CS270 Programming Assignment 6
"LC-3 Lite Simulator (C Programming)"

Program due Saturday, December 10th, 2011 (via Checkin by 4:59pm)

No late submissions are allowed for this assignment.

Assignments are to be completed individually. Discussing implementation specifics of the program is **forbidden**.

Last updated: 12/06/2011 for due date extension/clarification (submit via Checkin, not RamCT)

Goals

- Improve your C language skills (learn string and text file processing)
- Improve your understanding of the LC-3 machine architecture by building a simulator.

The Assignment

In this assignment, you will develop a "lite" version of the LC-3 simulator using C programming. Due to the scale of this project, we provided you with a large portion of the program code and a working version of the simulator executable. The simulator is an extension of the basic LC-3 machine structure you implemented in PA2 (the files provided in this project include the solution for PA2). The executable will only work on the CS department lab machines (i.e., 64-bit HP Workstations). Also, the program has not been rigorously tested, so there might be a few minor issues.

Download the executable and skeleton code from here:
http://www.cs.colostate.edu/~cs270/Assignments/PA6/PA6.PROVIDED.tar.gz

Running the Executable:
The executable is named `lc3sim-lite`. It accepts a commands file similar to lc3sim. Many of the commands supported by lc3sim are supported by this version except breakpoint manipulation and single stepping through the program. The simulator itself supports most of the LC-3 opcodes except RTI and TRAP (will bail-out on assembly programs that have these instructions). A sample test case is provided for you to test it out.

```
$ tar -zxf PA6.PROVIDED.tar.gz; cd PA6.PROVIDED
$ cd testcases; ../lc3sim-lite cmds.shifttest.txt
```

Part-A:

In the first part of the assignment, you will need to implement string tokeniznum_tokensation and parsing the LC-3 symbol table file. The function implementations are located in `infilesparser.c` (and the declarations are in `infilesparser.h`). The function signatures are as follows:

```
/**
 * Function to tokenize an input line into separate tokens
 *
 * The first argument is the line to be tokenized and the second argument points to
 * a 2-dimensional array of char (i.e., an array of strings).  The number of rows of this array should be
 * at least MAX_TOKENS_PER_LINE size, and the number of columns (i.e., length
 * of each string should be at least MAX_TOKEN_SIZE). The tokens are delimited on
 * on spaces and tabs.
 *
 * Returns 0 on success and negative value on failure
 */
```

```
static int __tokenize(char *line, char tokens[][MAX_TOKEN_SIZE], int *num_tokens);
/**
 * Get the next symbol and its associated address from the symbol table file
 *
 * First arg is a file stream pointer to the symbol table file, and
 * second arg is a 2-d array to hold the label string and the address string
 *
 * Returns 0 on success (and copies the symbol information to the 2nd arg),
 * EOF if the file has reached end-of-file or a negative value on failure
 */
int symbol_table_getnext(FILE *symfile, char label2addr[][MAX_TOKEN_SIZE]);
```

The tokenizing function splits the provided input string (argument-1) into separate tokens and stores those tokens in tokens[][] array. **The tokens are separated on space and tab characters**. The function must also trim any newline characters (or carriage return) and **must ignore empty lines**. The tokenizing function also sets num_tokens to the number of tokens found.

The symbol_table_getnext() function reads the contents of a symbol table file and returns the next label and its associated address in the second argument label2addr[][]. That is, label2addr[0] holds the label string and label2addr[1] holds the address string. To understand how to parse the symbol table file, take a look at any LC-3 symbol table file from your previous assignments. Note that this function internally calls __tokenize() to tokenize the input from the symbol table file. Look at the function commands_file_getnext() for an example on how to use the tokenizing function.

Note:
If you are unable to complete this part, you can proceed to the next part by linking your program against `infilesparser.o.PROVIDED`. In order to do that you need to modify the provided Makefile (the details are for you to determine). However you will lose points for not implementing Part-A.

Part-B:

In this part of the assignment, you will implement the LC-3 machine in `lc3sim.c`. The main program calls `lc3_evaluate_instruction()`, which fetches the next instruction, decodes it and executes the instruction. The function returns with an error if it encounters the reserved instruction or an RTI instruction. **You do not need to implement the RTI and TRAP instructions**. The function returns a special `LC3_SIM_HALT` value upon executing the halt instruction (which is really the TRAP instruction with trap vector x25). As a start, we implemented the NOT, ADD, and AND instructions and provided a general structure for `lc3_evaluate_instruction()`

Note that the file `lc3sim.h` declares several functions. However, you are free to change these functions and implement the functionality in your own way (with certain rules described below). These are provided as a guideline for you to implement the simulated machine (our implementation is based on these functions).

About Registers & Memory:

You do not have direct access to the LC-3 registers or the memory layout of the machine in the simulator. These are hidden away from you in the provided library file `lc3memlib.a`. However, we provided accessor functions to read/write from/to registers and memory. These are declared in `lc3registers.h` file. The description of these functions is as follows:

```
/**
 * Writes the contents of (argument) "word" to the register specified
 * by "register_name". Refer to lc3machine.h for the register names.
 *
 * Prints an error if an incorrect register name is specified
 */
void write_register(lc3_register_names_t register_name, lc3_word_t word);

/**
 * Reads the contents of a register specified by argument "register_name"
 * and returns the value inside that register.
 *
 * Returns a null word if an incorrect register name is specified
 */
lc3_register_t read_register(lc3_register_names_t register_name);

/**
 * Writes the contents of MDR register into the location pointed to by MAR
 *
 * Aborts the program (and prints an error) if unable to write to a location
 */
void write_memory();

/**
 * Reads contents of the location pointed to by MAR and stores it into MDR
 *
 * Prints an error message to the console and sets MDR to null word if the
 * location pointed to by MAR is not a location that has been previously
 * written into (i.e., an invalid read).
 */
void read_memory();
```

Refer to the file `lc3machine.h` for the register names. Note that read_memory() and write_memory() function read the address from the MAR (memory address register) and read-from and write-to MDR (memory data register). Before calling one of these functions, you need to make sure that the address to access is in MAR.

About Status Bits:

The LC-3 machine status bits can be accessed using the following accessor functions.

```
/**
 * Sets (to 1) the status bit specified by the argument "code" in the PSR.
 * Refer to lc3machines.h for the status codes.
 */
void set_status_code(lc3_status_codes_t code);

/**
 * Resets (to 0) the status bit specified by the argument "code" in the PSR
 */
void reset_status_code(lc3_status_codes_t code);

/**
 * Tests whether a particular status bit (specified by the argument "code") is
 * set to 1 or 0
 *
 * Returns 1 if the status bit is 1 in the PSR, otherwise returns 0
 */
int test_status_code(lc3_status_codes_t code);
```

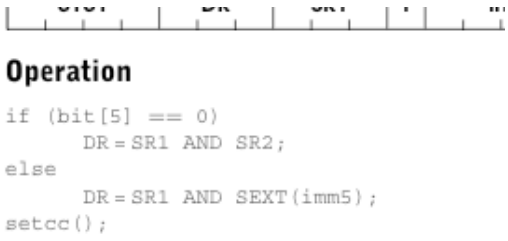
Refer to the file `lc3machine.h` for the status code names. Note that our simulator does not implement other status codes defined by the LC-3 machine (i.e., privilege mode .. etc).

About Implementing Instructions:

- The implementation must use these accessor functions to read and write to registers, memory and set/reset, test the status bits.
- The implementation must use the logic operations defined in `logic.c` to implement all functionality.
    - Note that the function `lc3_evaluate_instruction()` follows more or less the description given in Lectures 10 and 11.
- You should refer to Appendix A of the textbook to understand how to implement the individual instructions.

For example:

Look at the description for the AND instruction (from page 527 in Appendix A) from the textbook.



Operation

```
if (bit[5] == 0)
      DR = SR1 AND SR2;
else
      DR = SR1 AND SEXT(imm5);
setcc();
```

Here it says that if bit-5 of the instruction is set, then SR1 and SR2 are selected, otherwise SR1 and the sign extended register are selected. The sign extender must operate on the last five bits of the instruction. In the provide implementation, a function called `sign_extend_imm5()`performs this operation. Now look at the implementation for AND:

```
lc3_word_t          ir_  = (lc3_word_t)read_register(IR);
lc3_register_names_t dr_  = gpr_select(ir_.bit[4], ir_.bit[5], ir_.bit[6]);
lc3_register_names_t sr1_ = gpr_select(ir_.bit[7], ir_.bit[8], ir_.bit[9]);
```

First, the destination register and source register-1 are selected, and we get the contents of IR.

```
if (ir_.bit[10] == 0) {
```

Next, the instruction register's bit 5 is tested and the appropriate operation is selected. To do an immediate add:

```
  sign_extend_imm5();
  word_and(&dstw_, (lc3_word_t)read_register(sr1_),
        (lc3_word_t)read_register(SEXT));
  write_register(dr_, dstw_);
```

The function to sign extend the immediate 5-bit value is called. This function implicitly writes to the SEXT register. Then the two words are ANDed using word_and, and then finally the resulting word is stored back in the destination register. Note that word_add() accepts its parameters as lc3_word_t struct variables. Therefore we needed to use temporary variables.

Submission Instructions

Submit a tar.gz file called PA6.tar.gz with your implementation.
If you were unable to complete Part-A (i.e., tokenizing and parsing the symbol table file) and end up using `infilesparser.o.PROVIDED`, then also submit a README file describing your progress with implementing Part-A to be considered for partial credit. Do the same if you were unable to complete Part-B.

Failure to submit the README file will result in no partial credit.

Grading Criteria

This assignment will be graded in full for having the required functionality. Non-functional programs will only receive partial credit. We will award 30% for implementing Part-A, and the remaining 70% of the points will be awarded for a fully working Part-B implementation.

Note that in order to receive partial credit for non-functional programs, your code must be well documented and well structured, otherwise the partial credit will be minimal (and remember, you will not receive any partial credit if you fail to submit the README file).

Late Policy

Late submissions are not allowed for this assignment.