Floating-point (FP) numbers

- Computers need to deal with real numbers
  - Fractional numbers (e.g., 3.1416)
  - Very small numbers (e.g., 0.000001)
  - Very larger numbers (e.g., 2.7596×10^9)
- Components in a binary FP number
  - $(-1)^{\text{sign}} \times$ *significand* (a.k.a. *mantissa*) $\times 2^{exponent}$
  - More bits in *significand* gives higher accuracy
  - More bits in *exponent* gives wider range
- A case for FP representation standard
  - Portability issues
  - Improved implementations
  - ⇒ IEEE-754

Representing "floats" with binary

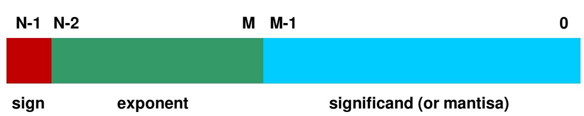
- We can add a "point" in binary notation:
  - 101.1010b
  - integral part is simply 5d
  - fractional part is $1 \times 2^{-1} + 1 \times 2^{-3} = 0.5 + 0.125 = 0.625$
  - thus, 101.1010b is 5.625d
- **Normal form**: shift "point" so there's only a leading 1
  - 101.1010b = 1.011010×2^2, shift to the left by 2 positions
  - 0.0001101 = 1.101×2^{-4}, shift to the right by 4 positions
  - typically, we use the normal form (much like scientific notation)
- Just like integers, we have a choice of representation
  - **IEEE 754** is our focus (there are other choices, though)

Format choice issues

- Example floating-point numbers (base-10)
  - 1.4×10^{-2}
  - $-20.0 = -2.00 \times 10^{1}$
- What components do we have?
  - $(-1)^{sign} \times$ *significand* (a.k.a. *mantissa*) $\times 2^{exponent}$
    - Sign
    - Significand
    - Exponent
- Representing sign is easy (0=positive, 1=negative)
- Significand is unsigned (sign-magnitude)
- Exponent is a signed integer. What method do we use?

IEEE 754

- A standard for representing FP numbers in computers
  - Single precision (32 bits): 8-bit exponent, 23-bit significand
  - Double precision (64 bits): 11-bit exponent, 52-bit significand



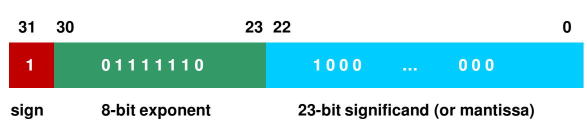
- Leading "1" in significand is implicit (why?)
- Exponent is a signed number **in biased format**
  - "Biased" format – for easier sorting of FP numbers
  - All 0's is the smallest, all 1's is the largest
  - Bias of 127 for SP and 1023 for DP
- Hence, to obtain the actual value of a representation
  - $(-1)^{sign} \times$ (1#"."#significand) $\times 2^{exponent}$: here "#" is concatenation
  - exponent is a **biased number (see next slide)**
  - exponent effectively shifts the "decimal point" in represented value

Biased representation

- Yet another binary number representation
  - Signed number allowed
- **000…000** is the smallest number, **111 … 111** is largest number!
- To get the real value, subtract a pre-determined "bias" from the unsigned evaluation of the bit pattern
- In other words, *representation = value + bias*
- Bias for the "exponent" field in IEEE 754
  - 127 (SP), 1023 (DP)
- E.g., suppose exponent field = 01111101b = 125d
  - b/c we added the bias, we must subtract it to get decimal value
  - thus, exponent in decimal is really: 125d - 127d = -2d
  - what's the decimal value for 10000111b=135d? (135d - 127d = 8d)

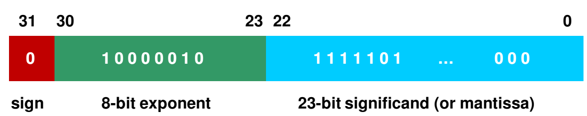
IEEE 754 example

- -0.75_{ten}
  - Same as $-3/4 = -3/2^2$
  - In binary, $-11_{two}/2^2_{ten}$ or -0.11_{two}
  - In a normalized form, it's $-1.1_{two} \times 2^{-1}$
- In IEEE 754
  - Sign bit is 1 – number is negative!
  - Significand is 0.1 – the leading 1 is implicit!
  - Exponent is -1; (-1 + 127 = **126** in biased representation)
    - 126 is in exponent field, so "decimal exponent" value is 126 - 127 = -1



IEEE 754 example #2

- Let's try 15.625
  - integral part is 15d=1111b
  - fractional part is:
    - 0.625 × 2 = **1**.25, generate the leading "1", carry down ".25"
    - 0.25 × 2 = **0**.5, generate the leading "0", carry down ".5"
    - 0.5 × 2 = **1**.0, generate the leading "1", nothing remains ".0"
    - answer is 0.101b
  - now, we have 1111.101b
  - normalize the result by shifting right by 3 positions
    - 1.111101 × 2^3, thus, significand = 0.111101, exponent is 3
  - get the exponent bias: 3 + 127 = 130d = 10000010b



IEEE 754 summary

Single Precision		Double Precision		Represented Object
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	non-zero	0	non-zero	+/- denormalized number
1~254	anything	1~2046	anything	+/- floating-point numbers
255	0	2047	0	+/- infinity
255	non-zero	2047	non-zero	NaN (Not a Number)

Denormal number

- Smallest normal (leading **1 is implicit**): 1.0×2^{Emin}
- Below, use "denormal" (no leading 1): $0.f \times 2^{Emin}$

- $e = E_{min}-1$, f != 0
- #00000000#######################

e.g., smallest without denormal

$1.00000000000000000000000 \times 2^{Emin}$

smallest with denormal

$0.00000000000000000000001 \times 2^{-Emin-23}$

NaN

- Not a Number
- Result of illegal computation
  - 0/0, infinity/infinity, infinity – infinity, …
  - Any computation involving a NaN

- $e = E_{max}+1$, f != 0
- #11111111#######################
- Many NaN's

Values represented with IEEE 754

Type	Sign	Exponent	Significand	Value
Zero	0	0000 0000	000 0000 0000 0000 0000 0000	0.0
One	0	0111 1111	000 0000 0000 0000 0000 0000	1.0
Minus One	1	0111 1111	000 0000 0000 0000 0000 0000	−1.0
Smallest denormalized number	*	0000 0000	000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	0000 0000	100 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	0000 0000	111 1111 1111 1111 1111 1111	$\pm(1-2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	0000 0001	000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx 1.18 \times 10^{-38}$
Largest normalized number	*	1111 1110	111 1111 1111 1111 1111 1111	$\pm(1-2^{-24}) \times 2^{128} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	1111 1111	000 0000 0000 0000 0000 0000	$+\infty$
Negative infinity	1	1111 1111	000 0000 0000 0000 0000 0000	$-\infty$
Not a number	*	1111 1111	non zero	NaN
* Sign bit can be either 0 or 1 .				

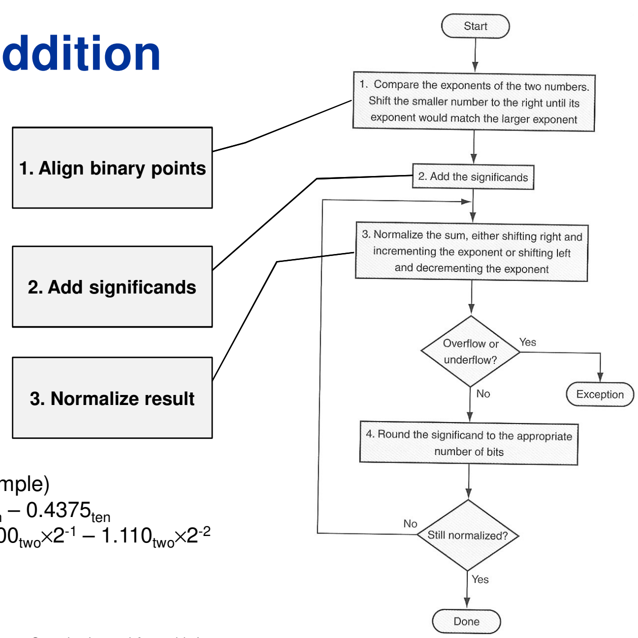
FP arithmetic operations

- We want to support four arithmetic functions (+, –, ×, ∕)
- (+, –): Must equalize exponents first. Why?
- (×, ∕): Multiply/divide significand, add/subtract exponents.
- Use "rounding" when result is not accurate
- Exception conditions
  - E.g., Overflow, underflow (what is underflow?)
- Error conditions
  - E.g., divide-by-zero

Overflow and underflow

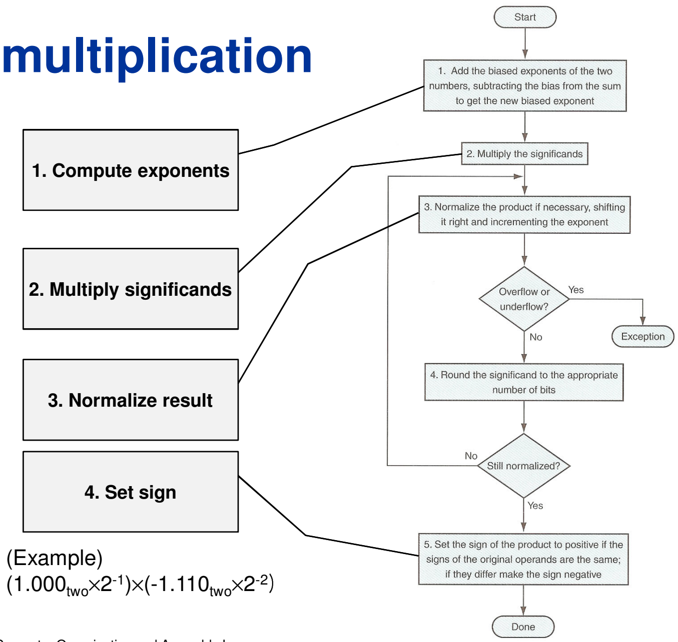
- Overflow
  - The exponent is too large to fit in the exponent field
- Underflow
  - The exponent is too small to fit in the exponent field

FP addition



(Example)
$0.5_{ten} - 0.4375_{ten}$
$=1.000_{two} \times 2^{-1} - 1.110_{two} \times 2^{-2}$

FP multiplication



(Example)
$(1.000_{two} \times 2^{-1}) \times (-1.110_{two} \times 2^{-2})$