A Parametrizable CPU Cache in System Verilog
with Approximate LRU Page Replacement Policy

Matthew Whiteside & Jared Rue

November 11, 2015

1 Overview

We've implemented a CPU cache with the following features:

- Parametrizable:
  - word size (1 byte by default)
  - page size (a.k.a. words per page)
  - number of rows (a.k.a. pages; 16 by default)
  - number of sets (a.k.a. associativity)
  - address bus width (16 bits by default)
- A pseudo-LRU page replacement which implements the 'clock' algorithm (detailed in section 2 below)
- Some (unrealistic) **simplifying assumptions** of the design:
  - the bus to main memory is the width of a full page (8 bytes/64 bits by default), which means an entire cache line (page) is filled in one cycle when the fetch returns from DRAM
  - Changing operations while one is in progress is not really handled robustly. In other words if you change from a read to a write operation before the read is finished, the behavior is undefined. This means you must hold the current operation until you receive the hit signal

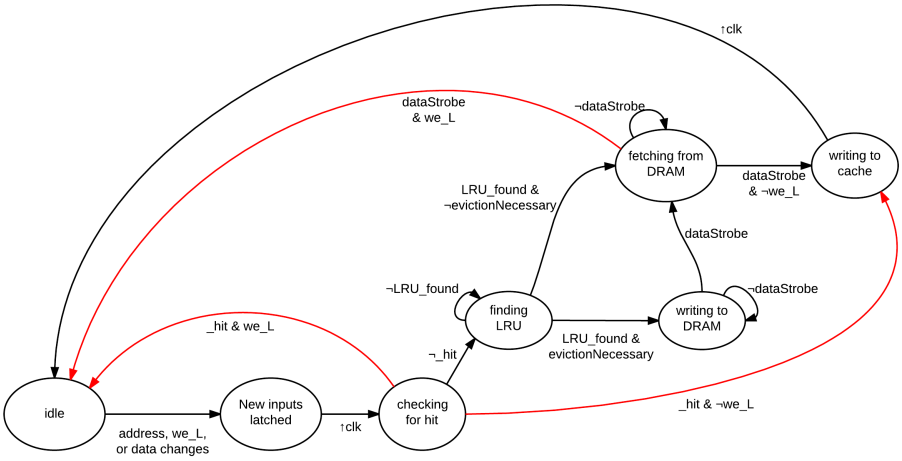
2 Page Replacement Policy

As stated above we've implemented the clock algorithm. It works in the following way. Each set has 3 extra bits per row, a valid bit, a used bit, and a dirty bit. At any time, the LRU_set_ptr is pointing at one of the sets. If the cache receives a request, the first thing checked is whether the address results in a

hit. For a cache hit there's no need to replace anything. Otherwise (on a cache miss), if the used bit is set in the LRU_set, we first clear it, then increment the LRU_set_ptr. During the next cycle, we check the used bit in the new LRU_set. If it's used, we clear it and increment the LRU_set_ptr again. This process continues until we've either found a set whose used bit is clear, or arrived back at the original set, whose used bit we cleared earlier, and is now available.

This reason we chose this algorithm is that it is relatively simple, and performs well compared to a true LRU algorithm (see page 12 of [1]).

3 State Transition Diagram



The 'hit' output signal is high on the states pointed to by the red arcs, and remains high until the inputs change again.

4 Test Cases

We tested the following 6 scenarios:

1. Read A_1/write A_2/read A_1: and expect that the second read to A_1 results in a cache hit, and does not result in a latency of many cycles. Also see attached timing diagram for this scenario.

2. Successive reads that map to the same row index should go to different sets. See line 147 and after in CacheTestbench.sv.

3. sequential writes should go to the same set & cache line, and not cause cache misses

4. Reads to occupied sets/lines should result in evictions. See line 233 and after in CacheTestbench.sv.

5. When the machine enters the FINDING_LRU state, and all the sets in the specified row are full, it should transition to WRITING_TO_DRAM between 2 and 4 cycles later (see line 40 of CacheTestbench.sv).

6. Data should be available on the outputs 3 cycles after the hit signal is received (line 264 of CacheTestbench.sv).

5 Known Issues

There probably needs to be a 'write recovery' state, to allow the dataStrobe signal to fall after a write, before proceeding to a fetch. Ran out of time to implement this.

6 References

1. Arpaci-Dusseau, 2014. Three Easy Pieces