Introduction to Software Development – CS 6010
Lecture 12 – Number Representations

Master of Software Development (MSD) Program

Varun Shankar

Fall 2023

Brief note on Characters

- A single character is typically stored in 1 Byte (8 bits)
- The computer will “look them up” in the ASCII Table (When displaying)
- Addition works (sort of)
  - ‘A’ + 3 == ‘D’
- Check to see if a capital letter…
  - if a character is <= 90 and >= 65 // Note: it is bad to use the ASCII value…
  - Remember, never write 90, or 65. Use ‘Z’ or ‘A’
    - Or ‘a’ or ‘z’ or ‘m’ or ‘0’ (the letter zero) or whatever letter you need.
    - char >= ‘A’ && char <= ‘Z’
- How to convert the letter ‘3’ to the number 3?
  - ‘3’ – ‘0’
- Convert the letter ‘C’ to the number 12
  - ‘C’ – ‘A’ + 10

How do we handle signed (-) numbers?

- How do you represent negative numbers in binary?

- One possibility is to represent negative numbers using “One’s Complement”.

- Say for example we have 4 bits to represent both +ve and –ve numbers.

- Now say we want -2.

One’s Complement

- Write down 2 in binary using 4 bits: 0010

- Then **invert *all the bits* :** 1101. This is treated as -2.

- The name follows from the fact that if you add a number and its negative, you’ll always get 1111 (0010 + 1101 = 1111, for example).
  - But this is weird… not 0000! More in a second.

- You can always take the most significant bit and look at it. If it’s 1, the number is –ve, if it’s 0 the number is +ve.

Weirdness in One’s Complement

- With 4 bits (or really any number of bits), when we add a number to its own negative, we get 1111.

- But we’d expect 0000. So what’s 1111?

- It’s *negative* 0!

- So +0 is different from -0 in one’s complement, which is really annoying. Must test for both +0 and -0 if computer is using one’s complement.

- Old computers used this (until the 1980s), but most computers use a better scheme called Two’s Complement instead.
  - One’s complement is now primarily used in older or specialized devices, or completely different contexts than representing negative numbers.

Two’s Complement

- Two’s complement comes from a really simply idea.
- Say for instance we have 3 bits to represent an integer.
- If we only allow +ve numbers (and 0), this lets us represent the values 0 – 7 (000 = 0, 001 = 1,… 111 = 7).
- If we want to allow both +ve and –ve numbers, one way is to split the range.
  - 000, 001, 010, and 011 are +ve (0, 1, 2, and 3 respectively).
  - 100, 101, 110, and 111 are –ve (-4,-3, -2, and -1 respectively).
  - Notice there’s no -0 here.
  - Also notice the +ve numbers all have a 0 in the first bit.

Two’s Complement

- In Two’s Complement, we specifically find a –ve number using the following algorithm.

- Say we only have 3 bits available and we want to find the binary value of -2. Recall that 2 is 010.
  - First, as in one’s complement, flip all the bits: gives 101.
  - Then add 1 to it (and ignore any overflow): gives 110.
  - This is what we saw before on the previous slide!

- What about -0? 000 -> 111 + 1 -> 000 (ignoring overflow).
  - So -0 is the same as 0.

Two’s Complement: Bits matter

- The number of bits used to represent the integer really does matter for correctness.

- Say we have 4 bits to represent integers. Then +5 is 0101. What is -5?
  - First, flip all bits. We get 1010.
  - Then add 1. We get 1011.

- But say we have 8 bits instead. +5 is 0000 0101. What is -5?
  - First, flip all bits. We get 1111 1010.
  - Then add 1. We get 1111 1011.

- In all cases, notice negative numbers have 1 for the most significant bit.
  - But… that’s weird, right? What about the power expansion?

Two's Complement: Power Expansion??

- Let's go back to 3 bits. The number 2 is 010. -2 is then 110.
- But, 010 = $(0* 2^2) + (1*2^1) + (0*2^0)$.
- How do we power expand 110? We'll get a wrong answer if we do it naively!
- For two's complement, the power expansion MUST take the sign of the Most significant bit (MSB) into account!
- 110 = $(-1 * 2^2) + (1*2^1) + (0*2^0) = -4 + 2 + 0 = -2$.

Two's Complement: Properties

- MSB acts as both a sign bit and has a value.
- Power expansion takes sign of MSB into account (0 = +ve, 1 = -ve).
- No difference between 0 and -0.
- Number of bits available to represent integer matters!

Two's Complement: More properties

- If you represent an integer with 4 bits, what's the *smallest* number you can represent here?
  - It's 1000, ie, -8. If you replace any of the 0s with 1s, you make the number larger!
  - This is called the **most negative number.**

- What's the **largest** number you can represent?
  - 0111, i.e., 7.

- Sanity check: what do you get if you add these two?
  - 1000 + 0111 = 1111.
  - What is this number? Be careful, sign bit is 1, so it's –ve.
  - 1111 = (-1*2^3) + (1*2^2) + (1*2^1) + 1*2^0 = -8 + 4 + 2 + 1 = -1.
  - Since -8 + 7 = -1, this is indeed correct!

Two's Complement: minus of minus?

- Assume again 3 bits for integers. Recall that -2 is 110.
- What's -(-2)? Should be +2!
- Let's check.
  - First, flip all bits. 110 -> 001.
  - Next, add 1. This gives 010, which is indeed 2!
  - Seems good, right?

- Not quite. There's one edge case: the most negative number.
- Given 3 bits, what is the most negative number?
  - 100 = -1*2^2 + 0*2^1 + 0*2^0 = -4.
- Let's try and check if -(-4) = +4.
  - Flip bits: 100 - > 011.
  - Add 1: gives 100, but that's just -4 again?!
  - But if we add 100 to 100 and ignore overflow, we get 000…

Two’s Complement: Most negative number

- The most negative number **cannot** be negated in two’s complement!
- This can result in all sorts of programming errors if you’re not careful.
- However, the number added to its negative (which is just itself) still gives 0.

Bit Manipulation

- Tomorrow we will talk about bit manipulation, so it important to understand what bits represent a number before we start messing with them.

Floating Point Numbers

- What is this?
  - 3.4e12 // It means:
  - 3.4 * 10^{12}
- This is called Normalized Scientific Notation
  - One non-zero decimal digit before the decimal point.
  - In other words: 34e11 is rewritten as 3.4e12
  - If we had .34e13, we would rewrite it as 3.4e12
- So how are floating point numbers stored in a computer?
  - Basically a binary version of scientific notation
- Significand, Base, Exponent (Using the above 3.4e12)
  - Significand – 3.4
  - Base – In decimal, base is 10 (in Binary the base is 2)
  - Exponent – 12 (Which is used to mean 10^12)

Floating Point Numbers

- A binary number in scientific notation looks like:
  - 1.01011 * 2^{exp} (where exp is some exponent value)
  - 0.111 * 2^5
- Since this is *normalized*, what is the only value (in binary) that can come before the decimal point?
  - 1
  - In other words, if we had 0.1111 * 2E12, we would have to write it as?
  - 1.111 * 2E11
  - Because there is always a one in this position, we don't need to store it, and we (ie, the computer) can just assume it is there.
- Making sure we understand this, what does .01011 * 2^N become?
- 1.011 * 2^{N-2}
  - And technically is stored in the computer as .011 * 2^{N-2}

Floating Point Numbers

- 1.011 * 2^N
- Remember, floating point numbers store the Mantissa (Significand) and the Exponent (the Base is implied).
- Additionally, floating point numbers store a *sign* bit.
- The decimal portion (1.011) is called the Mantissa (Significand)
- Note, there is a sign bit in floating point number storage
- Floats have how many bits?
  - 32 // These 32 bits are allocated like this:
- S Mantissa Exponent (Range: -128 -> 127)

S	Mantissa	Exponent
1	23 bits	8 bits

- Floats are accurate to approximately 6 or 7 decimal digits
- Doubles (64 bits) are allocated like this:

1	52 bits	11 bits

Special Floating Point Numbers

- NaN – Not a Number
  - 0 / 0, etc.
  - Contagious
- +Inf, -Inf
- 0.1 cannot be represented in binary…
  - 0.0001100110011001100110011001100110011001100110011…
- Can't represent many numbers exactly, so we need to know that most floating point numbers are approximated (to some level of accuracy).
- Example in lab today is .1 + .2 == .3
  - This turns out to not be true.
  - How can we tell if two floating point number (say A and B) are equal?
    - Just check to see if they are (very) close:
    - abs(A – B) < tolerance

Unicode (Characters)

- Extension to the ASCII Table (Much Bigger)
  - Includes characters for all written languages, emojis, etc.
- UTF-8
  - ASCII Stuff works the same
  - Use more bytes for higher Unicode characters
  - But we want most text (well standard English characters) to still take only one byte.
  - Compromise:
    - Use 128 values (half the range) with the 1^{st} 128 characters in the ASCII Table.
    - If the character’s value is negative, then it is a Unicode character and will use 2-4 total bytes and will have to be decoded.

Midterm Review (i)

- Variables
  - Types, Values, Casting, Declaration, Definition, Assigning, Scope
    - int vs float
    - char * vs string
    - int[ ] vs vector
  - Manipulating Characters

Midterm Review (ii)

- Asking questions in code
  - if statements, else if, else
  - Boolean logic
- Looping
  - For vs While
  - Syntax
- Strings, Vectors
  - length, substring, indexing into
- Programmer defined datatypes
  - struct
  - Declaring, Creating a variable, accessing fields

Midterm Review (iii)

- Functions
  - Declaration, Definition
  - parameters
    - pass by value
    - pass by reference
  - Returning values from functions
    - return one thing (but it could be a struct and thus hold multiple things)
    - return using parameters (reference params)
  - .h vs .cpp files (multi-file projects / .o files / linking)
- Binary and Hex Numbers
  - Conversion to from decimal numbers

Midterm Review (iv)

- File I/O
  - opening/closing files
  - reading data (strings, numbers, lines)

Starting Practice Problems

- Write a function that takes in a list of the number of cars that passed my house each hour over the last day, and returns the average number of cars that pass my house that day.
- Write a function that takes in 3 integers, subtracts one from each, and returns them - do it in two different ways. [Note, there are 2 ways to "return" values from a function - what are they?]
- Write a function a function that takes in a sentence and returns that sentence without vowels in it. [Note: technically, the returned sentence is a new sentence.]
- Write a 'for each' loop that turns all numbers in a list into zero.
- Make sure you are comfortable writing each of the (shorter) functions from the labs and assignments.

Wednesday’s Assignment(s)

- Code Review
- Lab – Number Representations