CS 152 Laboratory Exercise 1 (Open-Ended Portion)

Instructor: John Lazzaro
TA: Eric Love
Department of Electrical Engineering & Computer Science
University of California, Berkeley

February 22, 2014

1 Introduction

In the open-ended portion of Lab1, you will explore the effectiveness of different branch predictor designs. To do so, you will use the version of the Sodor 5-stage core that includes a simulated branch predictor in C++.

The lab consists of two parts. In the first part, you will try to exploit whatever information you can to predict branches and their targets. In the second, you will be given a limited set of resources, and have to decide how best to use them for branch prediction. First, though, you will learn more about the new 5-stage datapath, and about the changes we've made to the process for obtaining and submitting the lab.

1.1 The Modified 5-stage Datapath

Figure 1 shows the modifications that have been made to the 5-stage datapath in order to accommodate a branch predictor and BTB in the fetch stage. In each cycle, whatever value is in the PC register is fed into the branch predictor at the same time as it is used to access instruction memory.

By the end of the cycle, the instruction memory will have returned the fetched instruction word, and the BP/BTB will return two predictions: (1.) whether that instruction is a taken branch or a jump, and (2.) the target address of that branch or jump. When the predictor thinks the result of instruction fetch is a taken branch or jump, and assuming there is no pipeline flush signal from the execute stage, then the selector signal sent to the PC mux will be set to use the predicted target as the next value to be stored into the PC register.

At the same time, as the instruction predicted to be a branch moves down the pipeline, it saves the predicted target address in new pipeline registers for this purpose. When it reaches the execute stage, where target addresses are calculated, the predicted target is compared with the actual target (or PC+4). If the two disagree, then a misprecict has occurred, and the fetch and decode stages must be killed just as they were previously on every branch/jump, introducing a two-cycle mispredict penalty. However, if they agree, then execution simply continues, and the branch executes with a CPI of one. In either case, the result of branch target calculation, and the branch direction (taken/not taken), are fed back to the predictor so that it can adjust its predictions with this new information.

We have made all the necessary modifications to the datapath and control logic in Chisel to introduce this BTB. If you are curious, you can learn more about these changes by examining the files in `src/rv32_cpp_bp/cpath.scala` and `src/rv32_cpp_bp/btb.scala`. The code in `btb.scala` can implement a simple BTB, or be configured to use a predictor simulated in C++ as described later in this document.

1.2 Obtaining and Submitting `lab1-open`

For this part of the lab, we have fixed the issue with publicly-visible git repos. This time you'll have your own private repo, which the TA will have created based on your github userid. If your userid is `ericlove`, then you will have access to a private repo in `ucberkeley-cs152-sp14 / cs152-ericlove`.

To start the lab, go to `github.com`, login with your userid, and find the URL for your private lab repo. It will look something like: `git@github.com:ucberkeley-cs152-sp14/cs152-ericlove.git`

Open an SSH connection to one of the CS152 instructional machines in `t7400-1.eecs.berkeley.edu` to `t7400-12.eecs.berkeley.edu`. Then, clone your private git repo into your home directory:

```
inst$ git clone git@github.com:ucberkeley-cs152-sp14/cs152-ericlove.git
inst$ cd cs152-ericlove
```

If you run the `ls` command, you'll see this directory is empty. To download the lab, this time you'll need to copy it from the CS152 public directory:

```
inst$ cp ~cs152/sp14/lab1-open.tar .
inst$ tar -xf lab1-open.tar
inst$ rm lab1-open.tar
```

Don't worry if these commands are unfamiliar–just follow the instructions precisely for now. Once you've extracted these files into your private repo directory, you'll need to commit them:

```
inst$ git add lab1-open
inst$ git commit -m "Initial commit of lab1-open"
inst$ git push
```

Now, if you open github in your browser and examine the contents of your personal repo, it should contain the `lab1-open` directory. Verify that this is true. For the rest of this document, assume that by `${LABROOT}$` we mean to refer to your copy of the `lab1-open` directory.

To start working, make sure you have run `./configure` in the `lab1-open/` directory:

```
inst$ cd lab1-open
inst$ ./configure
```

1.2.1 Saving and Submitting

Once you begin working on the lab, you should save your work frequently by following this sequence of steps:

- Run `git status` to see which files you've modified

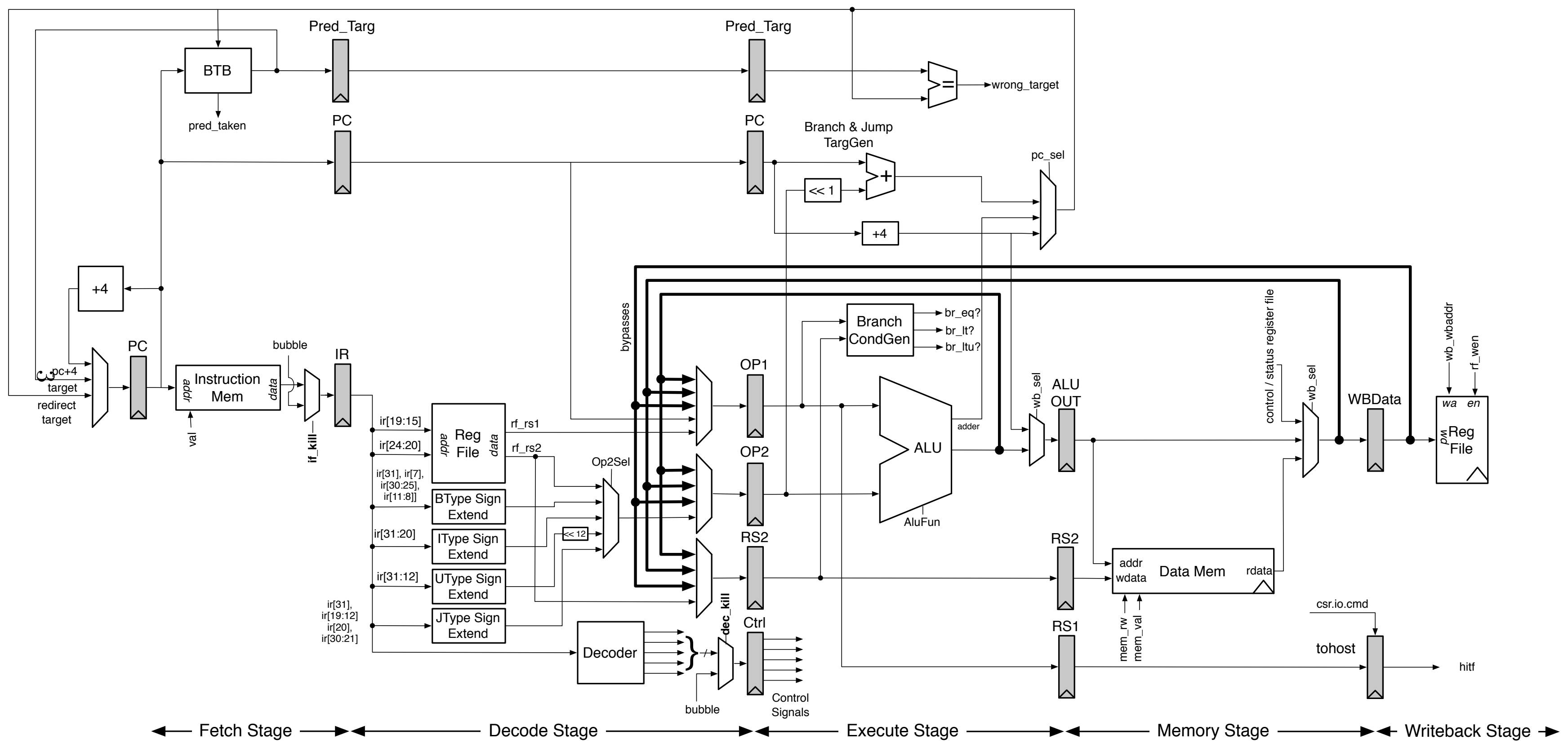

Figure 1: Datpath of the 5-stage Sodor core with a Branch Predictor in the Fetch Stage

- Use `git add <files>` to add the files you've updated to the commit list
- Commit your changes using `git commit -m "<Commit Message>"`
- Push your commit to github by running `git push`

Whatever you have *pushed to github* (and only that) will automatically be visible to the TA. Thus, to submit your lab, just make sure all your changes are pushed to github.

1.3 Getting Started with the C++ Branch Predictor Core

In this lab, you won't modify any Chisel code directly. Instead, navigate to the emulator directory for the 5-stage with C++-based branch prediction:

```
inst$ cd ${LABROOT}/emulator/rv32_cpp_bp
inst$ make run-bmarks-test
inst$ grep CPI output/*.riscv.out
```

This sequence of steps will build and run the simulated core, and then display the CPIs of the various benchmarks in the terminal. If you want more statistics about each one, you can open its `.out` file in `output/` as in the directed portion.

Now, open the C++ branch predictor code (replacing vim with your editor of choice):

```
inst$ vim ../common/bp.cpp
```

Scroll down to the declaration of the `BTB` class:

```
// Sample branch predictor provided for you: a simple branch target buffer.
class BTB : public BranchPredictor
{
  private:
    typedef struct {
      uint32_t target_pc;
      uint32_t tag_pc;
    }BTBEntry_t;
    BTBEntry_t* table;
    ..
```

This implements the baseline BTB, equivalent to the Chisel code in `btb.scala`. You will note that it includes the following functions:

- `uint32_t predict_fetch ( uint32_t pc )`: this returns the value of the predicted target PC of the instruction currently being fetched based on the value of the fetch PC reg. If the prediction is "not taken," it returns zero.
- `void update_execute ( ... )`: this is called every cycle with information about the instruction currently in the execute stage. Use this to update information in your predictor state.

You will also note the following line at the top of `bp.cpp`:

```
#define BRANCH_PREDICTOR BTB
```

The value "BTB" tells the simulator to use the branch predictor defined by the class with the name `BTB`. To choose your own branch predictor, simply change this to the name of your class. Test this out by changing the value to `InfiniteBTB` and recompiling. What are the new results? Note that your class must inherit from `BranchPredictor`, and should follow the general pattern set by the `PredictorTemplate` class (you should copy the constructor and virtual function definitions from this).

2 Analysis of Predictability

In the first part of this lab activity, you will attempt to predict branches as well as possible, given an unlimited amount of (simulated) hardware resources.

2.1 Getting Started

Start by measuring the performance of the `NoBP` (no branch predictor) predictor. What is the CPI for each benchmark? What is the contribution of branch mispredicts to CPI, given that the mispredict penalty for the 5-stage is two cycles? Note that for each benchmark, the number of instructions executed and number of mispredicts are output to the `.out` file by the `BranchPredictor` base class automatically, and prefixed with `##` so you can get a quick summary of these data by typing

```
inst$ grep "##\|CPI" output/*.riscv.out
```

Now change to the `InfiniteBTB` predictor that is also provided for you. How much better does it do? How much room for improvement is left?

2.2 Implement Your own Predictor

Try to improve on the infinite BTB with your own predictor design. You have complete freedom to try any kind of branch predictor style you like. Some things you might want to try include:

- Adding hysteresis (saturating counters). Does this improve any benchmarks?
- Track the patterns of branches globally and locally:
  - Use a pattern history table and global history register (i.e. track the directions of the most recently executed branches)
  - Track the history of *individual* branches: use the directions of the most recent N executions of a single branch to predict it's direction this time. What is the best value of N?
- Try something extra:
  - Add a return address stack. Look at the RISC-V ISA spec to see how function calls/returns are canonically encoded with JAL/JALR. Note that you'll need to learn which PCs correspond to return jumps, since the instruction isn't available during predictor access.

  - Implement your favorite machine learning algorithm on whatever branch data you choose to collect.

Please feel free to consult the slides from lecture/section, the textbook, Wikipedia, Google Scholar, or any other resources of your choice for ideas.

In your writeup, include a detailed description of your predictor's design, including diagrams if necessary, and an explanation of how and why it works.

3 Resource Constrained Design Problem

In this portion of the lab, you will try to design the best branch predictor given certain constraints on the hardware resources available to you. Your design constraints are as follows:

- You may not use more than 1024 bytes of total state
- Every separate memory structure increases the cycle time by 0.25% above the baseline. Thus, adding an 8-way associative predictor increases cycle time by 2%.
- Assume computation costs nothing. Be as crazy as you want to.

Thus, if the value `BTB_ADDR_BITS` for the default BTB implementation is set to 7, it will use

$$2^7 \times (4B + 4B) = 128 \times 8B = 1024B$$

total state, assuming 4 bytes to store the tag PC and another 4 for the target. Note that, in your C++ code, you will still want use the `uint32_t` type to hold bits of state even when you only intend for a subset of these to count towards your total storage. That's fine–just make sure that your code only uses the subset indicated by your design description.

Try out several designs and see which one works best. In your answer to this question, you should describe at least one design that worked best, and another that did not work so well. Your descriptions should include the following:

- A description of all state elements and their sizes
- An analysis of the runtime of the benchmarks according to the cycle time changes described above
- A comparison to the baseline BTB
- An intuitive account of why this style worked or did not work
- Diagrams of your design if you think they will help explain it

You should also include the code in your repo. We won't run it, but we will have a look at it.

4 What to Turn in for this Lab

You should submit a writeup that includes the items requested in both problems. Submit your writup by making a file called `report.pdf` or `report.txt` in the `lab1-open` directory of your personal repo. If you need to upload a PDF to the instructional machine, use the `scp` program on your personal machine (you should have this automatically if you use a Mac or Linux; if you use Windows, you should install Cygwin to gain access to this and other Unix utilities):

```
inst$ scp report.pdf ericlove@t7400-1.eecs.berkeley.edu:~/cs152-ericlove/lab1-open/
```

Then follow the normal git commit and push routine to upload it to github.