CPSC 213: Assignment 4

Due: Sunday, October 10, 2010 at 6:00 PM

Goal

In this assignment you extend the SM213 implementation to support static control flow, including static procedure calls. You use this expanded machine to examine the compiler implementation of for-loops, if-then-else statements, procedure calls and return. One goal is to understand the role of pc-relative addressing, conditional and unconditional branch statements in support of high-level language constructs such as loops and if, connecting your existing understanding of these language features to what you now know about the execution of programs in hardware.

Extending the ISA

You will implement these four control-flow instructions.

Instruction	Assembly	Format	Semantics
branch	br a	8-oo	pc ← (a = pc + oo*2)
branch if equal	beq rc, a	9coo	pc ← (a = pc + oo*2) if r[c]==0
branch if greater	bgt rc, a	acoo	pc ← (a = pc + oo*2) if r[c]>0
jump	j A	b--- AAAAAAAA	pc ← A

Code Snippets Used this Week

As explained in detail below, you will use the following code snippets in this assignment. There are C, Java and SM213 Assembly versions of each of these (except the C-loop-unrolled file, for which there is no Java).

- S5-loop
- S5-loop-unrolled
- S6-if

Your Implementation

You are implementing two methods of the CPU class in the Arch.SM213.Machine.Student package of the SM213 Simulator.

1. **fetch ()** loads instructions from memory into the **instruction** register, determines their length and adds this number to the **pc** register so that it points to the *next* instruction, and then loads the various pieces of the instruction into the registers **insOpCode**, **insOp0**, **insOp1**, **insOp2**, **insOpImm** and **insOpExt** (for 6 byte instructions). The meaning of each of these registers and a primmer on the Java syntax for accessing them was given in class and is part of the online lecture slides and Companion notes. *No changes are required to your fetch stage.*
2. **execute ()** uses the register values stored by the fetch stage to execute the instructions, transforming the register file (i.e,. **reg**) and main memory (i.e., **mem**) appropriately.

Using the Simulator to Test and Debug Your Code

The simulator displays the current value of the register file, main memory and the internal registers such as the pc and instruction registers.

You will use the simulator GUI to test and debug your ISA implementation and to analyze the code snippets. In each case you should single-step through the machine code and carefully observe the state changes that occur in the process registers, register file and main memory.

Here are a few quick things that you will find helpful.

1. You can edit instructions and data values directly in the simulator (including adding new lines or deleting them).
2. The simulator allows you to place "labels" on code and data lines. This label can then be used as a substitute for the address of those lines. For example, the variables **a** and **b** are at addresses 0x1000 and 0x2000 respectively, but can just be referred to using the labels **a** and **b**. Keep in mind, however, that this is just an simulator/assembly-code trick, the machine instructions still have the address hardcoded in them. You can see the machine code of each instruction to the left of the instructions in the *memory image* portion of the instruction pane.
3. You can change the program counter (i.e., **pc**) value by double-clicking on an instruction. And so, if you want to execute a particular instruction, double click it and then press the "Step" button. The instruction pointed to by the **pc** is coloured green.
4. Memory locations and registers read by the execution of an instruction are coloured blue and those written are coloured red. With each step of the machine the colours from previous steps fade so that you can see locations read/written by the past few instructions while distinguishing the cycle in which they were accessed.

Requirements

Here are the requirements for this week's assignment.

1. Implement the four control-flow instructions one at a time. Extend your test program to include tests for each of them. Test one before starting to implement the next.
2. Compare the C versions of snippets S5 and S5a, then compare their assembly code by running them in the simulator, S5a first. Document the changes you see in memory and registers for the first few iterations of the loop and the last one.
3. Examine all versions of S6. Step the assembly version through the simulator. Document the changes you see in memory and registers.

4. Using S5 and S6 as models, write, test and document the execution of an assembly code program that implements the following C program.

```
int min=1000000;
        int i;
        int a[] = {2,40,6,80,10,120,14,16,18,20};

        void foo () {
          for (i=0; i<10; i++)
            if (min > a[i])
              min = a[i]
        }
```

Material Provided

The Simulator code was provided as part of Assignment 1 and modified in Assignments 2 and 3. Use this code as the starting point for this assignment.

This assignment includes snippets 5-6 in the file snippets-5-6.zip.

What to Hand In

Use the handin program. The assignment directory is **a4**.

1. Your modified CPU.java
2. Your test program.
3. Your test description. Did all of the tests succeed? Does your implementation work?
4. A written description of the key things you noted about the machine execution while running snippets (S5 and S6), including observations required by Questions 2 and 3. Keep it brief, but point out what each instruction read and wrote etc.
5. Your implementation of the assembly-code program required by Question 4 along with a description of its machine execution.