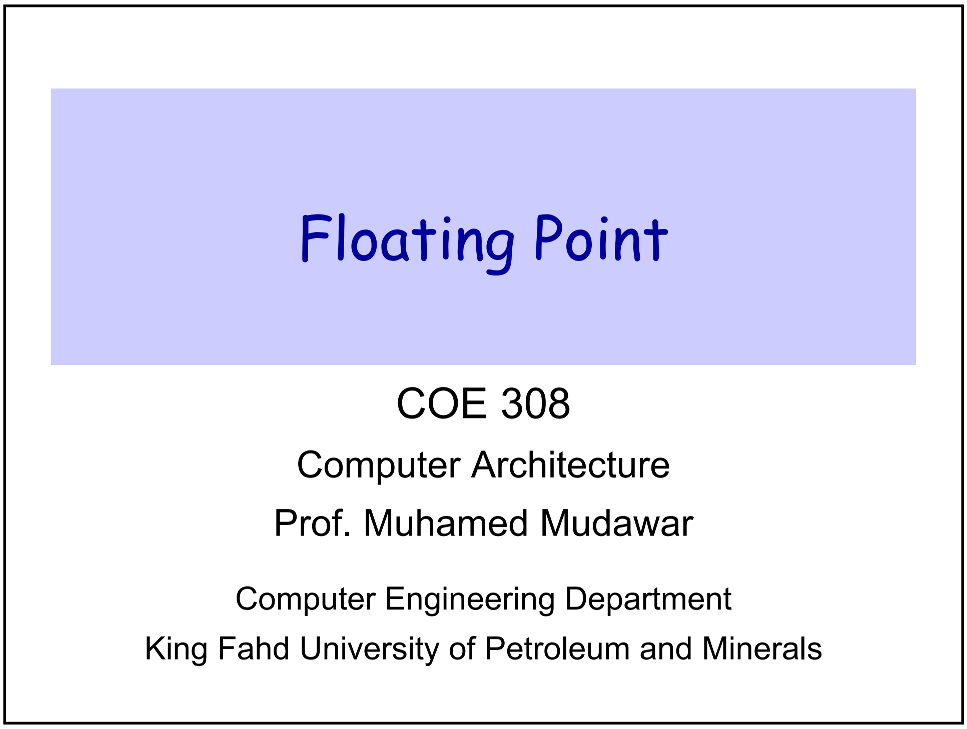


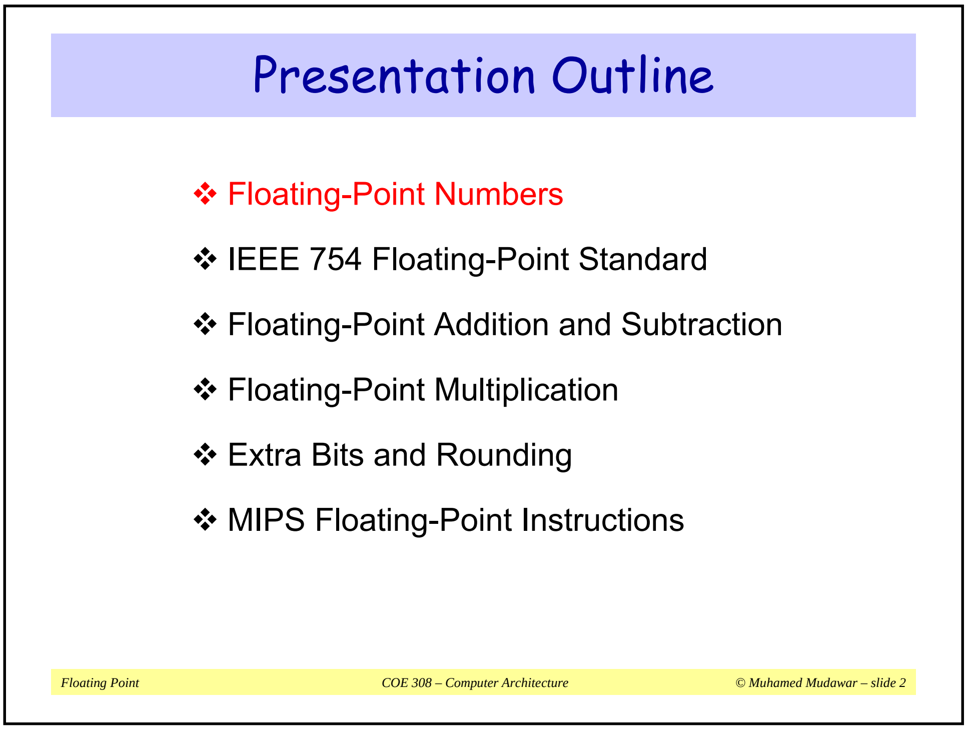
Floating Point

COE 308

Computer Architecture

Prof. Muhamed Mudawar

Computer Engineering Department
King Fahd University of Petroleum and Minerals



Presentation Outline

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ Extra Bits and Rounding
- ❖ MIPS Floating-Point Instructions

The World is Not Just Integers

- Programming languages support numbers with fraction
  - Called floating-point numbers
  - Examples:

    3.14159265… (π)

    2.71828… (e)

    0.000000001 or 1.0×10^{-9} (seconds in a nanosecond)

    86,400,000,000,000 or 8.64×10^{13} (nanoseconds in a day)

    last number is a large integer that cannot fit in a 32-bit integer
- We use a scientific notation to represent
  - Very small numbers (e.g. 1.0×10^{-9})
  - Very large numbers (e.g. 8.64×10^{13})
  - Scientific notation: $\pm\, d\,.\,f_1f_2f_3f_4 \ldots \times 10^{\pm e_1e_2e_3}$

Floating-Point Numbers

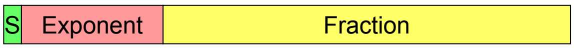
- Examples of floating-point numbers in base 10 …
  - 5.341×10^{3} , 0.05341×10^{5} , -2.013×10^{-1} , -201.3×10^{-3}

    decimal point
- Examples of floating-point numbers in base 2 …
  - 1.00101×2^{23} , 0.0100101×2^{25} , -1.101101×2^{-3} , -1101.101×2^{-6}
  - Exponents are kept in decimal for clarity

    binary point
  - The binary number $(1101.101)_2 = 2^3+2^2+2^0+2^{-1}+2^{-3} = 13.625$
- Floating-point numbers should be normalized
  - Exactly one non-zero digit should appear before the point
    - In a decimal number, this digit can be from 1 to 9
    - In a binary number, this digit should be 1
  - Normalized FP Numbers: 5.341×10^{3} and -1.101101×2^{-3}
  - NOT Normalized: 0.05341×10^{5} and -1101.101×2^{-6}

Floating-Point Representation

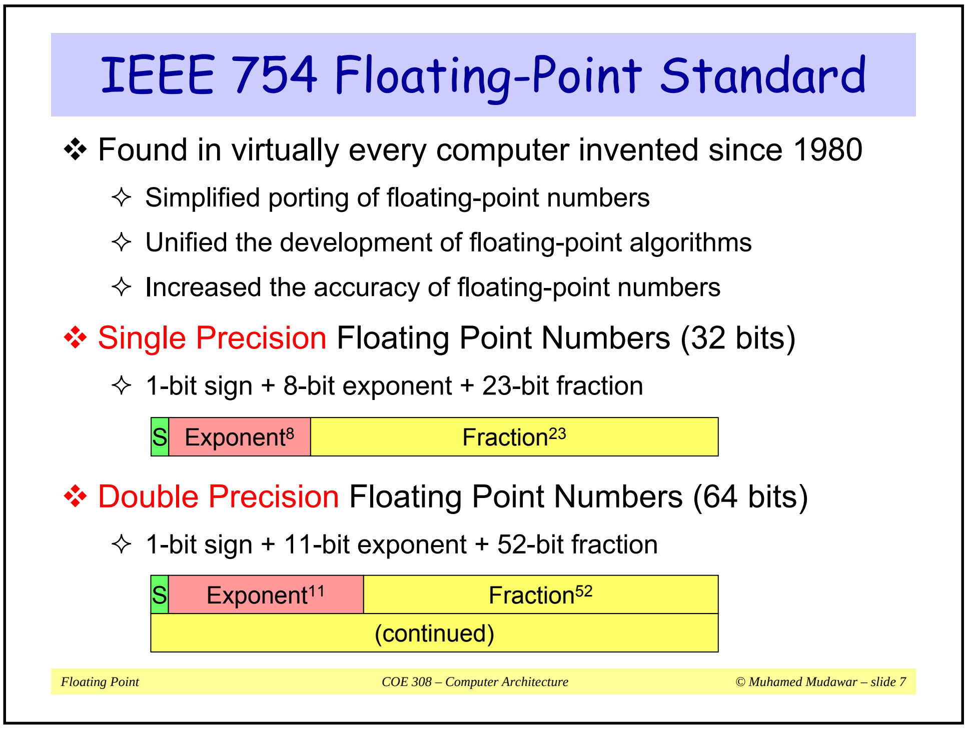
- A floating-point number is represented by the triple
  - *S* is the Sign bit (0 is positive and 1 is negative)
    - Representation is called sign and magnitude
  - *E* is the Exponent field (signed)
    - Very large numbers have large positive exponents
    - Very small close-to-zero numbers have negative exponents
    - More bits in exponent field increases range of values
  - *F* is the Fraction field (fraction after binary point)
    - More bits in fraction field improves the precision of FP numbers



Value of a floating-point number = $(-1)^{S} \times \text{val}(F) \times 2^{\text{val}(E)}$

Next . . .

- Floating-Point Numbers
- IEEE 754 Floating-Point Standard
- Floating-Point Addition and Subtraction
- Floating-Point Multiplication
- Extra Bits and Rounding
- MIPS Floating-Point Instructions



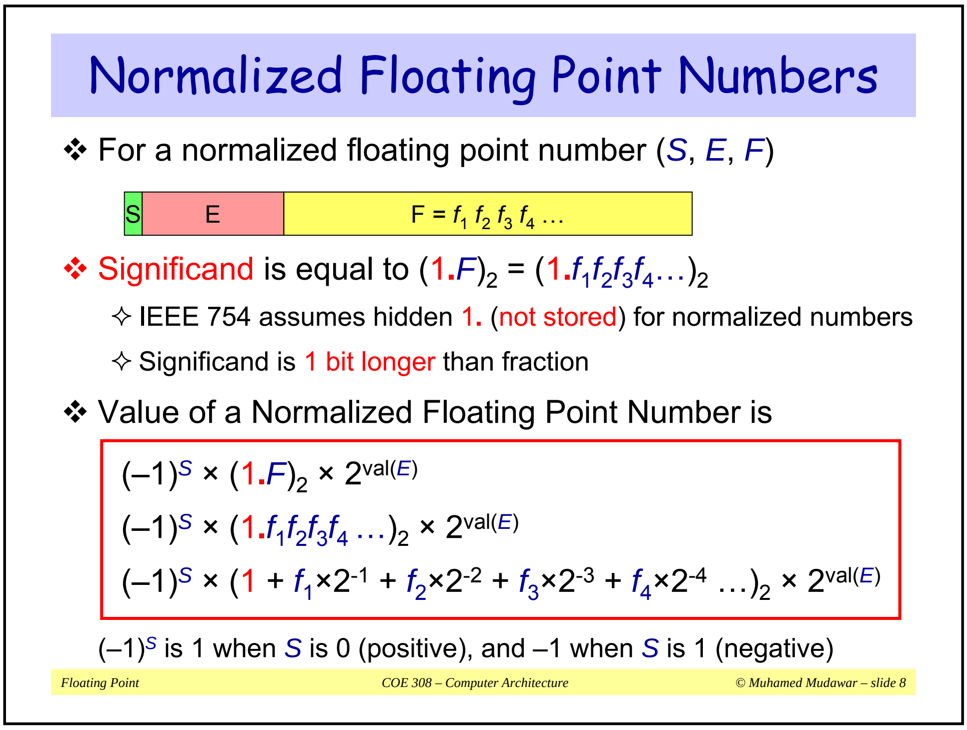
IEEE 754 Floating-Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
  - Increased the accuracy of floating-point numbers
- Single Precision Floating Point Numbers (32 bits)
  - 1-bit sign + 8-bit exponent + 23-bit fraction

S	Exponent^8	Fraction^{23}

- Double Precision Floating Point Numbers (64 bits)
  - 1-bit sign + 11-bit exponent + 52-bit fraction

S	Exponent^{11}	Fraction^{52}
(continued)		



Normalized Floating Point Numbers

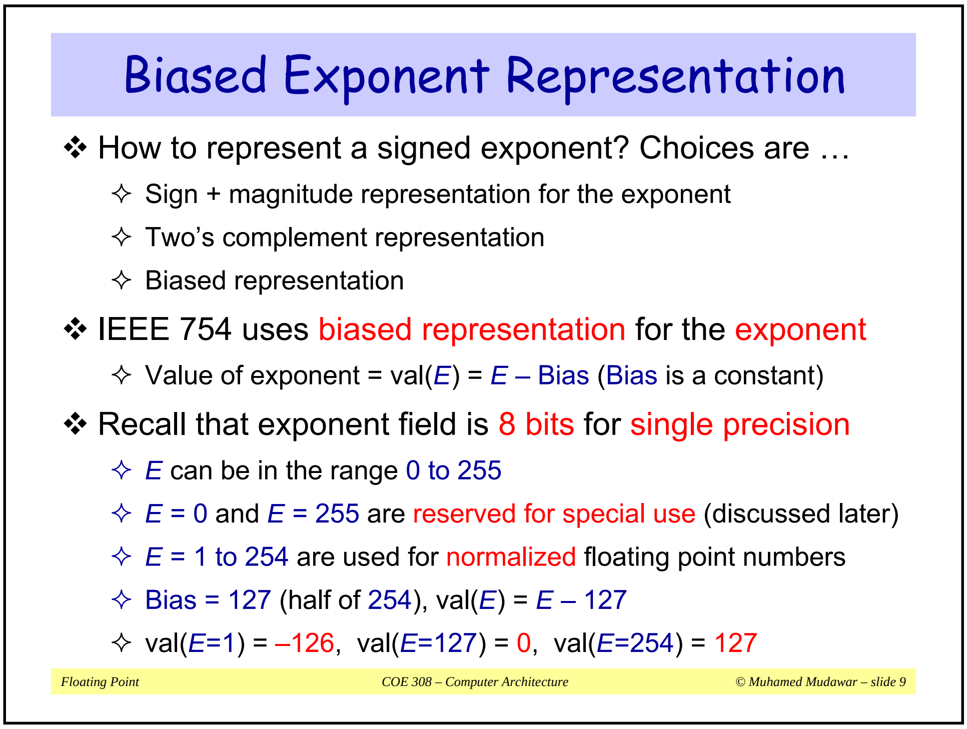
- For a normalized floating point number (S, E, F)

S	E	F = $f_1\ f_2\ f_3\ f_4 \ldots$

- Significand is equal to $(1.F)_2 = (1.f_1f_2f_3f_4\ldots)_2$
  - IEEE 754 assumes hidden 1. (not stored) for normalized numbers
  - Significand is 1 bit longer than fraction
- Value of a Normalized Floating Point Number is

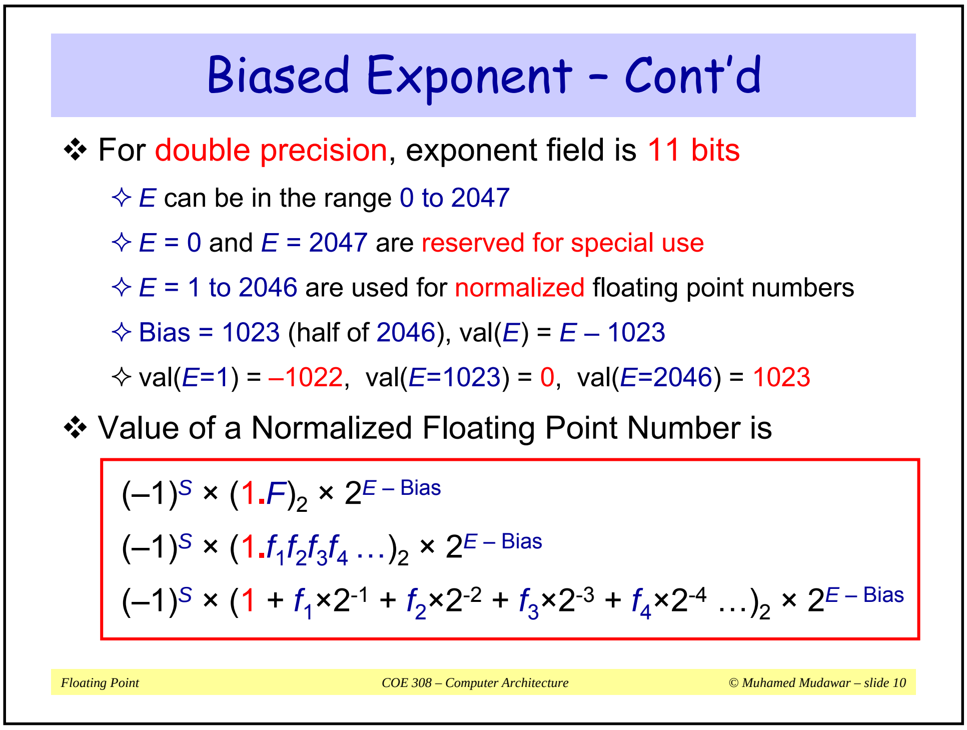
$$(-1)^S \times (1.F)_2 \times 2^{\text{val}(E)}$$
$$(-1)^S \times (1.f_1f_2f_3f_4\ldots)_2 \times 2^{\text{val}(E)}$$
$$(-1)^S \times (1 + f_1\times2^{-1} + f_2\times2^{-2} + f_3\times2^{-3} + f_4\times2^{-4}\ldots)_2 \times 2^{\text{val}(E)}$$

$(-1)^S$ is 1 when S is 0 (positive), and –1 when S is 1 (negative)



Biased Exponent Representation

- How to represent a signed exponent? Choices are …
  - Sign + magnitude representation for the exponent
  - Two's complement representation
  - Biased representation
- IEEE 754 uses biased representation for the exponent
  - Value of exponent = val(E) = E – Bias (Bias is a constant)
- Recall that exponent field is 8 bits for single precision
  - E can be in the range 0 to 255
  - E = 0 and E = 255 are reserved for special use (discussed later)
  - E = 1 to 254 are used for normalized floating point numbers
  - Bias = 127 (half of 254), val(E) = E – 127
  - val(E=1) = –126, val(E=127) = 0, val(E=254) = 127



Biased Exponent – Cont'd

- For double precision, exponent field is 11 bits
  - E can be in the range 0 to 2047
  - E = 0 and E = 2047 are reserved for special use
  - E = 1 to 2046 are used for normalized floating point numbers
  - Bias = 1023 (half of 2046), val(E) = E – 1023
  - val(E=1) = –1022, val(E=1023) = 0, val(E=2046) = 1023
- Value of a Normalized Floating Point Number is

$$(-1)^S \times (1.F)_2 \times 2^{E - \text{Bias}}$$
$$(-1)^S \times (1.f_1f_2f_3f_4 \ldots)_2 \times 2^{E - \text{Bias}}$$
$$(-1)^S \times (1 + f_1{\times}2^{-1} + f_2{\times}2^{-2} + f_3{\times}2^{-3} + f_4{\times}2^{-4} \ldots)_2 \times 2^{E - \text{Bias}}$$

Examples of Single Precision Float

❖ What is the decimal value of this Single Precision float?

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

❖ Solution:

- ✧ Sign = 1 is negative
- ✧ Exponent = $(01111100)_2$ = 124, *E* – bias = 124 – 127 = –3
- ✧ Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
- ✧ Value in decimal = $-1.25 \times 2^{-3} = -0.15625$

❖ What is the decimal value of?

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

❖ Solution:

implicit

- ✧ Value in decimal = $+(1.01001100 \ldots 0)_2 \times 2^{130-127} =$ $(1.01001100 \ldots 0)_2 \times 2^3 = (1010.01100 \ldots 0)_2 =$ 10.375

Examples of Double Precision Float

❖ What is the decimal value of this Double Precision float ?

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

❖ Solution:

- ✧ Value of exponent = $(10000000101)_2$ – Bias = 1029 – 1023 = 6
- ✧ Value of double float = $(1.00101010 \ldots 0)_2 \times 2^6$ (1. is implicit) = $(1001010.10 \ldots 0)_2$ = 74.5

❖ What is the decimal value of ?

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

❖ Do it yourself! (answer should be $-1.5 \times 2^{-7} = -0.01171875$)

Converting FP Decimal to Binary

❖ Convert –0.8125 to binary in single and double precision

❖ Solution:

✧ Fraction bits can be obtained using multiplication by 2

- 0.8125 × 2 = 1.625
- 0.625 × 2 = 1.25
- 0.25 × 2 = 0.5
- 0.5 × 2 = 1.0
- Stop when fractional part is 0

$0.8125 = (0.1101)_2 = ½ + ¼ + 1/16 = 13/16$

✧ Fraction = $(0.1101)_2 = (1.101)_2 \times 2^{-1}$ (Normalized)

✧ Exponent = –1 + Bias = 126 (single precision) and 1022 (double)

1	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Single Precision
1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Double Precision
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Largest Normalized Float

❖ What is the Largest normalized float?

❖ Solution for Single Precision:

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

✧ Exponent – bias = 254 – 127 = 127 (largest exponent for SP)

✧ Significand = $(1.111 \ldots 1)_2$ = almost 2

✧ Value in decimal ≈ $2 \times 2^{127} \approx 2^{128} \approx 3.4028 \ldots \times 10^{38}$

❖ Solution for Double Precision:

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

✧ Value in decimal ≈ $2 \times 2^{1023} \approx 2^{1024} \approx 1.79769 \ldots \times 10^{308}$

❖ Overflow: exponent is too large to fit in the exponent field

Smallest Normalized Float

- ❖ What is the smallest (in absolute value) normalized float?
- ❖ Solution for Single Precision:

```
0 00000001 00000000000000000000000
```

  - ✧ Exponent – bias = 1 – 127 = –126 (smallest exponent for SP)
  - ✧ Significand = $(1.000 \ldots 0)_2 = 1$
  - ✧ Value in decimal = $1 \times 2^{-126} = 1.17549 \ldots \times 10^{-38}$
- ❖ Solution for Double Precision:

```
0 00000000001 00000000000000000000
00000000000000000000000000000000
```

  - ✧ Value in decimal = $1 \times 2^{-1022} = 2.22507 \ldots \times 10^{-308}$
- ❖ Underflow: exponent is too small to fit in exponent field

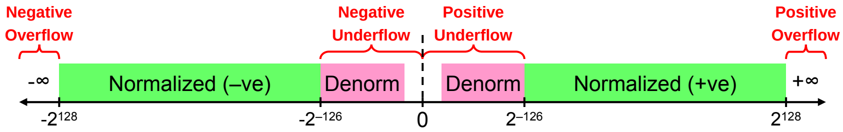
Zero, Infinity, and NaN

- ❖ Zero
  - ✧ Exponent field $E = 0$ and fraction $F = 0$
  - ✧ +0 and –0 are possible according to sign bit S
- ❖ Infinity
  - ✧ Infinity is a special value represented with maximum E and $F = 0$
    - For single precision with 8-bit exponent: maximum $E = 255$
    - For double precision with 11-bit exponent: maximum $E = 2047$
  - ✧ Infinity can result from overflow or division by zero
  - ✧ $+\infty$ and $-\infty$ are possible according to sign bit S
- ❖ NaN (Not a Number)
  - ✧ NaN is a special value represented with maximum E and $F \neq 0$
  - ✧ Result from exceptional situations, such as 0/0 or sqrt(negative)
  - ✧ Operation on a NaN results is NaN: Op(X, NaN) = NaN

Denormalized Numbers

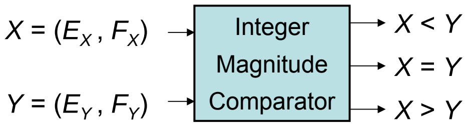
- ❖ IEEE standard uses denormalized numbers to …
  - ✧ Fill the gap between 0 and the smallest normalized float
  - ✧ Provide gradual underflow to zero
- ❖ Denormalized: exponent field E is 0 and fraction $F \neq 0$
  - ✧ Implicit 1. before the fraction now becomes 0. (not normalized)
- ❖ Value of denormalized number (S, 0, F)

Single precision:	$(-1)^S \times (0.F)_2 \times 2^{-126}$
Double precision:	$(-1)^S \times (0.F)_2 \times 2^{-1022}$



Floating-Point Comparison

- ❖ IEEE 754 floating point numbers are ordered
  - ✧ Because exponent uses a biased representation …
    - Exponent value and its binary representation have same ordering
  - ✧ Placing exponent before the fraction field orders the magnitude
    - Larger exponent ⇒ larger magnitude
    - For equal exponents, Larger fraction ⇒ larger magnitude
    - $0 < (0.F)_2 \times 2^{E_{min}} < (1.F)_2 \times 2^{E-Bias} < \infty$ ($E_{min} = 1 - Bias$)
  - ✧ Because sign bit is most significant ⇒ quick test of signed <
- ❖ Integer comparator can compare magnitudes



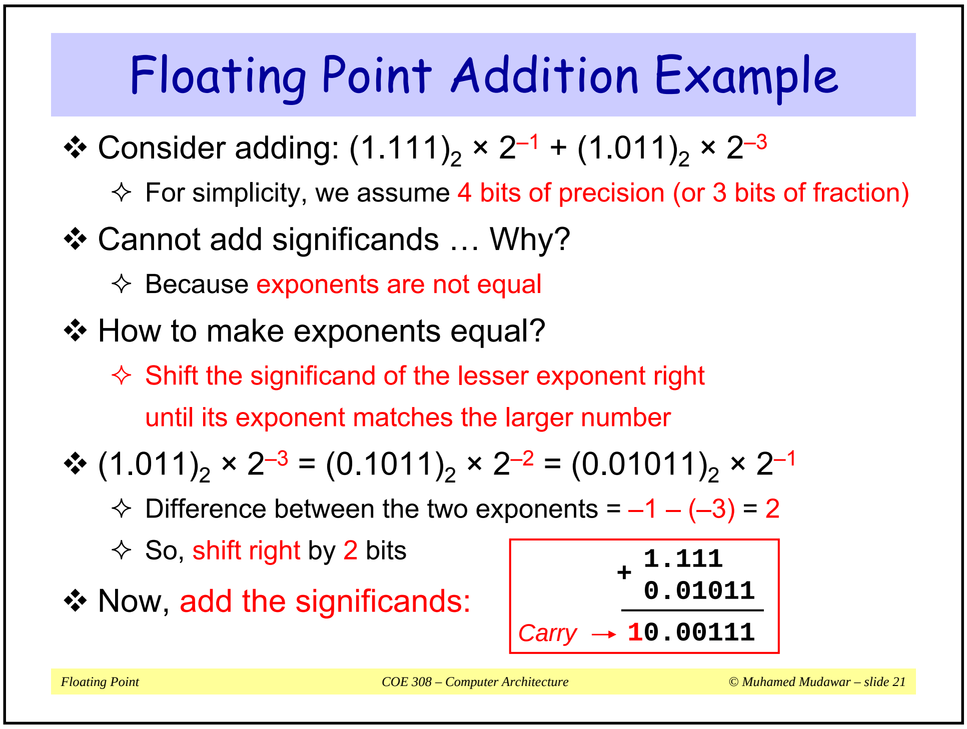
Summary of IEEE 754 Encoding

Single-Precision	Exponent = 8	Fraction = 23	Value
Normalized Number	1 to 254	Anything	$\pm (1.F)_2 \times 2^{E-127}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-126}$
Zero	0	0	± 0
Infinity	255	0	± ∞
NaN	255	nonzero	NaN

Double-Precision	Exponent = 11	Fraction = 52	Value
Normalized Number	1 to 2046	Anything	$\pm (1.F)_2 \times 2^{E-1023}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-1022}$
Zero	0	0	± 0
Infinity	2047	0	± ∞
NaN	2047	nonzero	NaN

Next . . .

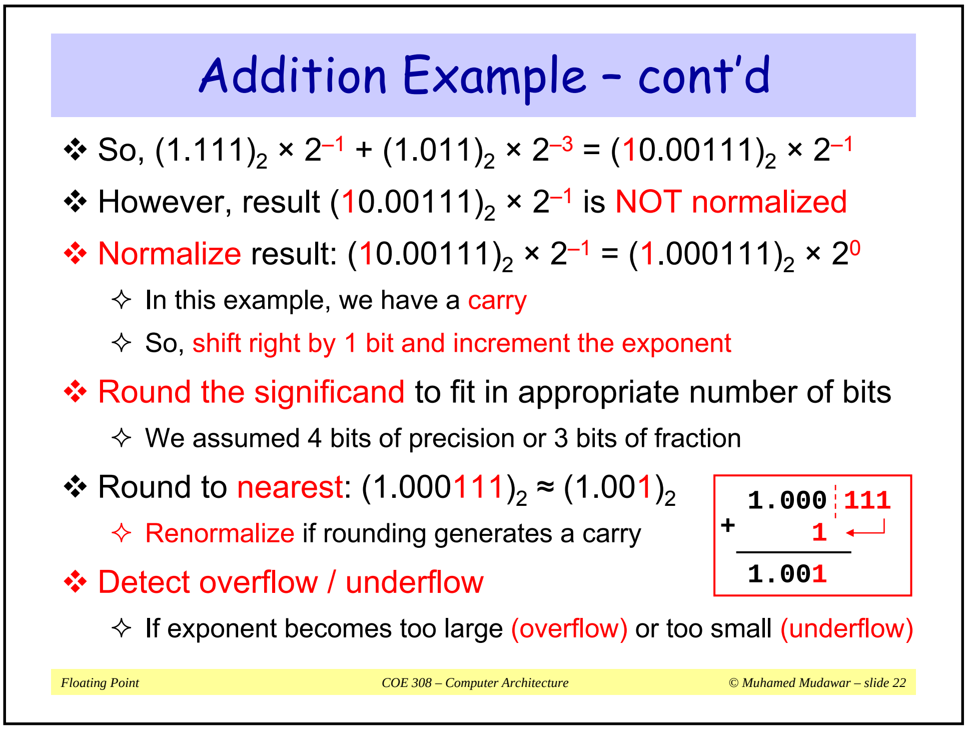
- Floating-Point Numbers
- IEEE 754 Floating-Point Standard
- Floating-Point Addition and Subtraction
- Floating-Point Multiplication
- Extra Bits and Rounding
- MIPS Floating-Point Instructions



Floating Point Addition Example

- Consider adding: $(1.111)_2 \times 2^{-1} + (1.011)_2 \times 2^{-3}$
  - For simplicity, we assume 4 bits of precision (or 3 bits of fraction)
- Cannot add significands … Why?
  - Because exponents are not equal
- How to make exponents equal?
  - Shift the significand of the lesser exponent right until its exponent matches the larger number
- $(1.011)_2 \times 2^{-3} = (0.1011)_2 \times 2^{-2} = (0.01011)_2 \times 2^{-1}$
  - Difference between the two exponents = –1 – (–3) = 2
  - So, shift right by 2 bits
- Now, add the significands:

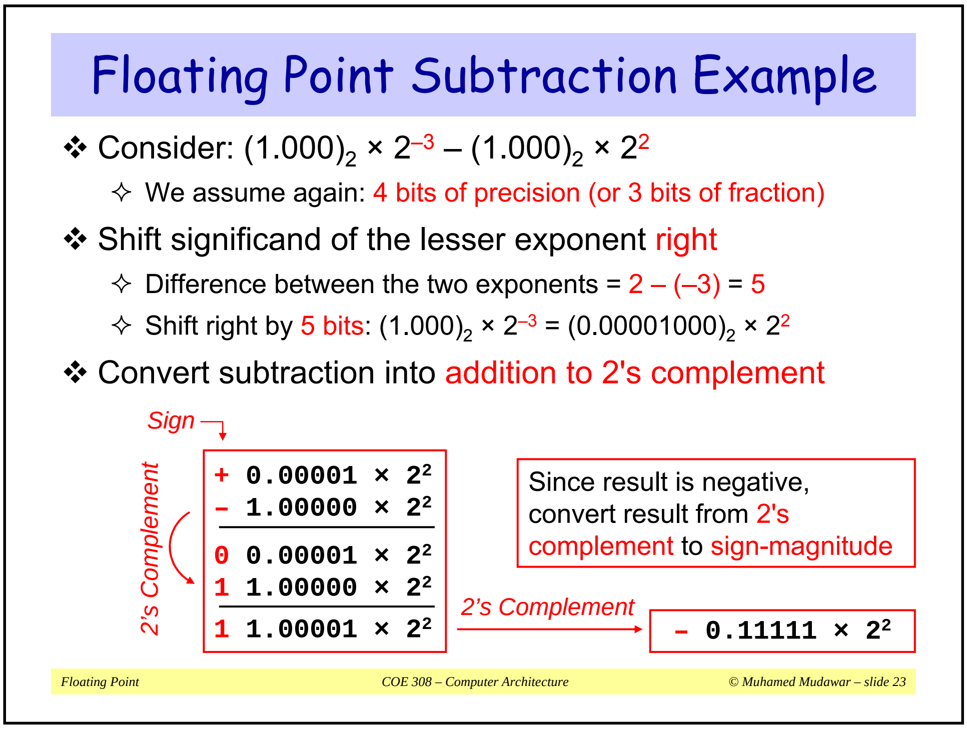
```
               1.111
             + 0.01011
               -------
Carry  →  10.00111
```



Addition Example – cont'd

- So, $(1.111)_2 \times 2^{-1} + (1.011)_2 \times 2^{-3} = (10.00111)_2 \times 2^{-1}$
- However, result $(10.00111)_2 \times 2^{-1}$ is NOT normalized
- Normalize result: $(10.00111)_2 \times 2^{-1} = (1.000111)_2 \times 2^{0}$
  - In this example, we have a carry
  - So, shift right by 1 bit and increment the exponent
- Round the significand to fit in appropriate number of bits
  - We assumed 4 bits of precision or 3 bits of fraction
- Round to nearest: $(1.000111)_2 \approx (1.001)_2$
  - Renormalize if rounding generates a carry
- Detect overflow / underflow
  - If exponent becomes too large (overflow) or too small (underflow)

```
  1.000 ¦ 111
+     1 ←┘
  -----
  1.001
```



Floating Point Subtraction Example

- Consider: $(1.000)_2 \times 2^{-3} - (1.000)_2 \times 2^{2}$
  - We assume again: 4 bits of precision (or 3 bits of fraction)
- Shift significand of the lesser exponent right
  - Difference between the two exponents = $2 - (-3) = 5$
  - Shift right by 5 bits: $(1.000)_2 \times 2^{-3} = (0.00001000)_2 \times 2^{2}$
- Convert subtraction into addition to 2's complement

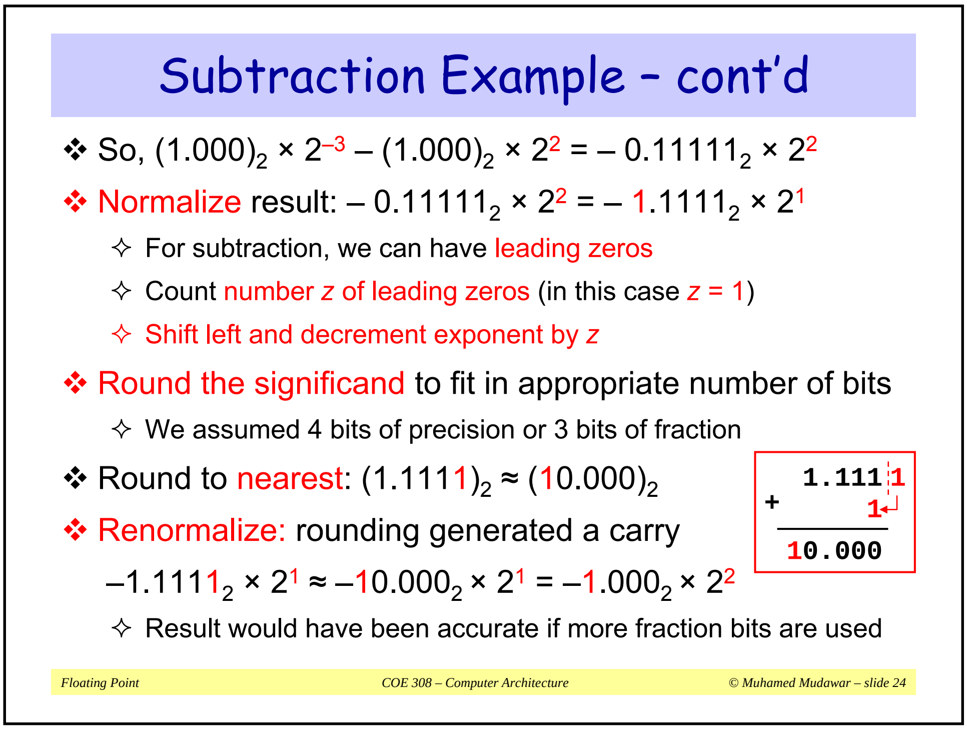
Sign

```
+ 0.00001 × 2²
– 1.00000 × 2²
--------------
0 0.00001 × 2²
1 1.00000 × 2²
--------------
1 1.00001 × 2²
```

2's Complement

Since result is negative, convert result from 2's complement to sign-magnitude

$1\ 1.00001 \times 2^{2} \xrightarrow{\text{2's Complement}} -\ 0.11111 \times 2^{2}$



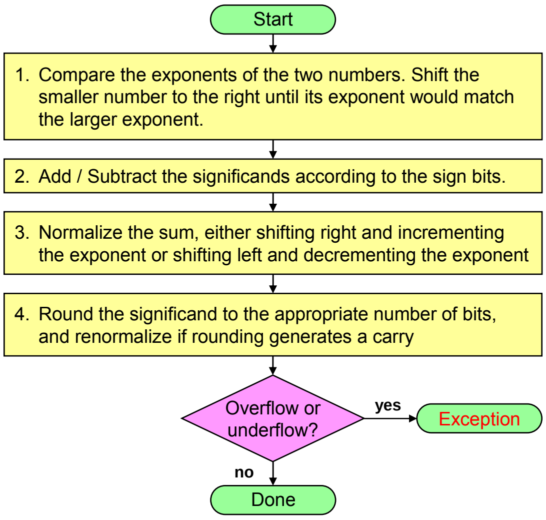
Subtraction Example – cont'd

- So, $(1.000)_2 \times 2^{-3} - (1.000)_2 \times 2^{2} = -\ 0.11111_2 \times 2^{2}$
- Normalize result: $-\ 0.11111_2 \times 2^{2} = -\ 1.1111_2 \times 2^{1}$
  - For subtraction, we can have leading zeros
  - Count number *z* of leading zeros (in this case $z = 1$)
  - Shift left and decrement exponent by *z*
- Round the significand to fit in appropriate number of bits
  - We assumed 4 bits of precision or 3 bits of fraction
- Round to nearest: $(1.1111)_2 \approx (10.000)_2$
- Renormalize: rounding generated a carry

  $-1.1111_2 \times 2^{1} \approx -10.000_2 \times 2^{1} = -1.000_2 \times 2^{2}$
  - Result would have been accurate if more fraction bits are used

```
  1.111 1
+     1
--------
 10.000
```

Floating Point Addition / Subtraction



Start

1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent.

2. Add / Subtract the significands according to the sign bits.

3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent

4. Round the significand to the appropriate number of bits, and renormalize if rounding generates a carry

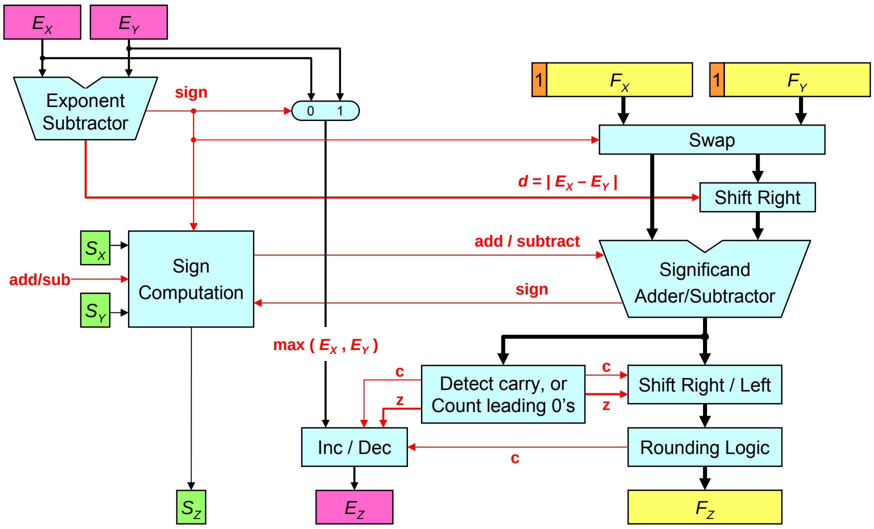
Shift significand right by $d = | E_X - E_Y |$

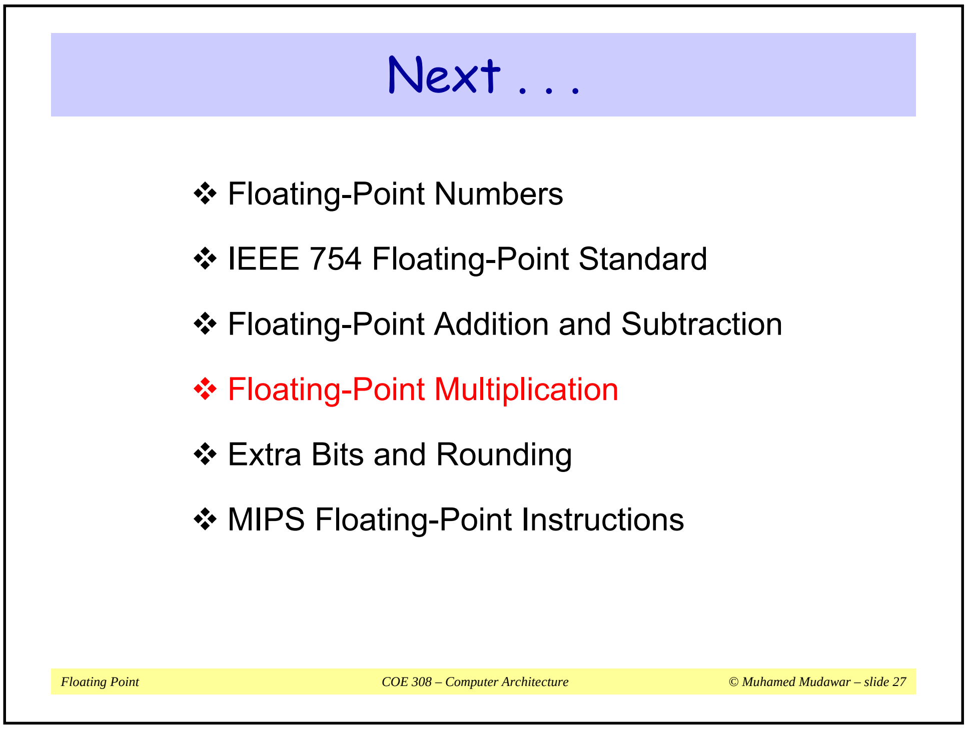
Add significands when signs of X and Y are identical, Subtract when different $X - Y$ becomes $X + (-Y)$

Normalization shifts right by 1 if there is a carry, or shifts left by the number of leading zeros in the case of subtraction

Rounding either truncates fraction, or adds a 1 to least significant fraction bit

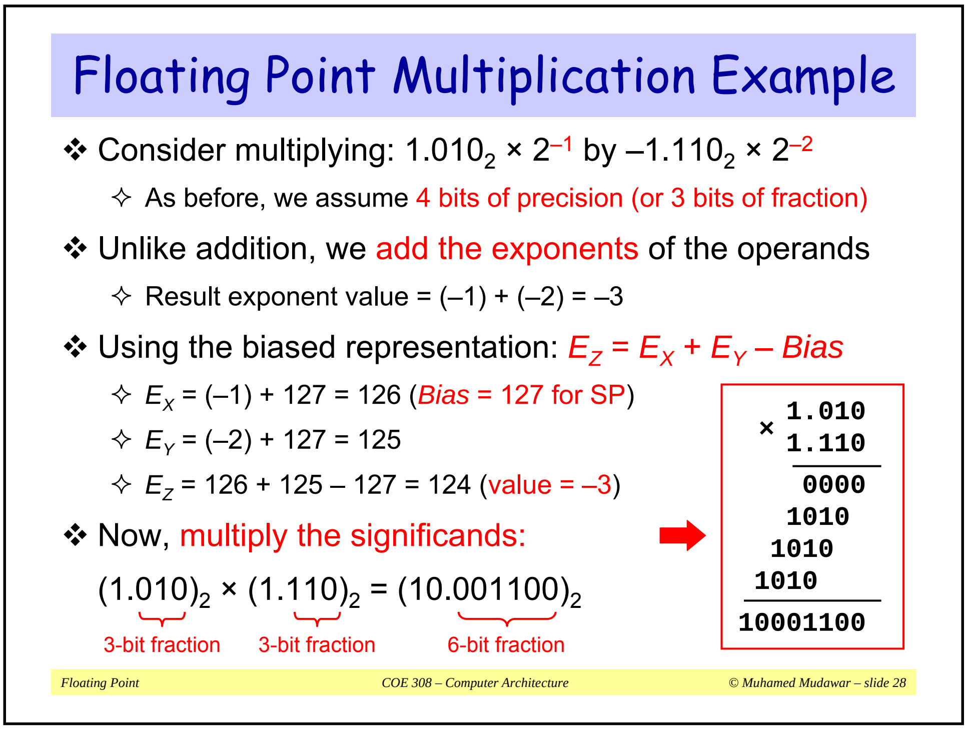
Floating Point Adder Block Diagram





Next . . .

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ Extra Bits and Rounding
- ❖ MIPS Floating-Point Instructions



Floating Point Multiplication Example

- ❖ Consider multiplying: $1.010_2 \times 2^{-1}$ by $-1.110_2 \times 2^{-2}$
  - ✧ As before, we assume 4 bits of precision (or 3 bits of fraction)
- ❖ Unlike addition, we add the exponents of the operands
  - ✧ Result exponent value = (–1) + (–2) = –3
- ❖ Using the biased representation: $E_Z = E_X + E_Y - Bias$
  - ✧ $E_X = (-1) + 127 = 126$ ($Bias$ = 127 for SP)
  - ✧ $E_Y = (-2) + 127 = 125$
  - ✧ $E_Z = 126 + 125 - 127 = 124$ (value = –3)
- ❖ Now, multiply the significands:

$$(1.\underbrace{010}_{\text{3-bit fraction}})_2 \times (1.\underbrace{110}_{\text{3-bit fraction}})_2 = (10.\underbrace{001100}_{\text{6-bit fraction}})_2$$

```
    1.010
 ×  1.110
 --------
     0000
    1010
   1010
  1010
 --------
 10001100
```

Multiplication Example – cont'd

- Since sign $S_X \neq S_Y$, sign of product $S_Z = 1$ (negative)
- So, $1.010_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} = -10.001100_2 \times 2^{-3}$
- However, result: $-10.001100_2 \times 2^{-3}$ is NOT normalized
- Normalize: $10.001100_2 \times 2^{-3} = 1.0001100_2 \times 2^{-2}$
  - Shift right by 1 bit and increment the exponent
  - At most 1 bit can be shifted right … Why?
- Round the significand to nearest:

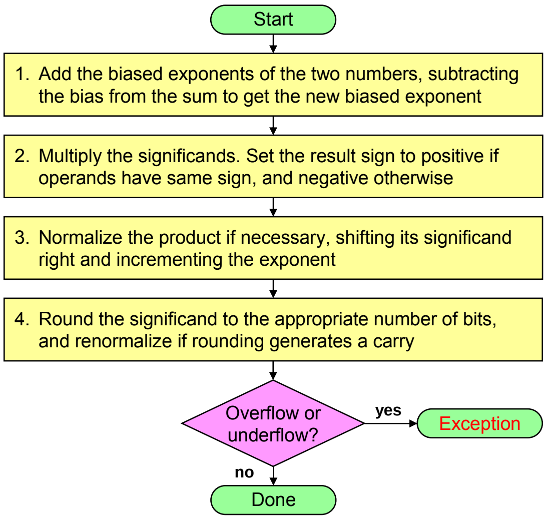
  $1.0001100_2 \approx 1.001_2$ (3-bit fraction)

  Result $\approx -1.001_2 \times 2^{-2}$ (normalized)

```
  1.000 1100
+     1
  --------
  1.001
```

- Detect overflow / underflow
  - No overflow / underflow because exponent is within range

Floating Point Multiplication



Start

1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands. Set the result sign to positive if operands have same sign, and negative otherwise
3. Normalize the product if necessary, shifting its significand right and incrementing the exponent
4. Round the significand to the appropriate number of bits, and renormalize if rounding generates a carry

Overflow or underflow?

yes → Exception

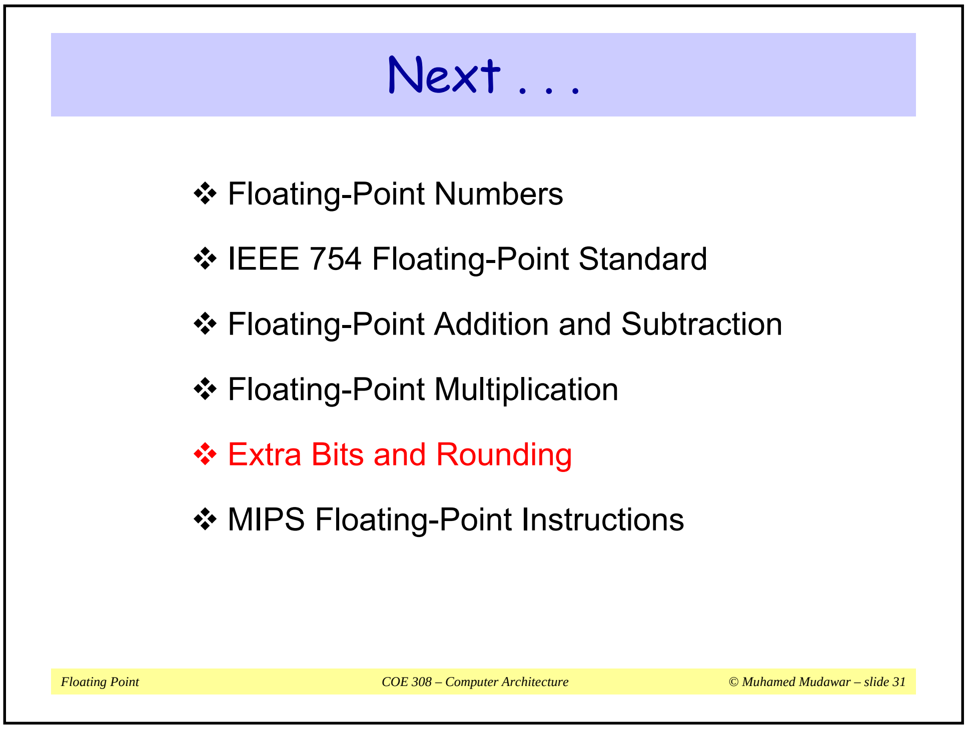
no → Done

Biased Exponent Addition
$E_Z = E_X + E_Y - Bias$

Result sign $S_Z = S_X$ **xor** S_Y can be computed independently

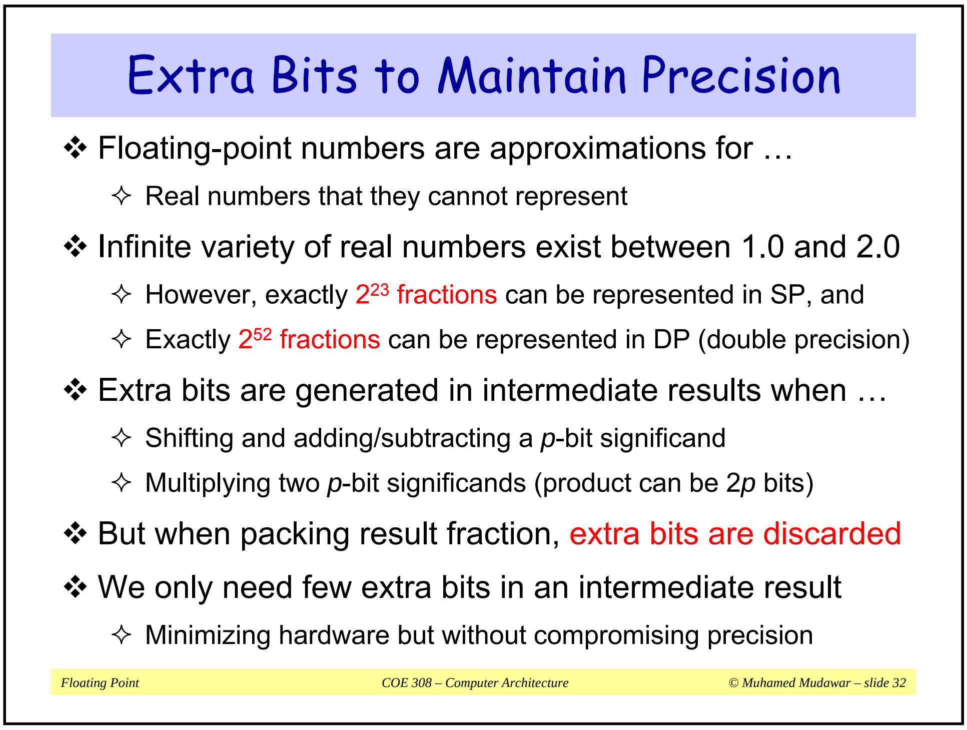
Since the operand significands $1.F_X$ and $1.F_Y$ are ≥ 1 and < 2, their product is ≥ 1 and < 4. To normalize product, we need to shift right by 1 bit only and increment exponent

Rounding either truncates fraction, or adds a 1 to least significant fraction bit



Next . . .

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ Extra Bits and Rounding
- ❖ MIPS Floating-Point Instructions

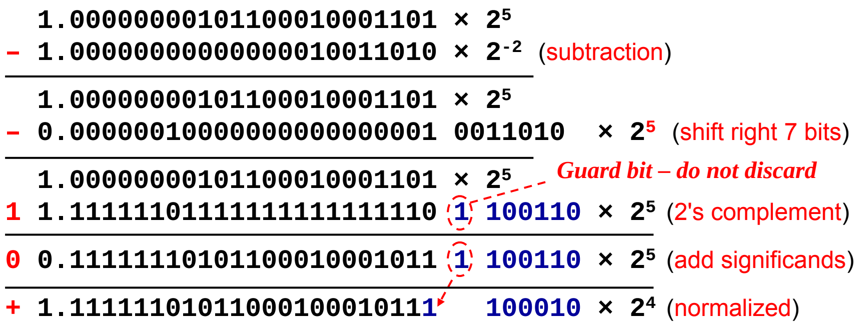


Extra Bits to Maintain Precision

- ❖ Floating-point numbers are approximations for …
  - ✧ Real numbers that they cannot represent
- ❖ Infinite variety of real numbers exist between 1.0 and 2.0
  - ✧ However, exactly 2^{23} fractions can be represented in SP, and
  - ✧ Exactly 2^{52} fractions can be represented in DP (double precision)
- ❖ Extra bits are generated in intermediate results when …
  - ✧ Shifting and adding/subtracting a *p*-bit significand
  - ✧ Multiplying two *p*-bit significands (product can be 2*p* bits)
- ❖ But when packing result fraction, extra bits are discarded
- ❖ We only need few extra bits in an intermediate result
  - ✧ Minimizing hardware but without compromising precision

Guard Bit

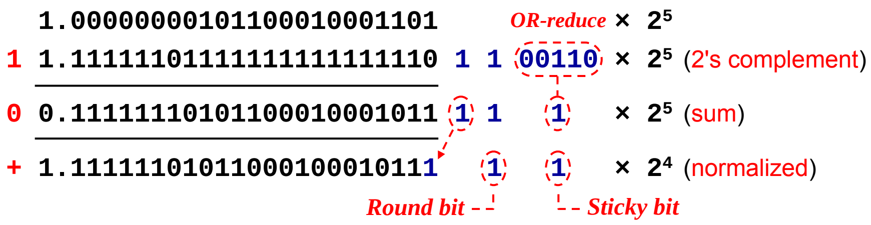
- Guard bit: guards against loss of a significant bit
  - Only one guard bit is needed to maintain accuracy of result
  - Shifted left (if needed) during normalization as last fraction bit
- Example on the need of a guard bit:



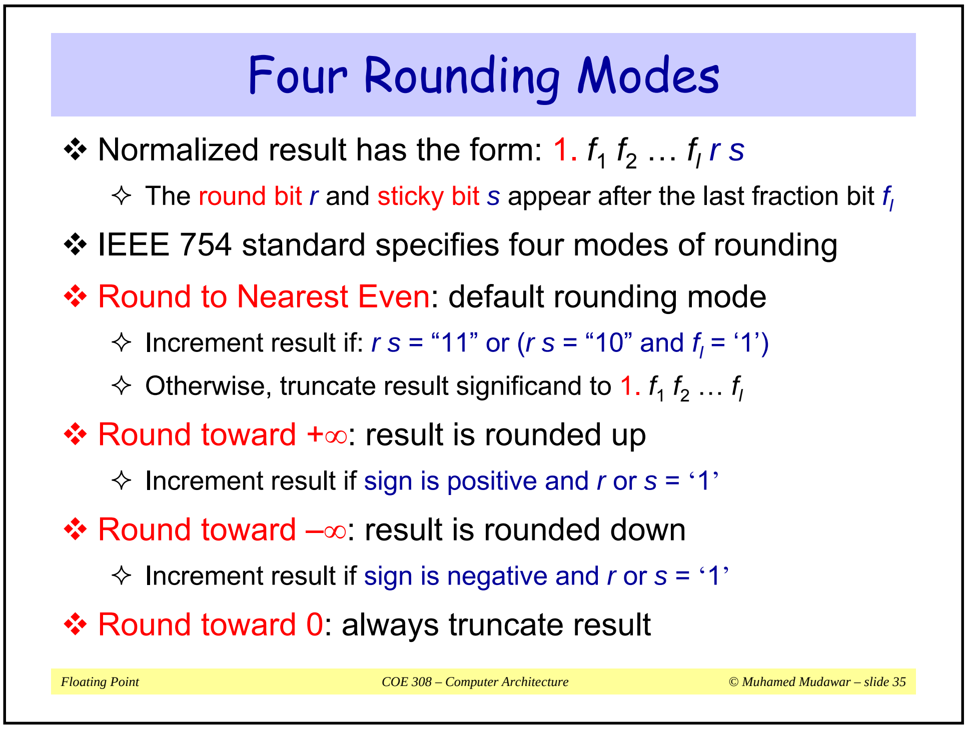
```
  1.00000000101100010001101 × 2^5
- 1.00000000000000010011010 × 2^-2  (subtraction)
-----------------------------------
  1.00000000101100010001101 × 2^5
- 0.00000010000000000000001 0011010   × 2^5  (shift right 7 bits)
-----------------------------------
  1.00000000101100010001101 × 2^5   Guard bit – do not discard
1 1.11111101111111111111110 1 100110 × 2^5 (2's complement)
-----------------------------------------------
0 0.11111110101100010001011 1 100110 × 2^5 (add significands)
-----------------------------------------------
+ 1.11111101011000100010111    100010 × 2^4 (normalized)
```

Round and Sticky Bits

- Two extra bits are needed for rounding
  - Just after normalizing a result significand
  - Round bit: appears just after the normalized significand
  - Sticky bit: appears after the round bit (OR of all additional bits)
  - Reduce the hardware and still achieve accurate arithmetic
  - As if result significand was computed exactly and rounded
- Consider the same example of previous slide:



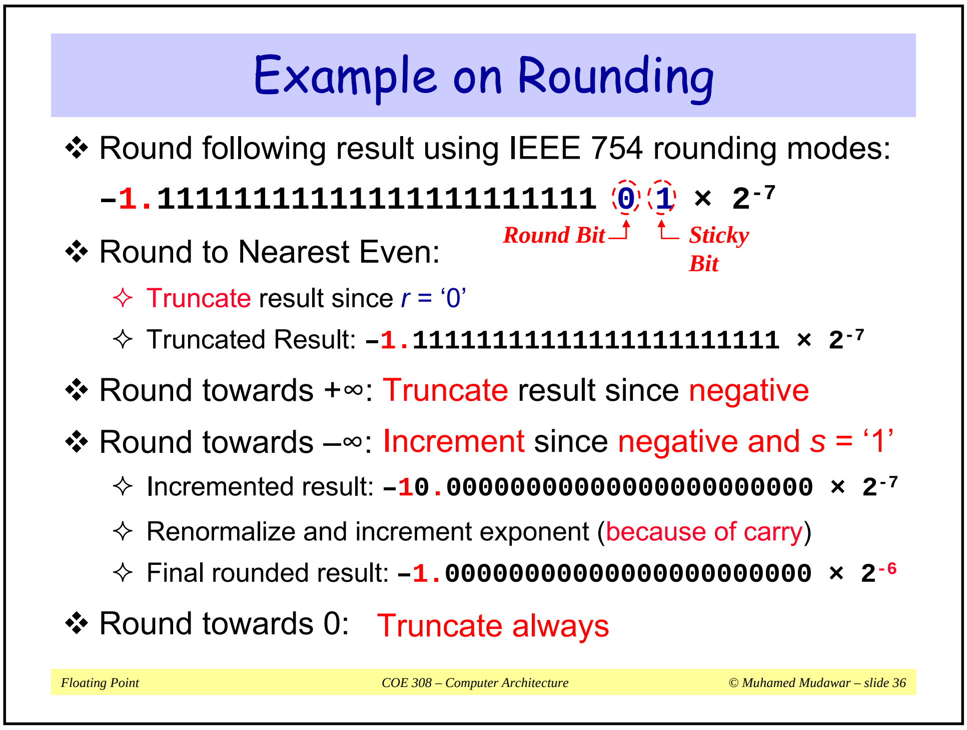
```
  1.00000000101100010001101        OR-reduce × 2^5
1 1.11111101111111111111110 1 1 00110 × 2^5 (2's complement)
-----------------------------
0 0.11111110101100010001011 1 1   1     × 2^5 (sum)
-----------------------------
+ 1.11111101011000100010111   1   1     × 2^4 (normalized)
                   Round bit - -   - - Sticky bit
```



Four Rounding Modes

- ❖ Normalized result has the form: 1. $f_1\ f_2 \ldots f_l\ r\ s$
  - ✧ The round bit r and sticky bit s appear after the last fraction bit f_l
- ❖ IEEE 754 standard specifies four modes of rounding
- ❖ Round to Nearest Even: default rounding mode
  - ✧ Increment result if: $r\ s$ = “11” or ($r\ s$ = “10” and f_l = ‘1’)
  - ✧ Otherwise, truncate result significand to 1. $f_1\ f_2 \ldots f_l$
- ❖ Round toward +∞: result is rounded up
  - ✧ Increment result if sign is positive and r or s = ‘1’
- ❖ Round toward –∞: result is rounded down
  - ✧ Increment result if sign is negative and r or s = ‘1’
- ❖ Round toward 0: always truncate result

Floating Point COE 308 – Computer Architecture © Muhamed Mudawar – slide 35



Example on Rounding

- ❖ Round following result using IEEE 754 rounding modes:

  –1.11111111111111111111111 0 1 × 2^{-7}

  Round Bit Sticky Bit
- ❖ Round to Nearest Even:
  - ✧ Truncate result since r = ‘0’
  - ✧ Truncated Result: **–1.11111111111111111111111 × 2^{-7}**
- ❖ Round towards +∞: Truncate result since negative
- ❖ Round towards –∞: Increment since negative and s = ‘1’
  - ✧ Incremented result: **–10.00000000000000000000000 × 2^{-7}**
  - ✧ Renormalize and increment exponent (because of carry)
  - ✧ Final rounded result: **–1.00000000000000000000000 × 2^{-6}**
- ❖ Round towards 0: Truncate always

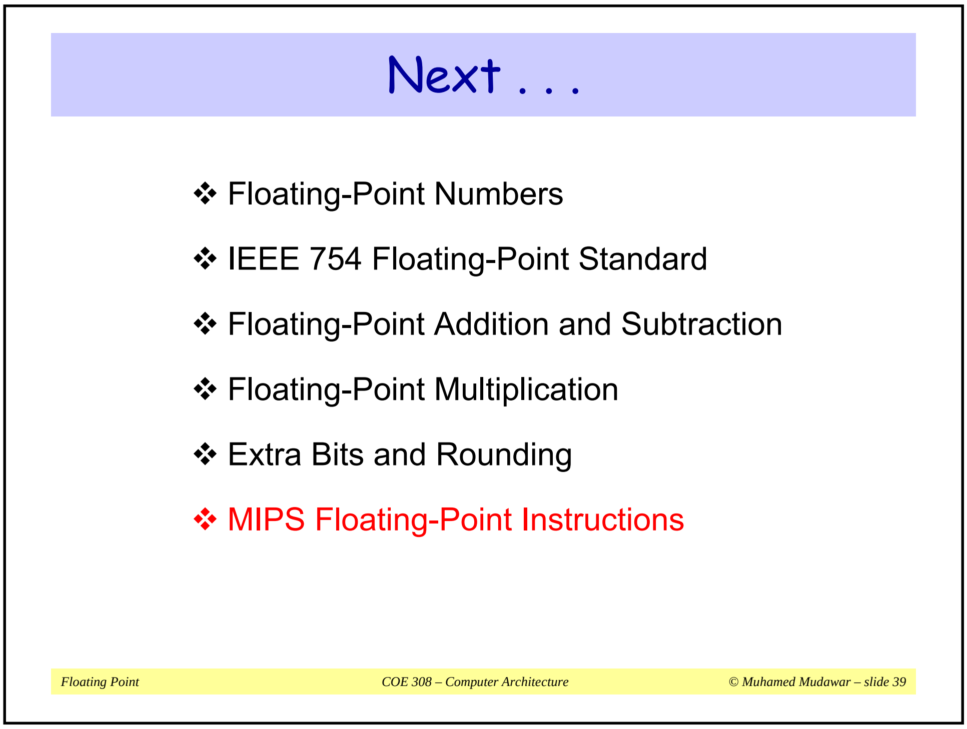
Floating Point COE 308 – Computer Architecture © Muhamed Mudawar – slide 36

Advantages of IEEE 754 Standard

- Used predominantly by the industry
- Encoding of exponent and fraction simplifies comparison
  - Integer comparator used to compare magnitude of FP numbers
- Includes special exceptional values: NaN and ±∞
  - Special rules are used such as:
    - 0/0 is NaN, *sqrt*(–1) is NaN, 1/0 is ∞, and 1/∞ is 0
  - Computation may continue in the face of exceptional conditions
- Denormalized numbers to fill the gap
  - Between smallest normalized number $1.0 \times 2^{E_{min}}$ and zero
  - Denormalized numbers , values $0.F \times 2^{E_{min}}$, are closer to zero
  - Gradual underflow to zero

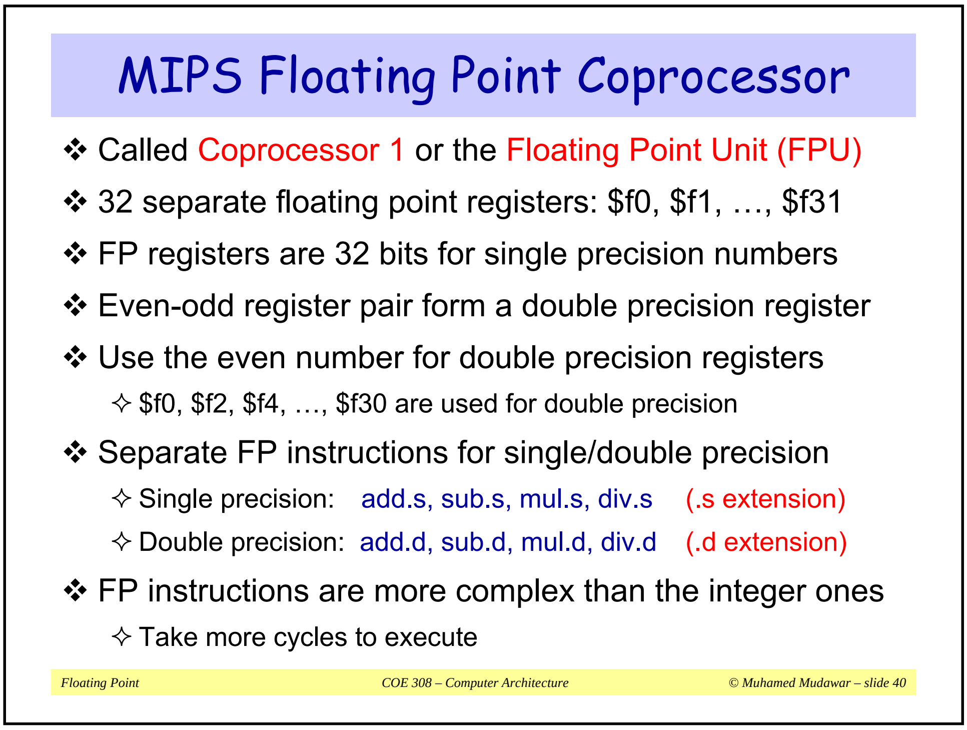
Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have underflow
- Accuracy can be a big problem
  - Extra bits to maintain precision: guard, round, and sticky
  - Four rounding modes
  - Division by zero yields Infinity
  - Zero divide by zero yields Not-a-Number
  - Other complexities
- Implementing the standard can be tricky
  - See text for description of 80x86 and Pentium bug!
- Not using the standard can be even worse



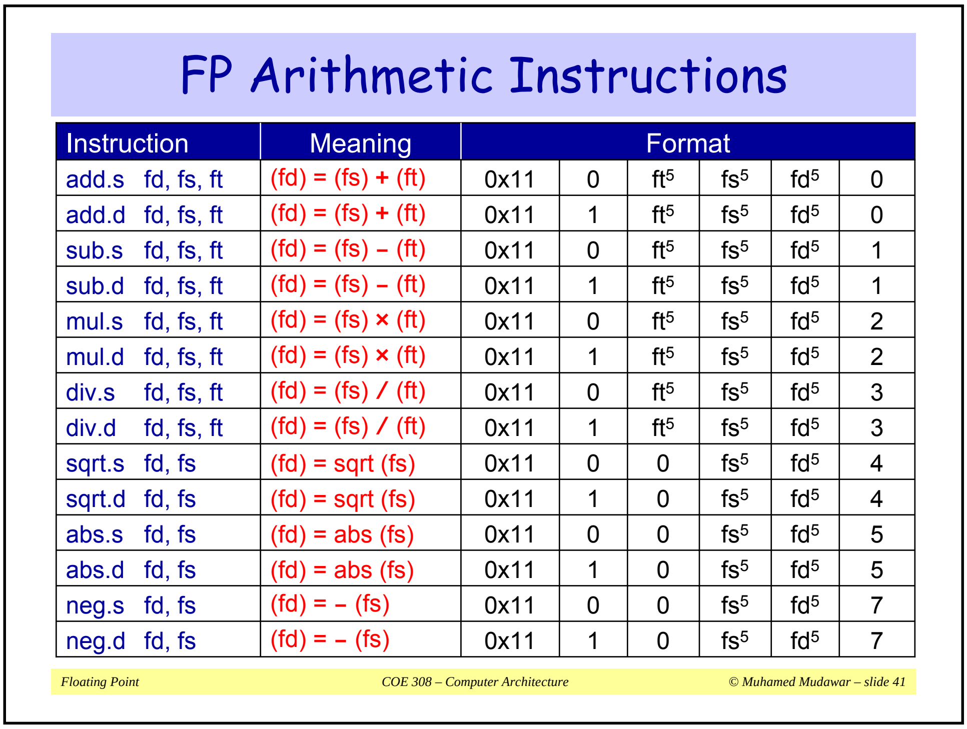
Next . . .

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ Extra Bits and Rounding
- ❖ MIPS Floating-Point Instructions



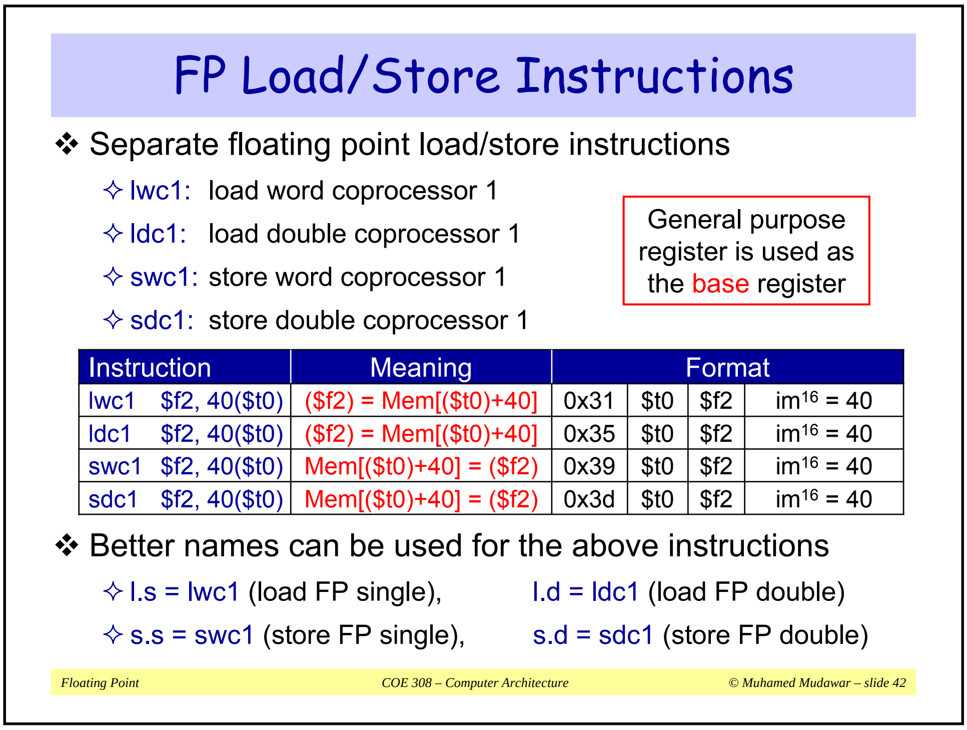
MIPS Floating Point Coprocessor

- ❖ Called Coprocessor 1 or the Floating Point Unit (FPU)
- ❖ 32 separate floating point registers: $f0, $f1, …, $f31
- ❖ FP registers are 32 bits for single precision numbers
- ❖ Even-odd register pair form a double precision register
- ❖ Use the even number for double precision registers
  - ✧ $f0, $f2, $f4, …, $f30 are used for double precision
- ❖ Separate FP instructions for single/double precision
  - ✧ Single precision: add.s, sub.s, mul.s, div.s (.s extension)
  - ✧ Double precision: add.d, sub.d, mul.d, div.d (.d extension)
- ❖ FP instructions are more complex than the integer ones
  - ✧ Take more cycles to execute



FP Arithmetic Instructions

Instruction	Meaning	Format					
add.s fd, fs, ft	(fd) = (fs) + (ft)	0x11	0	ft^5	fs^5	fd^5	0
add.d fd, fs, ft	(fd) = (fs) + (ft)	0x11	1	ft^5	fs^5	fd^5	0
sub.s fd, fs, ft	(fd) = (fs) – (ft)	0x11	0	ft^5	fs^5	fd^5	1
sub.d fd, fs, ft	(fd) = (fs) – (ft)	0x11	1	ft^5	fs^5	fd^5	1
mul.s fd, fs, ft	(fd) = (fs) × (ft)	0x11	0	ft^5	fs^5	fd^5	2
mul.d fd, fs, ft	(fd) = (fs) × (ft)	0x11	1	ft^5	fs^5	fd^5	2
div.s fd, fs, ft	(fd) = (fs) / (ft)	0x11	0	ft^5	fs^5	fd^5	3
div.d fd, fs, ft	(fd) = (fs) / (ft)	0x11	1	ft^5	fs^5	fd^5	3
sqrt.s fd, fs	(fd) = sqrt (fs)	0x11	0	0	fs^5	fd^5	4
sqrt.d fd, fs	(fd) = sqrt (fs)	0x11	1	0	fs^5	fd^5	4
abs.s fd, fs	(fd) = abs (fs)	0x11	0	0	fs^5	fd^5	5
abs.d fd, fs	(fd) = abs (fs)	0x11	1	0	fs^5	fd^5	5
neg.s fd, fs	(fd) = – (fs)	0x11	0	0	fs^5	fd^5	7
neg.d fd, fs	(fd) = – (fs)	0x11	1	0	fs^5	fd^5	7



FP Load/Store Instructions

❖ Separate floating point load/store instructions

✧ lwc1: load word coprocessor 1

✧ ldc1: load double coprocessor 1

✧ swc1: store word coprocessor 1

✧ sdc1: store double coprocessor 1

General purpose register is used as the base register

Instruction	Meaning	Format			
lwc1 $f2, 40($t0)	($f2) = Mem[($t0)+40]	0x31	$t0	$f2	im^{16} = 40
ldc1 $f2, 40($t0)	($f2) = Mem[($t0)+40]	0x35	$t0	$f2	im^{16} = 40
swc1 $f2, 40($t0)	Mem[($t0)+40] = ($f2)	0x39	$t0	$f2	im^{16} = 40
sdc1 $f2, 40($t0)	Mem[($t0)+40] = ($f2)	0x3d	$t0	$f2	im^{16} = 40

❖ Better names can be used for the above instructions

✧ l.s = lwc1 (load FP single), l.d = ldc1 (load FP double)

✧ s.s = swc1 (store FP single), s.d = sdc1 (store FP double)

FP Data Movement Instructions

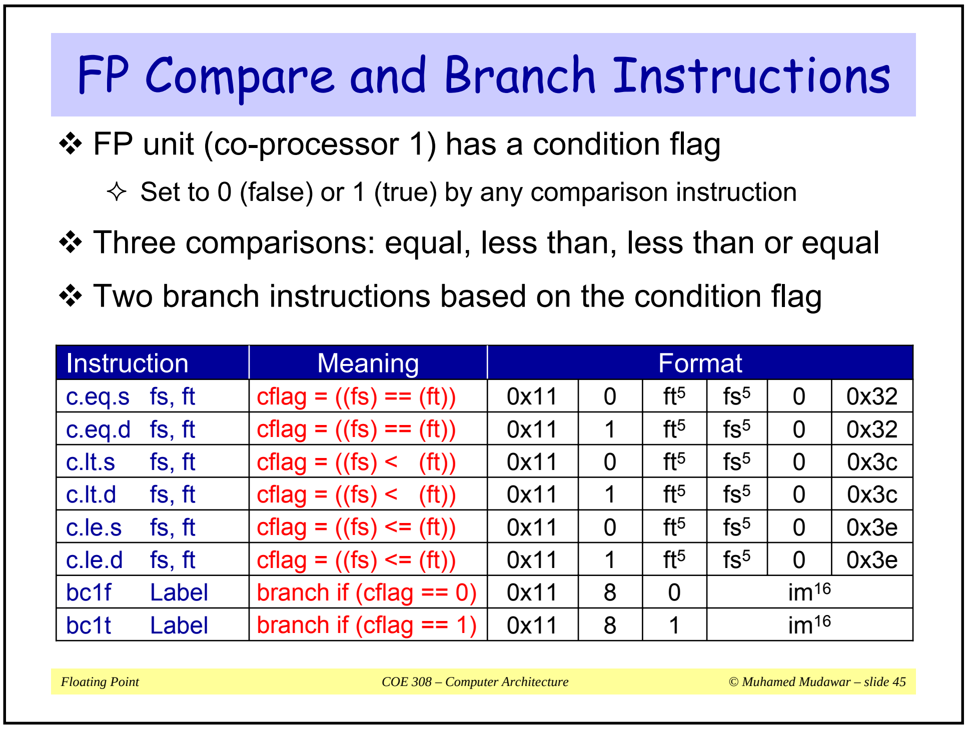
- Moving data between general purpose and FP registers
  - mfc1: move from coprocessor 1 (to general purpose register)
  - mtc1: move to coprocessor 1 (from general purpose register)
- Moving data between FP registers
  - mov.s: move single precision float
  - mov.d: move double precision float = even/odd pair of registers

Instruction	Meaning	Format					
mfc1 $t0, $f2	($t0) = ($f2)	0x11	0	$t0	$f2	0	0
mtc1 $t0, $f2	($f2) = ($t0)	0x11	4	$t0	$f2	0	0
mov.s $f4, $f2	($f4) = ($f2)	0x11	0	0	$f2	$f4	6
mov.d $f4, $f2	($f4) = ($f2)	0x11	1	0	$f2	$f4	6

FP Convert Instructions

- Convert instruction: cvt.x.y
  - Convert to destination format x from source format y
- Supported formats
  - Single precision float = .s (single precision float in FP register)
  - Double precision float = .d (double float in even-odd FP register)
  - Signed integer word = .w (signed integer in FP register)

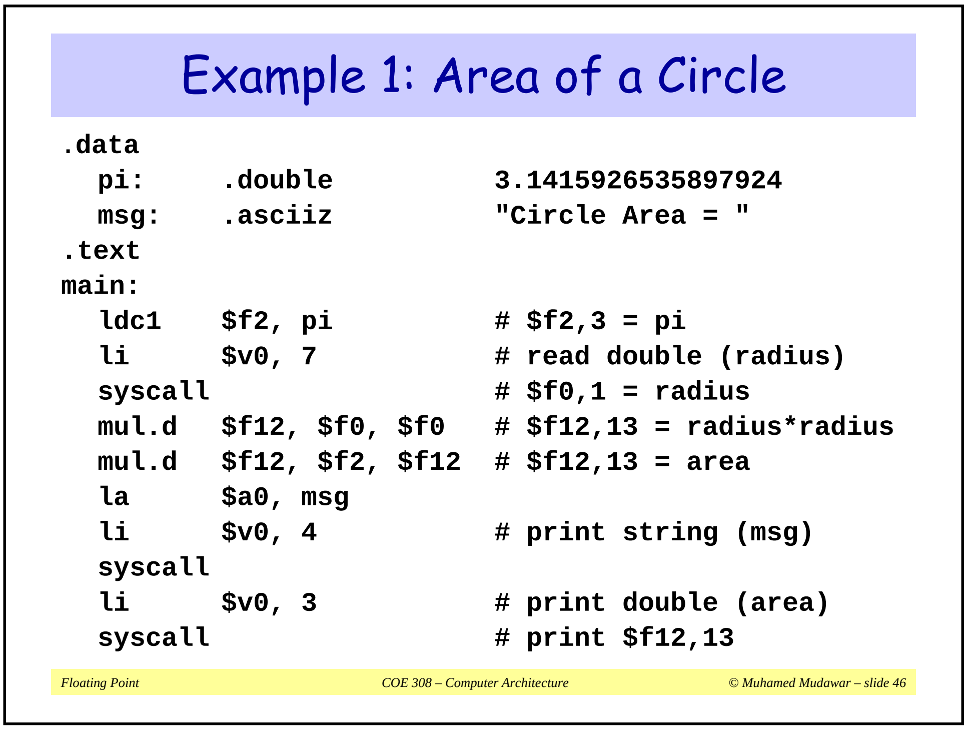
Instruction	Meaning	Format					
cvt.s.w fd, fs	to single from integer	0x11	0	0	fs^5	fd^5	0x20
cvt.s.d fd, fs	to single from double	0x11	1	0	fs^5	fd^5	0x20
cvt.d.w fd, fs	to double from integer	0x11	0	0	fs^5	fd^5	0x21
cvt.d.s fd, fs	to double from single	0x11	1	0	fs^5	fd^5	0x21
cvt.w.s fd, fs	to integer from single	0x11	0	0	fs^5	fd^5	0x24
cvt.w.d fd, fs	to integer from double	0x11	1	0	fs^5	fd^5	0x24



FP Compare and Branch Instructions

- FP unit (co-processor 1) has a condition flag
  - Set to 0 (false) or 1 (true) by any comparison instruction
- Three comparisons: equal, less than, less than or equal
- Two branch instructions based on the condition flag

Instruction	Meaning	Format					
c.eq.s fs, ft	cflag = ((fs) == (ft))	0x11	0	ft^5	fs^5	0	0x32
c.eq.d fs, ft	cflag = ((fs) == (ft))	0x11	1	ft^5	fs^5	0	0x32
c.lt.s fs, ft	cflag = ((fs) < (ft))	0x11	0	ft^5	fs^5	0	0x3c
c.lt.d fs, ft	cflag = ((fs) < (ft))	0x11	1	ft^5	fs^5	0	0x3c
c.le.s fs, ft	cflag = ((fs) <= (ft))	0x11	0	ft^5	fs^5	0	0x3e
c.le.d fs, ft	cflag = ((fs) <= (ft))	0x11	1	ft^5	fs^5	0	0x3e
bc1f Label	branch if (cflag == 0)	0x11	8	0	im^{16}		
bc1t Label	branch if (cflag == 1)	0x11	8	1	im^{16}		



Example 1: Area of a Circle

```
.data
  pi:     .double          3.1415926535897924
  msg:    .asciiz          "Circle Area = "
.text
main:
  ldc1    $f2, pi          # $f2,3 = pi
  li      $v0, 7           # read double (radius)
  syscall                  # $f0,1 = radius
  mul.d   $f12, $f0, $f0   # $f12,13 = radius*radius
  mul.d   $f12, $f2, $f12  # $f12,13 = area
  la      $a0, msg
  li      $v0, 4           # print string (msg)
  syscall
  li      $v0, 3           # print double (area)
  syscall                  # print $f12,13
```

Example 2: Matrix Multiplication

```
void mm (int n, double x[n][n], y[n][n], z[n][n]) {
  for (int i=0; i!=n; i=i+1)
    for (int j=0; j!=n; j=j+1) {
      double sum = 0.0;
      for (int k=0; k!=n; k=k+1)
        sum = sum + y[i][k] * z[k][j];
      x[i][j] = sum;
    }
}
```

❖ Matrices **x**, **y**, and **z** are **n×n** double precision float

❖ Matrix size is passed in **$a0 = n**

❖ Array addresses are passed in **$a1**, **$a2**, and **$a3**

❖ What is the MIPS assembly code for the procedure?

Matrix Multiplication Procedure – 1/3

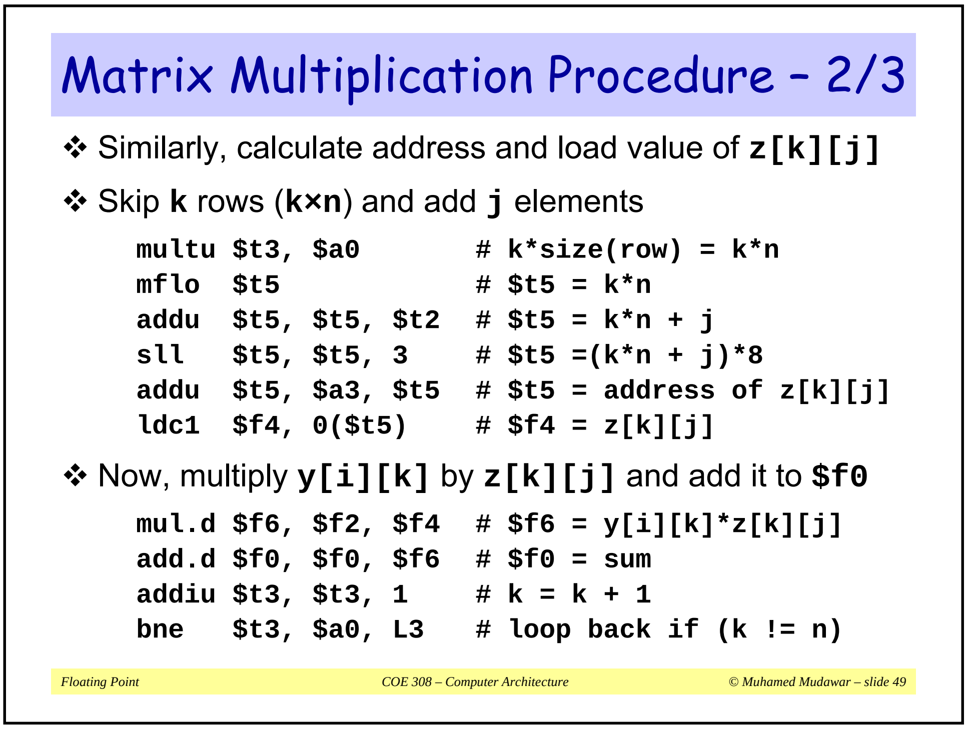
❖ Initialize Loop Variables

```
mm:  addu   $t1, $0, $0     # $t1 = i = 0; for 1st loop
L1:  addu   $t2, $0, $0     # $t2 = j = 0; for 2nd loop
L2:  addu   $t3, $0, $0     # $t3 = k = 0; for 3rd loop
     sub.d  $f0, $f0, $f0   # $f0 = sum = 0.0
```

❖ Calculate address of **y[i][k]** and load it into **$f2,$f3**

❖ Skip **i** rows (**i×n**) and add **k** elements

```
L3:  multu  $t1, $a0        # i*size(row) = i*n
     mflo   $t4             # $t4 = i*n
     addu   $t4, $t4, $t3   # $t4 = i*n + k
     sll    $t4, $t4, 3     # $t4 =(i*n + k)*8
     addu   $t4, $a2, $t4   # $t4 = address of y[i][k]
     ldc1   $f2, 0($t4)     # $f2 = y[i][k]
```



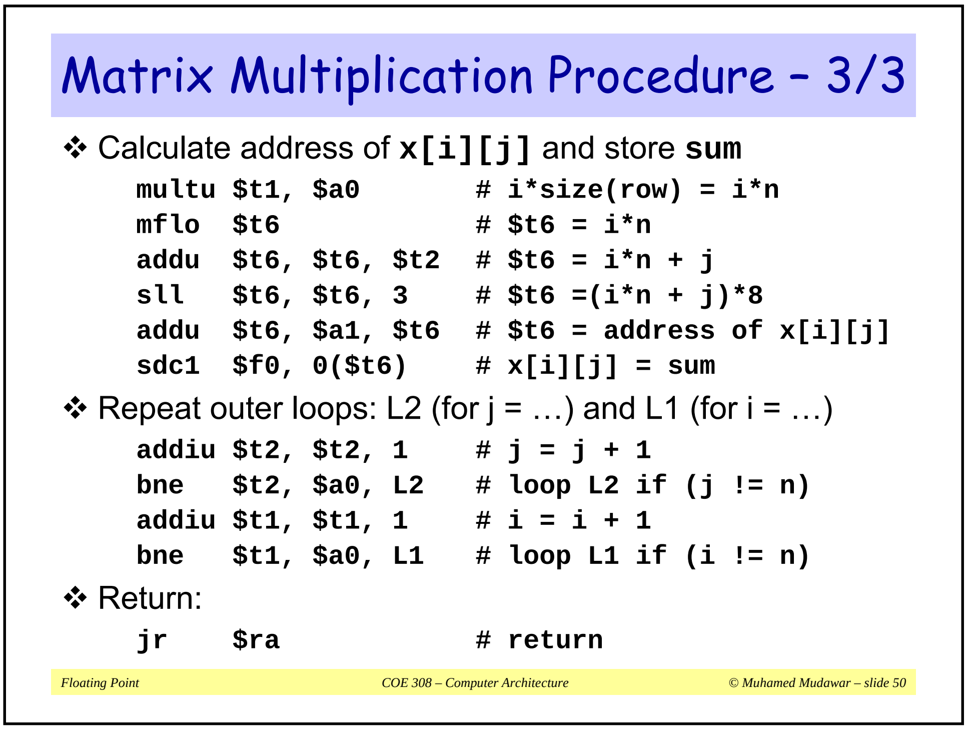
Matrix Multiplication Procedure - 2/3

- Similarly, calculate address and load value of **`z[k][j]`**
- Skip **`k`** rows (**`k×n`**) and add **`j`** elements

```
multu $t3, $a0       # k*size(row) = k*n
mflo  $t5            # $t5 = k*n
addu  $t5, $t5, $t2  # $t5 = k*n + j
sll   $t5, $t5, 3    # $t5 =(k*n + j)*8
addu  $t5, $a3, $t5  # $t5 = address of z[k][j]
ldc1  $f4, 0($t5)    # $f4 = z[k][j]
```

- Now, multiply **`y[i][k]`** by **`z[k][j]`** and add it to **`$f0`**

```
mul.d $f6, $f2, $f4  # $f6 = y[i][k]*z[k][j]
add.d $f0, $f0, $f6  # $f0 = sum
addiu $t3, $t3, 1    # k = k + 1
bne   $t3, $a0, L3   # loop back if (k != n)
```



Matrix Multiplication Procedure - 3/3

- Calculate address of **`x[i][j]`** and store **sum**

```
multu $t1, $a0       # i*size(row) = i*n
mflo  $t6            # $t6 = i*n
addu  $t6, $t6, $t2  # $t6 = i*n + j
sll   $t6, $t6, 3    # $t6 =(i*n + j)*8
addu  $t6, $a1, $t6  # $t6 = address of x[i][j]
sdc1  $f0, 0($t6)    # x[i][j] = sum
```

- Repeat outer loops: L2 (for j = …) and L1 (for i = …)

```
addiu $t2, $t2, 1    # j = j + 1
bne   $t2, $a0, L2   # loop L2 if (j != n)
addiu $t1, $t1, 1    # i = i + 1
bne   $t1, $a0, L1   # loop L1 if (i != n)
```

- Return:

```
jr    $ra            # return
```