N-Set Cache/RAM System with LRU Replacement Strategy Report

Brennan Borchert, Luke Saleh, Quinlan Stewart, Nikodem Gazda

GITHUB WITH ALL CODE AND DOCUMENTATION:

NikodemGazda/Comp-Arch-Final-Proj

VIDEO LINK: (1) N-set Cache/RAM System with LRU Replacement Strategy - YouTube

Original Proposal:

We will be implementing a cache/ram system using HDLs, mainly SystemVerilog and VHDL. In this system, we will have a RAM (of a parameterized size) and an N-set associative cache, configurable at compile time, with the replacement strategy being last recently used. In this design, we will compartmentalize each function by separating each responsibility into separate modules; storage for cache data, cache tags, and cache valid bits will all be separate. The LRU buffers and logic will be contained in one module, but will be separate from the hit/miss logic. These modules will make up the cache block, which will be separate from the RAM.

To ensure the functionality of the design, we will create directed and randomized testbenches for each module and then the entire design as a whole. So that our testbenches are powerful and so that we're confident in the design, we will use constrained random verification, not only to focus on edge cases, but also to guide the verification process, for example making sure we read from the same address after we write. We will also use assertions, which will constantly check that the outputs are correct.

The workload was split among the four group members as the following: Quinlan working on verification per module, Nikodem working on verification for the top-level, Brennan working on design of memories/storage (tag, valid_bat, cache data, RAM blocks), and Luke working on design of LRU buffers and logic for hits/misses. This workload was split across four stages throughout the remainder of the semester: planning/diagram stage, programming design and verification stage, testing and debugging stage, documentation and presentation stage.

Recap of Progress Report:

As of the time of the project progress report, the majority of the design and verification had been completed. Each individual block had been designed and tested, leaving only the top-level integration and testbench remaining. The completed designs include the cache data, cache tag, cache valid, cache hit, cache control, and cache LRU modules. Additionally, verification was completed for each of these modules, ensuring the success of the modules.

The cache has various functionalities that were finished at the point of the progress report due to the completion of the six aforementioned functional blocks. Both the total size and number of banks within the cache may be configured, but we only utilized a 4-set cache with a total size of 16 bytes in our testing. The cache has four operations: write hit, write miss, read hit, or a read miss. The miss operations take 3 cycles to complete, and the read operations take 2 cycles to complete. The cache interfaces with the RAM, which was also finished at the point of the progress report. The RAM is configured using only the RAM size, which we set to 256 bytes in our testing. RAM reads and writes only take 1 cycle each, with reads occurring when write enable is low. The RAM takes in an address, data, and a write enable while outputting data as well.

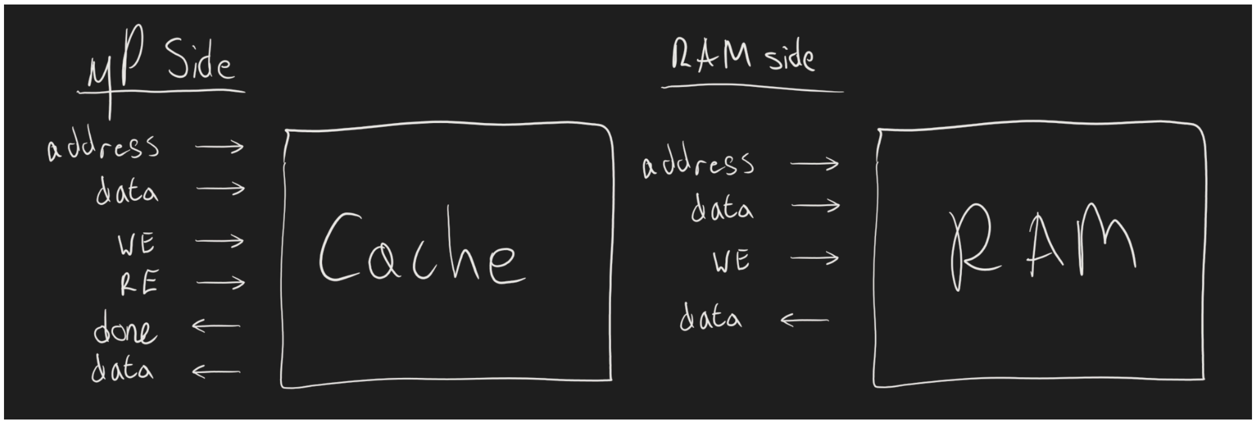

Figure 1: Block Diagram of Cache and Ram Interface.

Within the cache are six storage blocks, which can be viewed as split between storage and logic. The three storage blocks are the cache data, cache tag, and cache valid blocks. Cache data stores whatever is being stored in the cache, cache tag stores the tags for each byte of data stored in the cache, and cache valid stores the valid bit for each byte of data. On the other hand, the logic blocks include the cache hit, cache control, and cache LRU blocks. The cache hit block determines if the requested read/write is present in the cache (checking if the operation is a hit or miss) and outputs the bank relevant to the index of the current operation. The cache control block handles all miscellaneous logic in the cache - everything from the done signal to the write enables to the data muxing logic. The cache LRU block stores pointers to least recently used banks for each index and outputs the least recently used one.

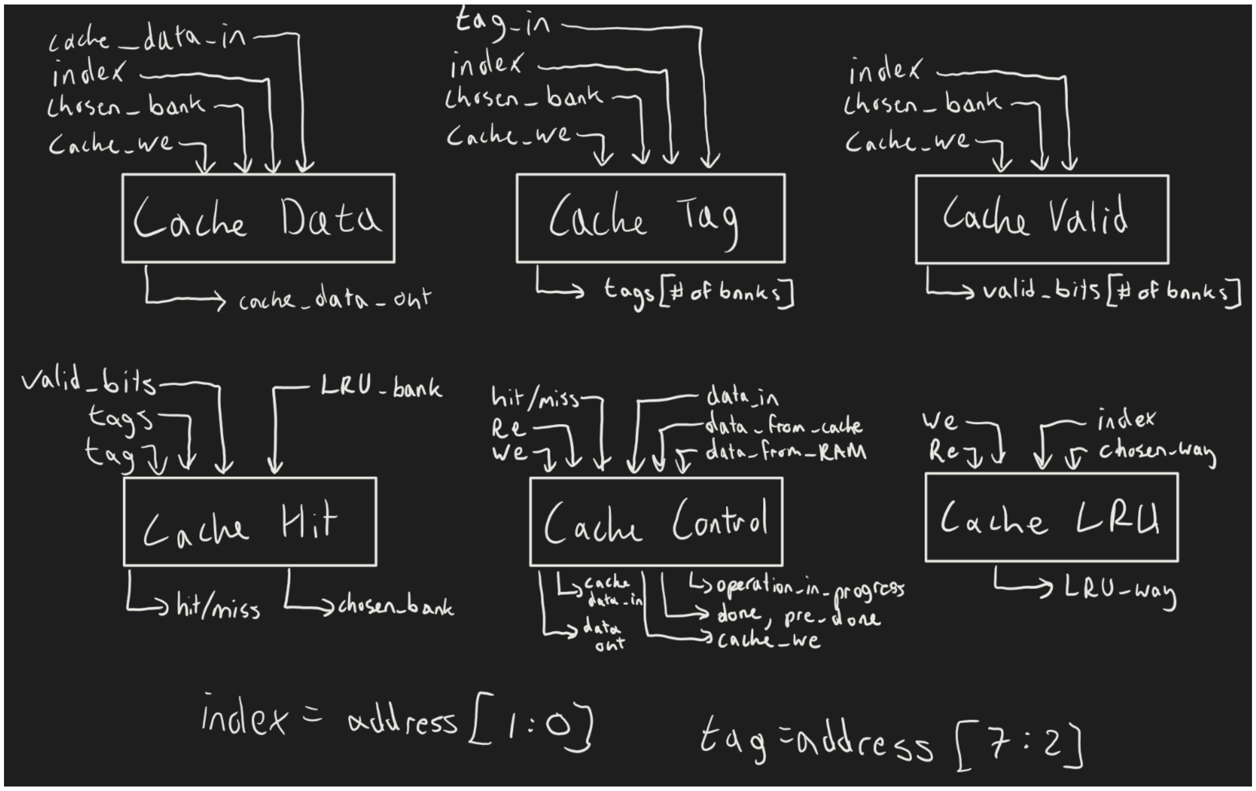

Figure 2: Cache Block Diagram

Several pieces of verification code were completed at the point of the progress report as well, including both directed tests and constrained random verification. There are six unique directed test cases, each repeated five times. Some of these tests included all inputs being 0s, all inputs being 1s, and all tags being 0,1,2,3 with a target tag of 1 and valid bits are 1 with LRU way being 3. The constrained-random verification was tested with the following constraints:

- Target tag, least recently used bank from the LRU, and the cache tag are all randomized with a 10% chance for all 0s, 10% chance for all 1s, and an 80% chance for all other combinations
- Cache tags (tags for each bank of the current index) must never be the same
- Target tag matches one of the cache tags 50% of the time to make sure we test for hits and misses equally

Each test had its own set of assertions and additional reference models/logic for testing. For instance, the cache hit block had the following assertions:

- Miss_check: if a miss is reported, the chosen bank must match the LRu bank

- Hit_check: if a hit is reported, the tag corresponding to the chosen bank must match the target tag.
- Tags_match: if any of the tags match the target tag, there should be a hit reported
- No_tags_match: when no tags match the target tag, we expect a miss

Results/Deliverable:

Since the progress report, the main result was the final deliverable in the form of the top level entity and its testbench. The top level entity provides the communication between the RAM and the cache. The testbench, like the verification for the cache submodules, contains both directed and random test cases. In the first directed test case, we test for a write miss and a read miss by writing to every RAM address first, then reading from every address afterwards. Since each address access is separated by the reads and writes of all the other addresses, each operation will be a miss. This test case ensures that data being written is maintained over longer periods of time, even as other indices and banks are written to and read from.

In the second directed test case, we test for write hit and read hit operations by writing to every RAM address twice and reading from the same address immediately afterwards, which is then repeated for each address. Since each address access is sequential, the second write and first read for each operation will be a hit. This test case ensures that data is written and read in the expected delay time. In a third directed test case, an example use is detailed by writing one character of a message to each address of the RAM. Once each character is written, the data is read back from each address, stored in a string in the testbench, and displayed as a returned message to compare against the original. Random verification, as previously stated, was also performed by randomizing the write enable, read enable, address, and data input. All of these test cases together show how the cache communicates properly with the RAM and performs its various operations correctly.

Summary:

This project was both challenging and interesting in the context of computer architecture. One of the main takeaways from the work was the importance of a clear functionality specification, which can help prevent redesigns and provide a smooth path towards functional code. We learned through various trials and errors that it is much better to spend more time planning a design than to continuously redesign it due to errors in a rushed plan. In the future, this project could be further improved with pipelining. This would not only improve the efficiency/throughput of the system, but could also simplify some of the logic to ensure operations do not overlap. Additionally, reconstructing some of the design to be a finite state machine rather than a set of logic blocks could streamline the process.

Through this project, we gained experience in design, verification, and utilization of SystemVerilog. This was a good practice of both understanding caches and hardware design/verification. By undertaking such a large project, we utilized a far greater number of SystemVerilog concepts to implement the algorithms that make caches work and streamline the design process.