NC State University

Department of Electrical and Computer Engineering

ECE 463/563 (Prof. Rotenberg)

Project #2: Branch Prediction (Version 1.0)

Due: Thursday, November 7, 2024, 11:59 PM

1. Preliminary Information

1.1. Academic integrity

- Academic integrity:
  - **No collaboration whatsoever:** Each student must work individually on their project. There is zero tolerance for collaboration on the project in any form.
  - **Source code:** Each student must design and write their source code alone. They must do this (design and write their source code) without the assistance of any other person in ECE 463/563 or not in ECE 463/563. They must do this (design and write their source code) without searching the web for past semesters' projects and without searching the web for source code with similar goals (*e.g.*, modeling computer architecture components such as caches, predictors, pipelines, *etc*.), which is strictly forbidden. They must do this (design and write their source code) without looking at anyone else's source code, without obtaining electronic or printed copies of anyone else's source code, *etc*. Use of ChatGPT or other generative AI tools is prohibited.
  - **Explicit debugging:** With respect to "explicit debugging" as part of the coding process (*e.g.*, using a debugger, inspecting code for bugs, inserting prints in the code, iteratively applying fixes, *etc*.), each student must explicitly debug their code without the assistance of any other person in ECE 463/563 or not in ECE 463/563.
  - **Report:** Each student must run their own experiments, collect and process their own data, and write their own report. Plagiarism (lifting text, graphs, illustrations, *etc*., from someone else, whether one adjusts the originals or not) is prohibited. Using someone else's data (in raw form and/or graph form) is prohibited. Fabricating data is prohibited.
  - **Sanctions:** The sanctions for violating the academic integrity policy are (1) a score of 0 on the project and (2) academic integrity probation for a first-time offense or suspension for a subsequent offense (the latter sanctions are administered by the Office of Student Conduct). Note that, when it comes to academic integrity violations, both the giver and receiver of aid are responsible and both are sanctioned. Please see the following RAIV form which has links to various policies and procedures and gives a sense of how academic integrity violations are confronted, documented, and sanctioned: RAIV form.
  - **Enforcement:** The TAs will scan source code (from current and past semesters) through tools available to us for detecting cheating. The outputs from these tools, combined with in-depth manual analysis of these outputs, will be the basis for investigating suspected academic integrity violations. TAs will identify suspected plagiarism and/or data fabrication and these cases will be investigated.

- Reasonable assistance: If a student has any doubts or questions, or if a student is stumped by a bug, the student is encouraged to seek assistance using both of the following channels.
    - Students may receive assistance from the TAs and instructor.
    - Students are encouraged to post their doubts, questions, and obstacles, on the Moodle message board for this project. The instructor and TAs will moderate the message board to ensure that reasonable assistance is provided to the student. To strictly abide by the "no collaboration policy", students may not reply to other students' project queries unless the instructor solicits input from other students for a particular query.

        * An example of reasonable assistance via the message board: Student A: "*I'm encountering the following problem: I'm getting fewer writebacks from L2 to main memory than the validation runs. What might I be doing wrong?*" Instructor/TA: "*Maybe you have implemented evictions in more than one place in your code rather than consolidating it into one place (efficient and less bug-prone). If so, make sure you are checking for and counting evicted dirty blocks at all such locations (or try to implement evictions at one place in the code).*"

        * Another example of a reasonable exchange: Student A: "*I'm unsure how to split the address into tag and index, and how to discard the block offset bits. I've successfully computed the # bits for each but I am stuck on how to manipulate the address in C/C++. Do you have any advice?*" Instructor/TA: "*We suggest you use an unsigned integer type for the address and use bitwise manipulation, such as ANDs (&), ORs (|), and left/right shifts (<<. >>) to extract values from the unsigned integer, the same as one would do in Verilog.*"

        * Another example of a reasonable exchange: Student A: "*I'm unsure how to dynamically allocate memory, such as dynamically allocating a 1D array of structs/classes or (more appropriately for a cache) a 2D array of structs/classes. Can you point me to some references on this?*" Instructor/TA: "*Sure, here is a web site or reference book that discusses dynamic memory allocation including 1D and 2D arrays.*"

1.2. Reduced project scope for ECE 463 students

The project scope is reduced but still substantial for ECE 463 students, as detailed in this specification.

1.3. Programming languages for this project

You must implement your project using the C, C++, or Java languages, for two reasons. First, these languages are preferred for computer architecture performance modeling. Second, our Gradescope autograder only supports compilation of these languages.

1.4. Responsibility for self-grading your project via Gradescope

You will submit, validate, and SELF-GRADE your project via Gradescope; the TAs will only manually grade the report. While you are developing your simulator, you are required to frequently check via Gradescope that your code compiles, runs, and gives expected outputs with respect to your current progress. This is necessary to resolve porting issues in a timely fashion (*i.e.*, well before the deadline), caused by different compiler versions in your programming environment and the Gradescope backend. This is also necessary to resolve non-compliance issues (*i.e.*, how you specify the simulator's command-line arguments, how you format the simulator's outputs, *etc*.) in a timely fashion (*i.e.*, well before the deadline).

2. Project Description

In this project, you will construct a branch predictor simulator and use it to evaluate different configurations of branch predictors.

3. Simulator Specification

3.1. ECE 463 and ECE 563 students: Branch predictors

Model a *gshare* branch predictor with parameters {*m*, *n*}, where:

- *m* is the number of low-order PC bits used to form the prediction table index. **Note**: discard the lowest two bits of the PC, since these are always zero, *i.e.*, use bits ***m*+1** through **2** of the PC.
- *n* is the number of bits in the global branch history register. **Note**: $n \le m$. **Note**: *n* may equal zero, in which case we have the simple *bimodal* branch predictor.

3.1.1. n=0: bimodal branch predictor

When n=0, the *gshare* predictor reduces to a simple *bimodal* predictor. In this case, the index is based on only the branch's PC, as shown in Fig. 1 below.

Entry in the prediction table:
An entry in the prediction table contains a single 2-bit counter. All entries in the prediction table should be initialized to **2** ("weakly taken") when the simulation begins.

Regarding branch interference:
Different branches may index the same entry in the prediction table. This is called "interference". Interference is not explicitly detected or avoided: it just happens. (There is no tag array, no tag checking, and no "miss" signal for the prediction table!)

Steps:
When you get a branch from the trace file, there are three steps:
(1) Determine the branch's **index** into the prediction table.
(2) Make a prediction. Use **index** to get the branch's counter from the prediction table. If the counter value is greater than or equal to **2**, then the branch is predicted *taken*, else it is predicted *not-taken*.
(3) Update the branch predictor based on the branch's actual outcome. The branch's counter in the prediction table is incremented if the branch was taken, decremented if the branch was not-taken. The counter *saturates* at the extremes (**0** and **3**), however.

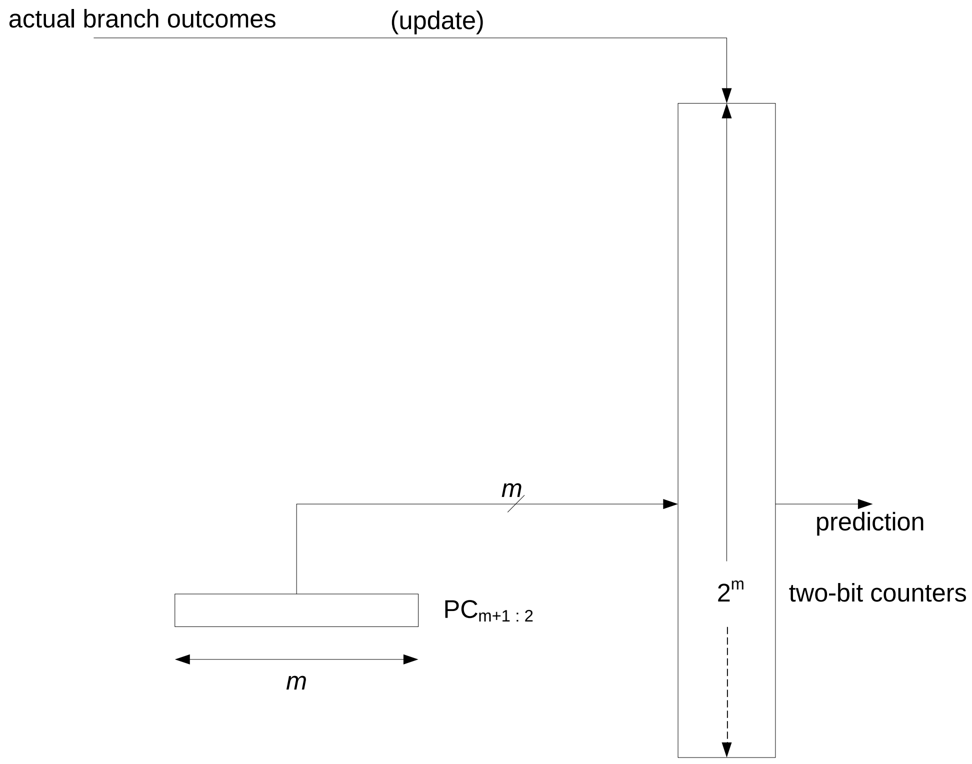

Figure 1. Bimodal branch predictor.

3.1.2. n>0: gshare branch predictor

When n > 0, there is an n-bit global branch history register. In this case, the index is based on both the branch's PC and the global branch history register, as shown in Fig. 2 below. The global branch history register is initialized to all zeroes (00...0) at the beginning of the simulation.

<u>Steps:</u>
When you get a branch from the trace file, there are four steps:
(1) Determine the branch's **index** into the prediction table. Fig. 2 shows how to generate the index: the current n-bit global branch history register is XORed with the uppermost n bits of the m PC bits.
(2) Make a prediction. Use **index** to get the branch's counter from the prediction table. If the counter value is greater than or equal to **2**, then the branch is predicted *taken*, else it is predicted *not-taken*.
(3) Update the branch predictor based on the branch's actual outcome. The branch's counter in the prediction table is incremented if the branch was taken, decremented if the branch was not-taken. The counter *saturates* at the extremes (**0** and **3**), however.
(4) Update the global branch history register. Shift the register right by 1 bit position, and place the branch's actual outcome into the most-significant bit position of the register.

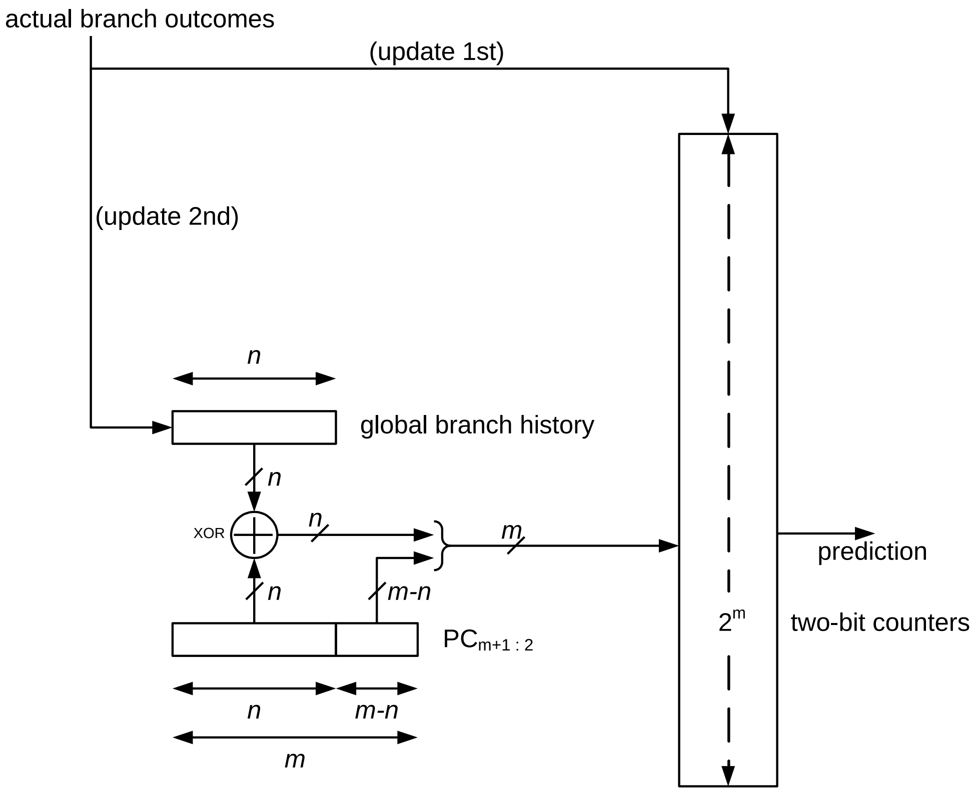

Figure 2. Gshare branch predictor.

3.2. ECE 563 students only: Hybrid branch predictor

Model a **hybrid predictor** that selects between the *bimodal* and the *gshare* predictors, using a chooser table of 2^k 2-bit counters. All counters in the chooser table are initialized to **1** at the beginning of the simulation.

Steps:

When you get a branch from the trace file, there are six top-level steps:

(1) Obtain two predictions, one from the *gshare* predictor (follow steps 1 and 2 in Section 3.1.2) and one from the *bimodal* predictor (follow steps 1 and 2 in Section 3.1.1).

(2) Determine the branch's **index** into the chooser table. The index for the chooser table is bit ***k*+1** to bit **2** of the branch PC (*i.e.*, as before, discard the lowest two bits of the PC).

(3) Make an overall prediction. Use **index** to get the branch's chooser counter from the chooser table. If the chooser counter value is greater than or equal to **2**, then use the prediction that was obtained from the *gshare* predictor, otherwise use the prediction that was obtained from the *bimodal* predictor.

(4) Update the selected branch predictor based on the branch's actual outcome. Only the branch predictor that was selected in step 3, above, is updated (if *gshare* was selected, follow step 3 in Section 3.1.2, otherwise follow step 3 in Section 3.1.1).

(5) Note that the *gshare's* global branch history register must always be updated, even if *bimodal* was selected (follow step 4 in Section 3.1.2).

(6) Update the branch's chooser counter using the following rule:

	Results from predictors:		
	both incorrect ***or*** ***both correct***	***gshare correct,*** ***bimodal incorrect***	***bimodal correct,*** ***gshare incorrect***
Chooser counter update policy:	no change	increment (but saturates at 3)	decrement (but saturates at 0)

4. Inputs to Simulator

4.1. Traces

Traces are posted on the Moodle site. The simulator reads a trace file in the following format:

```
<hex branch PC> t|n
<hex branch PC> t|n
...
```

Where:

- `<hex branch PC>` is the address of the branch instruction in memory. This field is used to index into predictors.
- `"t"` indicates the branch is <u>actually taken</u> (Note! <u>*Not*</u> that it is predicted taken!). Similarly, `"n"` indicates the branch is actually not-taken.

Example:

```
00a3b5fc t
00a3b604 t
00a3b60c n
...
```

4.2. Command-line arguments to the simulator

The simulator executable built by your Makefile must be named "sim" (the Makefile is discussed in Section 6).

Your simulator must accept command-line arguments as follows: (Note: <tracefile> is the filename of the input trace.)

To simulate a bimodal predictor: **`sim bimodal <M2> <tracefile>`**, where M2 is the number of PC bits used to index the bimodal table.

To simulate a gshare predictor: **`sim gshare <M1> <N> <tracefile>`**, where M1 and N are the number of PC bits and global branch history register bits used to index the gshare table, respectively.

To simulate a hybrid predictor: **`sim hybrid <K> <M1> <N> <M2> <tracefile>`**, where K is the number of PC bits used to index the chooser table, M1 and N are the number of PC bits and global branch history register bits used to index the gshare table (respectively), and M2 is the number of PC bits used to index the bimodal table.

5. Outputs from Simulator

Your simulator should output the following:
(See Section 6 regarding the formatting of these outputs and validating your simulator.)

1. The simulator command (which indicates the branch predictor configuration and trace file).
2. The following measurements:
   a. number of predictions (*i.e.*, number of dynamic branches in the trace)
   b. number of mispredictions (predicted *taken* when *not-taken*, or predicted *not-taken* when *taken*)
   c. misprediction rate (# mispredictions/# predictions)
3. The final contents of the branch predictor.

6. Submit, Validate, and Self-Grade with Gradescope

Sample simulation outputs are provided on the Moodle site. These are called “validation runs”. **Refer to the validation runs to see how to format the outputs of your simulator.**

You must submit, validate, and self-grade[1] your project using Gradescope. Here is how Gradescope (1) receives your project (zip file), (2) compiles your simulator (Makefile), and (3) runs and checks your simulator (arguments, print-to-console requirement, and “diff -iw”):

1. **How Gradescope receives your project: zip file.** While you are developing your simulator, you may continuously submit new zip files to Gradescope containing the latest version of your project. The latest submission is the one that is considered for your grade. Gradescope will accept a zip file consisting of three things: your source code, a Makefile to compile your source code, and your project report. In the early stages of your project, before creating the report, your zip file will have only source code and a Makefile. Once the report is completed, your zip file will contain everything.

   - Report (included in the zip file once available): The report must be a PDF file named “report.pdf” located at the top level of the zip file, because that is what Gradescope looks for when checking completeness of the submission. *See Section 7 and the required report template in Moodle for the required contents of the report.*
   - Makefile: The Makefile must be at the top level of the zip file, because Gradescope runs “make” with the expectation that the Makefile is at the top level.
   - Source code: Whether your source code is at the top level of the zip file or in directories below the top level, your Makefile must be designed to compile your source code, accordingly.

2. **How Gradescope compiles your simulator: Makefile.** Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named “sim”. An example Makefile is posted on the Moodle site, which you can copy and modify for your needs.

3. **How Gradescope runs and checks your simulator: arguments, print-to-console requirement, “diff -iw”, and timeout.**

   - Your simulator executable (created by your Makefile) must be named “sim” and take command-line arguments in the manner specified in Section 4.2, because Gradescope assumes these things.
   - Your simulator must print outputs to the console (*i.e.*, to the screen), because Gradescope assumes this.
   - Your output must match the validation runs both numerically and in terms of formatting, because Gradescope runs “diff -iw” to compare your output with the correct output. The -iw flags tell “diff” to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the validation

[1] The mystery runs component of your grade will not be published until we release it. The report will be manually graded by the TAs.

runs have whitespace. Note, however, that extra or missing blank lines are NOT ok: “diff -iw” does not ignore extra or missing blank lines.

- Gradescope’s autograder has a timeout for compiling the simulator and running all tests. The default timeout is 10 minutes. This is usually ample time for this course. If the autograder times out for your project (very inefficient simulator or a bug that causes deadlock), you will see a grade of zero for that submission. Please see Section 9.2 regarding optimizing run time. Also seek advice from the instructor and TAs, as needed.

7. Grading Breakdown, Experiments, and Report

See the required report template in Moodle for the grading breakdown, experiments, and report contents. Use the report template as the basis for the report that you submit (insert graphs, fill in answers to questions, *etc*.).

8. Penalties

Various deductions (out of 100 points):

-1 point for each day (24-hour period) late, according to the Gradescope timestamp. The late penalty is pro-rated on an hourly basis: -1/24 point for each hour late. We will use the “ceiling” function of the lateness time to get to the next higher hour, *e.g.*, ceiling(10 min. late) = 1 hour late, ceiling(1 hr, 10 min. late) = 2 hours late, and so forth. **For this second project, Gradescope will accept late submissions no more than one week after the deadline. The goal of this policy is to encourage forward progress for other work in the class.**

See Section 1.1 for penalties and sanctions for academic integrity violations.

9. Advice on backups and run time

9.1. Keeping backups

It is good practice to frequently make backups of all your project files, including source code, your report, *etc*. You can backup files to another hard drive (your NFS B: drive in your NCSU account, home PC, laptop … keep consistent copies in multiple places) or removable media (flash drive, *etc*.).

9.2. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important because you will be running many experiments: many branch predictor configurations and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast.

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the Moodle site includes the –O3 optimization flag.

Note that, when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and with –g. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The –g flag tells the compiler to include symbols (variable names, *etc*.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, *etc*. When you are done debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

As mentioned in Section 6, another reason for being wary of excessive run times is Gradescope's autograder timeout.