Floating Point II

ACKNOWLEDGEMENT: These slides have been modified by your your CMPT 295 instructor and CS:APP textbook authors. However, please report all mistakes to your instructor.

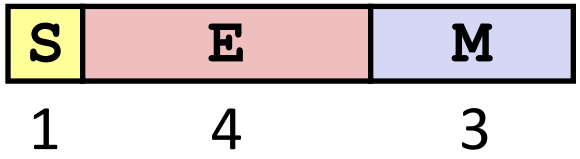
Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- **Floating-point operations and rounding**
- Floating-point in C

- There are many more details that we won't cover
  - It's a 58-page standard…

Tiny Floating Point Representation

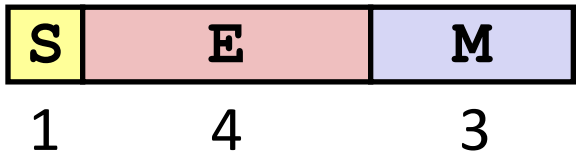
- We will use the following **8-bit** floating point representation to illustrate some key points:



- Assume that it has the same properties as IEEE floating point:
  - bias =
  - encoding of -0 =
  - encoding of $+\infty$ =
  - encoding of the largest (+) normalized # =
  - encoding of the smallest (+) normalized # =

Peer Instruction Question

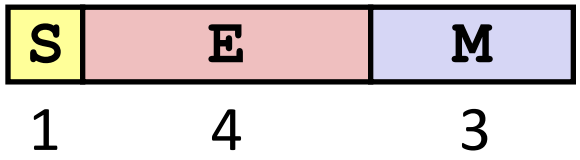
- Using our **8-bit** representation, what value gets stored when we try to encode **2.625** = $2^1 + 2^{-1} + 2^{-3}$?



A. + 2.5
B. + 2.625
C. + 2.75
D. + 3.25
E. We're lost…

Peer Instruction Question

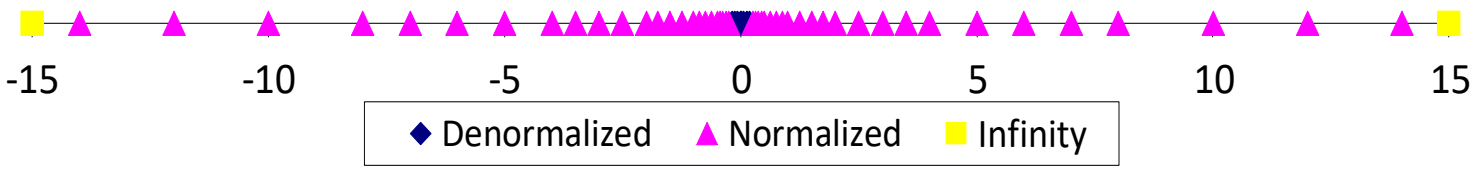
- Using our **8-bit** representation, what value gets stored when we try to encode **384** = $2^8 + 2^7$?



A. + 256
B. + 384
C. + ∞
D. NaN
E. We're lost…

Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity **Overflow** (Exp too large)
  - Between zero and smallest denorm **Underflow** (Exp too small)
  - Between norm numbers? **Rounding**
- Given a FP number, what's the bit pattern of the next largest representable number?
  - What is this "step" when Exp = 0?
  - What is this "step" when Exp = 100?
- Distribution of values is denser toward zero

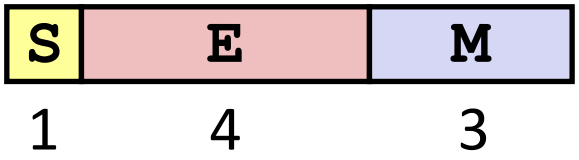


Floating Point Rounding

This is extra (non-testable) material

- The IEEE 754 standard actually specifies different rounding modes:
  - Round to nearest, ties to nearest even digit
  - Round toward $+\infty$ (round up)
  - Round toward $-\infty$ (round down)
  - Round toward 0 (truncation)

- In our tiny example:
  - Man = 1.001 01 rounded to M = 0b001
  - Man = 1.001 11 rounded to M = 0b010
  - Man = 1.001 10 rounded to M = 0b010



Floating Point Operations: Basic Idea

$$\text{Value} = (-1)^{S} \times \text{Mantissa} \times 2^{\text{Exponent}}$$

- $x +_f y = \text{Round}(x + y)$
- $x *_f y = \text{Round}(x * y)$

- Basic idea for floating point operations:
  - First, compute the exact result
  - Then *round* the result to make it fit into the specificed precision (width of M)
    - Possibly over/underflow if exponent outside of range

Mathematical Properties of FP Operations

- Overflow yields ±∞ and underflow yields 0
- Floats with value ±∞ and NaN can be used in operations
  - Result usually still ±∞ or NaN, but not always intuitive
- Floating point operations do not work like real math, due to rounding
  - Not associative: `(3.14+1e100)-1e100 != 3.14+(1e100-1e100)`

    `0` `3.14`
  - Not distributive: `100*(0.1+0.2) != 100*0.1+100*0.2`

    `30.000000000000003553` `30`
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- **Floating-point in C**

- There are many more details that we won't cover
  - It's a 58-page standard…

Floating Point in C

!!!

- Two common levels of precision:

`float`	`1.0f`	single precision (32-bit)
`double`	`1.0`	double precision (64-bit)

- `#include <math.h>` to get `INFINITY` and `NAN` constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!

Floating Point Conversions in C

- Casting between `int`, `float`, and `double` **changes the bit representation**
  - `int` → `float`
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - `int` or `float` → `double`
    - Exact conversion (all 32-bit `int`s representable)
  - `long` → `double`
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - "Not defined" when out of range or NaN: generally sets to `Tmin` (even if the value is a very big positive)

Peer Instruction Question

- We execute the following code in C. How many bytes are the same (value and position) between `i` and `f`?

```
int i = 384;  // 2^8 + 2^7
float f = (float) i;
```

A. 0 bytes
B. 1 byte
C. 2 bytes
D. 3 bytes
E. We're lost…

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for (i = 0; i < 10; i++)
    f2 += 1.0/10.0;

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.9f\n", f1);
  printf("f2 = %10.9f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - "Gaps" produced in representable numbers means we can lose precision, unlike `ints`
    - Some "simple fractions" have no exact representation (*e.g.* 0.2)
    - "Every operation gets a slightly wrong result"
- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results
- Never test floating point values for equality!
- Careful when converting between `ints` and `floats`!

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:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 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)