

CSE 141L: Project in Computer Architecture

Fall 2003

Lab 2: Construct Assembler & Instruction Set Simulator

(Last Updated 10/29/03)

Reports and electronic submissions are due at the beginning of class on Friday, November 7th.

In this lab, you will design an Assembler and an Instruction Set Simulator (ISS) for the 8-bit processor architecture you developed in Lab 1. You will assemble the three programs you developed in Lab 1 using the assembler and then run the three programs on the ISS, ensuring that they function correctly. This lab also has a homework assignment as specified below. There is nothing to turn in for the homework assignment.

What you will turn in for this Lab

Written Report Only:

- Summary of your ISA from Lab 1, including any changes you might have made.
- Static and dynamic instruction count for each one of the three programs.
- Answers to the questions below.

Written Report & Electronic Turn-in:

- Three assembly files annotated with machine code generated by your Assembler, as well as the .imi files for the three programs.
- A complete instruction trace for the three programs executing correctly on the ISS.
- A memory dump of D-Mem before and after running each one of the three programs.

Electronic Turn-in Only:

- Complete code for your Assembler, including all header files.
- Complete code for your ISS, including all header files.
- A makefile for compiling your Assembler and ISS. Alternatively, you must provide instructions on how to compile these programs.

Implementation of the Assembler

You may write your Assembler in C, C++, Java or as a Perl script. For a description of the input assembly file, see the Lab 1 project specifications. The following are the requirements for the assembler:

- Your assembler executable should be named `assembler` and should be able to be invoked on the command line by itself. The only exceptions are Assemblers coded in Java, which require the Java interpreter to accompany the executable name.
- Your assembler should take two (2) command line arguments. The input assembly file is specified with the `-i` option and the output file annotated with the machine code is specified with the `-o` option.
e.g. `assembler -i mean.s -o mean.dis`
- Before running the assembler program on your assembly code, you should remove the hand generated machine code from it.

- The machine code is the hexadecimal encoding of your assembly code. This 2 digit hex code will be inserted at the beginning of each assembly instruction line in your .s file to generate the corresponding .dis file.
- The assembler should generate a corresponding .imi file by default.
 - **e.g.** For above command line, mean .imi will be generated that has only the hex machine code, formatted one instruction per line.
- The assembler should be able to parse the required fields (PC, instruction mnemonic, operand(s)), as well as the optional label and comment fields.
- Your assembler may optionally add the PC field to your assembly code in addition to your machine code, if you wish.

Your assembler must compile and run on ieng9.ucsd.edu for credit. **No exceptions.**

Implementation of the ISS

You may write your ISS in C++ or Java (C is OK too, with suitable modifications to the instructions below). You should define a class (i.e. Proc) whose members are the resources of the processor (i.e. program counter, registers, instruction memory, data memory, etc.). You should define following methods:

- **loadIMem():** Reads a text file with hex machine code for your program and loads the instruction memory (I-Mem) starting from location 0.
- **loadDMem():** Reads a text file with data for your program and loads the data memory (D-Mem) starting from location 0.
- **reset():** Initializes the internal state of your processor (e.g. sets PC to address 0).
- **runLoop():** Executes instructions. This function may take an argument for number of instructions to simulate. This will enable you to advance the simulation by a suitable number of instructions, including single stepping. HALT instruction would stop execution of the program. RunLoop() needs to perform the following steps:
 1. **Fetch:** Read 8 bits from I-Mem at location pointed to by the current value of PC.
 2. **Decode:** Analyze the fetched instruction and find opcode and other fields, such as, register(s), immediate data, branch offset, etc. Read operand(s) from register(s) (if any).
 3. **Execute:** Do the ALU operation specified by the instruction. D-Mem may be read during this step. The result of the instruction is either the output of the ALU or the data read from D-Mem. If the instruction is a control-transfer instruction (e.g. a conditional branch) determine whether it is taken or not.
 4. **Memory/Register Writeback:** Write the result produced in Execute step above to the destination specified by the instruction, which may be a register and/or D-Mem.
 5. **Set Next PC:** If the instruction is not a branch, or branch not taken, set PC to PC+1. Otherwise set PC to the target of the branch.
 6. Generate instruction trace. For each instruction, you should print the disassembled instruction, which looks exactly like an instruction in your assembly file. In addition, it shows the values read for the operands from the register file and the effective address in the case of memory load or store instruction. You may include any additional information that may be useful for you to debug the programs.
 7. Go back to step 1 and continue processing until HALT instruction is encountered.

- **dumpIMem():** Dump contents of I-Mem. It may take arguments for the start and end memory locations and may be used for debugging your program.
- **dumpDMem():** Dump contents of the D-Mem. It may take arguments for the start and end memory locations and may be used to inspect the contents of the data memory before, during and after execution of a program.
- **main():** You would make an instance of your class, load I-Mem as well as D-Mem, reset the processor and go through the run loop. At the end of the simulation, ISS should output the dynamic instruction count for the complete program. You should also dump the contents of the data memory before and after execution of the program to show that the results of the program are correct. In main(), you may want to implement a command level interface that would receive input commands from the user and carry out specific functions, such as: advance the simulation by N instructions, dump memory contents in a given range, etc.

The following are the requirements for the instruction set simulator:

- Your ISS executable should be named `iss` and should be able to be invoked on the command line by itself. The only exception is an ISS coded in Java, which requires the Java interpreter to accompany the executable name.
- Your ISS should take three (3) command line arguments. The input machine code instruction file is specified with the `-i` option, the input data memory file is specified with the `-d` option, and the output file containing a memory dump of D-Mem and the total number of instructions executed at the termination of the program with the `-o` option.
 e.g. `iss -i mean.imi -d mean.dmi -o mean.out`
- The machine code is the hexadecimal encoding of your assembly code generated by your Assembler.
- For grading purposes, your ISS must run non-interactively by default.
- Extra credit is being offered for groups that implement a high quality user-interactive mode in their ISS. Those groups choosing to implement a user-interactive mode should not make the interactive mode the default behavior. The `-u` option should be used on the command line to invoke user-interactivity.

e.g. `iss -i mean.imi -d mean.dmi -o mean.out -u`

Your ISS must compile and run on `ieng9.ucsd.edu` for credit. **No exceptions.**

Questions

The following questions are designed to assess what you have learned about the importance of developing an ISS while designing a new processor architecture. Honestly share your experiences. There is no penalty if you report problems (most students will encounter them) as long as you found a way to fix them and your code runs successfully on your ISS.

1. Were there any bugs in your machine code related to the manual conversion of the assembly code in Lab 1? Did the Assembler catch these bugs?
2. Were your assembly programs bug free? What was the nature of the bugs in your programs that you uncovered while running on the ISS?
3. Did you uncover any deficiencies in the ISA or internal resources of your processor while debugging the programs? How did you fix them?

4. What is the dynamic instruction count for each one of the three programs? How do these numbers compare with the manually determined numbers reported by you in Lab 1?
5. Based on your experience, give at least three advantages of developing an ISS while designing a new processor architecture.

Electronic Turn-in

The instructions for turning projects in electronically will be posted on the on the CSE 141L course website.

Homework: Review LogicWorks 4

In Lab 3 you will design a datapath (i.e. register file, ALU and other elements that process data) for your 8-bit processor. Review the following items in LogicWorks 4 to prepare yourself for Lab 3:

- The various components available in Standard Libraries. In particular, D flip-flops, logic gates, multiplexors, registers, adders, clocks, binary switches, binary displays, hex keypads, hex displays, etc.
- How to connect a bus to various components.
- How to define a subcircuit bottom up (i.e. create a subcircuit and then use it to define the pins on the parent symbol).
- How to simulate a circuit and generate waveforms.
- How to print a circuit and waveforms.