SDC Simulator

CS 350: Computer Organization & Assembler Language Programming
Lab 8, due Fri Apr 4 ~~5~~

[Fixed due date]

Note: This lab is long but important; give yourself extra time to do it.

A. Why?

- Implementing the von Neumann architecture helps you understand how it works.

B. Outcomes

After this lab, you should be able to

- Code (in C) the framework of a simulator for a simple von Neumann computer.

C. Programming Problem [100 points total]

- For this lab, you'll be implementing a version of the Simple Decimal Computer (SDC) from lecture, in C. You'll write a line-oriented program that reads in the initial memory values and then lets the user execute the program one instruction at a time.
- The SDC is a decimal computer with memory addresses `0000` – `9999`, ten general-purpose registers numbered `0` – `9`, and ten instructions. The SDC uses word addressability, with a word being 4 decimal digits (plus a sign-magnitude sign).
- The simulator can be written in two parts: First, work on reading in memory and reading the commands. Then implement the actual instructions.
- To get you started with the program, there's a partial non-working skeleton `Lab08_skel.c`. Add, change, or delete lines in the skeleton as necessary; the STUB comments can be replaced with code. You don't have to use the skeleton if you don't want to, but you should understand how it works.
- There's also sample input and output in the files `Lab08_sample.txt` and `Lab08_soln_out.txt`.

- I'll post a I'll post a sample executable solution on `alpha`. You can execute `~sasaki/Lab08_soln` .

D. Program Specification

1. Prompt for and read in the values for memory. Store the first number you read into location `00`, the second number inro location `01`, etc. Read until you see a number $>$ `9999` or $<$ `−9999`. (Each memory location is supposed to contain a value $\leq$ `9999` and $\geq$ `−9999`, so we can use values outside that range as sentinels.) Initialize the rest of memory to all zeros.

   (Use `scanf` to read each integer; that way you can enter more than one value per line when running your program.)
2. Print out the memory values and initialize the control unit (the registers, program counter, and running flag, which gets set to true).
3. Until you see a quit command
   4. Prompt for and read a command. (Read the entire line including the carriage return <cr>.) Analyze the command and do one of steps 5 – 9 below:
   5. For (command) `q`, note that you've seen a quit (to stop your loop).
   6. For `d`, dump out the control unit (program counter, instruction register, and data registers) and the memory values.
   7. For `h` or `?`, print out a help message.
   8. For a number, call the `instruction_cycle` function that many times. (If the CPU running flag is false, `instruction_cycle` should just say that the machine has halted.)
   9. For the null command (the input line is <cr>), call `instruction_cycle` once. (So it's an abbreviation for the numeric command `1`.)
10. After you quit the command loop, dump the control unit and memory as for the `d` command.

E. Programming Notes

- A <cr> is the character `'\n'`.
- To read in an entire line, use `fgets(`*string*`,` *size*`, stdin);` where *string* is a character array and *size* is the length of *string*. C will copy the characters into *string*, including the <cr>; it also adds the terminating `'\0'`.
  - (Since `fgets` will not overflow the buffer *string*, it's safer than routines that don't guarantee non-overflow.)
- To check the command line for a character or number, you can use `sscanf(`*string*`,` *format*`, &`*variable*`);` where *string* is the command line, format is a `scanf` format string (with `%c` or `%d`), and *variable* is a character or integer variable. The `sscanf` behaves like `scanf` but reads its input from *string* instead of `stdin`.
  - For the null command, your character variable will equal `'\n'`.
  - The `sscanf` call returns the number of items that it read (zero or one in our case). If you try to read a number and find you get zero results, then the line did not begin with an integer.
    - Like `scanf`, `sscanf` treats spaces and tabs as separators but ignores them otherwise. (For example, the strings `"q"` and `" q "` are treated the same way.)
  - You don't need to know this for this lab, but the `sscanf` format can be quite complex. For example, `sscanf(`*string*`, "argle %lf %lf", &x, &y);` would check to see if the string begins with the word `argle` followed by two double-precision floating-point numbers, which it would read into `x` and `y`. If the *string* didn't begin with <whitespace> `argle` the `sscanf` would read in zero values.

