CSE 351

Number Representation
Number Bases

• Any numerical value can be represented as a linear combination of powers of 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
  • \(1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1\)
• 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
Converting Binary Numbers

• To convert the decimal number $d$ to binary, do the following:
  • Compute $(d \% 2)$. This will give you the lowest-order bit
  • Continue to divide $d$ by 2, round down to the nearest integer, and compute $(d \% 2)$ for successive bits
• Example: Convert 25 to binary
  • First bit: $(25 \% 2) = 1$
  • Second bit: $(12 \% 2) = 0$
  • Third bit: $6 \% 2 = 0$
  • Fourth bit: $3 \% 2 = 1$
  • Fifth bit: $1 \% 2 = 1$
  • Stop because we reached zero
Hexadecimal Numbers

• Same concept as decimal and binary, but the base is 16

• Why use hexadecimal?
  • Easy to convert between hex and binary
  • Much more compact than binary
Converting Hexadecimal Numbers

• 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 to hexadecimal
  • First digit: \((1234 \% 16) = 2\)
  • \(1234 / 16 = 77\)
  • Second digit: \((77 \% 16) = 13 = D\)
  • \(77 / 16 = 4\)
  • Third digit: \(4 \% 16 = 4\)
  • \(4 / 16 = 0\)
  • Stop because we reached zero
  • Result: 0x4D2
Representing Signed Integers

• There are several ways to represent signed integers
  • 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)
    • Also, arithmetic is tricky
  • Two’s Complement
    • Similar to regular binary representation
    • Highest bit has negative weight rather than positive
    • Works well with arithmetic, only one way to represent zero
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 $[-2^{(n-1)}, 2^{(n-1)})$.
  - Note the asymmetry of this range about 0.
- Note what happens when you overflow.
- If you still don’t understand it, speak up!
  - Very confusing concept.
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 Operators

• **NOT**: !
  - Evaluates the entire operand, rather than each bit
  - Produces a 1 if != 0, produces 0 otherwise

• **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:
  - ~(A|B) == (~A & ~B)
  - ~(A&B) == (~A | ~B)
Masks

• These are usually strings of 1s that are used to isolate a subset of bits in an operand
  • Example: the mask 0xFF 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 $x = x \^ \text{MASK}$
  • To erase everything but the desired 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 $M' = M \oplus C$
  • $M' = \quad 0b10111000$
• Bob applies $C$ to $M'$ to retrieve $M$
  • $M' \oplus C = \quad 0b11011010$
• 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 the output of a counter across clock domains using a multi-wire bus
  • 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 $n$:

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

• This will produce the original output of the counter as a binary number
Application: Bit Packing

• Let’s say you have the values x, y, and z that take 3, 4, and 1 bit to represent, respectively.

• If you store each using an int, that will result in 4*3 = 12 bytes of space.

• In C, we can pack them all together to only take up 1 byte of space.
  • Note, our implementation will end up getting padded to 4 bytes by the compiler.

• Download pack.c from the website to take a look at a simple bit packing example.