Lab 09

November 1, 2017

lab09.zip contains all of the starter template code for constructing your programs. Implement the functions `push_bits()` and `pop_bits()` for the first checkpoint, `bit_twiddling()` for the second, and `decode()` for the final checkpoint.

Pushing and popping bits

Using the shift left operator (`<<`), we can "append" bits onto our values by shifting and then applying an OR operation onto the new value. Left shifting a 1 onto a 0 will yield 1. Left shifting a 1 onto a 4 (`100`) will yield 9 (`1001`). Likewise, the right shift operator (`>>`) will shift values off the right (least significant) side, turning 1011 into 101 after a single right shift. Change the values you pass into `push_bits()`. Do you notice any patterns?

Bit Twiddling

Let's say you're interested in a specific range of 10 bits inside another value. How do you access them? Bit masks. A bit mask allows you to eliminate all irrelevant bits leaving behind only the bits that you care about. A very common way to create a bit mask for, say 10 bits would be to left shift 1 by 10 and then subtract one. Why? Because all that will remain will be 1's which will allow us to AND away the uninteresting values. (Try it: 1 « 10 = 1024, 1024 - 1 = 1023 and in binary, 0b1111111111). Now we can shift our values around and apply a mask to arrive at our values of interest.

Instruction decode (sort of)

Using the `bit_twiddling()` function you implemented above as well as this page of MIPS instruction formats, write the `decode()` function to test the appropriate bits of the instruction argument to determine whether it is an R-type instruction or not.