Chapter 3 Part 2
Arithmetic for Computers
-Floating Point

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers

  4,600,000,000 or 4.6×10^{9}

  0.0000000000000000000000000166 or 1.6×10^{-27}
- Like scientific notation
  - 2×10^{-7}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C

Can't use integer to represent

normalized

not normalized

```
float  a;  // single precision
double b;  //double precision
```

Floating Point Standard- IEEE Std 754-1985

- Single precision - 32-bit

single: 8 bits   single: 23 bits

S	Exponent	Fraction

Significand=1 +fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

$$x = (-1)^S \times (\text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalized number $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - Always has a leading 1, so no need to represent it explicitly (hidden bit)
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single precision: Bias = 127, Double precision: Bias = 1023

Floating-Point Example – single-precision

What number is represented by the following single-precision float?

$x = 11000000101000...00_2$ (32-bit)

- $S = 1$
- Fraction = $01000...00_2$
- Exponent = $10000001_2 = 129$

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

$= (-1) \times (1+1/4) \times 2^2$

$= -5.0$

Hidden

Floating-Point Example

- Represent –0.75 in single-precision floating point
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = ?
  - Fraction = ?
  - Exponent = ?

Hidden 1 is not represented

Why uses bias (excess presentation) in the exponents

- Easier to compare which exponent is larger
  - Just need to check the bit from left to right

8 bits		Bias=127		
127	01111111	254	11111110	
126	01111110	253	11111101	
		.....		
....		......		
1	00000001	128	10000000	
0	00000000	127	01111111	
-1	111111111	....		
....				
-126	10000010	1	00000001	
-127	10000001	0	00000000	reserved
-128	10000000	255	11111111	reserved

Floating Point Standard- IEEE Std 754-1985

- Double precision (64-bit)

double: 11 bits double: 52 bits

S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

$$x = (-1)^S \times (\text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- Normalized number $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - Have hidden 1 Fraction=Significand-1
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Double: Bias = 1023

Floating-Point Example – double-precision

- What number is represented by the following double float?

  x=$1011111111011000...00_2$(64-bit)

  - S = 1
  - Fraction = $1000...00_2$
  - Exponent = 01111111101_2

- x = $(-1)^1 \times (1 + .1_2) \times 2^{(1021 - 1023)}$

  $= (-1) \times (1+1/2) \times 2^{-2}$

  $= -3/8$

Hidden

Floating-Point Example

- Represent –0.75 in double-precision floating point
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S =?
  - Fraction =?
  - Exponent = ?

Hidden 1 is not represented

IEEE 754 Encoding of FP number

- Encoding
  - Exp. 00…00 and 111…11 reserved
  - Exp.=00000000 and Fract.=00000…00 => 0
  - Exp.=0, and Fract. != 0 => denormalized number (discuss later)
  - Exp.=111..111 and Fract.= 000...000 => $\pm\infty$(discuss later)
  - Exp.=111…111 and Fract.!=0 => Non a Number (NaN) (discuss later)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Denormalized Numbers

- (Review) Smallest normalized value
  - 00000001 00000000……0000
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - Exponent = 1 – 127 = –126
  - Smallest value = 1.0×2^{-126}
- How to represent number smaller than $1.0\text{x}2^{-126}$?
- E.g. $0.5\text{x}2^{-126}$ =>Use denormalized number

S	Exponent	Fraction

Denormalized Numbers (32-bit)

- Exponent = 00000000
- Fraction $\Rightarrow$ hidden bit is 0

$$x = (-1)^S \times (\text{Fraction}) \times 2^{-126}$$

$0.5x2^{-126}$ = 0 00000000 10000000000000…000

- Allow for gradual underflow, with diminishing precision
- Denormalized with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-126} = \pm 0.0$$

Two representations of 0.0!

Special number: Infinities and NaNs

- Exponent = 111…1, Fraction = 000…0
  - ±∞
  - Can be used in subsequent calculations, avoiding need for overflow check
  - E.g. F+(+∞)=+∞ , or F/∞=0
- Exponent = 111…1, Fraction ≠ 000…0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Hidden

Example

- Smallest positive single precision normalized number

$1.00000000\ldots00000_2 \times 2^{-126}$

S　Exp　　　　Fraction

-　-------------　　--

0 0000 0001　　0000 0000 0000 0000 0000 000

2^{-126}

- Smallest positive single precision denormalized no. (Hint: Fraction is 23-bit)

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent

$$9.999 \times 10^{1} + 0.016 \times 10^{1}$$

- 2. Add significands

$$9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$$

- 3. Normalize result & check for over/underflow

$$1.0015 \times 10^{2}$$

- 4. Round and renormalize if necessary

$$1.002 \times 10^{2}$$

Course Administration (4/24)

- 專題說明
- HW2 explanation
- HW2 is due on 5/1

Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$$

- 2. Add significands

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$$

- 3. Normalize result & check for over/underflow

$1.000_2 \times 2^{-4}$, with no over/underflow

- 4. Round and renormalize if necessary

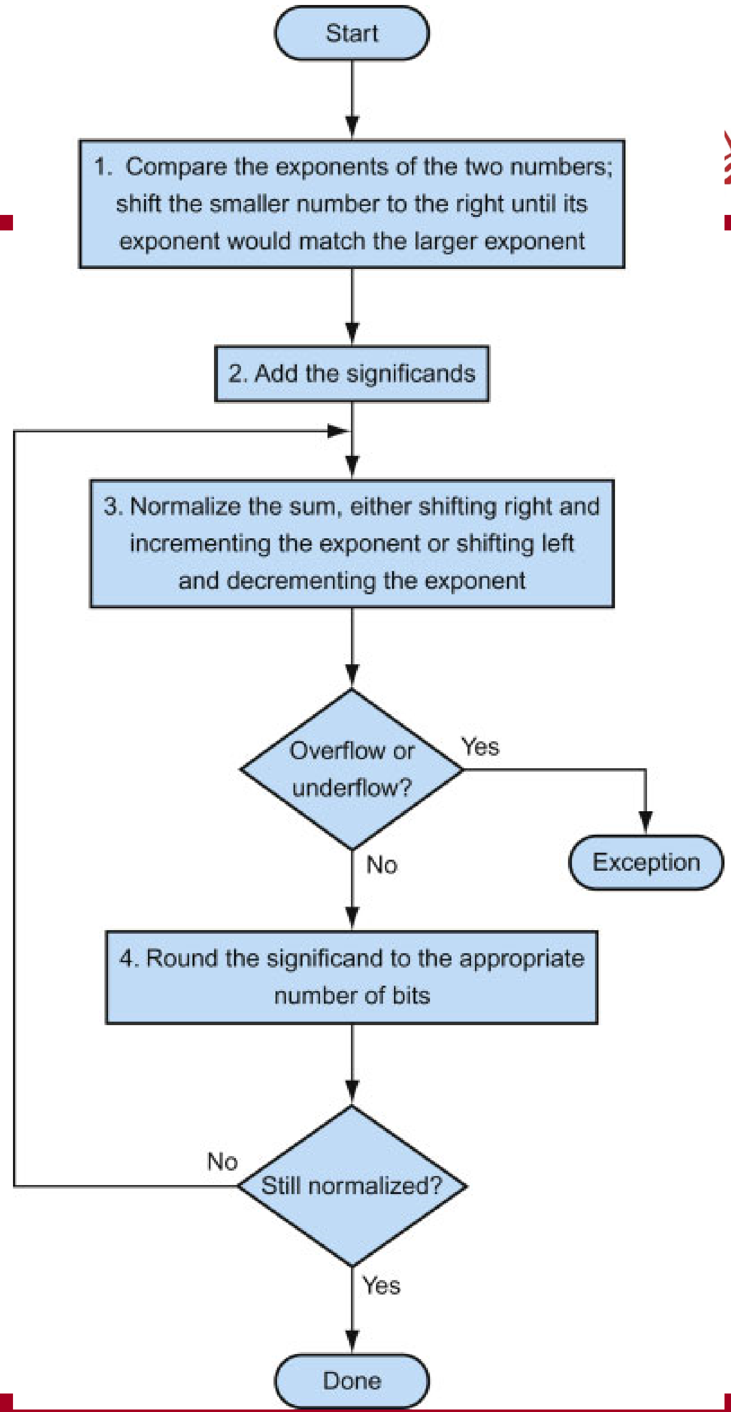
$1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

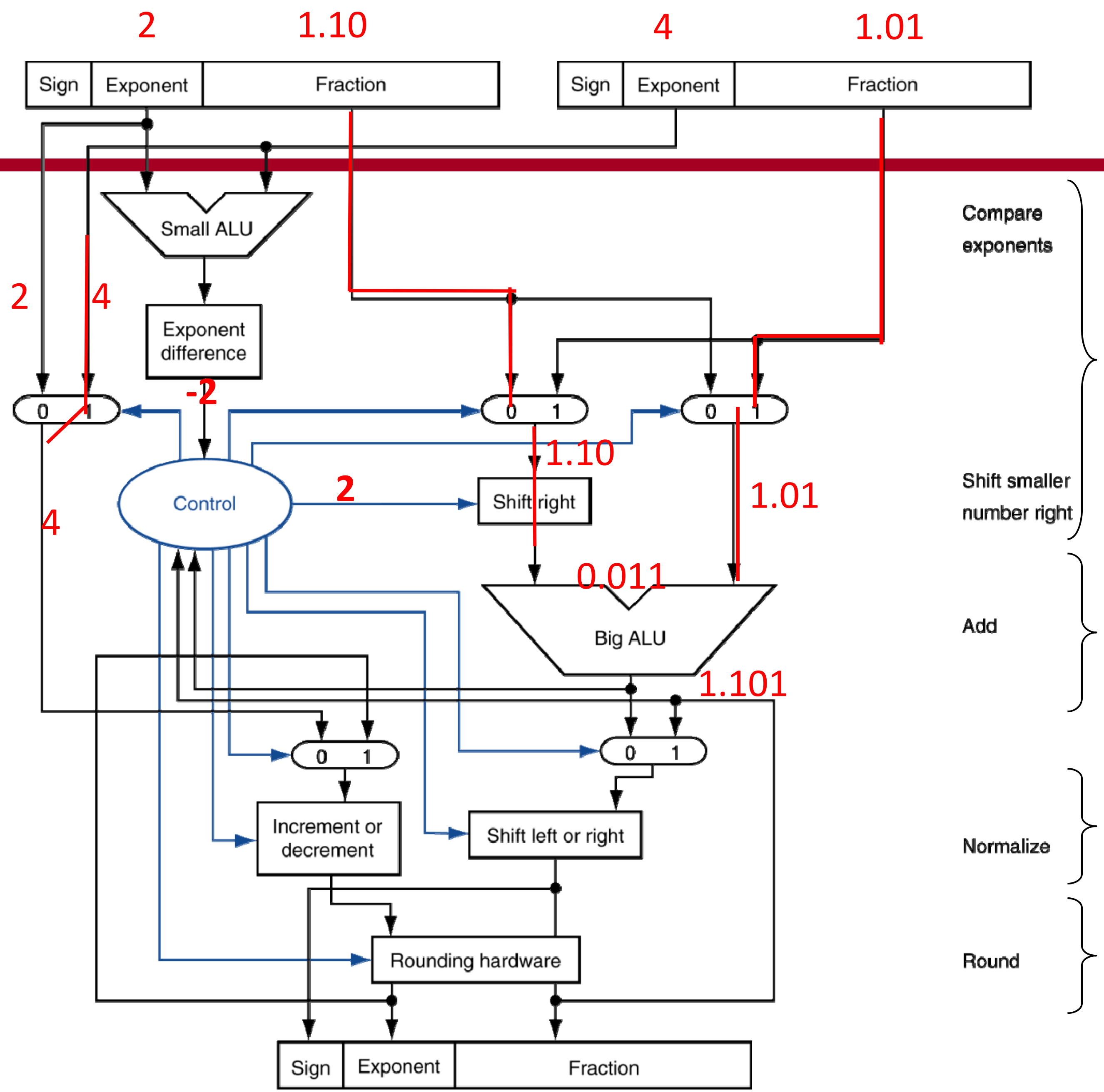
- Much more complex than integer adder
  - Steps includes shift exponents and fraction, add fraction, …, etc.
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined (see Chapter 4 about pipeline)

FP addition flow

Floating-point Addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.



FP Adder Hardware



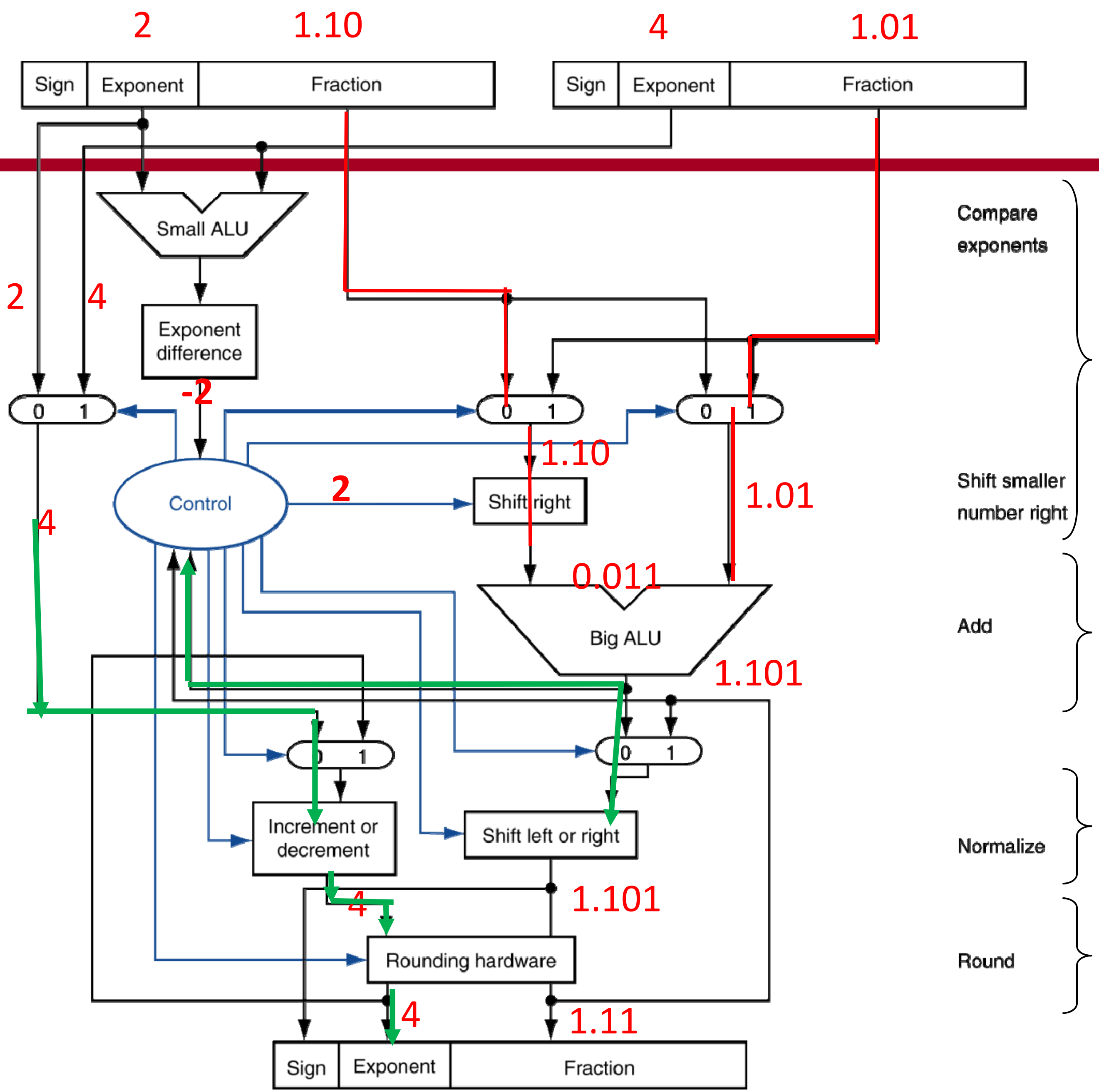
Step 1: Align binary points

Step 2: Add significands

Step 3:Normalize result & check for over/underflow

Step 4:Round and renormalize if necessary

FP Adder Hardware



Step 1: Align binary points

Step 2: Add significands

Step 3:Normalize result & check for over/underflow

Step 4:Round and renormalize if necessary

Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example

  $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127)$ -127 $= -3 + 254 - 127$

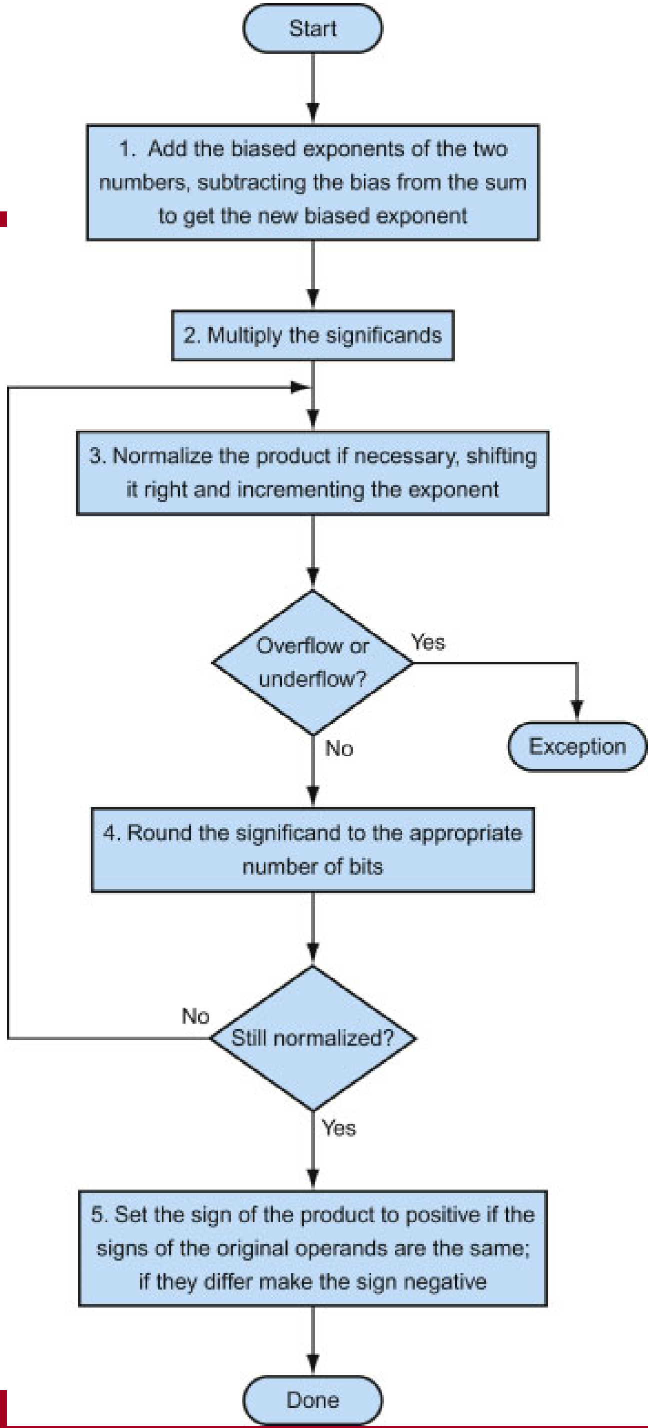
  Remove one bias
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+\text{ve} \times -\text{ve} \Rightarrow -\text{ve}$
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But do multiplication for significands instead of an addition
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined (See Chapter 4)

FP Multiplication

The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.



FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers for single precision
  - 32 single-precision: $f0, $f1, … $f31
- FP instructions operate only on FP registers
- Single-precision FP load and store instructions
  - lwc1, swc1

    e.g., lwc1 $f8, 32($sp)
- Single-precision arithmetic
  - add.s, sub.s, mul.s, div.s

    e.g., add.s $f0, $f1, $f6
- Single-precision comparison
  - c.*xx*.s (*xx* is eq, lt, le, …)

    e.g. c.lt.s $f3, $f4
  - Sets or clears FP condition-code bit

FP Instructions in MIPS for double-precision

- Separate FP registers
  - 32 FP registers
  - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, …
- FP Double-precision load and store instructions
  - ldc1, sdc1
- Double-precision arithmetic
  - add.d, sub.d, mul.d, div.d

mul.d \$f4, \$f4, \$f6

- Double-precision comparison
  - c.*xx*.d (*xx* is eq, lt, le, …)
  - Sets or clears FP condition-code bit

c.lt.d \$f4, \$f6

- Branch on FP condition code true or false
  - bc1t, bc1f
    - e.g., bc1t TargetLabel

Improve Accuracy

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
- Guard & round bits: two extra (hidden) bits on the right during intermediate additions
  - Improve precision

Consider the addition $2.56 \times 10^0 + 2.34 \times 10^2 = 2.3656$

Without guard and round bit

$$0.02 \times 10^2 + 2.34 \times 10^2 = 2.36 \times 10^2$$

With guard and round bit

$$0.0256 \times 10^2 + 2.3400 \times 10^2 = 2.3656 \times 10^2 = 2.37 \times 10^2$$

closer to accurate answer

Improve Accuracy: sticky bit

- Sticky bit: one bit is set when there are nonzero bits to the right of the round bit.
  - Allow computer to see the difference between $0.50000..0_{10}$ and $0.50000..1_{10}$

- Without Sticky bit

  2.3450000000001 will be stored as 2.345

- With Sticky bit

  2.3450000000001 will be stored as 2.345 and sticky bit =1

- Used for rounding

  2.345 with sticky bit=1 is larger than 2.345

Associativity

- Is (x+y)+z equal to x+(y+z) ???

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Parallel Programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail
- Need to validate parallel programs under varying degrees of parallelism

Fallacy: Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i and thus right shift divides by 2^i

  Wrong, Only for unsigned integers

- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4 = -1 …. -1

$11111011_2 >> 2 = 00111110_2 = 62$ not -1

Interpretation of Data

The BIG Picture

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs
- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow

Backup slides