TC2009B: Digital Design
Fixed-point and floating-point numbers

Isaac Pérez Andrade

ITESM Guadalajara
School of Engineering & Science
Department of Computer Science
August – December 2021

References

The following material has been adopted and adapted from

Patterson, D. A., Hennessy, J. L., *Computer Organization and design: The hardware/software interface – ARM edition*, Morgan Kaufmann, 2017.

S. L. Harris and D. M. Harris, *Digital design and computer architecture - ARM edition*, Morgan Kaufmann, 2016.

Numbers is digital systems

- How is information represented in digital systems?
- How do computers represent information?

Numbers is computer systems

- Fixed-point representation
  - Integers – limited precision
  - Non-integers – limited precision
- Floating-point representation
  - Real-world numbers
  - Extremely large or extremely small numbers

Fixed-point representation

Fixed-point integer representation

- Unsigned.

$$[0, 2^N - 1]$$

- Sign & magnitude.

$$\left[-(2^{(N-1)} - 1), 2^{(N-1)} - 1\right]$$

- One's complement.

$$\left[-(2^{(N-1)} - 1), 2^{(N-1)} - 1\right]$$

- Two's complement.

$$\left[-(2^{(N-1)}), 2^{(N-1)} - 1\right]$$

Signed integer representation

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

Fixed-point integer representation

- Unsigned.
- Sign & magnitude.
- One's complement.
- Two's complement.

- From all these binary integer representations, which one is most used in computer systems?

Two's complement integer range

- Q: What's the range (minimum and maximum values that can be represented) of an N-bit two's complement number?
- A: $\left[-(2^{(N-1)}), 2^{(N-1)} - 1\right]$

- For example, an 8-bit two's complement number may represent values in the range

$$[-2^{8-1}, 2^{8-1} - 1] = [-2^7, 2^7 - 1] = [-128, 127]$$

Bits required for representing decimals

- Q: What's the minimum number of bits N required for representing decimal number D?
- A: $N = \lceil \log_2 D \rceil$
- Example: How many bits are required for representing the decimal number 12345?
- A: $N = \lceil \log_2 12345 \rceil = \lceil 13.59 \rceil = 14$

Bits required for representing decimals

- Q: What's the minimum number of bits N required for representing fractional number F?
- A: $N = \lceil |\log_2 F| \rceil$
- Example: How many bits are required for representing the fractional number 0.12345?
- A: $N = \lceil |\log_2 0.12345| \rceil = \lceil |-3.3219| \rceil = 4$
- $2^{-4} = 0.0625$
- $2^{-3} = 0.125$

Fixed-point numbers

- What about real numbers?

Fixed-point real numbers

- Real numbers may not be represented with integer numbers.

```
integer a,b;
a = 1.5;
b = a + a; // b = ?
```

- Fixed-point representation allows real number representation with limited precision.
  - Qm.n representation
    - m $\rightarrow$ number of bits for representing integer part.
    - n $\rightarrow$ number of bits for representing non-integer part.
    - Range $[-(2^{m-1}), 2^{m-1} - 2^{-n}]$
    - Resolution is 2^{-n}

Fixed-point real numbers

- Qm.n example:
  - What is the range of Q4.4 representation?
  - $[-(2^{m-1}), 2^{m-1} - 2^{-n}]$
  - $[-(2^{4-1}), 2^{4-1} - 2^{-4}] = [-8, +7.9375]$
- Qm.n example:
  - What is the decimal value of the Q4.4 number 1000.0001?
    - We first take the 2's complement of the number
      - 0111.1111, which is 7.9375 in decimal
    - Therefore, Q4.4 1000.0001 represents –7.9375 decimal

Fixed-point in real-world applications

- Fixed-point representation is suitable for embedded applications requiring limited degree of fractional precision.
  - DOOM (1993 videogame) originally used a Q16.16 format for all non-integer operations https://doomwiki.org/wiki/Fixed_point

- What about high-precision applications?

Fixed-point limitations

- Example:
  - Consider Avogadro's number: 6.022×10^{23}
    - How many bits would you need to represent Avogadro's number?

$$\lceil \log_2(6.022 \times 10^{23}) \rceil = 79$$

  - What about a very small number such as Planck's constant: $6.62607004 \times 10^{-34}\ \mathrm{J \cdot s}$
    - How many bits (fractional fixed-point) would you need to represent Planck's constant?

$$\lceil |\log_2(6.62607004 \times 10^{-34})| \rceil = 111$$

  - We would need at least $79 + 111 = 190$ bits for representing both numbers.
    - Not feasible, waste of resources.
    - What if we need even smaller or larger numbers?

Floating-point representation

Floating-point

- Scientific notation
  - $\mathbf{+1.12345} \times 10^{\mathbf{-7}}$ ← normalized
  - $\mathbf{-123.456} \times 10^{9}$ ← not normalized
    - **Sign**
    - **Mantissa/significant**
    - Base with **Exponent**
- Normalized scientific notation
  - Absolute value of integer part m is in the range

$$[1, 10) \rightarrow 1 \leq m \leq 9$$

- Binary numbers may also be represented in scientific notation

0.5_{10}

$$1.0_2 \times 2^{-1} = 0.1_2$$ ← normalized

$$0.1_2 \times 2^{0} = 0.1_2$$ ← not normalized

Floating-point

- As the name suggest, binary point is not fixed.
- Representation for non-integral numbers
  - Including very small and very large numbers
  - $(-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent}-\text{bias})}$
  - For simplicity, we'll show the exponent in decimal.
- Programming languages refer to this representation as `float` and `double` types.

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)
- Simplifies exchange of data, arithmetic and increases accuracy.

IEEE Floating-point format

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	biased exponent	mantissa

Implicit 1

$$x = (-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent})}$$

- sign: (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Significand: $1.\text{mantissa}$
- Normalized significand: $1.0 \le |\text{significand}| < 2.0$
- biased exponent = exponent + bias
  - Ensures exponent is unsigned
  - Single: bias = 127
  - Double: bias = 1023

IEEE Floating-point format

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	biased exponent	mantissa

$$x = (-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent})}$$

- First example: 0.5 to single-precision floating-point

$$+0.5_{10} = +0.1_2 = +1.0_2 \times 2^{-1} = (-1)^0 1.0_2 \times 2^{-1}$$

For simplification, we are using decimal notation for the exponent

- From this, we can gather the following information:
  - sign: 0
  - mantissa: 0 (normalized)
  - exponent: -1
- Everything together:
  - biased exponent$= -1 + 127 = 126_{10}$ or 01111110_2
  - Normalized floating-point:

$$(-1)^0 1.0 \times 2^{(126-127)}$$

$$0\ 01111110\ 00000000000000000000000$$

IEEE Floating-point format

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	biased exponent	mantissa

$$x = (-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent})}$$

- First example: 0.5 to single-precision floating-point

$$+0.5_{10} = (-1)^{0} 1.0 \times 2^{(126-127)}$$

This 1 is not actually required here

0 01111110 000000000000000000000

- What about 0.5 in double-precision?

$$0.5_{10} = (-1)^{0} 1.0 \times 2^{(1022-1023)}$$

0 01111111110 0000000000000000000000000000000000000000000000
000000

Floating-point example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - sign = 1
  - mantissa = $1000\ldots00_2$
  - biased exponent = –1 + bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-point example

- Represent 5.625
  - $5.625_{10} = 101.101_2$
  - $5.625 = (-1)^0 \times 1.01101_2 \times 2^2$
  - $\text{sign} = 0$
  - $\text{mantissa} = 01101000000000000000000_2$
  - $\text{biased exponent} = 2 + \text{bias}$
    - Single: $2 + 127 = 129 = 10000001_2$
    - Double: $2 + 1023 = 1024 = 10000000001_2$
- Single: 01000000101101000000000000000000
- Double: 010000000001011010 0...00

Floating-point example

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	exponent	mantissa

- What number is represented by the single-precision float
  11000000101000000000000000000000
  - sign = 1
  - biased exponent = $10000001_2 = 129$
  - mantissa = $01000...00_2$
- $x = (-1)^1 \times (1.01_2) \times 2^{(129-127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

Remember that this 1 is always implied!

Floating-point exercises

- Convert the following real numbers to floating-point representation.

1. 1234.5625
2. 31.875
3. -219.125

Floating-point exercises

- Convert the following real numbers to floating-point representation.

1. 1234.5625 => 0x449a5200
2. 31.875 => 0x41ff0000
3. -219.125 => 0xc35b2000

Floating-point exercises

- Convert the following floating-point numbers to real numbers

1. 0xc3db2000
2. 0x40808000
3. 0x41200000

Floating-point exercises

- Convert the following floating-point numbers to real numbers

1. 0xc3db2000 => -438.25
2. 0x40808000 => 4.015625
3. 0x41200000 => 10.0

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 ≈ 2.0
- $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double precision range

Exponents 00000000000 and 11111111111 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 ≈ 2.0
- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-point precision

- Relative precision
  - Single: approx. $2^{-23} \rightarrow \approx 7$ decimal digits
  - Double: approx. $2^{-52} \rightarrow \approx 16$ decimal digits

Floating-point special representation

- Denormal numbers
  - In normalised numbers, significand have an implicit leading 1

$$x = (-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent})}$$

  - Denormal numbers have a leading 0 in the significand.
  - Biased exponent is 0.
  - These numbers allow to represent numbers smaller than the smaller normalised number, as well as special representation such as $\pm\infty$ and NaN $(\mathbf{0} \div \mathbf{0})$.

Denormal numbers

$$x = (-1)^{\text{sign}} \times (0.\text{mantissa}) \times 2^{0-\text{bias}}$$

- Smaller than normal numbers.

- Zero
  - sign $=$ 0,1
  - biased exponent $=$ 0
  - mantissa $=$ 0

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

Two representations of 0.0!

Denormalized numbers

Smallest denormalized value

- Single precision:
  $\pm 2^{-23} \times 2^{-126} \approx 1.4 \times 10^{-45}$
- Double precision:
  $\pm 2^{-52} \times 2^{-1022} \approx 4.9 \times 10^{-324}$

Largest denormalized value

- Single precision:
  $\pm(1 - 2^{-23}) \times 2^{-126} \approx \pm 1.17 \times 10^{-38}$
- Double precision:
  $\pm(1 - 2^{-52}) \times 2^{-1022} \approx \pm 2.22 \times 10^{-308}$

Infinities and NaNs

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

Floating-point special formats summary

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)

Floating-point rounding

Floating-point rounding

- Let's try to represent 0.1_{10} in floating-point.
  1. Represent 0.1_{10} in binary
     1. Integer part is 0
     2. Fractional part is:

$$\begin{aligned} 0.1 \times 2 &= 0 + 0.2 \\ 0.2 \times 2 &= 0 + 0.4 \\ 0.4 \times 2 &= 0 + 0.8 \\ 0.8 \times 2 &= 1 + 0.6 \\ 0.6 \times 2 &= 1 + 0.2 \\ 0.2 \times 2 &= 0 + 0.4 \\ 0.4 \times 2 &= 0 + 0.8 \\ 0.8 \times 2 &= 1 + 0.6 \\ 0.6 \times 2 &= 1 + 0.2 \\ &\vdots \end{aligned}$$

$$0.0\overline{0011}$$

This sequences repeats infinite times!

0.1 is not a machine number, which means, it may not be exactly represented in a computing system.

Floating-point rounding

- Our floating-point representation will have to be as close as possible to 0.1_{10}
  $0.0001100110011001100110011001100110011001\ldots_2$
- Remember that mantissa is 23 and 52 bits for single- and double-precision, respectively.
- IEEE employs rounding.
  - If first extra bit is 1, we add 1 to the rest of the mantissa bits – This is rounding up
  - If first extra bit is 0, we drop all extra bits – This is rounding down.
  - Special case if extra bits are 1000....000
    - Round up if last mantissa bit is 1
    - Round down if last mantissa bit is 0

Floating-point rounding

- Rounded normalized value:

$$1.10011001100110011001101_2 \times 2^{-4}$$

We rounded up for this example

$$1.10011001100110011001101_2 \times 2^{123-127} \text{ (single)}$$

- Floating-point representation:

Single: 1 0111011 10011001100110011001101

- Which represents the value of

$$0.100000001490116119384765625$$

- Similarly, double precision represents 0.1 as

0.1000000000000000055511151231257827021181583404541015625

Floating-point rounding

- In your favourite programming language try the following code using float or double data types.

  `0.1 + 0.1 + 0.1 == 0.3`

  Is the result TRUE or FALSE?
- Problems with accuracy
  - Several failures (in some cases with fatal consequences) have been reported due to numerical errors.

    http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html

Concluding remarks

- Bits have no inherent meaning
  - Interpretation depends on the application.
- Computer representations of numbers
  - Finite range and precision, even in double-precision floating-point representation.
  - Need to account for this in programs.

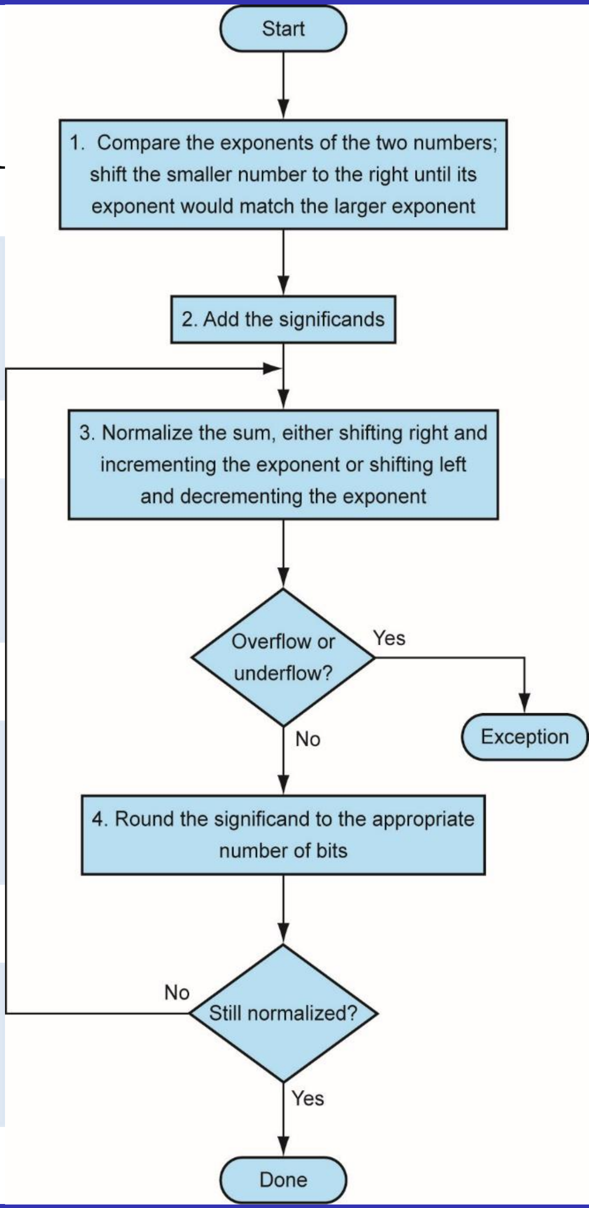
Floating-point addition

Floating-point addition

- Consider a 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} Rounding up to 3 decimal places

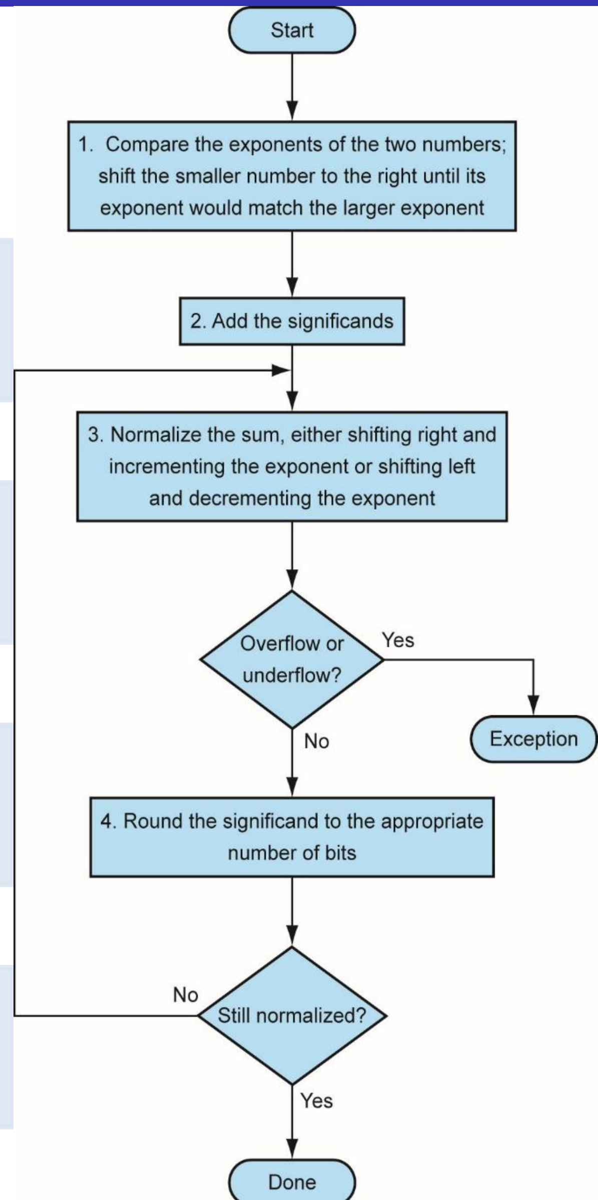


Floating-point addition

- Consider a floating-point example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ or (0.5+-0.4375)

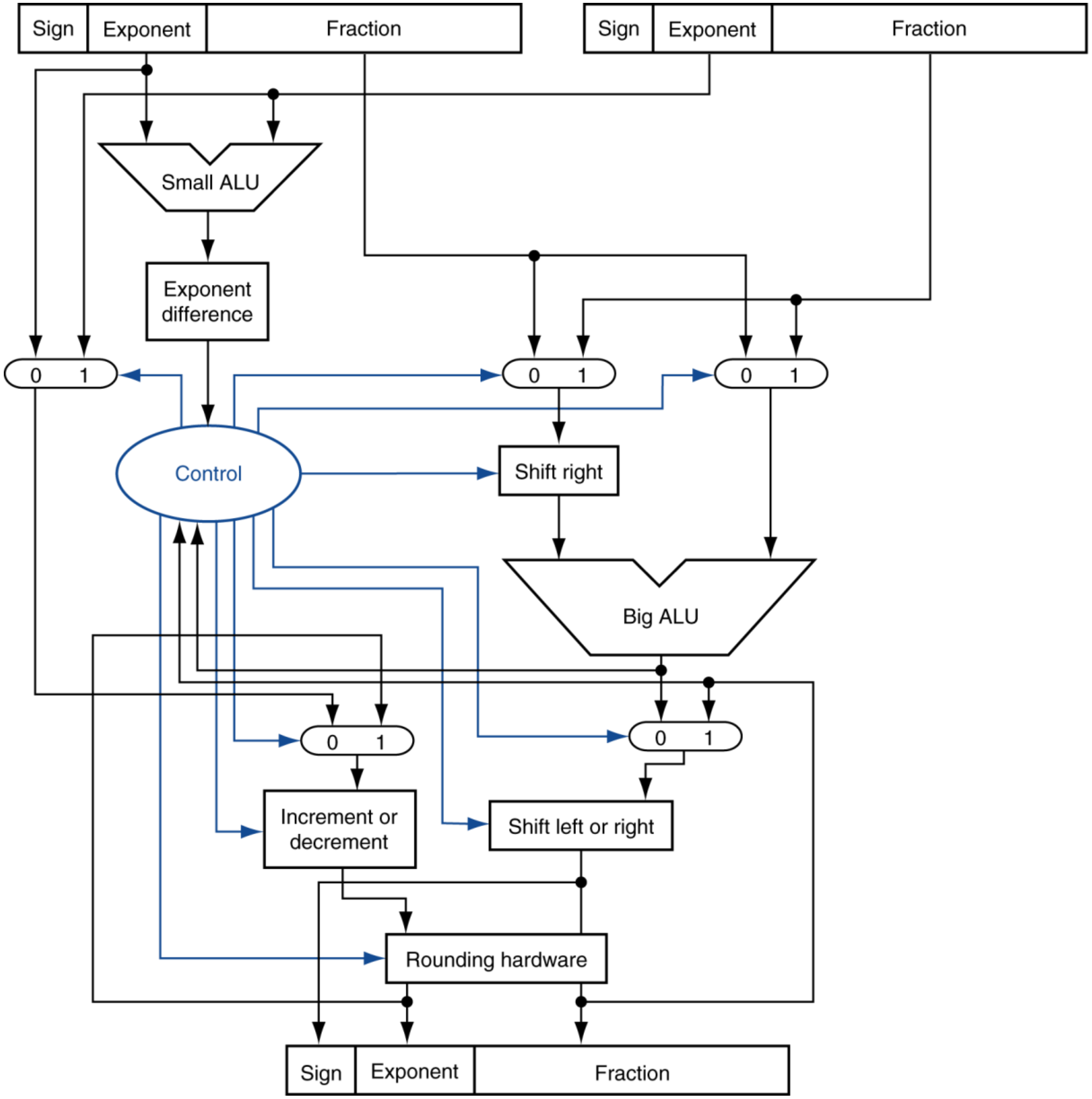
1. Align decimal points	Shift number with smaller exponent $1.000_2 \times 2^{-1} + -0.1110_2 \times 2^{-1}$
2. Add significands	$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalize result & check for over/underflow	$1.0_2 \times 2^{-4}$
4. Round and renormalize if necessary	$1.0_2 \times 2^{-4} = 0.0625$

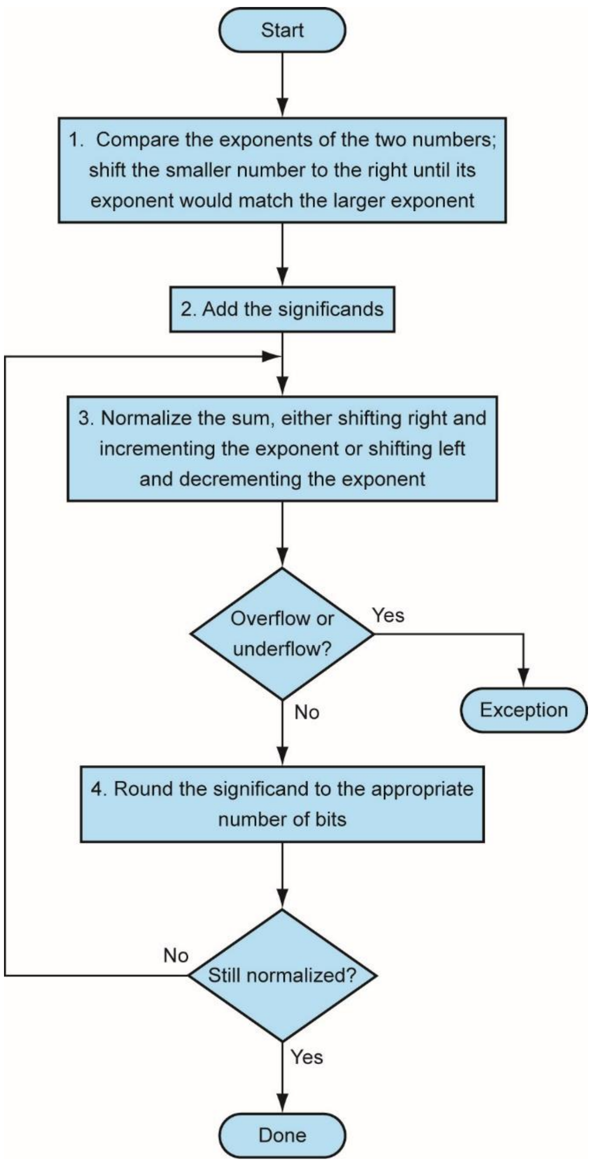


Floating-point adder hardware

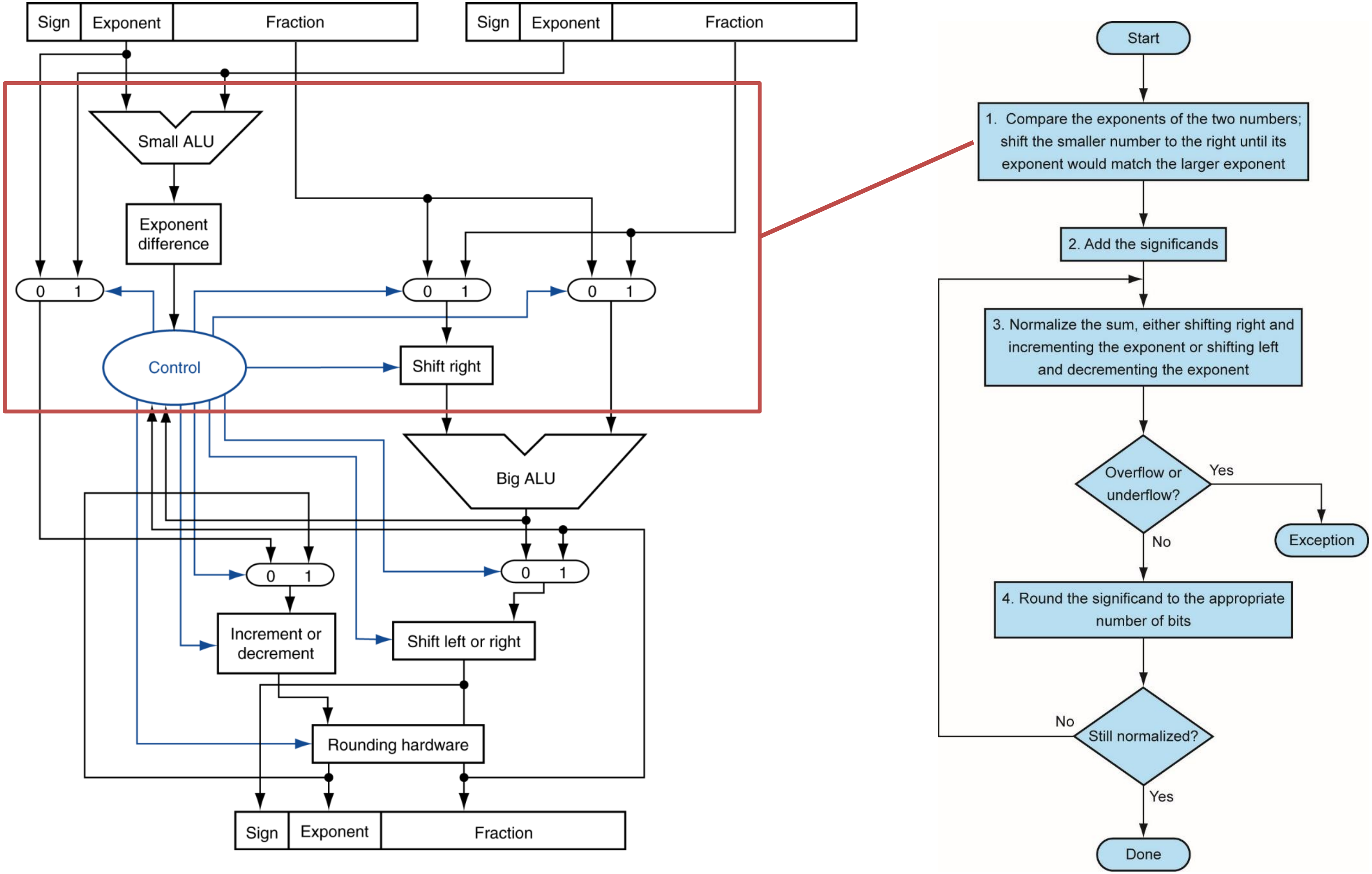
- Much more complex than integer adder.
- Doing it in one clock cycle would take too long.
  - Much longer than integer operations.
  - Slower clock would penalize all instructions.
- Floating-point adder usually takes several cycles
  - Can be pipelined

Floating-point adder hardware

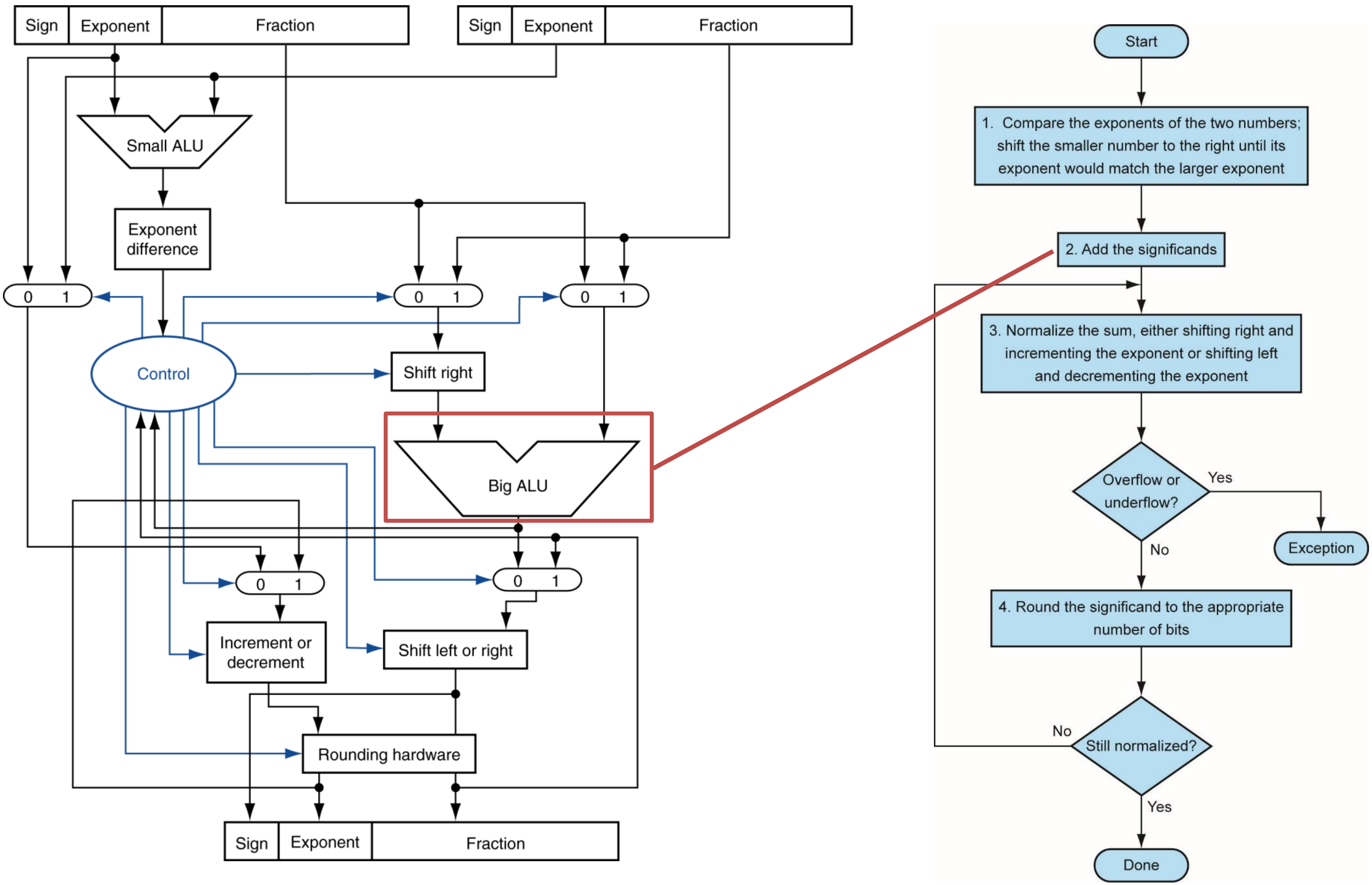




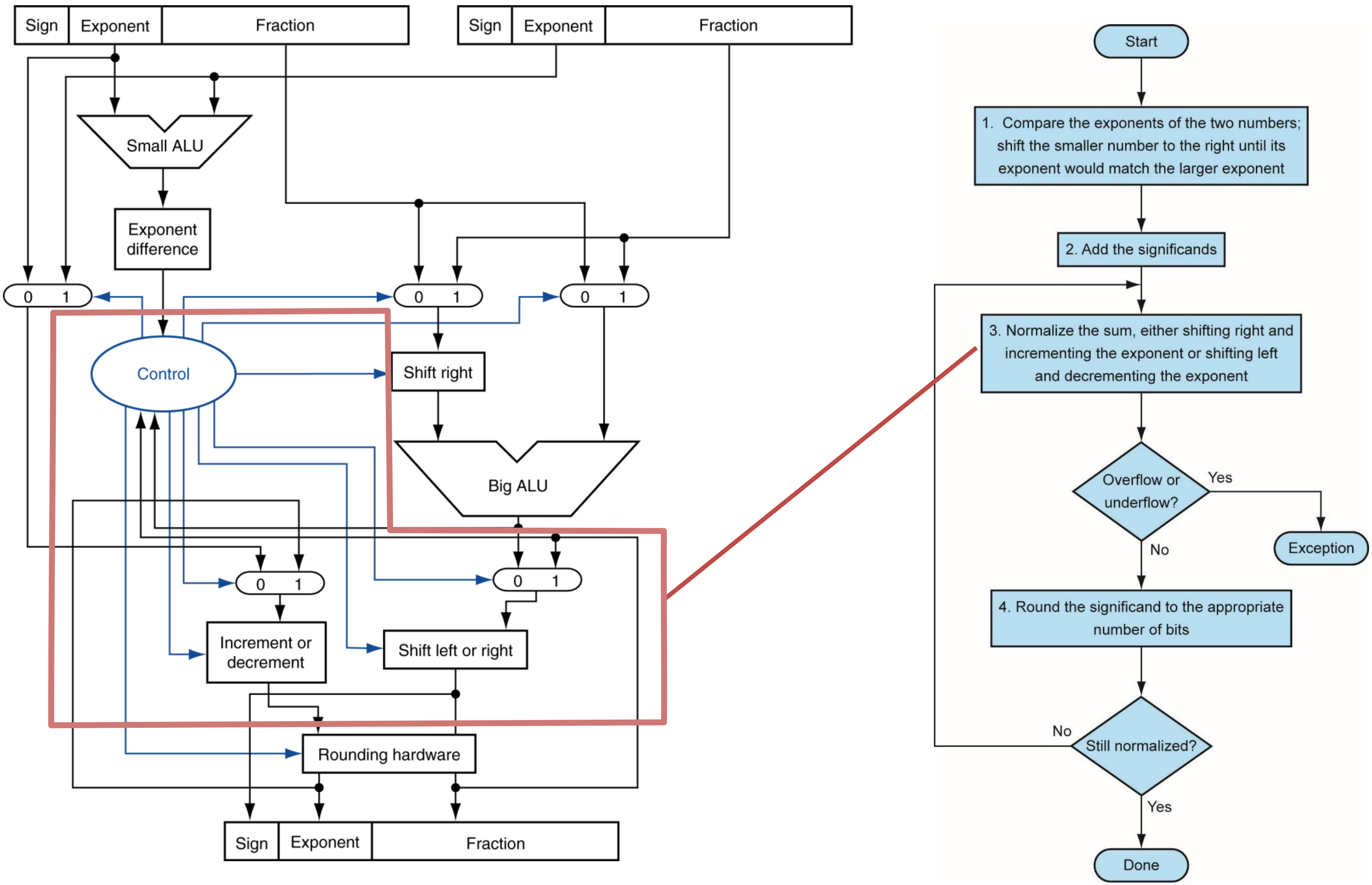
Floating-point adder hardware



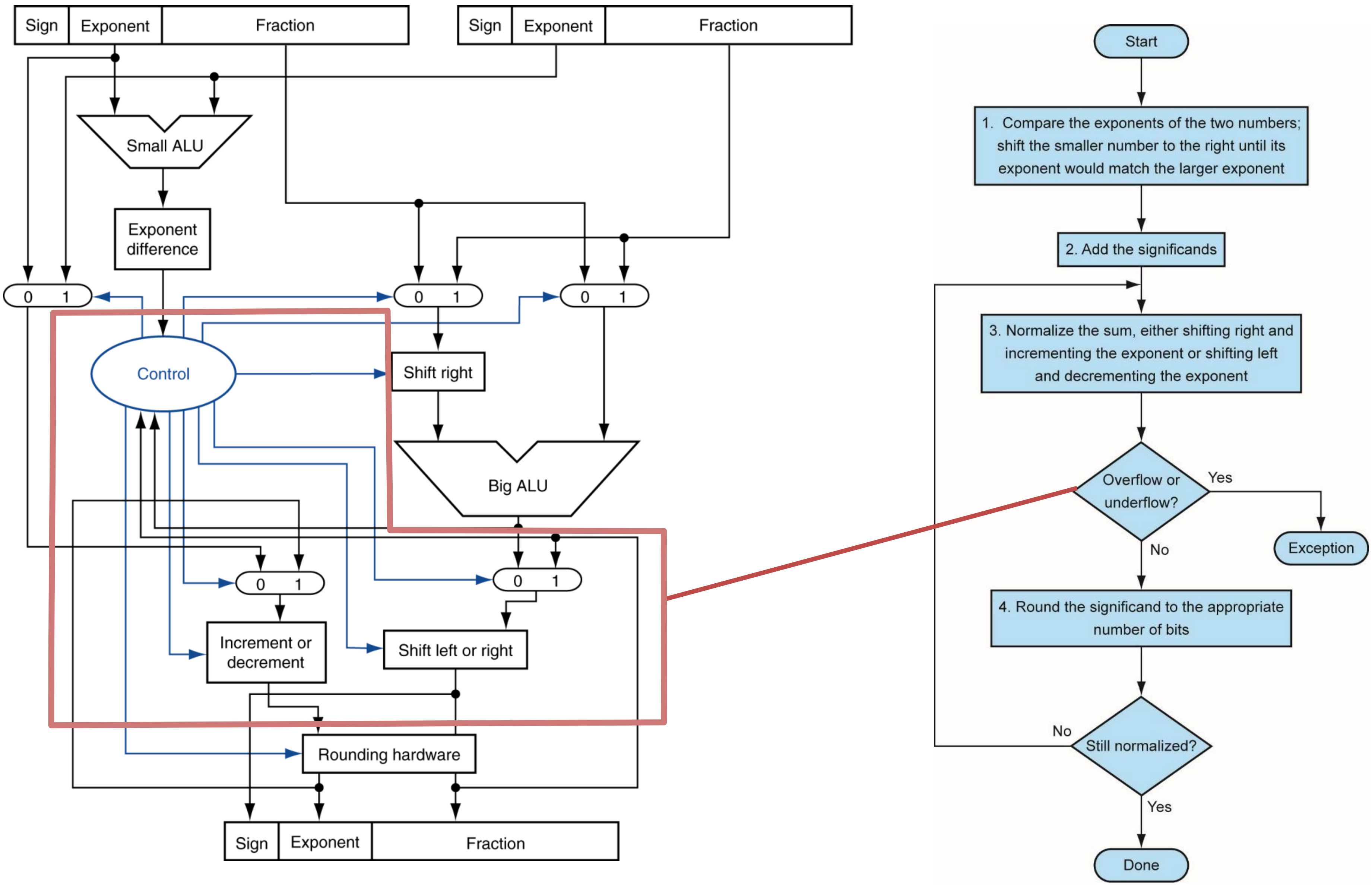
Floating-point adder hardware



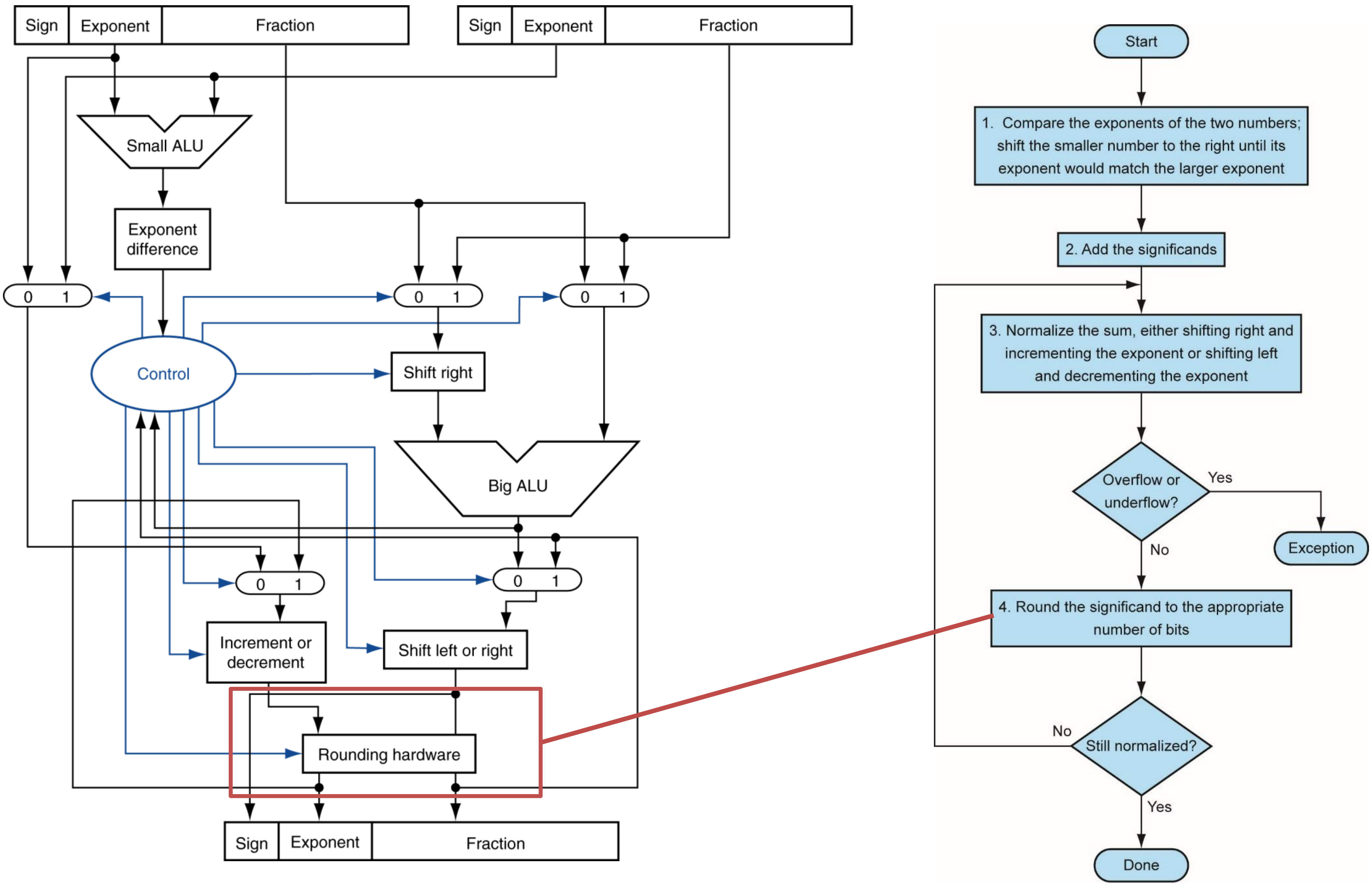
Floating-point adder hardware



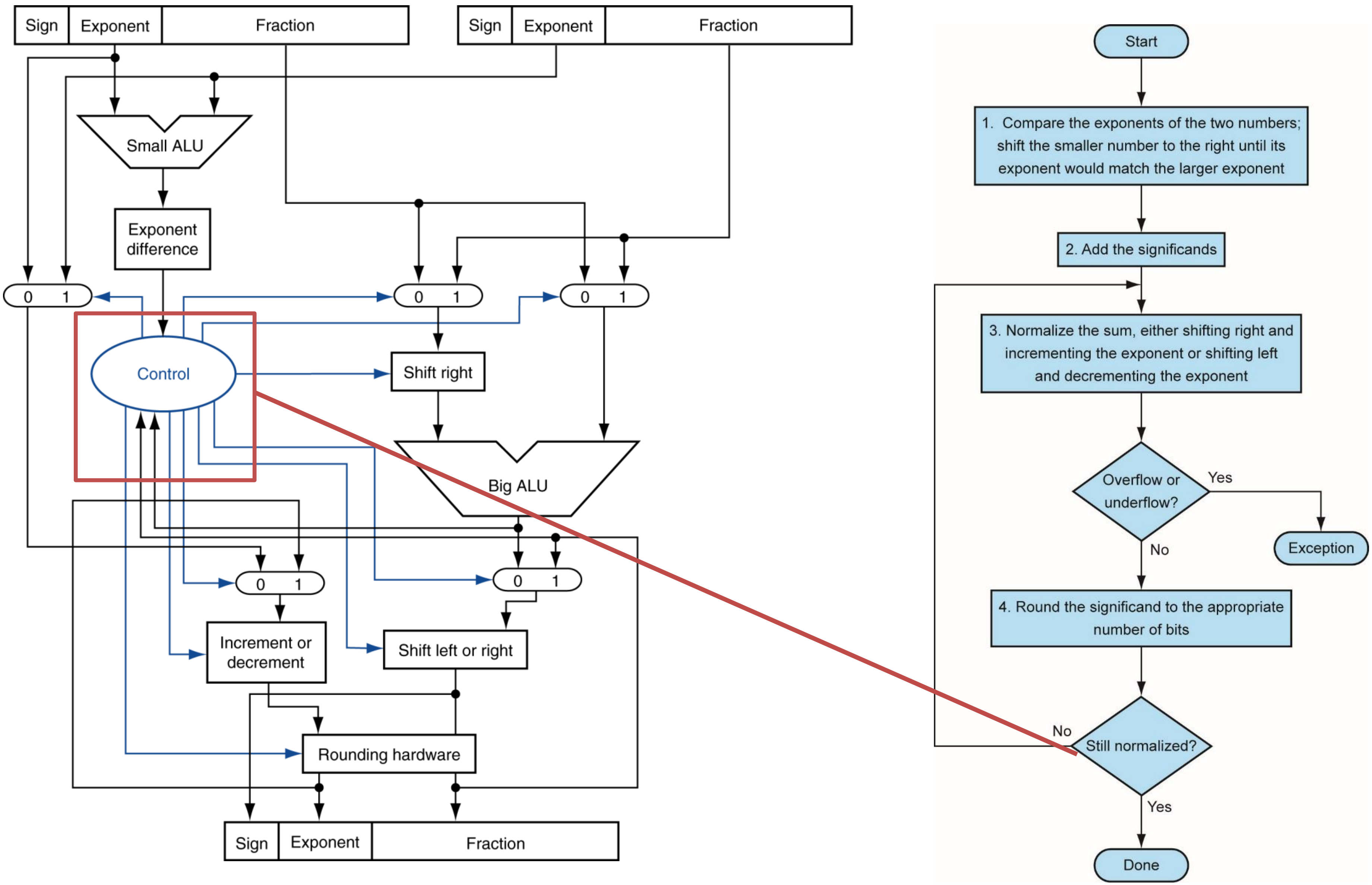
Floating-point adder hardware



Floating-point adder hardware



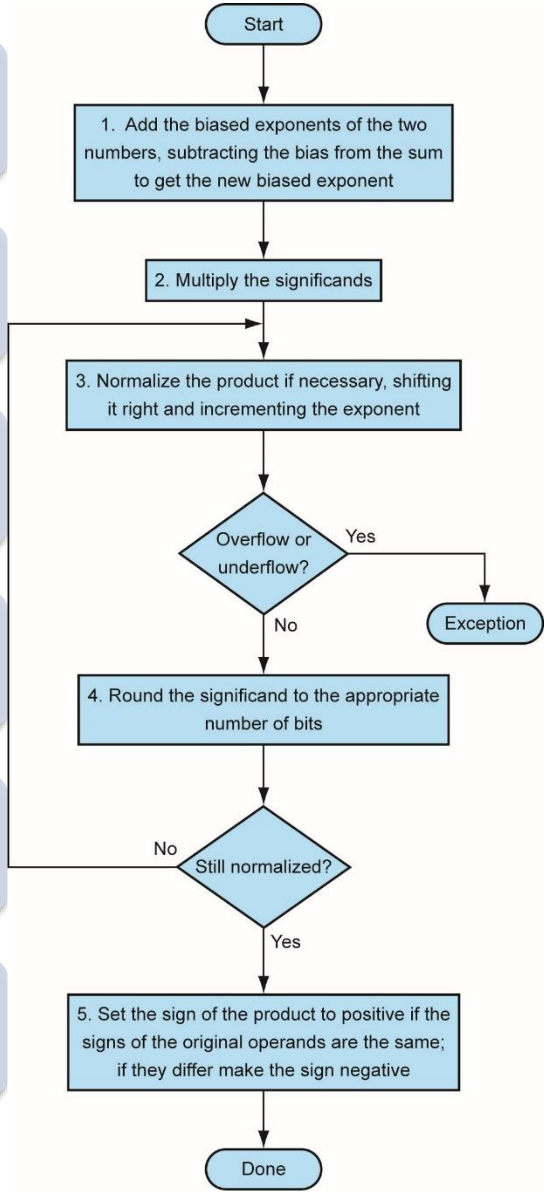
Floating-point adder hardware



Floating-point multiplication

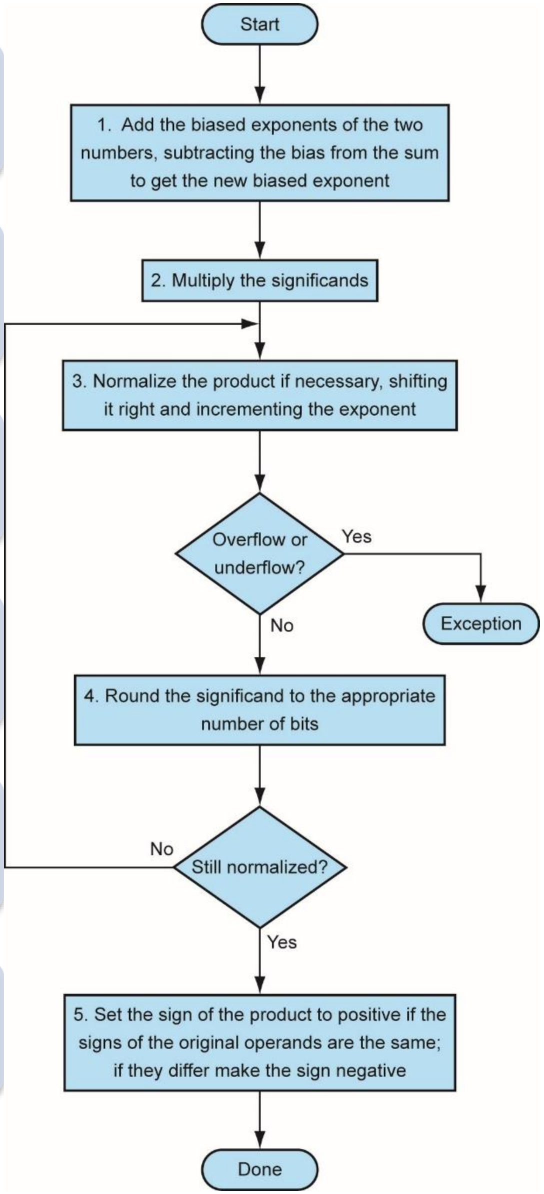
Floating-point multiplication

Consider a 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

Consider a floating-point example	-14.25×3.125 in decimal, or $-1.11001_2 \times 2^3 \times 1.1001_2 \times 2^1$
1. Add exponents	• Unbiased: $3 + 1 = 4$ • Biased: $(3+127)+(1+127)-127$ $= 4 + 254 - 127 = 4 + 127$
2. Multiply significands	$-1.11001_2 \times 1.1001_2$ $= 10.110010001_2$ $\Rightarrow 10.110010001_2 \times 2^4$
3. Normalize result & check for over/underflow	$1.0110010001_2 \times 2^5$ with no over/underflow
4. Round and renormalize if necessary	$1.0110010001_2 \times 2^5$ (no change)
5. Determine sign of result from signs of operands	$-1.0110010001_2 \times 2^5 = -44.53125$



Floating-point multiplication

- What's the floating-point of the previous result?

$$-14.25_{10} \times 3.125_{10} = -44.53125_{10}$$

$$(-1.11001_2 \times 2^3) \times (1.1001_2 \times 2^1) = -1.0110010001 \times 2^5$$

  - sign = 1
  - biased exponent
    - Single precision: $5 + 127 = 132 \rightarrow 10000100$
    - Double precision: $5 + 1023 = 1028 \rightarrow 10000000100$
  - mantissa = 01100100010000000000000
  - Floating point representation:

Single precision: 1 10000100 01100100010000000000000

Double precision: 1 10000000100 011001000100 ... 00

Floating-point arithmetic hardware

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