SDC Simulator

CS 350: Computer Organization & Assembler Language Programming
Lab 5, due Fri Mar 6

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

- Complete the framework of a simulator for a simple von Neumann computer.

C. Programming Problem [100 points total]

- For this lab, you'll be implementing (in C) a version of the Simple Decimal Computer (SDC) from lecture. The goal is a line-oriented program that reads in initial memory values and then lets the user enter commands to execute the program's instructions and inspect the registers and memory.
- There will be a sample executable solution available on `alpha` as `Lab05_soln` plus an `sample.sdc` input file.
- There's a partial non-working skeleton `Lab05_skel.c` attached to this handout. Add, change, or delete lines in the skeleton as necessary; the STUB comments should be replaced with code, along with any other code you add. You don't have to use the skeleton if you don't want to, but you should understand how it works.

D. Program Specification

1. When you run the linux command for your program, you can include a command line parameter, the name of the file containing the SDC memory values to read in. E.g., you might run `./a.out my.sdc` . If the command line parameter is left out, behave as though it were

`default.sdc` . Don't forget to tell the user what file you are trying to open.

2. Open the specified SDC data file and initialize memory from it. (If you can't open the file, complain and exit the program with an error.) Treat the file as a text file containing a sequence of lines. If a line begins with a number (4 digits or fewer, possibly negative), then copy that number into the memory array; the first number goes in location `00`, the second in `01`, etc. (If the data tries to go past location `99`, warn the user and stop reading; go on to step 5.) Ignore the rest of that line so that it can be a comment for that line.
3. If the line from the SDC data file doesn't begin with a number, ignore it (without complaining to the user), even if it contains a number somewhere inside the line. This way the data file can contain blank lines and comment lines.
4. If the line from the SDC data file begins with a number that's out of range (i.e., ≥ `10000` or ≤ `-10000`), treat it as a sentinel value (a signal) to stop reading; ignore it and the rest of the data file. And of course, stop reading if you hit end-of-file.
5. At this point, you've initialized memory. Either now or before step 2 above, initialize the control unit (the general purpose registers each ← 0, the running flag ← true and the IR and PC ← 0. If you want to keep the sign of the instruction separate from the rest of the IR (your design choice), then initialize it, too.
6. So now you've initialized the entire CPU (control unit and memory). Dump (print) the contents of the CPU (everything in the control unit and all of memory). Below is some sample output from `Lab05_soln`. Your program doesn't have to duplicate the output format, but it should include all the same information.


```
> ./Lab05_soln
SDC Simulator sample solution: CS 350 Lab 5

Initial control unit:
  PC:    00  IR:  0000  RUNNING: 1
  R0:     0  R1:     0  R2:     0  R3:     0  R4:     0
  R5:     0  R6:     0  R7:     0  R8:     0  R9:     0

Initialize memory from default.sdc
Sentinel 10000 found at location 25
  00:  5178 -5278  6189 -6289  2145    1345  3345  4367  7810  7009
  10:  8112  7011 -8214  7013  9011    9199  9221  9345 -9455  9500
  20:     0    97    65    48     0       0     0     0     0     0
  30:     0     0     0     0     0       0     0     0     0     0
  40:     0     0     0     0     0       0     0     0     0     0
  50:     0     0     0     0     0       0     0     0     0     0
  60:     0     0     0     0     0       0     0     0     0     0
  70:     0     0     0     0     0       0     0     0     0     0
  80:     0     0     0     0     0       0     0     0     0     0
  90:     0     0     0     0     0       0     0     0     0     0
```

7. Now run the command loop. Prompt for and read a command (the entire line).
8. For command `q`, note that you've seen a quit command and exit the program. (Don't dump things back out, just quit.)
9. For command `d`, dump out the control unit (program counter, instruction register, and data registers) and the memory values.
10. For `h` or `?`, print out a help message.
11. If the command is just a number (let's call it N), it's the number of times we should run the instruction cycle, but before doing that, first make sure N is reasonable: If $N < 1$, complain to the user and go on to the next command; if N is unreasonably large, the user and change N to a sane limit. (See the skeleton for the limit variable.) So now run the instruction cycle N times. If the running flag is/becomes false, say that execution has halted, skip the rest of N and go onto the next command.
12. If the command was empty/a newline, run the instruction cycle once.
13. Once you finish executing the current command, prompt for another one and go back to step 8. Note you do this if even if CPU execution has halted; that way the user has the option of entering a `d` command before quitting.

E. Programming Notes

- A <cr> is the character `'\n'`.
- To read in a line of text and see what's in it, the skeleton uses a combination of `getline` and `sscanf`. The `getline` routine reads in an entire line into a buffer (including the terminating <cr>). Save what `getline` returns; it's the number of characters it read into the buffer (`-1` for end of file). Then you can use `sscanf` on the buffer to do things like look for it beginning with a number or for beginning with a character. Like `scanf`, `sscanf` returns the number of items that that particular call managed to read, so you can tell whether or not the read found what you were looking for.
- The skeleton's `initialize_memory` function is set so that it correctly calls `getline` and then `sscanf` to look for a leading number. You'll need to complete the routine so that if you find a number, you copy it to the SDC memory array (and if you don't, you ignore the line).
- Here are some technical info on how `getline` works. (You can get the program to work without know all these details.) A typical call looks like

  ```
  bytes_read = getline(&buffer, &buffer_len, datafile);
  ```

  where `buffer` is a `char *`, `buffer_len` is the size of the buffer, and `datafile` is either `stdin` (to read from the keyboard) or the `FILE *` for the SDC data file (when you're initializing memory). The `bytes_read` and `buffer_len` variables are declared to be of the integer-like type `size_t` ("size type", defined by `stdio.h`).

  The `getline` routine automatically allocates/reallocates the buffer space, which is why we don't need to initialize it to point to the beginning of a `char` array. (We initialize it to `NULL`, which means that it's not pointing to anything.) A side effect of the way `getline` controls memory is that we need to return its space back to the runtime system when we're done it; that's done by the `free(buffer);` call after the loop.

