Imperial College London

# 113: Architecture

Spring 2018

*Lecture:* Floating Point Numbers and Arithmetics

**Instructor:** Dr. Jana Giceva

# Quick note for tomorrow!

- Lecture on Memory Hierarchies

- Tutorial on the material we cover today (Floating Point Numbers)

- Lab: hands-on assignment for x86
  - Slides for last week's material on x86 are online (procedures and data structures)
  - During the first half of tomorrow's tutorial – flipped classroom.
    Ask questions about the material from last week – you'll need it for the lab.
  - Set-up for the lab assignment from 2-4pm with me and helpers in the lab rooms (202; 206; 219).

# Today: Floating Point

- **Background: Fractional binary numbers**
- IEEE floating point standard: Definition
- Floating point ranges
- Rounding, addition, multiplication
- Summary

# Fractional binary numbers

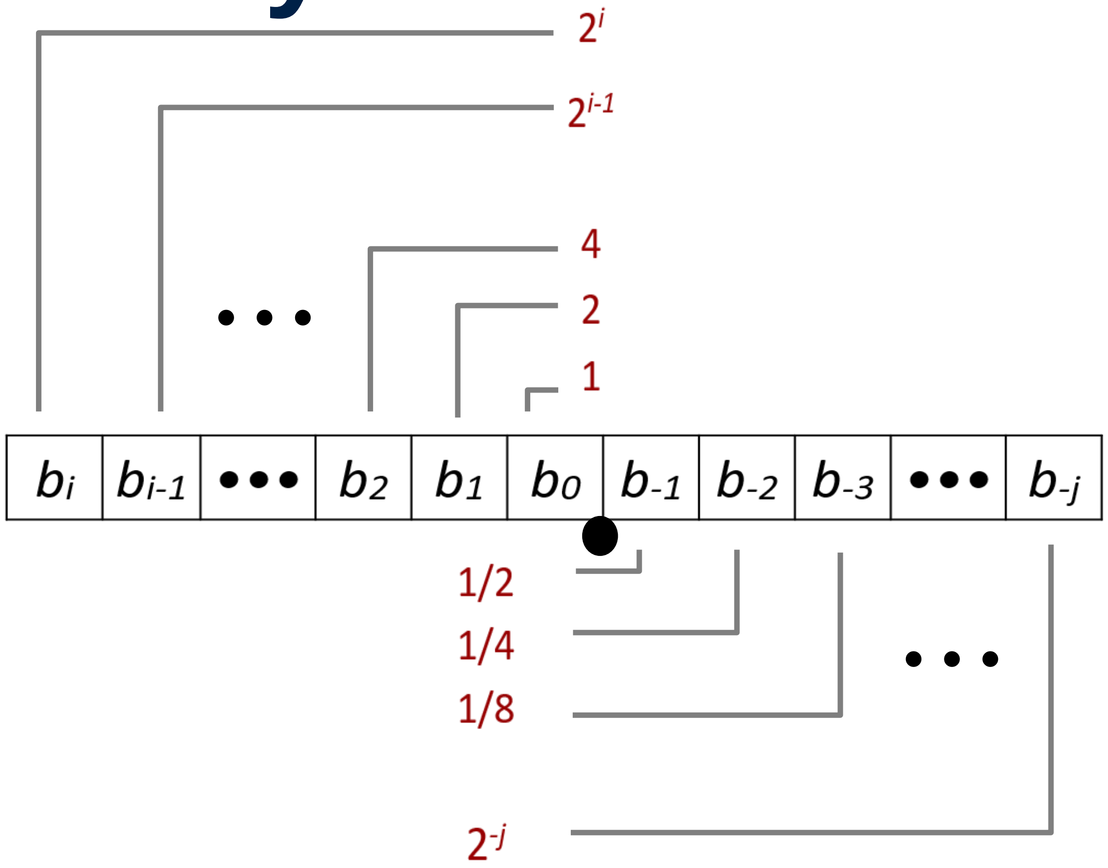
- **What is $\mathbf{1011.101_2}$?**

$$1\ 0\ 1\ 1\ .\ 1\ 0\ 1_2$$

$$\begin{array}{ccccccc} 8 & 4 & 2 & 1 & {}^1/_2 & {}^1/_4 & {}^1/_8 \\ 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} \end{array}$$

$$8 + 2 + 1 + {}^1/_2 + {}^1/_8 = 11.625_{10}$$

# Fractional Binary Numbers




- **Representation**
  - Bits to right of “binary point” represented fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

# Fractional Binary Numbers: Examples

- **Value** **Representation**

| Value | Representation |
|---|---|
| $5\ 3/4$ | $101.11_2$ |
| $2\ 7/8$ | $10.111_2$ |
| $1\ 7/16$ | $1.0111_2$ |

- **Observations:**
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0

# Representable Numbers

- **Limitation #1**
  - Can only exactly represent numbers of the form $\boldsymbol{x/2^k}$
  - Other rational numbers have repeating bit representations
  - Value Representation

| Value | Representation |
|---|---|
| $1/3$ | $0.0101010101[01]\ldots_2$ |
| $1/5$ | $0.001100110011[0011]\ldots_2$ |
| $1/10$ | $0.0001100110011[0011]\ldots_2$ |

- **Limitation #2**
  - Just one setting of binary point within the $w$ bits
    - Limited range of numbers (very small values? Very large?)

# Today: Floating Point

- Background: Fractional binary numbers
- **IEEE floating point standard: Definition**
- Floating point ranges
- Rounding, addition, multiplication
- Floating point in C
- Summary

# IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
  - Make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Supported by all major CPUs

- **Driven by numerical concerns**
  - **Scientists** want them to be as **real** as possible
  - **Engineers** want them to be **easy to implement** and **fast**
  - In the end: scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but …
    - Hard to make fast in hardware
    - **Can be order of magnitude slower than integer operations**

# Floating Point Representation

- **Numerical Form:**

$$\mathbf{V_{10}} = (-1)^{\boldsymbol{s}} * \boldsymbol{M} * \mathbf{2}^{\boldsymbol{E}}$$

  - Sign bit ***s*** determines whether number is negative or positive
  - Significand (mantissa) ***M*** normally a fractional value in range **[1.0,2.0)**.
  - Exponent ***E*** weights value by a (possibly negative) power of two

- **Encoding:**
  - **`MSB S`** is sign bit ***s***
  - **`exp`** field encodes ***E*** (but is ***not equal*** to **E**)
  - **`frac`** field encodes **M** (but is ***not equal*** to **M**)

| | | |
|---|---|---|
| s | exp | frac |

# Precision options

- **IEEE 754 Single precision (32 bits)**

| s | exp | frac |
|---|---|---|
| 1 | 8-bits | 23-bits |

Finite representation means that not all values can be represented exactly. Some will be approximated!

- **IEEE 754 Double precision (64 bits)**

| s | exp | frac |
|---|---|---|
| 1 | 11-bits | 52-bits |

- **Intel Extended Precision (80 bits)**

| s | exp | frac |
|---|---|---|
| 1 | 15-bits | 63 or 64-bits |

- **IEEE 754 Quadruple Precision (128 bits)**

| s | exp | frac |
|---|---|---|
| 1 | 15-bits | 112-bits |

# "Normalized" Values

$V = (-1)^{s} * M * 2^{E}$
$E = Exp - Bias$

- **When: exp ≠ 000…0 and exp ≠ 111…1**
  - Reserved for special values. We'll cover them later.
- **Exponent coded as a biased value: *E* = *Exp – Bias***
  - *Exp*: unsigned value **`exp`** ranging from 1 to $2^k - 2$ (k==#bits in **`exp`**)
  - *Bias* = $2^{k-1} - 1$
    - Single precision: 127 (*Exp*: 1…254, *E*:-126…127)
    - Double precision: 1023 (*Exp*: 1…2046, *E*:-1022…1023)
  - Enables negative values for E, for representing very small values!
- **Significand coded with implied leading 1: M = $1.\mathrm{xxx}\ldots\mathrm{x}_2$**
  - $\mathrm{xxx}\ldots\mathrm{x}$: bits of **`frac`**
  - Minimum when **`frac`** = 000…0 (M=1.0)
  - Maximum when **`frac`** = 111…1 (M=2.0-ε)
  - Get extra leading bit for "free"

# Normalized Encoding Example

$$V = (-1)^s * M * 2^E$$
$$E = Exp - Bias$$

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$ (normalized form)

- **Significand**

$$M = 1.1101101101101_2$$
$$\texttt{frac} = 11011011011010000000000_2$$

- **Exponent:** E = `exp` – Bias, so `exp` = E + Bias

$$E = 13$$
$$Bias = 127$$
$$Exp = 13 + 127 = 140 = 10001100_2$$

| 0 | 10001100 | 11011011011010000000000 |
|---|---|---|
| `s` | `exp` | `frac` |

# “Denormalized” Values

$V = (-1)^s * M * 2^E$
$E = 1 - Bias$

- **Conditions:** exp = 000…0

- **Exponent value:** ***E*** **= 1 – *Bias*** **(instead of** ***E*** **= 0 – *Bias*)**
- **Significand coded with implied leading 0:** $M = 0.\mathrm{xxx}\ldots\mathrm{x}_2$
  - xxx…x: bits of `frac`
- **Cases:**
  - `exp` = 000…0, `frac` = 000…0
    - Represents zero value
  - `exp` = 000…0, `frac` ≠ 000…0
    - Numbers closest to 0.0
    - Equi-spaced

# Interesting Numbers

| Description | `exp` | `frac` | Numerical value |
|---|---|---|---|
| **Zero** | `00…00` | `00…00` | $0.0$ |
| **Smallest possible denormalized**<br>• Single $\approx 1.4 \times 10^{-45}$<br>• Double $\approx 4.9 \times 10^{-324}$ | `00…00` | `00…01` | $2^{-\{23,52\}} \times 2^{-\{126,1022\}}$ |
| **Largest possible denormalized**<br>• Single $\approx 1.18 \times 10^{-38}$<br>• Double $\approx 2.2 \times 10^{-308}$ | `00…00` | `11…11` | $(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$ |
| **Smallest possible normalized**<br>• Just larger than largest denormalized | `00…01` | `00…00` | |
| **One** | | | $1.0$ |
| **Largest normalized**<br>• Single $\approx 3.4 \times 10^{38}$<br>• Double $\approx 1.8 \times 10^{308}$ | `11…10` | `11…11` | $(2.0 - \varepsilon) \times 2^{\{127,1023\}}$ |

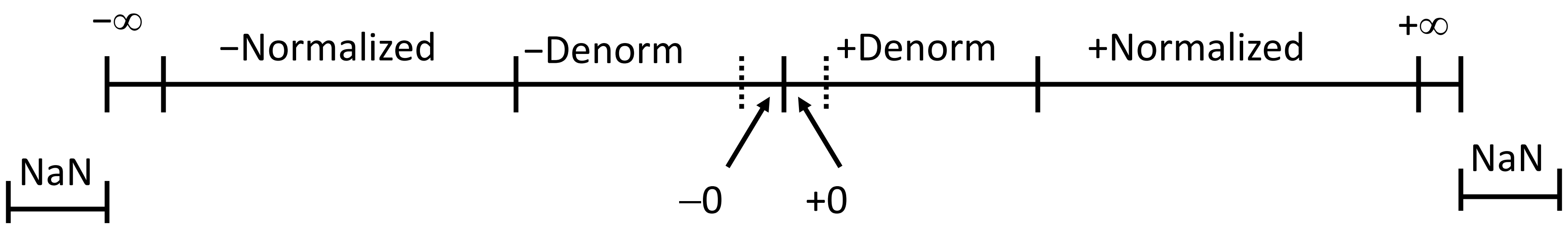
# Special Values

- **Condition: `exp` =** 111…1

- **Case: `exp` =** 111…1**, `frac` =** 000…0
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = -1.0/-0.0 = +∞, 1.0/-0.0 = -∞

- **Case: `exp` =** 111…1**, `frac` ≠** 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric values can be determined
  - E.g., sqrt(-1), ∞-∞, ∞x0

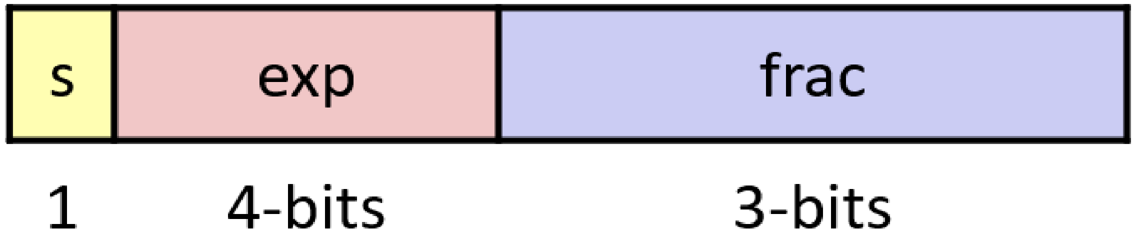
# Visualization: Floating Point Encodings



# Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- **Floating point ranges**
- Rounding, addition, multiplication
- Summary

# Tiny Floating Point Example




- **8-bit Floating Point Representation**
  - the sign bit is the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the `frac`

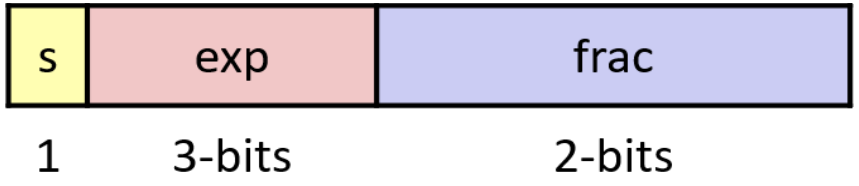
- **Same general form as the IEEE format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

# Dynamic range (positive only)

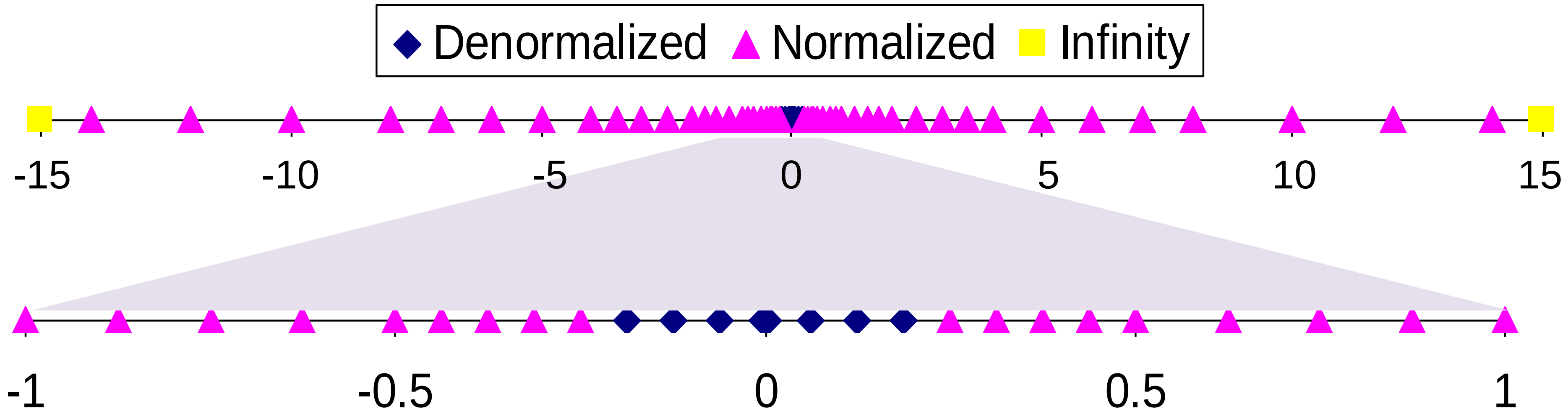
| | s | exp | frac | E | Value | |
|---|---|---|---|---|---|---|
| | 0 | 0000 | 000 | -6 | 0 | |
| | 0 | 0000 | 001 | -6 | 1/8*1/64 = 1/512 | **closest to zero** |
| **Denormalized Numbers** | 0 | 0000 | 010 | -6 | 2/8*1/64 = 2/512 | |
| | … | | | | | |
| | 0 | 0000 | 110 | -6 | 6/8*1/64 = 6/512 | |
| | 0 | 0000 | 111 | -6 | 7/8*1/64 = 7/512 | **largest denormalized** |
| | 0 | 0001 | 000 | -6 | 8/8*1/64 = 8/512 | **smallest normalized** |
| | 0 | 0001 | 001 | -6 | 9/8*1/64 = 9/512 | |
| | … | | | | | |
| | 0 | 0110 | 110 | -1 | 14/8*1/2 = 14/16 | |
| | 0 | 0110 | 111 | -1 | 15/8*1/2 = 15/16 | **closest to 1 below** |
| **Normalized Numbers** | 0 | 0111 | 000 | 0 | 8/8*1 = 1 | |
| | 0 | 0111 | 001 | 0 | 9/8*1 = 9/8 | **closest to 1 above** |
| | 0 | 0111 | 010 | 0 | 10/8*1 = 10/8 | |
| | … | | | | | |
| | 0 | 1110 | 110 | 7 | 14/8*128 = 224 | |
| | 0 | 1110 | 111 | 7 | 15/8*128 = 240 | **largest normalized** |
| | 0 | 1111 | 000 | n/a | inf | |

# Distribution of Values

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1} - 1 = 3$




- **Notice how the distribution gets denser toward zero.**



# Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Floating point ranges
- **Rounding, addition, multiplication**
- Floating point in C
- Summary

# Floating point operations: basic idea

$$x +_f y = \mathrm{Round}(x + y)$$
$$x \times_f y = \mathrm{Round}(x \times y)$$

- **First compute exact result**
- **Make it fit into desired precision**
  - Possibly overflow if exponent too large
  - Possibly round to fit into **`frac`**

# Rounding

**Note:** Nearest-even in this example rounds to the closest whole number, and in the case of equal distance -> rounds to the closest even number!

- **Rounding Modes (illustrate with £ rounding)**

| | £1.40 | £1.60 | £1.50 | £2.50 | -£1.50 |
|---|---|---|---|---|---|
| Towards zero | £1 | £1 | £1 | £2 | -£1 |
| Round down ($-\infty$) | £1 | £1 | £1 | £2 | -£2 |
| Round up ($+\infty$) | £2 | £2 | £2 | £3 | -£1 |
| Nearest Even (default) | £1 | £2 | £2 | £2 | -£2 |

- **What could happen if we are repeatedly rounding the results?**
  - If we always round in one direction, we can introduce a statistical bias.
- **Round-to-even avoids this bias by rounding up about half the time, and rounding-down about half the time → IEEE default mode.**

# Creating a floating point number

| s | exp | frac |
|---|---|---|
| 1 | 4-bits | 3-bits |

- **Steps**
  1. Normalize to have leading 1
  2. Round to fit within fraction
  3. Post-normalize to deal with effects of rounding
- **Case study**
  - Convert 8-bit unsigned numbers to tiny floating point format

| Value | Binary | Fraction | Exponent |
|---:|---:|---:|---:|
| 128 | 10000000 | | |
| 15 | 00001101 | | |
| 17 | 00010001 | | |
| 19 | 00010011 | | |
| 138 | 10001010 | | |
| 63 | 00111111 | | |

# Normalize

| s | exp | frac |
|---|---|---|
| 1 | 4-bits | 3-bits |

- **Requirement**
  - Set binary point so that numbers of form 1.xxxx
  - Adjust all to have leading one
  - Decrement exponent as shift left

| Value | Binary | Fraction | Exponent |
|---:|---:|---:|---:|
| 128 | 10000000 | 1.0000000 | 7 |
| 15 | 00001101 | 1.1010000 | 3 |
| 17 | 00010001 | 1.0001000 | 4 |
| 19 | 00010011 | 1.0011000 | 4 |
| 138 | 10001010 | 1.0001010 | 7 |
| 63 | 00111111 | 1.1111100 | 5 |

# Rounding

**1.BBGRXXX**

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **Round up conditions**
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

| Value | Fraction | GRS | Inc? | Rounded |
|---:|---:|---:|---:|---:|
| 128 | 1.0000000 | 000 | N | 1.000 |
| 15 | 1.1010000 | 100 | N | 1.101 |
| 17 | 1.0001000 | 010 | N | 1.000 |
| 19 | 1.0011000 | 110 | Y | 1.010 |
| 138 | 1.0001010 | 011 | Y | 1.001 |
| 63 | 1.1111100 | 111 | Y | 10.000 |

# Post-normalize

- **Issue**
  - Rounding may have caused overflow
  - Handle by shifting right once and incrementing exponent

| Value | Rounded | Exponent | Adjusted | Result |
|---:|---:|---:|---:|---:|
| 128 | 1.000 | 7 | | 128 |
| 15 | 1.101 | 3 | | 15 |
| 17 | 1.000 | 4 | | 16 |
| 19 | 1.010 | 4 | | 20 |
| 138 | 1.001 | 7 | | 134 |
| 63 | 10.000 | 5 | 1.000/6 | 64 |

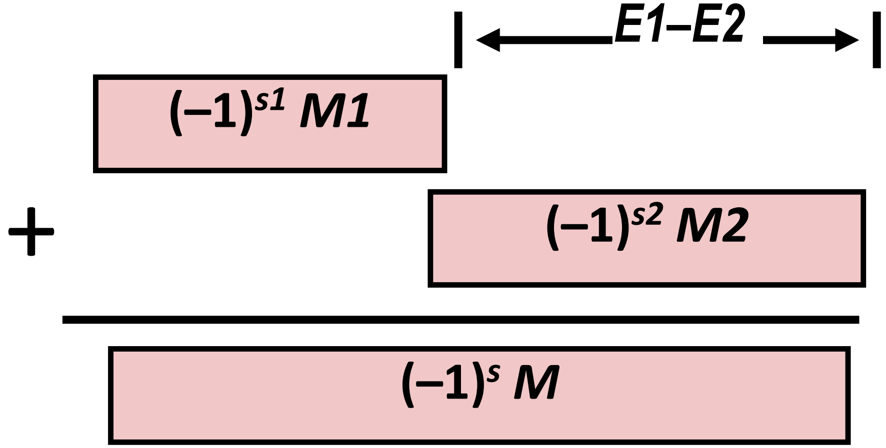
# Floating Point Multiplication

$$(-1)^{S1} M_1 2^{E1} \times (-1)^{S2} M_2 2^{E2}$$

- **Exact Result:** $(-1)^s M 2^E$
  - Sign s: $s_1$^$s_2$
  - Significand *M*: $M_1 * M_2$
  - Exponent *E*: $E_1 + E_2$
- **Fixing:**
  - If $M \geq 2$ shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit **`frac`** precision
- **Implementation:**
  - Biggest chore is multiplying significands

# Floating Point Addition

$(-1)^{S1} M_1 2^{E1} + (-1)^{S2} M_2 2^{E2}$




- **Assume $E_1 > E_2$**
- **Exact Result:** $(-1)^s M 2^E$
  - Sign *s*, significand *M*: result of signed align and add
  - Exponent *E* : $E_1$
- **Fixing:**
  - If $M \geq 2$, shift *M* right, increment *E*
  - If $M < 1$, shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* is out of range
  - Round *M* to fit `frac` precision

# Mathematical Properties of FP Operations

- **Exponent overflow yields +inf or –inf**

- **Floats with value +inf, -inf, and NaN can be used in operations**
  - Result usually still, +inf, -inf, or NaN; sometimes intuitive, sometimes not

- **Floating point ops do not work like real math, due to *rounding*!**
  - ***Not associative***: $(3.14 + 1\mathrm{e}100) - 1\mathrm{e}100 \neq 3.14 + (1.\mathrm{e}100 - 1\mathrm{e}100)$
  - ***Not distributive***: $100 * (0.1 + 0.2) \neq 100 * 0.1 + 100 * 0.2$

    30.000000000000003553 30

  - ***Not cumulative***: repeatedly adding a very small number to a large one may do nothing

# Number Representation Really Matters

- **1991: Patriot missile targeting error**
  - clock skew due to conversion from integer to floating point
- **1996: Ariane 5 rocket exploded ($1 billion)**
  - overflow converting 64-bit floating point to 16-bit integer
- **2000: Y2K problem**
  - limited (decimal) representation: overflow, wrap-around
- **2038: Unix epoch rollover**
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to Tmin in 2038
- **Other related bugs:**
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

# Summary

- **As with integers, floats suffer from the fixed number of bits available to represent them**
  - Can get overflow and underflow, just like `ints`
  - Some "simple fractions" have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike `ints`
    - "Every operation gets a slightly wrong result"
- **Mathematically equivalent ways of writing an expression may compute different results**
  - Violates associativity and distributivity
- ***Never* test floating point values for equality!**
- ***Careful* when converting between `ints` and `floats`!**

# Further reading

- "What Every Programmer Should Know About Floating-Point Arithmetic" – David Goldberg [http://www.itu.dk/~sestoft/bachelor/IEEE754_article.pdf]

- Chapter 2.4 in the Book "Computer Systems: A Programmer's Perspective".

- **IEEE 754 standards:**
  - Original document from 1985: http://ieeexplore.ieee.org/document/30711/
  - Revised version from 2008: http://ieeexplore.ieee.org/document/4610935/
  - ISO standard: https://www.iso.org/standard/57469.html