CSE30321 Computer Architecture I
Lab 4: Expanding an ISA and Coding in Assembly (50 points)
Assigned: September 22. Due: October 1

1 Introduction

Looking back to big pictures objectives of class, the purpose of this lab is to:

- Begin to see how the fundamental components of a modern microprocessor work together.
- Give you expertise in comparing and contrasting different computer architectures. A hardware implementation of the 6-instruction processor would be different from a hardware implementation of the MIPS datapath discussed in class. Memory is accessed in different ways, MIPS provides more instruction types ... which implies a more complex datapath, etc. These issues will all ultimately impact the time required to execute code written in a high-level language – as well as how it is executed.

2 Insight into Self Modifying Code

Background:

- Secs. 2 and 3 of the lab are designed to show you how the 6-instruction processor instruction set architecture can be used to execute more complex high-level language code – even with a limited number of instructions and a relatively simple datapath. In Sec. 2, we'll first see how more complex code can be executed with just 6-instruction. In Sec. 3 we'll quantitatively compare 6-instruction code to an equivalent MIPS version.

What do I do?

1. Open ISE and create a new project called "lab4" in an empty directory. Download and extract the zipped Verilog files from the course web page into a different directory. In ISE, add the Verilog files by clicking "Add Copy of Source..." under the "Project" menu.

   Note: This version of the processor is slightly different from the one in the previous labs. See Appendix for more details about this processor design.

2. Download the file `ArrayAccess.s` from the course web page. Convert the assembly code to machine code. We have provided a new assembly/simulator for the 6-instruction processor to aid this process. (The Java-based simulator does not work with self-modifying code. See the course web site for instructions on how to use the Linux-based assembler/simulator.)

3. Put the machine code in the memory by modifying "comb_mem.v" (values are in hexadecimal).

4. Simulate the code such that the loops are executed three times.

5. Whenever you observe changes in the register file and memory, record the time when this happens (i.e., during the execution of which instruction), the register and/or memory location, as well as the corresponding contents before and after the change.

Questions:

- (A) What does the sixth instruction try to achieve?
- (B) Based on your observations, describe what this code snippet accomplishes.

3 Writing Your Own Self Modifying Code

Background:
Here, you'll need to translate some pseudocode to (i) a set of equivalent 6-instruction processor instructions and (ii) a set of equivalent MIPS instructions. Armed with both instruction sequences, we can quantify the number of clock cycles to execute each snippet – and see how much *time* it takes to perform this function-level task given two different instruction set architectures[1].

What do I do?

1. Write an assembly program using the six instruction set discussed in class plus the instructions you have added in the previous labs. (Let's call this program `MaxFinder.s`.) The MaxFinder program steps through a *variable* length array stored in consecutive memory locations and determines which of the elements is the largest. The array is terminated by the element containing -1. The psuedocode for the MaxFinder program appears below:

```
a = [4 12 9 24 17 -1];
i = 0;
max = -1;
while(a[i] != -1)
    if(a[i] > max)
       max = a[i];
    i++;
```

2. You must use a loop structure to step through the array. To accomplish this, you should consult the code snippet which you stepped through in the previous section. Specifically, this version of MaxFinder requires self-modifying code to allow for the use of a `while` loop. Self-modifying code takes advantage of the ability to modify instructions in the memory (as you have seen in the first part of this lab). In this way a loop can be created by changing instructions to access sequential locations in memory.
3. Make sure that at the end of executing MaxFinder, the maximum value in the array is stored in register file location 0 (RF[0] = max).
4. To end a program execution cleanly, we have added a new instruction, `END`, to the ISA. The opcode for this instruction is "1111". Please use this instruction to end your program.
5. Test your MaxFinder program with the new Linux-based assembler/simulator. (Again, the Java-based simulator does not work with self-modifying code.) See the course web site for instructions on how to use the Linux-based assembler/simulator. Record the contents of the register file and memory locations at the end of the execution.

Questions:

- (C) Assume that the array contains 10 elements and the 11th one contains -1. Based on the timing data of your processor implementation (which you have collected in the previous labs), calculate how many cycles it takes to execute your MaxFinder program. You need to take any wait states in the Verilog controller into account. Show your work.
- (D) Write a sequence of MIPS instructions that will perform the functions given by the psuedo-code above:
- (E) Assuming the cycle counts associated with the MIPS multi-cycle implementation, how long would it take to execute the MIPS sequence?
- (F) Compare and contrast the 6-instruction code and the MIPS code. (Things to consider: Quantitatively, which performs better? Why? Hint: go back to CPU time equation in Lecture 17 review)
- (G) Briefly discuss the pros and cons of self-modifying code.

[1]Note: Be sure you understand what the term "Instruction Set Architecture" means in this context.

4 What To Turn In

A *typed* report that contains the following:

- The final memory contents of the new simulator for the MaxFinder program.
- Answers to Questions A-G.
- In addition, put any modified Verilog files and `MaxFinder.s` into one group member's dropbox, and note whose dropbox is used in the lab report. (Dropboxes are at /afs/nd.edu/courses/cse/cse30321.01/dropbox/your_nan

5 Useful Tips

The processor design used in this lab is based on the simple processor discussed in the textbook by Vahid but is altered such that the instruction and data memories are merged into a single memory component. Furthermore, all the registers and control logic are made to be negative clock edge triggered in order to reduce the number of clock cycles per instruction. A block diagram of the new processor design is given below. With the merged memory, a 2-to-1 MUX is introduced such that the memory address can either come from PC or from the controller directly.

The key feature of this processor is that it allows instructions to be modified in the same way as data being updated. For example, if a `store` instruction stores a register content into a memory location where an instruction resides, it essentially changes the instruction. Such a style of coding referred to as self-modifying code is one way to achieve looping through an array. Though self-modifying coding style is not widely used (especially in modern processors), processor designs with unified memory are actually used widely.

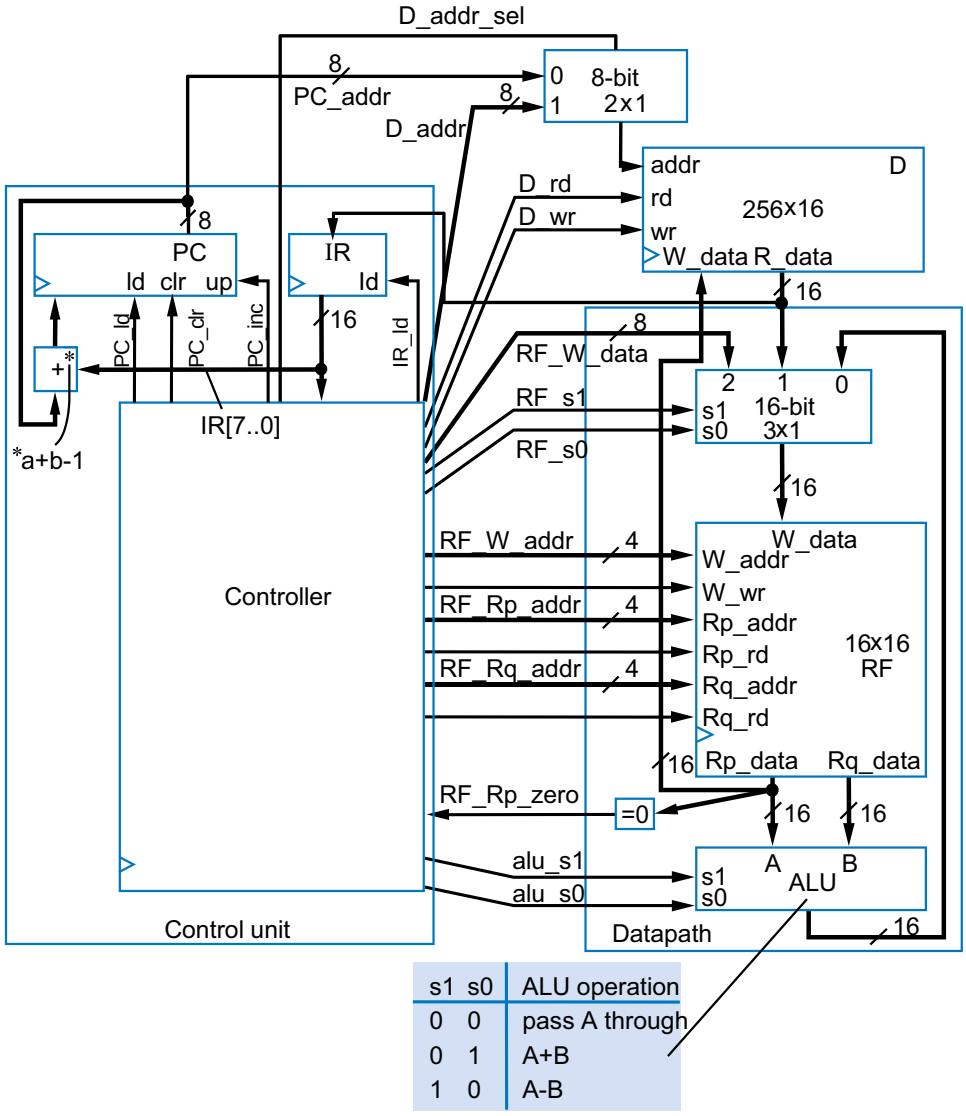

Figure 1: A simple processor with a unified memory.