Lab 12 & Final Project: LC-3 Simulator

CS 350: Computer Organization & Assembler Language Programming

Lab 12 Due Fri Nov 22 (2400 hrs)

Final Project Due Mon Dec 2 (2400 hrs)

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 simulator for the LC-3 in C. As in the simulator you did for the Simple Decimal Computer, you will be given a skeleton of the program; your job is to complete the implementation.
- Once again, the user interface will be console-oriented. Instead of typing in the program, however, the user will be able to read the program to be executed.
- This is an individual project.
- Your program should:
  - Prompt for and read the name of an input file and then read the input file to initialize memory (see below).
  - Run a command loop to execute the program or dump registers/memory.

D. Input File

- You should prompt the user for a file name (complain if the file doesn't exist). Use `input.hex` as the default filename. (If you can't get the prompting to work correctly, at least read from `input.hex`.)

- The file should be a `*.hex` file (the kind produced by the LC-3 editor on assembling a `*.asm` file). A `*.hex` file is a sequence of lines with a four-digit hex number on each line (no leading `x` or `0x`). The first line specifies the `.ORIG` to load the program to. The remaining lines comprise the program. E.g.,

Assembler File	Hex file
`ORIG x3000`	`3000`
`LD R1, X`	`2203`
`ADD R1, R1, 1`	`1261`
`ST R1, X`	`3201`
`HALT`	`F025`
`X .FILL 17`	`0011`
`.END`	

- The rest of memory should be initialized to zeros except for the last location, which should be initialized to a `HALT` instruction. (The textbook's LC3 simulator causes an error if you execute that location.)

E. The Command Loop

- Once the program is read into memory, initialize the program counter (to the `.ORIG` value), set the `IR` to all 0's, the condition code to `Z`, and start a command loop. There are five commands. The user should enter a carriage return after each command (so `h <cr>`, for example).
- `h` for help: Print a summary of the simulator commands.
- `d` for dump control unit: Print out the program counter, instruction register, condition code, and data registers.
- `m` n_1 n_2 for memory: Print out the values of memory at locations n_1, n_1+1, ..., n_2. (If $n_1 > n_2$, then no values get printed out.) The values n_1 and n_2 should be in hex with a leading `x`. Don't print out the value of a location if the value is zero. If n_1 or n_2 is an illegal address, print an error message and skip the memory dump.

- An integer: Run that number of instruction cycles (but stop early if you execute a `HALT`). If the machine is already halted, say so but don't run any instruction cycles.
- No input, just <`cr`> (carriage return): An abbreviation for `1<cr>`
- `q` for quit: Print out the control unit and memory and end the simulation. (Note executing `HALT` stops execution but does not stop the command loop. This lets you enter `r` and `m` (and `h`) commands after the program finishes.)

Some notes on executing instructions

- Don't simulate the interrupt or `TRAP` jump tables; make those memory locations contain zeros by default.
- You should implement traps `x20`, `x21`, `x22`, `x23`, and `x25`. For `x24` (`PUTSP`) print an error message but keep executing. For any other trap vector, print an error message and halt execution.
- Execute each `TRAP` command in one instruction cycle. (Don't simulate the I/O registers.) E.g., for `TRAP x20` (`GETC`), the "Execute Instruction" part of the instruction cycle should set `R7` ← `PC`, read a character from the keyboard (in C) and copy it into `R0`.
- The `RTI` instruction (`Return From Interrupt`, opcode 8) should cause an error message and halt execution.
- The unused opcode `13` should also cause an error message and halt execution. (The textbook's LC3 simulator goes into an infinite loop.)

F. The Skeleton Implementation

- You should extend the skeleton implementation in `FP_skeleton.c`. The skeleton doesn't compile.

Words and Addresses

- The CPU memory should be represented as an array of `Word` values, indexed by `Address` values. (These are user-`typedef` defined types.)
- A `Word` is a `short int`, 16 bits on the `alpha` machine.

- An `Address` is an `unsigned short int` (16 bits on `alpha`). Note: unsigned values are always ≥ 0. As `Address` values, hex `8000` to `ffff` represent 32,768 to 65,535. As Word values, they represent -32,768 to –1.
- To convert an `int` or `Word` to an `Address`, use the cast operation `(Address)(`*e*`)` where *e* is the `int` or `Word` to convert. Similarly, to convert an `int` or `Address` to a `Word`, use `(Word)(`*e*`)`.
- Arithmetic on short integers typically yields an `int` result that you'll want to truncate using a cast. E.g., if `x` and `y` are both of type `Word`, you can add them and assign the `Word` version of the result via

```
Word result = (Word) (x+y);
```

The CPU and CPU Pointer

- As in Lab 11, the central processing unit is declared as a structure called `CPU`. The main program declares a CPU value and a pointer to it. Routines that need the CPU get passed this pointer; in the body of the routine, we dereference the pointer and access the field we want
  - In the main program:
    ```
    CPU cpu_struct, *cpu;
    cpu = &cpu_struct;
    init_CPU(cpu);
    ```
  - For initializing the CPU:
    ```
    void init_CPU(CPU *cpu) {
        cpu->pgm_counter = 0;
        …
    }
    ```

G. Sample Solution and Output

- An executable working simulator will be posted to `alpha` at `~sasaki/FP_soln`. Sample *.hex files and sample runs of them will also be posted.
- About the output: 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 on the textbook's simulator: You have to wait until the `GETC` asks for your input before typing in the character. If you type in the

character immediately after the `PUTS` executes, your simulator should read that as a simulator command (`h` for help, `q` to quit, etc).

- See the sample output's execution of `readchar.hex` for an example.

H. Lab 12: Partial Final Project

- For Lab 12, get the top-level command loop working; you should at least recognize the different commands (`q`, `r`, `m`, *integer*`<cr>,` and `h`) and call stubs of routines to handle the commands. (Think of it as like the first part of the SDC simulator from Lab 8, but with the structures and pointers from Lab 11.)

I. Due Dates

- Lab 12 is due on Mon Nov 25. You get an extension to Tue Nov 26 if you attend lab on Nov 21 or 22.
- The final project is due on Mon Dec 2. No extension on the project for attending any labs. You can turn in multiple copies of your final project; I'll just grade the last one. If you turn in a new project, submit the whole project, not just the part you changed.
- Late penalty: Turn in on Tue Nov 28 for 5% off. No submissions after that.

J. Grading Scheme

- For a grade of "A"
  - The program compiles without warnings.
  - All the simulator commands work correctly.
  - The simulated CPU has words of the right length and correctly executes instructions, including overflow (e.g., adding x8000 and 1 gives x7999).
  - The program prompts for a hex file and reads it in correctly.
  - All the opcodes work completely. To count as completely working, you should print all the trace information associated with that opcode. You can see the sample solution for details, but as examples
    - For `ADD R1, R2, R3` you specify the names `R1`, `R2`, and `R3`, the values of `R2` and `R3`, and the result value copied into `R1`.

    - For `LD` *reg*, *offset*, you say which register you're loading, the *offset* value, the `PC` + *offset* address, and the value at `M`[`PC` + *offset*] you're copying to the register.
    - The output contains all the same information as the sample solution's output. Minor differences like spacing and case and what order you print things on a line aren't relevant.
  - The code should be commented, readable, with good-sized routines (don't have just a main program hundreds of lines long; each routine should do one conceptual thing).
- For a grade of "B"
  - The program compiles to an executable, though there may be warnings.
  - The simulator commands **`h`**, **`r`**, **`m`**, <*integer*>, and **`q`** should work, though the **`m`** command might not do everything it should (check for bad addresses, avoid printing zeros, etc).
  - The simulator has words of the right length, but it may get confused about overflow and truncation from **`int`** to **`short int`** etc.
  - The simulator always reads its hex input from the file **`input.hex`** (it doesn't prompt for a name, or it handles the name incorrectly and reads from **`input.hex`** instead).
  - 75% of the opcodes should be implemented. (This can range from 75% of the opcodes completely working to all the opcodes 75% working.) Each **`TRAP`** counts as a different opcode.
  - The code is mostly commented, mostly readable, and there may be a bit of imbalance in routine sizes (but again, not just a main program hundreds of lines long).
- For a grade of "C"
  - As for a grade of "B" but only 33% – 74% of the opcodes are implemented.
  - The **`m`** command doesn't support (or ignores) the memory limit values. (But **`m<cr>`** works and prints all of memory.)
- For a grade of "D"
  - The program doesn't compile.

- Or the program executes but has trouble recognizing commands correctly.
- Or the program executes but doesn't read a hex file correctly, not even just `input.hex`.
- Or the program reads in `input.hex` but doesn't simulate the instruction cycle correctly, or it doesn't recognize instructions correctly.
- Or fewer than 33% of the opcodes work.
- Commenting is mostly incomplete or not useful.

