



NATIONAL RESEARCH  
UNIVERSITY



# Computer Architecture and Operating Systems

## Lecture 7: Floating-Point Format

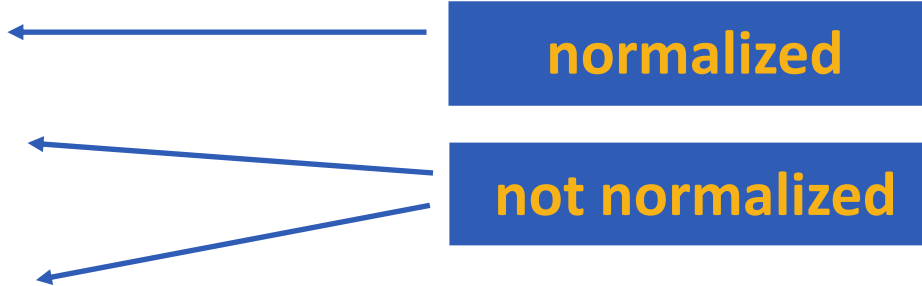
**Andrei Tatarnikov**

[atatarnikov@hse.ru](mailto:atatarnikov@hse.ru)

[@andrewt0301](https://www.instagram.com/andrewt0301)

# Floating-Point Format

- Representation for non-integral numbers
  - Including **very small** and **very large** numbers
- Like scientific notation
  - $-2.34 \times 10^{56}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^9$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types **float** and **double** in C



# 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



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

- **S**: sign bit (0  $\Rightarrow$  non-negative, 1  $\Rightarrow$  negative)
- Normalized **significand**:  $1.0 \leq |\text{significand}| < 2.0$ 
  - 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: 000000000001  $\Rightarrow$  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: 111111111110  $\Rightarrow$  actual exponent =  $2046 - 1023 = +1023$
  - Fraction: 111...11  $\Rightarrow$  significand  $\approx 2.0$
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

# 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

# Floating-Point Example

- Represent  $-0.75$ 
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - $S = 1$
  - Fraction =  $1000\dots00_2$
  - Exponent =  $-1 + \text{Bias}$ 
    - Single:  $-1 + 127 = 126 = 01111110_2$
    - Double:  $-1 + 1023 = 1022 = 011111111110_2$
- Single:  $10111111101000\dots00$
- Double:  $101111111111101000\dots00$



# Floating-Point Example

- What number is represented by the single-precision float  $11000000101000\dots00$ 
  - $S = 1$
  - Fraction =  $01000\dots00_2$
  - Exponent =  $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$ 
  - $= (-1) \times 1.25 \times 2^2$
  - $= -5.0$

# Denormal Numbers

- Exponent = 000...0  $\Rightarrow$  hidden bit is 0

$$x = (-1)^s \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^s \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations  
of 0.0!



# Infinites and NaNs

- Exponent = 111...1, Fraction = 000...0
  - $\pm$ Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction  $\neq$  000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g.,  $0.0 / 0.0$
  - Can be used in subsequent calculations

# Special Values

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

# 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
  - $1.002 \times 10^2$

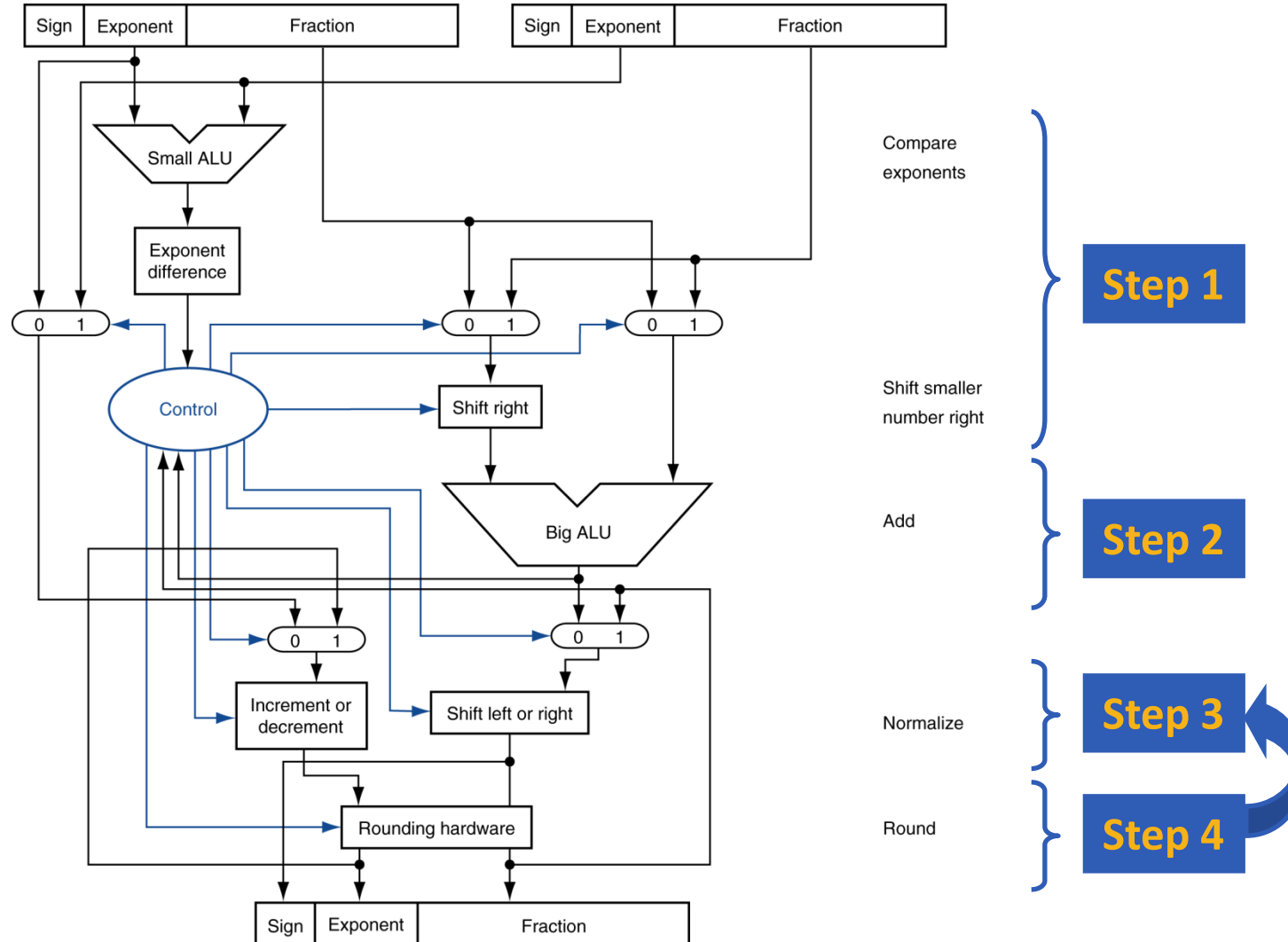
# Floating-Point Addition

- 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

- 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
  - $+1.021 \times 10^6$

# Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$  ( $0.5 \times -0.4375$ )
- 1. Add exponents
  - Unbiased:  $-1 + -2 = -3$
  - Biased:  $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 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 uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP  $\leftrightarrow$  integer conversion
- Operations usually takes several cycles
  - Can be pipelined

# FP Instructions in RISC-V

- Separate FP registers: f0, ..., f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- 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
  - flw, fld
  - fsw, fsd

# FP Instructions in RISC-V

- Single-precision arithmetic
  - `fadd.s`, `fsub.s`, `fmul.s`, `fdiv.s`, `fsqrt.s`
    - e.g., `fadds.s f2, f4, f6`
- Double-precision arithmetic
  - `fadd.d`, `fsub.d`, `fmul.d`, `fdiv.d`, `fsqrt.d`
    - e.g., `fadd.d f2, f4, f6`
- Single- and double-precision comparison
  - `feq.s`, `flt.s`, `fle.s`
  - `feq.d`, `flt.d`, `fle.d`
  - Result is 0 or 1 in integer destination register
    - Use `beq`, `bne` to branch on comparison result
- Branch on FP condition code true or false
  - `b.cond`

# FP Example: °F to °C

## ■ C code:

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

- fahr in f10, result in f10, literals in global memory space

## ■ Compiled RISC-V code:

f2c:

```
f1w    f0, const5(x3)    // f0 = 5.0f  
f1w    f1, const9(x3)    // f1 = 9.0f  
fdiv.s f0, f0, f1        // f0 = 5.0f / 9.0f  
f1w    f1, const32(x3)   // f1 = 32.0f  
fsub.s f10, f10, f1      // f10 = fahr - 32.0  
fmul.s f10, f0, f10      // f10 = (5.0f/9.0f) * (fahr-32.0f)  
jalr   x0, 0(x1)        // return
```

# FP Example: Array Multiplication

- $C = C + A \times B$

- All  $32 \times 32$  matrices, 64-bit double-precision elements

- C code:

```
void mm (double c[][], double a[][], double b[][]) {  
    size_t i, j, k;  
    for (i = 0; i < 32; i = i + 1)  
        for (j = 0; j < 32; j = j + 1)  
            for (k = 0; k < 32; k = k + 1)  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
}
```

- Addresses of c, a, b in x10, x11, x12, and  
i, j, k in x5, x6, x7

# FP Example: Array Multiplication

## ■ RISC-V code:

```
mm: . . .
    li    x28, 32    // x28 = 32 (row size/loop end)
    li    x5, 0      // i = 0; initialize 1st for loop
L1:   li    x6, 0      // j = 0; initialize 2nd for loop
L2:   li    x7, 0      // k = 0; initialize 3rd for loop
    slli  x30, x5, 5   // x30 = i * 2**5 (size of row of c)
    add   x30, x30, x6 // x30 = i * size(row) + j
    slli  x30, x30, 3  // x30 = byte offset of [i][j]
    add   x30, x10, x30 // x30 = byte address of c[i][j]
    fld   f0, 0(x30)  // f0 = c[i][j]
L3:   slli  x29, x7, 5 // x29 = k * 2**5 (size of row of b)
    add   x29, x29, x6 // x29 = k * size(row) + j
    slli  x29, x29, 3  // x29 = byte offset of [k][j]
    add   x29, x12, x29 // x29 = byte address of b[k][j]
    fld   f1, 0(x29)  // f1 = b[k][j]
```



# FP Example: Array Multiplication

...

```
slli    x29,x5,5      // x29 = i * 2**5 (size of row of a)
add     x29,x29,x7    // x29 = i * size(row) + k
slli    x29,x29,3     // x29 = byte offset of [i][k]
add     x29,x11,x29   // x29 = byte address of a[i][k]
fld     f2,0(x29)    // f2 = a[i][k]
fmul.d  f1, f2, f1    // f1 = a[i][k] * b[k][j]
fadd.d  f0, f0, f1    // f0 = c[i][j] + a[i][k] * b[k][j]
addi    x7,x7,1      // k = k + 1
bltu   x7,x28,L3     // if (k < 32) go to L3
fsd     f0,0(x30)    // c[i][j] = f0
addi    x6,x6,1      // j = j + 1
bltu   x6,x28,L2     // if (j < 32) go to L2
addi    x5,x5,1      // i = i + 1
bltu   x5,x28,L1     // if (i < 32) go to L1
```

# Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

# Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		$(x+y)+z$	$x+(y+z)$
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

- Need to validate parallel programs under varying degrees of parallelism

# Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” 😞
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

# Concluding Remarks

- 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

# Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow

# Any Questions?

```
                .text
__start:      addi t1, zero, 0x18
              addi t2, zero, 0x21
cycle:       beq t1, t2, done
              slt t0, t1, t2
              bne t0, zero, if_less
              nop
              sub t1, t1, t2
              j cycle
              nop
if_less:     sub t2, t2, t1
              j cycle
done:       add t3, t1, zero
```