ECSE 6700 – Computer Hardware Design

Lab 1 Report –Cache Coherence

Paul Nieves (RIN: 61999083)

Professor Liu Liu, Linsen Ma

February 26 2025

Table of Contents

Module Organization/Design Overview

The design implements a multi-core cache coherence system using the MESI protocol. It consists of five primary submodules: the RAM module, cache controller, set-associative cache, cache wrapper, and arbiter. Each processor core interfaces with its own cache module, which communicates with shared memory (RAM) and other caches via a common bus. The arbiter manages bus access requests to resolve conflicts between cores. The system ensures data consistency across caches through snooping and state transitions (Invalid, Shared, Exclusive, Modified).

Figure 1 illustrates the top-level architecture, where multiple caches connect to a shared bus and RAM. The cache controller orchestrates MESI state transitions, while the arbiter prioritizes bus requests. Data and control signals flow between the caches, RAM, and processors to maintain coherence.

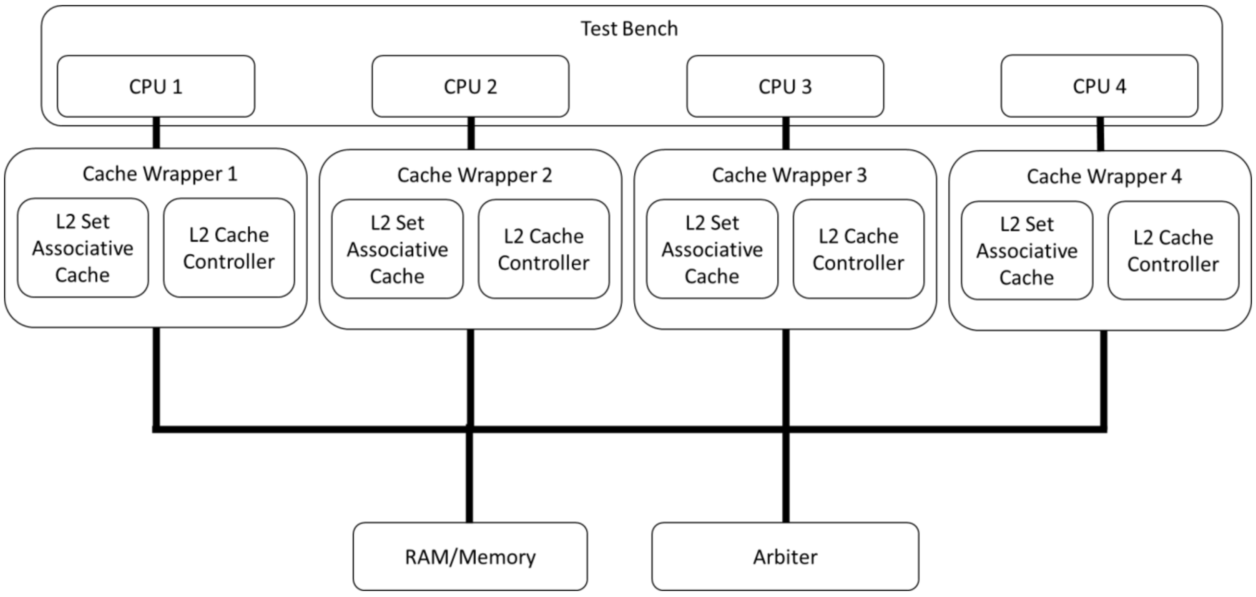


Figure 1: Top Module Cache Coherence Multi Core Diagram Overview

Description of Cache Components and Basic Operations

Top Module

The Top module serves as the primary integration point for the entire cache coherence system. It orchestrates the interaction between four cache instances (referred to as PrivC1 through PrivC4) and connects them to a shared bus, which in turn interfaces with the main memory and arbitration logic. Parameterized by the address width (32 bits) and additional cache-related parameters, the module establishes uniform addressing across the system. It instantiates four CacheWapperL1 modules, each corresponding to a distinct CPU-to-cache interface. These instances handle individual read and write requests from separate CPU cores. In addition to relaying these operations, the Top module combines and routes shared signals—such as bus requests, snoop signals, and invalidation acknowledgments—from all caches. It further instantiates the Arbiter module to resolve bus access conflicts and the RAM module to serve as the primary memory, ensuring that data coherency is maintained system-wide.

Direct Mapped L1 Cache Implementation Consideration

It is important to note that a direct mapped L1 cache has not been implemented in this design. Instead, the current design employs a set-associative cache for L2, which offers multiple ways per set and more flexible replacement options. However, implementing a direct mapped L1 cache would be straightforward. Essentially, you could modify the existing setAssociativiteCache module by reducing the associativity to one, thereby eliminating the need for complex pseudo-LRU replacement logic. Once this module is adapted to be direct mapped, it could be connected inside the Cache Wrapper module in series with the set-associative version. The MESI protocol, which is already managing coherence across caches, would continue to handle state transitions and ensure that the correct data is delivered to the CPU, regardless of which cache holds it. This approach confirms that when wanting to add a direct mapped; the design requires minimal modifications while maintaining robust cache coherence.

RAM Module

The RAM module acts as the central memory unit in the system, storing data in 128-bit blocks. It is designed to initialize from an external file if provided, or it can initialize its memory array to zeros based on a configurable parameter. This module features a tri-state buffer that controls when data is driven onto the shared bus, ensuring that the memory output does not interfere with other components unless explicitly permitted. During a write operation, the RAM module captures data from the bus and updates its internal memory array at the address specified by the cache. In read operations, after a brief delay to simulate realistic memory access times, the module outputs the requested data onto the bus. The RAM module also monitors an abort signal, which helps it gracefully handle scenarios where ongoing cache operations necessitate cancellation of a memory access. Overall, its design guarantees that memory operations—both read and write—are conducted reliably in a multi-core environment.

Cache Controller Module

The L2CacheController module implements the heart of the cache coherence logic by managing both the pseudo-LRU replacement mechanism and the MESI state machine. Its design is built on precise bit-level manipulation of addresses, using defined constants to extract index, tag, and block offset values for each cache line. For processor transactions, the module isolates the relevant address segments and processes requests to update the MESI state from Invalid, Shared, or Exclusive to Modified as needed. In doing so, it determines whether a read operation should leave the block in Exclusive or Shared mode based on whether the block is already held in a shared state. For snooping operations—triggered by bus transactions such as BusRd, BusRdX, or Invalidate—the controller similarly adjusts the MESI state of a cache line. The module utilizes a combination of always blocks with case statements to deduce the next MESI state for both processor and snoop paths.

Additionally, the controller incorporates a pseudo-LRU algorithm for cache replacement. It maintains an internal LRU structure for each set, updating this state based on which cache line (or "way") is accessed. Using combinational logic, the module evaluates the current LRU state and selects the appropriate cache line for replacement on a miss. The logic is carefully structured to update the internal LRU registers based on the accessed block, ensuring that the most appropriate candidate is evicted when needed. By blending the intricacies of both the LRU replacement policy and the MESI state machine, the L1CacheController module plays a critical role in maintaining high-performance and coherent cache operations across the system.

Set Associative Cache Module (L2)

Overview

The setAssociativiteCache module is a key component of a 4-way set-associative cache system that implements coherent cache operations using the MESI protocol. It is parameterized by constants such as SET_SIZE, ASSOCIATIVITY, ADDRESSSIZE, and MESI_SIZE, which define the cache's structure—including the number of sets, the number of ways per set, and the bit-widths for addressing and state fields. This module interacts with both the processor and a shared common bus, ensuring that data is stored, retrieved, and updated in accordance with cache coherence rules.

Interface and External Connectivity

The module's interface is designed to handle requests from the processor and to communicate with other caches and memory via a common bus. It accepts processor read and write commands, along with a 32-bit address that is divided into tag, index, and block offset fields using pre-defined macros. Data exchange occurs on an inout Data_Bus, and a CPU_stall signal is asserted when the processor must wait—typically during a cache miss or when a memory transaction is in progress. In addition to its processor-facing ports, the module drives a set of common bus signals such as Address_Com, Data_Bus_Com, BusRd, BusRdX, Invalidate, and Data_in_Bus, which are crucial for bus arbitration and for handling snoop requests from other processors.

Cache Wrapper Module

The Cache Wrapper module acts as a top-level wrapper for the L1/L2 cache subsystem. It integrates both the set-associative cache memory and the cache controller to create a cohesive unit that manages processor accesses, bus transactions, and coherence protocols. This module receives read and write commands (PrRd and PrWr) along with a full 32-bit address, which it then partitions into its constituent fields (tag, index, and block offset) according to the defined constants. Internally, the module instantiates a dedicated set-associative cache component that leverages a pseudo–Least Recently Used (LRU) replacement policy, ensuring that on a cache miss, the least-used block is replaced efficiently. In parallel, it instantiates the L2CacheController, which orchestrates the MESI protocol state transitions for both processor and snoop requests. Together, these submodules support coherent operations over a shared bus, handling signals such as bus requests, snooping (BusRd, BusRdX, Invalidate), and memory write-back indications. The Cache Wrapper module thereby abstracts the lower-level cache memory operations while interfacing with the wider cache coherence network, ensuring that data remains consistent across multiple cores.

Arbiter Module

The Arbiter module is dedicated to managing access to the shared bus, ensuring orderly communication between multiple cache modules and the memory subsystem. It receives separate request signals from both the processor side (via the Cache Wrapper modules) and the snooping side, where cache coherence transactions are handled. Using a priority-based scheme, the Arbiter assigns bus grants based on the order of the request signals; the lowest-numbered request is given the highest priority, which ensures a deterministic resolution when several caches request bus access simultaneously. This mechanism not only helps in coordinating processor and snoop transactions but also manages the memory snoop grant signal. By quickly resolving contentions, the Arbiter facilitates efficient invalidations and write-back operations, which are crucial for maintaining data consistency across the cache hierarchy.

Test Cases and Simulation Results

Test vector

The coherence system was validated through a series of test cases. Simulation test vectors cover various scenarios, such as a write by Cache 1 that transitions its state from Invalid to Modified, followed by a write by Cache 2 that forces an invalidation of Cache 1’s block. Subsequent read operations illustrate further state transitions: Cache 1 transitions to Exclusive mode after reading data fetched from Cache 2, and Cache 3’s read operation results in a shared state across multiple caches. Waveform diagrams from the simulations confirm that the MESI protocol is correctly implemented, demonstrating proper bus transactions and state transitions that ensure data consistency across the caches.

```
// Format (101-bit binary per line):
// [100:99] cache_sel [98] R [97] W [96:65] address [64:33] data_in [32:1] expected_data [0]
expected_ready
// Test Sequence:
// 1. Write 0xCAFEBABE to Cache1 @ 0x00000004
// 2. Write 0xDEADBEEF to Cache2 @ 0x00000004
// 3. Read from Cache1 @ 0x00000004 (expect 0xDEADBEEF)
// 4. Read from Cache2 @ 0x00000004 (expect 0xDEADBEEF)
// 5. Read from Cache3 @ 0x00000004 (expect 0xDEADBEEF)
// Write to Cache1 (Vector 0)
00_0_1_00000000000000000000000000000100_11001010111111101011101010111110_00000000000
000000000000000000000_1
// Write to Cache2 (Vector 1)
01_0_1_00000000000000000000000000000100_11011110101011011011111011101111_00000000000
000000000000000000000_1
// Read from Cache1 (Vector 1)
00_1_0_00000000000000000000000000000100_00000000000000000000000000000000_11011110101
011011011111011101111_1
// Read from Cache2 (Vector 1)
01_1_0_00000000000000000000000000000100_00000000000000000000000000000000_11011110101
011011011111011101111_1
// Read from Cache3 (Vector 1)
10_1_0_00000000000000000000000000000100_00000000000000000000000000000000_11011110101
011011011111011101111_1
```

Figure 2: Test Vector (CPU_Instructions.tv)

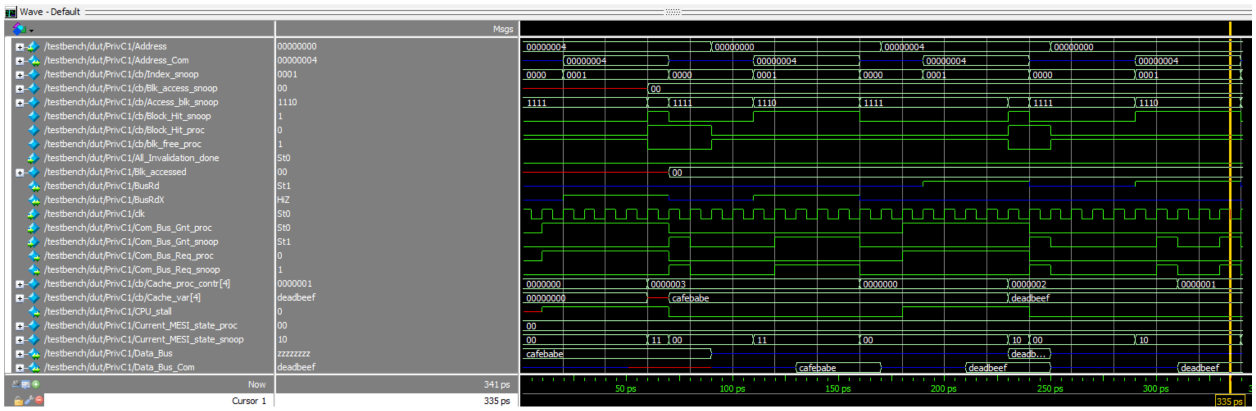
Cache Depth	Cache Memory Initialization (32 bits)	RAM Memory Initialization (128 bits)
0	128’b0000…..0000;	128’b000000000000000…..0000;
1	128’b0000…..0000;	128’b000000000000000…..0000;
….	…..	….
64	128’b0000…..0000;	128’b000000000000000…..0000;

Table 1: Memory Data Initialization

Test Vector	Tag = address [25:0] (in decimal)	MESI Transition Diagram (INVALID = 00, SHARED = 01, EXCLUSIVE =10, MODIFIED = 11) (in binary) (cache_proc_contr[4])	Offset = adress[1:0], Index= address[5:2] (in decimal)	CPU to Cache Data in/out = Data_Bus [31:0] (hex)	Cache1 Memory After Op (cache_var [4])	Cache2 Memory After Op (cache_var [4])	Cache3 Memory After Op (cache_var [4])
0 (Write Op Cache 1)	0	Cache 1: INVALID-> MODIFIED	0,1	CAFEBABE	CAFEBABE 00000000 00000000 00000000	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000
1 (Write Op Cache 2)	0	Cache 1: MODIFIED-> INVALID Cache 2: INVALID-> MODIFIED	0,1	DEADBEEF	CAFEBABE 00000000 00000000 00000000	DEADBEEF 00000000 00000000 00000000	00000000 00000000 00000000 00000000
2 (Read Op Cache 1)	0	Cache 1: INVALID -> EXCLUSIVE	0,1	DEADBEEF	DEADBEEF 00000000 00000000 00000000	DEADBEEF 00000000 00000000 00000000	00000000 00000000 00000000 00000000
3 (Read Op Cache 2)	0	No Transitions (Cache 2 Stays in modified)	0,1	DEADBEEF	DEADBEEF 00000000 00000000 00000000	DEADBEEF 00000000 00000000 00000000	00000000 00000000 00000000 00000000
4 (Read Op Cache 3)	0	Cache 1: EXCLUSIVE -> SHARED Cache 3: INVALID -> SHARED	0,1	DEADBEEF	DEADBEEF 00000000 00000000 00000000	DEADBEEF 00000000 00000000 00000000	DEADBEEF 00000000 00000000 00000000

Table 1: Expected Outputs for Test Vectors

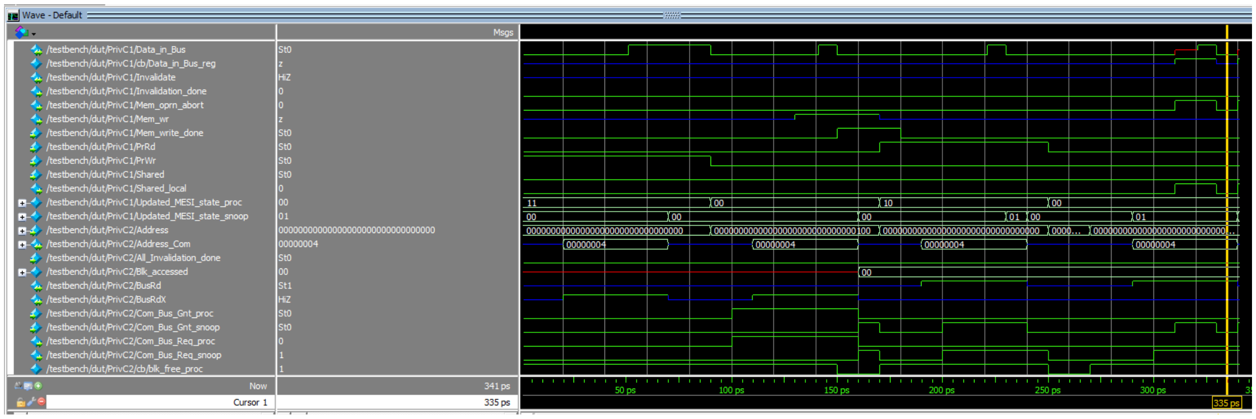
Simulation waveforms



Wave - Default

	Msgs
/testbench/dut/PrivC1/Address	00000000
/testbench/dut/PrivC1/Address_Com	00000004
/testbench/dut/PrivC1/cb/Index_snoop	0001
/testbench/dut/PrivC1/cb/Blk_access_snoop	00
/testbench/dut/PrivC1/cb/Access_blk_snoop	1110
/testbench/dut/PrivC1/cb/Block_Hit_snoop	1
/testbench/dut/PrivC1/cb/Block_Hit_proc	0
/testbench/dut/PrivC1/cb/blk_free_proc	1
/testbench/dut/PrivC1/All_Invalidation_done	St0
/testbench/dut/PrivC1/Blk_accessed	00
/testbench/dut/PrivC1/BusRd	St1
/testbench/dut/PrivC1/BusRdX	HiZ
/testbench/dut/PrivC1/clk	St0
/testbench/dut/PrivC1/Com_Bus_Gnt_proc	St0
/testbench/dut/PrivC1/Com_Bus_Gnt_snoop	St1
/testbench/dut/PrivC1/Com_Bus_Req_proc	0
/testbench/dut/PrivC1/Com_Bus_Req_snoop	1
/testbench/dut/PrivC1/cb/Cache_proc_contr[4]	0000001
/testbench/dut/PrivC1/cb/Cache_var[4]	deadbeef
/testbench/dut/PrivC1/CPU_stall	0
/testbench/dut/PrivC1/Current_MESI_state_proc	00
/testbench/dut/PrivC1/Current_MESI_state_snoop	10
/testbench/dut/PrivC1/Data_Bus	zzzzzzzz
/testbench/dut/PrivC1/Data_Bus_Com	deadbeef
Now	341 ps
Cursor 1	335 ps

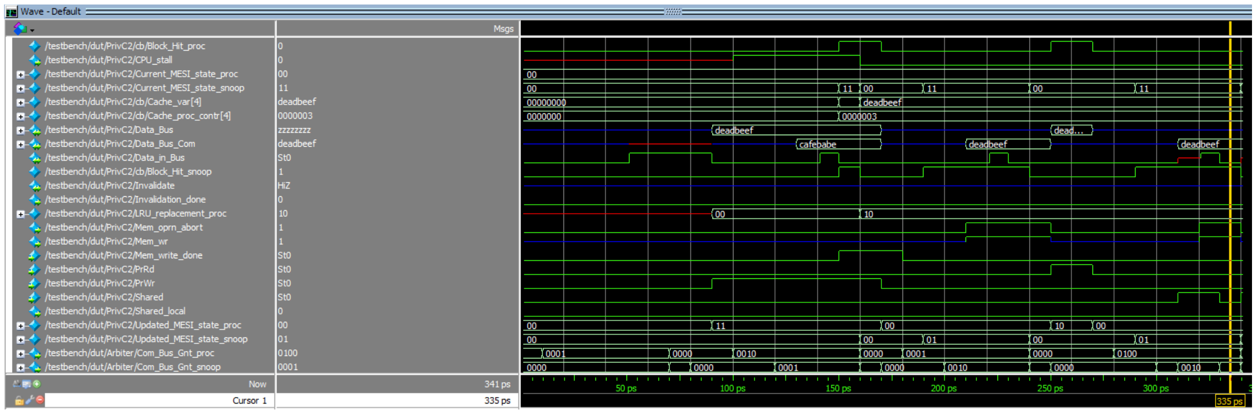
Figure 3: Simulation with Testbench Using Test Vectors Part 1 (CPU_Instructions.tv)



Wave - Default

	Msgs
/testbench/dut/PrivC1/Data_in_Bus	St0
/testbench/dut/PrivC1/cb/Data_in_Bus_reg	z
/testbench/dut/PrivC1/Invalidate	HiZ
/testbench/dut/PrivC1/Invalidation_done	0
/testbench/dut/PrivC1/Mem_oprn_abort	0
/testbench/dut/PrivC1/Mem_wr	z
/testbench/dut/PrivC1/Mem_write_done	St0
/testbench/dut/PrivC1/PrRd	St0
/testbench/dut/PrivC1/PrWr	St0
/testbench/dut/PrivC1/Shared	St0
/testbench/dut/PrivC1/Shared_local	0
/testbench/dut/PrivC1/Updated_MESI_state_proc	00
/testbench/dut/PrivC1/Updated_MESI_state_snoop	01
/testbench/dut/PrivC2/Address	00000000000000000000000000000000
/testbench/dut/PrivC2/Address_Com	00000004
/testbench/dut/PrivC2/All_Invalidation_done	St0
/testbench/dut/PrivC2/Blk_accessed	00
/testbench/dut/PrivC2/BusRd	St1
/testbench/dut/PrivC2/BusRdX	HiZ
/testbench/dut/PrivC2/Com_Bus_Gnt_proc	St0
/testbench/dut/PrivC2/Com_Bus_Gnt_snoop	St0
/testbench/dut/PrivC2/Com_Bus_Req_proc	0
/testbench/dut/PrivC2/Com_Bus_Req_snoop	1
/testbench/dut/PrivC2/cb/blk_free_proc	1
Now	341 ps
Cursor 1	335 ps

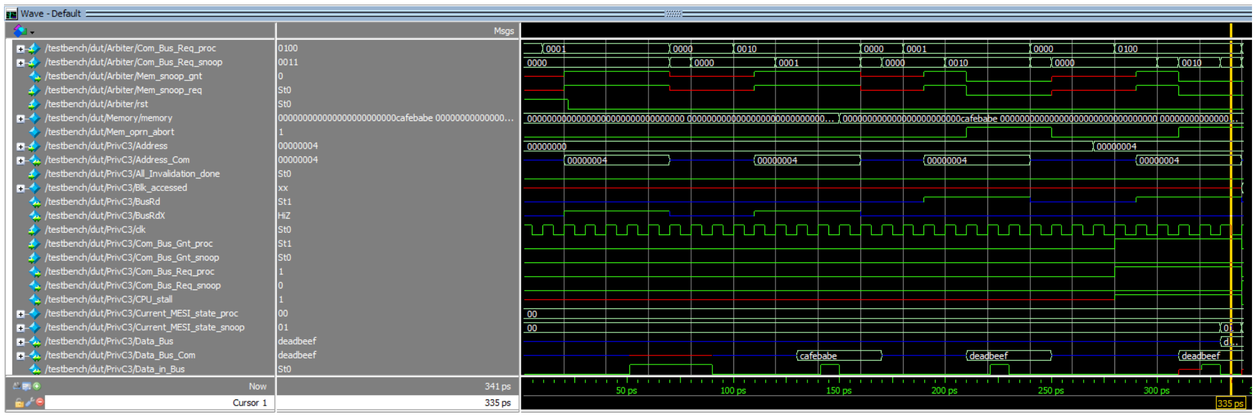
Figure 4: Simulation with Testbench Using Test Vectors Part 2 (CPU_Instructions.tv)



Wave - Default

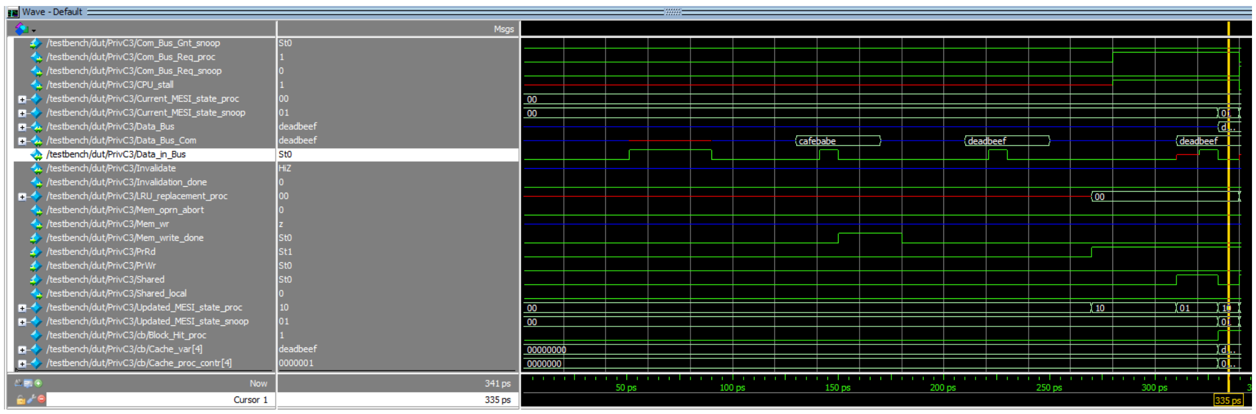
	Msgs
/testbench/dut/PrivC2/cb/Block_Hit_proc	0
/testbench/dut/PrivC2/CPU_stall	0
/testbench/dut/PrivC2/Current_MESI_state_proc	00
/testbench/dut/PrivC2/Current_MESI_state_snoop	11
/testbench/dut/PrivC2/cb/Cache_var[4]	deadbeef
/testbench/dut/PrivC2/cb/Cache_proc_contr[4]	0000003
/testbench/dut/PrivC2/Data_Bus	zzzzzzzz
/testbench/dut/PrivC2/Data_Bus_Com	deadbeef
/testbench/dut/PrivC2/Data_in_Bus	St0
/testbench/dut/PrivC2/cb/Block_Hit_snoop	1
/testbench/dut/PrivC2/Invalidate	HiZ
/testbench/dut/PrivC2/Invalidation_done	0
/testbench/dut/PrivC2/LRU_replacement_proc	10
/testbench/dut/PrivC2/Mem_oprn_abort	1
/testbench/dut/PrivC2/Mem_wr	1
/testbench/dut/PrivC2/Mem_write_done	St0
/testbench/dut/PrivC2/PrRd	St0
/testbench/dut/PrivC2/PrWr	St0
/testbench/dut/PrivC2/Shared	St0
/testbench/dut/PrivC2/Shared_local	0
/testbench/dut/PrivC2/Updated_MESI_state_proc	00
/testbench/dut/PrivC2/Updated_MESI_state_snoop	01
/testbench/dut/Arbiter/Com_Bus_Gnt_proc	0100
/testbench/dut/Arbiter/Com_Bus_Gnt_snoop	0001
Now	341 ps
Cursor 1	335 ps

Figure 5: Simulation with Testbench Using Test Vectors Part 3 (CPU_Instructions.tv)



```
Wave - Default
                                                  Msgs
/testbench/dut/Arbiter/Com_Bus_Req_proc           0100
/testbench/dut/Arbiter/Com_Bus_Req_snoop          0011
/testbench/dut/Arbiter/Mem_snoop_gnt              0
/testbench/dut/Arbiter/Mem_snoop_req              St0
/testbench/dut/Arbiter/rst                        St0
/testbench/dut/Memory/memory                      0000000000000000000000cafebabe 0000000000000...
/testbench/dut/Mem_oprn_abort                     1
/testbench/dut/PrivC3/Address                     00000004
/testbench/dut/PrivC3/Address_Com                 00000004
/testbench/dut/PrivC3/All_Invalidation_done       St0
/testbench/dut/PrivC3/Blk_accessed                xx
/testbench/dut/PrivC3/BusRd                       St1
/testbench/dut/PrivC3/BusRdX                      HiZ
/testbench/dut/PrivC3/clk                         St0
/testbench/dut/PrivC3/Com_Bus_Gnt_proc            St1
/testbench/dut/PrivC3/Com_Bus_Gnt_snoop           St0
/testbench/dut/PrivC3/Com_Bus_Req_proc            1
/testbench/dut/PrivC3/Com_Bus_Req_snoop           0
/testbench/dut/PrivC3/CPU_stall                   1
/testbench/dut/PrivC3/Current_MESI_state_proc     00
/testbench/dut/PrivC3/Current_MESI_state_snoop    01
/testbench/dut/PrivC3/Data_Bus                    deadbeef
/testbench/dut/PrivC3/Data_Bus_Com                deadbeef
/testbench/dut/PrivC3/Data_in_Bus                 St0
                                     Now          341 ps
                                Cursor 1          335 ps
```

Figure 6: Simulation with Testbench Using Test Vectors Part 4 (CPU_Instructions.tv)



```
Wave - Default
                                                  Msgs
/testbench/dut/PrivC3/Com_Bus_Gnt_snoop           St0
/testbench/dut/PrivC3/Com_Bus_Req_proc            1
/testbench/dut/PrivC3/Com_Bus_Req_snoop           0
/testbench/dut/PrivC3/CPU_stall                   1
/testbench/dut/PrivC3/Current_MESI_state_proc     00
/testbench/dut/PrivC3/Current_MESI_state_snoop    01
/testbench/dut/PrivC3/Data_Bus                    deadbeef
/testbench/dut/PrivC3/Data_Bus_Com                deadbeef
/testbench/dut/PrivC3/Data_in_Bus                 St0
/testbench/dut/PrivC3/Invalidate                  HiZ
/testbench/dut/PrivC3/Invalidation_done           0
/testbench/dut/PrivC3/LRU_replacement_proc        00
/testbench/dut/PrivC3/Mem_oprn_abort              0
/testbench/dut/PrivC3/Mem_wr                      z
/testbench/dut/PrivC3/Mem_write_done              St0
/testbench/dut/PrivC3/PrRd                        St1
/testbench/dut/PrivC3/PrWr                        St0
/testbench/dut/PrivC3/Shared                      St0
/testbench/dut/PrivC3/Shared_local                0
/testbench/dut/PrivC3/Updated_MESI_state_proc     10
/testbench/dut/PrivC3/Updated_MESI_state_snoop    01
/testbench/dut/PrivC3/cb/Block_Hit_proc           1
/testbench/dut/PrivC3/cb/Cache_var[4]             deadbeef
/testbench/dut/PrivC3/cb/Cache_proc_contr[4]      0000001
                                     Now          341 ps
                                Cursor 1          335 ps
```

Figure 6: Simulation with Testbench Using Test Vectors Part 5 (CPU_Instructions.tv)

RTL Viewer schematics

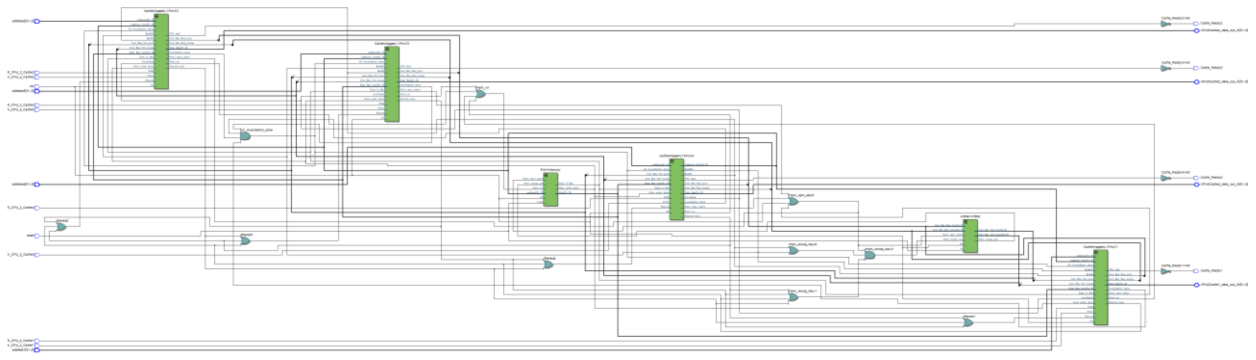


Figure 7: Block Diagram from RTL Viewer (Top Module for Cache Coherence Multi Core)

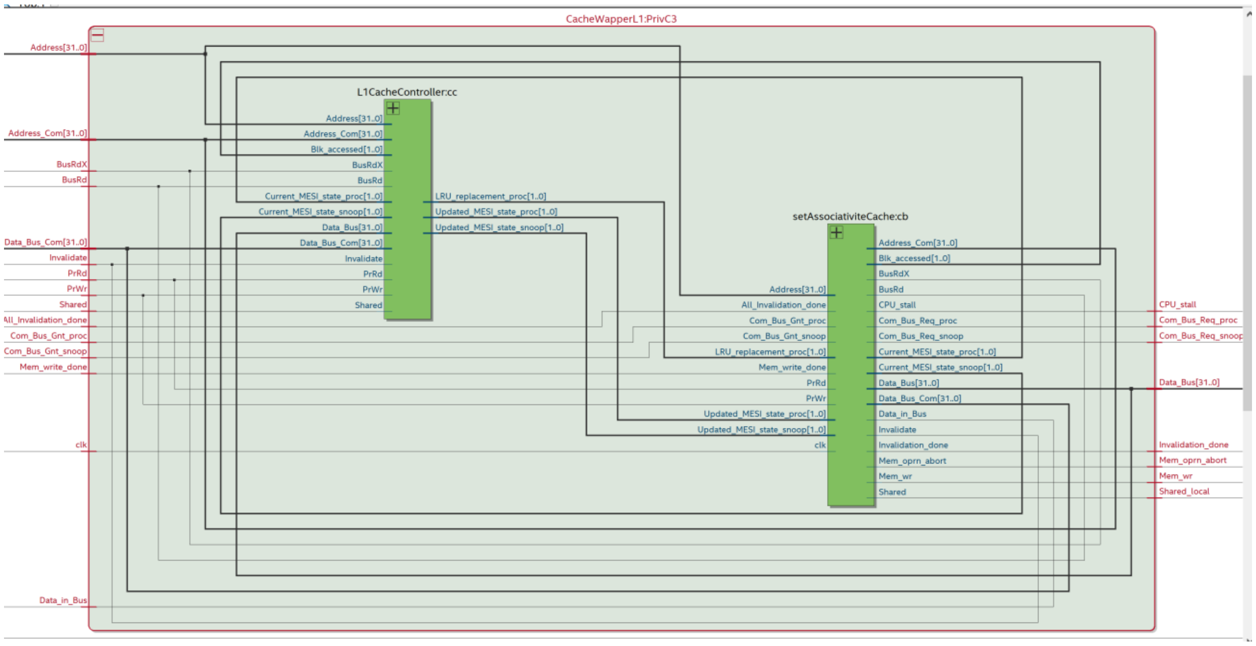


Figure 8: Block Diagram from RTL Viewer (Cache Wrapper)

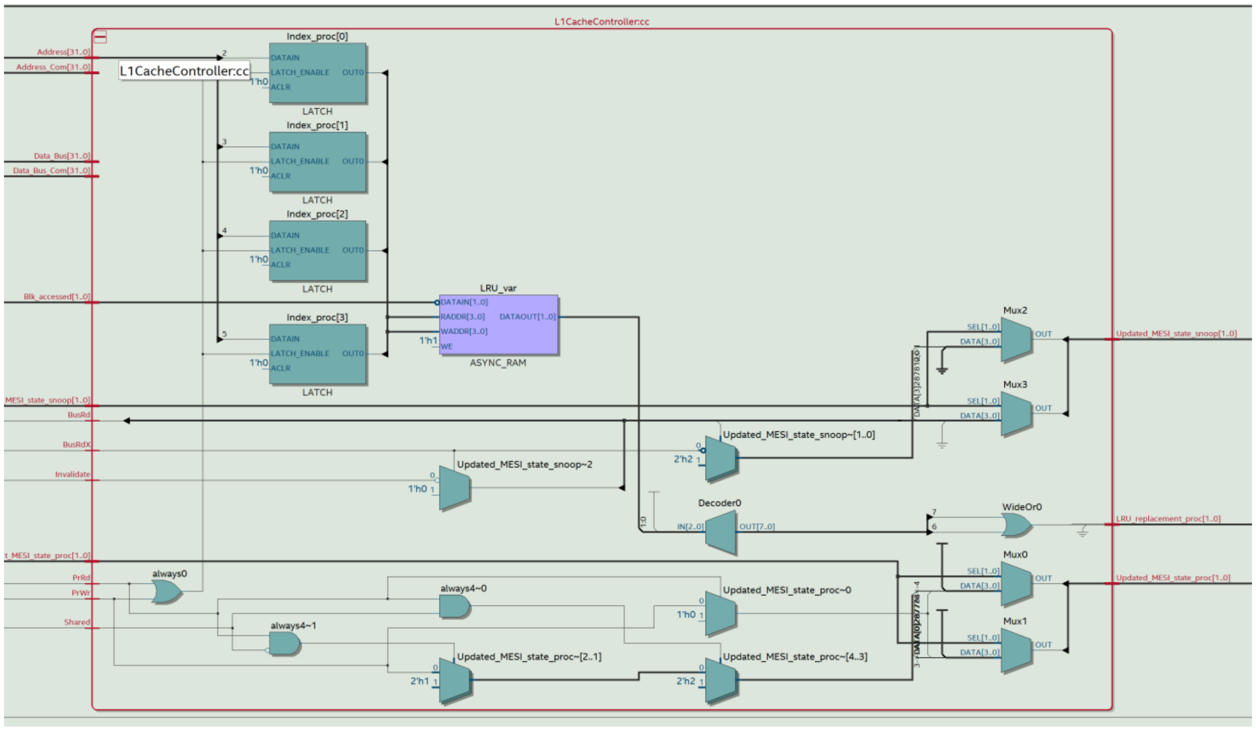


Figure 9: Block Diagram from RTL Viewer (Cache Controller)

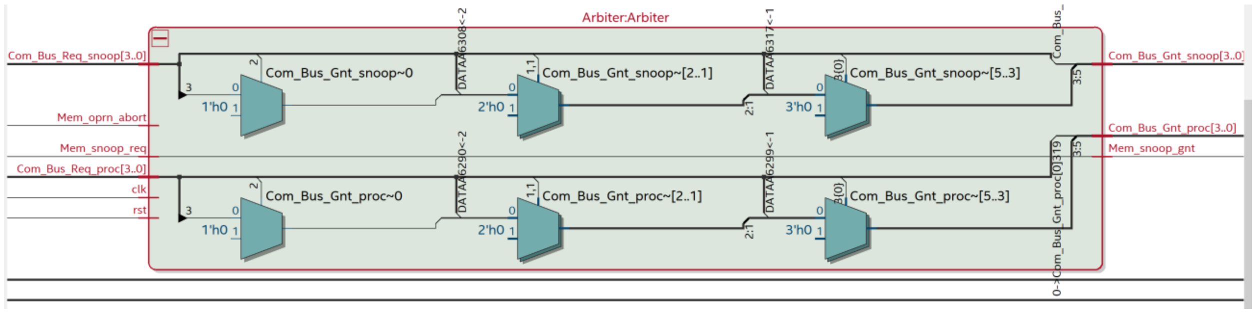


Figure 10: Block Diagram from RTL Viewer (Arbiter)