CPSC 213: Assignment 2

Last Modified: September 11, 2014 at 6:45pm

Due: Sunday, September 21 2014 at 11:59 pm

After an eight-hour grace period, no late assignments accepted.

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 what 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 the compiler hard-code 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 will translate two small C programs into sm213 assembly, using the simulator to help you.

Part I: Implement these SM213 Instructions

Implement the following instructions in the SM213 Simple Machine Simulator by modifying the **`execute()`** method of the **`CPU`** class.

The instructions are described in more detail, including examples, in the *213 Companion*.

Many of these instructions, but not all, are used in this week's code snippets at .

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] << s *s = v for left and -v for right*
halt	`halt`	`f000`	throw halt exception
nop	`nop`	`ff00`	do nothing (nop)

Code Snippets Used this Week

The file www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a2/code.zip contains code snippets for Part I as well as other files for Part II, described 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-dyn-array**

Your Instruction Implementation

Implement the execute() method of the CPU class in the arch.sm213.machine.student package. This method 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. The changes you need to make are indicated by "TODO" flags.

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 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. Name this file "**test.s**". Use this file to test and debug each instruction in turn. To test an instruction, write a couple of lines of assembly that use that instruction in test.s and then observe what the simulator does when you step through these instructions. For example, to test the *load-immediate* instruction, you might add lines like these to test.s:

```
ldi:  ld $0x11223344, r0
      ld $0x11223344, r7
      halt
```

Load test.s into the simulator and use its GUI to set initial values for r0 and r7, to step through these instructions, and to verify that the r0's and r7's ending values are 0x11223344 and that the values of the other registers did not change.

Alternatively, 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 SimpleMachineStudent213jar -i cli -a sm213 -v student
```

Then type **help** to see a list of commands. Note that register names are distinguished from label names by adding a percent sign to the register name; e.g., **%r0** is the name for register 0.

To run test the load instruction you might type the following (what you type is in bold):

```
% java -jar SimpleMachineStudent213.jar -i cli -a sm213 -v student
```

```
Simple Machine (SM213-Student)
(sm) l test.s
(sm) %r0=0
(sm) %r7=0
(sm) s
(sm) s
(sm) i reg
%r0:  0x11223344  287454020
%r1:  0x0         0
%r2:  0x0         0
%r3:  0x0         0
%r4:  0x0         0
%r5:  0x0         0
%r6:  0x0         0
%r7:  0x11223344  287454020
```

If you want to run it again, or when you have several tests and you want to go to a specific one, you can use the goto command with the label associated with the first instruction of that test. And, if you place a **`halt`** instruction after each test, you can use the run command instead of stepping. For example, in this case:

```
(sm) g ldi
(sm) r
```

Note that if you step past the end of the the loaded file you will get an address out of bounds exception. Note also that if you want to run the reference implementation from the command line you leave off the "-a" and "-v" command line flags and type the following instead.

```
java -jar SimpleMachine213.jar -i cli
```

Optional…

One other option is that you could download an new version of the simulator from www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a2/sm-student-213.zip to get an updated version of **`SimpleMachineStudent213.jar`** that includes a new command-line command called **`assert/a`**.

You can use this command to streamline the testing to look something like this.

```
% java -jar SimpleMachineStudent213.jar -i cli -a sm213 -v student
Simple Machine (SM213-Student)
(sm) l test.s
(sm) %r0=0
(sm) %r7=0
(sm) g ldi
(sm) r
(sm) a "ld imm"
(sm) a %r0==0x11223344
(sm) a %r1==0
(sm) a %r7==0x11223344
```

If an assertion succeeds, then it prints nothing. If it fails it prints something like this:

```
XXX ASSERTION FAILURE (ld imm): %r0 == 0x11223344 != 0x0
```

You could use this approach to run the entire test in a script and then scan its output for lines that indicate an assertion failure.

The script is simply a file that contains the commands to execute in sequence, for example, the file named **test-script** might look like this:

```
l test.s
%r0=0
%r1=0
g ldi
r
a "ld imm"
a %r0==0x11223344
a %r1==0
a %r2==0x11223344
```

You can then use the unix shell "<" operator to run the simulator with this as input.

```
java -jar Si… -v student < test-script
```

And you can then process the output to look for assertion failures by using a similar trick to pipe the output to a unix command called grep using the "|" operator like this.

```
java -jar Si… -v student < test-script | grep XXX
```

The resulting output is a list of the tests that failed.

All of this is entirely optional and it will take some work to setup, so don't worry about trying this if you aren't up for the challenge. We'll help you if you do want to try. The advantage of investing time on the setup is that testing will run a bit more smoothly for you.

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, testing each one before moving to the implementation of the next.

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.

Once you've tested every instruction, 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 this week's two snippets in the simulator to see what happens when they run. 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.

Part II: Translating C code into SM213 Assembly

Now, go back to the code file that contains the snippets. There you will find a set of example C programs and their sm213 assemble-code implementation as well as two C files named a2_1.c and a2_2.c.

Carefully example the example programs. Run the assembly versions in the simulator, step by step. Compare their execution to the C versions. Get comfortable with this translation from C to assembly and with how assembly executes in the machine.

Now, you're that you're getting comfortable, take the first of the C files that don't have assembly, a2_1.c and produce your own sm213 file that does what this program does. Every line of your assembly file must have a comment. The comment should explain what the line does by referencing C syntax whenever possible. For example, if an assembly instruction loads the value of a variable into a register, the comment should name that variable. Do the same for the other C file. The lines get progressively harder so do them in order.

Be sure to end your programs with a "halt" instruction so that the simulated CPU stops when it reaches the end of your program. If you don't do this, then it will keep running, interpreting whatever if finds in memory as instructions, probably doing very strange things.

Note that these snippets are not complete programs. Imagine that they were extracted from a larger program that did things like initialize the value of global variables etc. You should assume that pointer variables have been initialized to hold the address of an array of adequate size. That means that you need to create the array statically in the assembly and set the pointers to the static address of the array. This isn't what a real compiler would do, but by doing this you will be able to examine all of the data the program accesses in the simulator. The simulator only shows memory address that are specifically named in the assembly file.

What to Hand In

Use the **`handin`** program to hand in the following in a directory named **a2**. Please do not hand in anything that isn't listed below. In particular, **do not hand in your entire Eclipse workspace nor 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.

2. Your CPU.java that implements the instructions listed above.

3. The assembly program you used to test the instructions in a file called **`test.s`**.

4. A description of the test procedure you followed and the result. Did all of the tests succeed? Does your implementation work?

5. A written description of the key things you noted about the machine execution while running snippets S1 and S2.

6. Your implementation of test.s, a2_1.s and a2_2.s.