CSE 351: The Hardware/Software Interface

Section 2

Integer representations, two’s complement, and bitwise operators
Introduction

- CE ugrad (SP14) / 5th year CSE Masters student
  - Computer architecture, HW/SW Interface, digital design
  - SAMPA – Approximate Computing / NPU

- Experience
  - TA for CSE 351 (WI13) and CSE 352 (AU13)
  - Amazon
  - Lockheed Martin Aeronautics

- OH: Wed 2:30-3:20 in CSE 002, or by appointment
  - Contact: discussion board or email (wysem@cs)
In addition to decimal notation, it’s important to be able to understand binary and hexadecimal representations of integers.

- Decimal: 3735928559
  - No prefix, just the number
- Binary: 0b11011110101011011011111011101111
  - “0b” prefix denotes binary notation
- Hexadecimal: 0xDEADBEEF
  - “0x” prefix denotes hexadecimal notation

Which notation is the most compact of the three? Why use one over another?
Binary scale

- Each digit in binary notation is either 0b0 (zero) or 0b1 (one)
- To convert from (unsigned) binary to decimal notation, take the sum of the $n$th digit multiplied by $2^{n-1}$
  - As an example, $0b1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13$
To convert from decimal to binary, use a combination of division and modulus to get each digit, tracking the remainder.

As an example, let’s convert 11 to binary:

- \( (11 / 2^0) \% 2 = 1 \), so the first digit is 0b1. Remainder is \( 11 - 1 \times 2^0 = 10 \).
- \( (10 / 2^1) \% 2 = 0 \% 2 = 1 \), so the second digit is 0b1. Remainder is \( 10 - 1 \times 2^1 = 8 \).
- \( (8 / 2^2) \% 2 = 4 \% 2 = 0 \), so the third digit is 0b0. Remainder is \( 8 - 0 \times 2^2 = 8 \).
- \( (8 / 2^3) \% 2 = 1 \% 2 = 1 \), so the fourth digit is 0b1.

Finally, we have that 11 is 0b1011 in binary.
Hexadecimal scale

* 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 \( n \)th digit multiplied by \( 16^{n-1} \)
  * As an example, 0xACE = 0xA * 16^2 + 0xC * 16^1 + 0xE * 16^0 = 10 * 256 + 12 * 16 + 14 = 2766
Hexadecimal scale

- The decimal to hexadecimal conversion is the same process as decimal to binary except with 2 instead of 16.
- As an example, let’s convert 3254 to hexadecimal:
  - \((\frac{3254}{16^0}) \mod 16 = 6\), so the first digit is 0x6. Remainder is 3254 - 0x6 * 16^0 = 3248.
  - \((\frac{3248}{16^1}) \mod 16 = 203 \mod 16 = 11\), so the second digit is 0xB. Remainder is 3248 - 0xB * 16^1 = 3248 - 176 = 3072.
  - \((\frac{3072}{16^2}) \mod 16 = 12 \mod 16 = 12\), so the third digit is 0xC.
- Finally, we have that 3254 is 0xCB6 in hexadecimal.
- If we were to write a program to convert from decimal to binary or to hexadecimal, how could we compute the \(n\)th digit efficiently using bitwise operators and modulus (%)?
Two’s complement review

In class, we established that two’s complement is a nice format for representing signed integers for a couple different reasons. What were they?
Two’s complement review

Let’s say that we want to encode -5 in binary using two’s complement form and four bits

-5 = -8 + 2 + 1
= 1 * -2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0
= 10b1011

5 = 4 + 1
= 0 * -2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0
= 0b0101
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`
Operator uses

☆ Can express negation in terms of arithmetic not and addition
  ☆ For example, \(~4 + 1 = \sim0b0100 + 1 = 0b1011 + 1 = -5 + 1 = -4\)

☆ Can use shifting, bitwise and, and logical not to detect if a particular bit is set
  ☆ As a simple example, \(!!(x \& (0x1 << 1))\) evaluates to 1 if the second bit is set in \(x\) and 0 otherwise
  ☆ Useful for checking if a value is negative

☆ Can implement ternaries \((x = \_\_ ? \_\_ : \_\_)\) using bitwise and, bitwise or, and arithmetic not
  ☆ This has wide-ranging applications in lab 1
Bitwise operators in practice

* Is what we’re learning ever useful in practice?
  * Thankfully (or not, depending on how you look at it), it is
* Setting bits in permission strings
  * For example, to choose the permissions for `chmod` using octal codes
  * `chmod 744 <file> = chmod u+rwx,g+r,o+r`
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.
Packing and unpacking (C)

#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, 0x8ffffff};
    printf("sizeof(flags) is %ju and it stores 0x%x\n", sizeof(flags), *(int*) &flags);

    return 0;
}

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

★ Decompose each problem into smaller problems
★ If you are stuck on how to solve something, write it as a combination of functions and boolean logic
★ Over time, replace each function or boolean operator with a combination of permitted operators
★ Hint for detecting overflow: what is the sign of the integer produced by adding TMax to a positive value? What about when adding negative numbers?
★ Hint for counting bits: consider multiple bits at once. 40 operations isn’t enough to check each individually