Systems Architecture

Lecture 14: Floating Point Arithmetic

Jeremy R. Johnson

Anatole D. Ruslanov

William M. Mongan

Introduction

- **Objective: To provide hardware support for floating point arithmetic. To understand how to represent floating point numbers in the computer and how to perform arithmetic with them. Also to learn how to use floating point arithmetic in MIPS.**

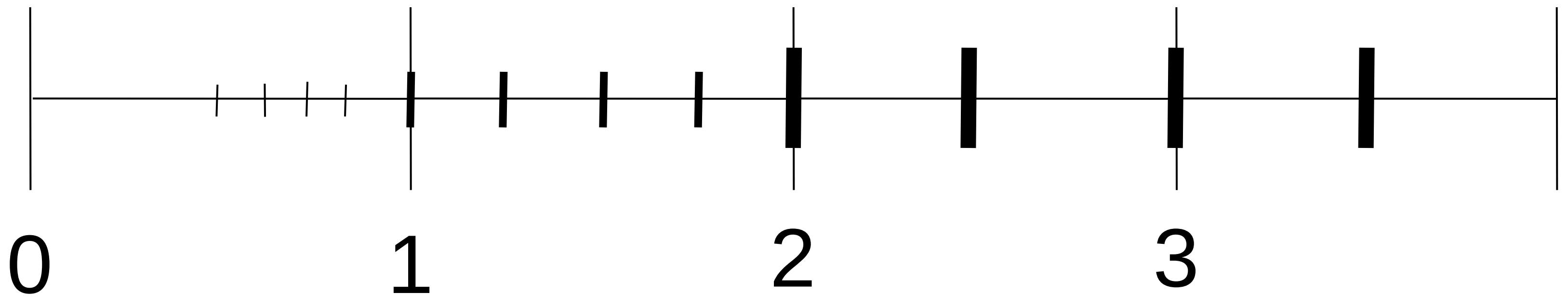
- **Approximate arithmetic**
  - **Finite Range**
  - **Limited Precision**

- **Topics**
  - **IEEE format for single and double precision floating point numbers**
  - **Floating point addition and multiplication**
  - **Support for floating point computation in MIPS**

Distribution of Floating Point Numbers

- **3 bit mantissa**
- **exponent {-1,0,1}**

e = -1	e = 0	e = 1
1.00 X 2^(-1) = 1/2	1.00 X 2^0 = 1	1.00 X 2^1 = 2
1.01 X 2^(-1) = 5/8	1.01 X 2^0 = 5/4	1.01 X 2^1 = 5/2
1.10 X 2^(-1) = 3/4	1.10 X 2^0 = 3/2	1.10 X 2^1= 3
1.11 X 2^(-1) = 7/8	1.11 X 2^0 = 7/4	1.11 X 2^1 = 7/2



Floating Point

- An IEEE floating point representation consists of
  - A Sign Bit (no surprise)
  - An Exponent ("times 2 to the what?")
  - Mantissa ("Significand"), which is assumed to be 1.xxxxx (thus, one bit of the mantissa is implied as 1)
  - This is called a normalized representation
- So a mantissa = 0 really is interpreted to be 1.0, and a mantissa of all 1111 is interpreted to be 1.1111
- Special cases are used to represent denormalized mantissas (true mantissa = 0), NaN, etc., as will be discussed.

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

[illegible]

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

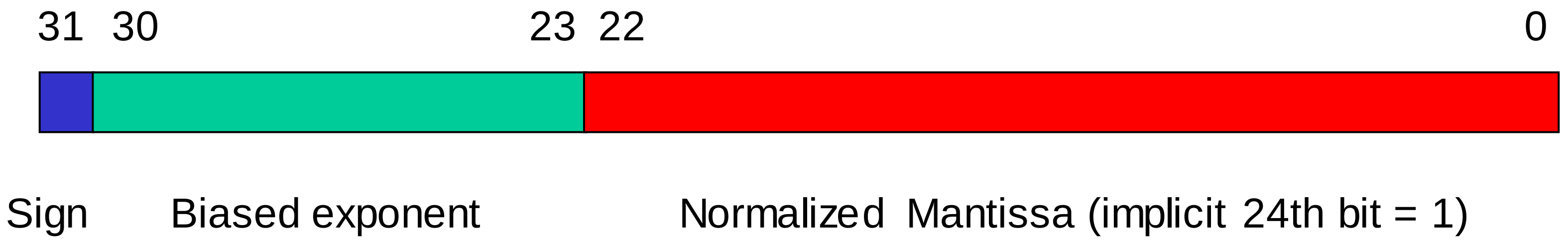
- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - **Exponent: 00000001**
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - **Fraction: 000…00 $\Rightarrow$ significand = 1.0**
  - **$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$**
- **Largest value**
  - **exponent: 11111110**
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - **Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0**
  - **$\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$**

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Representation of Floating Point Numbers

- **IEEE 754 single precision**



$$(-1)^s \times F \times 2^{E-127}$$

Exponent	Mantissa	Object Representec
0	0	0
0	non-zero	denormalized
1-254	anything	FP number
255	0	pm infinity
255	non-zero	NaN

Why biased exponent?

- For faster comparisons (for sorting, etc.), allow integer comparisons of floating point numbers:

- Unbiased exponent:

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

- Biased exponent:

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

Basic Technique

- Represent the decimal in the form +/- $1.xxx_b \times 2^y$
- And "fill in the fields"
  - Remember biased exponent and implicit "1." mantissa!
- Examples:
  - 0.0: 0 00000000 00000000000000000000000
  - 1.0 (1.0 x 2^0): 0 01111111 00000000000000000000000
  - 0.5 (0.1 binary = 1.0 x 2^-1): 0 01111110 00000000000000000000000
  - 0.75 (0.11 binary = 1.1 x 2^-1): 0 01111110 10000000000000000000000
  - 3.0 (11 binary = 1.1*2^1): 0 10000000 10000000000000000000000
  - -0.375 (-0.011 binary = -1.1*2^-2): 1 01111101 10000000000000000000000
  - 1 10000011 01000000000000000000000 = - 1.01 * 2^4 = -20.0

Basic Technique

- One can compute the mantissa just similar to the way one would convert decimal whole numbers to binary.
- Take the decimal and repeatedly multiply the fractional component by 2. The whole number portion is the next binary bit.
- For whole numbers, append the binary whole number to the mantissa and shift the exponent until the mantissa is in normalized form.

http://www.newton.dep.anl.gov/newton/askasci/1995/math/MATH065.HTM

Floating-Point Example

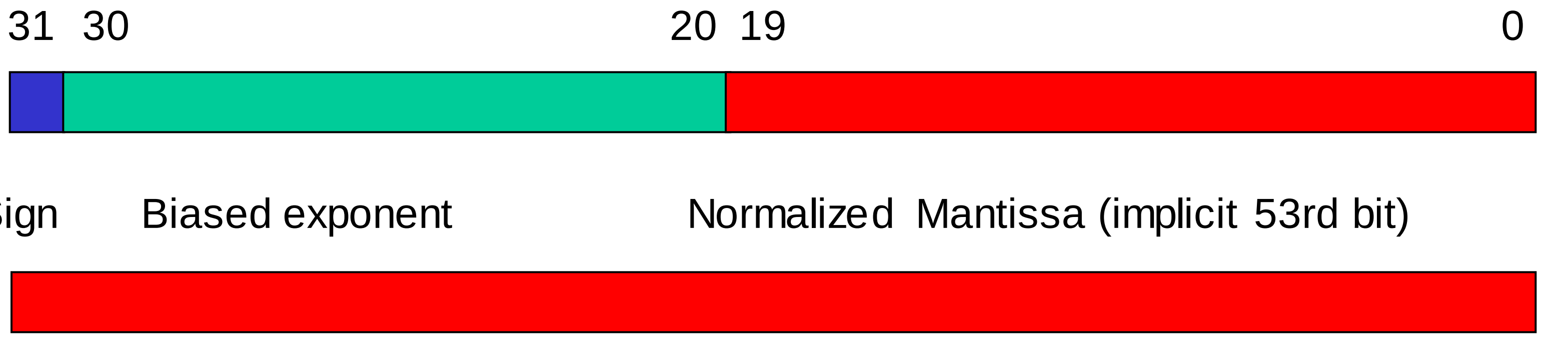
- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float
  11000000101000…00
  - S = 1
  - Fraction = $01000…00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

Representation of Floating Point Numbers

- **IEEE 754 double precision**



$$(-1)^s \times F \times 2^{E-1023}$$

Exponent	Mantissa	Object Representec
0	0	0
0	non-zero	denormalized
1-2046	anything	FP number
2047	0	pm infinity
2047	non-zero	NaN

Floating Point Arithmetic

- fl(x) = nearest floating point number to x
- Relative error (precision = s digits)
  - $|x - fl(x)|/|x| \leq 1/2\beta^{1-s}$ for $\beta = 2$, 2^{-s}
- Arithmetic
  - $x \oplus y = fl(x+y) = (x + y)(1 + \varepsilon)$ for $\varepsilon < u$
  - $x \otimes y = fl(x \times y)(1 + \varepsilon)$ for $\varepsilon < u$

ULP—*U*nit in the *L*ast *P*lace is the smallest possible increment or decrement that can be made using the machine's FP arithmetic.

Floating-Point Precision

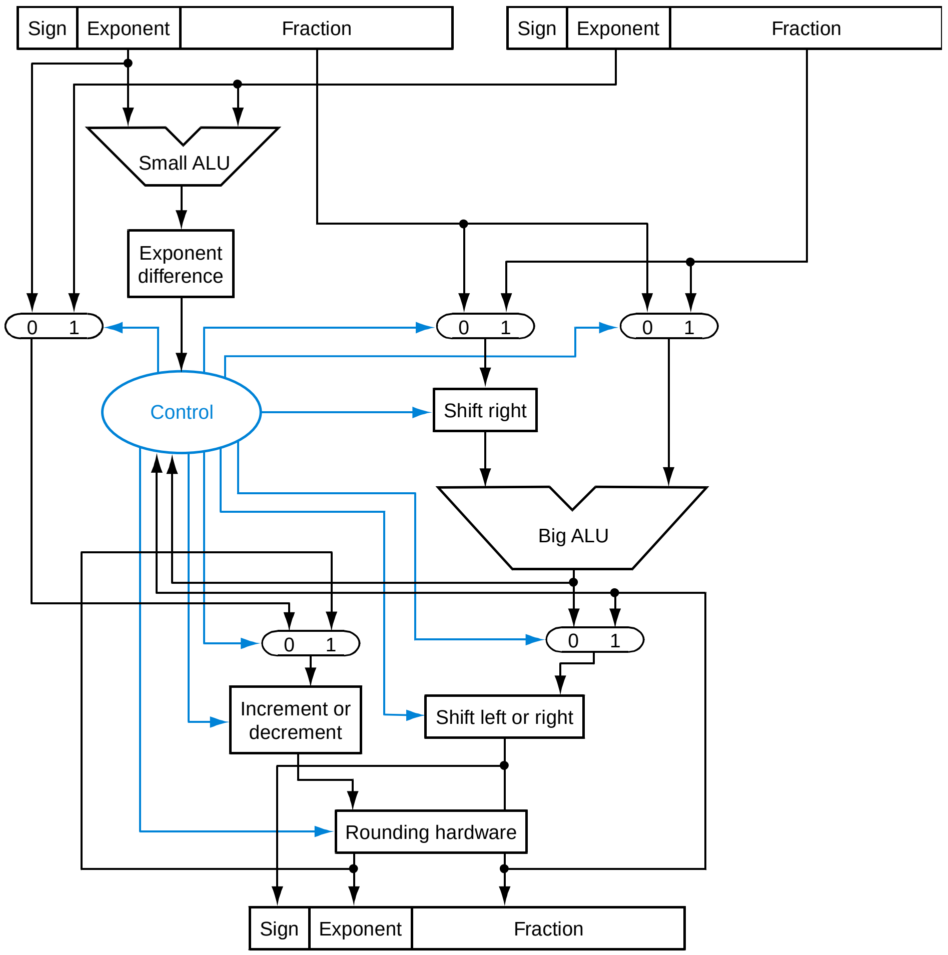
- **Relative precision**
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

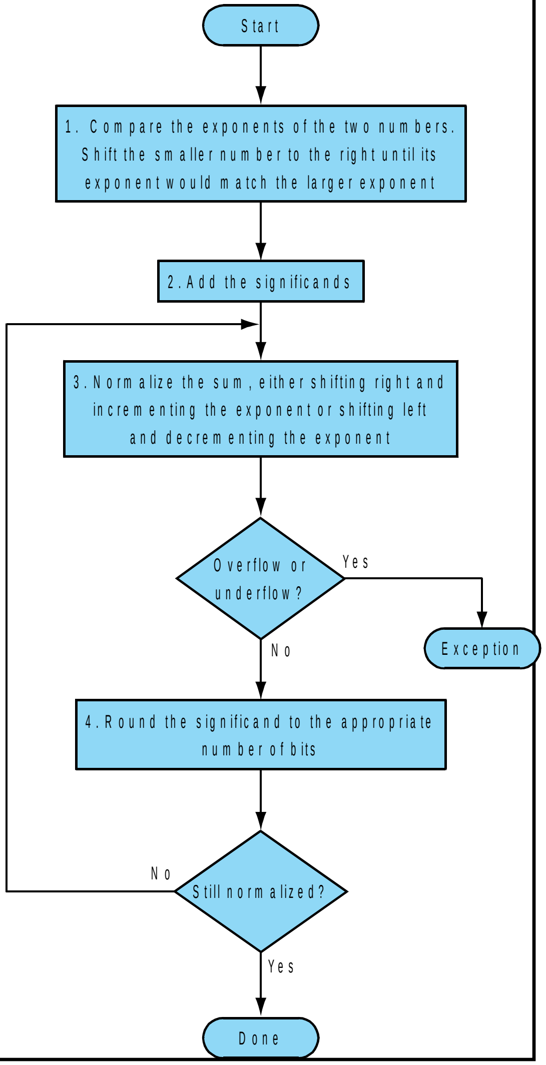
Is FP addition associative?

- **Associativity law for addition: a + (b + c) = (a + b) + c**
- **Let a = – 2.7 x 10^{23}, b = 2.7 x 10^{23}, and c = 1.0**
- **a + (b + c) = – 2.7 x 10^{23} + (2.7 x 10^{23} + 1.0) = – 2.7 x 10^{23} + 2.7 x 10^{23} = 0.0**
- **(a + b) + c = (– 2.7 x 10^{23} + 2.7 x 10^{23}) + 1.0 = 0.0 + 1.0 = 1.0**
- **Beware – Floating Point addition not associative!**
- **The result is approximate…**
- **Why the smaller number disappeared?**

Floating point addition

•





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 the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

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
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

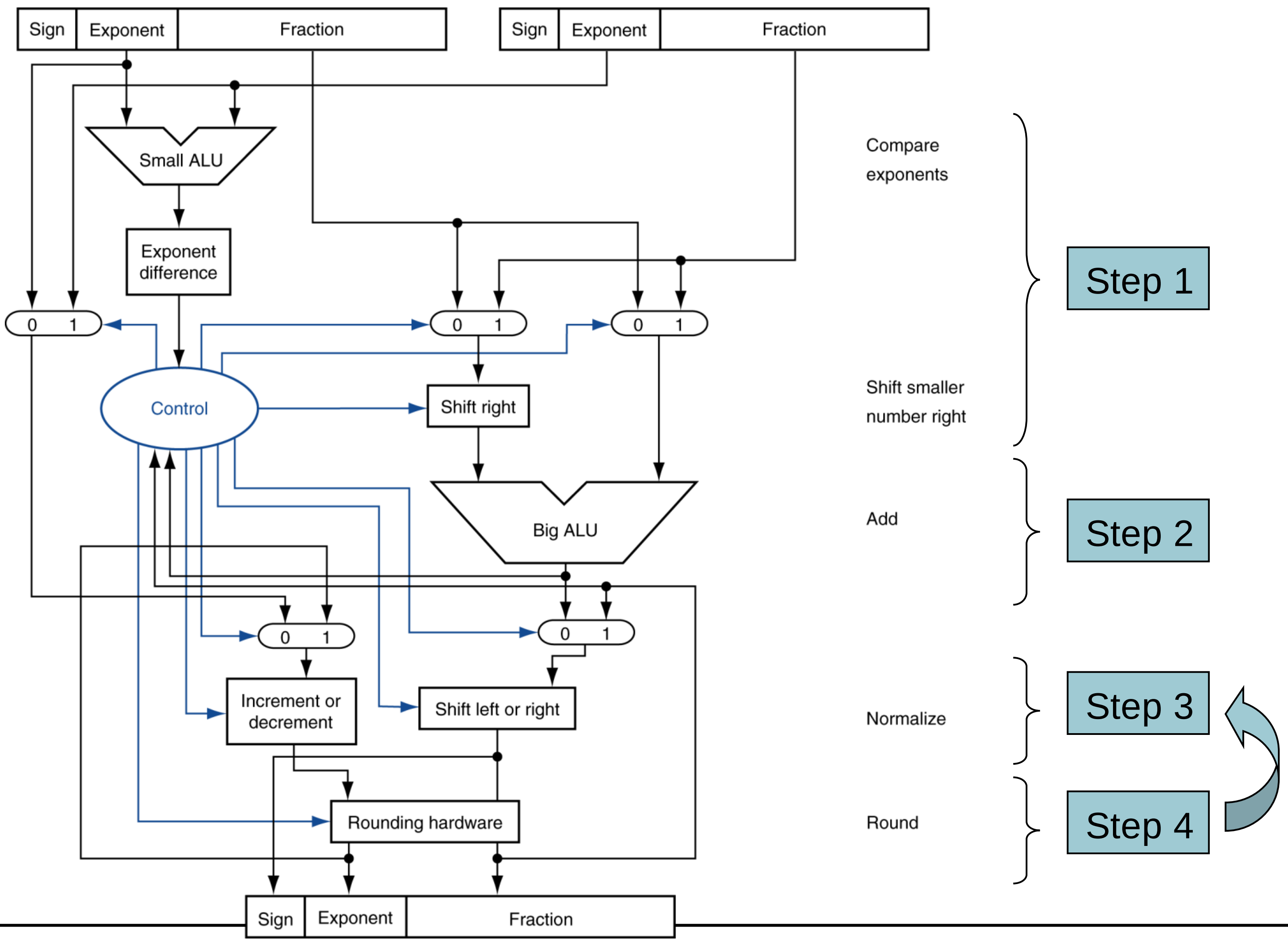
Floating-Point Addition

- 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

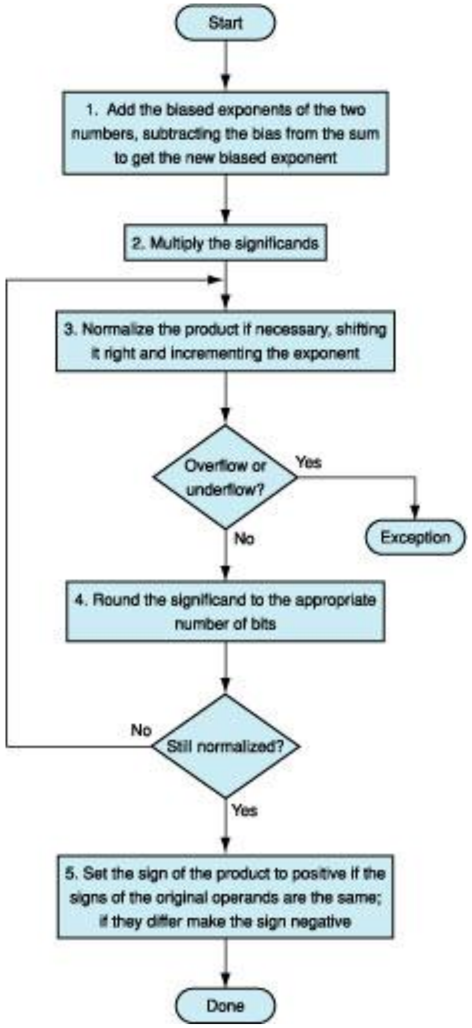
FP Adder Hardware

- Much more complex than integer adder
- 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

FP Adder Hardware



Floating Point Multiplication Algorithm


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
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- **FP hardware is coprocessor 1**
  - **Adjunct processor that extends the ISA**
- **Separate FP registers**
  - **32 single-precision: $f0, $f1, … $f31**
  - **Paired for double-precision: $f0/$f1, $f2/$f3, …**
    - **Release 2 of MIPs ISA supports 32 × 64-bit FP reg's**
- **FP instructions operate only on FP registers**
  - **Programs generally don't do integer ops on FP data, or vice versa**
  - **More registers with minimal code-size impact**
- **FP load and store instructions**
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - **e.g.,** `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- **Single-precision arithmetic**
  - `add.s`, `sub.s`, `mul.s`, `div.s`
    - e.g., `add.s $f0, $f1, $f6`
- **Double-precision arithmetic**
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- **Single- and double-precision comparison**
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `eq`, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- **Branch on FP condition code true or false**
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- **C code:**

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` **in $f12, result in $f0, literals in global memory space**

- **Compiled MIPS code:**

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Rounding

- **Guard and round digits and sticky bit**
  - **When computing result, assume there are several extra digits available for shifting and computation. This improves accuracy of computation.**
  - **Guard digit: first extra digit/bit to the right of mantissa -- used for rounding addition results**
  - **Round digit: second extra digit/bit to the right of mantissa -- used for rounding multiplication results**
  - **Sticky bit: third extra digit/bit to the right of mantissa – used for resolving ties such as 0.50...00 vs. 0.50...01**

Rounding examples

- An example without guard and round digits
  - Add 9.76 x 10^{25} and 2.59 x 10^{24} assuming 3 digit mantissa
    - Shift mantissa of the smaller number to the right: 0.25 x 10^{25}
    - Add mantissas: 10.01x 10^{25}
    - Check and normalize mantissa if necessary: 1.00x 10^{26}

- An example with guard and round digits
  - Add 9.76 x 10^{25} and 2.59 x 10^{24} assuming 3 digit mantissa
    - Internal registers have extra two digits: 9.7600 x 10^{25} and 2.5900 x 10^{24}
    - Shift mantissa of the smaller number to the right: 0.2590 x 10^{25}
    - Add mantissas: 10.0190 x 10^{25}
    - Check and normalize mantissa if necessary: 1.0019 x 10^{26}
    - Round the result: 1.00 x 10^{26}

Rounding examples

- **An example without guard and round digits**
  - **Add 9.78 x 10^{25} and 8.79 x 10^{24} assuming 3 digit mantissa**
    - **Shift mantissa of the smaller number to the right: 0.87 x 10^{25}**
    - **Add mantissas: 10.65 x 10^{25}**
    - **Normalize mantissa if necessary: 1.06 x 10^{26}**

- **An example with guard and round digits**
  - **Add 9.78 x 10^{25} and 8.79 x 10^{24} assuming 3 digit mantissa**
    - **Internal registers have extra two digits: 9.7800 x 10^{25} and 8.7900 x 10^{24}**
    - **Shift mantissa of the smaller number to the right: 0.8790 x 10^{25}**
    - **Add mantissas (note extra digit on the left): 10.6590 x 10^{25}**
    - **Check and normalize mantissa if necessary: 1.0659 x 10^{26}**
    - **Round the result: 1.07 x 10^{26}**

IEEE Rounding Modes

1. Round toward – Infinity: always round toward the smaller number
2. Round toward + Infinity: always round toward the larger number
3. Round to Zero: always round toward the smallest absolute (truncate)
4. Round toward Nearest Even: always round so that least significant bit (lsb) is zero

	1.40	1.60	1.50	2.50	–1.50
Zero	1.00	2.00	1.00	2.00	–1.00
– ∞	1.00	2.00	1.00	2.00	–2.00
+ ∞	1.00	2.00	2.00	3.00	–1.00
Nearest Even (default)					
	1.00	2.00	2.00	2.00	–2.00

- When rounding a binary fraction, the least significant digit of rounded result will be either 1 or 0. Nearest even mode always rounds the number so that the lsb is 0. Hence, the name. (If we omit the binary point, the rounded number would be even.)
- It can be shown that if we assume *uniform distribution* of digits, rounding to nearest mode tends to have mean error = 0.

FP Instructions in MIPS

- Floating point operations are slower than integer operations
- Data is rarely converted from integers to float within the same procedure
- 1980’s solution – place FP processing unit in a separate chip
- Today’s solution – imbed FP processing unit in processor chip
- Co-processor 1 features:
  - Contains 32 single precision floating point registers: $f0, $f1, … $f31
  - These registers can also act as 16 double precision registers: $f0/$f1, $f2/$f3, … , $f30/$f31 (only the first one is specified in the instructions)
  - Uses special floating point instructions, which are similar (in format) to integer instructions but have .s or .d attached to signify that they work on fp numbers
  - Several special instructions to move between “regular” registers and the co-processor registers

FP Instructions in MIPS

- **lwc1 / swc1 – load/store word coprocessor 1**

- **Move instructions (between processors)**

mfc1 rt, rd Move floating point register rd to CPU register rt

mtc1 rd, rt Move CPU register rt to floating point register rd

mfc1.d rdest, frsrc1
Move frsrc1 & frsrc1 + 1 to regs rdest & rdest + 1

- **Single and double precision arithmetic instructions**

Single add.s, sub.s, mul.s, div.s, c.lt.s

Double add.d, sub.d, mul.d, div.d, c.lt.d

- **Examples: add.s $f0, $f1, $f2 sub.d $f0, $f2, $f4**