CPSC 213 – Assignment 2

CPU and Static Scalars and Arrays

Due: Friday, September 22nd, 2017 at 11:59pm
After an twelve-hour grace period, no late assignments accepted.

Goal

With this assignment we begin exploring assembly language, its connection to C and to the processor that implements it.

You'll begin by examining the Java/C/assembly code snippets we discuss during the lectures covering Unit 1b. The lectures showed the allocation of and access to a *global* `int` and a *global array of* `int`'s. Using this assembly code as a template, you'll write a couple of very small bits of assembly code. In each case, you'll execute this code in the reference-implementation version of the simulator (i.e., the solution) and examine carefully how its execution affects the register file and main memory.

Then, you will dig into the implementation of the processor. You will implement a significant subset of the SM213 ISA we are developing in class: the memory-access and arithmetic instructions that are listed below. You will test your implementation using the test program provided with the assignment.

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 the 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 hard-coded 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.

Examine and Write Assembly

The SM213 distribution you downloaded for Assignment 1 contains the reference implementation (i.e., solution) of the machine simulator you will build over the next few assignments. You can execute this version by clicking on the file `SimpleMachine213.jar` in file window on your machine or from the command line by typing:

```
java –jar SimpleMachine213.jar
```

The file *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a2/code.zip* contains two code snippets. We will discuss `S1-global-static` and `S2-global-dyn-array` in lecture (if all goes to plan). Your first task is to examine these files (there are C, Java and sm2130-assembly versions of each of them) and to run the assembly versions in the simulator.

To run them, click "Show Animations" to turn animation on and then click "Run Slowly" to begin the execution. The line for the next instruction that will execute is coloured green. When an instruction reads a register or memory location it turns blue and when it writes one of these it turns red. You can pause the animation by clicking "Pause" and you and restart it at any instruction by double clicking on that instruction's address. More details on how to use the simulator are provided below on the "*Some tips for using the Simulator*" Section.

Note that to begin working through this assignment, you might want to do just the first snippet now, and save the other one until the first one is complete. You can proceed to the first question without examining this second snippet.

Questions 1 and 2: Write Assembly [50% of the total marks]

1. [25%] Now for marks, using `S1-global-static.s` as a guide, implement the following C program in assembly and place the code in a file named `swap.s`. Note: just as in the snippet, ignore the procedure declaration (we will come to this later). Your program should thus just do two things: (1) allocate the global variables and (2) swap the two array elements. If you'd like, you can avoid using memory for the variable `t` by just using a register in place of the variable. Load your program into the simulator. Assign different values to the two elements of `array`, run the program to confirm that it works.

```
int t;
int array[2];

void swap() {
  t        = array[0];
  array[0] = array[1];
  array[1] = t;
}
```

2. [25%] Now lets bring in the math instructions. You'll find a complete list of the ISA in the *Companion* and below (for the math and memory instructions). Implement the following C program in assembly, using your previous work as a guide. Place your code in a file named `math.s`. Again, skip the procedure itself; just show the code inside of the procedure. Write, run, and test your program. Note: both `a` and `b` must be global variables in memory and the variable `a` must store the correct value when the execution completes; that's what we'll check to determine if your code works.

```
int a,b;

void math() {
   a = ((((b + 1) + 4) / 2) & b) << 2;
```

```
}
```

Some tips for 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.

Implement SM213 Memory and Math Instructions

The file *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a2/code.zip* contains the files `Memory.java` and `CPU.java`. The first is a complete implementation of the *sm213* machine's main memory; i.e., the solution to Assignment 1. You can use this file or your version from Assignment 1. The second is the starting point for the CPU implementation. In contains the implementation of some instructions and leaves to you the implementation of others. Copy this file into your IntelliJ environment to replace the `CPU.java` that is there now in the `arch.sm213.machine.student` package.

Here is a description of the memory and math instructions for your convenience. These instructions are described in more detail, including examples, in the *213 Companion*.

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`	`1psd`	r[d] ← m[o=p*4+r[s]]

Instruction	Assembly	Format	Semantics
load indexed	`ld (rs,ri,4), rd`	`2sid`	r[d] ← m[r[s]+r[i]*4]
store base + offset	`st rs, o(rd)`	`3spd`	m[o=p*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)

Question 3: Implement and Test the Remaining Instructions [50%]

Modify the `execute()` method in `CPU.java` to replace every "`TODO`" flag with the appropriate code. Each step of the processor (i.e,. cycle) has two stages: *fetch* and *execute*. The first stage *fetches* the next instruction from memory and loads it into the special-purpose register named `ins`. The second stage *executes* the instruction. For convenience the fetch stage places the various parts of the instruction into the following special-purpose registers

`insOpCode`, `insOp0`, `insOp0`, `insOpImm`, and `insOpExt`

and the execute stage uses these values plus the general-purpose register file, `reg`, and main memory, `mem`, to perform the instruction's specified computation.

To test your implementation of an instruction you need to execute some code in the simulator that uses that instruction and then manually examine the register and/or memory contents to ensure that it worked. You can set breakpoints in the Java to examine Java variables. You can use the

lower left panel of the simulator to see the value of the CPU's special-purpose registers. And you can set breakpoints in the assembly by clicking the little box next to an instruction, turning it red.

For example, to test the *load-immediate* instruction, you might write a little program called `test.s` that looks like this:

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

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

We recommend that you implement one instruction at a time and test each one before moving on.

The provided code includes a file named `test.s` that you can use to test the instructions you implement. Each test starts with a label that identifies the instruction it tests and ends with a `halt` instruction that stops the simulation. To test an instruction, click on its label in the simulator and then click "`Run`" or "`Step`".

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. Use the instructions that are already implemented as your guide.

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 IntelliJ/Eclipse workspace nor the entire source tree for the simulator and do not hand in any class files**.

1. `README.txt` – that contains the name and student number of you and your partner
2. `PARTNER.txt` – containing your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form a0z1). Your partner should not submit anything. Remember, this must be a plain-text file.
3. `swap.s` – that contains your code for Question 1.
4. `math.s` – that contains your code for Question 2.
5. `CPU.java` – that contains your implementation for Question 3. But sure you don't turn in the *.class* file.

OPTIONAL – Testing Using the Command Line

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.

To invoke the command-line version of the simulator you need to locate three items in your file system: your implementations of `MainMemory.class` and `CPU.class` and `SimpleMachineStudent213.jar` that you downloaded for this assignment (or Assignment 1). Lets say that all three are in the same directory and you have "**`cd`**"ed into that directory (i.e., if you type **`ls`** you see these three files) and thus the directory is called "`.`". To invoke the command-line simulator you type:

```
java -cp .:SimpleMachineStudent213jar SimpleMachine -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 -cp .:SimpleMachineStudent213jar SimpleMachine -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
```

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

```
% java -cp .:SimpleMachineStudent213jar SimpleMachine -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 -cp… -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 -cp… -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.