Lab 9: Building a direct-mapped cache in Logisim

Introduction

In this lab, you will build an 8-byte, write-through, direct-mapped cache in Logisim. There will be four cache lines; each line will hold two bytes. Your cache will have a 64-byte RAM as its backing store.

Your cache will ultimately have three inputs and one output. The inputs will be an 8-bit address, a one-bit write enable control, and an 8-bit write data field. The output will be an 8-bit read data field.

By the end of this lab, you will appreciate the complexity of even the simplest type of cache. However, if you are careful, it certainly can be built in a reasonable amount of time.Label the parts of your cache as you go along.

Part A: Creating storage and parsing the address

Given there are 4 cache lines, how many bits is the cache index? ____

Given each word is 1 byte and each cache line contains 2 bytes, how many bits is the cache offset? ___

Given addresses are 8 bits, how many bits is the cache tag field? ____

On the left of the canvas, create an 8-bit address input. Use splitters (under Wiring) to split the address into the tag, index, and offset fields. Recall that the offset should be the low order bit, the index should be next, and the tag should be the remainder. (Why?)

For each cache line, create a valid register, a tag register, and a data register. Label the registers and set the number of data bits appropriately. Line them up in the middle of the Logisim canvas (with plenty of space between them) and wire them all up to a clock input.

Part B: Handling cache hits

On the right of your canvas, create an 16-bit data output.

To output the correct data, your circuit will need to accomplish the following:

1. Use a MUX to select the tag for the correct cache line.
2. Test whether the tag stored in the cache matches the tag field of the address input. (Note that an XNOR gate tests if two bits are equal; after testing each pair of bits, we can test if *all* the bits are equal using an AND gate.) If the tags are equal, we have a hit!
3. If we have a cache hit, then select and output the data field of the correct cache line. (Use another MUX; wire the output of your tag equality tester to the *enable* control of the MUX so that it outputs a value only if the tags match.)

Poke some data into cache tag and data registers, then into the read address, to test your cache. Verify that it reads from the cache line with the correct index. Verify that it produces no output (XXXX) if the tag field of the address does not match the tag stored in the cache.

There's just one problem remaining: We haven't yet used the offset to select one byte from the cache line. How can we do that?

4. Use a splitter to divide the 16-bit output into two 8-bit bytes.
5. Use a MUX to select one of the two bytes, based on the offset field of the address input.
6. Connect an 8-bit output to display the read data.

Poke at the offset bit of the address input to test your implementation.

Part C: Handling cache misses

To handle cache misses, your cache need to read from RAM.

1. Add a RAM component to your canvas. (I put mine off to the upper right.) Since we won't be storing new values to the RAM in this part, don't connect a clock to the RAM yet.
2. Logically, our RAM has 8-bit addresses and 8-bit data width, storing a total of 256 bytes. However, it's much more convenient if we think of our RAM as storing cache blocks rather than individual bytes. (Why?) Therefore, set the address bit width to 7 and the data bit width to 16.
3. Set the RAM's data interface to "separate load and store ports."
4. Connect the upper 7 bits of the address input (everything but the offset) to the address ("A") input of the RAM.
5. You have already built circuitry to detect when there is a cache hit. You can invert this signal to detect when there is a cache miss. When there is a cache miss, the memory should load (output) the value at the given address. Connect your "cache miss" line to the the load control on the RAM.

In addition to reading from RAM, your cache must write the data value into the correct cache line's data field, and also update that cache line's tag field to match the tag field of the address.

6. Feed the data loaded from RAM into all the cache data registers.
7. Feed the tag field of the address input into all the cache tag registers.
8. Finally, wire up the enable controls for the cache tag and data registers so that each register will store a new value only if there is a cache miss AND the index field of the address input matches the index for that particular cache line. (You will need to add a decoder.)

Finally, go over your circuit and make sure your connections are good. Test that as cache misses occur, data is copied from memory into the cache: Using the Poke tool, set some values in RAM. Set the address so that there will be a cache miss. Tick the clock (CTRL-T) to write to the registers and resolve the cache miss.

Note there is one important problem left to solve: Initially, the data in the cache may not match the data in memory.

To solve this problem, we use the valid bit. If the valid bit for a cache line is 0, we know that cache line hasn't been written yet and therefore is invalid.

9. If a cache line's valid bit is 0, any attempt to access that cache line should result in a cache miss, even if the tag matches.

10. Whenever the tag and data for a cache line are written, the line's valid bit should be set to 1 (use a constant input, under Wiring).

Reset the simulation and test your cache. How many cycles is the miss penalty?

Part D: Handling writes

Now, you will extend your cache to handle writes. Your cache will be a write-through, write-allocate cache. "Write-allocate" means that we will handle cache misses by loading blocks from RAM before we write the new data. "Write-through" means we will write the new data to both the cache and the RAM.

1. Add two inputs to your canvas: Write data (8 bits) and write enable (1 bit).
2. Add a clock input to the RAM so that the RAM can store new values.
3. Add circuitry (some splitters and multiplexers) to combine the new write data with the existing block in the cache data register. If the offset is 0, then the write data should be placed in the lower 8 bits. If the offset is 1, then the write data should be placed in the upper 8 bits. The other 8 bits should be filled in from the old cache data. The new 16-bit block should go into the data input of the RAM.
4. Set the store control on the RAM so that the new block value is written to RAM only if there was a cache hit AND the write enable bit is set.
5. Since our cache is a write-through cache, we need to write the new data block to the cache as well as to RAM. Create combinational logic to implement the following behavior:
    - On a cache hit with the write enable bit set, store the new data block (the same as is written to RAM).
    - On a cache miss, store the data data block that was loaded from RAM (as before).
    - Otherwise, do not change the stored data value.

Test your cache and ensure it can write values correctly. (I found this was the trickiest part!)

If you are hungry for more, see these suggestions for implementing a set associative cache.

Janet Davis (davisjan@cs.grinnell.edu)

Created November 24, 2013
Last revised December 3, 2013

Acknowledgment: This lab is inspired a lab for ECS 154A at UC Davis.