MIPS Simulator: Branch Predictor
Project 4 – CS 3339 – Spring 2019

Due Date per TRACS. Friday before 11:55pm, late until Saturday noon -10pts, after that no submissions accepted.

Submission must be in tar format and compile on zeus per instructions to be graded.

PROBLEM STATEMENT

In this project, you will reduce the number of pipeline flushes required by adding a branch predictor to your simulator. This predictor will predict only conditional branch instructions, i.e. *beq* and *bne*.

You should begin with a copy of your Project 3 submission. I have provided a new Makefile as well as a class skeleton for a BranchPred class intended to be instantiated inside your existing CPU class.

You should simulate a dynamic branch predictor that uses a 64-entry direct-mapped table with no tag and no valid bits. Each entry stores a 2-bit saturating counter (described below) used to generate a "taken" or "not-taken" prediction, as well as a target PC (i.e., the PC the branch is predicted to jump to if it is taken). The table should be indexed as follows: `index = (PC`$_{branch}$` >> 2) % BPRED_SIZE`. (I have defined BPRED_SIZE for you in `BranchPred.h`).

All counters should be initialized to zero. Whenever a branch is taken, update the corresponding 2-bit saturating counter by incrementing it, unless it is already at its maximum value (3). Whenever a branch is not taken, update the counter by decrementing it, unless it is already at its minimum value (0). The prediction should be "branch taken" if PC_{branch}'s count is at least 2 and "branch not taken" otherwise.

You may structure the interaction between your CPU and branch predictor however you'd like. However, **be careful: for any branch instruction, first make a prediction, *then* update the predictor**. In other words, you can only use *past* information to make a prediction. To make sure, it's simplest to have separate BranchPred::predict() and BranchPred::update() functions.

Up until now, your processor has flushed the pipeline behind any taken branch instruction. Modify your code so that a flush occurs only when a branch has been ***mispredicted***.

Here's my recommended approach from a CPU.cpp decode of `beq` and `bne` perspective:

1.) Based on the PC, predict whether the operation will be a taken branch and to what address.
2.) Determine its actual direction (taken vs. not-taken) and target PC.
3.) Determine if there was a misprediction. A misprediction can be one of two types:
    a. Direction mispredict: compare the predicted and actual taken vs. not-taken determination
    b. Target mispredict: *if the branch was correctly predicted taken*, compare the predicted and actual target PCs

    If there was a misprediction, flush the pipeline behind the branch as you did for all taken branches in Project 3. For a correct prediction no pipeline flush is necessary.
4.) Update the 2-bit saturating counter for the branch's PC, based on its actual taken or not-taken determination. *If the branch was taken*, also update the predictor's target PC.

Your branch predictor model will calculate and report the following statistics:

- The total number of **predicted branches**
- The number of branches that were **predicted taken** vs. **predicted not-taken**
- The total number of **mispredictions**
- The number of **direction mispredictions** (i.e., T vs. NT decision was wrong)
- The number of **target mispredictions** (i.e., the branch was correctly predicted taken but the target PC was wrong)
- The **overall predictor accuracy**

I have provided the following new files:

- A `BranchPred.h` class specification file, which you'll need to add member variables and function prototypes to
- A `BranchPred.cpp` class implementation file, which you should enhance to model the described branch predictor and count predictions and mispredictions
- A new Makefile
- A sample output file `sssp.out` for you to confirm your output and check formatting.
- A debug output for `qsort.mips` in a separate tar file.

In addition to enhancing the `BranchPred.h/.cpp` skeleton, you will also need to modify `CPU.h` to instantiate a BranchPred object, and `CPU.cpp` to call your BranchPred class functions and call Stats::flush() only on mispredictions. You'll also need to change `CPU::printFinalStats()` to match my expected output format (see below).

ASSIGNMENT SPECIFICS

All provided files are on TRACS cs3339_project4bp.tgz.

Begin by copying your project3 files into a new project4 directory. Then extract the additional files in cs3339_project4bp.tgz and add them to your project4 directory. You can compile and run the simulator program identically to previous projects, and test it using the same `*.mips` inputs.

Here are a few steps to get you up and running after setting up your directory.

1) Write your name in the header of BranchPred.cpp and also confirm your name is in your CPU.cpp and Stats.cpp as well.
2) Include BranchPred.h in your CPU.h file and instantiate an object named `bpred`. Refer to the ALU instantiation as an example.
3) At the bottom of CPU.cpp remove the unneeded p3 outputs and add `bpred.printFinalStats();`
4) You should now be able to compile and run your project, but the output will be incorrect. It is highly recommended that your compile frequently as you work on your code.

Since you have not implemented the branch prediction code the output will be incorrect. You should see:

```
Branches predicted: 0
  Pred T: 0 (-nan%)
  Pred NT: 0
Mispredictions: 0 (-nan%)
  Mispredicted direction: 0
  Mispredicted target: 0
Predictor accuracy: -nan%
```

Table 1 contains the expected pre-branch-predictor cycle, flush, and CPI metrics for some of the inputs. ***You should not begin implementing your branch predictor until your baseline code matches these numbers!*** (The other 3 inputs have un-interesting branch behavior and will not be tested).

Input	Cycle Count	Flushes	CPI
hash.mips	134904	6526	3.45
nqueens.mips	629285095	24671130	3.03
prime.mips	659619	54450	3.60
qsort.mips	167458	8334	3.64
sssp.mips	1627234	51990	3.62

Table 1: Initial values before Branch Prediction

The following is the expected output for sssp.mips with branch prediction. Your output must match this format verbatim; you should check it by using `diff` and the provided sssp.out file. Misspellings and capitalization will throw off the grading scripts.

```
CS 3339 MIPS Simulator
Branch Predictor Entries: 64
Running: sssp.mips

 7 1

Program finished at pc = 0x400440  (449513 instructions executed)

Cycles: 1586032
CPI: 3.53

Bubbles: 1125724
Flushes: 10788

Branches predicted: 42954
  Pred T: 25851 (60.2%)
  Pred NT: 17103
Mispredictions: 5393 (12.6%)
  Mispredicted direction: 2803
  Mispredicted target: 2590
Predictor accuracy: 87.4%
```

Table 2 shows the expected cycle count, flush count, CPI, and predictor accuracy after the inclusion of the simulated branch predictor.

Input	Cycle Count	Flushes	CPI	Predictor Accuracy
hash.mips	128836	458	3.29	98.0%
nqueens.mips	611789985	7176020	2.95	92.8%
prime.mips	940807	6482	3.42	91.6%
qsort.mips	160130	1006	3.49	95.6%
sssp.mips	1586032	10788	3.53	87.4%

Table 2: Expected values with Branch Prediction

Additional Requirements:

- **Your code must compile with the given `Makefile` and run on zeus.cs.txstate.edu**
- Your code must be well-commented, sufficient to prove you understand its operation
- Make sure your code doesn't produce unwanted output such as debugging messages. (You can accomplish this by using the `D(x)` macro defined in `Debug.h`)
- Make sure your code's runtime is not excessive
- Make sure your code is correctly indented and uses a consistent coding style
- Clean up your code before submitting i.e., make sure there are no unused variables, unreachable code, etc.

DEBUGGING

Most importantly make sure your initial outputs match the values in Table 1. There is a D(); output in the provided BranchPred.cpp file which will be enabled when debug outputs are enabled in Debug.h. You can use this to trace the operation of your branch predictor; a debug output file is provided for qsort.mips. Remember the debug output files are VERY LARGE be sure to delete them and never run with the nqueen.mips input. Turn off debug outputs before submission.

To redirect your output to a file:

```
$ ./simulator qsort.mips > temp.txt
```

Or if you want to see the output on the screen at the same time use:

```
$ ./simulator qsort.mips | tee temp.txt
```

The following command will show you just the branch instructions:

```
$ grep -B1 -A10 'BPRED' temp.txt | more
```

The grep command extracts all lines containing 'BPRED', the -B1 option displays one line before the match and the -A10 option displays 10 lines after the match so you can see the next instruction and tell whether the branch was taken. 10 points extra-credit to the first person who emails me a simple way to remove the register dump from this output.

SUBMISSION INSTRUCTIONS

Submit all of the code necessary to compile your simulator (all of the `.cpp` and `.h` files as well as the `Makefile`) as a compressed tarball. You can do this using the following Linux command:

```
$ tar czvf yourNetIDproject4.tgz *.cpp *.h Makefile
```

Do not submit the executables (`*.mips` files), outputs (*.out) or any other files. Do not rename any of the files inside of the tar file. Do not leave Debug outputs enabled. Do not submit a zip or OSX compressed archive; only tar files generated with above command will work. Any special instructions or comments to the grader, including notes about features you know do not work, should be included in a separate text file (not inside the tarball) named `yourNetID_README.txt`.

All project files are to be submitted using TRACS. You should not assume your submission has been made until you have a confirmation email from TRACS.

You may submit your file(s) as many times as you'd like before the deadline. Only the last submission will be graded. ***TRACS will not allow submission after the deadline***, so I strongly recommend that you don't come down to the final seconds of the assignment window. Late assignments will not be accepted.

ACADEMIC HONESTY (from course syllabus)

You are expected to adhere to the Texas State University Honor Code which is described here http://www.txstate.edu/honorcodecouncil/Academic-Integrity.html

All assignments and exams must be done individually, unless otherwise noted in the assignment handout. Turning in an exam or assignment that is not entirely your own work is cheating and will not be tolerated.

Group discussion about course content is NOT cheating and is in fact strongly encouraged! You are welcome to study together and to discuss information and concepts covered in class, as well as to offer (or receive) help with debugging or understanding course concepts to (or from) other students. However, this cooperation should never involve students possessing a copy of work done by another student, including solutions from previous semesters, other course sections, the Internet, or any other sources.

Turning in an assignment any part of which is copied from the Internet, another student's work, or any other non-approved source will result in a 0 grade on the assignment and will be reported to the Texas State Honor Code Council. Should one student copy from another, both the student who copied work and the student who gave material to be copied will receive 0s and be reported to the Honor Code Council. You should never grant anyone access to your files or email your programs to anyone (other than the instructor)!