CSE 351

Section 2: Integer representations, two’s complement, and bitwise operators
Integer Representations

- Decimal (No prefix)
  - 369845, -513
  - 564U (unsigned)
  - printf("%d", x);
- Binary ("0b" prefix)
  - 0b110100101011011001111110011010
  - No printf() formatter 😞
- Hexadecimal ("0x" prefix)
  - 0xFF18AB07
  - printf("%x", x);
- What are some tradeoffs between these representations?
Binary Numbers

- Each digit is either 1 or 0
- Each place is a power of 2
  - 1’s place, 2’s place, 4’s place, 8’s place, etc.
  - Analogous to 10’s place, 100’s place in Decimal

- Converting to Decimal (unsigned):
  - take the sum of the nth digit multiplied by $2^{n-1}$
  - Example: $0b110101 = ?$
Converting Binary Numbers

- To convert from decimal to binary, use a combination of division and modulus to get each digit, tracking the remainder.
- Divide by $2^{n-1}$, then mod 2 for $n^{th}$ digit.
- Subtract each result from the original number until 0 remains.
- Example: $11 = 0d$??
  - First step:
    - $(11 / 2^0) \% 2 = 1$, so the first digit is $0b1$. Remainder is $11 - 1 \times 2^0 = 10$.
- Answer:
  - $0b1011$
Hexadecimal Numbers

- Each digit ranges in value from 0x0 (zero) to 0xF (fifteen)
  - A => ten, B => eleven, C => twelve, D => thirteen, E => fourteen, F => fifteen
- To convert from (unsigned) hexadecimal to decimal notation, take the sum of the nth digit multiplied by 16^(n-1)
  - Example: 0xACE = ?
Converting Hexadecimal

- Use the same division/modulus method as before, substituting 16 for 2

- Example: $3254 = 0x??$
  - First step: $(3254 / 16^0) \% 16 = 6$, so first digit is $0x6$. Remainder is $3254 - 0x6 \times 16^0 = 3248$

- Answer: $0xCB6$
Signed Integer Representation

- **Sign Bit**
  - Negative numbers have 1 as their Most Significant Bit
- **Problems?**
  - Two zeros
  - Arithmetic difficult
Signed Integer Representation

• One’s complement
  • $\sim x = -x$

• Problems?
  • Two zeros
Signed Integer Representation

- Two’s complement
  - MSB represents $-2^{w-1}$
  - Negative/Positive wraps around
    - Why is this important for a programmer to understand?
- How do we represent -13 using 4 bits?
Operator Review

• ~ is arithmetic not (flip all bits)
  • Example: ~0b1010 = 0b0101

• ! is logical not (1 if 0b0, else 0)
  • Example: !0b100 = 0, !0b0 = 1

• & is bitwise and
  • Example: 0b101 & 0b110 = 0b100

• | is bitwise or
  • Example: 0b101 | 0b100 = 0b101

• >> is bitwise right shift
  • Example: 0b1010 >> 1 = 0b1101, 0b0101 >> 1 = 0b0010

• << is bitwise left shift
  • Example: 0b1010 << 1 = 0b0100, 0b1000 << 1 = 0b0000
Notes about Operators

• Analogous first order logic operators:
  • ~ => \( \neg \)
  • | => V
  • & => \( \land \)

• DeMorgan’s Laws Work:
  • \( \neg(A \lor B) = (\neg A \land \neg B) \)

• ! is not bitwise
  • Any non-zero value evaluates to 0, else 1
  • How can this be used to create a function that takes an int and returns 0 if the input is 0, else 1?
Notes about Operators

- $\ll$ is sort of like multiplication
  - $0x1 \ll n = 2^n$
  - $0x3 \ll n = 3 \times 2^n$

- $\gg$ is an arithmetic shift
  - If MSB is 1, then new MSBs are also 1.
  - The opposite of an arithmetic shift is a logical shift

- Why is arithmetic shifting useful?
Application: Packing and Unpacking

• Let’s say that you have values x, y, and z that take 3, 4, and 1 bit to represent, respectively
• Is there a way to store these three values using only eight bits?
• In C, we can define a struct that specifies the width in bits of each value
  • …though the compiler will add padding to make the struct a certain size if you don’t do so yourself
• In Java, there are no structs, and we have to use bitwise operators
Application: Packing and Unpacking

#include <stdio.h>

typedef struct {
    int x : 3;
    int y : 4;
    int z : 1;
    int padding : 24;
} Flags;

int main(int argc, char* argv[]) {
    Flags flags = {3, 8, 1, 0x8fffff};
    printf("sizeof(flags) is %ju and it stores 0x%x\n",
           sizeof(flags), *(int*) &flags);
    return 0;
}
Application: Packing and Unpacking

// Pack some values into a byte
byte bitValue = 0;
bitValue |= 3;
bitValue |= 8 << 3;
bitValue |= 1 << 7;

// Unpack the values from the byte
byte x = bitValue & 0x7;
byte y = bitValue & 0x78;
byte z = bitValue & 0x80;

// Alternatively, we could have shifted a particular
// mask instead, e.g. (0x1 << 7) instead of 0x80
Masks

- Strings of 1s that allow us to extract/change select parts from a bitstring
- Example: \(0b11111111 = 0xFF = 255\)
- Card example:
  - How could we use a mask to get just the suit?
Using a Mask

• Mask $\ll$, $\gg$ number
  • Moves the location of the mask
  • Beware of arithmetic right shift

• Mask $|$ number
  • Sets bits in selection to 1

• Mask $\&$ number
  • Copies bits in selection

• Mask $^\wedge$ number
  • Inverts bits in selection ($1 \Rightarrow 0$)
Lab 1

• Two big obstacles:
  • No numbers bigger than 255
  • No “-” operator

• How do we create arbitrary numbers?
  • Answer: complement and shift operations

• How do we subtract?
  • \( \sim x + 1 = -x \)
  • \( z - x = z + (\sim x + 1) \)
Number Exercises

- Create -1 without using “~” or “-”
- Create 24 using no constants greater than 5
- Create the largest positive 32-bit integer
Mask Exercises

• Generate a mask for the leftmost n bits
• Generate a mask for the rightmost n bits
• Generate a mask for a string of n bits at position p (p is 0 indexed, assume n and p are reasonable for a 32 bit word)
More Exercises

• Replace the leftmost byte of a 32 bit integer with 0xAB
• Clear (set to 0) the middle 12 bits of a 32 bit integer
• Count the number of nonzero bytes within a 32 bit integer (i.e. bytes that are not 0x00)
• Swap the first and last bytes of a 32 bit integer