# Floating Point Representation

- **Numerical Form:**

$$(-1)^s \, M \; 2^E$$

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

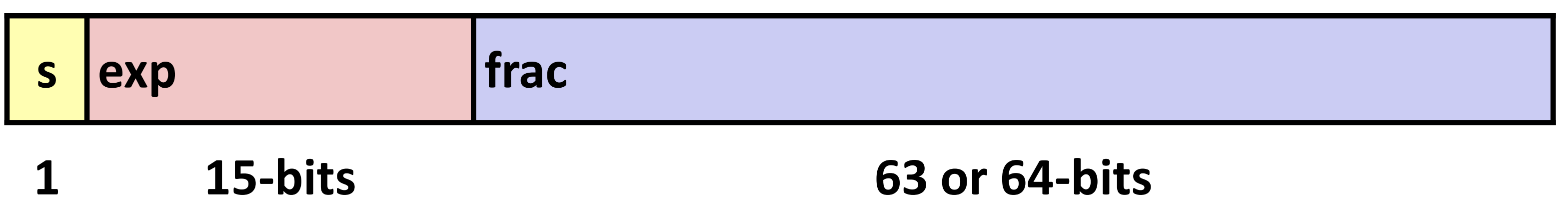
- **Encoding**
  - MSB s is sign bit ***s***
  - **exp** field encodes ***E*** (but is not equal to E)
  - **frac** field encodes ***M*** (but is not equal to M)

# Precision options

- **Single precision: 32 bits**


- **Double precision: 64 bits**


- **Extended precision: 80 bits (Intel only)**



# "Normalized" Values

$v = (-1)^s\, M\, 2^E$

- **When: exp ≠ 000…0 and exp ≠ 111…1**

- **Exponent coded as a *biased* value: *E* = *Exp* – *Bias***
  - *Exp*: unsigned value of exp field
  - *Bias* = $2^{k-1} - 1$, where *k* is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

- **Significand coded with implied leading 1: *M* =** $1.xxx…x_2$
  - xxx…x: bits of frac field
  - Minimum when frac = 000…0 (M = 1.0)
  - Maximum when frac = 111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

# Bias Notes

- **Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder.**
  - To solve this problem the exponent is biased to put it within an unsigned range suitable for comparison.
  - By arranging the fields so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits, the resulting value will be ordered properly, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.
- **When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent.**

# Significand Notes

- **Represents the fraction, or precision bits of the number.**
- **It is composed of an implicit (i.e., hidden) leading bit and the fraction bits.**
- **In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form.**
  - This basically puts the radix point after the first non-zero digit
  - Nice optimization available in base two, since the only possible non-zero digit is 1.
  - Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly.
  - As a result, the mantissa/significand has effectively 24 bits of resolution, by way of 23 fraction bits.

# Normalized Encoding Example

$$v = (-1)^s \, M \, 2^E$$
$$E = Exp - Bias$$

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

- **Significand**

  $M$ = `1.`$\underline{\texttt{1101101101101}}_2$

  `frac=` $\underline{\texttt{1101101101101}}\texttt{0000000000}_2$

- **Exponent**

  $E$ = 13

  $Bias$ = 127

  $Exp$ = 140 = `10001100`$_2$

- **Result:**

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

# Normalized Encoding Example 2

- **Value: π, rounded to 24 bits of precision**
  - sign: 0

- **Significand**

| | | |
|---|---|---|
| *s* | = | `11.0010010000111111011011` (including hidden bit) |
| *M* | = | $1.10010010000111111011011_2$ (x $2^1$) |
| **`frac`** | = | $10010010000111111011011_2$ |

- **Exponent**

| | | |
|---|---|---|
| *E* | = | 1 |
| *Bias* | = | 127 |
| *Exp* | = | 128 = $10000000_2$ |

- **Result:**

| 0 | 10000000 | 10010010000111111011011 |
|---|---|---|
| **s** | **exp** | **frac** |

# Denormalized Values

- **Also called denormal or subnormal numbers**
- **Values that are very close to zero**
  - Fill the “underflow” gap around zero
  - Gradual underflow = numeric values are spaced evenly near 0.0
- **Any number with magnitude smaller than the smallest normal number**
  - When the exponent field is all zeros
  - E = 1-bias
  - Significand M = f without implied leading 1
  - h = 0 (hidden bit)
- **Representation of numeric value 0**
  - -0.0 and +0.0 are considered different in some ways and the same in others

# Denormalized Values

- In a normal floating point value there are no leading zeros in the significand, instead leading zeros are moved to the exponent.

- e.g., 0.0123 would be written as $1.23 * 10^{-2}$

- Denormal numbers are numbers where this representation would result in an exponent that is too small (the exponent usually having a limited range). Such numbers are represented using leading zeros in the significand.

# Denormalized Values

$v = (-1)^s M\ 2^E$
$E = 1 - Bias$

- **Condition:** exp = 000…0

- **Exponent value: $E$ = 1 – Bias (instead of $E$ = 0 – *Bias*)**
- **Significand coded with implied leading 0: $M$ = 0.xxx…x$_2$**
  - `xxx…x`: bits of `frac`
- **Cases**
  - `exp` = 000…0, `frac` = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - `exp` = 000…0, `frac` ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

# 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., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- **Case: `exp` = `111…1`, `frac` ≠ `000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

# Interesting Numbers `{single,double}`

| *Description* | *exp* | *frac* | *Numeric Value* |
|---|---|---|---|
| **Zero** | **00…00** | **00…00** | **0.0** |
| **Smallest Pos. Denorm.** | **00…00** | **00…01** | $2^{-\{23,52\}} \times 2^{-\{126,1022\}}$ |
| Single ≈ $1.4 \times 10^{-45}$ | | | |
| Double ≈ $4.9 \times 10^{-324}$ | | | |
| **Largest Denormalized** | **00…00** | **11…11** | $(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$ |
| Single ≈ $1.18 \times 10^{-38}$ | | | |
| Double ≈ $2.2 \times 10^{-308}$ | | | |
| **Smallest Pos. Normalized** | **00…01** | **00…00** | $1.0 \times 2^{-\{126,1022\}}$ |
| Just larger than largest denormalized | | | |
| **One** | **01…11** | **00…00** | **1.0** |
| **Largest Normalized** | **11…10** | **11…11** | $(2.0 - \varepsilon) \times 2^{\{127,1023\}}$ |
| Single ≈ $3.4 \times 10^{38}$ | | | |
| Double ≈ $1.8 \times 10^{308}$ | | | |

# Dynamic Range (Positive Only)

$v = (-1)^s M\, 2^E$

*n: E = Exp – Bias*

*d: E = 1 – Bias*

| | s | exp | frac | E | Value | |
|---|---|---|---|---|---|---|
| Denormalized numbers | 0 | 0000 | 000 | -6 | 0 | |
| | 0 | 0000 | 001 | -6 | 1/8*1/64 = 1/512 | closest to zero |
| | 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 denorm |
| Normalized numbers | 0 | 0001 | 000 | -6 | 8/8*1/64 = 8/512 | smallest norm |
| | 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 |
| | 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 norm |
| | 0 | 1111 | 000 | n/a | inf | |

# Rounding

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

| | **$1.40** | **$1.60** | **$1.50** | **$2.50** | **–$1.50** |
|---|---|---|---|---|---|
| Towards zero | $1 | $1 | $1 | $2 | –$1 |
| Round down (−∞) | $1 | $1 | $1 | $2 | –$2 |
| Round up (+∞) | $2 | $2 | $2 | $3 | –$1 |
| Nearest Even (default) | $1 | $2 | $2 | $2 | –$2 |

# Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- **Applying to Other Decimal Places / Bit Positions**
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

| | | |
|---|---|---|
| 7.8949999 | 7.89 | (Less than half way) |
| 7.8950001 | 7.90 | (Greater than half way) |
| 7.8950000 | 7.90 | (Half way—round up) |
| 7.8850000 | 7.88 | (Half way—round down) |

# Rounding Binary Numbers

- **Binary Fractional Numbers**
  - "Even" when least significant bit is `0`
  - "Half way" when bits to right of rounding position = `100`…$_2$

- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

| Value | Binary | Rounded | Action | Rounded Value |
|---|---|---|---|---|
| 2 3/32 | $10.00011_2$ | $10.00_2$ | (<1/2—down) | 2 |
| 2 3/16 | $10.00110_2$ | $10.01_2$ | (>1/2—up) | 2 1/4 |
| 2 7/8 | $10.11100_2$ | $11.00_2$ | ( 1/2—up) | 3 |
| 2 5/8 | $10.10100_2$ | $10.10_2$ | ( 1/2—down) | 2 1/2 |

# Rounding

**1.BBGRXXX**

**Guard bit: LSB of result**

**Sticky bit: OR of remaining bits**

**Round bit: 1st bit removed**

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

| *Value* | *Fraction* | *GRS* | *Incr?* | *Rounded* |
|---|---|---|---|---|
| 128 | 1.0000000 | 000 | N | 1.000 |
| 13 | 1.1010000 | 100 | N | 1.101 |
| 17 | 1.0001000 | 010 | N | 1.000 |
| 19 | 1.0011000 | 110 | Y | 1.010 |
| 142 | 1.0001110 | 011 | Y | 1.001 |
| 63 | 1.1111100 | 111 | Y | 10.000 |

# FP Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*: $s1$ ^ $s2$
  - Significand *M*: $M1 \times M2$
  - Exponent *E*: $E1 + E2$

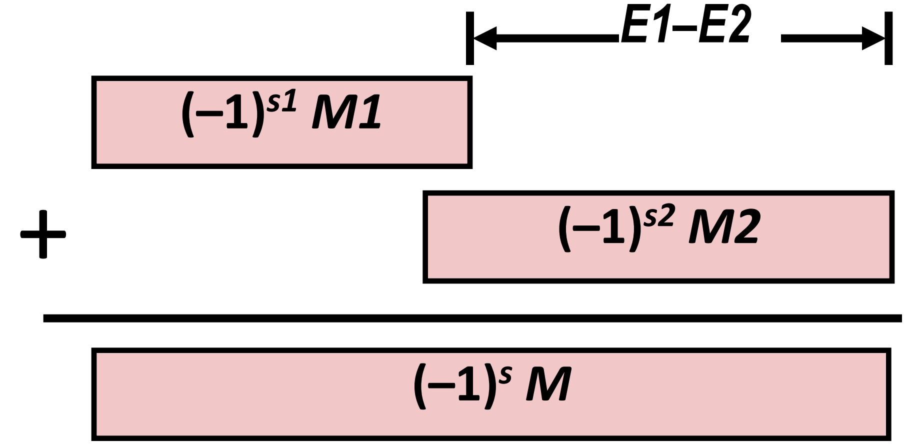
- **Fixing**
  - If $M \geq 2$, shift $M$ right, increment $E$
  - If $E$ out of range, overflow
  - Round $M$ to fit `frac` precision

- **Implementation**
  - Biggest chore is multiplying significands

# Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume $E1 > E2$

Get binary points lined up




- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*

- **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* out of range
  - Round *M* to fit `frac` precision

# Floating Point in C

- **C Guarantees Two Levels**
  - **`float`** single precision
  - **`double`** double precision

- **Conversions/Casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`int`** → **`float`**
    - Cannot overflow; will round according to rounding mode
  - **`int/float`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`float/double`** → **`int`**
    - Truncates fractional part; like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to Tmin
  - **`double`** → **`float`**
    - Can overflow (range smaller); may be rounded (precision smaller)

# Floating Point Puzzles

- **For each of the following C expressions, either:**
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NaN

```
F • x == (int)(float) x
T • x == (int)(double) x
T • f == (float)(double) f
F • d == (double)(float) d
T • f == -(-f);
F • 2/3 == 2/3.0
T • d < 0.0      ⇒  ((d*2) < 0.0)
T • d > f        ⇒  -f > -d
T • d * d >= 0.0
F • (d+f)-d == f
```