CSE 351

Number Representation & Operators

Section 2

January 14, 2016
Number Bases

- Any numerical value can be represented as a linear combination of powers of base $n$, where $n$ is an integer greater than 1.
- Example: decimal ($n=10$)
  - Decimal numbers are just linear combinations of 1, 10, 100, 1000, etc.
  - E.g.: $1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$
- We can also use the base $n=2$ (binary) or $n=16$ (hexadecimal).
Binary Numbers

- Each digit is either a 1 or a 0
- Each digit corresponds to a power of 2
- Why use binary?
  - Easy to physically represent two states in memory, registers, across wires, etc.
  - High/Low voltage levels
  - This can scale to much larger numbers by using more hardware to store more bits
Decimal to Binary Conversion

To convert the decimal number $d$ to binary, do the following:

1. Compute $(d \mod 2)$. This will give you the lowest-order bit.
2. Divide $d$ by 2, round down to the nearest integer, and continue the process to get the higher order bits.

**Example:** Convert $25_{10}$ to binary.

<table>
<thead>
<tr>
<th>Bit</th>
<th>$d$ (mod 2)</th>
<th>$(d \div 2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>First bit</td>
<td>25 % 2 = 1</td>
<td>(25 / 2) = 12</td>
</tr>
<tr>
<td>Second bit</td>
<td>12 % 2 = 0</td>
<td>(12 / 2) = 6</td>
</tr>
<tr>
<td>Third bit</td>
<td>6 % 2 = 0</td>
<td>(6 / 2) = 3</td>
</tr>
<tr>
<td>Fourth bit</td>
<td>3 % 2 = 1</td>
<td>(3 / 2) = 1</td>
</tr>
<tr>
<td>Fifth bit</td>
<td>1 % 2 = 1</td>
<td>(1 / 2) = 0</td>
</tr>
</tbody>
</table>

Since we hit 0, we’re done! $25_{10} = 11001_2$. 
Hexadecimal Numbers

- Same concept as decimal and binary, but the base is 16
- Why use hexadecimal?
  - Easy to convert between hex and binary (1 hex digit represents 4 bits)
  - Much more compact than binary
  - And best of all, you can make fun words with letters A-F!
Decimal to Hexadecimal Conversion

To convert a decimal number to hexadecimal, use the same technique we used for binary, but divide/mod by 16 instead of 2.

- Hexadecimal numbers have a prefix of “0x”

*Example*: Convert $1234_{10}$ to hexadecimal

First digit: \[1234 \% 16 = 2\] \[\frac{1234}{16} = 77\]

Second digit: \[77 \% 16 = 13_{10} = D_{16}\] \[\frac{77}{16} = 4\]

Third digit: \[4 \% 16 = 4\] \[\frac{4}{16} = 0\]

We’re done because we hit 0!

$1234_{10} = 0x4D2 = 4D2_{16}$
Representing Signed Integers

- Two common ways:
  - **Sign & Magnitude**
    - Use 1 bit for the sign, remaining bits for magnitude
    - Works OK, but:
      - There are 2 ways to represent zero (−0 and 0)
      - Arithmetic is tricky (4 − 3 ≠ 4 + (−3))
  - **Two's Complement**
    - For positives, similar to regular binary representation
    - But, highest bit has a negative weight
    - Solves Sign-and-Magnitude’s problems!
Two’s Complement

- This is an example of the range of numbers that can be represented by a 4-bit two’s complement number.
- An \( n \)-bit, two’s complement number can represent the range \([-2^{n-1}, 2^{n-1} - 1]\).
  - Note the asymmetry of this range about 0 – there’s one more negative number than positive.
- Note what happens when you overflow.
- If you still don’t understand it, speak up!
  - Very confusing concept.
Understanding Two’s Complement Numbers

- Understanding the two’s complement representation of integers is much like converting binary to decimal, but with one catch.
- Because of the cyclic nature of two’s complement, you **subtract** the value of the most significant bit.

*Example:* In 8-bit land, what signed int does 0b11010110 represent?

- We follow the same process as binary to decimal conversion. The less significant 7 bits give us:
  \[(1 \times 2^1) + (1 \times 2^2) + (1 \times 2^4) + (1 \times 2^6) = 86\]
- Now, however, we **subtract** the highest bit value.
  \[86 - (1 \times 2^7) = -42_{10}\]
Understanding Two’s Complement
[A Handy Trick]

- There’s a simpler way to find the value of a two’s complement number, using the handy formula:
  \[ \sim x + 1 = -x. \]
- We can rewrite this as \( x = \sim(-x - 1) \), i.e. subtract 1 from the given number, and flip the bits to get the positive portion of the number.

**Example**: \( \text{0b11010110} \)

- Subtract 1: \( \text{0b11010110} - 1 = \text{0b11010101} \)
- Flip the bits: \( \text{0b00101010} = (32+8+2)_{10} = 42_{10} \)
- So the original number we had was \(-42_{10}\).
Bitwise Operators

- **NOT:** ~
  - This will flip all bits in the operand

- **AND:** &
  - This will perform a bitwise AND on every pair of bits

- **OR:** |
  - This will perform a bitwise OR on every pair of bits

- **XOR:** ^
  - This will perform a bitwise XOR on every pair of bits

- **SHIFT:** <<, >>
  - This will shift the bits right or left
    - logical vs. arithmetic
Logical Operators

- **NOT: !**
  - Evaluates the entire operand, rather than each bit
  - Produces a 1 if \( \equiv 0 \), produces 0 if nonzero
- **AND: &&**
  - Produces 1 if both operands are nonzero
- **OR: ||**
  - Produces 1 if either operand is nonzero
Common Operator Uses

- A double bang (! !) is useful when normalizing values to 0 or 1
  - Imitates Boolean types
- Shifts are useful for multiplying/dividing quickly
  - Most multiplications are reduced to shifts when possible by GCC already
  - When writing assembly routines, shifts will be more useful
  - Shifts are also consistent for negative numbers (thanks to sign extension)
- DeMorgan’s Laws:
  - \( \sim (A \lor B) \equiv (\sim A \land \sim B) \)
  - \( \sim (A \land B) \equiv (\sim A \lor \sim B) \)
Masks

- These are usually strings of 1s that are used to isolate a subset of bits in an operand
  - Example: the mask 0xFF = ...0011111111 will “mask” the first byte of an integer
- Once you have created a mask, you can shift it left or right
  - Example: the mask 0xFF << 8 will “mask” the second byte of an integer
- You can apply a mask in different ways
  - To set bits in x, you can do $x = x \mid \text{MASK}$
  - To invert bits in x, you can do $x = x \wedge \text{MASK}$
  - To erase everything but the masked bits in x, do $x = x \& \text{MASK}$
Masks

• These are usually strings of 1s that are used to isolate a subset of bits in an operand
  • Example: the mask \( 0xFF = \ldots 0011111111 \) will “mask” the first byte of an integer

• Once you have created a mask, you can shift it left or right
  • Example: the mask \( 0xFF \ll 8 \) will “mask” the second byte of an integer

• You can apply a mask in different ways
  • To set bits in \( x \), you can do \( x = x | \text{MASK} \)
  • To invert bits in \( x \), you can do \( x = x \oplus \text{MASK} \)
  • To erase everything but the masked bits in \( x \), do \( x = x \& \text{MASK} \)
Application: Symmetric Encryption

- This is an example that shows how XOR can be used to encrypt data.
- Say Alice wishes to communicate message \( M \) to Bob
  - Let \( M \) be the bit string: \( 0b11011010 \)
- Both Alice and Bob have a secret cipher key \( C \)
  - Let \( C \) be the bit string: \( 0b01100010 \)
- Alice sends Bob the encrypted message \( M' = M \ ^ \ ^ C \)
  - \( M' = 0b10111000 \)
- Bob applies \( C \) to \( M' \) to retrieve \( M \), since \( (M ^ C) ^ C = M \)
  - \( M' ^ C = 0b11011010 \)
- XOR ciphers are not very secure by themselves, but the XOR operation is used in some modes of AES encryption
Application: Gray Codes

- Gray Codes encode numbers such that consecutive numbers only differ in their representations by 1 bit
  - Useful when trying to transfer counter values across different clock domains (common in FIFOs)
  - If each wire represents one binary digit, we want to ensure that when the counter increments, the voltage level changes only on one wire

- Let \( n \) be our counter output
  - \((n >> 1) \ ^ \ ^ \ n\) will produce a gray coded version of \( n \)

- If we receive the gray code \( g \), we need to convert it to \( n \):

```c
for (int mask = g >> 1; mask != 0; mask >> 1) {
    g = g ^ mask;
}
```

For an example, compile and run `gray_code.c`
Lab 1

- Worksheet in class
- Tips:
  - Work on 8-bit versions first, then scale your solution to work for 32-bit inputs
  - Save intermediate results in variables for clarity
  - SHIFTING BY MORE THAN 31 BITS IS UNDEFINED! This will not yield 0.
- Any questions for the good of the class?
Example Problems

- Create 0xFFFFFFFF using only one operator
  - Limited to constants from 0x00 to 0xFF
  - Naïve approach:
    $0xFF + (0xFF \ll 8) + (0xFF \ll 16) \ldots$
  - Smart approach:
    $\sim0x00 = 0xFFFFFFFF$
Example Problems

• Replace the leftmost byte of a 32-bit integer with 0xAB

• Let our integer be \( x \)

• First, we want to create a mask for the lower 24 bits of the image
  • \( \sim(0xFF << 24) \) will do that using just two operations
  • \((x \& \text{mask})\) will zero out the leftmost 8 bits

• Now, we want to OR in 0xAB to those zeroed-out bits

• Final result:
  \[(x \& \text{mask}) \mid (0xAB << 24)\]

• Total operators: 5!