CS 6810 Homework #3

Due: 9:10 a.m. Oct. 1 (no late submissions will be graded)

General instructions: Same as usual but since this assignment involves programming you will hand in your code and output file using the CADE lab handin facility mechanism. Just in case you are NOT a member of the cs6810 group for some reason, you should test that you can use the handin script IN ADVANCE (as in try it NOW) and send email to ald if you have a problem. Create an empty dummy file called dummy.txt and hand it in. You won't need to delete it since it will be ignored. For those not familiar with the CADE handin process there are 2 ways you may hand in files:

1. Using a web interface via https://cgi.eng.utah.edu
   - then just follow the instructions
2. On a CADE Linux machine using the command line:
   - e.g. handin cs6810 HW3 file1.c file2.txt

Each student will have a directory automatically generated for you by the handin script and the 2 files that you will hand in will have a specific format as specified in the problem statement.

It's time to get past the manual problem solving bit – this problem will require you to write some code to investigate a variety of branch prediction strategies on a given set of data. Branch prediction is a HUGE influence on performance and hence this is the focus of this assignment. There is a bit of a wrinkle here to hopefully force you to write your modeling code EARLY and then use it to evaluate a different branch trace that will be available on Monday Sept. 28.

Problem 1: [100 points] Branch Prediction - BHT

The purpose of this question is to force you to understand the branch prediction world a little more deeply than the read, listen, and regurgitate style of assignment. Hopefully it will be more fun than frustrating. Simulating some things like scoreboarding or Tomasulo's algorithm is a bit too complex for a 1.5 week programming assignment but branch prediction is not so bad and the CD that came with your text has example code for Case Study 2 which models a BTB style predictor – hence if you're confused about how to get started take a look at this code even though it's for a different type of predictor.

In the CD that came with your text in Content>CaseStudyResources>Ch02, you will find a file called **history.txt** – this will be the early test case for your code. The format of history.txt is described in Case Study 2 in Chapter 2 of your text. A different **real-history.txt** file will be made available to you on Sept. 28 and it will have the same format as history.txt. Note that **real-history.txt** may have a different number of entries than **history.txt**.

Write a program in C or Java that models a variety of branch prediction strategies that you should know about at this point. Make sure your program is a single file and only uses standard C libraries, or whatever the Java equivalent is. The assignment also covers a variety of branch history table sizes. If you forgot, then check out lecture 7 to refresh your memory. The variant that you will be evaluating will depend on four global arrays that you will declare in your program (if it's in C) as follows:

```
int hist_file[]= { 0, 1 };
int bht_size[]= { 256, 512, 1024, 2048, 4096 };
int pred_type[]= { 0, 1 };
int glt[] = { 0, 1, 2 };
```

where the meanings of the values in these arrays are as follows:

1. **hist_file**
   - 0: use history.txt as your input trace file
   - 1: use real-history.txt as your input trace file
2. **bht_size**
   - the value indicates the size of the branch history table that you will be modeling
3. **pred_type**
   - 0: use a 2 bit saturating counter for your predictor
   - 1: use the finite state machine shown in Fig 2.4 of your text as your 2-bit predictor
4. **glt**
   - 0: you will be modeling a Gselect local predictor
   - 1: you will be modeling a Gshare global predictor
   - 2: you will be modeling a Tournament predictor

Again refer to lecture 7 if you're confused about the terminology.

If we take the 4 arrays and view them as sets your code will be modeling a variety of experiments which are the Cartesian product of these sets e.g. [0:256:1:2] means you will be using history.txt as the input file, your bht size will be 256 entries, you will be using the finite state machine shown in Figure 2.4 of your text as the 2-bit predictor model, and you will be modeling a tournament predictor. Namely you will be modeling 30 different predictor configurations on 2 versions of the trace files - labeled as:.

[0:256:0:0]
[0:256:0:1]
[0:256:0:2]
[0:256:1:0]
…
[1:4096:1:2]

To simplify things somewhat – you will always just use 6 bits of local history for the Gselect model, and your hash function will be a simple XOR exactly like the description in Lecture 7. The size of your tournament predictor will be the same as your bht_size.

You will also be running your models in 2 different modes let's call them Hot and Cold. For Cold you will start out with empty BHT's and when you see the branch for the first time you will assume not-taken for a forward branch and taken for a backward branch which implies that your 2-bit predictor will be initialized to 10 for a backward branch and 01 for a forward branch whether you are using the saturating counter or the FSM predictor. You will then take a look at the 3rd datum in the history files to learn whether the branch was taken or not and update your predictor accordingly. For the Hot mode, you will first run the Cold mode over the history tables to warm up your tables and then run through the history file again to compute the Hot statistics.

In order to simplify things to some reasonable level (I hope) we will ignore computing metrics involving filled and unfilled branch delay slots and take the simple view that mispredicts are bad.

Organize you code anyway you want but you source code will be named **bp.c**, and your output file will be an ASCII text file named **bp.txt**. Your **bp.txt** file will be 61 tab delimited rows – the first row will be a set of column headers in the following order:

Config, C_MISP, C_Culprit, C_Culprit%, C-Eff , H_MISP, H_Culprit, H_Culprit%, H-Eff

Where these column entries mean:

- Config – the configuration you are running with the obvious labels that will appear as the first entry in the subsequent 30 rows of **bp.txt**:
    - [0:256:0:0]
    - [0:256:0:1]
    - [0:256:0:2]
    - [0:256:1:0]
    - …
    - [1:4096:1:2]
- C_MISP – is the total number of cold mispredictions for the associated history file
- C_Culprit – is the instruction address of the branch causing the largest number of mispredicts in cold mode
- C_Culprit% - is the % of total cold mispredicts caused by the culprit
- C-Eff – is the cold efficiency. In general performance is proportional to 1/total_mispredicts; cost is proportional to the amount of state (or number of bits) that you need to implement your branch predictor raised to some power; and efficiency is performance/cost. Hence C-Eff is 1/(C_MISP * (# of

bits in your predictor)**1.4). Note that there is a lot of logic involved in implementing a predictor and the cost of the logic - particularly the PC comparators is non-negligible. The 1.4 exponent on the # of bits term has been shown to be approximately correct in terms of cost, where cost in this case is a mix of chip area and delay. If cost includes power it gets even worse but let's simply use the 1.4 exponent for this exercise.

- The remaining four entries are for your second pass over the history files with your prediction tables warmed up and have the same meaning as the previous 4 row entries but for the Hot mode.

Hints: As you will see in history.txt, instruction addresses are 32-bits wide, the machine is byte addressable but the addresses shown in history.txt are not 32-bit word aligned which is weird so unlike what one would normally do, don't ignore the low order 2 bits of the instruction address to index your prediction tables.

Note – as mentioned in class we will use a parse tree tool to compare different student codes and we will re-run your code and compare the bp.txt that you handed in with the one that we generate by running your code. Again – you are encouraged to work in groups to discuss the basic solution approach but you are expected to write your own code**. bp.c must be well commented so that we can understand your reasoning.**

Good luck and hopefully this will be a fun homework.