EENG 467 Final Project: Caches

Zeb Mehring

December 18, 2017

I. Overview

The purpose of this project was to design and implement a simple processor cache using Verilog. The design provided is a write-back, write-allocate cache that implements a least-recently used (LRU) replacement policy. As designed, cache hits require one cycle to return the requested data and misses require three cycles.

The data cache implements a 32 KiB, 4-way set associative, 2-word block cache with 32 bit words. The instruction cache implements a 16 KiB, 2-way set associative, 1-word block cache with 32 bit words.

II. Functionality

A. Organization

This implementation stores cache information across multiple registers. These register groupings are logically divided by cache way and by function; each way has its own group (array) of registers for: the valid bit, the dirty bit, the LRU bits, the tag bits, and the data bits. Each register array has a number of entries equal to the number of sets in the cache. The width of each entry is dependent on the size of the data it contains. The module is parameterized so that these sizes are easily adjustable.

B. State Machine

A 1-bit (2 state) state machine is used to process CPU requests, access, and update the data appropriately. The `idle` state compares the address tag to that of the entry with the same index. If one of the tags in the cache set matches the tag of the address supplied by the CPU, the data is returned (or written), the cache reports a hit until the next request, and the state is not changed. Note that the value 0 is returned to the CPU on a successful write. To save cycles in the event of a miss, a request is sent to memory in `idle` for the address provided by the CPU (memory reading is not enabled unless a miss occurs).

If a hit is not made in `idle`, then the machine enters the `miss` state, which writes the least-recently used block back to memory (if it's dirty and there are no invalid blocks), receives the requested data from memory, and writes it to the first invalid (or least-recently used) block in the correct set. The state is then reset to `idle` as if the miss had not occurred.

III. Timing

The timing of the cache is determined by that of the state machine it implements. Hits are performed as fast as possible (in one cycle). The constant index lookup and tag comparison do not depend on any state (register) variables, the success or failure of the search is a boolean value obtained by ORing together the comparison results, and the data can be returned using a multiplexor with the decoded value of the inputs to the OR gate as the select. In this implementation, these features are inferred from conditional statements rather than implemented directly.

To ameliorate the penalty incurred by a miss, a request is sent to memory as each new CPU request is received. This ensures that the requested memory will be available on the next clock edge, if a miss occurs. Note that memory reading is enabled only on a miss. The `miss` state does constant time comparisons of the valid bit in each way and dispatches the least-recently used block to memory if all the ways are valid and an eviction is required. It also fills the evicted or invalid block with the data returned from the memory request. Neither of these tasks depends on state variables and can thus both be accomplished in a single cycle.

While it is possible to have the `miss` state update the LRU bits and return the requested data to the CPU before returning to `idle`, this would mean essentially copying the implementation of the hit state into the miss state, which makes the code clunkier, less elegant, and less modularized. Such an approach was not taken, so a miss requires 3 clock cycles rather than 2.

IV. Testing

A sample test bench file has been provided for each cache (`d_cache_tb.v` and `i_cache_tb.v`, respectively). Each includes three simulations of CPU requests which involve reading, writing, and a combination of both.

The default behavior of the test bench for the data cache (`d_cache_tb.v`) performs repeated reads and writes, evicts dirty data to memory, and brings it back to verify that the data has been written correctly. It also generates requests for offsets within a block to ensure that the correct word is returned. The expected behavior is as follows (since all the addresses have the same index, references to it have been omitted):

1) `0x00000008` is a miss. It is brought into way 1 of the cache and is returned after 3 cycles.
2) `0x00000008` is a hit in way 1, word 1 of the block and should be written to the following cycle.

3) `0x0000000B` is a hit in way 1, word 1 of the block and should be written to the following cycle.
4) `0x0000000C` is a hit in way 1, word 2 of the block and should be written to the following cycle.
5) `0x10000008` is a miss. It is brought into way 2 of the cache and is returned after 3 cycles.
6) `0x20000008` is a miss. It is brought into way 3 of the cache and is returned after 3 cycles.
7) `0x30000008` is a miss. It is brought into way 4 of the cache and is returned after 3 cycles.
8) `0x40000008` is a miss. It is brought into way 1 of the cache, evicting the existing block there, and is returned after 3 cycles.
9) `0x00000008` is a miss (it has just been evicted). It is brought into way 2 of the cache, evicting the existing block there, and is returned after 3 cycles.
10) `0x10000008` is a miss. It is brought into way 3 of the cache, evicting the existing block there, and is written to after 3 cycles.

Note that instructions 3 and 4 demonstrate the correct byte addressing within a block, instructions 8 and 9 demonstrate proper write-back behavior, and instruction 10 demonstrates that the cache is indeed write-allocate. The rest of the module represents basic functionality: repeated reading and writing to the cache, and reading and writing to memory.

The default behavior of the test bench for the instruction cache (`i_cache_tb.v`) performs repeated reads, evicting and replacing with an LRU replacement policy. The expected output is (again, references to the index have been omitted):

1) `0x00000004` is a miss. It is brought into way 1 of the cache and is returned after 3 cycles.
2) `0x10000008` is a miss. It is brought into way 2 of the cache and is returned after 3 cycles.
3) `0x00000005` is a hit in way 1 and should be returned the following cycle.
4) `0x10000006` is a hit in way 2 and should be returned the following cycle.
5) `0x00000007` is a hit in way 1 and should be returned the following cycle.

Note that instructions 3 and 5 demonstrate that data is not written to the cache unless the CPU provides a write enable signal. The rest of the module represents basic functionality: repeated reading from the cache and memory.

Other sample CPU requests are included in the test bench files, but have been commented out. For the data cache, these other groups of requests perform a subset of the tests in the default requests, and for the instruction cache, the commented tests perform non-instruction cache functionality (writing). The other sets of requests should generate behavior described by the comments in the test bench files.

V. Challenges

At the start, the greatest difficulty was parsing the terminology and developing and understanding for the specification. Once it became clear what the hardware "looked like," the implementation was a relatively straightforward programming assignment.

The biggest challenge encountered was managing an eviction policy, but an algorithm was eventually designed for doing so. To update the LRU bits on a hit, each way's LRU bits are compared against the (not yet updated) LRU bits of the way that caused the hit. If the LRU value of a given way is less than the value of the LRU value in the way that caused the hit, then the LRU value of that way is incremented by one. Finally, the LRU value of the way that caused the hit is reset to 0. This ensures proper incrementation both as the cache set fills up and proper maintenance as blocks are read from, written to, and replaced.

An attempt was made (with partial success) to implement a "hardware" solution closer to an actual design for synthesis where hits were determined by a pure combinatorial comparison of the tags in a set, and the results were reported through the use of a multiplexor and a decoder. However, it quickly became clear that the design decision to store cache information across multiple registers rendered this process useless for write hits (conditional statements would still need to be used to write to the correct way). The solution was ultimately abandoned in favor of the working simulation, but the relic files (implementing a basic decoder and multiplexor) can be found in the `hardware` directory.

Another challenge faced was reducing the miss penalty. Initially, the design of the state machine had included two more states: one for requesting data from memory and evicting the LRU block and another to wait for the memory to respond (replacement was performed in what is now the `miss` state). This latency was reduced by requesting from memory in the `idle` state (enabling the request only on a miss) and combining the jobs of reading from and writing to memory into a single state (which was made possible because of the non-blocking assignment in Verilog).