CS 520: Computer Architecture and Organization
Fall 2015: Prof. Timothy N. Miller
Homework 5
Deadline: Monday, Nov 23

As always homework assignments are to be worked on individually. Students may discuss questions in order to understand them but may not discuss or share answers.

This homework assignment includes a set of text files to be used as sample input, along with part of the corresponding outputs. You will write programs to parse input files, perform processing, and produce output. Your answers will be in the form of source code. The grader will compile your program and execute it using an input **different** from the one you are provided.

Any programming language is permissible as long as your code is:

- Compilable and can run on the Debian GNU/Linux systems operated by the Computer Science department
- *And* supported by MOSS (http://theory.stanford.edu/%7Eaiken/moss/)

Please include appropriate commands and/or Makefiles necessary to build and run your programs.

Your output will be compared to the reference output using the following command:

```
diff --ignore-all-space reference_output your_output
```

In order to determine if your output is in the correct format, copy the appropriate number of lines of your output to a temporary file and execute the appropriate diff command. If diff does not find any differences, then you have the correct output in the correct format.

Problem 1

A program containing **ten** branches was executed, and a trace was captured of the dynamic instruction sequence. All non-branch instructions have been removed, and branch addresses have been renumbered from 0 to 9. This trace of 10000 branches is provided in "sample_branch_sequence.txt". (There is a program of 10 branches, and the sample input file is a sequence of 10000 executions of those branches.)

Your task is to write a **tournament branch predictor** that attempts to predict these branches. The tournament branch predictor is described in the class notes on pages 258 and 259, and a diagram is provided below. You are also encouraged to read the Wikipedia article at http://en.wikipedia.org/wiki/Branch_prediction, where this is referred to as a "Hybrid predictor." Two-bit saturating counters are described on pages 238 to 240. To understand global predictors, read pages 248 to 254, and consider the case where k=0.

Step 1: Parse the input file. The input file is in the following format:

```
0n1
1t7
7n8
8t3
3n4
```

On each line:

- The first character is the address of the branch being executed.
- The second character is 'n' if the branch was not resolved to be taken or 't' if the branch was resolved to be taken. This field indicates the true (non-speculative) direction of the branch.
- The third character is the address of the next instruction executed (the target if taken, the fall-through address otherwise).

Step 2: Predict branches. This is to be a tournament predictor, which is comprised of four parts.

Part A is a local predictor: An array of 2-bit saturating counters, indexed by the instruction address. All counters initialize to zero.

Part B is a global predictor: An array of 2-bit saturating counters, indexed by the sequence of **six resolved branch** that occurred in time just prior to the branch being predicted. (This predictor considers only global history, not the PC of the branch.) All counters initialize to zero, and the running branch history initializes to six not-taken branches.

Part C is a selector: An array of 2-bit saturating counters, indexed by the instruction address, used to select between the local and global predictors when making a final prediction. All counters initialize to zero.

Part D is the tournament selection: Use the selector to choose between the local and global predictors to get the final prediction.

You will compare each of your predictions to the corresponding branch resolution. After making each prediction, counters from each of your three predictors must be updated to learn from the newly resolved branch. Your program should keep track of the number of correctly predicted branches for the local predictor, the global predictor, and the tournament predictor.

Note: Your tournament predictor will not be 100% accurate. The assignment is not to correctly predict every branch but instead to match the behavior of the tournament predictor specified above.

Updating counters:

Based on the true outcome of each branch, updates to counters are as follows:

- For the local and global predictors:
    - If the branch was taken, increment the saturating counter. If the branch was not taken, decrement the counter.

- For the selector:
  - If the local and global predictors agree (right or wrong), do not modify the counter.
  - If the local predictor is correct and the global predictor is incorrect, decrement the counter.
  - If the local predictor is incorrect and the global predictor is correct, increment the counter.

Step 3: Output your results. The output file shall be in the following format. Note that what is shown here is the actual expected output for the first five executed branches.

```
0nnlnn
1nnlnt
7nnlnn
8nnlnt
3nnlnn
```

On each line:

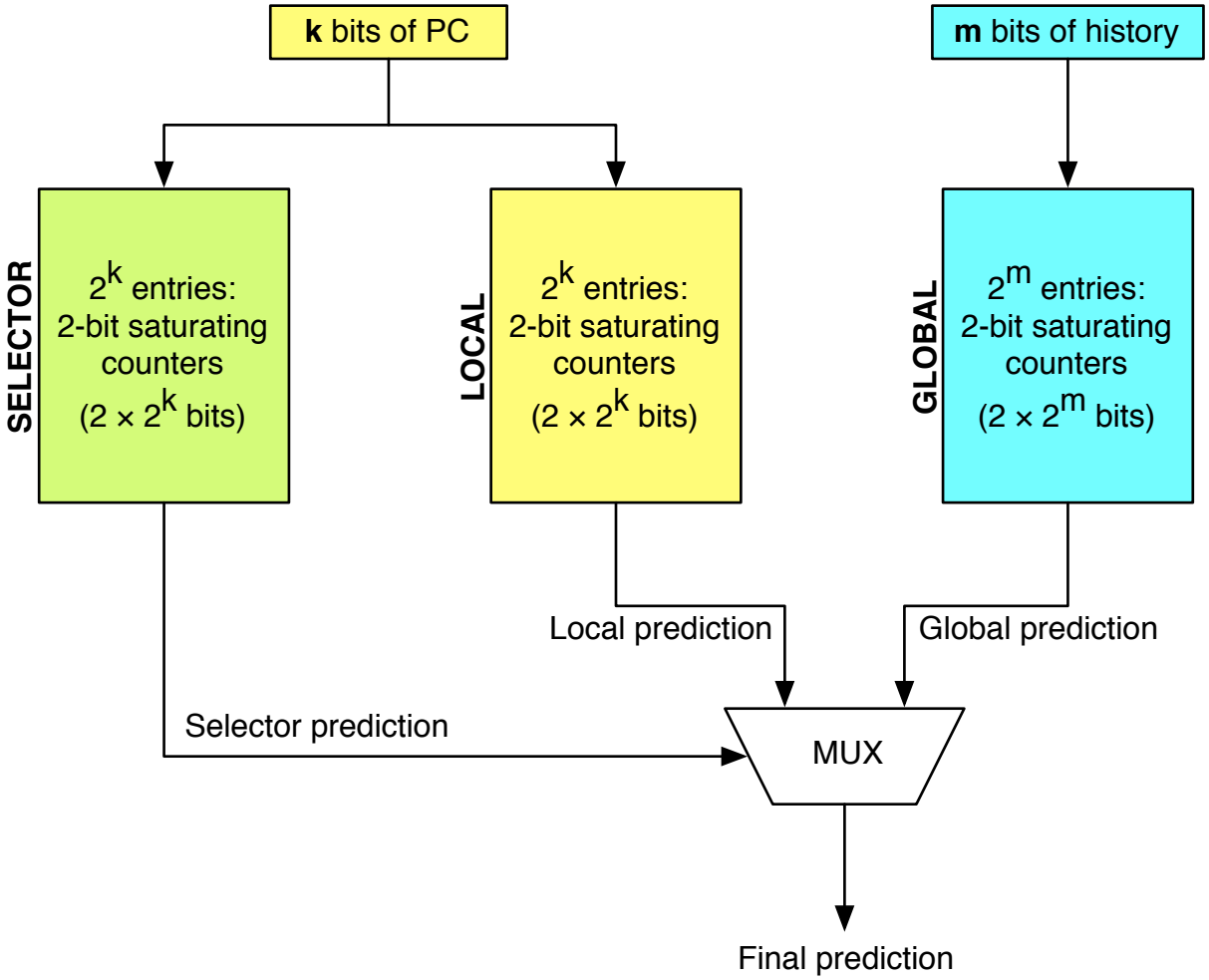
- The first character is the program address of the branch being predicted.
- The second character is the **local** prediction: 't' if taken, 'n' if not taken.
- The third character is the **global** prediction: 't' if taken, 'n' if not taken.
- The fourth character is the **selector** prediction: 'l' if local, 'g' if global.
- The fifth character is the **final** prediction: 't' if taken, 'n' if not taken.
- The sixth character is the **actual direction** of the branch: 't' if taken, 'n' if not taken.

Problem 1.1 (20 points)

Process by hand the first ten branches from the sample input. Provide your answer in the following format:

		Local				Global				Selector					
		Sat. counter				Sat. counter				Sat. counter				Final	Actual
Cycle	PC	Index	Old	New	Pred'n	Index	Old	New	Pred'n	Index	Old	New	Pred'n	Pred'n	Res'n
0	0				N				N				L	N	N
1	1				N				N				L	N	T
2	7				N				N				L	N	N
3	8				N				N				L	N	T
4	3				N				N				L	N	N
5															
6															
7															
8															
9															

Some of the answer is already provided for reference. Provide your global predictor index in **binary**.


k bits of PC
m bits of history
SELECTOR
2^k entries:
2-bit saturating
counters
(2 × 2^k bits)
LOCAL
2^k entries:
2-bit saturating
counters
(2 × 2^k bits)
GLOBAL
2^m entries:
2-bit saturating
counters
(2 × 2^m bits)
Local prediction
Global prediction
Selector prediction
MUX
Final prediction

Problem 1.2 (20 points)

This sub-problem corresponds to the branch predictor program described above, where your answer is in the form of source code.

You are provided with a sample input file, "sample_branch_sequence.txt", which contains a sequence of 10000 branch executions, as described above. You are also provided with the first 50 lines of the correct output in "sample_branch_output.txt". You should use these to test and debug your program. **You will be graded on how your program performs on an entirely different input.**

Your program will take input from stdin, and output to stdout. You may also use stderr for debug info or to report statistics. The grader will execute your program using a command of this general format:

```
{your-command} < input_file > output_file
```

Example:

```
ruby tnm-bpred.rb < branch_sequence.txt > tnm-prediction.txt
```

Source code, README (with instructions for executing your program), and Makefile (optional, for compiled languages) should be bundled in a single tar.gz file, whose name is in this format (note the "p1"):

```
firstname-lastname-hw5p1.tar.gz
```

The command to create this file is:

```
tar zcvf firstname-lastname-hw5p1.tar.gz {your files}
```

Email your file to the grader.

Problem 1.3 (10 points)

This problem pertains to your program run on the sample input file.

A. If your local predictor were used in isolation, how many of the 10000 predictions would have been correct?
B. If your global predictor were used in isolation, how many of the 10000 predictions would have been correct?
C. Using your tournament predictor, how many of the 10000 predictions were correct?

Problem 2

A program was executed, and a trace was captured of its memory accesses. For your convenience, all non-memory instructions have been removed. A trace of 10000 loads and stores is provided in "sample_memory_sequence.txt".

Your task is to write a **DRAM scheduler**. DRAMs are explained in the class notes from page 288 to 300. For this problem, you will be working with a simple single-bank synchronous DRAM (SDRAM). You will schedule a sequence of memory accesses such that the timing characteristics of the SDRAM are respected. See the following for more information:

- http://en.wikipedia.org/wiki/Synchronous_dynamic_random-access_memory
- http://en.wikipedia.org/wiki/Dynamic_random-access_memory#Memory_timing
- http://en.wikipedia.org/wiki/Memory_timings
- http://download.micron.com/pdf/datasheets/dram/sdram/512MbSDRAM.pdf

Step 1: Parse the input file. The input file is in the following format:

```
L04223038
Sfefff288
Sfefff168
Sfefff1b8
L04221cb0
```

On each line:

- The first character is 'L' for a read or 'S' for write.
- The remaining characters are a memory (byte) address in hexadecimal.

Step 2: Schedule memory accesses.
This DRAM can execute the following commands:

- **A**ctivate – Copy a row from the data array into the active row buffer
- **R**ead – Read a data word from the row buffer
- **W**rite – Write a data word to the row buffer
- **P**recharge – Copy a row from the active row buffer back into the data array
- Refresh – Refresh a row (You will not need this command for this problem)
- **I**dle – No command issued (Use this when you cannot issue any other command)

Some commands initiate internal activity in the DRAM chip that takes more than one cycle to complete. In the following chart, the rows are the previous command, the columns are the current command that you want to schedule, and the cells indicate the number of cycles required to wait between the two commands. (A delay of 1 means that commands can be consecutive.)

	A	R	W	P
A	Invalid	tRCD=12	tRCD=12	tRAS=30
R	Invalid	1	1+CL=4	CL=3
W	Invalid	1	1	1
P	tRP=20	Invalid	Invalid	Invalid

Row and column addresses: Given an address from the input file, map the lower 12 bits (3 hex digits) to the column and the upper 20 bits (5 hex digits) to the row.

Initial conditions: No row is active, and the DRAM is completely idle, so that the first non-idle command can be issued on cycle zero.

Step 3: Output your results. The output file will be in the following format:

```
A04223
I11
R038
I17
P
```

On each line:

- The first character is the first letter of the name of the command to execute
- **A** commands are followed by five row-address digits
- **R** and **W** commands are followed by three column-address digits
- **I** commands are followed by the number of cycles to idle
- **P** commands are not followed by any other characters

Problem 2.1 (20 points)

Process the first three memory accesses by hand. For this sub-problem, provide your answer in terms of **clock cycles**; for instance, if you require five idle cycles, then you will have five rows in the table with the Idle command.

Provide your answer in the following format:

Cycle	Command	Address	Output-enable
0	A	04223	F
1	I		F
2	I		F
3			
4			
5			
6			
7			
…			

Some of the answer is already provided here for reference. Your answer will require more rows than provided in this table. The output-enable column is 'True" (asserted) if the SDRAM chip is driving the data bus; otherwise, it is "False." Be sure to carefully consider CAS latency (tCL) when marking which cycles must have output-enable asserted.

Problem 2.2 (20 points)

This sub-problem corresponds to the memory scheduler program described above, where your answer is in the form of source code. (This time, compact each contiguous sequence of idle cycles into a single line, as described in Step 3 above.)

You are provided with a sample input file, "sample_memory_sequence.txt", which contains a sequence of 10000 memory reads and writes, as described above. You are also provided with the first 50 lines of the correct output in "sample_memory_output.txt". You should use these to test and debug your program. You will be graded on how your program performs on an entirely different input.

Your program will take input from stdin, and output to stdout. You may also use stderr for debug info or to report statistics. The grader will execute your program using a command of this general format:

```
{your command} < input_file > output_file
```

Example:

```
python tnm-sdram.py < memory_sequence.txt > tnm-sdram.txt
```

Source code, README (with instructions for executing your program), and Makefile (optional for interpreted languages, required for compiled languages) should be bundled in a single tar.gz file, whose name is in this format (note the "p2"):

```
firstname-lastname-hw5p2.tar.gz
```

The command to create this file is:

```
tar zcvf firstname-lastname-hw5p2.tar.gz {your files}
```

Email your file to the grader.