Cache Controller

By Sorin Beșleaga and Yosif Al-Yafeai

Git Repository: https://github.com/sorykkk/Cache-Model

Project Description

3.1 Cache Controller Specifications

The cache controller to be designed must adhere to the following specifications:

- Cache Type: 4-way set associative
- Cache Size: 32 KB
- Block Size: 64 bytes
- Number of Sets: 128 sets
- Word size: 4 B
- Words per block: 16 words/block
- Associativity Level: 4-way
- Replacement Policy: Least Recently Used (LRU)
- Write Policy: Write back with write allocate
- Endiannes: Little-Endian (assumed one)

3.2 FSM-Based Design

FSM states:

- IDLE (3'b000): Waiting for a new request.
- READ HIT (3'b001): Accessing data for a read request found in the cache.
- READ MISS (3'b010): Handling read requests when data is not in the cache.
- WRITE HIT (3'b011): Managing write operations when data is found in the cache.
- WRITE MISS (3'b100): Processing write operations when data is not found in the cache.
- EVICT (3'b101): Evicting data from the cache according to the LRU policy.

Each transition is based in rd_en, wr_en inputs and hit, rd_m, wr_m internal state registers.

3.3 Hardware Description Language

HDL used:

- System Verilog 2012

3.4 Simulation and Testing Tools

For simulation and testing with waveforms we used :

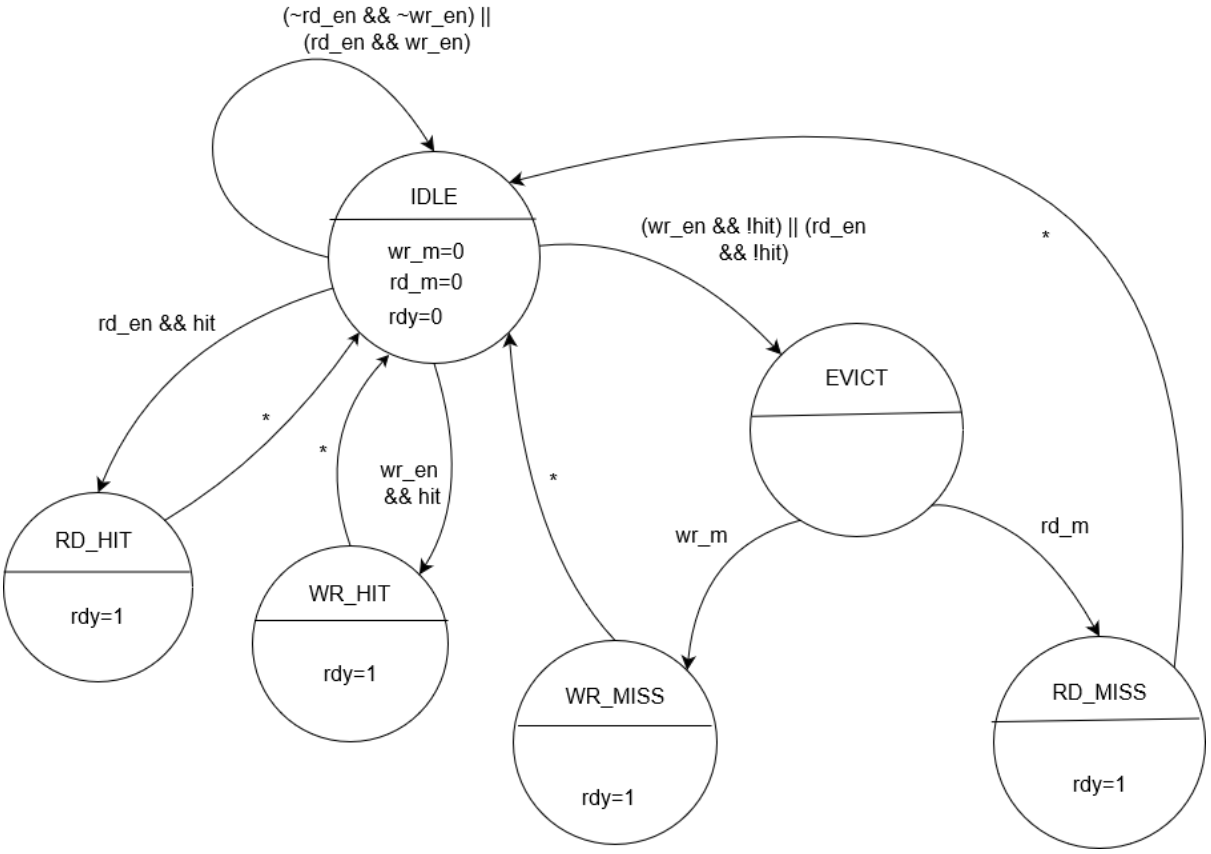
- Icarus Verilog

Compile : **`iverilog -g2012 -o cache_waves.vvp cache_tb.sv`**
Generate waves: **`vvp cache_waves.vvp`**
Inspect waves: **`gtkwave`** (then open wave file into the tool)

Deliverables

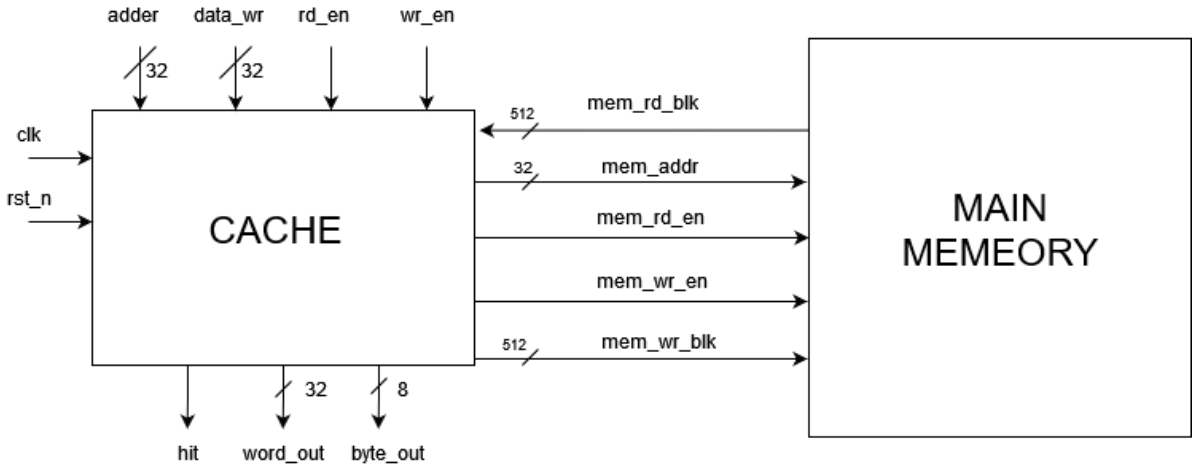
4.1 Detailed Design Specification

- Detailed FSM diagram of our cache controller:



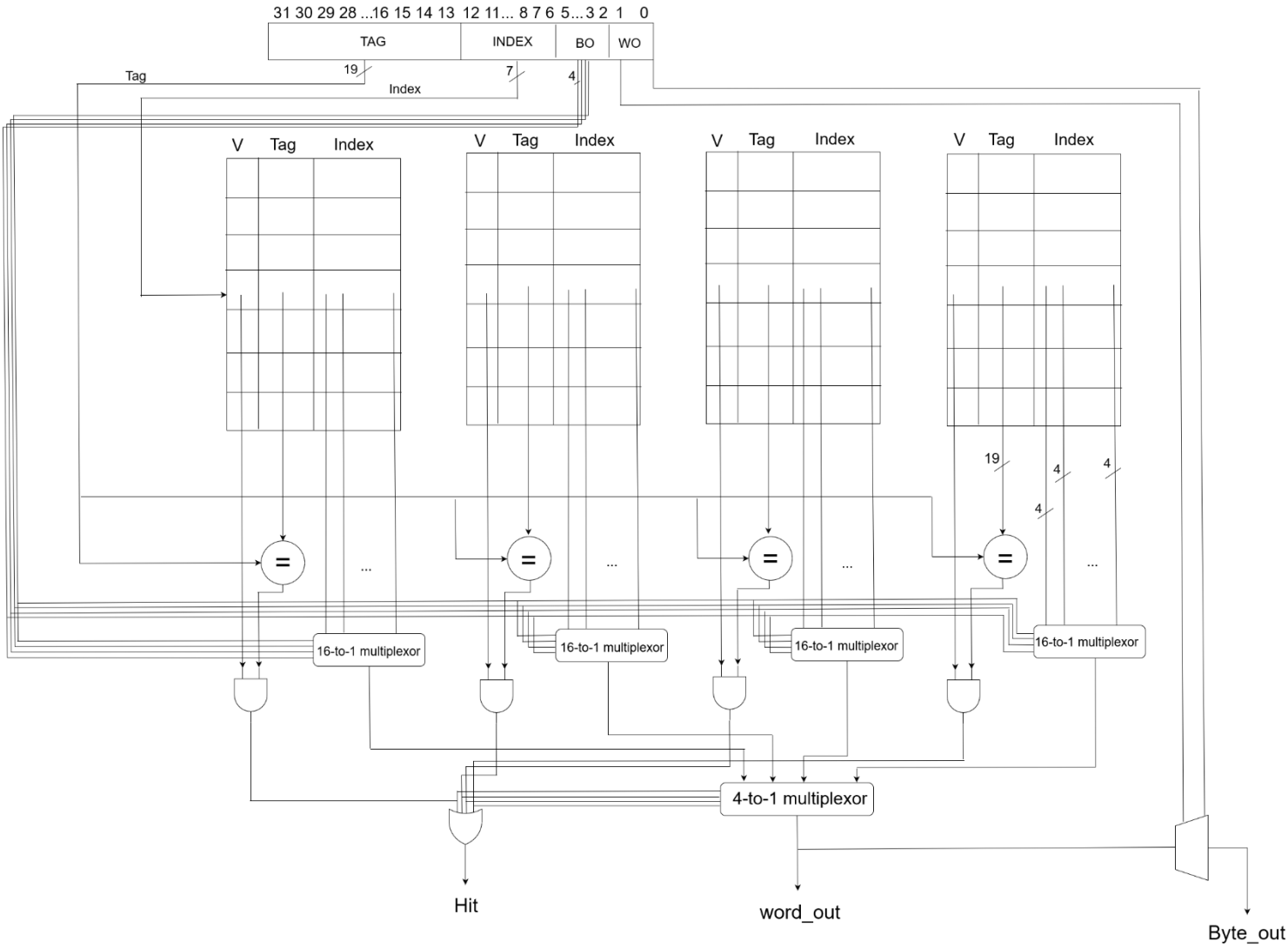
This controller can be driven only by rd_en, wr_en and of course the addr – address what is needed for CPU. The rest of the pins are used for communication with main memory or to output information from cache.

- Functional block diagram of cache controller :



Here we can see the interface wires that communicate with main memory that is also implemented and that uses SEED to autopopulate itself with pseudo-random numbers.

- Complete detailed scheme for cache implementation :



Here we have put distinctive output pins for byte and word, because we can't see the reasoning behind wanting only a byte from the extracted 4B word, so we made both to be outputed.

- HDL motivation : We know this language a little better and all the Verilog prior to System Verilog because it offers more flexibility with pins defined as regs or logic.

4.2 Implementation

- https://github.com/sorykkk/Cache-Model

The project is hosted on a GitHub repository, having following structure:

- `docs` folder : contains all necessary documentation, like requirements, deliverables and instructions/commands how to run system verilog code for simulation and waveforms view.
- `src` folder : contains all implemented files, like
  - `cache_data.sv` – file that implements cache controller
  - `mem.sv` – file that implements memory as an array of bytes that extracts bytes from block and that can compose blocks from it's bytes given the address
  - `macros.sv` – file that stores all the constants and slices needed througout all the implementation, and is easier to modify one value than the same value all over the code
  - `cache_tb.sv` – file that test the implemented module, and that covers all the possible states and outcomes for the DUT.

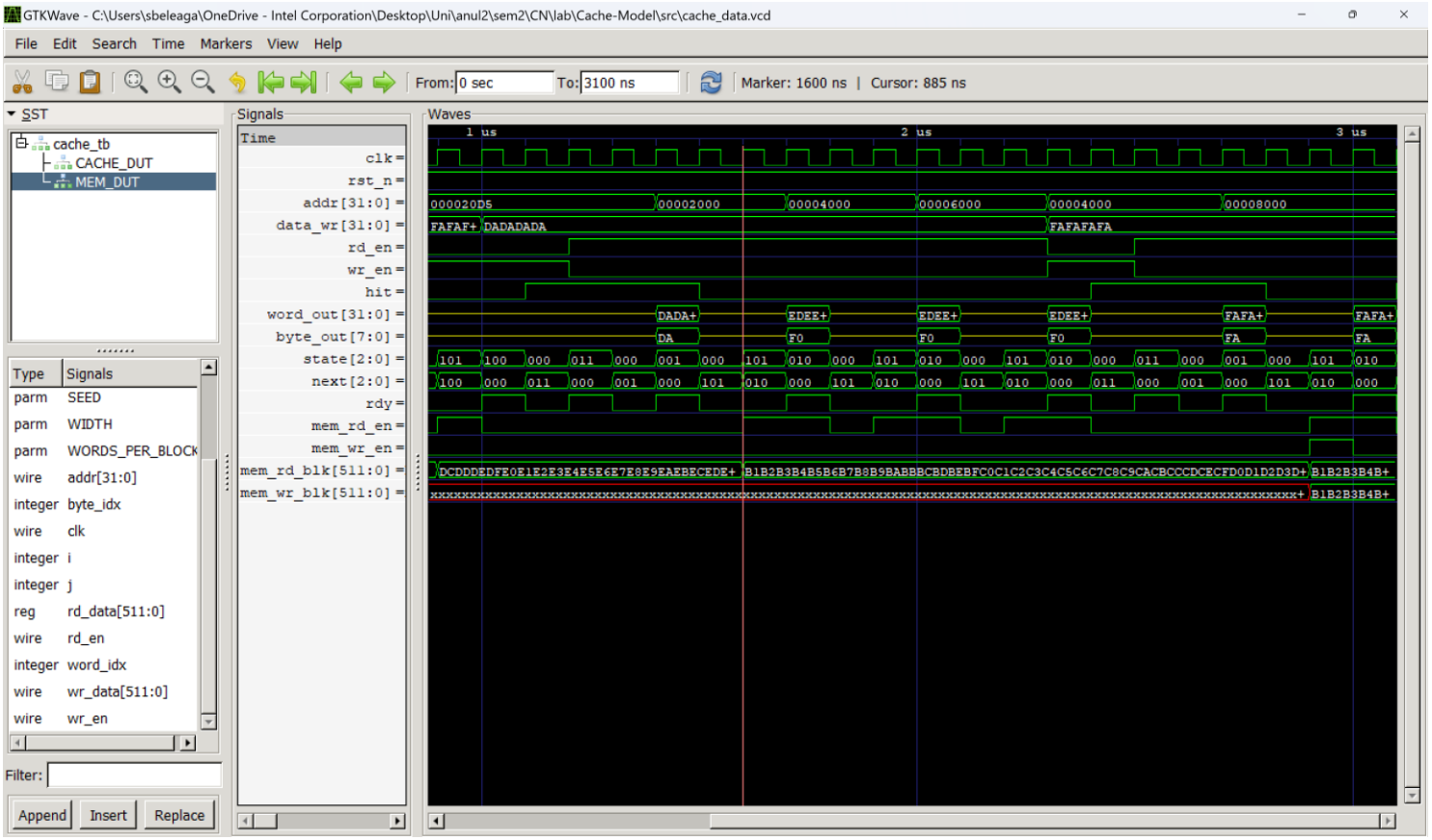
4.3 Test Bench and simulation result

We have covered all possible test cases, like: read hit, write hit, read miss, write miss, write/read miss + replacement, write hit + lru increment , etc...

In the following screenshot you can see last testbench results for the DUT, all the interesting signals are listed on the left, near the wave monitor, all the signal and modules under test are on the left.

Metrics:

- Access time : 1 c.c.
- Miss Penalty : 3 c. c. (1 access clock + 2 clocks for evicting and replacing)



4.4 Final Report

We implemented all the DUT necessary for testing the cache controller. In mem module we made a kind of memory, that is represented by an array of bytes that is populated with SEED parameter and generates 512 blocks, that are accessed by cache when there is miss.

Memory Communication

The cache module can communicate with memory through : mem_addr (send address for data retrieval), mem_rd_en (signal to read from memory), mem_wr_en (signal to write to memory), mem_wr_blk (the block that is needed to be written in the memory after write-back dirty blocks), mem_rd_blk (the block that is send to the cache when mem_rd_en is active and there is a miss in the cache). When both mem_wr_en and mem_rd_en are active, the mem_rd_blk will get immediately the value of mem_wr_blk to send it back the the cache.

Module Functionality

The `cache_data` module interfaces with the CPU and main memory to perform read and write operations, ensuring data is stored and retrieved efficiently. The primary operations include:

- **Read Operation (rd_en)**: Checks if the requested data is present in the cache (hit) and retrieves it. If the data is not present (miss), it initiates a memory read to fetch the block from main memory.
- **Write Operation (wr_en)**: Writes data to the cache if the block is present (hit). On a miss, the module evicts a block if necessary and writes the new data to the appropriate set.

Key Components and Registers

The cache structure comprises several critical components:

- **Valid Bits (valid)**: Indicates whether a cache block contains valid data.
- **Dirty Bits (dirty)**: Indicates whether a cache block has been modified and needs to be written back to main memory before being evicted.
- **Least Recently Used (LRU) Registers (lru)**: Maintains the order of block usage to determine the least recently used block for eviction.
- **Tags (tag)**: Stores the tag portion of the address for cache lookup.
- **Data Blocks (data)**: Stores the actual data blocks.

LRU Management

The LRU mechanism is critical for managing cache block replacement. It ensures that the least recently used block is identified and evicted when necessary. The LRU register values are updated as follows:

- On a cache hit, the LRU counter for the accessed block is reset to zero, and other LRU counters are incremented.
- On a cache miss and subsequent block replacement, the LRU counters are adjusted accordingly to reflect the new access pattern.

Initialization and Simulation

The cache is initialized to ensure all valid, dirty, and LRU registers are set to zero. This ensures that the cache starts in a clean state without any residual data from previous operations.

Conclusion

The 4-way set associative cache design detailed in this report aims to optimize memory access times by efficiently managing data retrieval and storage through its associative structure, LRU replacement policy, and robust state machine. The implementation in SystemVerilog provides a comprehensive solution to address the needs of high-speed data access in modern computing systems.

Our technical challenges encountered were memory implementation that needed to be correctly implemented to test the cache controller, the next challange was to select the corresponding test data to test all possible scenarios, this were kind of unexpectedly hard.