Y86 Emulator
Srihari Shankar

Algorithm

Instead of the typical switch case, I decided to go with an *array of of function pointers.* How this works is very interesting. Each Opcode is a byte long. The hi bit is the main determining factor of what our instruction is going to do. For the most part this works. But what if there are multiple opcodes per high bit, meaning, what if the low bit is not 0? Then, the method that goes in the main array will call another array of pointers according to the low bit. For example for the jump instructions the high bit is 7, so a method called jumplookuptable() will go in the main array. then the jmplookuptable() will index another array of function pointers containing the jump instructions according to the low bit. In essence calling each function is reduced to O(1) time because the operation is just an array index. Unlike the switch case which takes O(n) time because it will have to do at most n comparisons. The downside of my algorithm is that we create space complexity by creating the array. But the benefits outweigh the costs because it is a.) faster and b.) simpler to read. The speed difference may not be noticeable for y86 but for bigger emulator it will be. Overall the function pointer approach is cleaner and more efficient.

Reflections.

This assignment was really annoying to do. The hardest part was to load the stuff in memory because c is annoying with string manipulations and alot of Hacky stuff had to be done. Writing the methods was super tedious, but when prog1 and prog2 worked it was super cool to see how the machine instructions got converted to working code.

Extra Credit

I was able to write a working y86 disassembler and a prog3.y86 that did a Hello World. The disassembler will write the whole instruction meaning it will basically look like valid assembly code. So for example something like..

rrmovl esp,ebp

Essentially the disassembler will contain the whole instruction
Never again will I write stuff in hex machine instructions.