CPSC 313: Computer Hardware and Operating Systems
Assignment #2, due Due Friday, May 18 at midnight. Late penalty of 20% per day for up to 3 days.

Introduction

In this assignment you will make three improvements to the instructions of the Y86-Seq implementation described in the textbook. Unlike the textbook, we are examining a Java implementation, not an HCL (Hardware Control Language) implementation. This Java-based approach was described briefly in class and is implemented by the Simple Machine simulator that was provided with assignment #1. The Y86-Seq hardware is implemented by the class `arch.y86.machine.seq.student.CPU` of the simulator. You should not need to look at any other classes to complete this assignment. Javadoc documentation is included with the Simple Machine source code.

Problems

Each of the following questions asks you to implement or modify a small number of instructions by adding them to the provided `Y86-Seq` processor description (in Java). For each question, you should:

1. First, use the notation described in class (and the book) to document each stage's role in implementing the instructions. That is, use the signal names (i.e., `PC`, `valP`, `iCd`, `iFn`, `rA`, `rB`, `valC`, `srcA`, `srcB`, `dstE`, `dstM`, `valA`, `valB`, `valE`, and `valM`), and the notation `R[x]` for access to register number `x` and M_b`[a]` for access to `b` bytes of memory starting at address `a`. List the behaviour of each stage separately: Fetch, Decode, Execute, Memory, Write-back and PC-update.

2. Modify the class `arch.y86.seq.student.CPU` we provided to add the instructions to the `Y86-Seq` implementation.

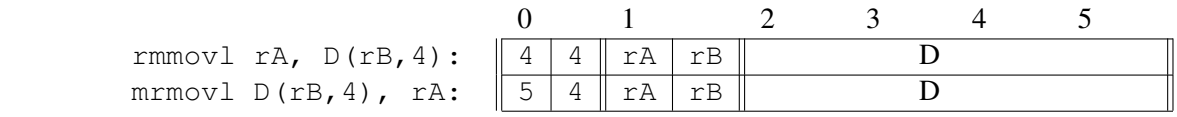
3. Design a tiny assembly-language test program to test the instructions. Explain how your test provides good test coverage for these instructions and what happens when you run the test (i.e., does the test succeed or fail).

Answer the questions one at a time, testing each set of instructions before moving to the next. You should modify only one version of file `CPU.java` so that once you have answered all of the questions you have a single version of this class that implements all of the instructions. The instructions you should add to the CPU are:

1. Improved `rmmovl` and `mrmovl` instructions that, when given an extra parameter 4, will multiply the contents of register `rB` by 4 before adding the displacement. Their semantics are:

   `rmmovl rA, D(rB,4)`: M_4`[D + 4 × R[rB]]` ← `R[rA]`
   `mrmovl D(rB,4), rA`: `R[rA]` ← M_4`[D + 4 × R[rB]]`

and their memory formats are

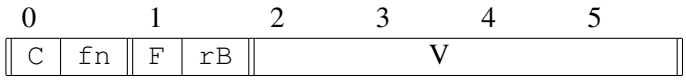


These instructions would be useful when accessing elements of an integer array, for instance.

2. Seven instructions

```
iaddl V, rB;  isubl V, rB;  ixorl V, rB;  iandl V, rB;
imull V, rB;  idivl V, rB;  imodl V, rB;
```

that add, subtract, xor, and, multiply, divide or mod a constant value to/from/with the contents of register `rB`. The constant to use for these instructions in the `switch` statements in file `CPU.java` is `I_IOPL`, its memory format is:



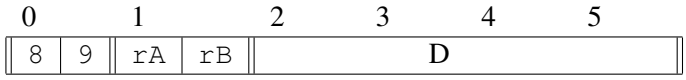
and its semantics are:

$\texttt{R[rB]} \leftarrow \texttt{R[rB]}$ OP $\texttt{V}$

Don't worry about the fact that we are asking you to add seven instructions. Doing this is no harder than adding a single one of these instructions; the number of lines of code that need to be added or modified is exactly the same.

3. An instruction `call *D(rB), rA` that calls the function whose address is in memory at location `D + R[rB]`, storing the return address in register `rA`. Such an instruction would be helpful, for instance, when calling Java methods: if `D` is the address of the array containing a list of addresses for the methods in a class, and register `rB` contains the index of a method (times 4), then this instruction will call the method with this index. The reason why we are storing the return address in register `rA`, instead of pushing it on the stack, is that the ALU is not available to decrement the stack pointer, being already used to add the displacement to the contents of `rB`. Consequently, the very first operation performed inside the method should be to push `rA` onto the stack.

   The constant to use for this instruction in the `switch` statements in file `CPU.java` is `I_CALL`, its memory format is:



   and its semantics is

$$\texttt{valP} \leftarrow \texttt{PC} + 6$$
$$\texttt{PC} \leftarrow \texttt{M}_4\texttt{[D + R[rB]]}$$
$$\texttt{R[rA]} \leftarrow \texttt{valP}$$

Challenge

Since January 2011, the CPSC 121 students have used a Logisim™ implementation of the Y86 CPU in the lab. As mentioned in class briefly, the only change needed was to add an extra byte of padding to the `halt`, `nop`, `ret`, `jXX` and `call` instructions to ensure every instruction has an even length. Along with this change, the CPU's memory was split into two separate memory modules, whose locations each contain one 32-bit word. The *even* module thus contains bytes 0 to 3 (address 0 in that module), bytes 8 to 11 (address 1 in that module), bytes 16 to 19 (address 2 in that module), etc. And the *odd* module contains bytes 4 to 7 (address 0 in that module), bytes 12 to 15 (address 2 in that module), etc.

This assignment's challenge problem is to modify the circuit to implement the instructions you added to the CPU. Please draw a red rectangle around each portion of the circuit that you modified, to make them easier to find for the teaching assistant. You will need Logisim™ version 2.7.1 or later (as of this writing, version 2.7.1 is the latest version available) to open the file; this version is installed on the undergraduate servers, and can be executed by typing `logisim` on the command line.

Deliverables

You should use the `handin` program to submit your assignment. The files to submit are:

1. The file `arch/y86/seq/student/CPU.java` containing your implementation of the three instructions.
2. A file in either text or PDF format that contains the following information:
   - Your name and student number.
   - Your pseudo-code, stage-by-stage description of the implementation of the three instructions.
   - A description of why your test programs provide sufficient test coverage.
   - A description of what happens when you run your each of your tests, including the regression tests. That is, does your solution pass all of the tests, and if not, what happens.
   - How long it took you to do the assignment (not including any time you may have spent revising before starting to work on it).
3. A copy of all test programs you used (one per file), named `movl.s`, `iopl.s` and `call.s` respectively.
4. If you completed the challenge problem, then you should also submit an updated copy of the file `Y86-cpu.circ`.

To submit the assignment, make sure that these files are in a directory called `~/cs313/a2` and run the command `handin cs313 a2`.