CPSC 313: Computer Hardware and Operating Systems
Assignment #4, due Monday June 4, 2012 at midnight. Late penalty of 20% per day for up to 3 days.

Introduction and Objectives

You may do this assignment in groups of 2, if you like.

This assignment has two parts.

In the first part, you will implement data forwarding and branch prediction for the `Y86-Pipe` implementation of the CPU. The objective of this assignment is to give you a taste for the challenges that arise when implementing data forwarding and branch prediction in a CPU.

In the second part, you will examine the execution of the `heapsort` program in the `Y86-Pipe` implementation with the goal of improving its pipeline performance. This can be achieved by changing the program's source code and/or the CPU implementation. You will get a good grade for showing some improvement. It is not necessary to find every possible improvement, though A+ type grades will be reserved for those who get close.

Part I

1. Implement `Pipe` with data forwarding and jump prediction as described in class and in the textbook: register–register data hazards should result in zero stalls, load-use hazards should result in one stall, `ret` instructions should result in three stalls, and you should predict that jumps are taken (and shoot instructions down when it is discovered that the guess was incorrect).

   Implement your solution by modifying the class `arch.y86.pipe.student.CPU`. Your coding can be confined to the following helper methods (although you are allowed to make changes to any method you want): `pipelineHazardControl`, `fetch_SelectPC`, `fetch_PredictPC`, and `decode_ReadRegisterWithForwarding`. Clearly indicate the code you added or changed, using comments in the code.

   Hints:

   - First implement data forwarding, and test it thoroughly. You do not need to worry about jump prediction while doing this, as the first few tests in file `pipe-test.s` do not have any conditional jump instructions. Once you are satisfied that your data forwarding code works as expected, then worry about jump prediction.
   - Hazard detection is pretty much the same in both the `PipeMinus` and `Pipe` versions of the CPU. It's only how the hazard is dealt with that differs.
   - Look at the final version of the code for your `PipeMinus` implementation of method `pipelineHazardControl` for inspiration about what the `Pipe` implementation of the same method should do.

- Method `pipelineHazardControl` is the only method that should set pipeline stages to stall or bubble. Even if you are tempted to put such code in one of the pipeline stages (for instance, in `fetch_PredictPC` or `fetch_SelectPC`), don't!

2. Use `pipe-test.s` to test your solution. For your solution to be correct it must execute programs correctly and with the appropriate number of stalls. Document the results of each test by indicating whether or not your code passed the test.

3. Answer question 2 from the previous assignment, but now for your `Pipe` implementation. Compare these results with those from question 2 of assignment 3 on `PipeMinus`.

Part II

Now your task is to improve the performance of the program `heapsort.s` by modifying `heapsort.s` and your pipe implementation. The file `heapsort.s` you will start with is provided on with the assignment link on the course web site (in code.zip). Measure performance using the `cCnt` cycle counter register and by computing CPI using the `cCnt` and `iCnt` registers. That is, record both the total number of cycles and the pipeline efficiency.

If you are unable to complete the first part of this assignment, you can get partial credit for the by using the reference implementation for Part II and focusing your improvements on changing the program.

Coding Rules: You are free to make any modifications you wish within the following constraints:

- Your final version of `heapsort.s` must work for arbitrary size arrays. You might be tempted to hardwire your solution for 13-element arrays, but this would be a bad idea because your solution will be graded based on its performance on arbitrary arrays.

- Your final version of `heapsort.s` must run correctly, and can only implement the `Heapsort` algorithm. You are not allowed to switch to the implementation of a different algorithm.

- You may not change the semantics of any instruction in the standard Y86 ISA, or create any new instructions other than those you added in assignment 2.

Out of the 30 marks allocated for this part of the assignment, 20 will come from the improvements you make to the CPU, and 10 will come from the changes to file `heapsort.s`. Here are some possible CPU improvements you could make (all were discussed in class), along with the number of marks that a correct implementation of each improvement would be worth:

- Predict that forward jumps will not be taken, but backward jumps will: 8 marks.

- Use a small hardware stack to predict targets for `ret` instructions: 8 marks.

- Avoid load/use hazards when possible: 8 marks.

- Dynamic jump prediction: 12 marks.

There are other possible improvements; how much each is worth will be determined on a case by case basis. If you obtain more than 20 marks for improvements to the CPU, then your grade for this part of the assignment will be computed as $20 + (x - 20)/2$, where x is the number of marks you obtained.

Note that your modifications to the CPU should all lower the CPI for `heapsort.s`. Modifications to the program itself may in fact increase the CPI, because they may reduce the number of instructions executed without changing the number of bubbles.

Deliverables

You should use the `handin` program to submit this assignment. The assignment name is `a4`, and the files to submit for this part are:

1. Two completed `CPU.java` files (with comments) — one for Part I and one for Part II — located in subdirectories named "part1" and "part2".
2. Your modified version of `heapsort.s` in the main directory.
3. A file in the main directory in either text or PDF format that contains the following information:
   - Your name(s) and student number(s).
   - [**Part I**] A description of the results for the 8 tests in file `pipe-test.s`, including the number of stalls you observed, and whether or not the test produced the result you expected.
   - [**Part I**] Your CPI results from question 3.
   - [**Part II**] Which features you added to the implementation of the CPU, and a short summary of the changes you made to the methods in file `CPU.java`.
   - [**Part II**] A short description of the changes you made to file `heapsort.s`.
   - [**Part II**] Your final `cCnt` and CPI results for `sum.s`, `max.s`, the version of `heapsort.s` provided, and your improved version of `heapsort.s`.
   - Number of hours it took you to complete Part I. For pairs, sum the individual times.
   - Number of hours it took you to complete Part II. For pairs, sum the individual times.

If you are working in a pair, **one** of you only should submit the files as listed above. The other member of the pair should submit a single **plain text** file called `partner.txt` that contains the undergraduate login ID of the other member (for instance, `c3p0`). This will allow both grades to be recorded correctly, without forcing the TA to grade the assignment twice.

Here is a rough breakdown of the marking scheme for this assignment:

- The Part I implementation of `pipelineHazardControl` will be worth 12 marks.
- The Part I change(s) to `fetch_SelectPC` are worth 6 marks.
- The Part I change(s) to `fetch_PredictPC` are worth 3 marks.
- The Part I implementation of `decode_ReadRegisterWithForwarding` is worth 9 marks.
- The documentation, CPI results and time spent are worth 5 marks in total.
- Part II, 30 marks as described above.