CSC161 2011S *Imperative Problem Solving*

Class 14: Binary Representation and Bitwise Operators

Held: Tuesday, 15 February 2011

Summary: We begin to study binary representation, focusing on representations of integers.

Related Pages:

- EBoard.
- Lab: C's Bitwise Operations.
- Reading: K&R 2.9, 6.9 ; Wright: Tutorial on Binary Numbers.

Notes:

- I will reserve time at the start of class for questions on the exam.
- Please start Firefox/Iceweasel *now*.
- Good questions on the reading. I've done my best to answer the ones that I received. I've also incorporated many of them into today's "lecture".
- An "eyes closed" question: Does anyone ever read my answers?
- Missing questions from: AA, ME, DF, MH, JJ, JTL, SR.
- EC: Thursday's CS Extra.
- EC: Friday's CS Table.
- EC: ReNew fundraiser Thursday 7 pm in ARH 302.

Overview:

- Why study underlying representations?
- Basics of binary.
- Unsigned integers.
- Signed integers.
- Some of C's bitwise operations.
- Bit fields.
- Optional lab.

Why Study Representations

- As you'll note, we have a few classes devoted to underlying representations of a variety of types of numbers.
- Why do we study these issues in this course?
- As you've noted, C makes some assumptions that you understand the underlying representations.
  - Key types like `short`, `long`, and more.
  - Bitwise operations
- Successful programming in C requires you to understand these underlying representations.

- Some of the most important:
  - Unsigned integers
  - Signed integers
  - IEEE floating-point numbers.
  - Characters (ASCII and Unicode)

Binary

- On most computers, the smallest unit of information is the *bit*, which has only two possible values: off/on, 0/1, false/true, whatever.
- We choose two possible values because it's easy to represent two values in circuitry. (1 is "on"; 0 is "off").
- We combine bits into reasonable groups, such as the *byte* and *word*.
  - On most computers, a byte is 8 bits and a word is big enough to hold an address in memory.
    - On 32-bit computers, a word is 32 bits.
    - On 64-bit computers, a word is 64 bits.
- Clearly, we need ways to interpret sequences of bits.
- The interpretation is just that: An agreed-upon way to understand the meanings of the bits.
  - Common interpretations are encoded in most hardware.
- Generally, we have rules for interpreting bit sequences as integers, and then rules for interpreting other values in terms of integers.
  - E.g., characters
- For floating-point numbers, we have a different representation.

Unsigned Integers

- Base two numbers. Nothing more, and nothing less.
- Practice!

Signed Integers

- First problem: How to represent the sign.
- Typical solution: Use the leftmost bit to indicate sign.
  - 0 means "positive"
  - 1 means "negative"
- Next problem: How does one interpret the remaining bits?
- Many possible options. Here are four of the most common.
  - "Normally". The remaining N-1 bits are simply an unsigned integer.
    - Formal term: *Signed magnitude*
  - "Backwards". For the remaining bits, 0 represents a negative 1, 1 represents 0.
    - Formal term: *One's complement*
  - "Encoded". To represent signed N in k bits, we write unsigned $N+2^{k-1}$.
    - Note that in this system, a leading 0 means "negative" and a leading 1 means "positive".

        - This system is called *Excess 2^{m-1}*
    - "Just plain weird": We think procedurally. To negate a number, we flip all the bits and add 1.
        - This system is called *Two's complement*
- Exercise: Let's try a few numbers in 5 bit notation.
- What criteria might one use to decide which one to use?
    - Ease of interpreting numbers.
    - Ease of adding numbers.
    - Ease of negating
    - Ease of subtracting
    - Range of numbers representable
    - Others ...
- We'll try each of these

Bitwise Operations in C

Logical

- & - bitwise "and"
    - 0 and 0 is 0
    - 0 and 1 is 0
    - 1 and 0 is 0
    - 1 and 1 is 1
- | - bitwise "or"
    - 0 or 0 is 0
    - 0 or 1 is 1
    - 1 or 0 is 1
    - 1 or 1 is 1
- ~ - bitwise "not"
    - not 0 is 1
    - not 1 is 0
    - Why is this different than negate?
- We can use these to extract bits from an integer.
    - To access the kth bit of i, compute 2^k and and it with i
    - If the result is 0, the bit was 0. If the result is non-zero (true, in C), the bit was 1.
- We can use these to change bits in an integers
    - To change the kth bit of i, compute 2^k and or it with i.
- We often use integers to store a variety of flags (settings)
    - One bit per flag
    - If the bit is on, the flag is set
    - If the bit is off, the flag is not set

Shifting

- << - left shift
- >> - right shift
- Lots of variants.

Bit Fields

- Many programs have a variety of “flags” that they need to pass around.
  - E.g., What operations are permitted on this file?
- For efficiency, we only want to use as much space is necessary for those flags.
- Bit fields are a way of treating an integer as an array of bits for just this purpose.
- We need to decide what bit serves what purpose.
- We use various operations to do the computation.

Lab

- I’ve prepared a Lab.
- This is an optional lab. We probably won’t have time for it.
