Department of Electrical and Computer Engineering

The University of Texas at Austin

EE 460N/382N.1, Spring 2017
Lab Assignment 2
Due: Sunday, February 19, 11:59 pm
Yale Patt, Instructor
Chirag Sakhuja, Sarbartha Banarjee, Jon Dahm, Arjun Teh, TAs

Introduction

For this assignment, you will write an instruction-level simulator for the LC-3b. The simulator will take one input file entitled isaprogram, which is an assembled LC-3b program.
The simulator will execute the input LC-3b program, one instruction at a time, modifying the architectural state of the LC-3b after each instruction.

Note: The file isaprogram is the output file from Lab Assignment 1. This file should consist of 4 hex characters per line. Each line of 4 hex characters should be prefixed with '0x'. For example, the instruction NOT R1, R6 is assembled to 1001001110111111. This instruction would be represented in the isaprogram file as 0x93BF.

The simulator is partitioned into two main sections: the shell and the simulation routines. We are providing you with the shell. Your job is to write the simulation routines.

The Shell

The purpose of the shell is to provide the user with commands to control the execution of the simulator. The shell accepts one or more ISA programs as arguments and loads them into the memory image. The address of the first ISA program listed at the command line is loaded into the PC before the simulation begins. If you compiled your simulator to an executable called "simulate," then you can run the simulator by typing the following into your terminal:

```
./simulate <main_program_file> [extra_file] [extra_file] ...
```

(Added Feb 9 2017 at 9pm)

In order to extract information from the simulator, a file named “dumpsim” will be created to hold information requested from the simulator. The shell supports the following commands:

1. go – simulate the program until a HALT instruction is executed.

2. run <n> – simulate the execution of the machine for n instructions
3. mdump <low> <high> – dump the contents of memory, from location **low** to location **high** to the screen and the dump file
4. rdump – dump the current instruction count, the contents of R0–R7, PC, and condition codes to the screen and the dump file.
5. ? – print out a list of all shell commands.
6. quit – quit the shell

The Simulation Routines

The simulation routines carry out the instruction-level simulation of the input LC-3b program. During the execution of an instruction, the simulator should take the current architectural state and modify it according to the ISA description of the instruction in Appendix A. The architectural state includes the PC, the general purpose registers, the condition codes and the memory image. The state is modeled by the following C code:

```
#define WORDS_IN_MEM    0x08000
#define LC_3b_REGS 8

typedef struct System_Latches_Struct{

  int PC,                /* program counter */
    N,                   /* n condition bit */
    Z,                   /* z condition bit */
    P;                   /* p condition bit */
  int REGS[LC_3b_REGS];  /* register file. */
} System_Latches;

System_Latches CURRENT_LATCHES, NEXT_LATCHES;

int MEMORY[WORDS_IN_MEM][2];
```

The shell code we provide includes the skeleton of a function named process_instruction, which is called by the shell to simulate the next instruction. You have to write the code for process_instruction to simulate the execution of instructions. You can also write additional functions to make the simulation modular. You should read the current system state from the global variable CURRENT_LATCHES. The results of executing the current instruction should be written to the global variable NEXT_LATCHES. (Added Feb 9 2017 at 9pm)

What To Do

The shell has been written for you. From your ECE LRC account, copy the following file to your work directory:

lc3bsim2.c

At present, the shell reads in the input program and initializes the machine state. It is your responsibility to complete the simulation routines that simulate the instruction execution of the LC-3b.

Add your code to the end of the shell code. **Do not modify the shell code.**

The accuracy of your simulator is your main priority. Specifically, make sure the architectural state is correctly updated after the execution of each instruction.

It is your responsibility to verify that your simulator is working correctly. You should write one or more programs using all of the LC-3b instructions and execute them one instruction at a time (run 1). You can use the rdump command to verify that the state of the machine is updated correctly after the execution of each instruction.

Since we will be evaluating your code on linux, you must be sure that your code compiles on one of the ECE linux machines using gcc with the -ansi flag. This means that you need to write your code in C such that it conforms to the ANSI C standard. You should also make sure that your code runs correctly on one of the ECE linux machines. Among other things, the ANSI standard forbids single line comments (// …) and requires declaring variables at the beginning of a block (e.g. for(int i = 0; i < 100; i += 1) is an illegal statement in ANSI C because of the declaration of i inside the for loop). clang on Mac OS X (clang is invoked even if you type gcc) does not comply with the ANSI standard even with the -ansi flag. You are responsible for checking for correctness on the LRC machines before submitting. **Absolutely no exceptions will be made for code that does not compile.**

If you need to copy any text files from Windows to Linux (for example, your C program or your test cases), use the dos2unix program to convert them. This program will strip away the extra '\r' end-of-line characters which are not used in Linux.

What To Turn In

Please submit your code electronically following the posted instructions. You will submit only the lc3bsim2.c file with adequately documented source code of your simulation routines.

Important

1. In Appendix A, please correct the operation of the JSR/JSRR instruction to read:

```
TEMP = PC†
if (bit(11)==0)
   PC = BaseR;
```

```
else
   PC = PC† + LSHF(SEXT(PCoffset11), 1);
R7 = TEMP;

* PC†: incremented PC
```

2. Please note that LEA **does NOT** set condition codes.
3. LC-3b registers are 16 bits wide. However, when you perform arithmetic or bitwise operations in C on int data types on the Linux x86 machines you are using 32 bits. Therefore, you must be careful about not keeping the higher 16 bits of the results in the architectural state. The shell code includes a macro called Low16bits that you can use to avoid this problem.
4. **You and your partner are allowed only one test case! (not one per partner!)**

Lab Assignment 2 Clarifications

NOTE: FAQ's for this semester will be posted here. Please check back regularly.

1. **You *must* implement the TRAP instruction as the LC-3b ISA defines in Appendix A.** However, you do not need to implement the TRAP routines. The trap vector table will be initialized to all zeroes by the shell code provided. Thus, whenever a TRAP instruction is processed, PC will be set to 0. The shell code provided will halt the simulator whenever PC becomes 0.
2. You do not have to implement the RTI instruction for this lab. You can assume that the input file to your simulator will not contain any RTI instructions.
3. For this assignment, you can assume that the programmer will always give aligned addresses, and your simulator does not need to worry about unaligned cases.
4. If you decide to use any of the math functions in math.h, you also have to link the math library by using the command: (command corrected Feb 9 2017 at 9pm)
   ```
   gcc -lm -ansi -o simulate lc3bsim2.c
   ```
5. You do not need to implement memory mapped I/O for this lab.
6. You may assume that the code running on your simulator has been assembled correctly and that the instructions your simulator sees comply with the ISA specification.
7. In your code that you write for lab 2, do not assign the current latches to the next latches. This is already done in the shell code.
8. The Low16bits macro provided in the shell code zeroes out the top 16 bits of its argument, like so:
   ```
   #define Low16bits(x) ((x) & 0xFFFF)
   ```
   You may use this macro to avoid getting values like xFFFFFFFF in architectural registers. The variables in your C program are 32-bit values, so the number -1 is xFFFFFFFF. However, the LC-3b is a 16-bit machine, in which -1 is represented as xFFFF. Thus, you have to make sure that when you store a value in a variable representing an LC-3b register, you mask it properly (for example, using the macro given above).

9. LEA should **NOT** set the machine's condition codes, despite what the current reference documents may say.
10. To use the provided assembler.linux executable, download the file from the labs page and run the command `chmod 700 assembler.linux`. After this it can be used like the assembler from lab 1.