Homework Assignment 4

CIS501 Fall 2005

Due: November 17, 2005

Instructions: Your solution to this assignment must be type-written, except for the questions that you answer on the provided worksheet.

1. **Branch Prediction**. Consider the following three branch direction predictors: (1) a single Taken/Not-Taken 1-bit predictor, (2) a simple two-entry predictor with 1-bit Taken/Not-Taken entries, (3) a simple two-entry predictor with 2-bit entries (values denoted as "Strongly Taken (T)", "Weakly Taken (t)", "Weakly Not-Taken (n)", and "Strongly Not-Taken (N)", and (4) a two-entry global history predictor using a single bit of history (a single T or N). All instructions are 4-byte fixed-length instructions, so consider the "block size" of the predictor table to be 4 bytes (and use the correct corresponding index bit or bits). Fill in the tables on the separate assignment "worksheet" for the sequence of branches in the tables.

2. **Branch Prediction Coding and Simulation.** Similarly to the cache simulation module for SimpleScalar you wrote for a previous assignment, for this question you will write a simple branch predictor module for SimpleScalar. You will also perform an experiment to show the difference in prediction accuracy for one-bit versus two-bit counters for various predictor sizes.

   The branch predictor code you will write will model a combined branch target buffer (BTB) and directional predictor (that is, a single predictor table is used for both target prediction and direction prediction). The predictor has n entries, and it is indexed using the least significant non-zero bits of the PC (as all Alpha instructions are four-bytes instructions, that means not using the two lowest bits of the PC that are always zero). The table is direct mapped (unlike some BTBs that are set associative), and each entry has: (1) a valid bit, (2) a tag field, (3) a "last target" field, and (4) a counter. You will create two versions of this predictor: a one-bit counter version and a two-bit counter version. The one-bit counters should be initialized to the "not taken" state; the two-bit counters should be initialize to the "weakly not taken" state.

   Branch Predictor Interface. The branch predictor module has three functions:

   ```
   void bpred_init(int num_entries);
   md_addr_t predict_branch(md_addr_t pc);
   void bpred_update(md_addr_t pc, md_addr_t next_pc);
   ```

   The first function initializes the predictor. The second function accesses the predictor to predict the next PC of the instruction (either the predicted target if the branch is predicted "taken" or PC+4 if the target is unknown or the branch is predicted "not taken"). The third function updates the predictor giving the PC and the actual next PC as determined from the execution.

sim-bpred. The SimpleScalar code and branch predictor module template can be found at `~cis501/SimpleScalar/hwk4/` on standard servers such as eniac-l and halfdome. The only two files you need to modify are `bpred_1bc.c` and `bpred_2bc.c`. **Turn in both of these files via BlackBoard.**

To build the predictor, use the following command:

```
make MY_BPRED=bpred_1bc.o
```

Replacing `bpred_1bc.o` with `bpred_2bc.o` as required.

To run the simulator, for example, with a 1024 entry predictor on the `twolf` benchmark:

```
sim-bpred -bpred:entries 1024 programs/twolf.eio
```

The above execution should take a minute or so to run and report a branch misprediction rate of 25% for one-bit counters and 18% for two-bit counters.

Experiments. Run your one-bit and two-bit predictors for both `twolf` and `vpr.route` for at least predictor sizes 128, 512, 2048, 8192 entries (for a total of 16 runs). Although not required, you may run the intermediate sizes of 64, 256, 1k, 4k, and 16k entries (if time allows). To tell SimpleScalar not to print out the intermediate statistics, specify the option "-insn:progress 0" on the command line. To simplifying running these experiments, we've included a `run.sh` script that will run these 16 experiments (which will take 10-20 minutes).

Data Presentation. Report the branch misprediction rate for all of these runs **both in a table and a graph**. The graph should have four lines (one line for each predictor/benchmark combination). The predictor size should be the x axis and the branch misprediction rate should be on the y axis. Be sure to label the graph appropriately (a legend for the lines, labels on each axes, title for the graph, etc.).

Data Analysis. Based on these experiments, which predictor would you choose? Remember, too large a predictor will take up chip area and burn power; too small a predictor will reduce performance due to branch mispredictions.

3. **SuperScalar Execution and Loop Unrolling**. Consider the assembly code below for copying a C-style null-terminated string. R1 and R2 contain the address of the beginning of the source and destination strings, respectively.

```
#1: ld [r1+0] -> r3
#2: st r3 -> [r2+0]
#3: br-zero r3, #7
#4: addi r1, 1 -> r1
#5: addi r2, 1 -> r2
#6: br #1
#7: ...
```

Consider a dual-issue processor with a fully-bypassed 5-stage pipeline and a single-cycle load-use penalty. Assume all branches are predicted perfectly, but the processor cannot fetch a second instruction beyond a taken branch in the same cycle (however, it has "perfect" dual-instruction fetch for sequential code).

To increase performance, you're going to apply loop unrolling to this loop by: replicating instructions #1 through #3, renaming the registers as needed to avoid false dependencies, adjusting the address offsets in the loads and stores, adjusting the amount of the immediate in instructions #4 and #5, and (finally) optimally re-schedule the code. As you need replicate only the first three instructions, this loop with an unrolling factor of n has $3n+3$ instructions per loop body. To ease scheduling the code, you may assume (1) that the source and destination strings do not overlap (and thus you may reorder loads and stores) and (2) the values in R1 and R2 are not used after the loop (and thus their values after the loop terminates do not matter).

Part (a): Optimally schedule the code **without unrolling** it and complete the pipeline diagram on the worksheet provided. What is the CPI (cycles per instruction)? How many "cycles per character" will it take to copy a long string?

Part (b): Optimally schedule the code **with an unrolling factor of two** and complete the pipeline diagram on the worksheet provided. What is the CPI (cycles per instruction)? How many "cycles per character" will it take to copy a long string?

Part (c): Optimally schedule the code **with an unrolling factor of four** and complete the pipeline diagram on the worksheet provided. What is the CPI (cycles per instruction)? How many "cycles per character" will it take to copy a long string?