Floating Point

- An IEEE floating point representation consists of
  - A Sign Bit (no surprise)
  - An Exponent (“times 2 to the what?”)
  - Mantissa (“Significand”), which is assumed to be 1.xxxxx (thus, one bit of the mantissa is implied as 1)
  - This is called a normalized representation
- So a mantissa = 0 really is interpreted to be 1.0, and a mantissa of all 1111 is interpreted to be 1.1111
- Special cases are used to represent denormalized mantissas (true mantissa = 0), NaN, etc., as will be discussed.

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

S	Exponent	Fraction

[illegible]

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: 1.0 ≤ |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

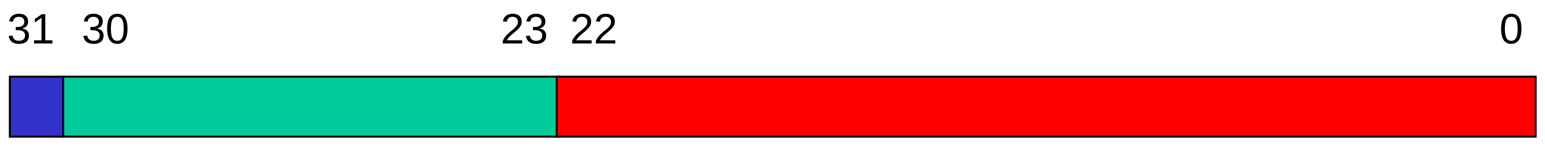
- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 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: 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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Representation of Floating Point Numbers

- **IEEE 754 single precision**



Sign Biased exponent Normalized Mantissa (implicit 24th bit = 1)

$$(-1)^s \times F \times 2^{E-127}$$

Exponent	Mantissa	Object Representec
0	0	0
0	non-zero	denormalized
1-254	anything	FP number
255	0	pm infinity
255	non-zero	NaN

Why biased exponent?

- For faster comparisons (for sorting, etc.), allow integer comparisons of floating point numbers:

- Unbiased exponent:

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

- Biased exponent:

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

Basic Technique

- **Represent the decimal in the form +/- $1.xxx_b$ x 2^y**
- **And "fill in the fields"**
  - **Remember biased exponent and implicit "1." mantissa!**
- **Examples:**
  - 0.0: 0 00000000 00000000000000000000000
  - 1.0 (1.0 x 2^0): 0 01111111 00000000000000000000000
  - 0.5 (0.1 binary = 1.0 x 2^-1): 0 01111110 00000000000000000000000
  - 0.75 (0.11 binary = 1.1 x 2^-1): 0 01111110 10000000000000000000000
  - 3.0 (11 binary = 1.1*2^1): 0 10000000 10000000000000000000000
  - -0.375 (-0.011 binary = -1.1*2^-2): 1 01111101 10000000000000000000000
  - 1 10000011 01000000000000000000000 = - 1.01 * 2^4 = -20.0

Basic Technique

- One can compute the mantissa just similar to the way one would convert decimal whole numbers to binary.
- Take the decimal and repeatedly multiply the fractional component by 2. The whole number portion is the next binary bit.
- For whole numbers, append the binary whole number to the mantissa and shift the exponent until the mantissa is in normalized form.

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000...00

  - S = 1
  - Fraction = $01000...00_2$
  - Fxponent = $10000001_2 = 129$

- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Converting <u>to</u> Floating Point

- E.g., Express 36.5625_{10} as a 32-bit floating point number (in hexadecimal)

- Step 1
  - Express original value in binary

36.5625_{10} =

100100.1001_2

- Step 2
  - Normalize

$100100.1001_2 =$

$1.001001001_2 \times 2^5$

- Step 3
  - Determine S, E, and M

$$\underbrace{+}_{S}1.\underbrace{001001001}_{M}{}_2 \times 2^{\underbrace{5}_{n}}$$

$$\boxed{\begin{aligned} E &= n + 127 \\ &= 5 + 127 \\ &= 132 \\ &= 10000100_2 \end{aligned}}$$

$$\boxed{S = 0 \text{ (because the value is positive)}}$$

- Step 4
  - Put S, E, and M together to form 32-bit binary result

$$\underbrace{0}_{S}\quad\underbrace{10000100}_{E}\quad\underbrace{00100100100000000000000}_{M}{}_2$$

- Step 5
  - Express in hexadecimal

```
0 10000100 00100100100000000000000₂ =

0100 0010 0001 0010 0100 0000 0000 0000₂ =

  4    2    1    2    4    0    0    0₁₆
```

Answer: 42124000_{16}

Converting <u>from</u> Floating Point

- E.g., What decimal value is represented by the following 32-bit floating point number?

$\texttt{C17B0000}_{16}$

- Step 1
  - Express in binary and find S, E, and M

$\mathtt{C17B0000}_{16} =$

$$\underbrace{\mathtt{1}}_{S}\ \underbrace{\mathtt{10000010}}_{E}\ \underbrace{\mathtt{11110110000000000000000}}_{M}{}_{2}$$

1 = negative
0 = positive

- Step 2
  - Find “real” exponent, *n*
  - $n = E - 127$

    $= 10000010_2 - 127$

    $= 130 - 127$

    $= 3$

- Step 3
  - Put S, M, and *n* together to form binary result
  - (Don’t forget the implied “1.” on the left of the mantissa.)

$-1.1111011_2 \text{ x } 2^n =$

$-1.1111011_2 \text{ x } 2^3 =$

-1111.1011_2

- Step 4
  - Express result in decimal

$$\underbrace{-1111}_{-15}.\underline{1011}_2$$

$$\begin{aligned}2^{-1} &= 0.5\\ 2^{-3} &= 0.125\\ 2^{-4} &= \underline{0.0625}\\ &\ 0.6875\end{aligned}$$

Answer: -15.6875

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

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

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

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

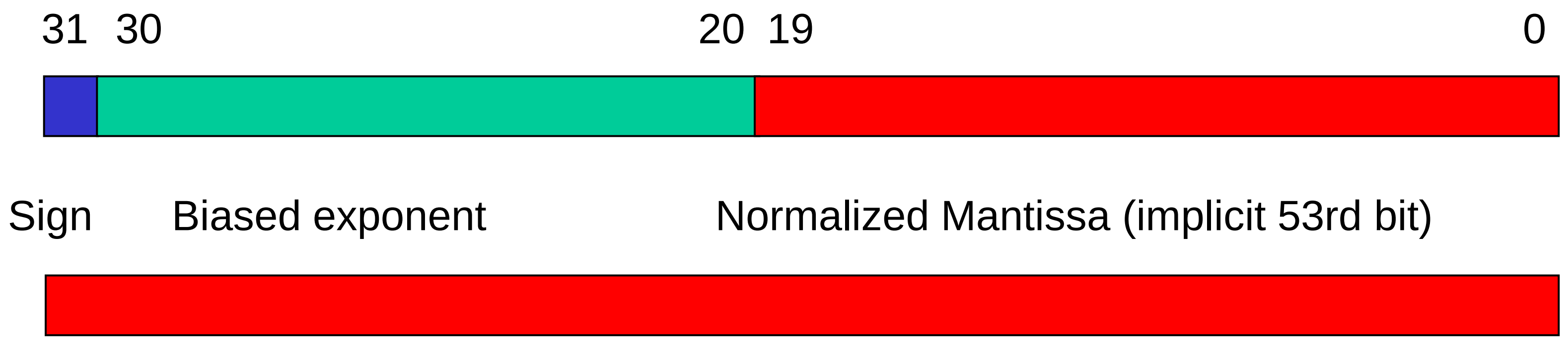
Two representations of 0.0!

Infinities and NaNs

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

Representation of Floating Point Numbers

- **IEEE 754 double precision**



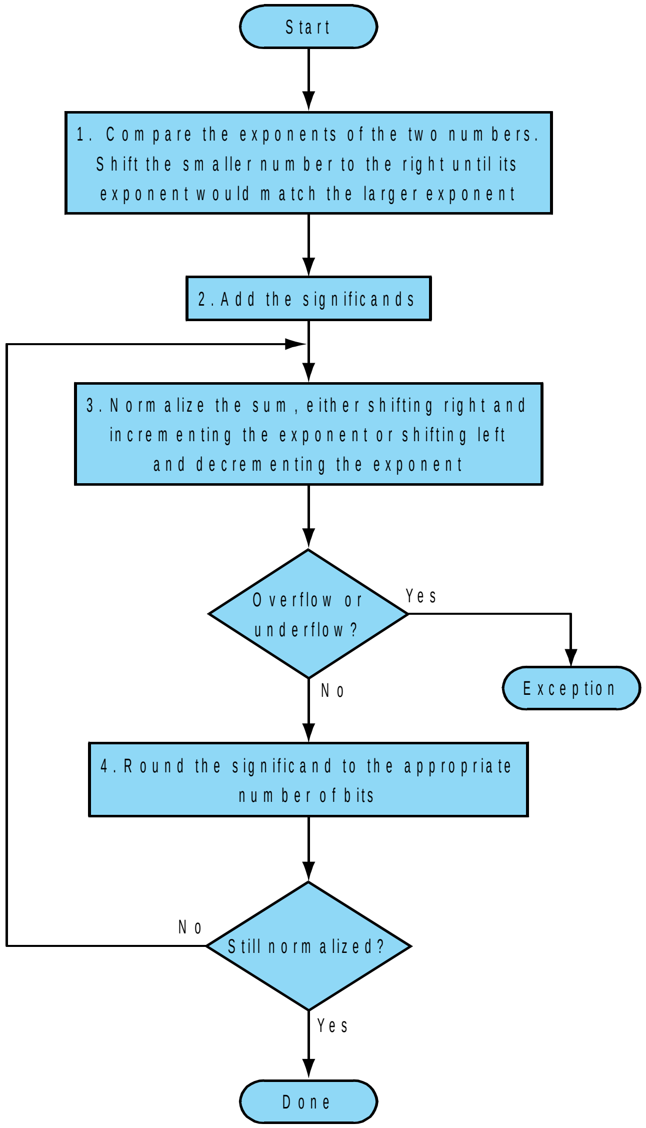
$$(-1)^s \times F \times 2^{E-1023}$$

Exponent	Mantissa	Object Representec
0	0	0
0	non-zero	denormalized
1-2046	anything	FP number
2047	0	pm infinity
2047	non-zero	NaN

Is FP addition associative?

- Associativity law for addition: $a + (b + c) = (a + b) + c$
- Let $a = – 2.7 \times 10^{23}$, $b = 2.7 \times 10^{23}$, and $c = 1.0$
- $a + (b + c) = – 2.7 \times 10^{23} + (2.7 \times 10^{23} + 1.0) = – 2.7 \times 10^{23} + 2.7 \times 10^{23} = 0.0$
- $(a + b) + c = (– 2.7 \times 10^{23} + 2.7 \times 10^{23}) + 1.0 = 0.0 + 1.0 = 1.0$
- Beware – Floating Point addition not associative!
- The result is approximate…

Floating point addition


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

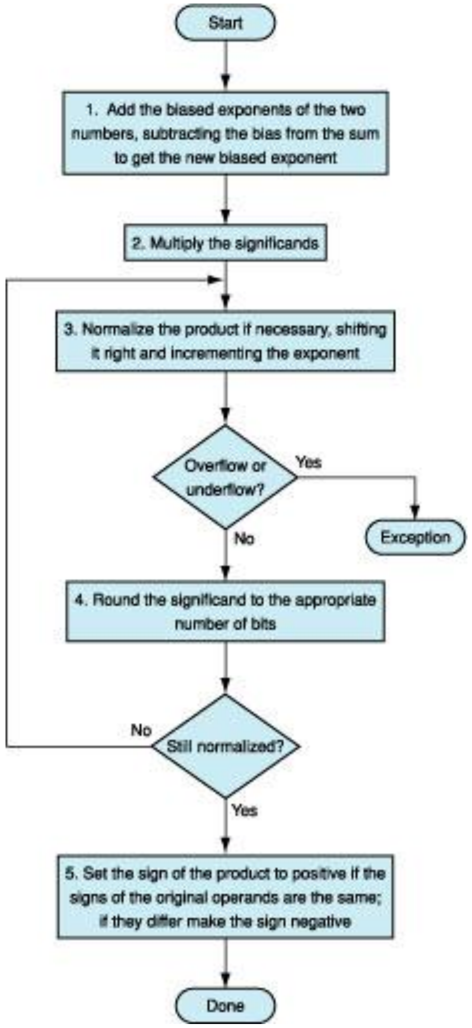
Floating-Point Addition

- 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
- FP adder usually takes several cycles
  - Can be pipelined

Floating Point Multiplication Algorithm


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

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 × –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 × –ve $\Rightarrow$ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$