# CSC 411

## Computer Organization (Spring 2022)
## Lecture 14: FP Addition and Multiplication

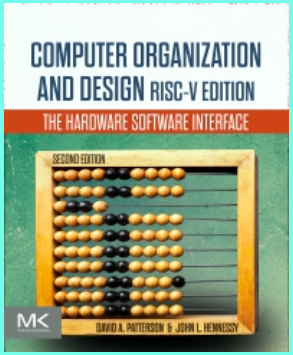
Prof. Marco Alvarez, University of Rhode Island

# Disclaimer

Some of the following slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface




# IEEE 754 encoding of FP numbers

| 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) |

# FP 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
  - $1.002 \times 10^{2}$

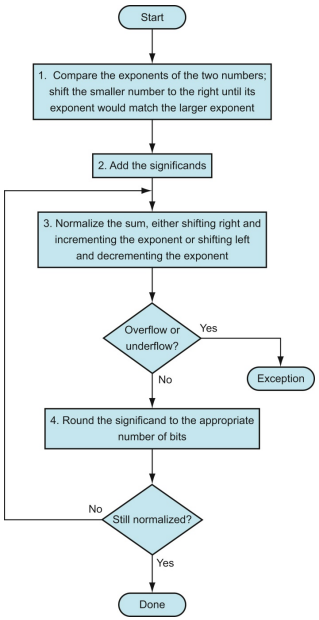
# FP addition

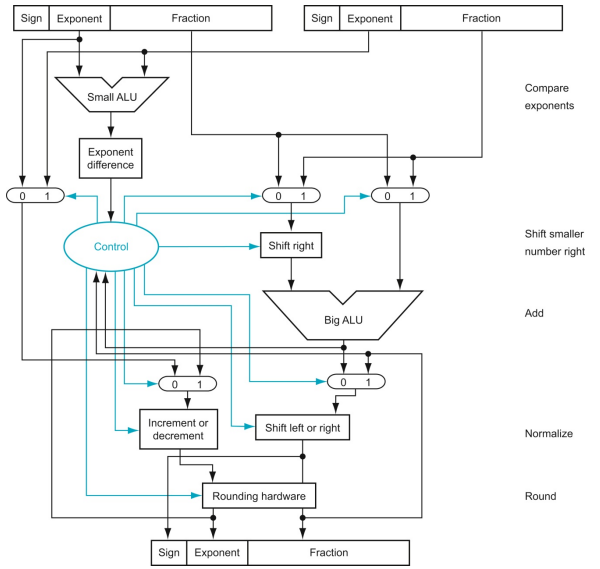
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 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 addition




Block diagram of an ALU dedicated to FP addition

# FP mult

- 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
  - $1.0212 \times 10^{6}$
- 4. Round and renormalize if necessary
  - $1.021 \times 10^{6}$
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

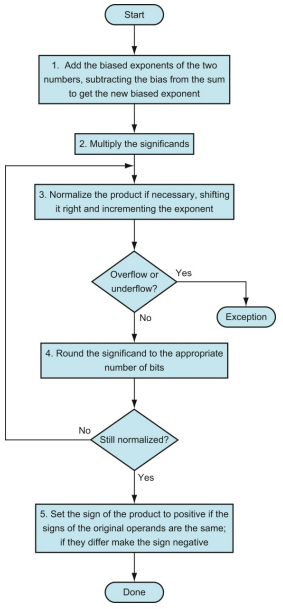
# FP mult

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ ($0.5 \times -0.4375$)
- 1. Add exponents
  - unbiased: –1 + –2 = –3
  - biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 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 × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

# 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 multiplication



## RISC-V floating-point operands

| | | |
|---|---|---|
| 32 floating-point registers | `f0-f31` | An *f*-register can hold either a single-precision floating-point number or a double-precision floating-point number. |
| $2^{30}$ memory words | Memory[0], Memory[4], ..., Memory[4,294,967,292] | Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers. |

## RISC-V floating-point assembly language

| | | | | |
|---|---|---|---|---|
| Arithmetic | FP add single | `fadd.s f0, f1, f2` | `f0 = f1 + f2` | FP add (single precision) |
| | FP subtract single | `fsub.s f0, f1, f2` | `f0 = f1 - f2` | FP subtract (single precision) |
| | FP multiply single | `fmul.s f0, f1, f2` | `f0 = f1 * f2` | FP multiply (single precision) |
| | FP divide single | `fdiv.s f0, f1, f2` | `f0 = f1 / f2` | FP divide (single precision) |
| | FP square root single | `fsqrt.s f0, f1` | `f0 = √f1` | FP square root (single precision) |
| | FP add double | `fadd.d f0, f1, f2` | `f0 = f1 + f2` | FP add (double precision) |
| | FP subtract double | `fsub.d f0, f1, f2` | `f0 = f1 - f2` | FP subtract (double precision) |
| | FP multiply double | `fmul.d f0, f1, f2` | `f0 = f1 * f2` | FP multiply (double precision) |
| | FP divide double | `fdiv.d f0, f1, f2` | `f0 = f1 / f2` | FP divide (double precision) |
| | FP square root double | `fsqrt.d f0, f1` | `f0 = √f1` | FP square root (double precision) |
| Comparison | FP equality single | `feq.s x5, f0, f1` | `x5 = 1 if f0 == f1, else 0` | FP comparison (single precision) |
| | FP less than single | `flt.s x5, f0, f1` | `x5 = 1 if f0 < f1, else 0` | FP comparison (single precision) |
| | FP less than or equals single | `fle.s x5, f0, f1` | `x5 = 1 if f0 <= f1, else 0` | FP comparison (single precision) |
| | FP equality double | `feq.d x5, f0, f1` | `x5 = 1 if f0 == f1, else 0` | FP comparison (double precision) |
| | FP less than double | `flt.d x5, f0, f1` | `x5 = 1 if f0 < f1, else 0` | FP comparison (double precision) |
| | FP less than or equals double | `fle.d x5, f0, f1` | `x5 = 1 if f0 <= f1, else 0` | FP comparison (double precision) |
| Data transfer | FP load word | `flw f0, 4(x5)` | `f0 = Memory[x5 + 4]` | Load single-precision from memory |
| | FP load doubleword | `fld f0, 8(x5)` | `f0 = Memory[x5 + 8]` | Load double-precision from memory |
| | FP store word | `fsw f0, 4(x5)` | `Memory[x5 + 4] = f0` | Store single-precision from memory |
| | FP store doubleword | `fsd f0, 8(x5)` | `Memory[x5 + 8] = f0` | Store double-precision from memory |