Final Project: An LC-3 Simulator

CS 350: Computer Organization & Assembler Language Programming

Due Fri Apr 1 (phase 1) and Fri Apr 22 (phase 2)

[3/22 p.8]

Links: `FP1_tests.zip`, `FP2_tests.zip`

A. Why?

- Implementing a computer really helps you understand its structure.

B. Outcomes

After this project, you should

- Know how to implement the major parts of a computer in a higher-level language.
- Know how to access structures via pointers in C.

C. Project

- You are to implement of a simple console-oriented simulator for the LC-3 in C using your SDC simulator from the labs as a starting point.
- Your program should take the name of a `*.hex` file as a command line parameter, load it into memory, initialize the control unit, and run a command loop. Commands can cause program execution and other functions.
- Phase 1 of the project only needs to load the hex file into memory; phase 2 includes handling the simulator commands and executing instructions.

D. Simulating The LC-3 Architecture

- As in the SDC simulator , the LC-3 CPU should be modeled as a value of type `struct CPU`, allocated in the main program. Routines that need the CPU should be passed a pointer to the `CPU` value.
- A `Word` is `typedef`'ed to be a `short int`, 16 bits on the `alpha` machine. Since `Word` values are signed, `0x8000` – `0xffff` represent `-32768` to `-1`.

- An `Address` is an `unsigned short int` (compare with the `unsigned char` used for the SDC).
- Represent the CPU's memory as an array of `Word` values indexed by `Address` values. Represent the general purpose registers as an array `Word` values indexed by integers (0 – 7). The `PC` is an `Address`; the `IR` is a `Word`.
- Some warnings about the interaction of using unsigned and/or short integers:
  - Since unsigned values are never negative, as `Address` values, `0x8000` – `0xffff` represent `32768` to `65535`.`
  - You may need a cast to convert between `Word`, `Address`, and integer values. E.g., as a `Word`, `0xffff` is `-1`, but as an `Address`, `0xffff` is `65535`. A base register contains a `Word`; to use its value as an `Address`, you need to cast it: (`Address`) *word_value* + *offset*.
  - Loops of the form "for every `Address`, do ...." are a little tricky. If `i` is an `Address` variable, then `for (i = 0; i < 65536; i++) { .... }` will cause an infinite loop because an `Address` is always < `65536`. You an check for `i == 65535` (in which case you're one iteration short), or you can check for the second time that `i == 0` (which is a little tricky), or you can make `i` not be an `Address`.

E. Phase 1: Initialize CPU; Initialize Memory From an Input File

- As in the SDC simulator, the input file should be a command line parameter. If it's omitted, use `default.hex` as the default filename.
- You should be able to process a `*.hex` file produced by the LC-3 editor when it assembles a `*.asm` file.
  - A `*.hex` file is a text file containing a sequence of lines with a four-digit hex number on each line (no leading `x` or `0x`). An `sscanf` format of `%x` will read in an unsigned integer, which you can cast to a `Word` of memory.
  - The first line specifies the `.ORIG` location to start loading values into. The remaining lines contain the values to load.

  - If you read a value into location `xFFFF`, wrap around to `x0000` as the next location. No complaining is necessary.
  - If anything appears on a line after the four-digit hex number, ignore it. (This will let us add comments to our `hex` files.)
    - **Note**: In general, the LC-3 text editor can translate hex files to executable object code via *Translate → Convert Base 16*, but it won't handle a hex file with comments added to it.
- All other memory locations should be set to the value zero. In particular, don't simulate the `TRAP` table in low memory (`x00` – `xFF`) or the trap-handling code in upper memory.
- Once the program is read into memory, initialize the `PC` to the `.ORIG` value, the `IR` to zero, the condition code to `Z`, and the running flag to true, then dump the CPU and memory.
  - For the CPU, print out the `PC`, `IR`, `CC`, running flag, and data registers.
  - For memory, print out the memory address and its value, but **only for nonzero values**. Don't print 2^{16} lines of memory values (almost all of which are zero).
  - Print out the memory value three times: Once in hex, once in decimal, and once using assembler mnemonics. For example, if locations `x8000` – `x8002` contain `x12A0`, `0`, and `x172F` respectively, you might print

```
x8000: x12A0    4768  ADD   R1, R2, 0
x8002: x172F    5935  ADD   R3, R4, 15
```

  - (Notice that location `x8001` wasn't printed out, since it contains a zero.)
- For Phase 1 of the project, you're done!

F. Phase 2: The Command Loop and Instruction Execution

- In Phase 2, we extend the Phase 1 program by adding the command loop and execution of LC-3 instructions.
- Once the hex file is loaded into memory, display a prompt to start the command loop. The user should enter a carriage return after each command (so `h\n`, for example).

Command	Syntax	Action
Help	`?` or `h`	Print a summary of the simulator commands
Dump	`d`	Print the CPU and memory addresses/values
Execute	*integer*	Execute that many instruction cycles
Execute	(none: just the \n)	Equivalent to the integer 1 (execute one cycle)
Quit	`q`	End the simulation

Simulator Commands

Notes

- For the `d` (dump) command, print out the CPU and non-zero memory values. (Just use the same dump code you did after initializing the CPU and memory.)
- For the numeric execute command, the number of cycles to run should be a positive decimal integer. (If not, complain and go on to the next command.) If execution has already halted (the CPU running flag is false), say so and go on to the next command. If the number of cycles to run is way too large (say, > 100), complain and use 100 instead. If execution halts as you run the cycles, stop early and go on to the next command.
- The `HALT` trap stops execution but does not stop the command loop. This lets you enter `d`/`g`/`s`/`h` commands after the program finishes. Since the `g` command turns the CPU running flag on, you can use it to restart program execution after a `HALT`.

Command Format

- **Case Sensitivity**:
  - You can assume the command names `?`, `h`, `d`, `q` are in lower case.
- **Whitespace**: Let "whitespace" mean a sequence of one or more spaces or tabs.
  - Whitespace is allowed before the command; it's not allowed inside an integer.
  - Whitespace can appear before the `\n` of the empty (execute one instruction cycle) command. E.g., `\n` and `..\n` are equivalent.
  - The `sscanf` formats can take care of these requirements pretty straightforwardly. E.g., a format with `" q"` will skip over any whitespace and then look for the letter `q`.

- **Ignore Text After The Command**: Ignore any text (whitespace or non-whitespace) that appears after the command but before the end of the line. Again, the kinds of `sscanf` format shown above can take care of this without any extra work on your part.

G. Executing LC-3 Instructions — Trace of Execution

Your simulator should print a trace as execution progresses. The exact format of the trace is up to you. You're welcome to follow the sample solution, but it's not a requirement.

- Give the the instruction and its address in hex and its mnemonic representation.
- E.g., say we're executing the instruction at location `x2468`, with `M[x1234] = x6281` , then we'd name the location `x2468`, its value `x6281`, and its mnemonic representation `LDR R1, R2, 1`
- If an instruction uses or modifies a register or memory location, give the register name or memory location and its value. (Make sure the trace says whether you're using or modifying the value.)
  - Don't forget that the `PC` is a register used in `PC`-offset addressing. Also, the `PC` gets modified when you do a jump or branch.
- If an instruction uses a constant (an offset or immediate value, for example), name it.
- If an instruction calculates an intermediate value (such as address + offset), name both parts and the result.
- E.g., continuing with the example of `LDR R1, R2, 1`, if `R2 = x1234`, we would show the address calculation `x1235 = x1234 + 1` and the value of `M[x1235]` that we're copying to `R1`.

Differences From the Patt & Patel Simulator

Your simulator should produce the same results as the Patt & Patel simulator with some slight differences.

- If you execute an instruction at `xFFFF` then incrementing the `PC` should wrap it around to `x0000`. (Patt & Patel's simulator causes an error if you execute the instruction at that location.) By default, for us, `M[xFFFF] = 0` is a `NOP`.

- Executing the `RTI` instruction (`Return From Interrupt`, opcode 8) or the unused opcode `13` should print an error message but continue execution. (Patt & Patel's LC-3 simulator behaves differently.)
- Only traps `x20`, `x21`, `x22`, `x23`, and `x25` need to be implemented. For any other trap vector (including `x24`: `PUTSP`), print an error message and halt execution (set the running flag to false).
- Execute each `TRAP` command in one instruction cycle. (Don't simulate the I/O registers.) E.g., executing `PUTS` should print out the entire string pointed to by `R0`.
- For the `IN` and `GETC` traps, the user should enter a `\n` after the character to be read in. If the user just enters `\n` without a preceding character, then use `\n` as the character read in.
- Technically, the `OUT` trap is only supposed to print the right byte of `R0`, but if your `OUT` only works if the left byte of `R0` is `x00` or `xFF`, that's okay. Similarly, the `IN` and `GETC` traps are supposed to overwrite only the right byte of `R0` and leave the left byte unchanged, but if you set the left byte to `x00` or `xFF`, that's okay too.
- Because the simulator prints out a trace of execution, printing a prompt and doing a read (using `PUTS` and `GETC`) doesn't behave exactly like it does with Patt & Patel's simulator: You have to wait until the `GETC` executes and asks for your input before actually typing in the character.

H. Code Framework

- You should be able to adapt your SDC simulator to do the final project.
- Feel free to use standard library functions like `strcmp`. (Don't forget to `#include <string.h>`.)
- Remember, your program gets tested on `alpha.cs.iit.edu`, so do not use libraries like `conio` that aren't available.
- The method for representing the condition code is up to you: an integer is fine, but it could also be the rightmost three bits of an `unsigned char`, for example. Or literally the character `'N'`, `"Z'`, or `'P'`.

- Also feel free to add more fields or named constants to your program, such as `#define CC_ZERO` … to represent condition code zero.
- Random hint: If you don't already know it, look up the difference between `%x` and `%hx` in `printf` formats.

I. Sample Solutions and Test Data

- Working simulators are available on `alpha` in `~sasaki/CS350/fp1` and `~sasaki/CS350/fp2`. There's a link to some sample `*.hex` files at the top of this document, but you should create your own to help you test your program.

J. Due Dates, Collaboration, and Outside Help

- **Due Dates**: Part 1 is due Fri Apr 1; Part 2 is due Fri Apr 22.
- There will be different blackboard submissions for the phases. As usual, (for each phase), if you submit multiple times, I'll grade the last submission and toss the earlier ones.
- **Collaboration**: You can work with others on phase 1 but not phase 2.

K. Grading Scheme

Projects will be given a letter grade (with + and - variants possible). Here are descriptions for typical programs for each letter grade.

A+ Everything for an A plus extra credit.

A Program is bug-free; program structure and commenting are well-done: Each routine is cohesive (does just one thing); related routines are collected together and commented. Comments for any tricky code.

B Program loading and command loop works. Instruction execution has mostly minor bugs, maybe one major bug. Trace output missing a few details or is kind of hard to read. Program structure and commenting okay.

C Some major bugs (e.g., negative offsets or addresses ≥ x8000 fail); some output bugs (e.g., `xffffffff` for -1); execution trace missing major pieces (such as readability). Program has some structure issues (e.g., very long routines). Commenting so-so.

D Compiles without errors or serious warnings. Memory loading (phase 1) works. Command loop mostly okay (maybe missing the <nbr> *XOR* <cr> command). Missing/incorrect execution for some instructions. Trace output poor. Program not very structured; no real comments.

F Compilation causes error or generates serious warning messages.

L. Extra Credit Possibilities

- Extra credit is only available if your program works.
- Fancy program output. Some examples:
  - Initial memory dump starts at *origin* and wraps around `xFFFF` to `x0000` to *origin*`-1`. (The sample solution does this.)
  - Small values are printed in decimal; large ones in hex. (E.g., instead of `x0`, `x1`, …, some limit, print just `0`, `1`, `2`, ...
  - Print *address* instead of showing *address* `+ 0` = *address*
  - Print `ADD` calculations in decimal.
- Commands to let you
  - Dump a specified section of memory.
  - Change the value of a memory location.
  - Change the value of a register.
  - Change the PC.
  - Change the running flag back to `1` after `HALT` sets it to `0`.
  - Change the condition code.

M. What to Turn In [Added 3/22]

- Please just submit the `*.c` file to Blackboard. No need to zip it, and no need to include object files or hex files. If your program has bugs or includes extra credit, please add that information to the comments at the top of your `*.c` file.

