Tikrit University
Electrical Engineering Department

EE-317
Computer Engineering
2024-2025

Arithmetic: Floating Point Operations

Jalal Nazar Abdulbaqi, Ph.D.

jalal.abdulbaqi@tu.edu.iq

Outline

- Arithmetic on Floating Point
  - Addition
  - Multiplication

Floating-Point Addition Example

• **Example: 4-digit Decimal**	$\mathbf{9.999 \times 10^{1} + 1.610 \times 10^{-1}}$
1. Align decimal points → Shift number with smaller exponent	$9.999 \times 10^{1} + \mathbf{0.016} \times \mathbf{10^{1}}$
2. Add significands	$9.999 \times 10^{1} + 0.016 \times 10^{1}$ $= \mathbf{10.015 \times 10^{1}}$
3. Normalize result & check for over/underflow	$\mathbf{1}.0015 \times \mathbf{10^{2}}$
4. Round and renormalize if necessary	$1.00\mathbf{2} \times 10^{2}$

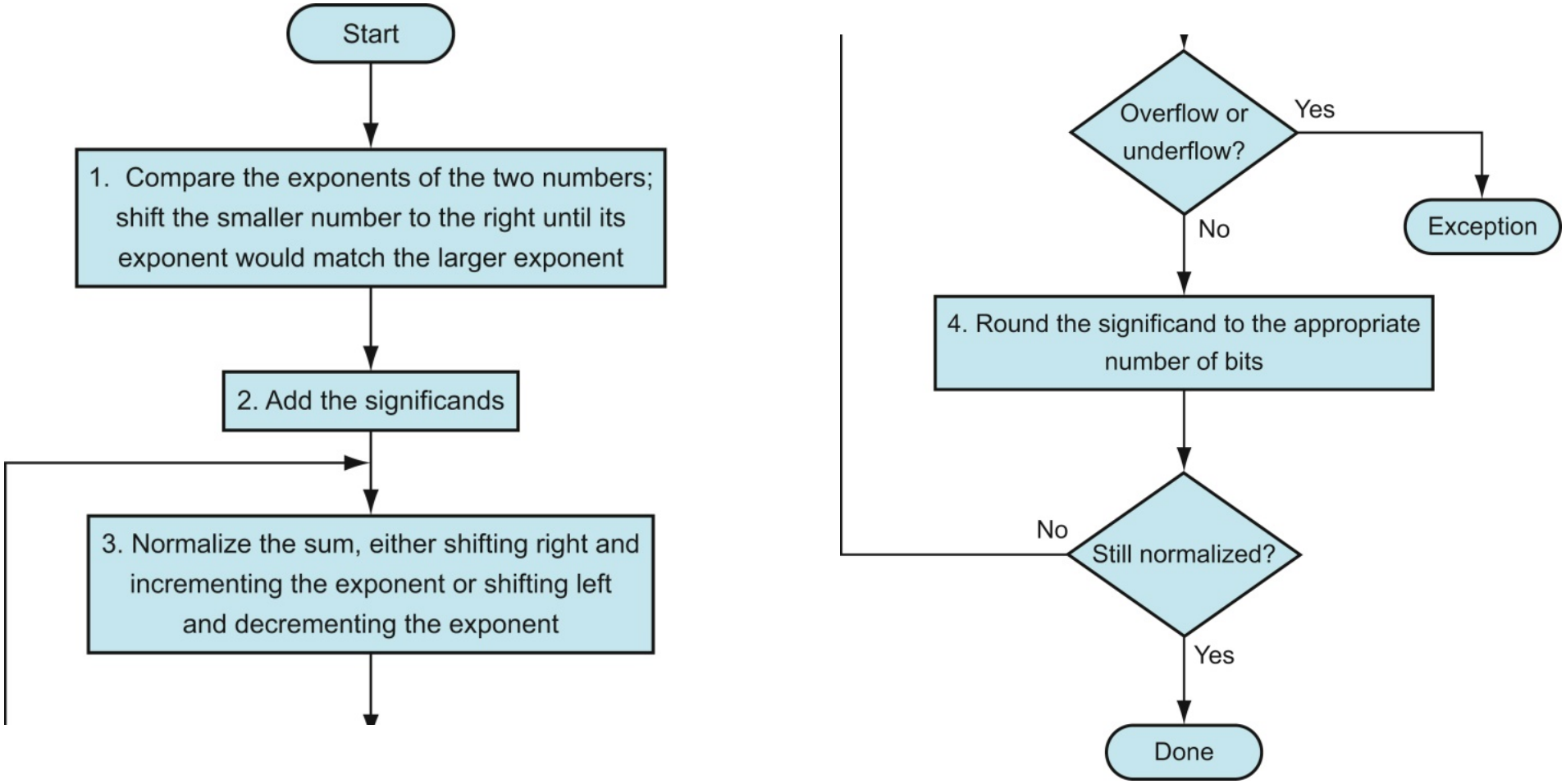
Floating-Point Addition Example

• **Example: 4-digit Binary**	$\mathbf{1.000_{two} \times 2^{-1} + -1.110_{two} \times 2^{-2}}$ $\mathbf{(0.5_{ten} + -0.4375_{ten})}$
1. Align decimal points → Shift number with smaller exponent	$1.000_{two} \times 2^{-1} + \mathbf{-0.111_{two} \times 2^{-1}}$
2. Add significands	$1.000_{two} \times 2^{-1} + -0.111_{two} \times 2^{-1}$ $= \mathbf{0.001_{two} \times 2^{-1}}$
3. Normalize result & check for over/underflow	$\mathbf{1.000_{two} \times 2^{-4}}$
4. Round and renormalize if necessary	$1.000_{two} \times 2^{-4}$ (no change) $= \mathbf{0.0625_{ten}}$

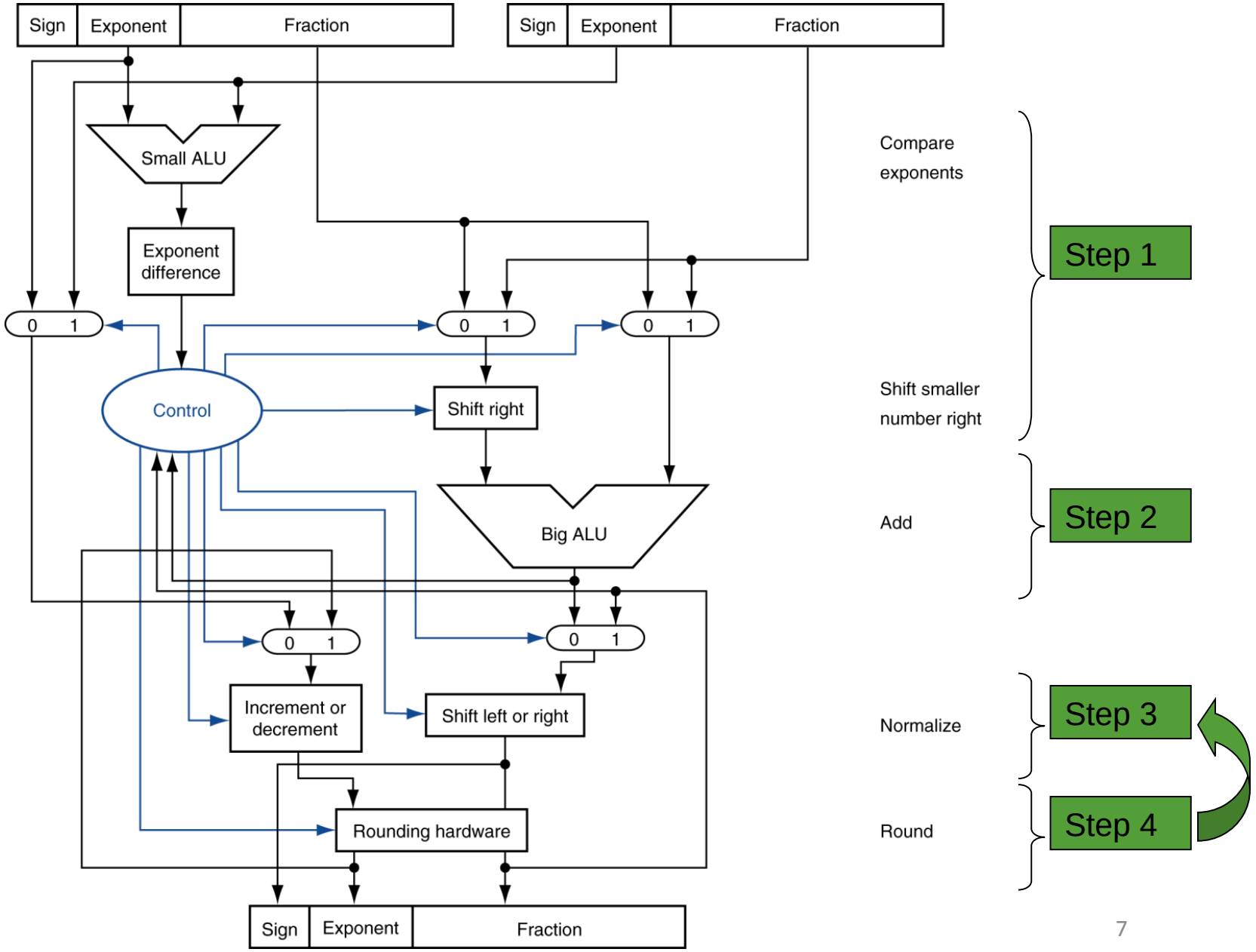
Floating-Point Addition Algorithm

1. the binary point has to be aligned this means that the significand of the ***smaller number*** is shifted to the right until the decimal points are aligned.
2. then the addition of the significand takes place
3. the result needs to be *normalized*, which means the binary point is shifted left and exponent increases.
4. the result needs to be *truncated* to available number of digits and rounded off (add 1 to the last available digit if number to the right is 5 or larger)

Floating-Point Addition Algorithm



Floating-Point Adder Hardware



Floating-Point 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
- Floating-Point adder usually takes several cycles
  - Can be pipelined

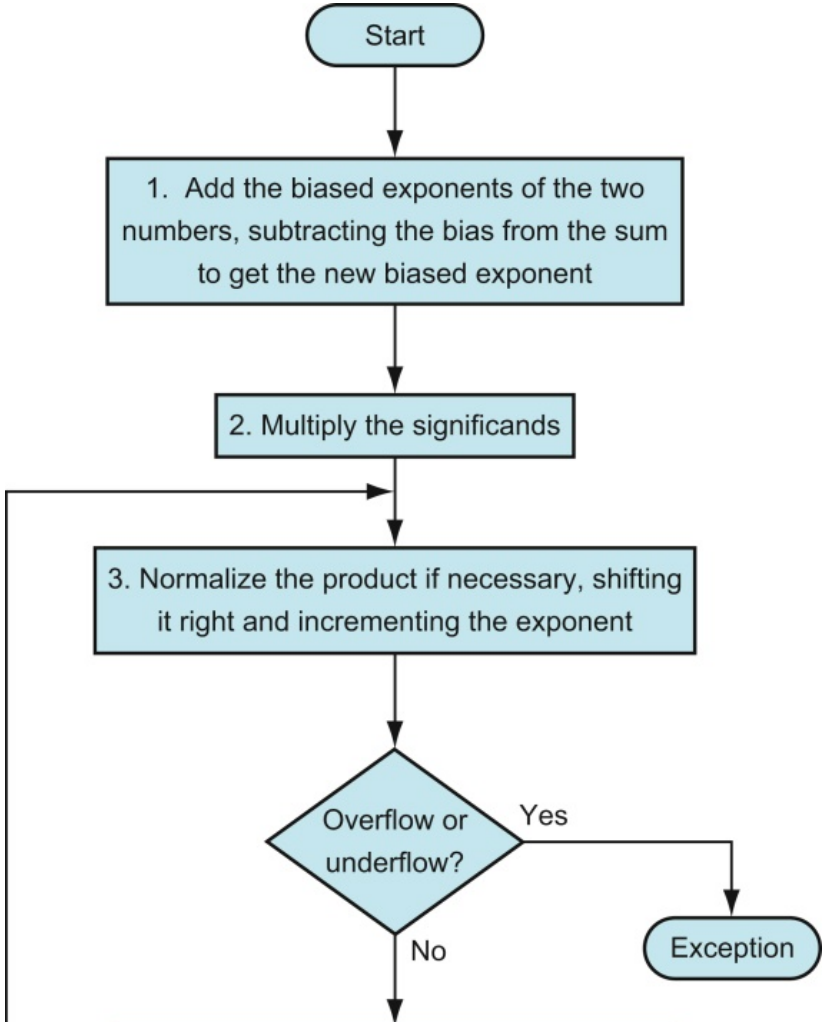
Floating-Point Multiplication Example

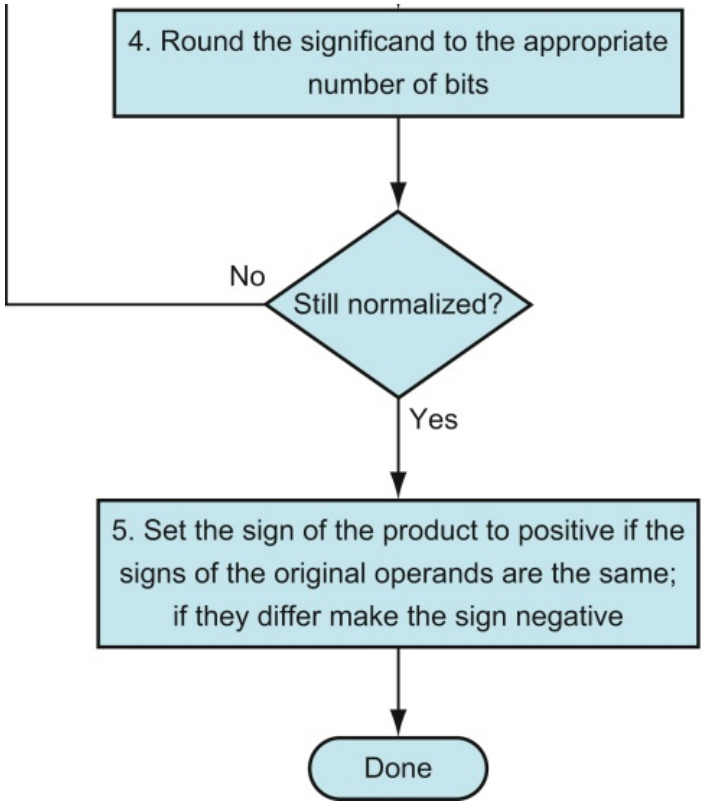
• **Example: 4-digit Decimal**	**$1.110 \times 10^{10} \times 9.200 \times 10^{-5}$**
1. Add exponents → For biased exponents, subtract bias from sum	**10** + **–5** = **5**
2. Multiply significands	$1.110 \times 9.200 = 10.212$ $\Rightarrow$ **10.212×10^5**
3. Normalize result & check for over/underflow	**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 Example

• **Example: 4-digit Binary**	**$1.000_{two} \times 2^{-1} \times -1.110_{two} \times 2^{-2}$** $(0.5_{ten} \times -0.4375_{ten})$
1. Add exponents → For biased exponents, subtract bias from sum	**Unbiased**: –1 + –2 = –3 **Biased**: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
2. Multiply significands	$1.000_{two} \times 1.110_{two} = 1.110_{two}$ $\Rightarrow 1.110_{two} \times 2^{-3}$
3. Normalize result & check for over/underflow	$1.110_{two} \times 2^{-3}$ (no change)
4. Round and renormalize if necessary	$1.110_{two} \times 2^{-3}$ (no change)
5. Determine sign of result from signs of operands	$-1.110_{two} \times 2^{-3}$ $= -0.21875_{ten}$

Floating-Point Multiplication Algorithm





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

Accurate Arithmetic

- During the preceding examples, arithmetic operations results can have larger number of digits than what the registers can hold, therefore,
- IEEE 754 always keeps **two** extra bits on the right called guard and round.

Values 00-49 → round down, while 51-99 → round up

if values 50 → extra bit (sticky bit) is set to 1 if there are nonzero bits to the right of the round bit

Accurate Arithmetic

IEEE 754 has four rounding modes:

1. always round up (toward $+\infty$),
2. always round down (toward $-\infty$),
3. truncate, and
4. **round to nearest even**.

"Round to nearest even" determines what to do if the number is exactly halfway in between. IEEE 754 says that if the least significant bit (LSB) retained in a halfway case would be odd, add one; if it's even, truncate. In other words, choose the nearest even number.

Accurate Arithmetic

Example #1: round the following binary numbers to the nearest two bits fraction:

0.11101 → 1.00 (round up)

0.11011 → 0.11 (round down)

0.11100 → **tie-breaking case**, the number in the halfway case

→ **round to nearest even**

round **up**: 1.00 (**Even**)

round **down**: 0.11 (Odd)

then we round **up** (1.00) because it is **even**.

Accurate Arithmetic

Example #2: round the following binary numbers to the nearest two bits fraction:

0.10101 → 0.11 (round up)

0.10011 → 0.10 (round down)

0.10100 → **tie-breaking case**, the number in the halfway case

→ **round to nearest even**

round **up**: 0.11 (Odd)

round **down**: 0.10 (**Even**)

then we round **down** (0.10) because it is **even**.

Accurate Arithmetic – Example 1

OP	3-digit significant	Guard	Round	Sticky
	$8.76 \times 10^1 + 1.47 \times 10^2$	0	0	0
Align	$0.876 \times 10^2 + 1.47 \times 10^2$	6	0	0
Add	2.3460×10^2	6	0	0
Norm	2.3460×10^2	6	0	0
Round	2.35×10^2	0	0	0

Accurate Arithmetic – Example 2

OP	3-digit significant	Guard	Round	Sticky
	$5.01 \times 10^{-1} + 1.34 \times 10^{2}$	0	0	0
Align	$0.00501 00 \times 10^{2} + 1.34 \times 10^{2}$	5	0	1
Add	1.3450×10^{2}	5	0	1
Norm	1.3450×10^{2}	5	0	1
Round	$1.3\mathbf{5} \times 10^{2}$	0	0	0

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`** Result is 0 or 1 in integer destination register
  - **`feq.d, flt.d, fle.d`** Use **`beq, bne`** to branch on comparison result
- Branch on FP condition code true or false
  - `B.cond`