CS356: Discussion #3

Floating Point Representation

USC University of Southern California

IEEE 754 Standard: 32-bit

Binary32 Format (`float`)

		Mantissa
sign	**exponent**	**fraction**
1 bit	8 bits	23 bits

[sign (1 bit) | exponent (8 bits) | fraction (23 bits)]

- **Exponent** encodes values [-126, 127] as unsigned integers with bias
- Exponent of all 0's reserved for:
  - Zeros: `0x00000000` (0.0), `0x80000000` (-0.0)
  - Denormalized values: $(-1)^{sign} \times 0.(\text{fraction}) \times 2^{1-127}$ (nonzero fraction)

Denormals allow representation of numbers very close to 0. (very small number)

- Exponent of all 1's reserved for:
  - Infinity: `0x7F800000` (∞), `0xFF800000` (-∞)
  - NaN: with any nonzero fraction
- **Decimal value** (Normalized): $(-1)^{sign} \times 1.(\text{fraction}) \times 2^{\text{exponent - 127}}$
- **Decimal range**: (7 significant decimal digits) $\times 10^{\pm 38}$

Special Numbers (32-bit)

Description	exp (8 bits)	frac (23 bits)	Lower 31 bits (hex)	Decimal value
Zero	00…00	00…00	`0x00000000`	0.0
Smallest Pos Denormalized	00…00	00…01	`0x00000001`	$2^{-23} \times 2^{-126}$
Largest Denormalized	00…00	11…11	`0x007FFFFF`	$(1.0-\varepsilon) \times 2^{-126}$
Smallest Pos Normalized	00…01	00…00	`0x00800000`	1.0×2^{-126}
One	01…11	00…00	`0x3F800000`	1.0
Largest Normalized	11…10	11…11	`0x7F7FFFFF`	$(2.0-\varepsilon) \times 2^{127}$
Infinity	11…11	00…00	**`0x7F800000`**	Infinity
NaN	11…11	Nonzero	> **`0x7F800000`**	NaN

Any 0x7F8xxxxx with nonzero fraction = NaN

0x7F800000 → +∞
0xFF800000 → –∞

NaN, sign bit is ignored

IEEE 754 Standard: 64-bit

Binary64 Format (`double`)

sign	exponent	fraction
1 bit	11 bits	52 bits

- **Exponent** encodes values [-1022, 1023] as unsigned integers with bias
- Exponent of all 0's reserved for:
  - Zeros: `0x0000000000000000` (0.0), `0x8000000000000000` (-0.0)
  - Denormalized values: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{1-1023}$ (nonzero fraction)
- Exponent of all 1's reserved for:
  - Infinity: `0x7FF0000000000000` (∞), `0xFFF0000000000000` (-∞)
  - NaN: any nonzero fraction
- **Decimal value** (Normalized): $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - 1023}}$
- **Decimal range**: (≈ 16 significant decimal digits) $\times 10^{\pm 308}$

Other formats, same patterns

1 sign bit, **k** bits for exponent, **m** bits for fraction

Bias = $2^{k-1}-1$

Normalized: $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - Bias}}$

Denormalized: $(-1)^{\text{sign}} \times \mathbf{0.}(\text{fraction}) \times 2^{1\text{-Bias}}$

To **negate**, just flip the sign bit (except for NaN)

Exercise: IEEE 754 to Decimal Conversion

What number is represented by the single-precision float

11000000101000....

Exercise: IEEE 754 to Decimal Conversion

What number is represented by the single-precision float

11000000101000....

Sign = -1
Fraction = 1.01
Biased-Exponent = 129
Exponent = 127

$$x = (-1) * 1.01 * 2^2$$
$$= -1 * 101$$
$$= (-5)_{10}$$

Exercise: Decimal to IEEE 754 Conversion

What will be the single precision representation of the decimal number:

85.625

Exercise: Decimal to IEEE 754 Conversion

What will be the single precision representation of the decimal number:

85.625

85 / 2 = 42 remainder 1
42 / 2 = 21 remainder 0
21 / 2 = 10 remainder 1
10 / 2 = 5 remainder 0
5 / 2 = 2 remainder 1
2 / 2 = 1 remainder 0
1 / 2 = 0 remainder 1

0.625 × 2 = 1.25 → 1
0.25 × 2 = 0.5 → 0
0.5 × 2 = 1.0 → 1

85.625 = 1010101.101

$x = 1.010101101 * 2^6$

Where did 6 come from? -> moved 6 points up, which means 2^6

Sign = 0

Exponent = 127 + 6 = 133 = $(10000101)_2$

Stored exponent = 6 (2^6) + 127 (Bias) = 133.
Binary of 133 = 10000101.

Mantissa = $(0101011010000\ldots)_2$

We take the bits after the leading 1. → 010101101... to fill the 23 bits

Answer = 0 10000101 010 1011 0100 0000....
= 0x42AB4000

Exercise: Detect Denormalized Numbers

Write a function `int denorm(unsigned int x)` that returns 1 if `x` is denormalized, and 0 otherwise.

https://godbolt.org/z/h39M3rq57

Exercise: Detect Denormalized Numbers

Write a function `int denorm(unsigned int x)` that returns 1 if `x` is denormalized, and 0 otherwise.

Solution 1 (5 Operators)

```
int denorm(unsigned int x) {
    return !((x >> 23) & 0xFF) && (x & 0x007FFFFF);
}
```

Explanation
1. (x >> 23) & 0xFF
- Shifts x right by 23 → isolates the exponent field (8 bits).
- & 0xFF masks it to exactly those 8 bits.
- If this equals 0, exponent is all zeros.
2. !((x >> 23) & 0xFF)
- Returns true if exponent = 0.
3. (x & 0x007FFFFF)
- Masks out the bottom 23 bits → the fraction field.
- Nonzero means fraction ≠ 0.
4. Combine with &&
- True only if exponent = 0 AND fraction ≠ 0.
- Exactly the condition for denormalized numbers.

Exercise: Detect Denormalized Numbers

Write a function `int denorm(unsigned int x)` that returns 1 if `x` is denormalized, and 0 otherwise.

Solution 1 (5 Operators)

```
int denorm(unsigned int x) {
    return !((x >> 23) & 0xFF) && (x & 0x007FFFFF);
}
```

Solution 2 (4 Operators)

```
int denorm(unsigned int x) {
    int t = x & 0x7FFFFFFF;
    if (t < 0x800000 && t > 0)
        return 1;
    else
        return 0;
}
```

0x00000001 … 0x007FFFFF = all denormals.

Rounding and Casting in C

The IEEE 754 standard defines four **rounding modes**:

- **Round to nearest, ties to even**: default rounding in C for `float/double` ops
- **Round towards zero** (truncation): used to cast `float/double` to `int`
- **Round up** (ceiling): go towards +∞ (gives an upper bound)
- **Round down** (floor): go towards -∞ (gives a lower bound)

Example: 8-bit frac → 4-bit frac, **Round to nearest, ties to even**

10101001	→	**1011**
10100110	→	**1010**
10101000	→	**1010**
10111000	→	**1100**

Exercise: Casting

```
short s;
int i;
float f;
double d;
```

Do the following statements always hold?

- `(float) ((double) f) == f` YES
- `(double) ((float) d) == d` NO

double can represent every float exactly (53-bit vs 24-bit precision respectively.).

- `(int) ((double) i) == i` YES
- `(int) ((float) i) == i` NO
- `(short) ((float) s) == s` YES

double has 53 bits of integer precision, so all 32-bit ints are exact.
Float only holds 24, not 32.
short is 15 bits.

Floating point operations in C

Floating point operations

- Addition and subtraction are **not associative**
    - Add small-magnitude numbers before large-magnitude ones
- Multiplication and division are **not associative** (**nor distributive**)
    - Control magnitude with divisions (if possible)
      `(big1 * big2) / (big3 * big4)` overflows on first multiplication
      `1/big3 * 1/big4 * big1 * big2` underflows on first multiplication
      `(big1 / big3) * (big2 / big4)` is likely better
- Comparison should use `fabs(x-y) < epsilon` instead of `x==y`
- **Instead for integers (last week):**
    - Addition of unsigned or signed (2's complement) integers is associative, even in the case of overflow
    - You can use `x==y`

DataLab: What to implement (2)

Floating-point Problems: 4-byte constants (`0x12345678`), loops (`for`, `while`), conditionals (`if`), comparisons (`x==y`, `x>y`), operators `- && ||`, but no macros (`INT_MAX`), no `float` types or operations.

The `unsigned` input and `int` output are the **bit-level equivalent** of 32-bit `float`s

- `unsigned floatAbsVal(unsigned uf)`
- `int floatIsEqual(unsigned uf, unsigned ug)`
- `int floatPower2(int x)`

Exercise: Floating-point Sign

Write a function `int sign(unsigned int x)` that returns the sign of `x` as 1/-1

```
int sign(unsigned int x) {

}
```

https://godbolt.org/z/jxG33rPYo

Exercise: Floating-point Sign

Write a function `int sign(unsigned int x)` that returns the sign of `x` as 1/-1

```
int sign(unsigned int x) {
    return (x & 0x80000000) ? -1 : 1;
}
```

```
         x: 10101010 01010101 10101010 01010101
0x80000000: 10000000 00000000 00000000 00000000

  negative: 10000000 00000000 00000000 00000000
  positive: 00000000 00000000 00000000 00000000
```

https://godbolt.org/z/jxG33rPYo

Exercise: Floating-point Sign

Write a function `int sign(unsigned int x)` that returns the sign of `x` as 1/-1

```
int sign(unsigned int x) {
    return ((int)x >> 31) | 0x1;
}
```

```
        x: 10101010 01010101 10101010 01010101
  x >> 31: 11111111 11111111 11111111 11111111

       -1: 11111111 11111111 11111111 11111111
        1: 00000000 00000000 00000000 00000001
```

https://godbolt.org/z/jxG33rPYo

Exercise: Extract Exponent

Write a function `int exponent(unsigned int x)` that returns the exponent of `x` (as is, including the bias).

```
int exponent(unsigned int x) {

}
```

https://godbolt.org/z/an8sjPeK3

Exercise: Extract Exponent

Write a function `int exponent(unsigned int x)` that returns the exponent of `x` (as is, including the bias).

```
int exponent(unsigned int x) {
    return (x >> 23) & 0xFF;
}
```

```
x:    00111111 10000000 00000000 00000000
       exponent
```

https://godbolt.org/z/an8sjPeK3

Exercise: Extract Fraction

Write a function `int fraction(unsigned int x)` returning the fraction of `x`, including the implicit leading bit equal to 1 (ignore denormalized numbers).

```
int fraction(unsigned int x) {

}
```

https://godbolt.org/z/an8sjPeK3

Exercise: Extract Fraction

Write a function `int fraction(unsigned int x)` returning the fraction of `x`, including the implicit leading bit equal to 1 (ignore denormalized numbers).

```
int fraction(unsigned int x) {
    return (x & 0x007FFFFF) | 0x00800000;
}
```

```
 x:    00111111 01101001 00000000 00000000
                 fraction (without leading bit)

                11101001 00000000 00000000
                 fraction (with leading bit 1)
```

https://godbolt.org/z/an8sjPeK3

Exercise: Detect Floating-point Zero

Write a function `int is_zero(unsigned int x)` returning 1 if `x` is 0.0 or -0.0, and 0 otherwise.

```
int is_zero(unsigned int x) {

}
```

https://godbolt.org/z/ajMMsb8o5

Exercise: Detect Floating-point Zero

Write a function `int is_zero(unsigned int x)` returning 1 if `x` is 0.0 or -0.0, and 0 otherwise.

```
int is_zero(unsigned int x) {
    return (x == 0x00000000 || x == 0x80000000) ? 1 : 0;
}
```

```
+0:  00000000 00000000 00000000 00000000
-0:  10000000 00000000 00000000 00000000
```

https://godbolt.org/z/ajMMsb8o5

Exercise: Detect Floating-point Zero

Write a function `int is_zero(unsigned int x)` returning 1 if `x` is 0.0 or -0.0, and 0 otherwise.

```
int is_zero(unsigned int x) {
    return !(x & 0x7FFFFFFF) ;
}
```

```
+0:  00000000 00000000 00000000 00000000
-0:  10000000 00000000 00000000 00000000
```

https://godbolt.org/z/ajMMsb8o5