Winter 2021 CS 33 Lecture Notes

Kyle Chui

2021-3-24

Contents

1 Bits and Bytes

1.1 Representing Data in Bits and Bytes

1.1.1 Everything is Bits

- Every bit is either a 0 or a 1.
- We can use sets of bits to not only tell the computer what to do (give it instructions), but also represent data.
- We use bits because they are easy to store, and reliably transmitted on noisy/inaccurate wires.

Example. *Counting in Binary*
We can use bits to represent numbers in base 2, or the binary number system. Thus we can use a set of bits to encode any number we want.

1.1.2 Encoding Byte Values

Definition. *Byte*
A *byte* is 8 bits, and can be thought of as a string of 0's and 1's of length eight.

- In binary, a byte can range from 00000000_2 to 11111111_2.
- In decimal, a byte can range from 0_{10} to 255_{10}.
- In hexadecimal (base 16), a byte can range from 00_{16} to FF_{16}.

  Note. In hexadecimal, once you finish using the digits 1 through 9, you use the letters A through F to represent values of 10_{10} through 15_{10}. In C, we append "0x" before a string to indicate that it is a hexadecimal number (which may either be uppercase or lowercase). For example, FA1D37B_{16} would be written as "0xFA1D37B" or "0xfa1d37b".

All data is just a long string of bits, so any value that we get out of it depends on the *context* of what we are reading in (a `double`, `char`, `int`, etc).

1.2 Bit Manipulation

1.2.1 Boolean Algebra

Developed by George Boole in the 19th century, *Boolean Algebra* is an algebraic representation of logic, where 1's denote a "true" value and 0's denote a "false" value. Some common logical operations are described in the tables below:

$$\underbrace{\begin{array}{c|cc} \& & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}}_{\text{“and” operator}} \qquad \underbrace{\begin{array}{c|cc} | & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}}_{\text{“or” operator}} \qquad \underbrace{\begin{array}{c|c} \sim & \\ \hline 0 & 1 \\ 1 & 0 \end{array}}_{\text{“not” operator}} \qquad \underbrace{\begin{array}{c|cc} \hat{} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}}_{\text{“xor” operator}}$$

Note. The "xor" operator stands for "exclusive-or", meaning "either one or the other, but not both".

All of these operations are applied "bitwise" on bit vectors—the jth bit of the result is obtained by applying the operation on the jth bit of the input(s).

Example. *Representing an Manipulating Sets*

Representation

- A width w bit vector can represent a subset of $\{0, \dots, w-1\}$.
- We do this by letting $a_j = 1$ if j is in our set.
- For example, the bit vector 01101001 would represent the set $\{0, 3, 5, 6\}$. We read from right to left, and we see that the 0th, 3rd, 5th, and 6th entries contain 1's.

Operations

- There are nice parallels between operations on bit vectors and operations on sets, i.e.
  - & on bit vectors is the same as set intersection.
  - | on bit vectors is the same as set union.
  - ˆ on bit vectors is the same as the symmetric difference.
  - ~ on bit vectors is the same as taking the complement of a set.

1.2.2 Bit-Level Operations in C

The four boolean operations covered thus far are available in C, and can apply to any "integral data type"(long, int, char, etc). The operators view the arguments as bit vectors and are applied bit-wise.

1.2.3 Logic Operations in C

Note. These are different than the bit-level operations, so don't get them confused.

There are also the logical operators in C, namely `&&`, `||`, `!`.

- These view 0 as "false", and anything non-zero as "true".
- They always return 0 or 1, and can terminate early.

1.2.4 Shift Operations

The shift operator allows you to "move" the bits in a bit vector to give them higher or lower significance.

- The left shift($x \ll y$) shifts the bit vector x to the left by y positions (extra bits on the left are tossed out and empty slots are filled with 0's).
- The right shift($x \gg y$) shifts the bit vector x to the right by y positions (extra bits on the right are tossed out).
  - In a logical shift, the empty slots are filled with 0's.
  - In an arithmetic shift, the empty slots are filled with duplicates of the most significant bit on the left.
- It is undefined behaviour if the shift amount is negative or greater than or equal to the string size.