Floating Point

CS 3410: Computer System Organization and Programming

[G. Guidi, A. Sampson, Z. Susag, and H. Weatherspoon]

Goals for today

Floats: Numbers with a decimal point (rather, a “binary point”!)

- Representing fractional numbers in binary
- Fixed point
- Floating points
  - Special cases
  - Other floating point formats
  - Guidelines

How to represent fractional numbers in binary?

- C has a float type like other languages
- Floats work for numbers with a decimal point in them
- **How do we represent fractional numbers with bits?**
- Implications on performance and accuracy

Example: float.c

```c
#include <stdio.h>

int main() {
    float n = 8.4f;
    printf("%f\n", n * 5.0f);
    return 0;
}
```

Fractional Numbers in Binary

Base10

hundreds ones tenths hundredths

$$19.64_{10} = 1 \bullet 10 + 9 \cdot 10^{1} + 6 \cdot 10^{-1} + 4 \cdot 10^{-2} = 19.64$$

Base 2

fours twos halves fourths

$$10.01_{2} = 1 \cdot 2^{1} + 0 \cdot 2^{0} + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2 + 1/4 = 2.25$$

Warning

- Fractional numbers in binary have a finite number of bits
- Thus,finite precision

- 1.0 + 2.0 != 3.0
- Seehttps://0.30000000000000004.com/

- Seenotable floating point errors
  - https://en.wikipedia.org/wiki/Ariane_5#Notable_launches
  - https://en.wikipedia.org/wiki/Ariane_flight_V88
  - https://en.wikipedia.org/wiki/Pentium_FDIV_bug

Example: float.c

```
#include <stdio.h>

int main() {
    float x = 0.00000001f;
    float y = 0.00000002f;

    printf("x = %e\n", x);
    printf("y = %e\n", y);
    printf("y - x = %e\n", y - x);

    printf("1+x = %e\n", 1.0f + x);
    printf("1+y = %e\n", 1.0f + y);
    printf("(1+y) - (1+x) = %e\n", (1.0f + y) - (1.0f + x));
    return 0;
}
```

Goals for today

Floats: Numbers with a decimal point (rather, a “binary point”!)

- Representing fractional numbers in binary
- **Fixed point**
- Floating points
  - Special cases
  - Other floating point formats
  - Guidelines

Fixed Point

- Key
  - Like scientific notation, but in base 2
  - E.g. $34.10_{10} \times 10^{-5}$
  - E.g. $1001_2 \times 2^{-2} = 10.01_2 = 2.25_{10}$
  - Notation: $i \times 2^e$ where i is the integer and e determines where the binary point goes
- Idea
  - How many bits. Call this bit count n
  - Where will the binary point go? Call this position e for *exponent*
    - $e = 0$ the binary point goes at the very end (so it's just a normal integer)
    - $e = -1$ means there is one bit after the binary point
    - $e = 1$ means tack on one zero before the binary point
- Examples
  - $n = 4$, $e = -2$, and bit pattern `1001`
    - $10.01_2 = 2.25_{10}$
  - n = 4, e = -3, and bit pattern `1111`
    - $1111 = 1 + ½ + ¼ + 1/8 = 1.875_{10}$
  - n = 4, e = 1, and bit pattern `0101`
    - $01010_2 = 10_{10}$

Fixed Point

- Good and bad
  - *e* is metadata and <u>*not*</u> part of the actual data that the computer stores
  - The same bit pattern can represent many different numbers!
    Depends on the exponent that the programmer has in mind
  - Very fast and used a lot for machine learning (ML) and digital signal processing (DSP)

- However, due to limitation of not being self contained, most software used a different strategy, **floating point**

Goals for today

Floats: Numbers with a decimal point (rather, a "binary point"!)

- Representing fractional numbers in binary
- Fixed point
- **Floating points**
  - Special cases
  - Other floating point formats
  - Guidelines

Floating Point

```c
#include <stdio.h>

int main() {
    float n = 34.10f;
    float big = n * 123456789.0f;
    float small = n / 123456789.0f;
    printf("big = %e\nsmall = %e\n", big, small);
    return 0;
}
```

Floating Point

- Float allows the binary point to float
- Every float consists of *sign*, *exponent*, and *significand* (*mantissa*), packed together
  - Where *s*, *e*, and *g* represent this number:

$$(-1)^s \times 1.g \times 2^{e-127}$$

- A 32-bit float has
  - 1-bit <u>*sign*</u>, **s**, which is a single bit
    - 0 for positive, 1 for negative
  - 8-bit <u>*exponent*</u>, **e**, which is an unsigned integer
    - Scaling term, 2^{e-127}, i.e. determines where the binary point goes
    - -127 is *bias* allowing the unsigned exponent to represent a wide range of both positive and negative binary-point positions
  - 23-bit <u>*significand*</u> (also called the <u>*mantissa*</u>), **g**, which is unsigned integer
    - Take the bits from *g* and put them all after the binary point, with a 1 in the ones place
    - The significand is the "main" part of the number,
      so (in the normal case) it always represents a number between 1.0 and 2.0

Check out https://float.exposed/

Floating Point

- Example1: Convert 8.25 to float
- Step 1: Write binary representation: 1000.01_2
- Step 2: Normalize: 1.00001×2^3
- Step 3: Break into the three components
  - s = 0
  - g = .00001 = 000 0100 0000 0000 0000 0000
  - e = 3 + 127 = 130
  - 32-bit float: 0100 0001 0000 0100 0000 0000 0000 0000 = 0x41040000

Check out https://float.exposed/

Floating Point

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main() {
    uint32_t bits = 0x41040000;

    // Copy the to a variable with a different type
    float val;
    memcpy(&val, &bits, sizeof(val));

    // Print the bits as a floating-point number
    printf(“%f\n", val);
    return 0;
}
```

Floating Point

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main() {
    uint32_t bits = 0x41040000;
    uint32_t mantissa = bits & 0x007fffff; // mask to isolate mantissa
    uint32_t exponent = (bits & 0x7f800000) >> 23; // bit and bit shift
    uint32_t sign = (bits & 80000000) >> 31; // mask and bit shift

    printf(“s = %b, e = %b, g = %b \n", sign, exponent, mantissa);
    return 0;
}
```

Special cases, Not a number (NaN) and Infinity

- +0.0 and -0.0, i.e. s = 0 or s=1, but you have to set both e=0 and g=0
- When e = 0, but g $\neq$ 0
  - *Denormalized number*
  - The rule is that denormalized numbers represent the value $(-1)^s \times 0.g \times 2^{-126}$
  - The important difference is that we now use 0.g instead of 1.g
  - These values are useful to eke out the last drops of precision for extremely small numbers.
- e is all ones and g = 0 is infinity (there is a +∞ and -∞, when s=0 or s=1!)
- e is all ones and g $\neq$ 0 is NaN

- Dividing zero by zero is NaN, but dividing other numbers by zero is infinity!

Floating Point

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main() {
    printf(“%f\n", 0.0f / 0.0f); // NaN
    printf(“%f\n", 5.0f / 0.0f); // Infinity
    return 0;
}
```

Other floating point formats

- float: 32-bit, “single precision”
  - 1-bit sign, 8-bit exponent, 23-bit significand
- double: 64-bit, “double precision”
  - 1-bit sign
  - 11-bit exponent
  - 54-bit significand
- Half-precision: 16-bit, “half precision"
  - 1-bit sign
  - 5-bit exponent
  - 10-bit significand
- bfloat, 16-bit, “brain floating point”
  - Invented for machine learning (ML): Deep learning needs more range, but less precisi
  - 1-bit sign
  - 8-bit exponent
  - 7-bit significand

Guidelines

- Floating-point numbers are *__not__* real numbers
  - Expect to accumulate some error when using floats
- Never use floating-point numbers to represent currency
  - When people say $123.45, they want that exact number of cents, not $123.40000152.
  - Use an integer number of cents: i.e., a fixed-point representation with a fixed decimal point
- Be suspicious of equality, f1 == f2
  - E.g. try (0.1 + 0.2) == 0.3 ?
  - Consider using an "error tolerance" in comparisons, like abs(f1 - f2) < epsilon.
- Floating-point arithmetic is expensive
  - It is slower and more energy than integer or fixed-point arithmetic
  - The flexibility is expensive since the complexity requires more complex for the hardware
  - As a result, a lot of applications such as ML convert (quantize) models to a fixed-point representation so they can run efficientl.