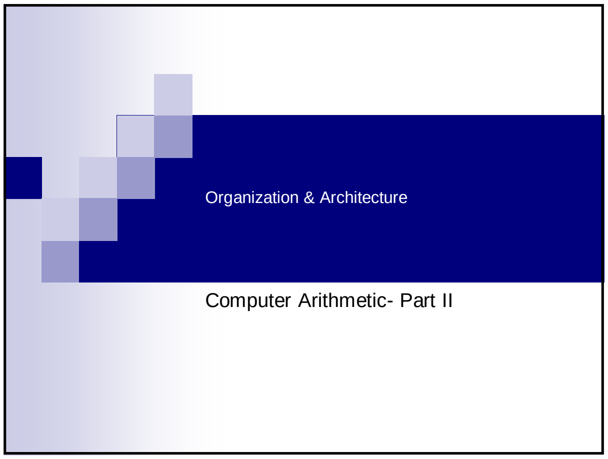
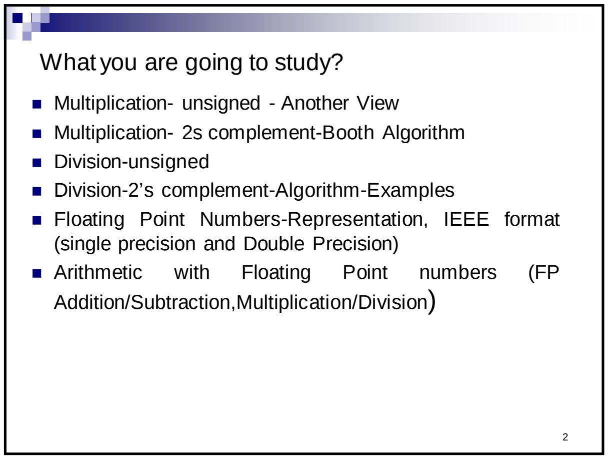


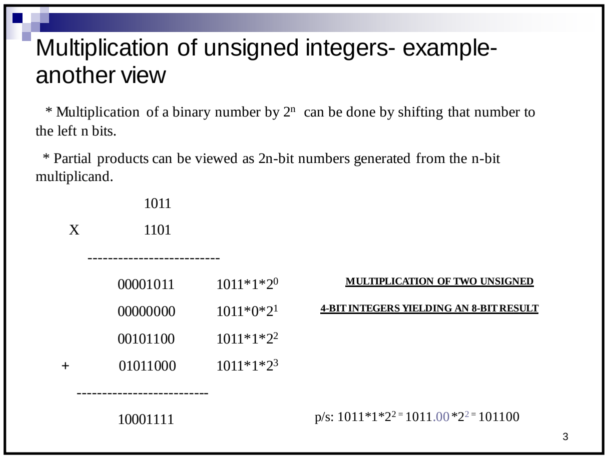
Organization & Architecture

Computer Arithmetic- Part II



What you are going to study?

- Multiplication- unsigned - Another View
- Multiplication- 2s complement-Booth Algorithm
- Division-unsigned
- Division-2's complement-Algorithm-Examples
- Floating Point Numbers-Representation, IEEE format (single precision and Double Precision)
- Arithmetic with Floating Point numbers (FP Addition/Subtraction,Multiplication/Division)



Multiplication of unsigned integers- example-another view

* Multiplication of a binary number by 2^n can be done by shifting that number to the left n bits.

* Partial products can be viewed as 2n-bit numbers generated from the n-bit multiplicand.

1011
X 1101

00001011 $1011*1*2^0$

00000000 $1011*0*2^1$

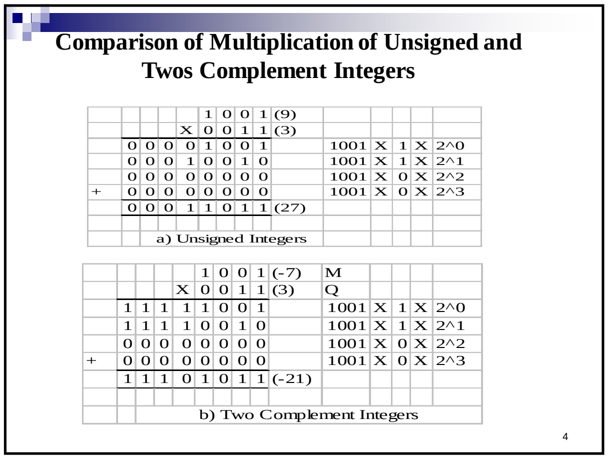
00101100 $1011*1*2^2$

+ 01011000 $1011*1*2^3$

10001111

MULTIPLICATION OF TWO UNSIGNED 4-BIT INTEGERS YIELDING AN 8-BIT RESULT

p/s: $1011*1*2^{2} = 1011.00*2^{2} = 101100$



Comparison of Multiplication of Unsigned and Twos Complement Integers

					1	0	0	1	(9)					
				X	0	0	1	1	(3)					
	0	0	0	0	1	0	0	1		1001	X	1	X	2^0
	0	0	0	1	0	0	1	0		1001	X	1	X	2^1
	0	0	0	0	0	0	0	0		1001	X	0	X	2^2
+	0	0	0	0	0	0	0	0		1001	X	0	X	2^3
	0	0	0	1	1	0	1	1	(27)					
		a) Unsigned Integers												

					1	0	0	1	(-7)	M				
				X	0	0	1	1	(3)	Q				
	1	1	1	1	1	0	0	1		1001	X	1	X	2^0
	1	1	1	1	0	0	1	0		1001	X	1	X	2^1
	0	0	0	0	0	0	0	0		1001	X	0	X	2^2
+	0	0	0	0	0	0	0	0		1001	X	0	X	2^3
	1	1	1	0	1	0	1	1	(-21)					
		b) Two Complement Integers												

Multiplying 2’s complement numbers

- If multiplier (Q) is negative, 7 X –3 , this does not work!

					0	1	1	1	(7)	M				
				X	1	1	0	1	(-3)	Q				
	1	1	1	1	0	1	1	1		0111	X	1	X	2^0
	0	0	0	0	0	0	0	0		0111	X	0	X	2^1
	1	1	0	1	1	1	0	0		0111	X	1	X	2^2
+	1	0	1	1	1	0	0	0		0111	X	1	X	2^3
10	1	0	0	0	1	0	1	1	(-117)					
									X					
		it cannot work when Q is negative												

Multiplying 2’s complement numbers

⌘Solution 1

- Convert both multiplier and multiplicand to positive if required
- Multiply as in unsigned binary
- If signs of the operands are different, negate answer (finding 2s complement of the result)

⌘Solution 2

- Booth’s algorithm-performs fewer additions and subtractions than a more straightforward algorithm

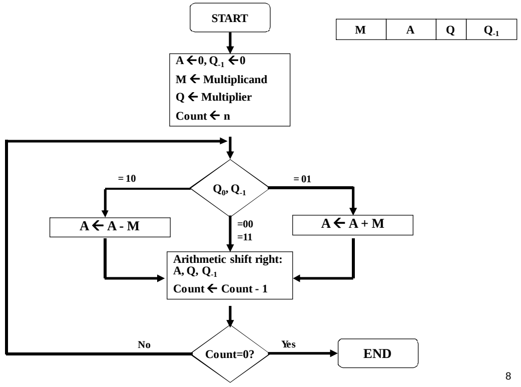
Solution 1

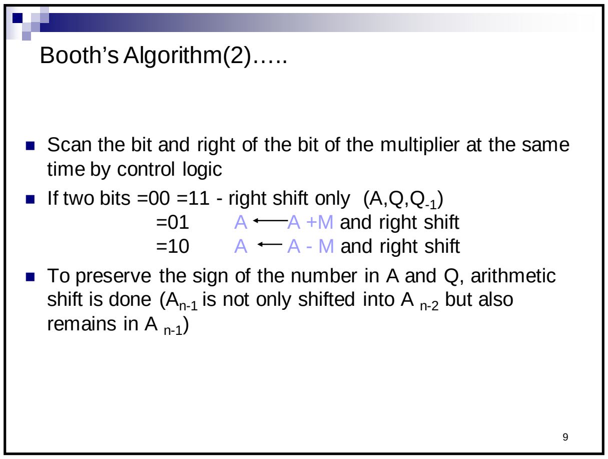
- To overcome this dilemma, first convert both multiplier and multiplicand to positive numbers, then perform multiplication and negate the product if the original numbers have different sign

						0	1	1	1	(7)	M									
					X	0	0	1	1	(3)	Q									
		0	0	0	0	0	1	1	1		0111	X	1	X	2^0					
		0	0	0	0	1	1	1	0		0111	X	1	X	2^1					
		0	0	0	0	0	0	0	0		0111	X	0	X	2^2					
	+	0	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

Solution 2 - Booth’s Algorithm (1)…….

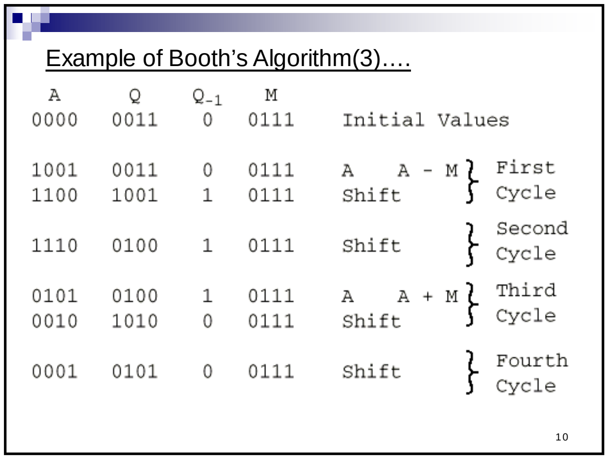




Booth's Algorithm(2)…..

- Scan the bit and right of the bit of the multiplier at the same time by control logic
- If two bits =00 =11 - right shift only (A,Q,Q_{-1})
  =01 A ⟵ A +M and right shift
  =10 A ⟵ A - M and right shift
- To preserve the sign of the number in A and Q, arithmetic shift is done (A_{n-1} is not only shifted into A_{n-2} but also remains in A_{n-1})

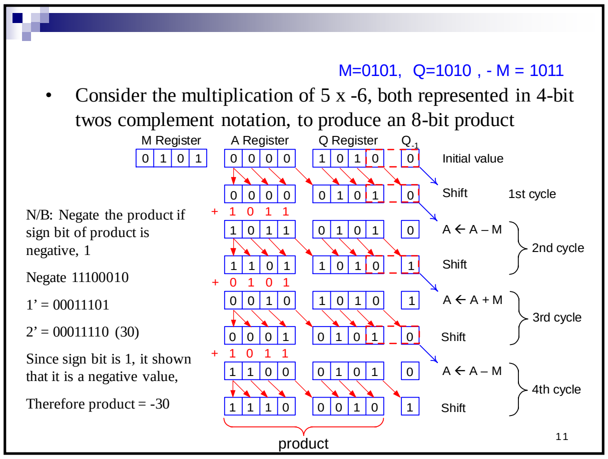
9



Example of Booth's Algorithm(3)….

```
 A      Q     Q-1    M
0000   0011    0    0111      Initial Values

1001   0011    0    0111      A    A - M } First
1100   1001    1    0111      Shift      } Cycle

1110   0100    1    0111      Shift      } Second
                                         } Cycle

0101   0100    1    0111      A    A + M } Third
0010   1010    0    0111      Shift      } Cycle

0001   0101    0    0111      Shift      } Fourth
                                         } Cycle
```

10



M=0101, Q=1010 , - M = 1011

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

```
M Register   A Register   Q Register   Q-1
0 1 0 1      0 0 0 0      1 0 1 0      0     Initial value

             0 0 0 0      0 1 0 1      0     Shift        1st cycle
           + 1 0 1 1
             1 0 1 1      0 1 0 1      0     A ← A – M  }
                                                        } 2nd cycle
             1 1 0 1      1 0 1 0      1     Shift      }
           + 0 1 0 1
             0 0 1 0      1 0 1 0      1     A ← A + M  }
                                                        } 3rd cycle
             0 0 0 1      0 1 0 1      0     Shift      }
           + 1 0 1 1
             1 1 0 0      0 1 0 1      0     A ← A – M  }
                                                        } 4th cycle
             1 1 1 0      0 0 1 0      1     Shift      }
             \________product________/
```

N/B: Negate the product if sign bit of product is negative, 1

Negate 11100010

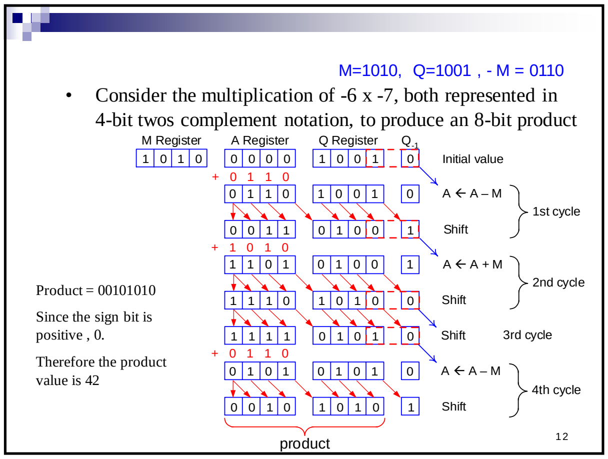
1' = 00011101

2' = 00011110 (30)

Since sign bit is 1, it shown that it is a negative value,

Therefore product = -30

11



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

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

```
M Register   A Register   Q Register   Q-1
1 0 1 0      0 0 0 0      1 0 0 1      0     Initial value
           + 0 1 1 0
             0 1 1 0      1 0 0 1      0     A ← A – M  }
                                                        } 1st cycle
             0 0 1 1      0 1 0 0      1     Shift      }
           + 1 0 1 0
             1 1 0 1      0 1 0 0      1     A ← A + M  }
                                                        } 2nd cycle
             1 1 1 0      1 0 1 0      0     Shift      }

             1 1 1 1      0 1 0 1      0     Shift        3rd cycle
           + 0 1 1 0
             0 1 0 1      0 1 0 1      0     A ← A – M  }
                                                        } 4th cycle
             0 0 1 0      1 0 1 0      1     Shift      }
             \________product________/
```

Product = 00101010

Since the sign bit is positive , 0.

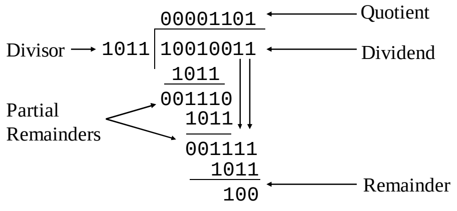
Therefore the product value is 42

12

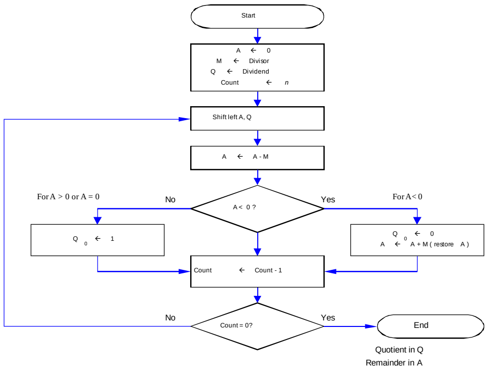
Division

- More complex than multiplication
- General principle is the same as multiplication.
- Operation involves repetitive shifting and add/sub.
- The basis for the algorithm is the paper and pencil approach.

Division of Unsigned Binary Integers

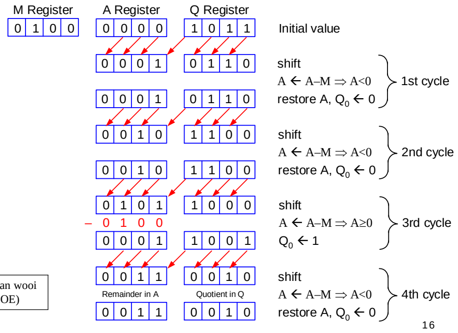


Division of Unsigned Binary Integers



- Consider the the division of two 4-bit unsigned integers: 1011_2 (DIVIDED, 11) ÷ 0100_2(DIVISOR, 4)

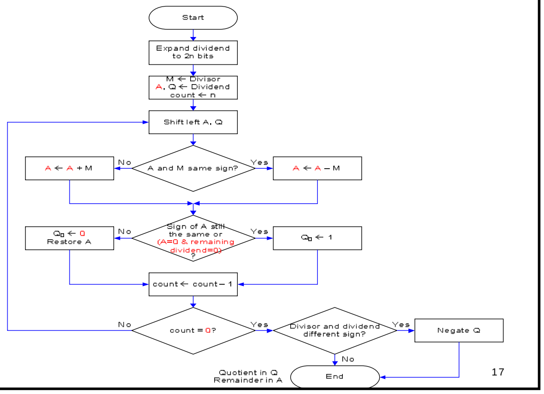
M = divisor , Q = divided



M Register	A Register	Q Register		
0 1 0 0	0 0 0 0	1 0 1 1	Initial value	
	0 0 0 1	0 1 1 0	shift	
			$A \leftarrow A-M \Rightarrow A<0$	1st cycle
	0 0 0 1	0 1 1 0	restore A, $Q_0 \leftarrow 0$	
	0 0 1 0	1 1 0 0	shift	
			$A \leftarrow A-M \Rightarrow A<0$	2nd cycle
	0 0 1 0	1 1 0 0	restore A, $Q_0 \leftarrow 0$	
	0 1 0 1	1 0 0 0	shift	
	– 0 1 0 0		$A \leftarrow A-M \Rightarrow A\geq0$	3rd cycle
	0 0 0 1	1 0 0 1	$Q_0 \leftarrow 1$	
	0 0 1 1	0 0 1 0	shift	
	Remainder in A	Quotient in Q	$A \leftarrow A-M \Rightarrow A<0$	4th cycle
	0 0 1 1	0 0 1 0	restore A, $Q_0 \leftarrow 0$	

Slides adapted from tan wooi haw's lecture notes (FOE)

Twos complement Division - Restoring division approach



Twos complement Division-Algorithm (3)…..

i. Expand dividend to 2n-bit. (For Ex. 4bit 0111 becomes 00000111, and 1001 becomes 11111001.
ii. Load divisor in M and dividend in A & Q.
iii. Shift left A & Q by 1 bit
iv. If M and A have the same sign, perform A←A–M, otherwise A←A+M
v. If the sign of A is the same before and after the operation or (A=0 & remaining dividend=0), set Q_0=1
vi. Otherwise, if the sign is different and (A≠0 or remaining dividend≠0), set Q_0=0 and restore A
vii. Negate Q if divisor and dividend have different sign
viii. Remainder in A, quotient in Q

What is remaining dividend = 0 ?
for example, dividend is 1100
If shift to left by 1 bit: 1000
so now the remaining dividend is 100
If shift to left again becomes: 0000
now the remaining dividend has become 00, which means remaining dividend is 0

Slides adapted from tan wooi haw's lecture notes (FOE)

continue …

Twos complement Restoring Division

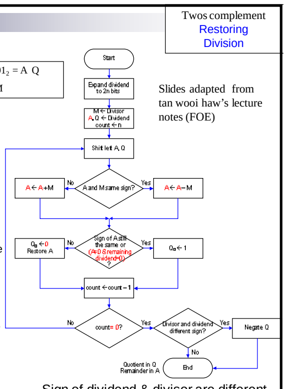
Example 4.22: -7 ÷ 2

$-7 = 1111\ 1001_2$ = A Q
2 = 0010 = M

Slides adapted from tan wooi haw's lecture notes (FOE)

M = 0010

A	Q		
1 1 1 1	1 0 0 1	Initial values	
1 1 1 1	0 0 1 0	Shift	1st cycle
+ 0 0 1 0		Add	
0 0 0 1			
1 1 1 1	0 0 1 0	Restore	
1 1 1 0	0 1 0 0	Shift	2nd cycle
+ 0 0 1 0		Add	
0 0 0 0			
1 1 1 0	0 1 0 0	Restore	
1 1 0 0	1 0 0 0	Shift	3rd cycle
+ 0 0 1 0		Add	
1 1 1 0			
1 1 1 0	1 0 0 1	Set Q_0 = 1	
1 1 0 1	0 0 1 0	Shift	4th cycle
+ 0 0 1 0		Add	
1 1 1 1			
1 1 1 1	0 0 1 1	Set Q_0 = 1	



Sign of dividend & divisor are different
→ negate Q
∴ Quotient = -Q $=1101_2 = -3_{10}$
Remainder $= 1111_2 = -1_{10}$

continue ...

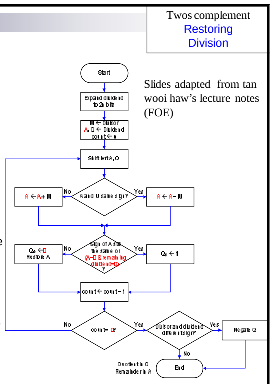
Twos complement Restoring Division

Example 4.23: 6 ÷ -3

Slides adapted from tan wooi haw's lecture notes (FOE)

M = 1101

A	Q		
0 0 0 0	0 1 1 0	Initial values	
0 0 0 0	1 1 0 0	Shift	1st cycle
+ 1 1 0 1		Add	
1 1 0 1			
0 0 0 0	1 1 0 0	Restore	
0 0 0 1	1 0 0 0	Shift	2nd cycle
+ 1 1 0 1		Add	
1 1 1 0			
0 0 0 1	1 0 0 0	Restore	
0 0 1 1	0 0 0 0	Shift	3rd cycle
+ 1 1 0 1		Add	
0 0 0 0			
0 0 0 0	0 0 0 1	Set Q_0 = 1	
0 0 0 0	0 0 1 0	Shift	4th cycle
+ 1 1 0 1		Add	
1 1 0 1			
0 0 0 0	0 0 1 0	Restore	



Sign of dividend & divisor are differrent → negate Q
∴ Quotient = -Q $=1110_2 = -2_{10}$
Remainder = 0

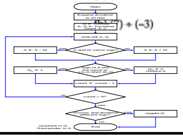
Twos complement Division-Examples (1)…..

A	Q	M = 0011
0000	0111	Initial Value
0000	1110	Shift
1101		Subtract
0000	1110	Restore
0001	1100	Shift
1110		Subtract
0001	1100	Restore
0011	1000	Shift
0000		Subtract
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Subtract
0001	0010	Restore

(a) (7) + (3)

A	Q	M = 1101
0000	0111	Initial Value
0000	1110	Shift
1101		Add
0000	1110	Restore
0001	1100	Shift
1110		Add
0001	1100	Restore
0011	1000	Shift
0000		Add
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Add
0001	0010	Restore

(b) (7) + (−3)



B repres ents Q

From reference book – not valid if - 6 divide by 2 , where quotient of -2 and remainder of -2

Twos complement Division-Examples (2)….

A	Q	M = 0011
1111	1001	Initial Value
1111	0010	Shift
0010		Add
1111	0010	Restore
1110	0100	Shift
0001		Add
1110	0100	Restore
1100	1000	Shift
1111		Add
1111	1001	Set $Q_0 = 1$
1111	0010	Shift
0010		Add
1111	0010	Restore

(c) (−7) + (3)

A	Q	M = 1101
1111	1001	Initial Value
1111	0010	Shift
0010		Subtract
1111	0010	Restore
1110	0100	Shift
0001		Subtract
1110	0100	Restore
1100	1000	Shift
1111		Subtract
1111	1001	Set $Q_0 = 1$
1111	0010	Shift
0010		Subtract
1111	0010	Restore

(d) (−7) ÷ (−3)

Figure 8.16 2’s complement division examples

From reference book – not valid if - 6 divide by 2

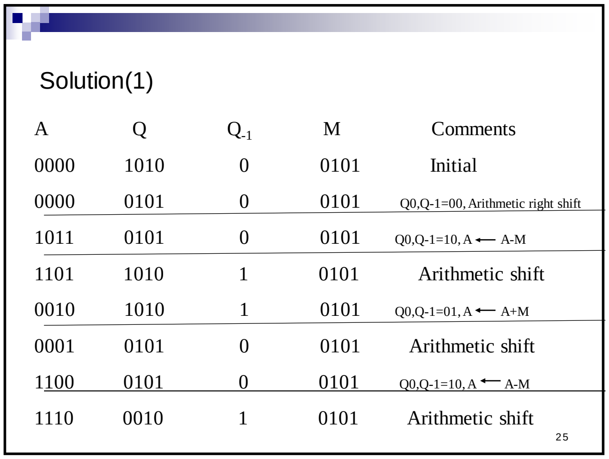
Twos complement Division (3)

- Remainder is defined by
- D=Q*V+R
- D=Dividend, Q=Quotient, V=Divisor, R=Remainder

N/b: find out the remainder of 7/-3 & –7 /3 by using the formula above and check with the slides on page 21, 22. The result of figures from both slides are consistent with the formula

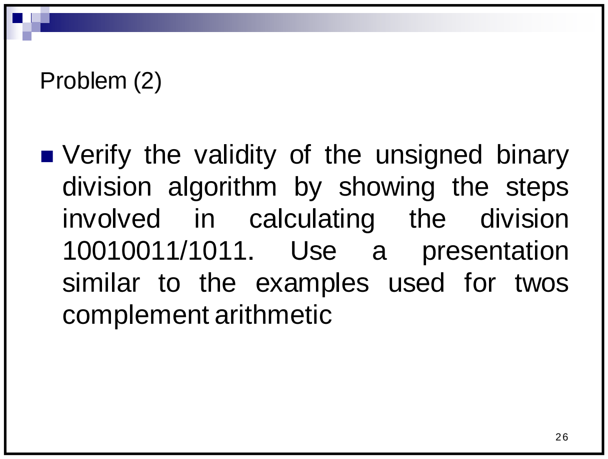
Problem (1)

- 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



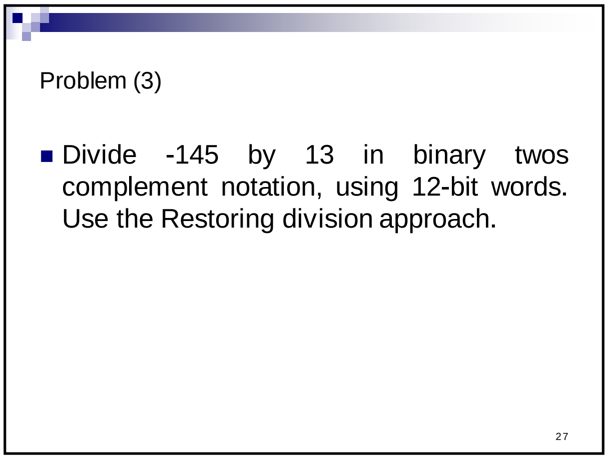
Solution(1)

A	Q	Q_{-1}	M	Comments
0000	1010	0	0101	Initial
0000	0101	0	0101	Q0,Q-1=00, Arithmetic right shift
1011	0101	0	0101	Q0,Q-1=10, A ⟵ A-M
1101	1010	1	0101	Arithmetic shift
0010	1010	1	0101	Q0,Q-1=01, A ⟵ A+M
0001	0101	0	0101	Arithmetic shift
1100	0101	0	0101	Q0,Q-1=10, A ⟵ A-M
1110	0010	1	0101	Arithmetic shift



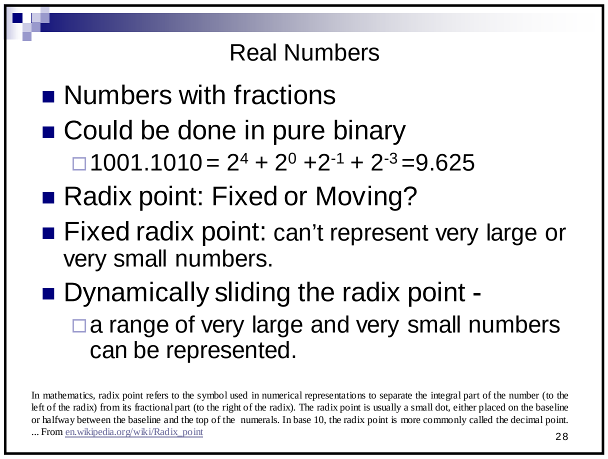
Problem (2)

- Verify the validity of the unsigned binary division algorithm by showing the steps involved in calculating the division 10010011/1011. Use a presentation similar to the examples used for twos complement arithmetic



Problem (3)

- Divide -145 by 13 in binary twos complement notation, using 12-bit words. Use the Restoring division approach.



Real Numbers

- Numbers with fractions
- Could be done in pure binary
  - □ 1001.1010 = $2^4 + 2^0 + 2^{-1} + 2^{-3}$ =9.625
- Radix point: Fixed or Moving?
- Fixed radix point: can't represent very large or very small numbers.
- Dynamically sliding the radix point -
  - □ a range of very large and very small numbers can be represented.

In mathematics, radix point refers to the symbol used in numerical representations to separate the integral part of the number (to the left of the radix) from its fractional part (to the right of the radix). The radix point is usually a small dot, either placed on the baseline or halfway between the baseline and the top of the numerals. In base 10, the radix point is more commonly called the decimal point. ... From en.wikipedia.org/wiki/Radix_point

Floating Point

Sign bit	Biased Exponent	Significand or Mantissa

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

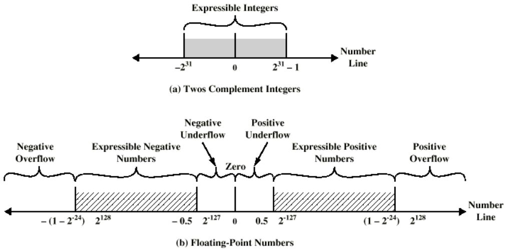
Signs for Floating Point

- Mantissa is stored in 2s compliment.
- Exponent is in excess or biased notation.
- Excess (biased exponent) 128 means
  - □ 8 bit exponent field
  - □ Pure value range 0-255
  - □ Subtract 128 (2^{k-1} - 1)to get correct value
  - □ Range -128 to +127

Normalization

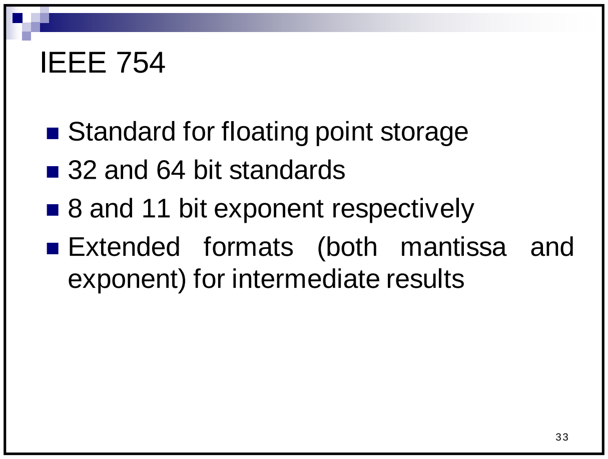
- FP numbers are usually normalized
  - □ exponent is adjusted so that leading bit (MSB) of mantissa is 1
  - □ Since it is always 1 there is no need to store it
  - □ (Scientific notation where numbers are normalized to give a single digit before the decimal point e.g. 3.123 x 10^3)
- In FP representation: not representing more individual values, but spreading the numbers.

Expressible Numbers



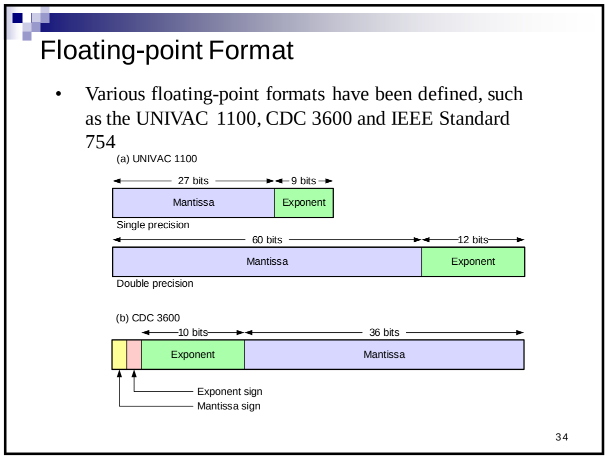
(a) Twos Complement Integers

(b) Floating-Point Numbers



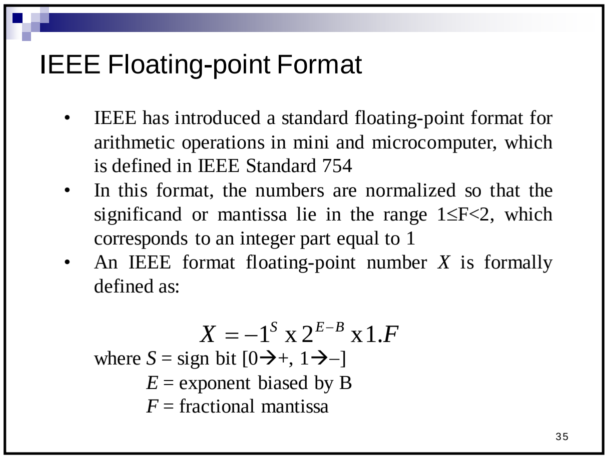
IEEE 754

- 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



Floating-point Format

- Various floating-point formats have been defined, such as the UNIVAC 1100, CDC 3600 and IEEE Standard 754

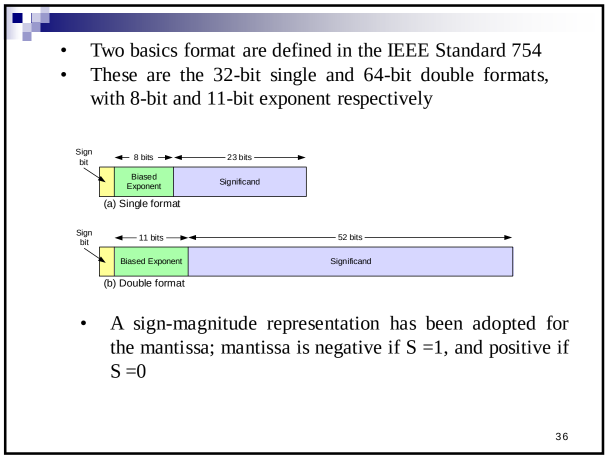


IEEE Floating-point Format

- IEEE has introduced a standard floating-point format for arithmetic operations in mini and microcomputer, which is defined in IEEE Standard 754
- In this format, the numbers are normalized so that the significand or mantissa lie in the range 1≤F<2, which corresponds to an integer part equal to 1
- An IEEE format floating-point number X is formally defined as:

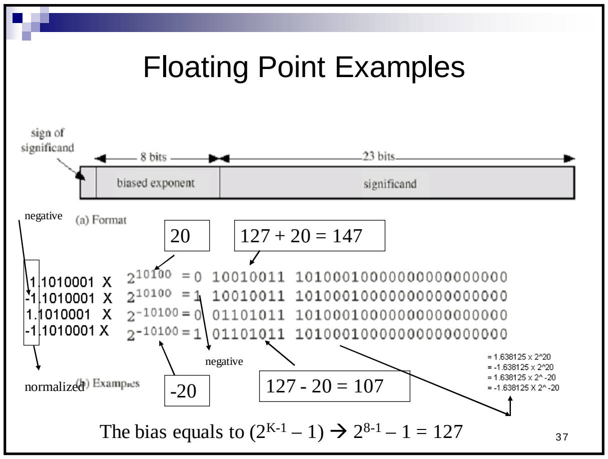
$$X = -1^{S} \text{ x } 2^{E-B} \text{ x} 1.F$$

where S = sign bit [0→+, 1→–]
E = exponent biased by B
F = fractional mantissa



- Two basics format are defined in the IEEE Standard 754
- These are the 32-bit single and 64-bit double formats, with 8-bit and 11-bit exponent respectively

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



Floating Point Examples

sign of significand

8 bits

23 bits

biased exponent

significand

negative

(a) Format

20

127 + 20 = 147

1.1010001 X 2^{10100} = 0 10010011 10100010000000000000000

-1.1010001 X 2^{10100} = 1 10010011 10100010000000000000000

1.1010001 X 2^{-10100} = 0 01101011 10100010000000000000000

-1.1010001 X 2^{-10100} = 1 01101011 10100010000000000000000

negative

= 1.638125 x 2^20
= -1.638125 x 2^20
= 1.638125 x 2^ -20
= -1.638125 X 2^ -20

normalized

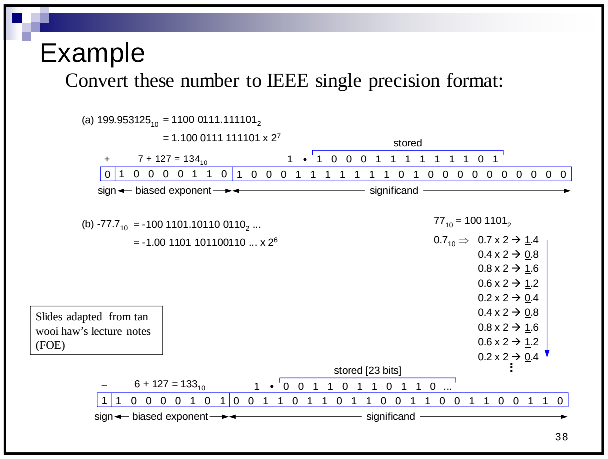
(b) Examples

-20

127 - 20 = 107

The bias equals to $(2^{K-1} - 1) \rightarrow 2^{8-1} - 1 = 127$

37



Example

Convert these number to IEEE single precision format:

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

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

stored

+ $7 + 127 = 134_{10}$ 1 • 1 0 0 0 1 1 1 1 1 1 1 0 1

0 | 1 0 0 0 0 1 1 0 | 1 0 0 0 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0

sign ← biased exponent → ← significand →

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

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

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

$0.7_{10} \Rightarrow$ 0.7 x 2 → $\underline{1}$.4
0.4 x 2 → $\underline{0}$.8
0.8 x 2 → $\underline{1}$.6
0.6 x 2 → $\underline{1}$.2
0.2 x 2 → $\underline{0}$.4
0.4 x 2 → $\underline{0}$.8
0.8 x 2 → $\underline{1}$.6
0.6 x 2 → $\underline{1}$.2
0.2 x 2 → $\underline{0}$.4
⋮

Slides adapted from tan wooi haw's lecture notes (FOE)

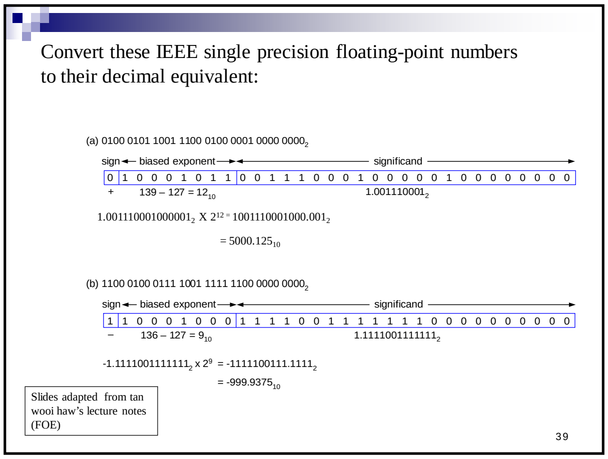
stored [23 bits]

– $6 + 127 = 133_{10}$ 1 • 0 0 1 1 0 1 1 0 1 1 0 ...

1 | 1 0 0 0 0 1 0 1 | 0 0 1 1 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0

sign ← biased exponent → ← significand →

38



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

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

sign ← biased exponent → ← significand →

0 | 1 0 0 0 1 0 1 1 | 0 0 1 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0

+ $139 - 127 = 12_{10}$ 1.001110001_2

1.001110001000001_2 X 2^{12} = 1001110001000.001_2

$= 5000.125_{10}$

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

sign ← biased exponent → ← significand →

1 | 1 0 0 0 1 0 0 0 | 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0

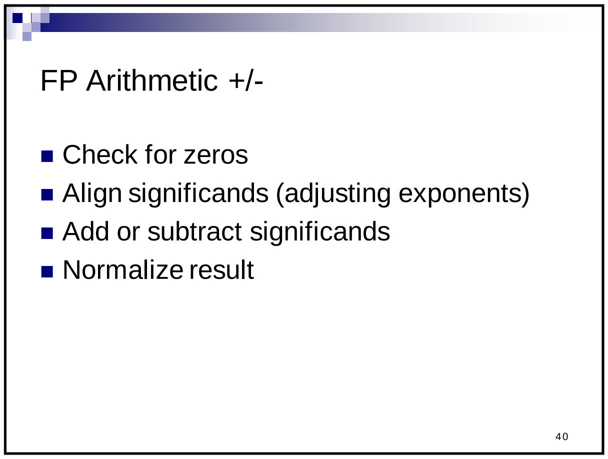
– $136 - 127 = 9_{10}$ 1.1111001111111_2

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

$= -999.9375_{10}$

Slides adapted from tan wooi haw's lecture notes (FOE)

39



FP Arithmetic +/-

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

40

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 Arithmetic (cont.)

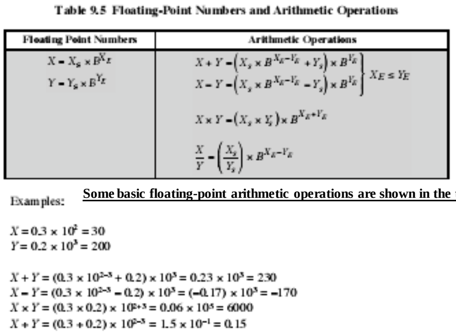


Table 9.5 Floating-Point Numbers and Arithmetic Operations

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

Examples:

Some basic floating-point arithmetic operations are shown in the table

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

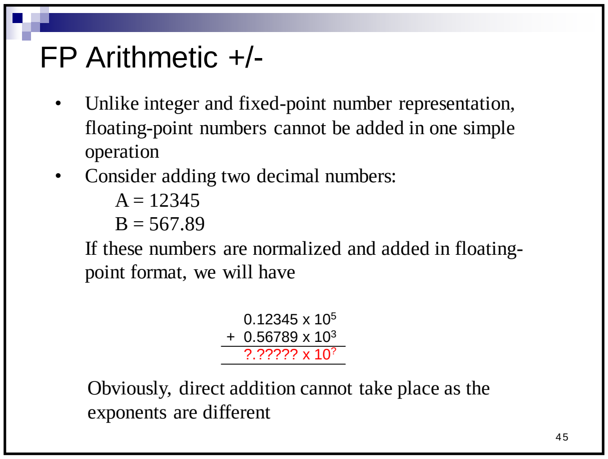
$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$
$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$
$X + Y = (0.3 + 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

Floating-point Arithmetic (cont.)

- For addition and subtraction, it is necessary to ensure that both operand exponents have the same value
- This may involves shifting the radix point of one of the operand to achieve alignment

Floating-point Arithmetic (cont.)

- Some problems that may arise during arithmetic operations are:
  i. Exponent overflow: A positive exponent exceeds the maximum possible exponent value and this may leads to +∞ or -∞ in some systems
  ii. Exponent underflow: A negative exponent is less than the minimum possible exponent value (eg. 2^{-200}), the number is too small to be represented and maybe reported as 0
  iii. Significand underflow: In the process of aligning significands, the smaller number may have a significand which is too small to be represented
  iv. Significand overflow: The addition of two significands of the same sign may result in a carry out from the most significant bit



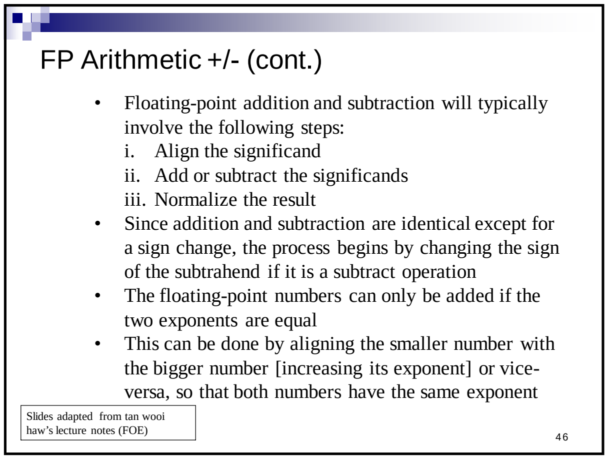
FP Arithmetic +/-

- Unlike integer and fixed-point number representation, floating-point numbers cannot be added in one simple operation
- Consider adding two decimal numbers:
  A = 12345
  B = 567.89
  If these numbers are normalized and added in floating-point format, we will have

$$\begin{array}{r} 0.12345 \times 10^5 \\ +\ 0.56789 \times 10^3 \\ \hline ?.????? \times 10^? \\ \hline \end{array}$$

Obviously, direct addition cannot take place as the exponents are different

45

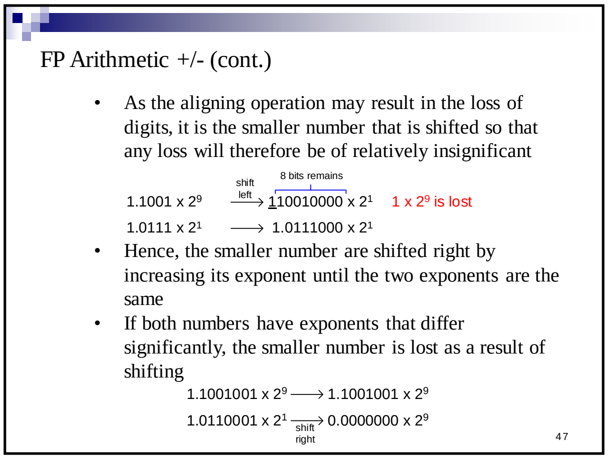


FP Arithmetic +/- (cont.)

- Floating-point addition and subtraction will typically involve the following steps:
  i. Align the significand
  ii. Add or subtract the significands
  iii. Normalize the result
- Since addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation
- The floating-point numbers can only be added if the two exponents are equal
- This can be done by aligning the smaller number with the bigger number [increasing its exponent] or vice-versa, so that both numbers have the same exponent

Slides adapted from tan wooi haw's lecture notes (FOE)

46



FP Arithmetic +/- (cont.)

- As the aligning operation may result in the loss of digits, it is the smaller number that is shifted so that any loss will therefore be of relatively insignificant

$$1.1001 \times 2^9 \xrightarrow{\text{shift left}} \underline{1}\overbrace{10010000}^{\text{8 bits remains}} \times 2^1 \quad \text{1 x } 2^9 \text{ is lost}$$

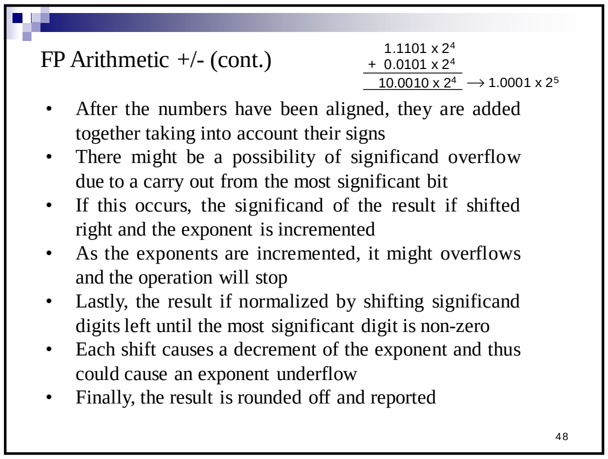
$$1.0111 \times 2^1 \longrightarrow 1.0111000 \times 2^1$$

- Hence, the smaller number are shifted right by increasing its exponent until the two exponents are the same
- If both numbers have exponents that differ significantly, the smaller number is lost as a result of shifting

$$1.1001001 \times 2^9 \longrightarrow 1.1001001 \times 2^9$$

$$1.0110001 \times 2^1 \xrightarrow[\text{shift right}]{} 0.0000000 \times 2^9$$

47

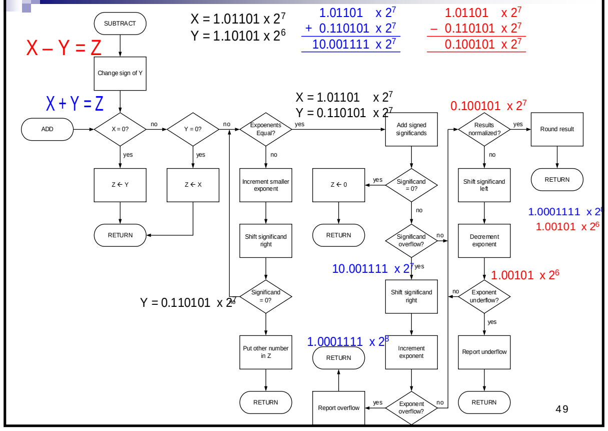


FP Arithmetic +/- (cont.)

$$\begin{array}{r} 1.1101 \times 2^4 \\ +\ 0.0101 \times 2^4 \\ \hline 10.0010 \times 2^4 \\ \hline \end{array} \longrightarrow 1.0001 \times 2^5$$

- After the numbers have been aligned, they are added together taking into account their signs
- There might be a possibility of significand overflow due to a carry out from the most significant bit
- If this occurs, the significand of the result if shifted right and the exponent is incremented
- As the exponents are incremented, it might overflows and the operation will stop
- Lastly, the result if normalized by shifting significand digits left until the most significant digit is non-zero
- Each shift causes a decrement of the exponent and thus could cause an exponent underflow
- Finally, the result is rounded off and reported

48



FP Arithmetic +/- (cont.)

- Some of the floating-point arithmetic will lead to an increase number of bits in the mantissa
- For example, consider adding these 5 significant bits floating-point numbers:

  A = 0.11001 x 2^4
  B = 0.10001 x 2^3

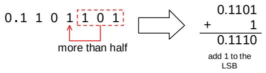
$$\begin{array}{rl} A = & 0.11001 \quad \times 2^4 \\ B = & \underline{0.010001 \ \times 2^4} \\ & 1.000011 \ \times 2^4 \xrightarrow{\text{normalize}} 0.10000\underline{11} \times 2^5 \end{array}$$

- The result has two extra bit of precision which cannot be fitted into the floating point format
- For simplicity, the number can be truncated to give 0.10000 x 2^5

FP Arithmetic +/- (cont.)

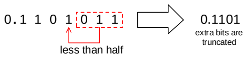
- Truncation is the simplest method which involves nothing more than taking away the extra bits
- A much better technique is rounding in which if the value of the extra bits is greater than half the least significant bit of the retained bits, 1 is added to the LSB of the remaining digits
- For example, consider rounding these numbers to 4 significant bits:

  i. 0.1101101
     extra bits → 0.0000<u>101</u>
     LSB of retained bits → 0.000<u>1</u>



FP Arithmetic +/- (cont.)

ii. 0.1101011
    extra bits → 0.0000<u>011</u>
    LSB of retained bits → 0.000<u>1</u>



- Truncation always undervalues the result, leading to a systematic error, whereas rounding sometimes reduces the result and sometimes increases it
- Rounding is always preferred to truncation partly because it is more accurate and partly it gives rise to an unbiased error
- Major disadvantage of rounding is that it requires a further arithmetic operation on the result

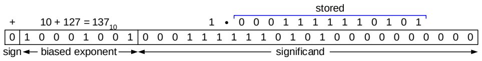
Example

Perform the following arithmetic operation using floating point arithmetic, In each case, show how the numbers would be stored using IEEE single-precision format

i. $1150.625_{10} - 525.25_{10}$

1150.625_{10} = $100\ 0111\ 1110.\ 101_2$

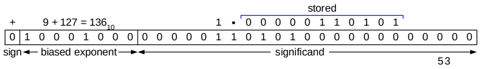
= $1.\ 0001\ 1111\ 10101 \text{ x } 2^{10}$



0 | 1 0 0 0 1 0 0 1 | 0 0 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0

525.25_{10} = $10\ 0000\ 1101.01_2$

= $1.\ 0000\ 0110\ 101 \text{ x } 2^9$



0 | 1 0 0 0 1 0 0 0 | 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0

continue ...

As these numbers have different exponents, the smaller number is shifted right to align with the larger number

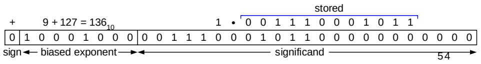
1000 1000	1.00000110101 →	1000 1001	0.100000110101
exponent	mantissa	exponent	mantissa

Subtract the mantissa

$$\begin{array}{r} 1.0001111110101 \\ -\ 0.100000110101 \\ \hline 0.1001110001011 \\ \hline \end{array}$$

Normalize the result

1000 1001	0.1001110001011 →	1000 1000	1.001110001011
exponent	mantissa	exponent	mantissa



0 | 1 0 0 0 1 0 0 0 | 0 0 1 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0

continue ...

ii. $68.3_{10} + 12.2_{10}$

68.3_{10} = 100 0100.01001 1001 ...

= $1.00\ 0100\ 01001\ 1001 \ldots \text{ x } 2^6$

$68_{10} = 100\ 0100_2$

$0.3_{10} \Rightarrow$ 0.3 x 2 → $\underline{0}.6$

0.6 x 2 → $\underline{1}.2$

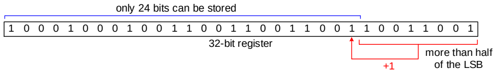
0.2 x 2 → $\underline{0}.4$

0.4 x 2 → $\underline{0}.8$

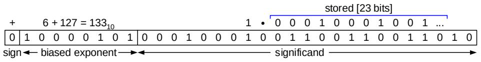
0.8 x 2 → $\underline{1}.6$

0.6 x 2 → $\underline{1}.2$

⋮



1 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1



0 | 1 0 0 0 0 1 0 1 | 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0

continue ...

12.2_{10} = 1100.0011 0011 ...

= $1.100\ 0011\ 0011 \ldots \text{ x } 2^3$

$12_{10} = 1100_2$

$0.2_{10} \Rightarrow$ 0.2 x 2 → $\underline{0}.4$

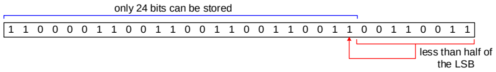
0.4 x 2 → $\underline{0}.8$

0.8 x 2 → $\underline{1}.6$

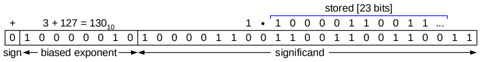
0.6 x 2 → $\underline{1}.2$

0.2 x 2 → $\underline{0}.4$

⋮



1 1 0 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1



0 | 1 0 0 0 0 0 1 0 | 1 0 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

continue ...

Align the smaller number with the larger number by shifting it to the right [increasing the exponent]

1000 0010	1.1000011001100110011	→	1000 0101	0.0011000011001100110011
exponent	mantissa		exponent	mantissa

Subtract the mantissa

$$\begin{array}{r} 1.0001000100110011001101\,0 \\ +\ 0.0011000011001100110011\,0011 \\ \hline 1.0100001000000000000000\,0011 \\ \hline \end{array}$$

less than half of the LSB

Store the result in IEEE single-precision format

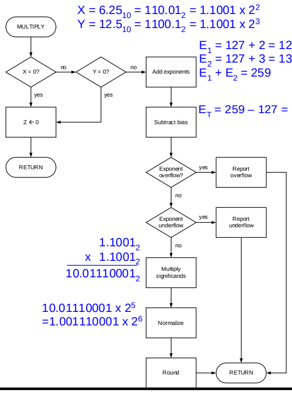
0	1 0 0 0 0 1 0 1	0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
sign	biased exponent	significand

Floating-point Multiplication

X x Y = Z

$X = 6.25_{10} = 110.01_2 = 1.1001 \times 2^2$

$Y = 12.5_{10} = 1100.1_2 = 1.1001 \times 2^3$



$E_1 = 127 + 2 = 129$

$E_2 = 127 + 3 = 130$

$E_1 + E_2 = 259$

$E_T = 259 - 127 = 132$

$$\begin{array}{r} 1.1001_2 \\ \times\ \ 1.1001_2 \\ \hline 10.01110001_2 \end{array}$$

10.01110001×2^5

$=1.001110001 \times 2^6$

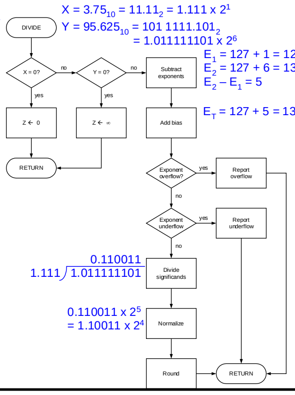
Floating-point Division

Y ÷ X = Z

$X = 3.75_{10} = 11.11_2 = 1.111 \times 2^1$

$Y = 95.625_{10} = 101\ 1111.101_2$

$= 1.011111101 \times 2^6$



$E_1 = 127 + 1 = 128$

$E_2 = 127 + 6 = 133$

$E_2 - E_1 = 5$

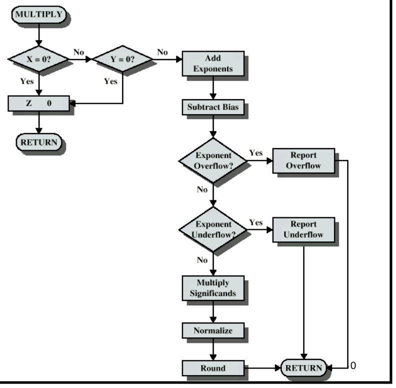
$E_T = 127 + 5 = 132$

$$\begin{array}{r} 0.110011 \\ 1.111 \overline{\smash{)}\,1.011111101} \end{array}$$

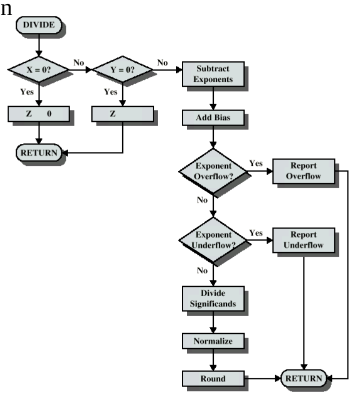
0.110011×2^5

$= 1.10011 \times 2^4$

Floating Point Multiplication



Floating Point Division



PROBLEM (1)

- 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

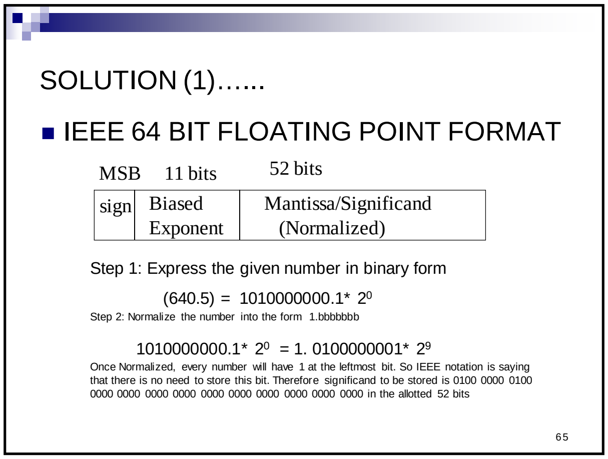
SOLUTION (1)…….

- 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 0010 0000 0000 000



SOLUTION (1)……

- IEEE 64 BIT FLOATING POINT FORMAT

MSB	11 bits	52 bits
sign	Biased Exponent	Mantissa/Significand (Normalized)

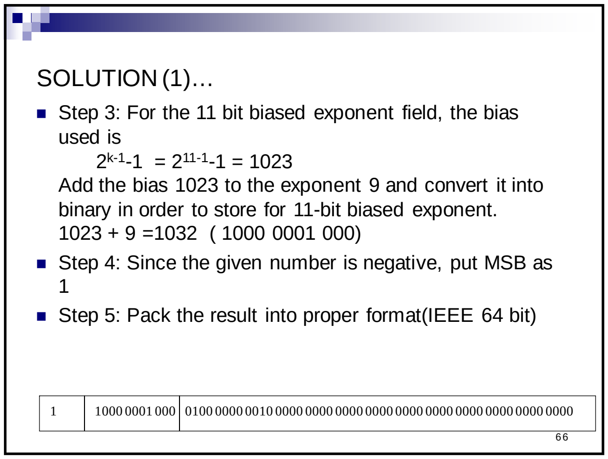
Step 1: Express the given number in binary form

$$(640.5) = 1010000000.1 * 2^0$$

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 in the allotted 52 bits



SOLUTION (1)…

- 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 0010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000