CPSC 213: Assignment 2

Due: Sunday, Sept 26, 2010 at 6:00pm

Goal

In this assignment you begin to implement the SM213 ISA that we are developing in class. You will implement a substantial subset of the ISA in this assignment, though its actually only 5 of instructions. The goal of this assignment is to build a processor that can execute C global scalars and arrays (both static and dynamic). You will test your implementation using the code snippets used in class and included with this assignment.

By implementing these instructions in the simulator you will see what is required to implement them in hardware and you will deepen your understanding of what a global variable is, what memory is, and the role that compiler and hardware play in implementing them. A key thing to think about while doing this is: "what does the compiler know about these variables" and so what can be hard-coded by the compiler in the machine code it generates. For global variables, recall that the compiler knows their address. And so the address of a global variable is hardcoded in the instructions that access it. But, the compiler does not know the address of a dynamic array and so, even though it knows the address of the variable that stores the array reference, it must generate code to read the array's address from memory when the program runs.

The ISA to Implement

This week, you will implement the following instructions.

Instruction	Assembly	Format	Semantics
load immediate	ld $v, rd	0d-- vvvvvvvv	r[rd] ← v
load base + offset	ld 0x0(rs), rd	10sd	r[rd] ← m[r[rs]]
load indexed	ld (rs,ri,4), rd	2sid	r[rd] ← m[r[rs]+r[ri]*4]
store base + offset	st rs, 0x0(rd)	3s0d	m[r[rd]] ← r[rs]
store indexed	st rs, (rd,ri,4)	4sdi	m[r[rd]+r[ri]*4] ← r[rs]

You will note that the *load* and *store base + offset* instructions are a bit different from the full ISA description. At the moment, for the current set of code snippets, we only need an instruction that loads/stores with memory address specified by a register. Next week we will look at implementing instance variables and we will then decide to extend these two instructions (in a simple and straightforward way). If you want, you can look ahead and implement the full

base+offset instruction (it is backward compatible with the simplified version you are doing this time). But, all we need right now is the simpler instruction and so that is all you are required to implement this week.

Code Snippets Used this Week

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

- S1-global-static
- S2-global-syn-array

Your Implementation

You are implementing a SM213 CPU by writing Java code for the SimpleMachine simulator. You loaded this simulator into your local development environment (likely Eclipse or XCode) in Assignment 1. In Assignment 1 you also implemented the MainMemory class. This class will be used automatically in Assignment 2 and subsequently. And so, if you were unable to get Assignment 1 completed, you may need some extra help this week. Get his help early.

You will implement two methods of the CPU class in the Arch.SM213.Machine.Student package.

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** 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. Note that there is another register, **insOpImm**, that we are not using until next week.
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.

And so, the first thing you might do is to create a test assembly file with various forms of the five instructions that you will implement (you can use the snippet files too, but you might want a bit more control in the early stages of debugging). Load this file into the simulator. And then, implement the first instruction's fetch stage. Step the simulator so that it executes the instruction and check to see that the instruction/pc registers have the expected value. Do this for the other instructions. You can now be confident that you've correctly implemented the fetch stage.

Now implement the execute stage for the first instruction and start again, stepping the simulator's execution through the first instruction. This time check to see whether you see the expected changes in the register file and main memory. Repeat this for the other instructions.

Be sure that this test file provides good test coverage. If it does and the tests pass, you can now reasonably believe that your CPU implementation works.

Finally, run the snippet programs and observe what happens and answer the question listed below about the C pointer math file.

You'll get help using the simulator in the labs, but 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 variable's **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 and test the five instructions listed above.
2. Write an assembly-code test file that provides good test coverage for these instructions.
3. Test your implementation using your test program.
4. Run snippets S1-global-static.s and S2-global-dyn-array.s through the simulator. Step each instruction separately. Note what happens in each step.

Material Provided

The Simulator code was provided with Assignment 1. This assignment includes snippets 1-2 in the file snippets-1-2.zip.

What to Hand In

Use the handin program. The assignment directory is **a2**.

1. 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 S1 and S2.