UNSW SYDNEY

COMP1521 24T2 Lec09

Floating Point Representation

2024
Angela Finlayson
Format from Hammond Pearce,
material from COMP1521

Assignment 1 is due Friday 6pm

Recap Exercise

For all of these assume we are working with uint8_t variables

Question 1: Assume mask = 2. What effect do the following have?

- z = z | mask
- z = z & ~mask
- z = z ^ mask

Question 2: How could I check whether the 2 most significant bits of z are 1’s?

Floating Point Representation

- Learn **IEEE 754**, the industry standard
- Crucial for working with numerical computations in computing
- Understand precision and accuracy limitations
  - Why using them for finance is unwise
  - Why sometimes
    - a + b == a   (even if b is not 0)
    - if (a == b) is not a good idea

Floating Point Numbers

C has 3 floating point types

- **`float`** ... typically 32-bit quantity (lower precision, narrower range)
- **`double`** ... typically 64-bit quantity (higher precision, wider range)
- **`long double`** … typically 128-bit quantity (but maybe only 80 bits used)

Literal floating point values by default are **`double`**: `3.14159`, `1.0/3`, `1.0e-9`

Reminder: division of 2 ints gives an int e.g. 1/2

Code Demos

floating_types.c

double_output.c

Fractions in different bases

The decimal fraction 0.75 means

- $7*10^{-1} + 5*10^{-2} = 0.7 + 0.05 = 0.75$
- or equivalently $75/10^{2} = 75/100 = 0.75$

Similarly 0b0.11 means

- $1*2^{-1} + 1*2^{-2} = 0.5 + 0.25 = 0.75$
- or equivalently $3/2^{2} = 3/4 = 0.75$

Similarly 0x0.C means

- $12*16^{-1} = 0.75$
- or equivalently $12/16^{1} = 3/4 = 0.75$

Note: We call the . a radix point rather than a decimal point when we are dealing with other bases.

Converting fractions to other bases

The algorithm to convert a decimal fraction to another base is:

- take the fractional component and multiply by the base
  - the whole number becomes the next digit to the right of the radix point in our converted fraction.
- repeat with the remaining fraction until the fractional part becomes exhausted or we have sufficient digits (this process is not guaranteed to terminate).

Example: Converting Fractions

For example if we want to convert 0.3125 to base 2

- 0.3125 * 2 = **0**.625
- 0.625 * 2 = **1**.25
- 0.25 * 2 = **0**.5
- 0.5 * 2 = **1**.0

Therefore 0.3125 = 0b0.0101

Exercise 1:

Convert the following decimal values into binary

- 12.625
- 0.1

Code Demos

double_lies.c

double_imprecision.c

Floating Point Issues

Representing floating point numbers with a fixed small number of bits

- a finite number of bit patterns
- can only represent a finite subset of reals
  - almost all real values will have no exact representation
  - value of arithmetic operations may be real with no exact representation
  - we must use closest value which can be exactly represented
  - this approximation introduces an error into our calculations
  - often, does not matter
  - sometimes ... can be disastrous
    - eg pacemakers, finance

Fixed Point Representation

Fixed-point is a simple trick to represent fractional numbers as integers

- every value is multiplied by a particular constant and stored as an integer
  - e.g. if constant is 1000 then 56125 represents 56.125
  - we could not represent 3.141592
- useful for some problems
- used on small embedded processors without silicon floating point
- major limitation is range:
  - 16 bits used for integer part and 16 bits for fraction
    - minimum $2^{-16} \approx 0.000015$
    - maximum $2^{15} \approx 32768$

IEEE Standard: Exponential Representation

Idea: use **scientific notation**

- e.g $6.0221515 * 10^{23}$

But in binary:

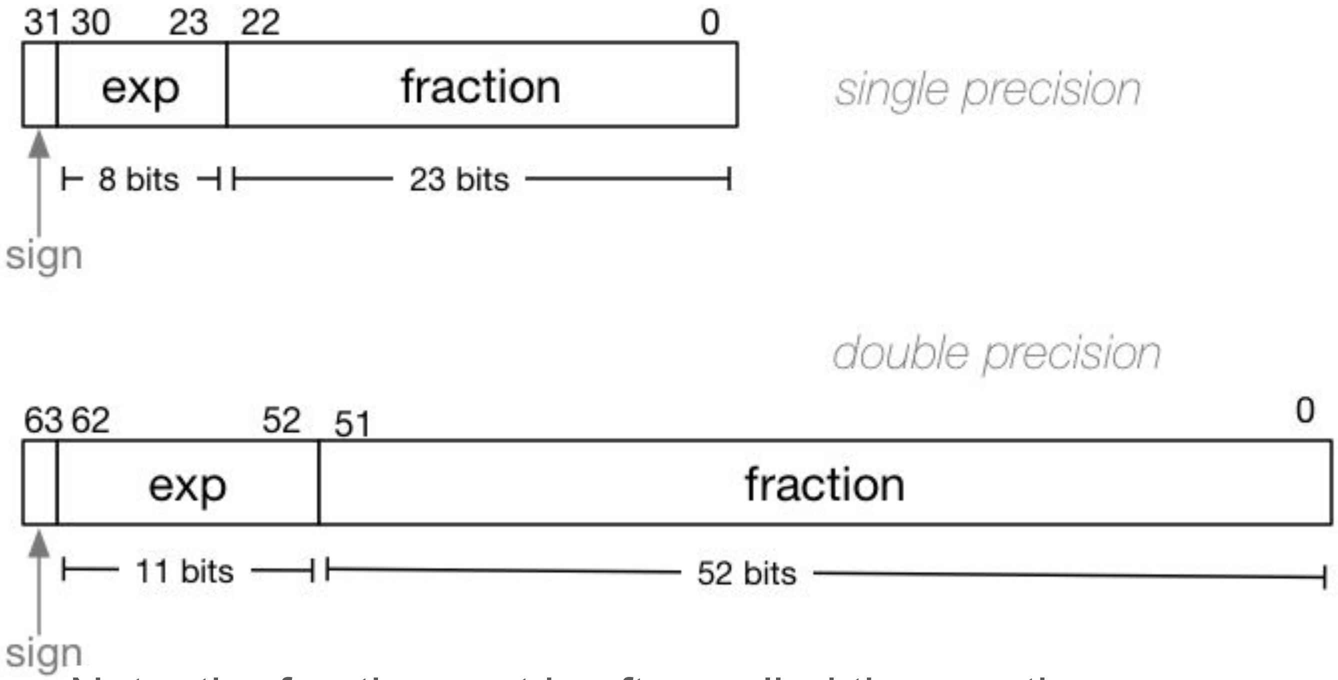
- $10.6875 = 1010.1011$

  $= 1.0101011 * 2^{3}$

Allows a much bigger range of values to be represented than fixed point

- 8 bits for the exponent can represent numbers from 10^{-38} .. 10^{38}
- 11 bits for the exponent can represent numbers from 10^{-308} .. 10^{308}

IEEE 754 Standard



Note: the fraction part is often called the mantissa

IEEE 754 Standard: Sign and Fraction

sign: 0 for positive, 1 for negative

We don’t want multiple representations of the same number so we normalise it

- (i.e. 1.1001×2^3 rather than 1100.1×2^0 or 11.001×2^2)
- better to have only one representation (one bit pattern) representing a value
  - multiple representations would make arithmetic slower on CPU

Weird hack: the first bit must be a one we don't need to store it

- as we long we have a special representation for zero
- To represent 1.**1001**$\times 2^3$ we would store **1001**000000… for the **fraction**.

IEEE 754 Standard: Exponent

Exponent is represented relative to a bias value *B*

- to represent exponent of x, we would store x+B
- for floats the bias is 127

So if we were representing 1.1001×2^3 we would store

(3+127) = 130 = 10000010 for a float

How bias is calculated:

- assume an 8-bit exponent, then bias B = 2^{8-1}-1 = 127
- valid bit patterns for exponent **00000001 .. 11111110** (1..254)
- exponent values we can represent -126 .. 127

IEEE 754 Example

150.75 = 10010110.11
// normalise fraction, compute exponent
= 1.001011011 × 2^7
// determine sign bit,
// map fraction to 24 bits, (don't store the leading 1)
// map exponent to 8 bits after adding on the bias of 127
= 01000011000101101100000000000000

where red is sign bit, green is exponent, blue is fraction

Note: *B*=127, *e*=2^7, so exponent = 127+7 = 134 = **10000110**

Check using explain_float_representation.c or Floating Point Calculator

Exercise 2: Floating Point Conversions

Question 1: Convert the decimal numbers 1 to a floating point number in IEEE 754 single-precision format.

Question 2: Convert the following IEEE 754 single-precision floating point numbers to decimal.

0 10000000 11000000000000000000000

1 01111110 10000000000000000000000

IEEE 754 Standard: Special Cases

Value	Exponent	Fraction	Example
0 (+ve or -ve)	0	0	
inf (∞ and -∞)	all 1's	0	1.0/0
nan	all 1's	<> 0	0.0/0

IEEE 754 infinity.c

Representation of +- infinity : propagates sensibly through calculations

```
double x = 1.0/0.0;

printf("%lf\n", x); //prints inf

printf("%lf\n", -x); //prints -inf

printf("%lf\n", x - 1); // prints inf

printf("%lf\n", 2 * atan(x)); // prints 3.141593

printf("%d\n", 42 < x); // prints 1 (true)

printf("%d\n", x == INFINITY); // prints 1 (true)
```

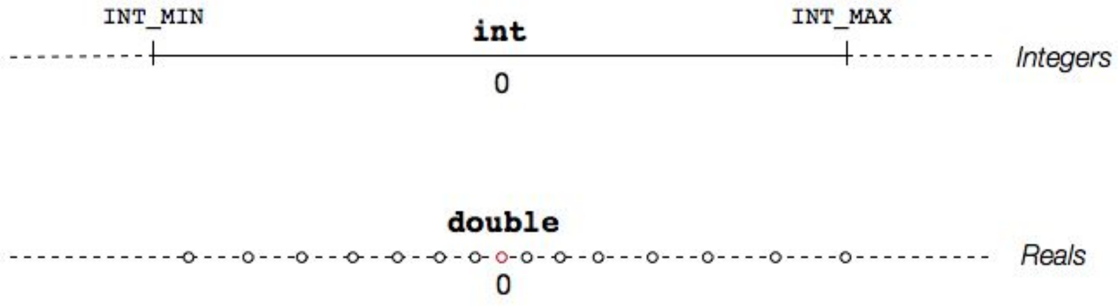
IEEE 754 nan.c

Representation for invalid results NaN (not a number)

- ensures errors propagates sensibly through calculations

```
double x = 0.0/0.0;

printf("%lf\n", x); //prints nan

printf("%lf\n", x - 1); // prints nan

printf("%d\n", x == x); // prints 0 (false)

printf("%d\n", isnan(x)); // prints 1 (true)
```

Distribution of Floating Point Numbers



integer ... subset (range) of the mathematical integers

floating point ... subset of the mathematical real numbers

floating point numbers not evenly distributed

- representations get further apart as values get bigger
- this works well for most calculations but can cause weird bugs

Distribution of Floating Point Numbers

double (IEEE 754 64 bit) has 52-bit fractions so:

- between 2^n and 2^{n+1} there are 2^{52} doubles evenly spaced
  - e.g. in the interval 2^{-42} and 2^{-43} there are 2^{52} doubles
  - and in the interval between 1 and 2 there are 2^{52} doubles
  - and in the interval between 2^{42} and 2^{43} there are 2^{52} doubles

- near 0.001 - doubles are about 0.00000000000000000002 apart
- near 1000 - doubles are about 0.0000000000002 apart
- near 1000000000000000 - doubles are about 0.25 apart
- **above 2^{53} - doubles are more than 1 apart**

Code Demos

double_catastrophe.c

double_not_always.c

double_disaster.c