SDC Simulator

CS 350: Computer Organization & Assembler Language Programming
Lab 6, due Wed Feb 24

Note: Lab 7 and the Final Project will extend your solution to this lab, so complete this lab even if you can't hand it in on time

Link: `Lab6_skel.c`

A. Why?

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

B. Outcomes

After this lab, you should be able to

- Initialize the simulator for a simple von Neumann computer.

C. Programming Problems

- For these labs, 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 from a file 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 `Lab6_soln` plus a `Lab6_default.sdc` input file, in the `~sasaki/CS350` directory.
- There's a partial non-working skeleton `Lab6_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. Lab 6 Programming Assignment [50 points]

For Lab 6, you should initialize the CPU, initialize memory (by reading its contents from a file) and dump (i.e., print) the contents of the CPU and memory.

1. To Initialize the CPU, set the general purpose registers, the `IR`, and the `PC` each to 0, the instruction sign to 1, and the running flag to true. The `IR` fields (`instr_sign`, `opcode`, `reg_R`, and `mem_MM`) don't get used in this lab, but it can't hurt to initialize them too.
2. When you run the linux command for your program, you can include the name of the file containing the initial values of the SDC memory. For example, 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, and if you can't open the file successfully, complain and exit the program with an error by returning 1.
3. In its simplest form, an SDC data file is a sequence of lines containing one integer per line. E.g.,

```
1234
3456
-4567
0
2568
```

   The first number goes into memory location `00`, the second into `01`, etc. Stop reading if you hit end-of-file or if you run out of memory locations (by trying to read past location `99`; warn the user if this happens). Also stop if you read in a "sentinel" value, one outside the range `-9999`...`9999`. (A sentinel value just tells you that you've run out of input.)
4. The SDC data file can contain comments, whitespace, and blank lines. If a line is blank or begins with something that isn't a number, ignore the whole line. If a line begins with a number, read that number into memory but ignore any text after it. E.g., the following input would be equivalent to the simple example above.

```
This is a comment
1234  so is this
3456  ; this too

-4567 3 5 7 (ignore the 3 5 7)
    0 note leading spaces
 2568
This line is ignored too
```

```
     12345 this sentinel tells us to stop reading
     1111 so everything after it is ignored
```

5. Dump (print) the contents of the CPU and memory Your program doesn't have to duplicate the output format of `Lab6_soln`, but it should include all the same information. For each memory address that contains a non-zero value, print the address, the value it holds, and a representation of that value as an instruction.

```
> ./Lab6_soln
SDC Simulator pt 1 sample solution: CS 350 Lab 6
Usage: ./Lab6_soln [default.sdc]

Initial CPU:
  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
Memory: @Loc, value, instr (non-zero values only):
@ 00   5178   LDM   R1, 78
@ 01  -5278   LDM   R2,-78
@ 02   6189   ADDM  R1, 89
@ 03  -6289   ADDM  R2,-89
@ 04  -2145   ST    R1, 45
@ 05   1345   LD    R3, 45
@ 06   3345   ADD   R3, 45
@ 07   4367   NEG   R3
@ 08   7810   BR    10
@ 09   7009   BR    09
@ 10   8112   BRP   R1, 12
@ 11   7011   BR    11
@ 12  -8214   BRN   R2, 14
@ 13   7013   BR    13
@ 14   9011   GETC
@ 15  -9199   OUT
@ 16   9221   PUTS  21
@ 17   9345   DMP
@ 18  -9455   MEM
@ 19   9500   NOP
@ 21   0097   HALT
@ 22   0065   HALT
@ 23   0048   HALT
```

Note: The sample solution does a very fancy job of printing out the instructions; you don't have to do this. (But you might think about how to do this because you might want to do something similar in your final project.)

- If an opcode doesn't use a register or memory field, it isn't printed out.
- For `LDM` and `ADDM`, the instruction sign is important for but it's printed out as part of the immediate value.
- The opcode for condition branch `BRC` is printed as `BRP` (branch if positive) or `BRN` (branch if negative), depending on the insttruction sign.

Again, you don't have to be very fancy with your instruction output. For example, if you want to print the value `-5278` as `-LDM R2,78` (note the minus sign), that's fine.

E. Programming Notes

- You're welcome to add functions that aren't mentioned in the skeleton, if you want. (Don't forget to add their prototypes to the top of the file.)
- To read in a line of text and see what's in it, we can't `scanf` because it ignores line ends. The skeleton uses `fgets` to read a line from the data file (as a string of characters) into a buffer; then it uses `sscanf` to do a formatted read of the buffer.
- First, the skeleton uses `fgets` to read in a line of text from the datafile into a buffer. `fgets` returns a pointer to `char`. If the pointer equals `NULL`, we hit end-of-file. (If it successfully reads data, `fgets` returns a pointer to the buffer you used.)
- The usual case is that `fgets` copies characters from the file into the buffer until it sees the end of the line (`'\n'`). (It does copy the `'\n'` into the buffer.) There's also a safety feature: We give `fgets` the length of the buffer; if `fgets` reaches the end of the buffer before seeing the `'\n'`, it stops, so as not to overrun the end of the buffer. This is a safety feature because a standard way to attack a program is to fill memory with evil code by writing way past the end of a buffer.
- After `fgets` reads characters into the buffer, we can use `sscanf` to read data from the buffer. Instead of `scanf(`*format*`, &`*var*$_1$`, …)`, which reads data from standard input, we use `sscanf(`*string*`,` *format*`, &`*var*$_1$`, …)` to read data from the string.

- Like `scanf`, `sscanf` returns the number of items that that particular call managed to read, so we can tell whether or not the read found everything. E.g., `x = sscanf(buffer, "%d %d", &y, &z);` tries to read an two integers from the string `buffer` into variables `y` and `z`. It sets `x` to `2`, `1`, or `0` depending on whether it set both `y` and `z` or just `y` or neither `y` nor `z`.
- **Note**: Just because `scanf` or `sscanf` doesn't find what it's looking for (e.g., by not finding an integer when reading with `%d`), that doesn't always mean you hit end-of-file or end-of-string; it might be that there was more input but it just didn't look like an integer.

F. Grading Guide [50 points total]

- [2 pts] Include your name and section in the program and in the program's output.
- [3 pts] Initialize the CPU.
- [4 pts] Correctly determine the name of the file to read memory values from.
- [3 pts] Open the memory file (handle errors if necessary).
- [5 pts] Read each line in the memory file; stop at EOF, sentinel, or bad memory location.
- [4 pts] Get memory value from each line and store it into memory.
- [5 pts] Ignore comments in the line; detect sentinel; detect end-of-file.
- [5 pts] Dump initial CPU using an output format that's readable.
- [10 pts] Dump memory, using an output format that's readable. (Doesn't have to be as fancy as the sample solution, just readable.)
- [5 pts] Style: Code is well-formatted and concise, variables are well-named and commented.
- [4 pts] Line or section comments are included when doing something tricky.