CS240 Review Notes

Table of Contents

1 Lecture 1 - Introduction **1**
1.1 Computers 1
1.2 Machine code and Assembly 1
1.3 Bits 2
1.4 Hexadecimal 3

1 Lecture 1 - Introduction

Welcome to CS241 - Foundations of Sequential Programs. The official summary of what you should learn in this course is the following:

DEFINITION
Course description

The relationship between high-level languages and the computer architecture that underlies their implementation, including basic machine architecture, assemblers, specification and translation of programming languages, linkers and loaders, block-structured languages, parameter passing mechanisms, and comparison of programming languages.

Shortly put, we will learn to write a program that reads a program and then outputs a program. (3meta5me)

For the duration of this course we will assume that computers exist, that is the actual physical parts of them. For instance, we will not worry ourselves with the physical parts or the way that computers process information to its physical parts at a very low level.

1.1 Computers

A computer really only serves two purposes. To **store** data and to **manipulate** data. Therefore, a computer is at its very core, has only a CPU and RAM (memory). The CPU is in charge of manipulation and RAM in charge of storing. They are connected bidirectionally.

The data that computers manipulate are called bits.

1.2 Machine code and Assembly

DEFINITION
Bit

A **bit** (short for **bi**nary digi**t**) is a unit of information expressed as either a 0 or a 1 in binary notation (base 2).

Obviously, no one wants to stare at endless streams of zeros and ones, for this reason we have created programming languages. In a computer, programs are just a large pile of bits, this pile has many meanings, but those meaning are interpreted based on the eye of the beholder. The goal of this course is to bridge the gap between computers and high-level programming languages.

Therefore we will create programs that turn programs written in a programming language into bits, simple enough. But what is the program that turns the programming language into bits written in? To end this cycle, we have machine code.

DEFINITION
Machine code

Machine code is a computer programming language consisting of binary or hexadecimal instruction that a computer can respond to directly. These are however, not the same across all platforms. Some CPUs expect the machine language *x86* whereas others could expect *ARM* machine code, among hundreds of different variations. For this class, we will focus on MIPS.

Still though, machine code is by no means legible. For this reason, we define an **assembly language**. You can think of this as one step up from machine code. Generally there is a one-to-one correspondence between the language and the architecture's (read x86, ARM, MIPS) machine code instructions.

1.3 Bits

I've been talking about sequences of bits for some time now, but let's delve a little deeper into them.

A bit can only have one of two positions, 0 or 1. However, putting bits in a sequence allows us to do interesting things, like count. For instance, if we had 4 bits in a row, how many different ways could we arrange the string? Well, the first character can be either 0 or 1, so two ways, the second is also two ways. Following this pattern we see that we can arrange a sequence of 4 bits in 2^4 different ways. Extending this for a sequence of n length, we have 2^n possible arrangements, which grows incredibly fast.

From this, we can get numbers. Say we had 4 bits and we let each one denote a power of two: 2, 4, 8 and 16. And then we create the arrangement 0011. We can see that this would equal 3, $0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0 = 3$. Well this is all fine and dandy, but how do you get a negative number?

Turns out there are multiple ways of doing this, but we will see **two's compliment**, the most popular.

DEFINITION
Two's Compliment

Two's compliment gives us an algorithm to assign positive and negative values with unsigned integers. The algorithm is as follows:

1. Write the unsigned number
2. Flip all the bits
3. Add 1

Let's see a few examples to see why this works.

EXAMPLE 1

Let's assume we have a 3 bit system. In this system we have the number 010, which is 2 in decimal. Next we'll flip the bits to get 101, and then add one to get 110, which we will say is equal to -6.

1.4 Hexadecimal

Hexadecimal numbers are very similar to bits, though instead of 2 different arrangements, there are 16, represented with the numbers 0-1 and the letters A, B, C, D, E and F. Conveniently, a group of 4 bits can define a single hexadecimal character.