# Floating Point Numbers

Computer Systems Organization (Spring 2016)
CSCI-UA 201, Section 2

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

# Fractions in Binary



## Fractional binary numbers

What is $1011.101_2$?

- we use the same idea that we use when interpreting the decimal numbers, except here we multiply by powers of 2, not 10
- the above number is

$1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 +$ **$1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3}$**

$= 8 + 2 + 1 + ½ + ⅛ = 11.625$

**Simple enough?**

**DNHI**: Try to convert the following numbers to their binary representation $5\ ^1/_{16}$ , $2\ ⅞$ , $15\ ¾$ . Now, try $^1/_{10}$ and and see how that goes.
Convert the following binary numbers to their decimal representations: 0.1101, 101.010, 10.101.

## Not good enough

That way of representing floating point numbers is simple, but has many limitations.

- Only numbers that can be written as the sum of powers of 2 can be represented exactly, other numbers have repeating bit representation (this is the same problem as trying to represent ⅓ in decimal as 0.3333333....).
  - ⅓ is 0.010101010101 …
  - ⅕ is 0.01100110011 …
  - 1/10 is 0.001100110011 …
- Just one possible location for the binary point. This limits how many bits can be used for the fractional part and the whole number part. We can either represent very large numbers well or very small numbers well, but not both.

Up until 1985 floating point numbers were computer scientist nightmare because everybody was using different standards that dealt with the above problems.

**But do not forget that notation just yet - we will use it as part of the better notation.**

# 1985: IEEE Standard 754



## IEEE Floating Point

- Established 1985
- Provides uniform standard for floating point arithmetic used by most (if not all) of current CPUs
- Standards for rounding, overflow, underflow
- Concerns for numerical accuracy were more important than fast hardware implementation ⇒ not very good hardware performance



## Floating Point Representation

- Numerical Form:

  $$(-1)^s\ M * 2^E$$

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

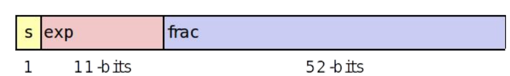
- Encoding
  - the most significant bit **s** is the sign bit **s**
  - **exp** field encodes **E** (but is not equal to E)
  - **frac** field encodes **M** (but is not equal to M)



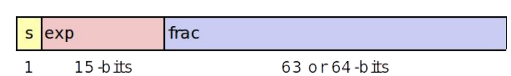

## Using Different Number of Bytes

**Single precision: 32 bits**


**Double precision: 64 bits**




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



# Interpreting Values of IEEE 754



## **Normalized** Values

- Condition: **exp** ≠ 000…0 and **exp** ≠ 111…1
- Exponent is: $\mathbf{E = exp - \underbrace{(2^{k-1} - 1)}_{bias}}$, k is the # of exponent bits
  - Single precision: $2^{k-1} - 1 = 127$, exp = 1…254 ⇒ E = -126…127
  - Double precision : $2^{k-1} - 1 = 1023$, exp = 1…2046 ⇒ E = -1022…1023

  (once we know the number of bits in exp, we can figure out the bias)

- Significand has implied leading 1: $M = 1.xxx\ldots x_2$
  - xxx…x – bits of frac
  - Smallest value when all bits are zero: 000…0, M = 1.0
  - Largest value when all bits are one: 111…1, M = 2.0- ε
  - By assuming the leading bit is 1, we get "an extra bit for free"



## Normalized Values - Example

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

**$E = exp - (2^{k-1} - 1)$**

- Value: floating point number F = 15213.0 represented using single precision

  $152131.0 = 11101101101101.0_2$
  $= 1.1101101101101_2 * 2^{13}$ (same binary sequence)
- Significand
```
M    =  1.1101101101101₂
⇒ frac =   11011011011010000000000
```
- Exponent
```
E      =  13
Bias   =  127 (for single precision)
Exp    =  140₁₀  =  10001100₂
```

Result:

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



## DNHI

Perform similar conversion for the following floating point numbers. Use the single precision IEEE representation:

1024

¼

17.75

- 17.75

## Why is this encoding better?

- For single precision
  - The value of exp is in the range 0 <= exp <= 255
  - ⇒ the value of E is in the range -127 <= E <= 128
  - ⇒ we can represent fairly large numbers when using $2^{128}$ and some fairly small numbers when using $2^{-127}$
- For double precision
  - well, you get the point

But we always have the leading one in the value of significand/mantissa, so we cannot represent numbers that are reeeeeaaaaly small.

- We need to talk about what happens when `exp` is all zeroes or all ones.



## **Denormalized** Encoding

- Condition: exp = 000…0 (all zeroes)

- Exponent value: **E = 1 - bias** (instead of 0 - bias)
- Significand has implied leading 0 (not 1): **M = 0.xxx…x2**
  - xxx…x – bits of frac

Cases:

- exp = 000…0, frac = 000…0 represents zero value
  - Note that we have two distinct values for zero: +0 and -0 (Why?)
- exp = 000…0, frac ≠ 000…0 represent numbers very close to zero (all denormalized encodings represent reeeeeaaaaly small numbers)



## **Special Values** Encoding

- Condition: exp = 111…1

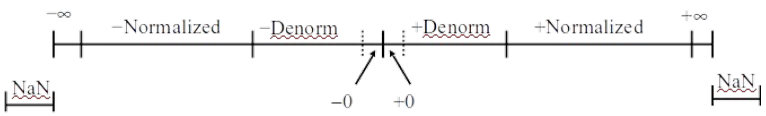
There are only two (well three cases here):

- Case 1, 2: exp = 111…1, frac = 000…0
  - Represents value ∞ (infinity)
  - Operations that overflow
  - Both positive and negative
  - Eg: 1.0/0.0 = -1.0/-0.0 = +∞, -1.0/0.0 = 1.0/-0.0 = −∞

- Case 3: exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - Eg: `sqrt(-1)`, ∞-∞, ∞*0



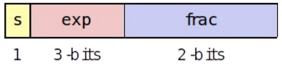
## Number Line (not to scale)



# Example



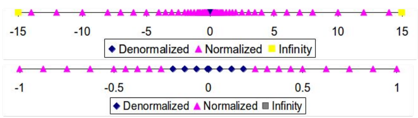
# 6-bit IEEE-like encoding




- most significant bit is the sign bit
- 3-bit encoding for exp
  - ⇒ bias $2^2-1 = 3$
- 2-bit encoding for frac
- **special values**: `011100, 011101, 011110, 011111, 111100, 111101, 111110, 111111`
- **denormalized values**: `000000, 000001, 000010, 000011, 100000, 100001, 100010, 100011`
  - smallest negative: `100011`
    M = 0.11,  E = 1 - bias = -2, val = $-0.11 * 2^{-2} = -0.1875_{10}$
  - smallest positive (larger than zero): `000001`
    M = 0.01,  E = 1 - bias = -2, val = $0.01 * 2^{-2} = 0.0625_{10}$
- normalized values: all others
  - smallest positive: `000100`
    M = 1.00,  E = 1 - bias = -2, val = $1.0 * 2^{-2} = 0.25_{10}$
  - largest positive: `011011`
    M = 1.11,  E = 6 - bias = 3, val = $1.11 * 2^3 = 14.00_{10}$

All possible 6-bit sequences:

| | | | |
|---|---|---|---|
| 000000 | 010000 | 100000 | 110000 |
| 000001 | 010001 | 100001 | 110001 |
| 000010 | 010010 | 100010 | 110010 |
| 000011 | 010011 | 100011 | 110011 |
| 000100 | 010100 | 100100 | 110100 |
| 000101 | 010101 | 100101 | 110101 |
| 000110 | 010110 | 100110 | 110110 |
| 000111 | 010111 | 100111 | 110111 |
| 001000 | 011000 | 101000 | 111000 |
| 001001 | 011001 | 101001 | 111001 |
| 001010 | 011010 | 101010 | 111010 |
| 001011 | 011011 | 101011 | 111011 |
| 001100 | 011100 | 101100 | 111100 |
| 001101 | 011101 | 101101 | 111101 |
| 001110 | 011110 | 101110 | 111110 |
| 001111 | 011111 | 101111 | 111111 |




**DNHI**: Pick 10 different sequences from the table above and figure out their values in decimal.

# Properties and Rules of IEEE 754



# Special properties of IEEE encoding

- FP Zero Same as Integer Zero: all bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - will be greater than any other values
    - what should comparison yield?
  - Otherwise proper ordering
    - denorm vs. normalized
    - normalized vs. infinity

# Arithmetic Operations with Rounding

- $x +_f y = \text{Round}(x + y)$
- $x *_f y = \text{Round}(x * y)$
- Basic idea
  - Compute exact result
  - Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into frac



# Different Rounding Modes

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 |
| **Towards Nearest Even (default)** | **$1** | **$2** | **$2** | **$2** | **–$2** |

Same as regular rounding when we are not at the halfway point.

Round towards nearest even number when the value is at the halfway point.



# Round to Even - a closer look

- Default Rounding Mode
- All others are statistically biased
  - Sum of a set of positive numbers will consistently be over- or under- estimated
- Applying to Other Decimal Places
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth
    - 7.89**49999** 7.89 (Less than halfway - round down)
    - 7.89**50001** 7.90 (Greater than halfway - round up)
    - 7.89**50000** 7.90 (Halfway - round up)
    - 7.88**50000** 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 = $100\ldots_2$
- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

| Value | Binary | Rounded | Action | Rounded Value |
|---|---|---|---|---|
| 2 3/32 | $10.00\mathbf{011}_2$ | $10.00_2$ | (< ⅛ - down) | 2 |
| 2 3/16 | $10.00\mathbf{110}_2$ | $10.01_2$ | (> ⅛ - up) | 2 ¼ |
| 2 ⅞ | $10.11\mathbf{100}_2$ | $11.00_2$ | ( ⅛ - up) | 3 |
| 2 ⅝ | $10.10\mathbf{100}_2$ | $10.10_2$ | ( ⅛ - down) | 2 ½ |

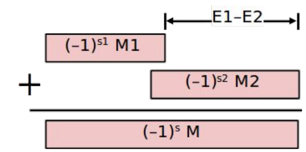
## Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ *\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign `s`: `s1 ^ s2` `(this is xor, not exponentiation)`
  - Significand `M`: `M1 * 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
  - Most expensive is multiplication of significands (but that is done just like for integers)



## Addition

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






## Properties of Floating Point Addition

- Closed under addition? **YES**
  - But may generate infinity or NaN
- Commutative? **YES**
- Associative? **NO**
  - Overflow and inexactness of rounding:

    $(3.14 + 1e^{10}) - 1e^{10} = 0$, $3.14 + (1e^{10} - 1e^{10}) = 3.14$
- 0 is additive identity? **YES**
- Every element has additive inverse? **ALMOST**
  - Yes, except for infinities & NaNs
- Monotonicity **ALMOST**
  - $a \geq b \Rightarrow a+c \geq b+c$?
  - Except for infinities & NaNs



## Properties of Floating Point Multiplication

- Closed under multiplication? **YES**
  - But may generate infinity or NaN
- Multiplication Commutative? **YES**
- Multiplication is Associative? **NO**
  - Possibility of overflow, inexactness of rounding

    Ex: $(1e^{20} * 1e^{20}) * 1e^{-20} = \text{inf}$, $1e^{20} * (1e^{20} * 1e^{-20}) = 1e^{20}$
- 1 is multiplicative identity? **YES**
- Multiplication distributes over addition? **NO**
  - Possibility of overflow, inexactness of rounding

    $1e^{20} * (1e^{20} - 1e^{20}) = 0.0$, $1e^{20} * 1e^{20} - 1e^{20} * 1e^{20} = \text{NaN}$
- Monotonicity **ALMOST**
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$?
  - Except for infinities & NaNs

# Floating Point in C



## C Language

- C Guarantees Two Levels
  - `float` single precision
  - `double` double precision
- Conversions/Casting
  - casting between `int`, `float`, and `double` **changes** bit representation
  - `double/float → int`
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - `int → double`
    - Exact conversion, as long as int has ≤ 53 bit word size
  - `int → float`
    - Will round according to rounding mode



## Puzzles (DNHI)

For each of the following C expressions, either:

- Argue that it is true for all possible argument values
- Explain why if not true

Assume:

- ```
  int x = …;
  float f = …;
  double d = …;
  ```
- neither `d` nor `f` is `NaN`

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