SDC Simulator, Part 1

CS 350: Computer Organization & Assembler Language Programming

Lab 5, due Tue Oct 11 [2 week lab]

9/26 eve: misc tweaks

Note:

- Lab 6 extends Lab 5 and the Final Project is similar to Lab 6, so get this lab to work even if you can't hand it in on time.

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 Labs 5 and 6, 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 `fourier` as `Lab5_soln` in the `~sasaki/CS350` directory.
- There's a partial non-working skeleton `Lab5_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 5 Programming Assignment [50 points]

For Lab 5, you should initialize the control unit, initialize memory (by reading its contents from a file) and dump (i.e., print) the contents of the control unit and memory.

1. To Initialize the control unit, 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 control unit and memory Your program doesn't have to duplicate the output format of `Lab5_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.

```
> ./Lab5_soln
SDC Simulator
  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 24
Memory: @Loc, value, instr (nonzero values only):
@ 00   5178   LDM   R1, 78
@ 01  -5278   LDM   R2,-78
@ 02   6189   ADDM  R1, 89
@ 03  -6289   SUBM  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   BRGE  R1, 12
@ 11   7011   BR    11
@ 12  -8214   BRLE  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
```

- The sample solution does some fancy formatting that you don't have to do, namely:
  - If an opcode doesn't use a register or memory field, it's not printed out.
  - For opcode 5 (`LDM`), negative immediate values are printed with a leading '`-`'.

- For opcode 6, `ADDM` or `SUBM` is printed depending on the instruction sign.
- For opcode 7, `BRGE` (branch if ≥ 0) or `BRLE` (branch if ≤ 0) is printed depending on the instruction 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. For opcode 6, you can print `ADDM` or `-ADDM`; for opcode 7 you can always print `BRC` or `-BRC` (branch conditional).

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. It uses a combination of `fgets` and `sscanf` instead.
  - 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**: If `sscanf` for (say) an integer fails (returns 0 as the number of data items it read), there might still be data in there; it just doesn't start with something that looks like an integer. E.g., if the input was "`xyz junk`", then scanning for a `%d` will fail because `sscanf` hits `xyz` instead of an integer.

F. Grading Guide [50 points total]

We'll compile your program on `fourier`, with `gcc -Wall -std=c89 -lm`. If we get error messages and no executable `a.out`, the program gets a score of zero.

- [2 points] Include your name and section are in the program and in the output.
- [3 points] Initialize the control unit.
- [20 points total] Initialize memory:
  - [3 points] Get the name of the file to read memory values from.
  - [3 points] Open the memory file (handle errors if necessary).
  - [3 points] Get a value from each line in the memory file; store it into memory.
  - [3 points] Ignore blank lines and lines that don't begin with a number; ignore anything that appears after the number.
  - [3 points] Stop reading the memory file at EOF.
  - [3 points] Stop reading the memory file at a sentinel.
  - [2 points] Stop reading the memory file on bad memory location.
- [5 points] Dump the control unit, using a readable output format.
- [10 points total] Dump memory, using a complete but readable output format. (It doesn't have to be as fancy as the sample solution, just complete and readable.)
  - [2 points] When dumping memory, skip the locations with a value of zero.
  - [2 points] Print the instruction as a 4-digit number with sign.
  - [3 points] Print a mnemonic opcode, the register (`R`) field, and the memory (`MM`) field. You can omit `R` or `MM` if the instruction never uses it (e.g., `HALT`). (This isn't required, however.

    - [3 points] For opcodes 5 (load immediate), 6 (add immediate), and 7 (branch conditional), the sign of the instruction is important, so include it with the mnemonic printing of the instruction. (Print a minus sign somewhere or distinguish between `ADDM/SUBM` and `BRGE/BRLE`.)
- [10 points total] Structure and comments
    - [2 points] Variables and functions are well-named.
    - [2 points] Code is well-formatted.
    - [2 points] Code is concise.
    - [2 points] There's a header comment for each function.
    - [2 points] There are comments whenever you do something tricky. (Like figuring out which mnemonic to use.)
- Code that doesn't compile earns a zero.

