ECE 4750 Computer Architecture, Fall 2023
Lab 4: Branch Predictors

School of Electrical and Computer Engineering
Cornell University

revision: 2023-11-22-06-33

This assignment aims to delve into the design and evaluation of branch prediction mechanisms. Branch predictors aim to optimize throughput by accurately predicting the outcomes of branch instructions so that a processor may fetch past unresolved branches unabated. In this lab, you will implement a Bimodal Branch Predictor for Design #1, a Global Branch Predictor for Design #2, and a GShare Branch Predictor for Design #3 (new to you). The branch predictor structure should be parameterizable, and as part of your lab report, you should evaluate the different designs for various sizes.

You must implement all three designs, verify them using an effective testing strategy, and conduct an evaluation comparing the three implementations. **You should consult the course lab assignment assessment rubric for more information about the expectations for all lab assignments and how they will be assessed. The rubrics for lab and report grading will be adjusted for three designs.**

This handout assumes you have read and understood the course tutorials and lab assessment rubric. To begin, download the starter files from the course website and extract them into a directory titled `sim` with the following commands:

```
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
# download the file to this folder from Canvas
% tar -xvzf lab4.tar.gz
% cd sim
% make setup
```

You can run tests (none are included) like this:

```
% cd ${HOME}/ece4750/sim/lab4_branch
% make run-all
```

For this lab, you will be working in the `lab4_branch` subproject, which includes the following files:

- `BranchBimodal.v` – Bimodal Branch Predictor
- `BranchGlobal.v` – Global Branch Predictor
- `BranchGShare.v` – GShare Branch Predictor
- `default.config` – Config file for Makefile to allow testbench reuse.

1. Introduction

In high-performance processors, control flow hazards can be costly. In that context, dynamic branch prediction becomes critical to reducing stalls and delivering high throughput. This assignment aims to explore and design three distinct predictors to enhance the efficiency of a processor.

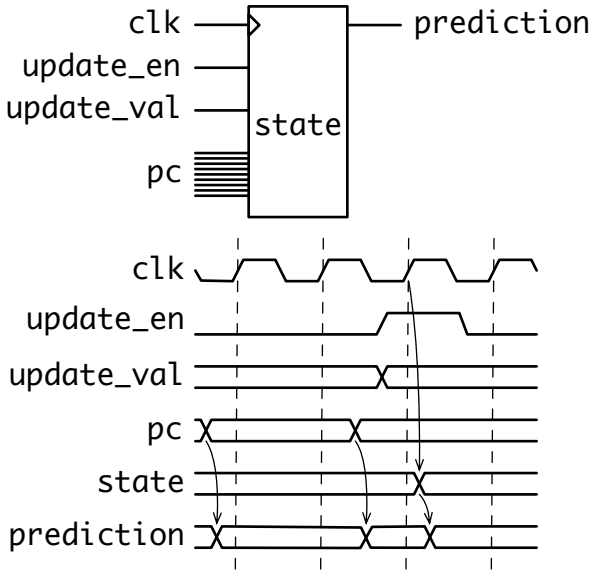

Figure 1: Block and timing diagrams for the branch predictor designs that use branch PC to make a prediction (Bimodal and GShare). Predictions are calculated combinationally on the input PC value and the internal state. The internal state of the branch predictor only changes at the positive edge of the clock when `update_en` is asserted. Recall that, in the case of the Global predictor, predictions are insensitive to the PC.

2. General Design

You will create a branch predictor module that takes inputs *clk*, *reset*, *pc*, *update_en*, and *update_val*, and a single output *prediction*. Refer to the figure 1 for a sample waveform on how they should interact.

- `clk` and `reset` are the same signals you have seen in the previous labs.
- `PC` is the PC of the branch being predicted. The branch predictor uses this information and its internal state to predict combinationally. (Recall that the Global Branch Predictor ignores the PC of a branch, relying solely on the internal state.) The prediction signal should be 0 if the branch is predicted not taken and 1 if it is.
- Asserting `update_en` tells the branch predictor to update itself at the next rising edge of the clock, based on the `PC` provided at that time and the actual outcome for that branch, (`update_val`)– also 0 for not taken and 1 for taken.

Your branch predictor design should accept a parameter `pht_size`, expressed in number of entries. You should adjust it up and down as part of your evaluation of its prediction accuracy profile. Please use only two-bit saturating counters in the PHT. Also, please note that using `vc` lib is expressly prohibited in synthesizable code.

3. The GShare Predictor

The GShare predictor is a surprisingly powerful design that tries to capture both local and global behavior with virtually no additional logic over a Bimodal or Global predictor. In a GShare predictor

with a 2^k-entry PHT, a k-bit Global History Register (GHR) is XOR'd with the lower k bits of the PC (excluding the two least significant bits of the PC as in the Bimodal predictor, since instructions are word-aligned in RISC-V), and the result is used to index the PHT and make a prediction. Updating the branch predictor is done exactly the same as the Global predictor: 1) update the saturating counter of the PHT entry used in the last prediction, then shift the GHR one position to the left, dropping the most significant bit and shifting into the least significant but the most recent outcome of the branch (0 not taken, 1 taken). That's it!

4. Testing Strategy

A key part of this lab is verification. As with all previous labs, you must verify that your branch predictor is working as expected. As always, as part of the verification process, you will have to demonstrate full line and toggle coverage and correct functionality for common and edge cases. Additionally, **you will write individual unit testbenches for all the modules** inside the `lab4_branch` folder, which you can run as follows:

```
% cd ${HOME}/ece4750/sim/lab3_cache
% make utb_XXXX.v.sim [DESIGN=${DESIGN}$] [RUN_ARG=--trace] [COVERAGE=${COVERAGE}]
```

You will add your directed and random tests in your unit testbench. You will be writing a lot of test cases but do not just make the given test case larger.

Some suggestions for what you might want to test are listed below. Each of these would probably be a separate test case.

- Loop with no branches
- Loop with a single branch, always taken
- Loop with a single branch, always not taken
- Loop with a single branch, alternating between taken and not taken
- Loop with a single branch, with various other patterns such as AAB, BBA, AAAB, BBBA, ABBA, ABAB
- Loop with N-branches, all taken
- Loop with N-branches, all not taken
- Loop with N-branches, alternating between taken and not taken
- Loop with N-branches, with various other patterns such as AAB, BBA, AAAB, BBBA, ABBA, ABAB
- Nested loop
- Nested loop with branches
- A series of branches all taken
- A series of branches all not taken
- A series of branches alternating between taken and not taken
- A series of branches with various other patterns such as AAB, BBA, AAAB, BBBA, ABBA, ABAB
- Branches with conflict in PHT

5. Evaluation

Your goal is to plot the prediction accuracy of the three designs for various sizes. You should plot your results on a single graph, where the x-axis shows the total size of the PHT (number of entries) and the y-axis shows the prediction accuracy (higher is better). Please obtain the prediction accuracy for all power-of-two sizes between 16 and 1,024 entries. In addition to writing your own microbench-

marks to do this, the course staff may provide a mandatory microbenchmark for you to include in your report at a later point in time.

6. Submission

The code submission for both milestone and final submission is the exact same process. For this lab you will be submitting a compressed tar file (with the file ending `.tar.gz`) to Canvas with the following files/folders:

- `Makefile` – Makefile to run the Verilog simulator
- `verilator.cpp` – C++ harness for the Verilog simulator
- `group$XX.txt` – XX is your group number. File contains all the netids of all of the group members (one per line).
- `lab4_branch/` –

The `lab4_branch` sub-project folder should have the following files at minimum:

- `BranchBimodal.v` – Bimodal Branch Predictor
- `BranchGlobal.v` – Global Branch Predictor
- `BranchGShare.v` – GShare Branch Predictor
- `utb_BranchBimodal.v` – Unittesting for Bimodal Branch Predictor
- `utb_BranchGlobal.v` – Unittesting for Global Branch Predictor
- `utb_BranchGShare.v` – Unittesting for GShare Branch Predictor
- `default.config` – Config file for Makefile to allow testbench reuse.

In addition to the following mandatory files as listed above, you should submit any additional modules, `.config` files, and testbenches (files with the `tb/ub/utb` prefix) you have created. Make sure you do not modify the provided module declarations, the Makefile, `verilator.cpp`, and `vc` files.

You can create a compressed tar file by the following command:

```
% tar -czvf lab4.tar.gz file1 [file2] [...]
```

For your convenience, we have a make rule to assist you in creating the tar file. However, you are ultimately responsible for the contents (and the lack of) of the tar file you submitted.

```
% cd ${HOME}/ece4750/sim
% make build-tar
```

Acknowledgments

Socrates Wong and José Martínez prepared this version of Lab 4 as part of the course ECE 4750 Computer Architecture at Cornell University.