CMU 18-640: Foundations of Computer Architecture

Project 1: Branch Prediction

**** Due 11:59pm, 2/11/2014 ****

1. Preliminaries

In this project, you will model the Alpha 21264's branch predictor in a trace-driven simulator. Your trace-driven simulator will be fed a sequence of program counter and other info about a branch from a trace file. For each branch, your simulated branch predictor should return a branch decision (taken or not taken). After each prediction, your simulated branch predictor will then be given the branch outcome for learning/training.

This project assumes you are proficient in C and Unix. This project must be completed in groups of 3 or 4 students. After forming a group, the designated "lead student" from the team should email the TA (Berkin) with the names and ECE email addresses of all members by January 31. Use the subject heading "18640 Project 1 Group".

2. Branch Traces

A collection of 16 traces can be found in `/afs/ece/class/ece640/project/project1/traces`. They are collected from a suite of integer, floating-point, multimedia, and server workloads. The trace files are in plain-text format. If you wish to write your own traces for testing, branch traces are formatted as "[PC] [Branch Target] [Is Indirect] [Is Conditional] [Is Call] [Is Return] [Is Taken]".

The following is an example branch trace:

```
40ee8a 40eeb8 0 1 0 0 0
40eea9 40eed3 0 0 0 0 1
40eed7 40f306 0 1 0 0 0
40efb4 488420 0 1 0 0 0
40f308 40f34f 0 1 0 0 0
```

Note the branch targets are included in the trace. They can be ignored for this project.

You are provided with the skeleton code of the simulator in `/afs/ece/class/ece640/project/project1/release`. The skeleton code manages trace processing and drives the four functions (prescribed below) to be implemented by you. These four functions are to be compiled with the provided skeleton code to form a complete trace-driven simulator.

void initPredictor()
int getPrediction(branch_record_t *branch)
void updatePredictor(branch_record_t *branch, int is_taken)
void dumpStats()

- **initPredictor( )** is called before any processing begins for you to initialize your data structures.

  int getPrediction(branch_record_t *branch) is called once every branch in the trace. The argument provided is a branch record that contains information on the branch PC. If the branch is predicted to be taken, **getPrediction( )** should return 1, otherwise return 0. The following is a table with the branch record fields:

• uint instruction_addr	• The branch's program counter
• int is_indirect	• 1 if the target is not PC-relative; 0 if it's PC-relative
• int is_conditional	• 1 if the branch is conditional type
• int is_call	• 1 if the branch is a call type
• int is_return	• 1 if the branch is a return type

- **void updatePredictor( )** is called right after **getPrediction( )** to return the actual outcome to your branch predictor for training.

- **void dumpStats( )** is called after all of the branch traces have been processed. Use this function to print out any internal statistics at the end of the simulation.

Additional details will be included in the source package for the skeleton code. The following code fragment gives you an idea of the sequence of events in the simulation loop:

```
{
        initPredictor();

        while(…trace is not EOF…) {
                branch_record_t *branch = <branch from the trace file>
                prediction = getPrediction(branch)
                … check your prediction
                … count statistics
                … notify your predictor of the outcome
                updatePredictor(branch, isTaken);
        }

        … print out stats including your misprediction rate …
        dumpStats();
}
```

3. Alpha 21264 Tournament Predictor

For this project, you are to model a branch predictor based off the Alpha 21264 Tournament Branch Predictor. Please refer to Kessler, R. E., "The Alpha 21264 Microprocessor", IEEE Micro, March/April 1999, pp. 24-36 (available at ieeexplore.ieee.org).

The following is a summary of the predictor design:

Local history table	1024 entries, indexed by PC; each entry is a 10-bit history of a specific branch's set of previous outcomes
Local prediction table	1024 entries, indexed by the 10-bit history from the local history table, each entry is a 3-bit saturating counter
Global history register	A single 12-bit register that encodes the last 12 branch outcomes
Global prediction table	4096 entries, indexed by the global history register, each entry is a 2-bit saturating counter
Chooser prediction table	4096 entries, indexed by the global history register, each entry is a 2-bit saturating counter

The local history table maintains history entries for individual branches. Each 10-bit history (which encodes the last 10 outcomes for this branch) is used to index into a local prediction table, which

maintains entries that have 3-bit saturating counters. When a 3-bit saturating counter reaches 100_2 or above, a "taken" prediction is made.

The global history register maintains a single 12-bit history (which encodes the outcomes for the previous 12 branches) used to index into the global prediction table, which has 2-bit saturating counters per entry. When a 2-bit saturating counter reaches 10_2 or above, a "taken" prediction is made.

The local and global predictions are compared against one another, and if they agree, the prediction is made. However, if they disagree, they should consult the chooser prediction table, which is also indexed by the global history register. If the saturating counter is 10_2 or above, the global predictor's decision should be used; otherwise, the local predictor's decision is picked. The chooser prediction table is updated depending on the outcome of disagreeing predictors. Depending on whether a predictor chose correctly, the saturating counter is incremented or decremented.

Some specific implementation decisions are open to your implementation. If your predictor is designed correctly, it should have an overall branch misprediction rate of at most **5%** (see section 4 on how we compute the branch misprediction rate).

You are allowed to deviate from the baseline design subject to the following restrictions. (Note that you are allowed more storage bits than you need for the baseline design. Use the extra bits well to improve your misprediction rate.)

Design constraints
Total number of storage used must be less than 33792 bits.
All storage tables must have power-of-2 entries
If you employ associativity, it must be 8-way or less
If you employ a random replacement policy, it must be reproducible
Extravagant logic functions (such as dividing/multiplying by non-constants or non-powers of 2) are not allowed

4. Branch Misprediction Rate

The measure of success is the branch misprediction rate for running each benchmark under `/afs/ece/class/ece640/project/project1/traces` once, in sequence. This means adding up all of the branch mispredictions for all of the benchmarks and dividing by the total number of branches for all of the benchmarks. (Do not take the average of the branch misprediction rates of individual traces. That will not yield a meaningful number.)

5. Requirements

The following is a list of what we expect for a completed project:

- All source files
- Makefile
- Readme file
- Project Report

You may use any operating system for your project, but your project will be tested on the ECE Linux cluster (e.g. ece000.ece.cmu.edu). Please make sure your final submission works on those machines. You may use any language as long as you can provide a wrapper that will interface to our object code developed for C (you must use the provided `main.o`).

Please include a README file with your code, which should contain the names of the group members, and e-mail addresses at which you can be contacted in case there are any problems. If there are any other comments about the project (e.g. project does not work), please include it in the README file. A sample README file can be found at `/afs/ece/class/ece640/project/project1/release`.

When testing your predictor, we will issue the command "`make predictor`" against your Makefile; this should build an executable called "`predictor`". We will issue the command "`zcat tracefile.gz | ./predictor`" to run your simulator against a compressed trace file. Please test your project submission on one of the ECE Linux cluster machines.

6. Grading

Your project will be graded according to the following:

40 pts	Predictor builds and runs without crashing
25 pts	Branch misprediction rate of at most 5%
15 pts	Performance based scoring
20 pts	Report

For the first 40 points, your submission should build a simulator that can run the traces without crashing with a branch misprediction rate of less than 10% averaged over all of the traces. For the next 25 points, your branch predictor must achieve a branch misprediction rate less than 5% averaged over all of the traces.

15 points is based on your branch misprediction rate relative to the rest of the class. If you are in the lowest (best) quartile, you will receive the full 15 of 15 points. If you are in the second lowest quartile, you will receive 10 of 15 points. If you are in the third lowest quartile, you will receive 5 of 15 points. If you are in the top quartile, you will receive 0 of 15 points. An exception is that you will receive the score of a lower (better) quartile if your branch misprediction rate is within 0.2% of the highest misprediction rate in that quartile.

Finally, 20 points is based on the quality of your project report. The report should present your branch predictor design (in your own words even if you simply replicated the baseline 21264 design) and document the results of your branch predictor. Pay special attention to highlight the changes you tried and how effective they were. Clearly outline how you allocated your storage budget in the final design. Finally, the report must address the following questions.

1. State three reasons why your trace-driven simulator would not perfectly reflect how you predictor would perform if implemented in a real processor pipeline.
2. What is aliasing? Describe cases where aliasing is desirable and undesirable.
3. Does increasing the number of bits in a saturation counter always improve performance? Give an example to support your position.

7. Submission

Copy all relevant submission files to the directory `/afs/ece/class/ece640/project/project1/submission/lead_student_ece_id`. Only the lead student can submit. The directory is unlocked until the deadline so you can make changes until then if you submit early.