Final Project: An LC-3 Simulator

CS 350: Computer Organization & Assembler Language Programming

Phase 1: Due Sat Nov 19; Phase 2: Due Sat Dec 3

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 `fourier` machine. An `Address` is also 16 bits but it 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 of `Word` values indexed by register numbers (0 – 7). The `PC` is an `Address`; the `IR` is a `Word`.
- Below is a skeleton declaration for the `CPU` structure. Rather than mandate that you represent (e.g.) the condition code with a certain datatype, I've used `typedef` to give a name for the datatypes `ConditionCode`, `Opcode`, `Flag`, and `Reg` You can just replace the `???` with whatever you decide to use (like `char` or `int` etc).

```
// CPU Declarations -- a CPU is a structure with fields for the
// different parts of the CPU.
//
    typedef short int Word;               // A word of LC-3 memory
    typedef unsigned short int Address;   // An LC-3 address
    typedef ??? ConditionCode;          // Condition code: <, =, or > 0
    typedef ??? Opcode;                 // Opcode 0-15
    typedef ??? Flag;                   // Boolean flag
    typedef ??? Reg;                    // Register number 0-7

    #define MEMLEN 65536
    #define NREG 8

    typedef struct {
        Word mem[MEMLEN];
        Word reg[NREG];           // Note: "register" is a reserved word
        Address pc;               // Program Counter
        ConditionCode cc;         // positive, zero, or negative
        Flag running;             // is CPU executing instructions?
        Word ir;                  // Instruction Register
        Opcode opcode;            // opcode 0-15
    } CPU;
```

- In real life, you'd put the type definitions and similar items in a header file (`*.h`), but for this project, don't bother.

E. Interactions Between Signed/Unsigned and Short/regular integers

- Here are some warnings about the interactions between signed and unsigned and short or regular integers:
- Since unsigned values are never negative, as `Address` values, `0x8000` – `0xffff` represent `32768` to `65535`. Since `Word` values are signed, `0x8000` - `0xffff` represent `-32768` to `-1`.

- The arithmetic operators (`+`, `-`, etc.) yield regular-sized integer results, not `short` integers. If you want to cast a regular integer into a `Word` or Address, you can use a cast expression (`Word`) *regular_integer* or (`Address`) *regular_integer*.
- Similarly, if you need to take a `Word` value and treat it as an `Address`, you can cast it using `(Address)` *word_value*. (This is helpful for converting base register values to addresses in `LDR` and `STR`.)
- Loops of the form "For every `Address`, do something" 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 less than `65536`. (The `gcc` compiler will give you a warning — heed it.)
  - Various alternatives:
    - You can check for `i == 65535` (but you'll be one iteration short).
    - You can check for the **second** time that `i == 0` (which is a little tricky).
    - You can make `i` not be `short`.

F. Phase 1: Everything Except for Actual Instruction Execution

- 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. Don't simulate the `TRAP` table in low memory (`x00` – `xFF`) or the trap-handling code or memory-mapped I/O addresses in high 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
```

    - Note that location `x8001` wasn't printed out, since it contains a zero.
- Once you do the initial dump, pass control to the simulator command loop. As with the SDC simulator, you should repeatedly prompt for a command, read it in, and execute it. For Phase 1, don't actually execute the LC-3 instructions; you'll add that in Phase 2.

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 a \n)	Equivalent to the integer 1 (execute one cycle)
Quit	`q`	End the simulation

Simulator Commands

- The commands and command format are the same as in the SDC simulator: They should appear one per line, where each line is terminated by a `'\n'`.
  - You should ignore white space (spaces or tabs) before or after the command. (E.g., "` 12 \n`" and "`12\n`" should behave the same; so should "`\n`" and "` \n`".) As with the SDC simulator, reading a line using `fgets` and `sscanf` formats will take care of these requirements.
  - Ignore any text (whitespace or non-whitespace) that appears after the command but before the end of the line. Again, `sscanf` should handle this for you.
- You can assume the command names `?`, `h`, `d`, `q` are in lower case.
- For the `d` (dump) command, print out the CPU and **non-zero** memory values.
- As in the SDC simulator, for the numeric execute command, the number of cycles to execute should be a decimal integer ≥ 1 and ≤ some sanity check limit like 100. For an insanely large number, complain and substitute a sane number.
- For Phase 1, just print a message instead of executing whatever number of LC-3 instructions.

G. Phase 2: Executing LC-3 Instructions

- For Phase 2, implement execution of the different LC-3 instructions.
- If you're running a number of instruction cycles and execution halts before you run all cycles, stop early and go on to the next command. (The `HALT` trap stops execution but does not stop the command loop.)
- Print trace information as you execute instructions. You should print out as much trace information as the sample solution, but you don't have to follow its exact format.
- For each instruction you execute, print its memory location in hex and print the instruction in hex and using mnemonics.
- E.g., say we're executing the instruction at location `x2468`, with `M[x2468] = x6281` , then we'd print 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.)
    - For these purposes, the `PC` is a register when it's used in `PC`-offset addressing or 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`.

H. 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 `xFFFF`.) By default, for us, `M[xFFFF] = 0` is a `NOP`.
- Executing the `RTI` instruction (opcode 8: `Return From Interrupt`) 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`. The different `TRAP` instructions do set the condition code (see the technical footnote in the **Branch Instruction** part of Lecture 10).

- 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 sign-fill the character you read in to get 16 bits, that's okay.
- 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.

I. 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 `fourier.cs.iit.edu`, so **do not** use nonstandard libraries like `conio`.
- 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.
- When you print a negative `Word` value in hex, just print four hex digits. (So for `-1`, print `xffff`, not `xffffffff`. Hint: Look up the difference between `%x` and `%hx` in `printf` formats.

J. Sample Solutions and Test Data

- Sample data and solutions are available on `fourier` in the `~sasaki/CS350` directory: The simulators are named `fp1` and `fp2` (for Phase 1 and Phase 2); the

test data is in `fp_testdata.zip`. The `load*.hex` files are intended to test Phase 1; the others are for Phase 2. You should create your own test data to help you test your program as you write it.

K. Due Dates, Collaboration, What to Submit

- Due Dates: Phase 1 is due **Sat Nov 19**; Phase 2 is due **Sat Dec 3**. Phase 1 can be handed in late; Phase 2 can't. For both phases, there's no extension for attending lab.
- As usual, if you submit multiple times, I'll grade the last submission and toss the earlier ones.
- **Collaboration**: Allowed on Phase 1 but not Phase 2. (Working with the TAs or instructor is always okay, however.)
- **What to turn in**: Just submit the `*.c` file to Blackboard. **Don't** zip it, and **don't** include object files or hex files, etc. If your program has bugs or includes extra credit, please add that information to the comments at the top of your `*.c` file.

L. 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 or has small number of very minor bugs (e.g., spelling errors). 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 essentially 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 works. Command loop mostly okay (e.g., doesn't check for sane limits). 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.

M. Extra Credit Possibilities

- Extra credit is only available if your program works.
- **Fancy 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`, ...
  - Instead of printing "*address* `+ 0` = *address*", just print "*address*".
- **Extra Commands**
  - 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.

