Lab 4a: Control Flow and Branch Prediction

Assigned: Wed., 2/20; Due: **Fri., 3/21** (Midnight)

Instructor: Onur Mutlu
TAs: Rachata Ausavarungnirun, Varun Kohli, Paraj Tyle, Xiao Bo Zhao

1. Introduction

In this lab, you will extend your pipelined ARM machine to support the control flow instructions that we deferred from the previous lab (conditional and unconditional branches). You will also implement two branch predictors: (1) A simple global history-based branch predictor, and (2) a tournament-style branch predictor similar to the Alpha 21264 that we learned about in class. Along the way, you will learn about the squashing and correctness issues that arise due to control flow in a real pipeline, and when you are finished, you will have a complete pipeline that supports the core essentials of a real ISA.

2. Additions to the ARM Machine

2.1. Architecture

- **Instruction Set.** Using your five-stage pipelined ARM core **with forwarding** as a starting point, you will add support for control flow instructions (jumps and branches) *without* branch delay slots, as in the previous labs. The machine supports all ARM instructions specified in Lab 1. The behavior of your pipeline should be match that of your (now correct) implementation of the Lab 1 simulator for the 51 instructions in the following table.

`ADC`	`ADD`	`AND`	`B`	`BIC`
`BL`	`CMN`	`CMP`	`EOR`	`LDR`
`LDRB`	`MLA`	`MOV`	`MUL`	`MVN`
`ORR`	`RSB`	`RSC`	`SBC`	`STR`
`STRB`	`SUB`	`TEQ`	`TST`	`SWI`

- **System Call Instruction.** Terminates the program (same as Lab 3). In addition, for debugging and grading purposes, you must ensure that the contents of the register file are dumped out in the **same format** as Lab 3.
- **Exceptions.** No support (same as Lab 3).

2.2. Microarchitecture

Pipeline Modifications.

In this lab, you will modify your five-stage pipeline to support branch and jump instructions by modifying the execute stage. Specifically, in addition to performing ALU and load/store operations, the execute stage will handle instructions that modify *control flow* as well. To do so, you will modify the execute stage to calculate the branch condition and the branch target address. Once the target address is computed, it can be sent to the fetch stage to change the location of fetch when a branch is taken.

Pipeline Flushing.

In between when a branch is fetched and when the first instruction from the resolved branch target is fetched, other instructions will have entered your pipeline. It is your job to figure out how to flush the pipeline so that from the programmer's point of view, only the correct instructions are executed.

Branch Prediction.

Now that you have control flow instructions working, you can start improving the performance of your machine using branch prediction. You will implement two types of branch predictors that we learned about in class, and submit each of these implementations *separately.*

1. **Global Branch Predictor.** The first branch predictor you will implement will be a simple global predictor, similar to the one we talked about in class. The global predictor consists of the following components:
   - A global history register that contains the outcome of the past 12 branches.
   - A pattern history table that contains 4096 entries.
   - A 2-bit saturating counter for each entry in the pattern history table.

   You are free to determine the starting values for these components. An important aspect of your design will be *when* you decide to update the global history register and pattern history table entries. One option is to update them speculatively based on the *predicted* outcome of the branch, another is to wait until the branch direction has been determined and update them based on the *actual* outcome of the branch. Experiment with both techniques and find a configuration that works well[1].
2. **Tournament Branch Predictor.** The second branch predictor you will implement is a hybrid branch predictor that includes a local branch predictor and a global branch predictor. A separate choice predictor chooses which of the two branch predictors to use for each branch.

   For the global branch predictor, you may reuse the design that you implemented earlier. The local predictor consists of the following components:
   - 16 local history registers.
   - A 10-bit branch history for each local history register based on the PC of the branch.
   - A pattern history table containing 1024 entries.
   - A 2-bit saturating counter for each entry in the pattern history table.

   The choice predictor uses the 12 bits from the global history register to index into a 4096-entry choice history table. Each entry in the choice history table contains a 2-bit saturating counter specifying which predictor (local or global) to use, and is updated during branch retirement if the local and global predictors disagreed in the branch direction.

   You are free to determine when to update the local history register and also the initial values for the components. Once again, experiment with different designs to try and find a configuration that achieves good performance.

3. Submission

3.1. Lab Section Checkoff

So that the TAs can check you off, please come to any of the lab sections *before* Sat., 4/1. **Note that you must be checked off for both versions of your implementation (the global branch predictor and the tournament branch predictor).** You can get them checked off separately in different lab sections or all in one sitting. Please come *early* during the lab section. During the Lab Section, the TAs may ask you:

- to answer questions about your implementations,

[1] A useful paper that analyzes and discusses the trade-offs of these two update strategies is on the course web page at: `http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=p228-hao.pdf`.

- to simulate your implementations using various test inputs (some of which you may have not seen before),
- to show that your implementation can both update the history registers and pattern history table entries based either on the predicted outcome of branches or the actual outcome of branches (and explain why one works better than the other),

3.2. Source Code

Make sure that your source code is readable and documented. Please submit the lab by executing the following commands. **Make sure you format your folders as specified. We will penalize wrong submissions.**

Global Branch Predictor.

```
$ cp -r src /afs/ece/class/ece447/handin/lab4a/andrewID/global
$ cp -r inputs /afs/ece/class/ece447/handin/lab4a/andrewID/global
```

Tournament Branch Predictor.

```
$ cp -r src /afs/ece/class/ece447/handin/lab4a/andrewID/tournament
$ cp -r inputs /afs/ece/class/ece447/handin/lab4a/andrewID/tournament
```

3.3. README

In addition, please submit two `README.txt` files. To submit these files, execute the following command.

```
$ cp README.txt /afs/ece/class/ece447/handin/lab4a/andrewID/global/README.txt
$ cp README.txt /afs/ece/class/ece447/handin/lab4a/andrewID/tournament/README.txt
```

The `README.txt` file must contain the following three pieces of information.

1. A high-level description of your design (including what are the initial values of your components).
2. The results of your experimentation with updating the various history registers and history table entries. Note that you do not need to report every permutation of local versus global update for each component, but at least show results for (1) updating all components with the predicted outcome, (2) updating all components with the actual outcome, and (3) discuss which design that you experimented with performs the best (which may be a hybrid of the two update strategies) and why.
3. The percentage speedup of each of your branch predictor designs over your machine without branch prediction for the `primes` input program.

It may also contain information about any additional aspect of your lab.

3.4. Late Days

We will write-lock the handin directories at midnight on the due date. For late submissions, please send an email to `447-instructors@ece.cmu.edu` with tarballs of what you would have submitted to the handin directory.

```
$ tar cvzf lab4a_global_andrewID.tar.gz src inputs README.txt
$ tar cvzf lab4a_tournament_andrewID.tar.gz src inputs README.txt
```

Remember, you have only 7 late lab days for the entire semester (applies only to lab submissions, not to homeworks, not to anything else). If we receive the tarball within 24 hours, we will deduct 1 late lab day. If we receive the tarball within 24 to 48 hours, we will deduct 2 late lab days... During this time, you may send updated versions of the tarballs (but try not to send too many). However, once a tarball is received, it will immediately invalidate a previous tarball you may have sent. You may not take it back. We will take your very last tarball to calculate how many late lab days to deduct. If we don't hear from you at all, we won't deduct any late lab days, but you will receive a 0 score for the lab.

4. Extra Credit

For extra credit on this lab, we will hold a performance competition. Among all implementations that are correct, the "top"[2] students that have the lowest execution time for an undisclosed set of test inputs will receive up to 20% additional credit *for this lab* (equivalent to 1% additional credit for the course) as well as "prizes" (at the discretion of the instructor). You may choose to submit the global or the tournament version of the machine without making any effort to optimize it further. As an initial idea, you can take a look at the perceptron predictor at `https://www.cs.utexas.edu/~lin/papers/hpca01.pdf`.

All of the guidelines for Lab 4a specified in this handout also apply to the extra credit, except for the following differences.

- Submission path: `/afs/ece/class/ece447/handin/lab4a/andrewID/extra`
- Tarball (for late submissions): `lab4a_extra_andrewID.tar.gz`

For late submissions, we must receive all three tarballs (stalling, forwarding, extra credit) on the same day. If not, we will deduct the lab late days for whichever tarball we received last.

[2]The instructor reserves all rights for the precise definition of the word "top".