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}

Can't be represented in integer

- Like scientific notation
  - 2×10^{-7}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

normalized

- Types float and double in C

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

Floating Point Standard- IEEE Std 754-1985

- Single precision - 32-bit

Significand=1+fraction

single: 8 bits		single: 23 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 $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - 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?

$x = 11000000101000\ldots00_2$ (32-bit, single precision)

- S = 1
- Fraction = $01000\ldots00_2$
- 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

Floating-Point Example

- Represent –0.75 in single-precision floating point
  - $-0.75 = -(1/4+1/2)=(-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = 1000…00
  - Exponent = –1 + Bias=126=01111110_2

Hidden 1 is not represented

Answer: 1011111101000…00

Represent 3.4375×10^{-1} in single-precision floating point

$$3.4375 \times 10^{-1} = 0.3475$$
$$= 0.0101100 = 1.0110000000 \times 2^{-2}$$

0.34375 *2 =0.6875 ... (0)
0.8750 *2 = 1.375 .... (1)
0.375*2 = 0.75 ....(0)
0.75 *2 = 1.5 ... (1)
0.5 *2 = 1

- S = 0
- Fraction = 011000…00
- Exponent = –2+Bias (127)=125= 01111101_2

Answer: 001111101011000…00

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?

  $x=1011111111011000\ldots00_2$(64-bit)

  - S = 1
  - Fraction = $1000\ldots00_2$
  - 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 = ?

0.75*2 = 1.5 1

0.5 *2 = 1.0 1

Hidden 1 is not represented

Floating-Point Example

- Represent –0.75 in double-precision floating point
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias= -1+1023= 1022_{10}=01111111110_2

0.75*2 = 1.5 1
0.5 *2 = 1.0 1
0.75_{10}=0.11_2

Hidden 1 is not represented

Ans: 1011111111101000…00

Floating point - Half-precision

- Half precision

1 bit	5 bits	10 bits
S	Exponent	Fraction

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

- Bias = 15

Represent –0.75 in half-precision floating point

$-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$

•S = 1

•Fraction = 1000000000_2

•Exponent = –1 + 15=14 = 01110_2

Ans: $101110\ 1000000000_2$

IEEE 754 Encoding of FP number

- Exp.=0 and Fract.=0 => 0
- Exp.=0 and Fract. != 0 => denormalized number (discuss later)

- Exp.=111..111 and Fract.= 0 => ±∞ (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 (not 1)

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

$0.5\text{x}2^{-126}$: Exponent = 0 00000000

: Fraction = 1000000000000…000

$0.5\text{x}2^{-126}$ = 0 00000000 1000000000000…000

$0.25\text{x}2^{-126}$ = 0 00000000 0100000000000…000

- Allow for gradual underflow, with diminishing precision

Special number: Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - $\pm\infty$
  - 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)
  - Indicates Unrepresentable result (e.g. Sqrt(-4)
  - X op NaN = NaN, op can be +, -, *.....

Hidden

Example

- Smallest positive single precision normalized number

$1.00000000\ldots00000_2 \mathrm{x} 2^{-126}$

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)

Example

- Smallest positive single precision normalized number

$1.00000000...00000_2 x 2^{-126}$

S Exp Fraction

\- -------------- --

0 0000 0001 0000 0000 0000 0000 0000 000

2^{-126} x1.0

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

S Exp Fraction

\- -------------- --

0 0000 0000 0000 0000 0000 0000 0000 001

2^{-126} $\times 2^{-23}$

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

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

Why shift smaller number?

- $9.99999 \times 10^{40} + 1.610 \times 10^{-1}$
- If shifting 9.99999×10^{40} to 9999999999999….x 10^{-1}=> significand may be to large => overflow

- 1.610×10^{-1} to 10^{40} => significand may be very small, but still don't overflow, using round
- Therefore, we shift the smaller exponent

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

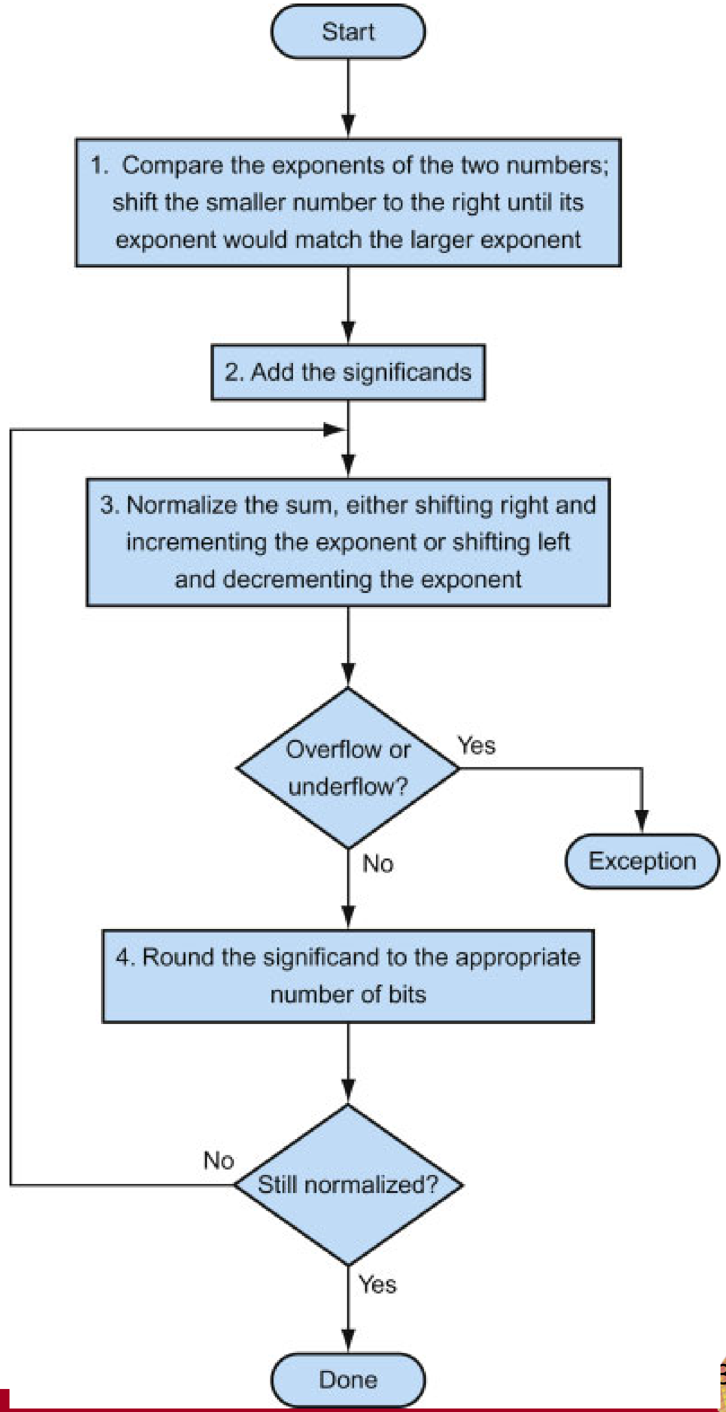
- Much more complex than integer adder
  - Steps includes shift exponents and fraction, add fraction, …, etc.
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined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

FP addition flow

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.

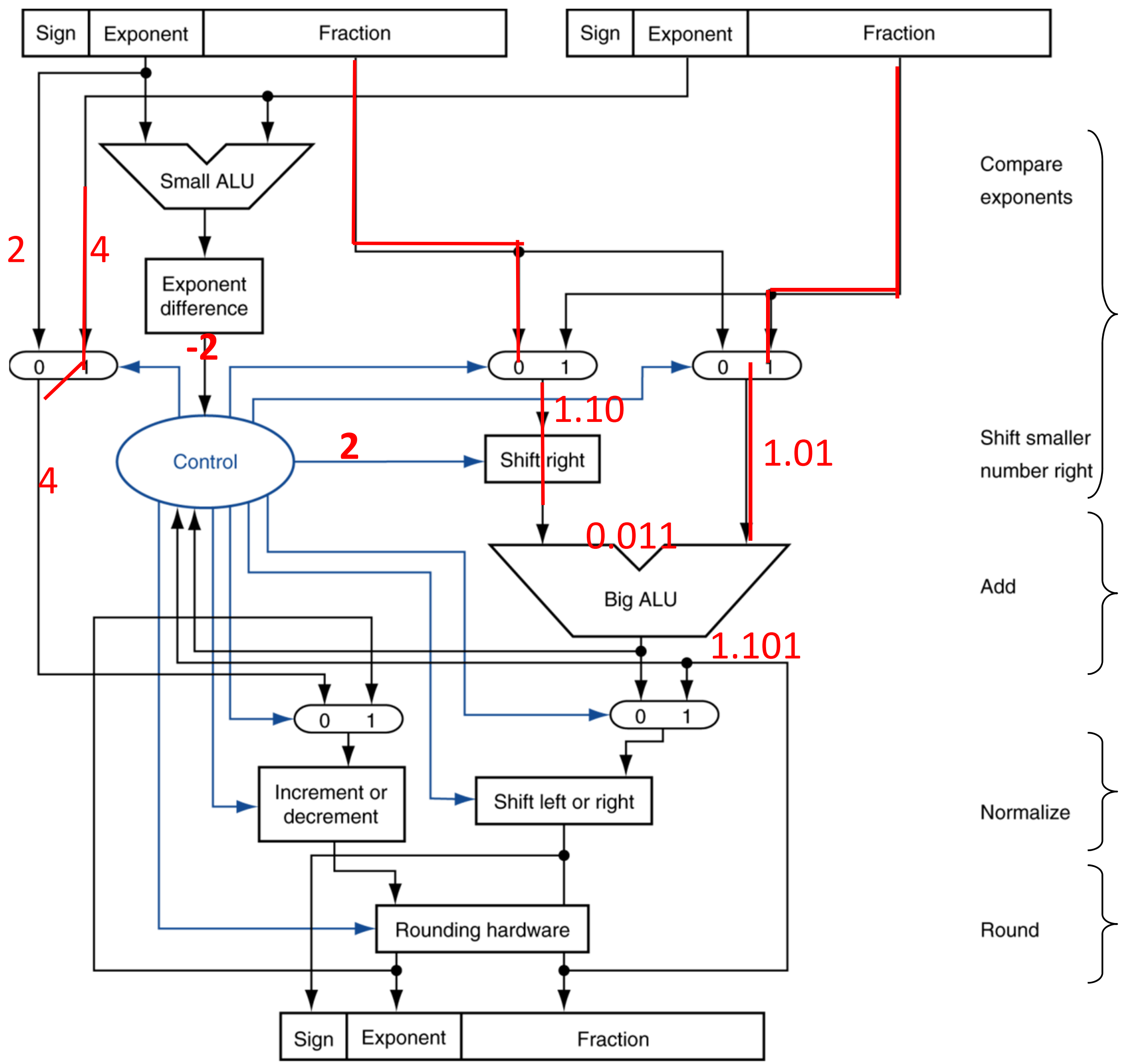
See an example in the next page

$1.10 \times 2^2 + 1.01 \times 2^4$



FP Adder Hardware

$1.10 \times 2^2 + 1.01 * 2^4$



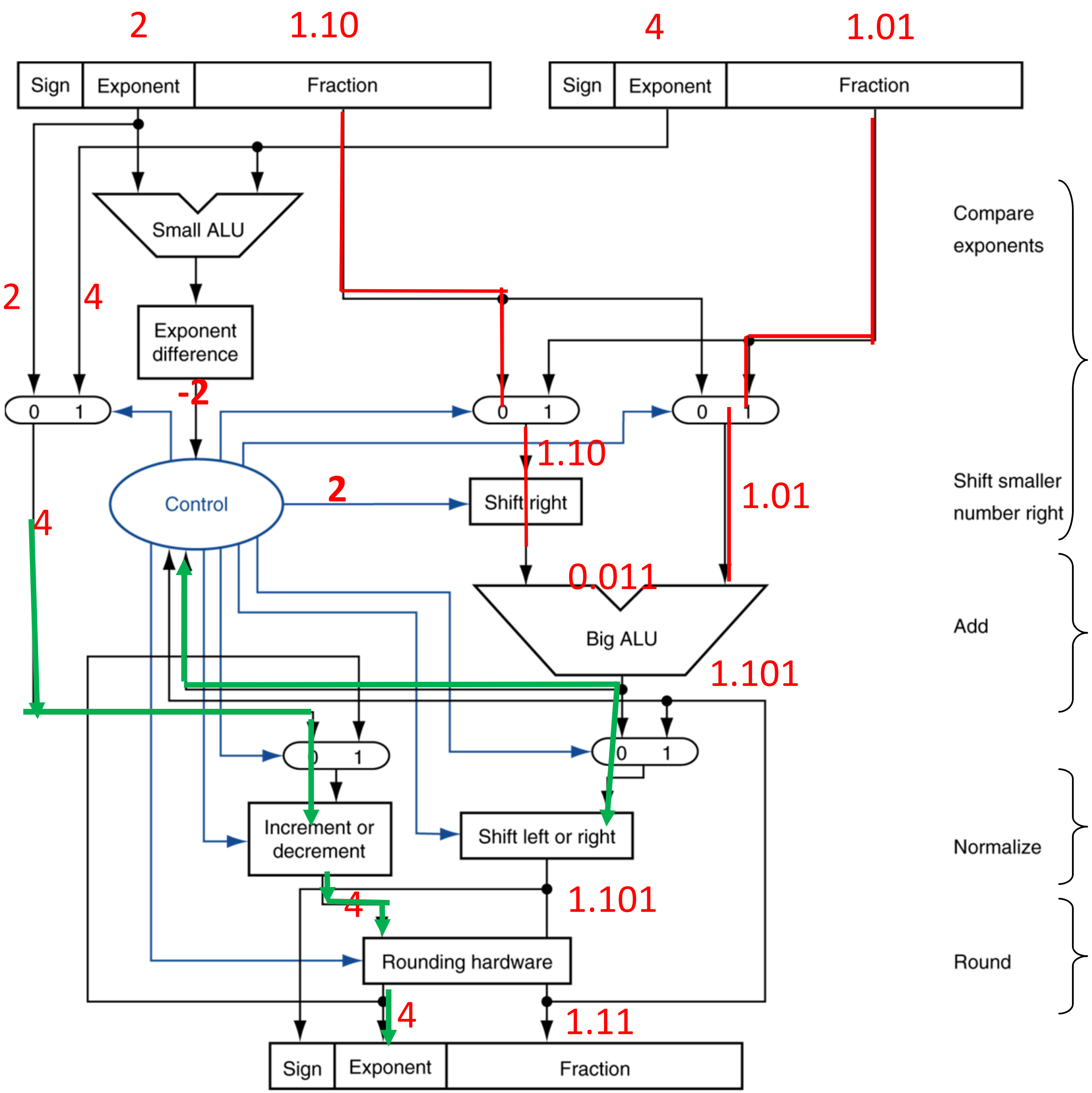
Step 1: Align binary points

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

   Remove one bias

   - Unbiased: $-1 + -2 = -3$
   - Biased: $(-1 + 127) + (-2 + 127)$ -127 $= -3 + 254 - 127$

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)

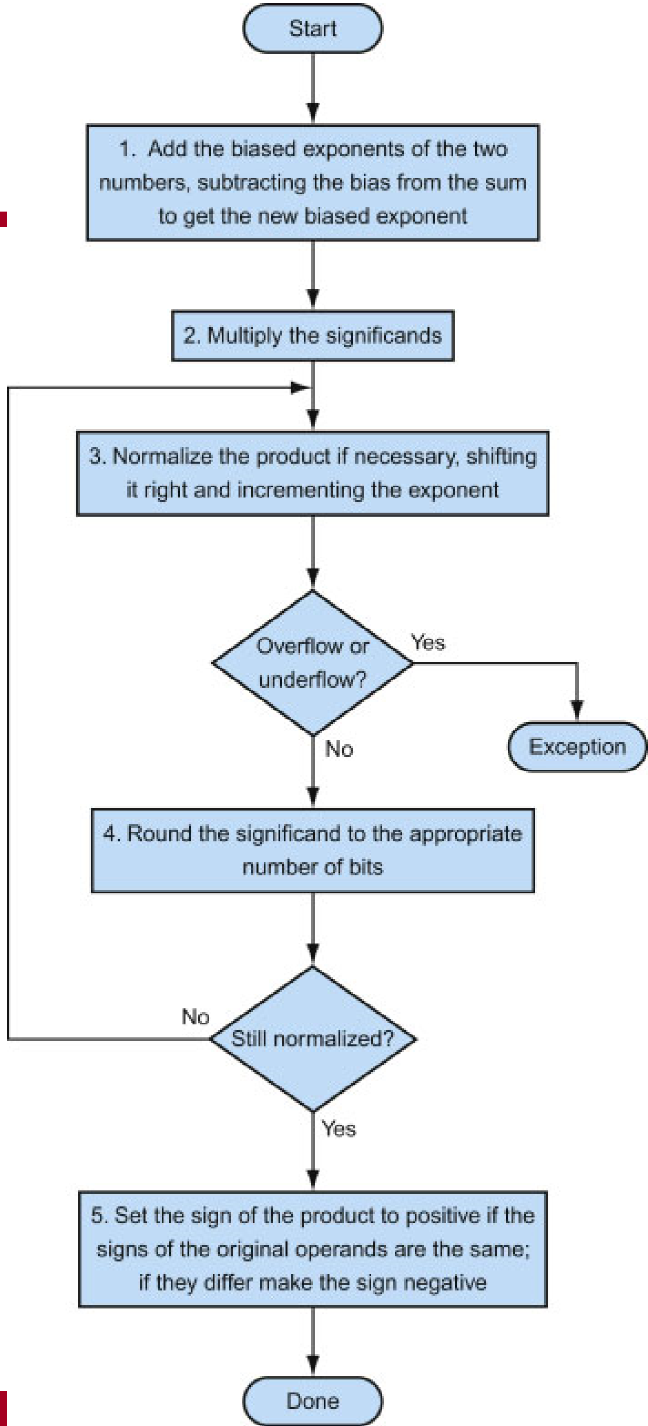
5. Determine sign: +ve $\times$ –ve $\Rightarrow$ –ve
   - $-1.110_2 \times 2^{-3} = -0.21875$

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.



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 $2.56 \times 10^0 + 2.34 \times 10^2 = 236.56$

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

Rounding: Round to nearest even

Guard, Round and Sticky bits are three bits are only used while doing calculations and aren't stored in the floating-point variable before or after the calculations.

GRS : Action
0xx : <0.5, round down = do nothing (x means any bit value, 0 or 1)
101 : >0.5, round up
110 : >0.5 round up
111 : >0.5 round up

GRS
01 100 ⇨ **10 ~~000~~**
00 100 ⇨ **00 ~~000~~**

100 - this is a **tie**: round up if the fraction's bit just before **G** is 1, else round down(=do nothing)

3.29 Calculate the sum of 2.6125×10^{1} and $4.150390625 \times 10^{-1}$ by hand, assuming both are stored in the 16-bit half precision. Assume 1 guard, 1 round bit, and 1 sticky bit and round to the nearest even.

$2.6125 \times 10^{1} + 4.150390625 \times 10^{-1}$

$2.6125 \times 10^{1} = 26.125 = 11010.001 = 1.1010001000 \times 2^{4}$

$4.150390625 \times 10^{-1} = .4150390625 = .0110101001 = 1.10101001 \times 2^{-2}$

For the second number, shift binary point 6 to the left to align exponents,

$1.1010100111 \times 2^{-2} \quad = 0.0000011010100111 \times 2^{4}$

GR

1.1010001000 00 (Guard bit = 0, Round bit = 0, Sticky bit= 0)

0.0000011010 10 0111 (Guard bit = 1, Round bit = 0, Sticky bit= 1) Right shift 6 bits

----

1.1010100010 101 (Guard bit = 1, Round bit = 0, Sticky bit= 1)

The extra bit (G,R,S) is more than half of the least significant bit (0).
Thus, the value is rounded up.

$1.1010100011 \times 2^{4} = 11010.100011 \times 2^{0} = 26.546875 = 2.6546875 \times 10^{1}$

Fallacy: Right Shift and Division

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

Correct for unsigned number, incorrect for signed number

- For unsigned number, this is correct

  00001011_2 >> 2 = 00000010_2 (11/4=2)

- For signed integers, this is incorrect
  - e.g., –5 / 4 = -1 …. -1

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

Pitfall: FP addition is not associative

- Is (x+y)+z equal to x+(y+z) ??? May be not true

		(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

Concluding Remarks

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

Backup slides