Final Project: LC-3 Simulator

Due Date: Friday 4/27/2018 11:59PM; No late handins

This is the final project for this course. It is a simulator for LC-3 computer from the Patt and Patel book. As you work on this project, we'll also be learning about the LC-3 architecture and how to write LC-3 assembly programs in lecture and labs. What we'll end up with here is a line-oriented command-line program that reads in initial memory contents from a file (in hexadecimal format) and then lets the user enter commands to execute the program's instructions and inspect registers and memory (much like an interactive debugger such as `gdb`). You'll also be implementing the code that executes instructions on the LC-3. After completing this project, you'll not only be able to understand how to implement a basic digital computer in software, but also how to implement a read-eval-print-loop (REPL)-based interpreter, like you are probably familiar with if you program with Python or Ruby (or other high-level scripting languages).

Your goal is to fill in all of the missing functionality in `lc3.c` which is indicated by "FILL ME IN" comments. You should add your own code in these locations. **Please do not hesitate to ask for help; this is a challenging project. You should be sure to get started early.**

Simulating the LC-3

- The LC-3 CPU should be modeled as a struct with type `cpu_t`, allocated in the main program. Routines that need the CPU should be passed a pointer to the CPU value.
- A `word_t` is `typedef`'ed to be a `short int`, 16 bits on `fourier`. Since `word_t`s are signed, `0x8000`–`0xffff` represent `-32768` to `-1`.
- An `address_t` is an `unsigned short int`.
- Make sure to read the comments on modeling instruction formats with `struct`s in C in `lc3.h`. This is a fairly advanced usage of C structs and may take some time to wrap your head around.
- The memory of the machine is represented as an array of `word_t` values indexed by `address_t`. Registers are represented as an array of `word_t` values indexed by integers (0-7 since there are 8 registers). The `PC` is an `address_t`; the `IR` is a `word_t`.
- Since unsigned values are never negative, as `address_t` values, `0x8000`–`0xffff` represent `32768` to `65535`.
- You may need a cast to convert between `word_t`, `address_t`, and integer values. E.g., as a `word_t`, `0xffff` is `-1`, but as an `address_t`, `0xffff` is `65535`. A base register contains a `word_t`; to use its value as an address, you need to cast it: `(address_t)(word_val + offset)`.

- Loops of the form "for every `address_t`, do ..." are a little tricky. If `i` is an `address_t` variable, then `for (i = 0; i < 65536; i++){...}` will cause an infinite loop because an `address_t` is always `< 65536`. You can 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_t`.

Coding Assignment (*100 points total*)

- The input file should be a command line parameter. If it's omitted, your program should produce an error.
- 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_t` to store in 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 errors are necessary here.
  - 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 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 `1`, then dump the CPU and memory.
- Once the `.hex` file is loaded into memory, a prompt will be displayed to start the command loop. The user should enter a carriage return after each command (`h\n`, for example).
- The possible commands include `?`, `h`, `d`, `q`, `N`, and an empty line, where `N` is some integer.
- You **must** be careful with whitespace. You can redirect your program's output to a file, do the same with the solution, and use the `diff` command to spot differences that aren't obvious to the eye.

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 `xFFF` 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 will 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`), an error message will be printed and execution will halt.
- A `TRAP` command will be executed in one instruction cycle. Their simulator actually simulates I/O register interactions. Because of this, we say that we are *emulating* I/O, rather than simulating.
- 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.
- 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.

Code Framework

- Feel free to use the standard library functions like `strcmp`.
  (Don't forget to `#include <string.h>`.)
- Remember, your program gets tested on `fourier`, so don't use libraries like `conio` that aren't available there.

Getting the Code Get the `lc3` project code by logging into fourier and running the following:

```
$> cp /home/khale/HANDOUT/final.tgz .
$> tar xvzf final.tgz
final/
final/tests/
final/tests/t1.exp
final/tests/t3.exp
final/tests/t8.hex
final/tests/t6.exp
final/tests/t5.exp
final/tests/t1.hex
final/tests/t7.hex
final/tests/t2.exp
final/tests/t8.exp
final/tests/t3.hex
final/tests/t2.hex
final/tests/t0.exp
final/tests/t4.hex
final/tests/t4.exp
final/tests/t10.exp
```

```
final/tests/t7.exp
final/tests/t9.exp
final/tests/t5.hex
final/tests/t6.hex
final/test_harness
final/Makefile
final/lc3.h
final/lc3.c
final/lc3-soln
$> cd final
$> ls
lc3.c  lc3.h  lc3-soln  Makefile  test_harness  tests
$> make
gcc -Wall -Wno-unused-function -Wno-unused-variable -Wno-unused-but-set-v
-Wno-packed-bitfield-compat -lm -o lc3 lc3.c
```

If you run the skeleton code you'll see the following:

```
$> ./lc3
CS 350 Final Project: LC-3 Simulator Usage: ./lc3 <.hex file>
```

Your job here is to fill in the functions in `lc3.c` which have `FILL ME IN` comments written in them. See the comments in the code for further instructions. I've also included a test harness for you to test your code. You can run it like so:

```
$> make test
# Testing lc3...
Test 0 - [FAIL]
Test 1 - [FAIL]
Test 2 - [FAIL]
Test 3 - [FAIL]
Test 4 - [FAIL]
Test 5 - [FAIL]
Test 6 - [FAIL]
Test 7 - [FAIL]
Test 8 - [FAIL]
Test 9 - [FAIL]
Test 10 - [FAIL]
0 out of 11 test cases passed
```

Notice that the code will initially fail all the test cases or hang. Your job is to make it pass them all. These are example `.hex` files, and I've included them in a separate directory named `tests`. Notice that if you add more `.hex` files in the `tests` directory this time, you will break the test harness. This is because there is something special going on in order to test your program interactively. For grading, we will be using other tests in addition to the ones in this directory, so make sure you pass all the tests we provide.

A good strategy here to get started is to start with the `main()` function and work your way down based on the functions that are called from there.

Programming Notes

- You're welcome to add functions that aren't already present in the skeleton (e.g. helper functions).
- To read in a line of text and see what's in it, we can't use `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 a line of text from the data file 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, &var1, ...)`, which reads data from standard input, we use `sscanf(string, format, &var1, ...)` 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 two integers from the string `buffer` into variables `y` and `z`. It sets `x` to 2,1, or 0, depending on whether it sets 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.

Extra Credit Opportunities

There are several possibilities for extra credit. They will involve some ambitious extension to the project or something related to it. The credit earned for such projects will vary, but can be significant for the more ambitious ones. It's really more for the fun of it. Come see me if any of these sound exciting to you.

- Extend your simulator to actually simulate I/O interactions at the register level (like the Patt & Patel simulator does.)
- Design the hardware for a floating point unit for LC-3 and add appropriate ISA extensions for it.
- Design the hardware to make the LC-3 virtualizable.
- Pick a one-instruction-set (OISC) computer of your choice and write an assembler that translates from LC-3 assembly to that OISC assembly. Some people would call such a tool a *trans-piler*.

- Write an x86 → LC-3 assembler.
- Implement LC-3 as a breadboard-based computer.
- Write a custom operating system for the LC-3.
- Add virtual memory support to the LC-3. This would, in theory, allow you to port a real-life OS like Linux to LC-3.
- Learn a hardware description language such as Verilog or VHDL and write an implementation of the LC-3. If you are *really* ambitious, and would like to synthesize your design (going from an HDL to a transistor based design using automated tools), I might even pay for you to have your chip design "taped out" (fabricated). It must, of course, work!
- Make a web-based LC-3 simulator, e.g. using Javascript, HTML5 canvases, or whatever hot web language the kids are using these days. This is something that could be used in later instantiations of this course!

Hand-in Instructions

Make sure to put your name on your submission. Submissions without names will be given zero points! For code, this means put a comment at the top of your C file with your name on it.

For the code, you must hand it in digitally. Once you're happy with your code, in the directory where your code is (`final`), run the following **on fourier**:

```
$> make handin
```

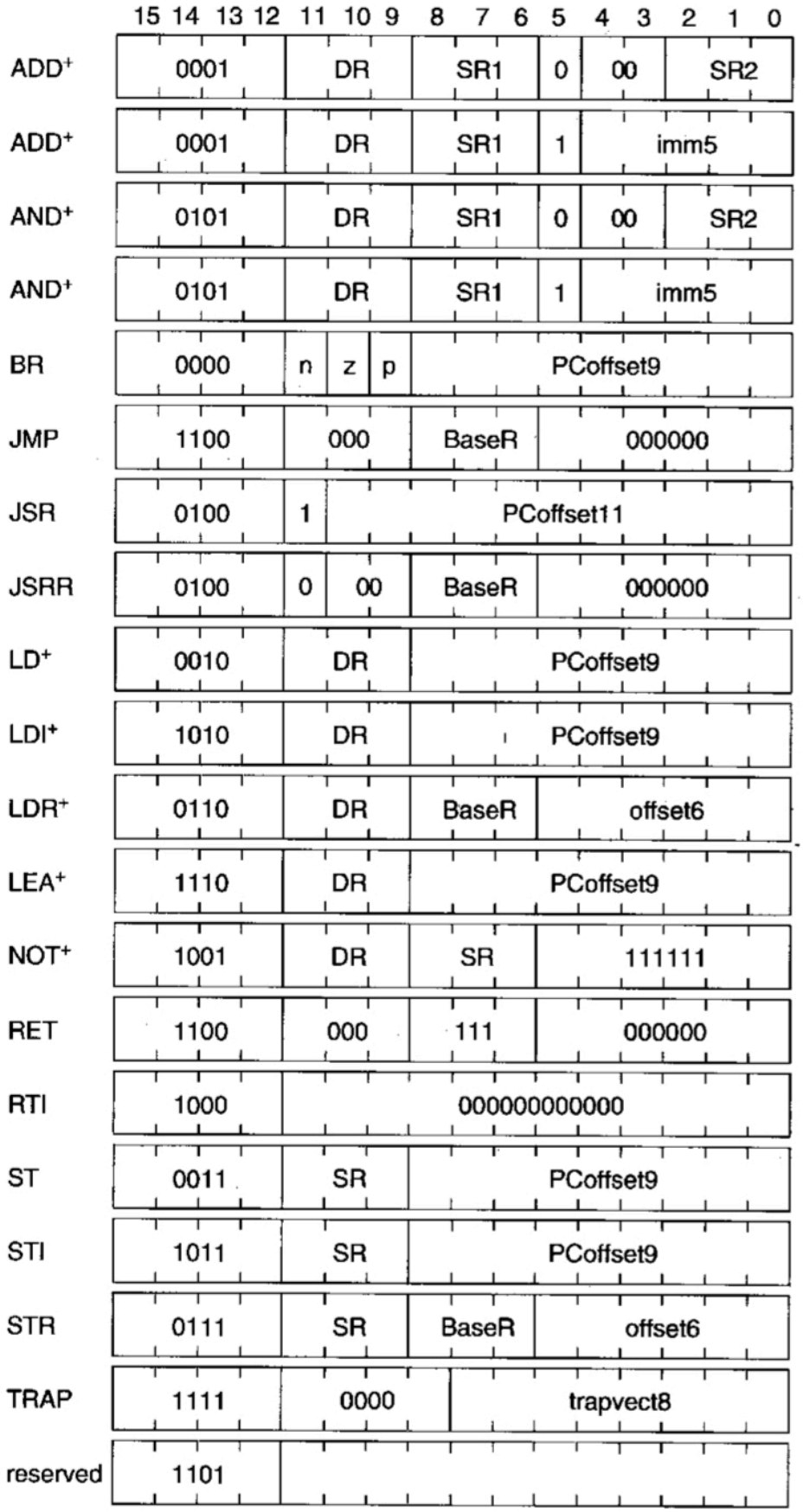
Late handins

NO LATE HANDINS FOR THIS ASSIGNMENT!

Programming Notes

- `lc3.c` contains the skeleton code for this lab.
- Remember, `sscanf` returns the number of items it was able to read. E.g., `x=sscanf(s, "%d", &y);` tries to read an integer from a string `s` into a variable `y`. If it succeeds, `x` is set to `1`, else `x` is set to `EOF` (which you should **not** assume will be zero).
- Note that this time your program is an interactive program. We are testing your program by recording interactions with the instructor solution, and making sure that your program behaves the same way. If you're curious how we do this, lookup the `expect` program. Pretty nifty.
- If, when you run `make test`, your program hangs, it is because `expect` does not understand your program's output and is waiting for the correct output. If this happens, invoke expect manually with `expect -f tests/tN.exp ./lc3` where `N` is the test number. Spot where the output stops with `expect` and look back to the previous input that `expect` typed at your prompt. The difference between the solution and your program will lie somewhere between those two points.

LC-3 ISA Reference



Note here that we will be ignoring the `RTI` instruction.