Lab Assignment 2: CPU and Static Scalars and Arrays

Due Sep 21, 2018 by 11:59pm **Points** 100 **Available** after Sep 17, 2018 at 12am

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 identify what the compiler can 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 double-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 **a2code.zip** contains some code snippets. The code `example_global_static` is similar to the code discussed in class. Your first task is to examine the C and sm213-assembly versions of this file.

The main function in `example_global_static.c` has two parameters: `argc` and `argv`. This is similar to Java's command line arguments, but in Java the main function has one parameter, an array of strings. In the C code, `argc` is the number of strings found on the command line that started the program. To run example_global_static.c you type the name of the executable + the 1 argument the program needs, making the value of `argc` 2 (two strings on the command line, the name of executable and the argument being passed). `argv` is a dynamic array that contains each of these strings (the name of executable and the argument). In C, a string is simply an array of char (i.e., a `char*`) that is terminated by the first null (i.e., 0) in the array - don't worry about the `*` for now, it will be discussed in unit 1c.

Your next task is to compile and run the C version and run the assembly version in the simulator.

How to Compile a C program

You will want to do this in a terminal on Mac/Linux machine or using Xshell with Windows as you did in Assignment 1 to compile your Java programs from the command line.

A C program consists of one or more files with extensions `.c` or `.h`. Files ending in `.c` are C source code. Files ending in `.h` are header (i.e., interface) files. Every program has at least one `.c` file. Some will use multiple `.c` files and the `.h` files will describe the interface each of them provides to the other. To import an interface, a `.c` file must include initial lines of the form `#include "foo.h"`. In this assignment, every program consists of a single `.c` file and so there are no ".h" files. Subsequent assignments will be more complex.

There are many dialects of C out in the wild. In this course we will use the 2011 GNU dialect, which is common (and the default on MacOS, but not on Linux). Dialects differ slightly in syntax and so you have to tell the compiler which dialect you are intending to use. You do this by adding the string `-std=gnu11` (the number 11/eleven, not the letter l). Another thing you have to tell the compiler is the name of the program you want it to create. You do this with the `-o foo` option (to make a program called `foo`). If you don't include this option, the compiler places the output in a file called `a.out`.

So, lets say you want to compile `example_global_static.c` to produce an executable named `example-gs`. You would type the following at the command line:

```
gcc -std=gnu11 -o example-gs example_global_static.c
```

Now to run the executable `example-gs` you would type the following at the command line:

```
./example-gs
```

After doing this you will notice that you get an error, as the program is expecting you to give it a command line argument. Specifically, the program needs a number that it will store into the global variable `a`. Therefore to run the executable `example-gs` you would type the following at the command line:

`./example-gs 5`

How to Run an sm213-assembly Program in the SimpleMachine

The Resources page on Canvas contains the reference implementation (i.e., solution) of the machine simulator you will build over the next few assignments. Assignment 1 has instructions to execute the simulator.

To run `example_global_static.s` click "Open" in the SimpleMachine interface and navigate to the location of `example_global_static.s`. Now 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". You can 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.

Part 1: Compile and Write C [5%]

1. `[5%]` If you have not already done so, compile and run the program `example_global_static.c` following the instructions for compilation above. Run this program a few times giving it different inputs each time: another number between 0 and 9, a negative number, a number larger than 9, a word. What happens in each case? Fix the program so that it will print "invalid input" and terminate if the given an input that is outside the bounds of array `b`.

Part 2: Write Assembly [45%]

1. `[10%]` Using `example_global_static.s` as a guide, implement the C code below (taken from `sum_and.c`) in assembly in a file named `sum_and.s`. Note: just as in `example_global_static.s`, ignore the main function and the `foo` procedure declaration (we will come to this later). Your program should thus just do two things: (1) allocate the global variables and (2) update the values of `sum` and `and`. You'll find a complete list of the ISA in the Companion and below (for the math and memory instructions).

   To test your solution: 1) compile and run `sum_and.c` with inputs for the array `b` given as command line arguments and 2) load `sum_and.s` into the simulator with the same inputs you gave to the C program for b; ensure the behaviour is the same. Remember, you should always have at least 2 test-cases!

```
int sum;
int and;
int b[2];
void foo () {
    sum = b[0] + b[1];
    and = b[0] & b[1];
}
```

2. `[15%]` Now lets bring in more math 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 main function and the procedure itself; just implement the code inside of the procedure. You are again provided with a complimentary `math.c` program that you can use to test your solution against as described in Question 1. 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. You may assume that b is a positive number.

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

3. `[20%]` Below you are provided with a C code snippet. You must write: 1) a corresponding C program and place it in `update.c` and 2) a corresponding Assembly program and place it in `update.s`.

```
int t;
int array[3];
void update() {
    t = array[2];
    t += 4;
    array[1] = array[2] + t;
    array[0] = t;
}
```

For `update.c`, please ensure you follow these requirements:

- The program should take 3 command line arguments to be converted to numbers and stored to the corresponding 3 indices of array. Use the other C programs provided as a guide (`sum_and.c`, `example_global_static.c`);
- `t` and `array` must be global variables and the correct values must be stored into `t` and `array` when the execution of the update function completes.
- When the program runs it should print out the values stored in `t` and in `array` after the update function executes. The output should be formatted as shown in this example output:

  ```
  t: 9
  array[0]: 9
  array[1]: 14
  array[2]: 5
  ```

For `update.s` you must ensure these requirements are followed:

- `t` and `array` must be global variables in memory, with corresponding labels in the Assembly file.
- the correct values must be stored into `t` and `array` when the execution completes.

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.

- You can edit instructions and data values directly in the simulator (including adding new lines or deleting them).
- 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 hard-coded 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.
- You can change the next instruction to be executed, corresponding to the PC (program counter) value, by double-clicking on an instruction. 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.
- 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.
- 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 **a2code.zip** (same as the one above) 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. It contains the implementation of some instructions and leaves to you the implementation of others. Copy this file into your IntelliJ (or other IDE)'s environment to replace the `CPU.java` that is there now in the `arch.sm213.machine.student` package.

Part 3: Implement and Test Instructions [50%]

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`. For convenience the fetch stage places the various parts of the instruction into the following special-purpose registers: `insOpCode`, `insOp0`, `insOp0`, `insOpImm`, and `insOpExt`. The second stage executes the instruction. It uses the special-purpose registers above plus the general-purpose register file, `reg`, and main memory, `mem`, to perform the instruction's specified computation.

Modify the `execute()` method in `CPU.java` to replace every "TODO" flag with the appropriate code. 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 IDE 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.

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

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.* Also do not hand in the executable files (or object files, if you're using gcc with the -c option) obtained from compiling the C files from parts 1 and 2.

- `README.txt` – contains the name and student number of you and your partner;
- `PARTNER.txt` – contains 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 (typing `cat PARTNER.txt` in a terminal should show only the name of your partner).
- `example_global_static.c` – contains your modified code for Part 1 - Question 1.
- `sum_and.s` – contains your code for Part 2 - Question 1.
- `math.s` – contains your code for Part 2 - Question 2.
- `update.c` and `update.s` – contains your code for Part 2 - Question 3.
- `CPU.java` – contains your implementation for Question 3. But sure you don't turn in the `CPU.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`, in addition to the `SimpleMachineStudent213.jar` file 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 .:SimpleMachineStudent213.jar 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 the line that starts with a `(sm)` prompt):

```
% java -cp .:SimpleMachineStudent213.jar 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 should 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 .:SimpleMachineStudent213.jar 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.

A testing 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.