Handout 10 / Lab 5: Caches and Branch Prediction with a Timing Simulator

Due: Friday, April 6, 2012, 9:20pm

(200 points, done individually)

Objective
In this lab, you will extend a timing simulator in C to model instruction and data caches and branch prediction in a pipelined processor. Unlike the RTL that you have developed in past labs, a timing simulator is not a direct or synthesizable implementation of the processor. Rather, it is a higher-level abstracted model designed to allow quick exploration. Operating at a higher level allows the designer to more quickly see how design choices (such as cache or branch predictor designs) will impact performance.

We will give you the base simulator (it has been developed to match the processor that we have implemented in previous labs). We will also fully specify the behavior of the branch predictor and the caches. Your job is to extend the simulator so that it implements the caches and branch predictor as specified.

Philosophy: High-Level Processor Simulation
Unlike in previous labs, where we built our pipeline in Verilog, we are not constrained to logic-level implementation in a timing simulator in C. Our only goal is to compute the number of cycles a program execution would take on the simulated processor. Because of this, many simplifications are possible. We do not actually need to model control logic and datapath details in each block of the processor; we only need to write code for each stage that performs the relatively high-level function of that pipeline stage (read the register file, access memory, etc.). In general, the simulator’s algorithms and structures do not need to exactly match the processor’s algorithms and structures, as long as the result is the same. While we no longer have a low-level implementation, and thus cannot determine critical path (or other cost metrics that we care about, such as area taken on a silicon chip, or power consumed during operation), we know how well the processor design will perform.

For example, a real hardware implementation of an associative cache will need some logic that determines which block in a given set to replace with a new block when a cache miss occurs and the new block is fetched (often, the least-recently-used block is replaced). Logic to track and determine “least-recently-used” can be complex, especially when the cache is highly associative. In the simulator, the cache model can simply keep an access timestamp on each block, and iterate through the blocks in the set to find the least recently used one. The result will be same as if every register and gate in the hardware were modeled, but the higher-level implementation is both easier to write (thus easier to make correct), and easier to modify when it is time to explore other design options.

Timing-Based Simulator: Structure and Principles
In the lab 5 handout directory (/afs/ece/class/ece447/labs/lab5), we give you a basic timing simulator that models a MIPS pipeline without branch prediction and without an instruction or data cache.

The general structure that we will follow in our simulator is to model each pipeline stage in a separate function, and model the state of the pipeline as a collection of pointers to “pipeline op” structures. Each pipeline op represents one instruction in the pipeline, any associated values, and can carry additional auxiliary information useful to the simulator. A pipeline op structure is allocated when an instruction is fetched, flows through the pipeline, and eventually arrives at the last (writeback) stage, where it is deallocated once the instruction completes. Each stage will typically take its input from some pipeline latch (corresponding to a pointer) and place its output in another pipeline latch. We can model pipeline stalls and flushes in how we manage these pointers. If a stage finds that the pointer corresponding to its output latch is not cleared
(not set to NULL), then this means that the following stage has stalled. Otherwise, the pipeline stage can perform its ordinary processing, move the pointer to the pipeline op from its input to its output, and clear its input so that the previous stage can feed it the next op.

**Timing Details: Caches and Branch Predictors**

As we have described above, your goal is to extend the timing simulator so that it models (i) an instruction cache and a data cache, and (ii) a branch predictor. We will fully specify the behavior of both of these structures now.

- The **instruction cache** is accessed every cycle by the fetch stage. It is a four-way-associative cache that is 8 KB in size, with 32-byte blocks. The cache indexes its 64 sets with bits \([10:5]\) of the address. When a cache access misses, the cache must access main memory to fetch the requested block. An access to main memory takes 50 cycles. When the data returns from memory, the cache replaces the least-recently-used block in the appropriate set with the new cache block. (If any block in the set is not present (has a valid bit set to 0), that block is replaced instead. Initially, no blocks are present in the cache.) The cache insertion happens on the 50th memory access cycle, so that in total, an instruction cache miss results in 50 stall cycles.

If a branch resolution needs to redirect the fetch stage while the instruction cache is in the middle of a memory access due to a cache miss, the PC can be changed and the branch can proceed down the pipeline, and the pending instruction cache miss is canceled. The contents of the cache remain as if the canceled miss had not happened (i.e., the new cache block is not inserted), even if the branch redirection that cancels the cache miss happens in the last stall cycle. Finally, note that a redirection that accesses the same cache block as a currently-pending miss does not cancel the pending miss. (Hint: you might want to implement this behavior in the branch recovery code that runs after each pipeline stage cycle function by checking if any instruction cache miss is pending. You will also need to check in the fetch stage if any branch recovery has been scheduled in the current cycle before inserting a new block, to handle the case where a branch resolves in the last stall cycle when the block would have been inserted.)

- The **data cache** is accessed whenever a load or store instruction is in the memory stage (the fourth pipeline stage). It also uses 32-byte blocks; however, it is eight-way associative and 64 KB in size. Its sets are indexed by bits \([12:5]\) of the address. Miss timing and replacement are identical to the instruction cache: a miss takes 50 cycles in total to service, resulting in 50 stall cycles in total, and the least-recently-used block is replaced (unless a block is not present; initially, no blocks are present).

For the purposes of this lab, assume that no code is self-modifying, so that the caches do not have to deal with the case where the same block is cached in both the instruction and data caches.

- The **branch predictor** consists of a gshare predictor and a direct-mapped 1024-entry branch target buffer. The gshare predictor uses an 8-bit global branch history. The predictor XORs the global branch history (most recent branch in the LSB; 1 = taken, 0 = not-taken) with bits \([9:2]\) of the program counter (i.e., shifts the PC right by 2, then takes the bottom 8 bits), and uses this 8-bit value to index into a single 256-entry PHT. Each entry of the PHT is a 2-bit saturating counter that operates as discussed in class (11 = strongly taken, 10 = weakly taken, 01 = weakly not taken, 00 = strongly not taken; on update, a taken branch increments the value, and a not-taken branch decrements the value). The branch target buffer contains 1024 entries indexed by bits \([11:2]\) of the PC (i.e., shift the PC right by 2, and take the bottom 10 bits). Each entry of the BTB contains (i) a tag, indicating the full PC; (ii) a valid bit; (iii) a bit indicating whether this branch is conditional or unconditional; and (iv) the target of the branch.

In each fetch cycle, the branch predictor computes the predicted next PC from the current PC and branch history as follows. It computes an index into the Pattern History table as specified above. It
loads this entry of the Pattern History Table and computes a taken/not-taken prediction based on the
2-bit entry. The branch predictor also performs a lookup in the BTB using the current PC. If there is
a BTB miss (the entry has a valid bit of “0,” or the tag does not match), then the current instruction
is likely not a branch, or is unknown, so the next PC is predicted as $PC + 4$, regardless of what the
gshare predictor predicts. If there is a BTB hit, and either (i) the BTB entry indicates that the branch
is unconditional or (ii) the BTB entry indicates that the branch is conditional, and the gshare predictor
computed a “taken” prediction, the next PC is predicted as the destination address supplied by the
BTB. Otherwise, the next PC is predicted as $PC + 4$.

The branch predictor tables are always updated in the Execute stage, where all branches are resolved.
The update consists of: (i) updating the corresponding 2-bit saturating counter in the Pattern History
Table, as appropriate; (ii) updating the global branch history; and (iii) updating the Branch Target
Buffer. Unconditional branches do not update the branch history or the PHT, but only the BTB (setting
the “unconditional branch” bit in the corresponding entry).

All branch predictor saturating counters are initialized to 0 (strongly not-taken), and all BTB entries
are initialized to invalid (not present). The global history register is initialized to all not-taken.

Getting Started
We recommend that you begin by getting comfortable with the simulator we have provided. Its shell is
largely the same as in Lab 1. Only the internals have changed, in order to model the processor timing. We
will provide an overview of C-based timing simulation in lab recitations. However, these sessions are not a
substitute for sitting down with the code and reading over it.

The two major tasks of this lab – implementing caches, and implementing branch prediction – are inde-
pendent, so you may tackle them in either order. One way to approach this lab is to first write a generic
implementation of a set-associative cache, and then plug it into both the fetch stage (instruction cache)
and the memory stage (data cache). Once cache accesses work, and stall the pipeline correctly, you can
implement the branch predictor.

As the simulator becomes more complex, it will be very helpful to have a rigorous methodology. We
recommend that you build an extensive suite of tests based on previous labs’ tests, and then use these tests
to validate your simulator against the lab 1 golden simulator.

Handin and Other Requirements
As in previous labs, please hand in your lab electronically in your handin directory:
/afs/ece/classes/ece447/handin/ANDREW-ID/lab5/.

You may modify any file in the simulator source (and/or add new files) except that you may not modify
shell.c. We reserve the right to use a modified shell.c file in order to facilitate autograding.

Grading
As in previous labs, we will grade this lab objectively based on a set of tests which are not released before-
hand. Your simulator must (i) be functionally correct, with respect to the Lab 1 golden simulator, and (ii)
be timing-correct, with respect to the cache and branch predictor definitions that we have given in this lab.
You will receive a certain number of points for functional correctness and for timing correctness on each
test case.

Reference Runs: Sample Timing Results
In order to make timing-correctness testing easier in your own simulator, we will give a few reference runs
here, which give reference cycle counts and number of instructions for several test programs. You should
use these reference runs to ensure that your caches and branch predictor are operating correctly. We may
release more reference runs as the lab progresses. Note that these tests are not exhaustive: we will test your submission with many other programs as well.

All of the benchmarks named here are available as part of the lab distribution.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Cycles</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand_tests/random3</td>
<td>15634</td>
<td>2104</td>
</tr>
<tr>
<td>brtests/test1</td>
<td>11098</td>
<td>1762</td>
</tr>
<tr>
<td>cache_tests/test1</td>
<td>802996</td>
<td>393220</td>
</tr>
<tr>
<td>long_tests/fibonacci</td>
<td>5243011</td>
<td>5242885</td>
</tr>
<tr>
<td>long_tests/primes</td>
<td>2210969</td>
<td>2096285</td>
</tr>
</tbody>
</table>

Extra Credit: Cache Exploration

For this lab, we will offer up to 15% extra credit for anyone who explores cache design with the Lab 5 simulator and turns in a two-part report on:

1. A sweep of cache parameters for a set of memory-intensive benchmarks. You should write a set of benchmarks that use significant amounts of memory (for example: accessing a large array in streaming or random patterns), and run your simulator to find IPC for various cache parameters. Show how changing the associativity, block size, and cache size affect performance.

2. An exploration of cache replacement and/or insertion policies. The cache replacement policy specifies which cache block in a set is replaced when a new block is inserted into the cache. The cache insertion policy specifies where in the list of blocks the new block is placed. Up to now, we have used a replacement policy that evicts (replaces) the least-recently-used block, and an insertion policy that places new blocks at the most-recently-used position. However, other replacement and insertion policies have been studied, and some have been shown to get significantly better performance (fewer cache misses) for certain access patterns [1, 2, 3, 4]. You should experiment with a variety of test programs and optimize the cache replacement/insertion policy. Finally, if you wish to explore further, you may also experiment with other aspects of the cache, e.g. the mapping from addresses to cache sets (see e.g. skewed cache mappings [5]).

The report that you hand in does not need to be more than a page. Simply summarize (i) your overall observations on the effect of each cache parameter (first part), and (ii) your findings on cache replacement algorithms, and specify your final algorithm (second part).

You should turn in a version of your simulator with the modified cache replacement policy to the lab5-ec handin directory that we have created for you. We will evaluate this version of your simulator with a set of test programs that we write, and we will award prizes to the top finishers, as we have for previous labs.

References


