

Computer Organization and Design

Chapter 3 Part 3

Arithmetic for Computers

(Ch. 3 – Section 3.5)

Floating-Point Representation and Operations

IEEE-754

- Defines the rules for representing floating-point numbers (called the IEEE-754 Floating-Point Standard)
- Standard has been implemented by computer manufacturers since the mid-1980s
- Purpose:
 - make it easier to port computer programs that performed floating-point arithmetic across different architectures
 - improve the quality (accuracy) of computer arithmetic
- Revision adopted in 2008 extends the standard with new features.

IEEE-754 Standard

- A technical standard for floating-point arithmetic
 - Established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE)
 - The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably.
 - Most hardware floating-point units use the IEEE-754 standard.
 - IEEE rules ask for standards be revisited periodically for updating.
 - 2000 committee drafted revised standards that were approved in 2008 to include additional features
 - Most recent revision in 2019 defined additional requirements

IEEE-754 Standard

- The standard defines:
 - arithmetic formats: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
 - interchange formats: encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form
 - rounding rules: properties to be satisfied when rounding numbers during arithmetic and conversions
 - operations: arithmetic and other operations (such as trigonometric functions) on arithmetic formats
 - exception handling: indications of exceptional conditions (such as division by zero, overflow, etc.)

Binary Floating-Point Representation

RISC-V F (Single-Precision) and
D (Double-Precision) Extensions

- Also called real number representation
- Numbers consist of an integer and fractional component
- General form for floating-point numbers is

$$(-1)^S \times F \times 2^E$$

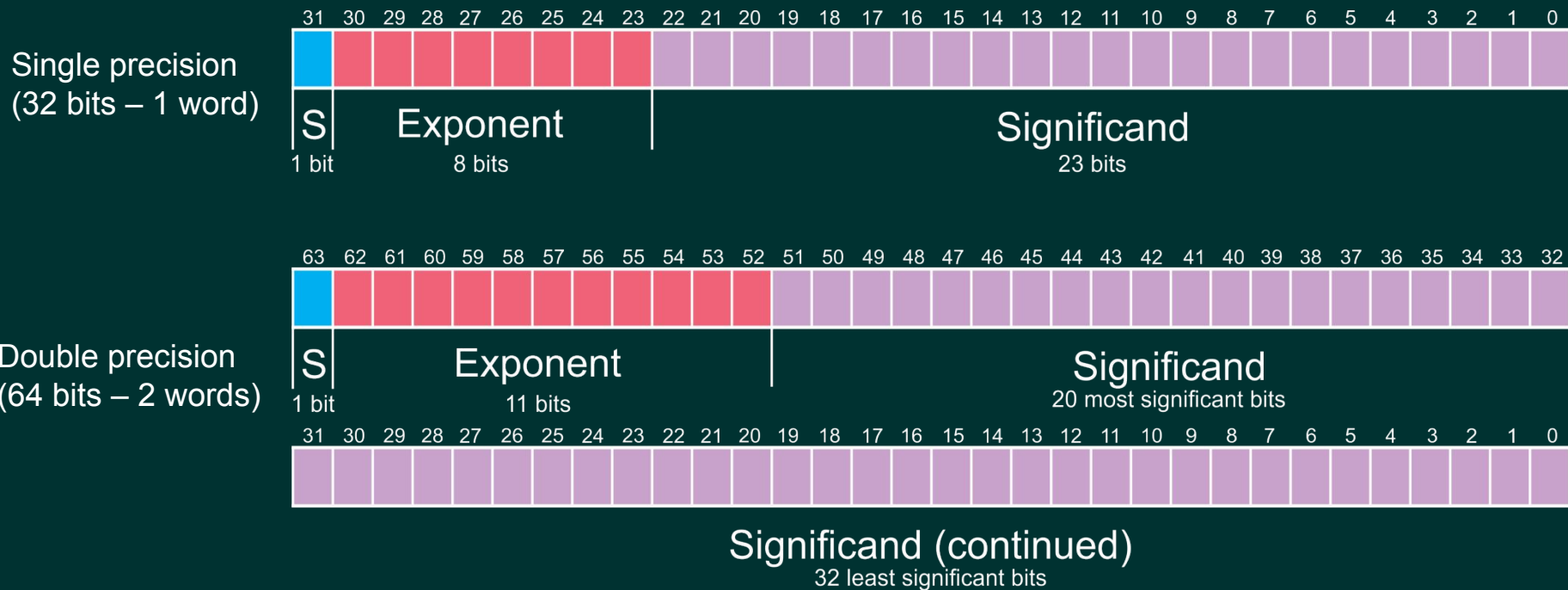
Where S = sign bit

F = significand (also called mantissa)

E = exponent

Storage of Floating-Point Numbers

- Floating-point numbers are a 32-bit or 64-bit value
- In most architectures, there is single precision and double precision representation
 - 32-bit single precision, 64-bit double precision



Floating-Point Characteristics

- Increasing the number of bits in the exponent increases the range of numbers that can be represented
- Increasing the number of bits in the significand increases the accuracy of numbers
- Thus there is a tradeoff between range and accuracy requiring a compromise between these two factors
 - Single precision range: $\sim 2.0_{10} \times 10^{-38}$ to $2.0_{10} \times 10^{+38}$
 - Approximately 5 digit accuracy
 - Double precision range: $\sim 2.0_{10} \times 10^{-308}$ to $2.0_{10} \times 10^{+308}$
 - Approximately 15 digit accuracy

Floating-Point Characteristics (continued)

- Overflow and underflow refer to issues that arise when the value being represented exceeds the range that can be handled by the floating-point format.
- Overflow
 - Values exceed the maximum representable limit, resulting in infinity or an error.
- Underflow
 - Values fall below the minimum representable limit (in absolute value), resulting in rounding to zero or representation as a subnormal number.
- Both phenomena can affect calculations, leading to loss of precision or incorrect results in numerical computations.

Normalized Representation

- Floating-point numbers are stored in a computer in **normalized form** – no leading zeros in the significand
- In binary, the form that represents normalized numbers is $1.nnnnnn_2 \times 2^{eeee}$
- The MSB of the significand is always 1
- This fact allows the computer to consider the MSB an implied 1 thus extending the actual number of bits of the significand in single precision to 24 bits (53 bits for double precision)

$$0.00010010... \quad \square \quad 1.001... \times 2^{-4} \quad \square \quad .001... \times 2^{-4}$$

Normalization Example

- Original binary floating-point value:

$$0.00010010 \times 2^0$$

- Shift the bits left until there is a 1 on the left side of the binary point and adjust the base 2 exponent (4-bit left shift: decrement exponent)

$$1.001... \times 2^{-4}$$

- The resulting floating-point value with the implied 1 omitted

$$.001... \times 2^{-4}$$

Normalized Representation (continued)

- Since the value zero does not have a leading 1, it is given the reserved exponent value of zero so the hardware won't automatically insert the leading 1
 - Significand bits are usually also all zeros
- The representation of floating-point for all numbers except zero is of the form
$$(-1)^S \times (1 + \text{significand}) \times 2^E$$
where the bits of the significand represent the fraction between 0 and 1 and E represents the exponent which can be positive or negative

Positive-Negative

- Floating-point numbers can be positive or negative
- Bit 31 (single precision) and bit 63 (double precision) represents the sign of the number just like for signed integers
- Exponents can also be positive or negative
- Desirable to have a floating-point representation in which the numbers can be easily processed in comparisons, for example sorting
- Having the exponent located before the significand makes comparisons seem simple since the larger the exponent, the larger the number

Negative Exponents

- If we adopt the convention of using the MSB of the exponent as a sign bit for the exponent, a problem arises when the exponent is negative
 - Since the significand is treated as unsigned, a fraction between 0 and 1, the exponent would become a 2's complement value
 - This complicates the certain operations, such as comparisons
 - Also would have two sign bits to deal with:
 - One for the number
 - One for the exponent

Exponent Bias

- The floating-point standard includes a convention by which the most negative exponent is represented as one and the most positive exponent is represented as all ones
- This is called **biased notation**
 - Also called excess notation
 - Biased notation forces the exponent to be an unsigned value when represented in hardware
- The bias is a number that is added to the value of the exponent when it is defined in the hardware.
 - To obtain the original exponent value, the bias must be subtracted from the stored exponent value

Exponent Bias

- Bias values
 - For single precision, the bias = 127
 - Also called excess-127
 - For double precision, the bias = 1023
 - Also called excess-1023
 - Bias value is automatically added to exponent when stored in hardware
- New floating-point representation:
$$(-1)^S \times (1 + \text{significand}) \times 2^{(\text{Exponent} - \text{Bias})}$$
- Example: from previous normalization example, the exponent of -4 would be stored as the unsigned value 123.

Floating-Point Accuracy

- For integers, given a specific number of bits, you can represent the exact value for a number
- With floating-point numbers, there is an infinite number of values but the hardware is limited to a finite number of bits to represent them
- In reality, floating-point numbers are approximations of some numbers that can't be exactly represented
- At best, we can only get so close to the actual value of a number

Floating-Point Accuracy (continued)

- Normalization is an attempt to improve accuracy
- Example using decimal numbers:
- $.12288 \times 10^4$
 $.012288 \times 10^5$
 $.0012288 \times 10^6$

Each of these numbers is the same value, just different representations.
- If we only allow max 5 digits to use for the mantissa, we can see how this limitation affects the accuracy

$$\begin{aligned} &.12288 \times 10^4 \\ &.01228 \times 10^5 \\ &.00122 \times 10^6 \end{aligned}$$

These numbers are no longer equal values due to truncation of the least significant digits of the Mantissa.

Floating-Point Accuracy (continued)

- If you don't normalize the binary significand, you lose significant bits resulting in reduced accuracy
- Allowing an implied 1 as the MSB of the significand (not actually having to store it) gives us one extra bit of accuracy
 - With 23 bits, we actually can represent a 24-bit significand
- Round off errors are representational errors caused by attempting to represent a real number in a finite number of significant digits

Round Off Errors

- Example: $1/5 = .2_{10} = 0.001100110011..._2$
- Representing $.2_{10}$ in a finite number of bits may result in the value $.1999999_{10}$ (not exactly $.2$)
- This small rounding error can lead to errors due to propagation:
$$1/5 + 1/5 + 1/5 + 1/5 + 1/5 \neq 1$$
- IEEE-754 standard includes specifications for rounding to assist the programmer (compiler) achieve the highest accuracy

IEEE-754 Rounding (1)

- Five methods of rounding divided into two groups
- Rounding to nearest
 - Round to nearest, ties to even
 - Rounds to the nearest value; if the number falls midway, it is rounded to the nearest value with an *even* least significant digit
 - This is the default for binary floating-point per the standard
 - Round to nearest, ties away from zero
 - Rounds to the nearest value; if the number falls midway, it is rounded to the nearest value above (for positive numbers) or below (for negative numbers)

IEEE-754 Rounding (2)

- Directed roundings
 - Round toward zero
 - Directed rounding towards zero, also known as **truncation**
 - Round toward $+\infty$
 - Directed rounding towards positive infinity, also known as rounding up or ceiling
 - Round toward $-\infty$
 - Directed rounding towards negative infinity, also know as rounding down or floor

IEEE-754 Rounding (3)

- The standard specifies that two extra LSB bits are to be kept during intermediate operations
- These extra bits are called the **guard** and **round** bits
- A third bit called the **sticky bit** is kept to indicate if there are non-zero bits to the right of the round bit.
 - Intended to guide more accurate rounding
- All of these extra bits are internal to the FPU and are not kept when storing the value into registers.

Floating-Point Hardware

- The hardware included in a computer for performing operations on floating-point numbers is generally referred to as the **floating-point unit (FPU)**
 - Other names include coprocessor, or math coprocessor
- Earlier computer systems, especially desktop systems, did not have coprocessor hardware; floating-point operations were handled by software
 - Only large mainframe type systems included FPUs
- In the 1980s, coprocessors were introduced as an optional separate IC (ex. Intel x87)
- Today's general-purpose processor technology incorporates FPU into the CPU circuitry

Decimal Real Numbers vs. Binary FP

- Decimal Real Number Place Values

... 10^4 10^3 10^2 10^1 10^0 . 10^{-1} 10^{-2} 10^{-3} 10^{-4} ...

↑
Decimal point

- Binary FP Place Values


... 2^4 2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} 2^{-4} ...

↑
Binary point

Converting Decimal Real Numbers to Binary Single-Precision Floating-Point

25.796875_{10}

1. Convert integer part

| | | | | |
|---------------|---|---|-----------------------------------------------------------------------------------|-----|
| $25 / 2 = 12$ | r | 1 |  | LSB |
| $12 / 2 = 6$ | r | 0 | | |
| $6 / 2 = 3$ | r | 0 | | |
| $3 / 2 = 1$ | r | 1 | | |
| $1 / 2 = 0$ | r | 1 | | MSB |

2. Convert fraction part

| | | | |
|-----------------|---|---|--------|
| $.796875 * 2 =$ | 1 | r | .59375 |
| $.59375 * 2 =$ | 1 | r | .1875 |
| $.1875 * 2 =$ | 0 | r | .375 |
| $.375 * 2 =$ | 0 | r | .75 |
| $.75 * 2 =$ | 1 | r | .5 |
| $.5 * 2 =$ | 1 | r | 0 |

3. Combine binary integer & fraction and add the exponent

$$11001.110011 \times 2^0$$

Continue next slide ...

Converting Decimal Real Numbers to Binary Single-Precision Floating-Point

$$11001.110011 \times 2^0$$

4. Normalize the number: need to shift the bits until there is a single 1 to the left of the binary point. Shift right 4 bits. Exponent must increment by the number of bits shifted. New exponent = 4.

$$1.1001110011 \times 2^4$$

5. Determine the sign bit, 8-bit biased exponent and 23-bit stored significand (implied bit not stored)

Sign bit = 0 (for positive number)

Biased exponent = $4 + 127 = 131 = 10000011$

Significand = 10011100110000000000000

Continue next slide ...

Converting Decimal Real Numbers to Binary Single-Precision Floating-Point

Sign bit = 0 Biased exponent = 10000011

Significand = 100111001100000000000000

5. State full 32-bit binary stored binary value and hex value.

sign exponent significant

0 100 0001 1100 1110 0110 0000 0000 0000

0x41ce6000

6. Verify using RARS

```
fp1:      .data
          .float    25.796875
```

| Data Segment | |
|--------------|------------|
| Address | Value (+0) |
| 0x10010000 | 0x41ce6000 |

Converting Binary Single-Precision Floating-Point to Decimal

0xbe200000

10111110001000000000000000000000

1. Divide the binary value into the three FP parts: sign bit, exponent, significand

| sign | exponent | significand |
|------|----------|--------------------------|
| 1 | 01111100 | 010000000000000000000000 |

2. Determine unbiased exponent

$01111100 = 124$

Subtract bias: $124 - 127 = -3$

Continue next slide ...

Converting Binary Single-Precision Floating-Point to Decimal

3. Insert the implied 1 in the significand and place the binary point after it.

$$1.010000000000000000000000$$

4. State the binary value of the significand with the base 2 exponent

$$1.010000000000000000000000 \times 2^{-3}$$

5. Shift the significand bits in the direction to achieve an exponent of zero. The exponent must increment from -3, so the bits must be shifted right by 3 bit positions (inserting zeros from the left).

$$.001010000000000000000000 \times 2^0$$

Continue next slide ...

Converting Binary Single-Precision Floating-Point to Decimal

.00101000000000000000000000000000 $\times 2^0$

6. Convert the binary value to decimal. Sum the binary place values.

$$2^{-3} + 2^{-5} = .15625$$

7. State the final value as decimal adding the sign of the number.
(Original sign bit = 1 for negative value.)

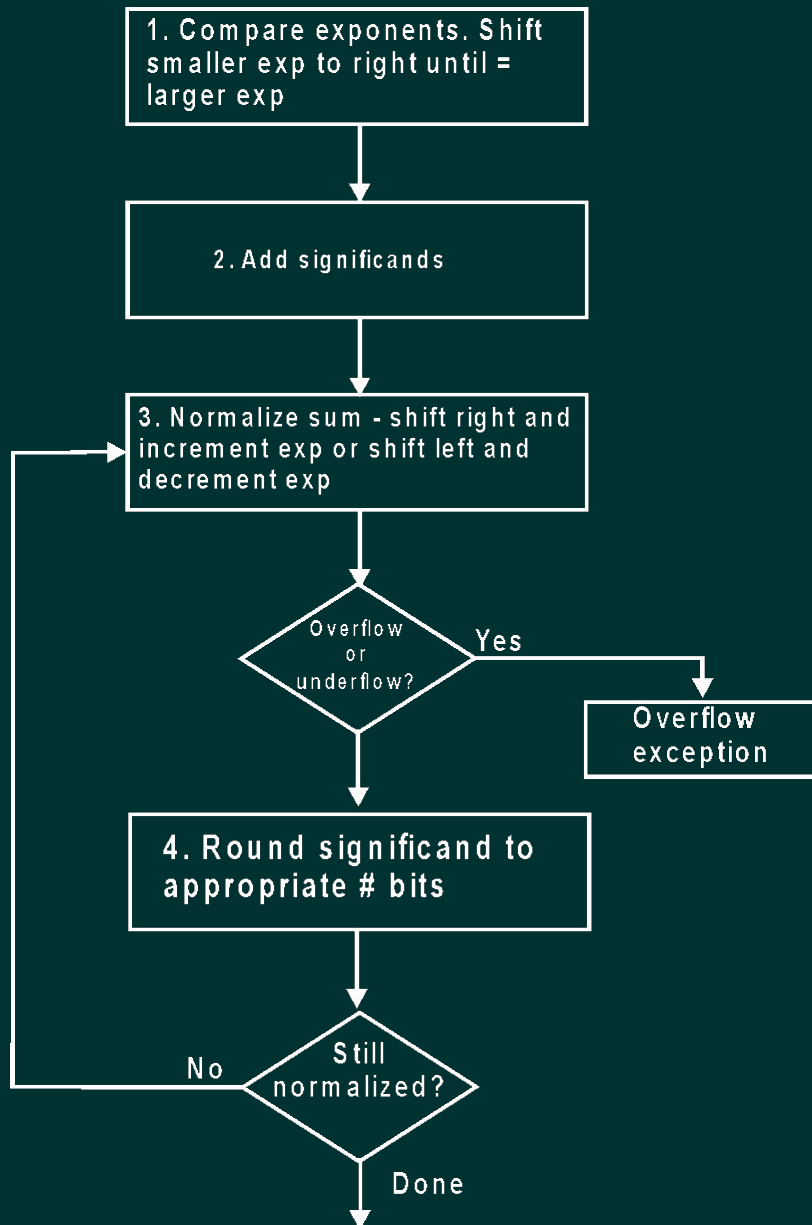
$$- 0.15625_{10}$$

8. Check using RARS.

```
fp2:      .data      -0.15625
          .float
```

| Data Segment | |
|--------------|------------|
| Address | Value (+0) |
| 0x10010000 | 0xbe200000 |

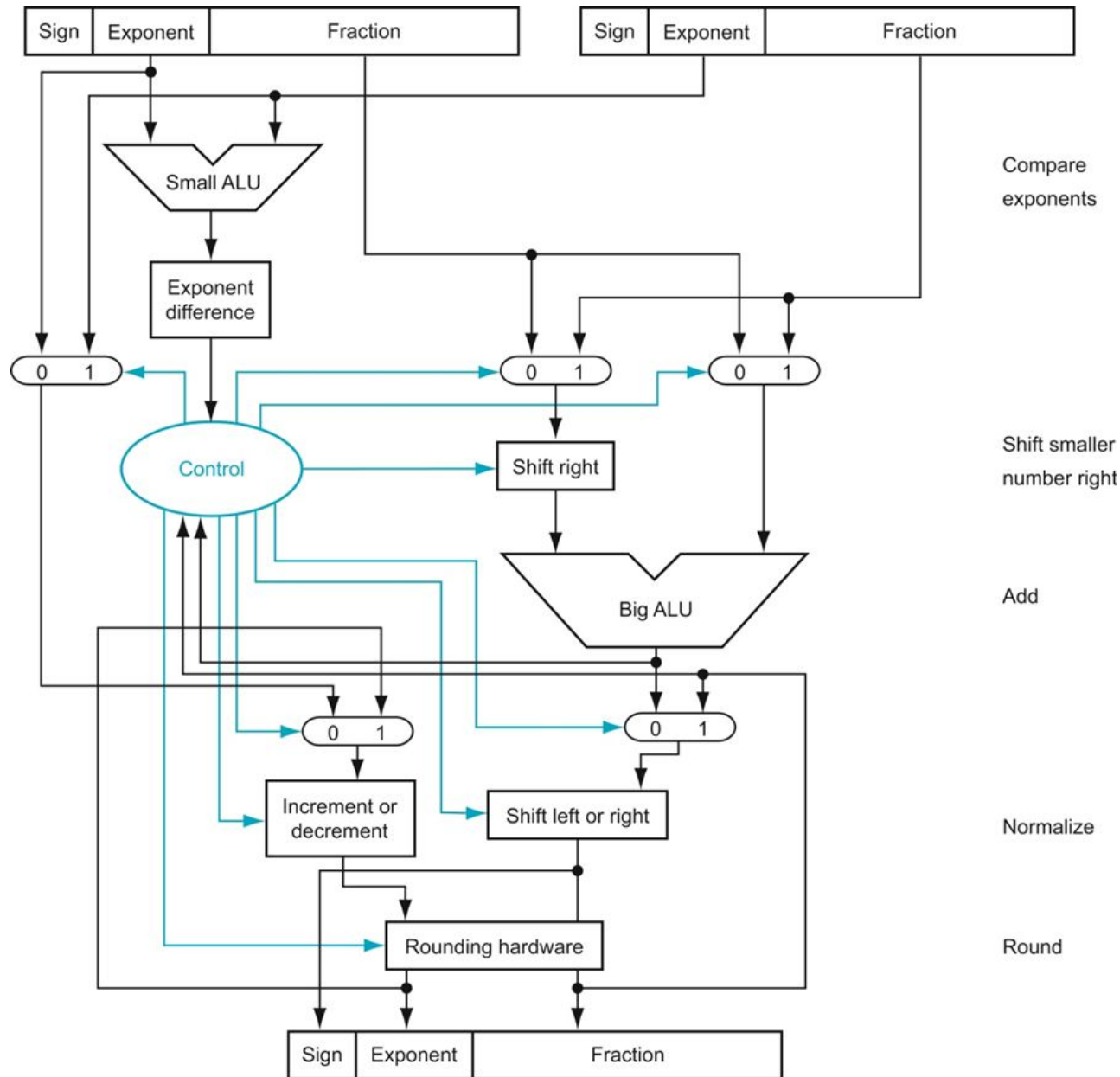
Floating-Point Addition



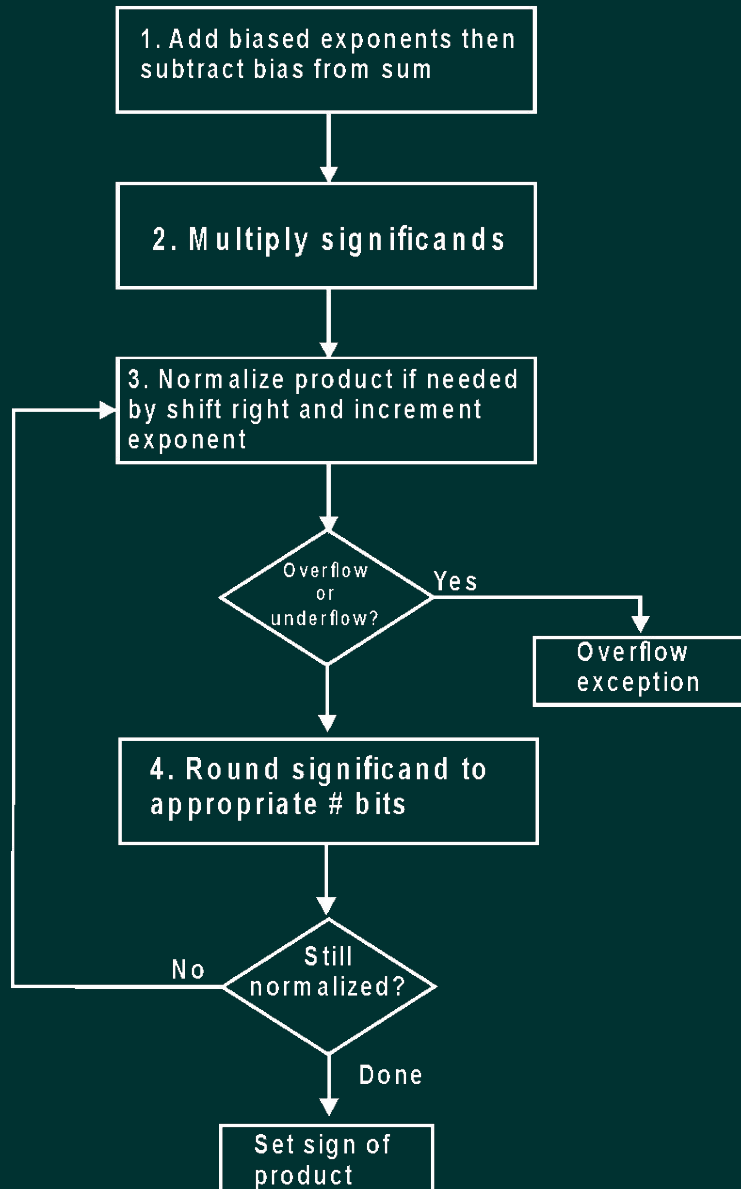
The normal path is to execute steps 3 and 4 only once, but if rounding causes the sum to become un-normalized, then step three has to be repeated.

If step 3 repeats, step 4 should have no affect on the sum the second time through.

ALU for Floating-Point Addition



Floating-Point Multiplication



Adding the exponents in step 1 generates too large a value for the exponent field because the original exponents are biased.

- doubled exponent value

To compensate, you have to subtract the bias after adding the exponents to get the correct exponent value.

Floating-Point Division

- Step 1: if divisor $\neq 0$, divide by zero exception
- Step 2: Subtract the exponent of the divisor from the exponent of the dividend
- Step 3: Divide the significands
- Step 4: Normalize the quotient if necessary by shifting it right and incrementing the exponent
- Step 5: Check for overflow/underflow
- Step 6: Round significand to appropriate number of bits
- Step 7: Check if still normalized; if not, repeat step 4
- Step 8: Set the sign of the quotient to positive if the signs of the divisor and dividend are the same; otherwise set the sign of the quotient to negative.

IEEE-754

- Other definitions in spec
- Special symbols (bit patterns) for unusual events
 - Divide by zero replaced with $+\infty$ or $-\infty$ (programmer/compiler option)
 - NaN (not a number) to represent $0/0$ or $\infty - \infty$
- Allows programmers to postpone some tests or decisions until later in the program

| Single Precision | | Double Precision | | Object Represented |
|------------------|-------------|------------------|-------------|-----------------------------|
| Exponent | Significand | Exponent | Significand | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | Non-zero | 0 | Non-zero | \pm denormalized number |
| 1 – 254 | Anything | 1 – 2046 | Anything | \pm floating-point number |
| 255 | 0 | 2047 | 0 | \pm infinity |
| 255 | Non-zero | 2047 | Non-zero | NaN |

RISC-V Floating-Point

- Addition: fadd.s (single) fadd.d (double)
- Subtractions: fsub.s fsub.d
- Multiplication: fmul.s fmul.d
- Division: fdiv.s fdiv.d
- Square root: fsqrt.s fsqrt.d
- Comparisons: feq.s feq.d
- flt.s flt.d
- fle.s fle.d
 - Comparisons set integer register to 0 (false) or 1 (true)
 - Integer branch instructions (beq, bne) used to branch
- Loads/Stores: flw fld
- fsw fsd

Floating-Point Registers

- Floating-point registers are numbered f0 – f31
- Floating-point register **names** and uses:
 - ft0 – ft11 (f0 – f7, f28 – f31): FP temporary registers (t)
 - fs0 – fs11 (f8 – f9, f18 – f27): FP saved registers (s)
 - fa0 – fa1 (f10 – f11): FP function argument, return value registers (a)
 - fa2 – fa7 (f12 – f17): FP function argument registers (a)
- FP register names are used in assembly language programming
- FP registers are 64 bits (RV64I w/ F & D extensions)
 - used for both single and double precision
 - single-precision occupies lower 32 bits

RISC-V Floating-Point (continued)

- Included in instruction set are conversion instructions
 - convert integer to floating-point and vice versa
 - both signed and unsigned integer conversions
 - both 32 and 64 bit conversions

| To | From | | | |
|-----------------------------|------------------------|---------------------------|-----------------------------|-----------------------------|
| | 32b signed integer (w) | 32b unsigned integer (wu) | 32b FP single-precision (s) | 64b FP double-precision (d) |
| 32b signed integer (w) | - | - | fcvt.w.s | fcvt.w.d |
| 32b unsigned integer (wu) | - | - | fcvt.wu.s | fcvt.wu.d |
| 32b FP single-precision (s) | fcvt.s.w | fcvt.s.wu | - | fcvt.s.d |
| 64b FP | | | | |

Operations Between Float & Double

- Operations between a float and a double cannot be directly performed due to representation
 - Float: 32-bit, 8-bit exponent, 23-bit significand
 - Double: 64-bit, 11-bit exponent, 52-bit significand
- Must convert to the same type before operation
 - Float \square Double (promotion) -or- Double \square Float
- In most HLLs, performing an operation between a float and a double causes the float value to be promoted to a double before the operation is performed.
 - This ensures that no precision is lost.
 - Result type: The result of the operation will be a double.

Copying FPR to GPR

- RISC-V includes two instructions to copy floating-point bits to integer register and vice versa
 - `fmv.s.x` - copy bits from integer register to FPR
 - `fmv.x.s` – copy bits from FPR to integer register
- Only needed in rare cases where the programmer needs to manipulate bits of floating-point number
 - e.g. extracting sign, exponent and significand
 - e.g. explicitly controlling how rounding is applied

RISC-V FP Rounding (1)

- RISC-V implements IEEE-754-2008 rounding modes
- The default rounding mode is RNE (Round to nearest, ties to even) specified in the fcsr register (floating-point control and status register)
- Static rounding is implemented that allows rounding modes to be defined for each floating-point computational instruction
 - An optional parameter is appended to the instruction that specifies the rounding mode

RISC-V FP Rounding (2)

- Rounding mode encoding

| Rounding Mode | Mnemonic | Meaning |
|---------------|----------|-----------------------------------------------------------------------------------------------------------------|
| 000 | RNE | Round to Nearest, ties to Even |
| 001 | RTZ | Round towards Zero |
| 010 | RDN | Round Down (towards $-\infty$) |
| 011 | RUP | Round Up (towards $+\infty$) |
| 100 | RMM | Round to Nearest, ties to Max Magnitude |
| 101 | | <i>Invalid. Reserved for future use.</i> |
| 110 | | <i>Invalid. Reserved for future use.</i> |
| 111 | DYN | In instruction's <i>rm</i> field, selects dynamic rounding mode; In Rounding Mode register, <i>Invalid</i> . |

- Floating-point arithmetic instruction format

| 31 | 27 26 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|-----------|-------|-------|-------|---------|-------|--------|---|
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode | |
| 5 | 2 | 5 | 5 | 3 | 5 | 7 | |
| FADD/FSUB | S | src2 | src1 | RM | dest | OP-FP | |
| FMUL/FDIV | S | src2 | src1 | RM | dest | OP-FP | |
| FSQRT | S | 0 | src | RM | dest | OP-FP | |
| FMIN-MAX | S | src2 | src1 | MIN/MAX | dest | OP-FP | |

Rounding mode field in instruction

RISC-V Assembly Language Rounding

- All the floating-point arithmetic and conversion instructions require specifying the rounding mode
- Rounding mode is appended to the end of the instruction as a fourth operand
- Dynamic rounding is preferred unless a different mode is required
- Example: `fadd.s ft3, ft2, ft1, dyn`
 - **dyn** specifies dynamic rounding using value of `frm` field in the `fcsr` which by default is 000 (RNE) round to the nearest even

RISC-V Example in RARS

```

.data
num1: .float    0.2
.text
main:  lui    s0, 0x10010
       flw    ft0, 0(s0)
       fmul.s  ft1, ft0, ft0, dyn
       fdiv.s  ft2, ft1, ft0, dyn
       fsqrt.s ft3, ft1, dyn
exit:   ori    a7, zero, 10
       ecall

```

0.2 is a binary repeating decimal

```
0.2 = 0x3e4ccccd = 0.2 (rounded)
→ = 0x3d23d70b = .040000003
→ = 0x3e4cccce = 0.20000002
→ = 0x3e4ccccd = 0.2
```

0011 1110 0100 1100 1100 1100 1100 1110

$$0.2_{10} = .001100110011\dots_2$$

$$1.100110011\dots \times 2^{-3} \text{ (normalized)}$$

0 0111100 10011001100110011001101 (rounded)

(Guard bit used to set LSB of significand)

Defining Floating-Point Values in RARS

```
.data
num1:  .float    123.456789
num2:  .float    123.456789e0
num3:  .float    1.23456789e2
num4:  .float    123456.789e-3
num5:  .float    123456789e-6  □ “implied decimal
point”
```

All the above definitions represent the same value.
Note: no spaces before or after the “e”.

Terminology / Vocabulary

- Binary addition & subtraction
- 2's complement
- Signed integers
- Unsigned integers
- Sign extension
- Overflow
- Exception/interrupt
- Exception handler
- Arithmetic and logic unit (ALU)
- Ripple-carry adder
- Carry-lookahead adder
- Restoring division
- Binary floating-point
- Single precision
- Double precision
- Normalized form
- Implied 1
- IEEE-754
- Biased exponent
- Rounding
- Guard/round bits
- Coprocessor or FPU