

ECE 5720, Fall 2020  
Exploring Branch Prediction Algorithms  
Assigned: Oct. 8, Due: Oct. 29

Some sections of this handout are copied from the following Wikipedia page:

[https://en.wikipedia.org/wiki/Branch\\_predictor](https://en.wikipedia.org/wiki/Branch_predictor)

## 1 Introduction

In this lab, you will learn and implement three branch prediction algorithms. Branch predictors are used in modern pipelined microprocessors to *guess* the outcome of a branch instruction, before it is known whether the branch will be taken or not taken.

Two-way branching is usually implemented with a conditional jump instruction. A conditional jump can either be "not taken" and continue execution with the first branch of code which follows immediately after the conditional jump, or it can be "taken" and jump to a different place in program memory where the second branch of code is stored. It is not known for certain whether a conditional jump will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline.

Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline. The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong, then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.

The time that is wasted in case of a branch misprediction is equal to the number of stages in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. As a result, making a pipeline longer increases the need for a more advanced branch predictor.

## 2 Branch Prediction Algorithms

You will implement the following three branch predictors:

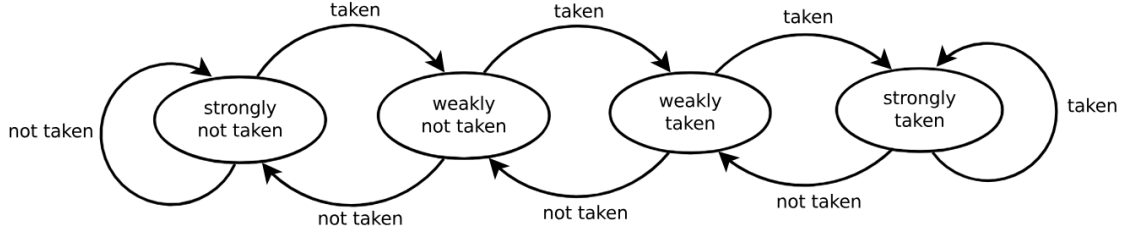


Figure 1: State diagram of 2-bit saturating counter.

## 2-bit Saturating Counter

As shown in figure 1, a 2-bit saturating counter is a state machine with four states:

- Strongly not taken
- Weakly not taken
- Weakly taken
- Strongly taken

When a branch is evaluated, the corresponding state machine is updated. Branches evaluated as not taken change the state toward strongly not taken, and branches evaluated as taken change the state toward strongly taken. The advantage of the two-bit counter scheme over a one-bit scheme is that a conditional jump has to deviate twice from what it has done most in the past before the prediction changes. For example, a loop-closing conditional jump is mispredicted once rather than twice. The predictor table is indexed with the instruction address bits.

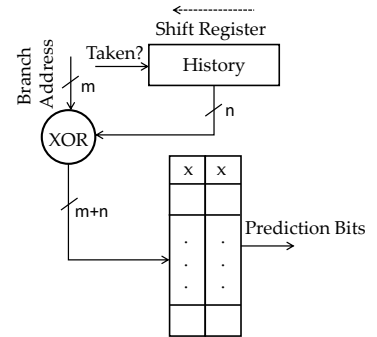


Figure 2: Gshare predictor.

## Gshare Predictor

Gshare predictor is a global branch predictor that keeps a shared history of all conditional jumps. As shown in figure 2, this method records the history of branches into a shift register. Then, it uses the address of the branch instruction (i.e., program counter or PC) and branch history (shift register) to XOR them together. The result is used to index the table of prediction bits. Note that you are free to choose any number of lower order bits for the branch address (as well as for the branch history register), in order to obtain the index for the history table. For a given index, the corresponding prediction bits are used to predict the branch outcome, similar to the 2-bit saturating counter.

## Gselect Predictor

Gselect predictor is similar to the Gshare predictor. The only difference is that, instead of XOR-ing the global history and the branch address (i.e., PC), it concatenates them to obtain the index of the prediction table. Like Gshare, the prediction bits for a given index will act identical to a 2-bit saturating counter.

## 3 Handout Instructions

Please download the `branch-pred.tar` file from the course wiki page. It's located in the Homework Schedule section.

Start by copying `branch-pred.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
unix> tar xvf branch-pred.tar
```

This will cause a number of files to be unpacked in the directory. The files that you will be modifying and turning in are `predictor.cc` and (optionally) `predictor.h`. You will implement the branch predictors inside the function `PredictorRunACycle()`. Please go through the README file for the instructions on how to compile your code and how to run the program executable. Note that you can consider different table sizes for different predictors to obtain the best performances for each predictor.

*You will evaluate the performance of your predictors based on the 16 zipped traces available in the course Canvas page as a single zipped file.*

Extract that zipped file in your working directory as follows, and you will see the 16 zipped (.bz2) traces.

```
unix> tar xvzf cbp3_traces1.tar.gz
```

## 4 Logistics

This project can be done individually or in a group of 2 students. All handins are electronic. You should work only on your SSD device.

## 5 Evaluation

Your score will be computed out of a maximum of 100 points based on the following distribution:

- 30** Saturating counter performance.
- 30** Gselect predictor performance.
- 30** Gshare predictor performance.
- 10** Coding style and commenting.

## Autograding your work

Once you run your code on all the 16 zipped traces, you can estimate your overall score as follows:

```
unix> ./generateScore.pl --from </path/to/output/txt/files>
```

Please exclude the output file for the sample `trace_test.bz2` trace file from your evaluation. Note that the autograder script assumes that the 16 output files correspond to the 16 zipped traces.

The score is evaluated based on the branch prediction penalties of the three predictors. So, the less the penalty, the better the performance. The autograder parses the *Final Score RunX\_Conditional\_MPPKI* value from the output file, and assigns the scores based on the geomean of those values across all the 16 traces.

### 2-bit Saturating Counter:

- geomean <975: score = 30
- 975 <= geomean <1000: score = 25
- geomean >1000: score = 20

### Gselect Predictor:

- geomean <700: score = 30
- 700 <= geomean <800: score = 25
- geomean >800: score = 20

### Gshare Predictor:

- geomean <580: score = 30
- 580 <= geomean <600: score = 25
- geomean >600: score = 20

## 6 Handin Instructions

Please submit `predictor.cc` and `predictor.h` through Canvas. Make sure your code compiles using the Makefile provided in the handout tar package. The TA will evaluate the performance of your predictor implementations with the 16 traces provided in the Canvas, and using the script `generateScore.pl`. PLEASE DO MENTION the name of your teammate in the comment section during the submission. Every student needs to make a submission.

## 7 Hints

- In the code, the address of an instruction is indicated by *uop->pc*. You can find out whether a branch has actually been taken or not, by using the variable *uop->br\_taken*.
- Update the fetch branch history register (*brh\_fetch*) at the end of the for-loop indicating the fetch stage. Similarly, update the retire branch history register (*brh\_retire*) at the end of the for-loop indicating the retire stage. Use these registers in the respective loops to obtain the index of the respective history tables for Gselect and Gshare.
- To update the history registers, left shift the register variable by one, and OR it with the outcome (i.e., taken or not) of the conditional branch.
- You are free to choose a separate history table with different size for each type of predictor, in order to obtain an optimal accuracy.
- You may use the functions *PredictorInit()* and *PredictorReset()* to declare and initialize the history table for each predictor.
- You may have a look at the following study material for the branch predictors. Page 11 describes why Gshare can outperform Gselect.

<http://inst.eecs.berkeley.edu/cs252/sp17/papers/McFarling-WRL-TN-36.pdf>