CPSC 213: Assignment 2

Due: Monday, January 23, 2012 at 6pm.

Late assignments are accepted until Saturday, January 28 at 6pm with a 20% penalty per day (or fraction of a day) past the due date. This rule is strictly applied and there are no exceptions.

Goal

In this assignment you will implement a significant subset of the SM213 ISA we are developing in class: the memory-access and arithmetic instructions, which are listed below.

You will then explore how these instructions are used to implement global scalars and arrays (both static and dynamic) in C by carefully examining code snippets in Java, C and assembly language. An important part of this evaluation is to carefully observe the dynamic behaviour of the assembly-language snippets by executing them in the simulator. You will develop intuition about the connection between the high-level language statements, their machine-code implementation and the execution of this code by the CPU hardware.

By implementing these instructions in the simulator you will see what is required to build them in hardware and you will deepen your understanding of what a global variable is, what memory is, and that 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.

Finally, you'll explore C's approach to arrays and pointer arithmetic by carefully examining a code snippet in C in the simulator.

Implement these SM213 Instructions

Implement the following instructions in the SM213 Simple Machine Simulator by modifying the `fetch()` and `execute()` methods of the `CPU` class. These instructions are described in more detail, including examples, in the *213 Companion*. The MainMemory class that you implemented in Assignment 1 will be used automatically in Assignment 2 and subsequently. If you were unable to get Assignment 1 completed, you may need some extra help this week. Get this help early!

Many of these instructions, but not all, are used in this week's code snippets. It is sufficient to implement (and test) only those that are required for running these snippets. By next week, you will be required to have implemented and fully tested all of this. But, if you run out of time, save this part for next week.

Memory-Access Instructions (and load immediate)

Instruction	Assembly	Format	Semantics
load immediate	ld $v, rd	0d—vvvvvvvv	r[d] ← v
load base + offset	ld o(rs), rd	1isd	r[d] ← m[i*4+r[s]]
load indexed	ld (rs,ri,4), rd	2sid	r[d] ← m[r[s]+r[i]*4]
store base + offset	st rs, o(rd)	3sid	m[i*4+r[d]] ← r[s]
store indexed	st rs, (rd,ri,4)	4sdi	m[r[d]+r[i]*4] ← r[s]

ALU Instructions

Instruction	Assembly	Format	Semantics
rr move	mov rs, rd	60sd	r[d] ← r[s]
add	add rs, rd	61sd	r[d] ← r[d] + r[s]
and	and rs, rd	62sd	r[d] ← r[d] & r[s]
inc	inc rd	63-d	r[d] ← r[d] + 1
inc addr	inca rd	64-d	r[d] ← r[d] + 4
dec	dec rd	65-d	r[d] ← r[d] - 1
dec addr	deca rd	66-d	r[d] ← r[d] - 4
not	not rd	67-d	r[d] ← ~r[d]
shift	shl $v, rd shr $v, rd	7dss	r[d] ← r[d] << ss *ss = v for left and -v for right*
halt	halt	f000	throw halt exception
nop	nop	ff00	do nothing (nop)

Code Snippets Used this Week

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
- S3-C-pointer-math

Your Instruction Implementation

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**, **insOpImm**, and **insOpExt** (for 6 byte instructions). The meaning of each of these registers and a primer on the Java syntax for accessing them was given in class and is part of the online lecture slides and Companion notes.
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.

Testing and Debugging Your Implementation

The simulator displays the current value of the register file, main memory and the internal registers such as the pc and instruction registers. Use the simulator aggressively to test and debug. If necessary, you can also set breakpoints in your CPU class, but most of the debugging can probably be done without doing this just be examining how the machine state changes and by paying careful attention to exception messages the simulator displays at the bottom of the window.

The first thing you will want to do is to create a simple test assembly file with various forms of the instructions that you implement. Use this file to test and debug each instruction in turn and then use the snippets to finish your testing. It is sufficient to just get the snippets working and to save exhaustive testing to next week.

If it helps you with your testing, the simulator has a command-line (i.e., non-GUI) interface. This would allow you, for example, to build a test script to automate regression testing.

You can invoke the command line by typing

```
java -jar yourjar.jar -i cli -a sm213 -v student
```

Then type help to see a list of commands.

Suggested Implementation Approach

The most important aspect of any strategy for implementing complicated software is to test as you go. Applied in this context, this might mean implementing each instruction in turn, first the fetch stage and then the execute stage, testing each one before moving to the implementation of the next.

You can test the fetch stage of an instruction by examining what displays in the CPU area at the bottom left of the simulator. Take the first instruction, implement its fetch, enter the instruction into the simulator, step the simulator through the instruction, and examine the value of these CPU registers. Once you've got the fetch stage right for this instruction, move to execute. Once you've implemented the execute stage for this instruction, load it into the simulator and step through it again, this time examining the state of the register file and main memory.

You will likely find that implementing and debugging the first instruction is the hardest. Once you have one working, you will see that adding others will follow a pattern that makes doing so relatively simple (compared to the first one, at least).

Once you've tested every instruction (or at least the ones needed for a snippet), then run the snippets to observe what they do.

Using the Simulator

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.
5. Instruction execution can be animated by clicking on the "Show Animation" button and then single stepping or running slowing.

Executing the Snippets

Once you have your implementation of the Simulator for these instructions working, the fun has only just begun. You now execute each of this week's snippets in the simulator to see what happens when they run. For this week, all that is required is that you single step through each of the snippets and observe what changes in the register file and/or main memory as the result of each instruction. Carefully record your observations to document what you do.

Material Provided

The file code.zip includes the Java, C and assembly versions of snippets 1-3.

What to Hand In

Use the *handin* program to hand in the following three files (which can be combined into a single zip file for using the web-based handin). Please do not hand in any other files. In particular, **do not hand in your entire Eclipse workspace, do not hand in the entire source tree for the simulator, and do not hand in any class files**.

1. A single file called "README.txt" that includes your name, student number, four-digit cs-department undergraduate id (e.g., the one that's something like a0b1), and all written material required by the assignment as listed below (items 4, 5, and 6).

2. Your CPU.java that implements the instructions listed above (or at least those that are needed to run the snippets).

3. The assembly program you used to test the instructions (a good name for it would be testlab2.s).

4. In the README.txt file, include a description of the test procedure you followed and the result. Did all of the tests succeed? Does your implementation work?

5. In the README.txt file, include a written description of the key things you noted about the machine execution while running snippets S1, S2 and S3.

6. In the README.txt file, include a careful explanation of why the C statement *c = c[3] is equivalent to c[3] = c[6] in S3-C-pointer-math using the original value of c by noting precisely what happens in the simulation for the relevant assembly instructions. In the quiz associated with this lab, you may be asked to provide a similar explanation for a similar program (but without the aid of the simulator) and so take your time here to be sure you really understand this.

   The *handin* assignment directory name is **a2**.