CSCE 714: Advanced Hardware Design Functional Verification

Texas A&M University

Project Verification Plan

Multicore MESI Based Cache Design HAS

Team 16

Team Members:

Team Member	Name	UIN
1	Vinay Bayaneni	131005670
2	Jyothi Swaroopa Myneedi	232003853
3	Michelle Madubuike	526003184

1. Section 1: Overview of the Design

The Overview of the Design Under Test (DUT) for our verification project is shown below in Figure 1. It consists of multiple IPs joined together through a number of signals for communicating among them.

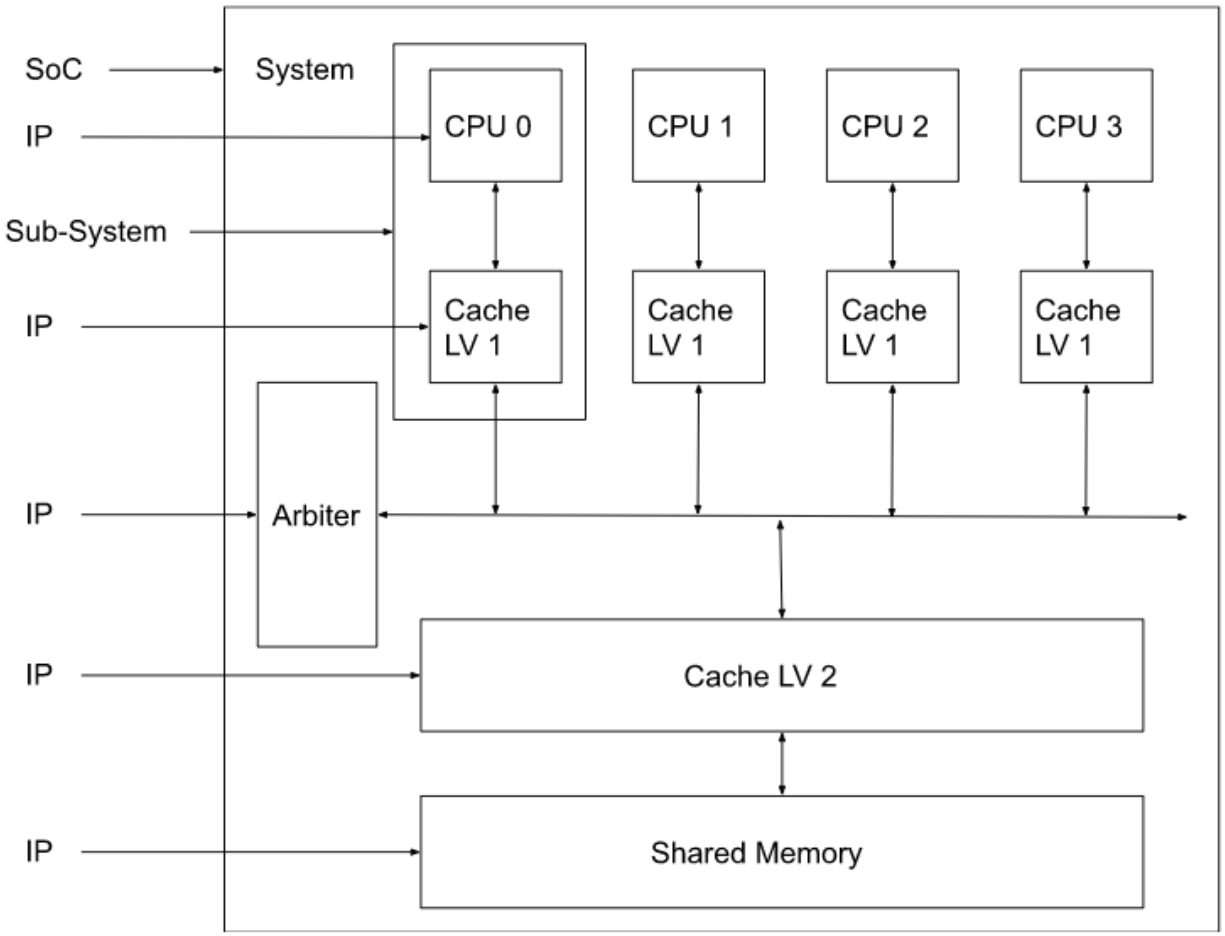

Figure 1. Overview of Design

The Design under Test (DUT) is a 4-core processor with distributed L1 cache and shared single L2 cache. All the communication between L1 caches and between any L1 cache and L2 cache is done through a system bus whose access/grant is decided by an arbiter. The arbiter does not grant bus requests from two or more processors simultaneously. Instead, the processor request grant is in serial manner depending on priority of the request. When there is a L2 cache miss, the request goes to memory and the memory is expected to serve all the requests. L1 cache and L2 caches are set associative caches with L1 being 4-way association and L2 being 8-way associative. Both L1 and L2 caches use Pseudo LRU replacement policy when there is a cache miss and the set is full. L1 has separate data and instruction caches whereas L2 is a unified cache. No write operation is allowed on Instruction Cache. The data from the same address can be shared by caches of different processors and the design involves MESI Cache Coherence protocol inorder to ensure Cache Coherency. Data stored in instruction level cache is not shared so no coherence protocol is needed. Data level and instruction level cache share one bus to communicate with the

processor. Only one of the two levels responds to the processor at a time and which one responds depends on the address. Any address less than 32'h4000_0000 is for instruction cache and the rest is for data cache. The design uses a write-back and write allocate scheme with no write buffers.

Each unicore L1 cache has a cmd and req switches. They also have two wrappers one each for instruction cache and data cache. Each Cache wrapper has a cache controller and a cache block. The Cache controller carries the components to perform LRU and MESI functionality. However, as the instruction cache does not need MESI, the cache controller for instruction cache wrapper does not have a MESI FSM component. The cache block is responsible for bringing the data requested by the processor going through all the required steps following a hit/miss depending on the address and the state of the cache. The main functional block coordinates the overall functionality of the cache system.

To enable simultaneous servicing of both processor side requests and snoop side requests, mesi_fsm_lv1 is separated into proc and snoop parts. The proc's MESI FSM looks at cpu_rd and cpu_wr from the processor along the current MESI state of the block. The next state of the MESI for the block is generated based on this FSM and communicated to cache_block_lv1_dl. The snoop's MESI looks at lv1 lv2 bus signals such as bus_rd and bus_rdx along with current snoop side MESI state. In addition, snoop also handles invalidation requests. It is synchronous to the clock signal, i.e. it computes all the mentioned functions only on the positive edge of the clock.

The functions of L2 cache is similar to that of unicore L1 cache except that the L2 cache does not have a MESI FSM component as there is no need to maintain coherency. Thus, it is similar to Instruction Cache of L1 in some sense. When the processor requests for data, the L1 is searched initially, and upon hit serves the request. If the data is not found in L1, by processor request and snoop request, the request is sent to L2 cache through system bus and finally to memory and eventually gets the data to the processor.

In this design, the complexity is high due to the presence of a large number of IPs like Caches, arbiter, system bus and Memory as shown in Figure 1. The signals connecting them will form a huge number of interfaces and thus involve high complexity in verification. The CPU block is simple as it has to request for either read or write operations on an address. Similarly, Memory is assumed to serve all the requests removing any complexity in its operation. Arbiter and L2 Cache are not as simple as the two previous blocks we discussed. Arbiter should make sure only one request is granted at a time on the system bus even after getting a huge volume of requests. This has to be done based on priority in a priority queue. Finally, the most complex block of the design is the L1 cache due to the communication volume it has with other blocks. So, primary focus shall be given to L1 cache blocks during verification.

2. Section 2: Description of Verification Levels

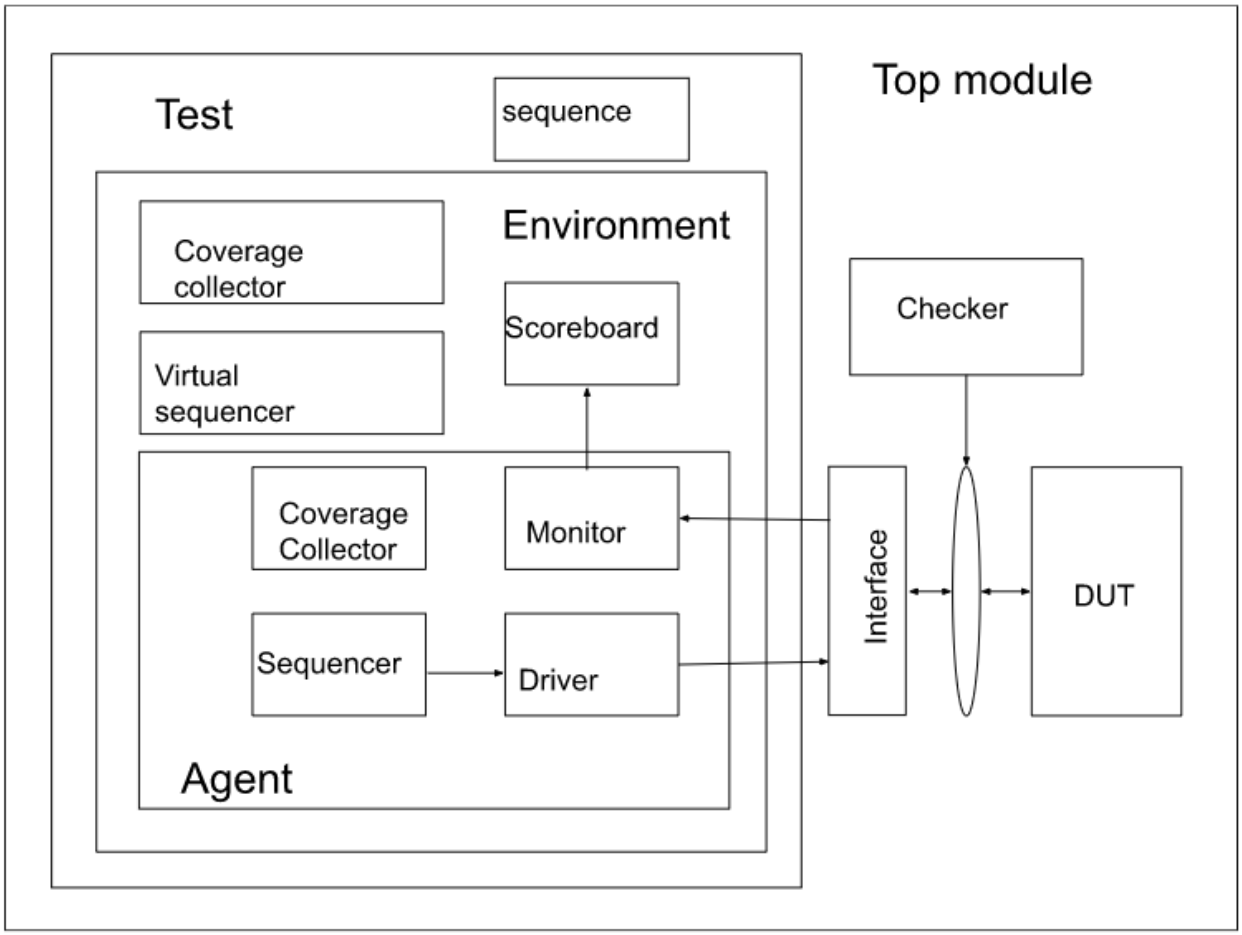

Figure 2. UVM Testbench Block Diagram

For our design, we will be verifying at only one level i.e., the SoC level or system level of design hierarchy. The SoC level is the full 4-core processor chip with distributed L1 and shared L2 cache. The components of our SoC level can be seen in the Overview of our Design in Figure 1. Our SoC level components include various Sub-Systems and IP blocks. In our project, the IPs can be referred to caches, memory, arbiter etc. We verify the functionality of interfaces connecting different IPs and the working of each component when they are combined together. We are not verifying the functionality of each individual component, for example, we are not checking if the LRU policy is correct in the cache_controller. Hence, we are working on a flat verification level, which is SoC level.

For our project, we will be using the Grey-box verification approach to verify our design. This would entail having some components hidden to us (Black-box) while having other components that would be visible to us (White-box). In the black box approach only the inputs and outputs will be visible to us. While, for the white-box the internal of the design will be visible to us as well as the inputs and outputs. Using the Grey Box approach has its own techniques as it would avoid the verification engineer to follow the intent of the designer by not disclosing all the internal components of the DUT.

3. Section 3: Features to be Verified

In this section, we have itemized the features that must be verified before we proceed with bug hunting and coverage closure. The functions to be verified are split into two categories.
1. Critical Features and 2. Secondary Features.

Critical Features are those functions whose failure makes the design to fail. Secondary Features are also important to verify but at a later point after all the critical features have been verified. The features to be verified can be broadly divided into five groups namely L1 cache, L2 cache, System bus, LRU policy and MESI FSM.

Critical Features:

1. **addr_segregator_<proc/snoop> functionality:** The address segregator takes the input address into tag, index and block offset and stores them accordingly for further processing. This feature needs to be tested for all possible addresses and also verify it for invalid addresses.

2. **access_blk_proc[] is onehot:** The access_blk_proc indicates the presence of the requested data in the cache by pointing to the cache line the block is present in. It should be onehot with 1 bit indicating the location of the requested data.

3. **Functionality of read hit in L1 cache:** When the processor read hit in the L1 cache, cache's data value is put in data_bus_cpu_lv1 of that processor and data_in_bus_cpu_lv1 is asserted. Bus_lv1_lv2_req_proc is deasserted.

4. **Functionality of read miss in L1 cache:** When the processor read miss in the L1 cache, cache requests for data from other caches or L2/memory and the MESI states change accordingly.

5. **Functionality of write hit in L1 cache:** When the processor write hit in the L1 cache, necessary transitions of mesi states need to be verified if the copy is in shared state.

6. **Functionality of write miss in L1 cache:** When the processor write miss in the L1 cache, cache requests for data from other caches or L2/memory and the MESI states change accordingly.

7. **Write operation on instruction Cache:** cache_block_lv1_il does not allow write operation.

8. **Timing check**: Timing synchronization to make sure the above-mentioned functions compute only on the positive edge of clock.

9. **MESI state transitions:** MESI cache coherence protocol for all the possible scenarios need to be verified to make sure that the coherency is maintained.

10. **Pseudo LRU policy for all states:** The block eviction based on pseudo least recently used policy for all the possible states need to be verified to ensure that only the expected blocks are getting evicted.

11. **Snoop side activity for read/write miss and data in other caches:** Snoop side activities need to be verified for a read or write miss and the address available in other caches.

Secondary Features:

12. **cp_in_cache check:** If block is hit in other snooping cache, level 1 caches with copies rise cp_in_cache then, cache_proc_contr value needs to be updated.

13. **lv2_wr and lv2_wr_done deassertion**: When lv2_wr is deasserted then, lv2_wr_done is deasserted in the next clock cycle.

14. **Blk_hit_proc assertion**: Blk_hit_proc should be 0 if access_blk_proc is 4b'0000. The former is one if the latter has a different state than all zeroes indicating a cache hit.

15. **blk_free flag status:** If there is an invalid block in the set, blk_free flag should be high.

16. **mem_wr_done assertion**: If the mem_wr_done signal is raised then, our mem_wr is to be deasserted.

Non-verifiable Feautures:

1. **Golden Memory Module:** Memory is considered to be ideal and its working is not verified.
2. **Arbiter mechanism:** Arbiter mechanism to grant bus requests and processor requests is also not verified.
3. **Write-back mechanism**: Write-back mechanism of L1-L2 cache interface is also not verified.

4. Section 4: Test Methods and Scenarios

In this section, we described our testcases and checkers that we developed to verify the features we mentioned in the previous section. Listed below we show our testcases and checking as well as the features that they would be verifying.

1. **Read_miss_icache_test** : Do a read operation on ICache. This test helps in verification of the functions 1,2, and 4 in section 3.
2. **five_sequence_test** : Do 5 reads on ICache. This test helps in verification of the functions 1, 2, 3, 4 and 8 in section 3.
3. **Read_miss_dcache** : Do a read operation on Data Cache. This test helps in verification of the functions 1, 2,4 , 13 and 16 in section 3.
4. **five_sequence_test** : Do 5 reads on Data Cache. This test helps in verification of the functions 1, 2, 3, 4 and 8 in section 3.
5. **illegal_write_test**: Do a write operation on ICache. This test is used to verify the function 7 in section 3.
6. **write_miss_dcache:** D0 a write operation on DCache. This test is used to verify the function 6 in section 3.
7. **rd_wr_rd_test**: Do a read and immediately write followed by read operations on DCache. This test is used to verify the function 4, 5, 6 and 14 in section 3.
8. **rd_wr_dcache_test** : Do a read and immediately write operation on Data Cache. This test helps in the verification of the functions 4, 5, 6 and 14 in section 3.
9. **five_wr_test** : Do five writes on D cache. This test helps in verification of the function 5, 6 and 11 in section 3.
10. **mixed_long_test**: Do 20 operations on D Cache with read and write operations alternatively. This test helps in verification of all the functions 1-6, 8 and 11 mentioned in section 3.
11. **rd_same_test** : Read from the same location 5 times. This test helps in the verification of the functions 1-3 in section 3.
12. **multiple_proc_rd_test** : Read from the multiple processors chosen randomly. This test helps in the verification of the functions 1-3 in section 3.
13. **rd_same_icache_test** : Read from the same location in Icache 5 times. This test helps in the verification of the functions 1-3 in section 3.
14. **wr_wr_rd_test** : Write to the same location twice followed by a read. This test helps in the verification of the function 5 and 9 in section 3.
15. **wr_rd_wr_test** : Write to the same location followed by a read and then write again. This test helps in the verification of the function 5 and 9 in section 3.
16. **rd_rd_wr_test** : Read to the same location followed by a read and then write again. This test helps in the verification of the function 5 and 9 in section 3.
17. **Shared_to_shared**: Test to verify the transition of MESI FSM from Shared to shared state. This test helps in the verification of the functions 9, 11 and 12 in section 3.
18. **Wr_same_dcache**: Write to same address in dcache from different processors. This test helps in the verification of the functions 5, 6 and 11 in section 3.
19. **Shared_test**: Test to verify the functioning of shared state in MESI FSM. This test helps in the verification of the functions 9, 11 and 12 in section 3.
20. **inv_to_shared_test**: Test to verify the functioning of transition from invalid to shared state in MESI FSM. This test helps in the verification of the functions 9, 11 and 12 in section 3.

21. **modified_to_invalid_test**: Test to verify the functioning of transition from modified to invalid state in MESI FSM. This test helps in the verification of the functions 9, 11 and 12 in section 3.
22. **mix_icache_dcache_test**: Read and write operations alternatively on icache and dcache. This test helps in the verification of the functions 1-6, 9, 11 and 12 in section 3.
23. **Read_for_replacement_test**: Do a read operation on same block to index into same cache line to verify the block eviction is following plru policy. This test helps in verification of the feature 10, 11 and 15 in section 3.
24. **five_read_on_two_proc:** Do 5 read on proc[0], then do 5 read on the same address as of proc[0] on proc[1]. This test helps us to verify feature 11 in section 3.
25. **Mesi_check:** Test to recreate all the possible transition scenarios of a mesi FSM. This test helps us to verify feature 9, 10, 11, 12 and 14 in section 3.
26. **Random_ops:** Test to choose the request_type randomly on different processors. This test is used to verify functions 8 and 11 in section 3.
27. **Lru_test:** Test to verify the block eviction is done according to lru in more randomized manner. This test helps us to verify feature 10, 11 and 15 in section 3.
28. **Lru_test:** Test to verify the block eviction is done according to lru in more randomized manner in icache. This test helps us to verify feature 10, 11 and 15 in section 3.
29. **Invalid_check:** Test to verify the functioning of invalid state in MESI FSM. This test helps in the verification of the functions 9, 11 and 12 in section 3.
30. **Full_random:** Test with randomized processors doing operations which are chosen randomly on random addresses. This test helps in the verification of the functions 1-16 section 3. It also helps to close the coverage holes and achieve coverage closure.
31. **test_write_miss_snoop:** Test to verify the functioning of snooping when write miss occurs. This test helps in the verification of functions 9, 10, 11 and 13 in section 3.

We can see that with the above-mentioned tests, we can cover all the features mentioned in Section 3.

Assertions/Checkers used:

Cpu-lv1 cache interface:

1. **Assert_simult_cpu_wr_rd:** Cpu_wr and cpu_rd should not be asserted simultaneously. This assertion is used to verify the functions 3, 4, 5, 6 and 8 in section 3.
2. **Assert_valid_addr_wr_rd :** address should not be invalid when rd/wr request is processed. This assertion is used to verify the functions 1 in section 3.
3. **Assert_valid_rd_txn:** cpu_rd should be followed by data_in_bus_cpu_lv1 and both should be deasserted according to HAS. This assertion is used to verify the functions 3, 4, 11 and 12 in section 3.
4. **Assert_valid_data_in_bus_rd:** If data_in_bus_cpu_lv1 is asserted, cpu_rd should be high. This assertion is used to verify the functions 3, 4, 5, 6 , 8, 11 and 12 in section 3.
5. **Assert_valid_cpu_wr_done_seq:** cpu_wr should be followed by cpu_wr_done and then both should be deasserted according to HAS. This assertion is used to verify the functions 1-6 and 8-15 in section 3.
6. **Assert_cpu_wr_done_timeout_check:** cpu_wr_done should be asserted within 100 cycles of asserting cpu_wr. This assertion is used to verify the functions 8 in section 3.

7. **Assert_data_in_bus_timeout_check:** data_in_bus_cpu_lv1 should be asserted within 100 cycles of asserting cpu_rd. This assertion is used to verify the functions 8 in section 3.
8. **Assert_prop_simult_cpu_wr_done_data_in_bus**: cpu_wr_done and data_in_bus_cpu_lv1 should not be asserted at the same clock cycle. This assertion is used to verify the functions 1- 6 and 8-15 in section 3.
9. **Assert_prop_simult_cpu_wr_done_cpu_rd:** cpu_wr_done and cpu_rd should not be asserted at the same clock cycle. This assertion is used to verify the functions 3, 4, 5, 6 and 8 in section 3.
10. **Assert_prop_simult_cpu_wr_data_in_bus:** cpu_wr and data_in_bus_cpu_lv1 should not be asserted at the same clock cycle. This assertion is used to verify the functions 3, 4, 5, 6 and 8 in section 3.
11. **Assert_valid_cpu_wr_on_cache:** no cpu_wr on I-cache. This assertion is used to verify the functions 7 in section 3.
12. **Assert_valid_data_in_bus:** data_in_bus_cpu_lv1 is asserted with a previous cpu_rd signal. This assertion is used to verify the functions1- 6 and 8-15 in section 3.
13. **Assert_cpu_rd_deassert:** data_in_bus_cpu_lv1 assertion should result in deassertion of cpu_rd signal. This assertion is used to verify the functions 7 and 8 in section 3.
14. **Assert_cpu_wr_deassert:** cpu_wr_done assertion results in deassertion of cpu_wr. This assertion is used to verify the functions 1- 6 and 8-15 in section 3.
15. **Assert_valid_cpu_wr_done:** cpu_wr_done is asserted with a previous cpu_wr signal. This assertion is used to verify the functions 1-6 and 8-15 in section 3.

Lv1-Lv2-System bus interface:

16. **assert_lv2_wr_done:** lv2_wr_done should not be asserted without lv2_wr being asserted in the previous cycle. This assertion is used to verify the functions 13 in section 3.
17. **assert_data_in_bus_lv1_lv2_cp_in_cache:** data_in_bus_lv1_lv2 and cp_in_cache should not be asserted without lv2_rd being asserted in previous cycle. This assertion is used to verify the functions 9, 11, 12, 14 and 15 in section 3.
18. **assert_valid_lv2_rd_txn:** lv2_rd should be followed by data_in_bus_lv1_lv2 and both should be deasserted acc to HAS. This assertion is used to verify the functions 4, 6 and 8 in section 3.
19. **assert_valid_data_in_lv2_bus_rd:** If data_in_bus_lv1_lv2 is asserted, lv2_rd should be high. This assertion is used to verify the functions 4, 6, 8,9 and 11 in section 3.
20. **assert_valid_lv2_wr_txn:** lv2_wr should be followed by lv2_wr_done and both should be deasserted acc to HAS. This assertion is used to verify the functions 4, 6 and 8 in section 3.
21. **assert_invalidation_done:** all_invalidation_done should not be asserted without invalidate being asserted in the previous cycle. This assertion is used to verify the functions 9, 11 and 12 in section 3.
22. **assert_prop_cp_in_cache:** Copy in cache should be asserted only when bus_rd or bus_rdx in previous cycle. This assertion is used to verify the functions 11 and 12 in section 3.
23. **assert_valid_lv2_rd_rdx:** bus_rd and bus_rdx must be asserted on processor receiving grant and lv2_rd. This assertion is used to verify the functions 4, 6 and 8 in section 3.
24. **assert_valid_lv2_wr_done:** lv2_wr_done should be asserted in previous cycle before lv2_wr gets deasserted. This assertion is used to verify the functions 4, 6 and 8 in section 3.

25. **assert_valid_bus_lv1_lv2_req_snoop:** bus_lv1_lv2_req_snoop should be followed by bus_lv1_lv2_gnt_snoop acc to HAS. This assertion is used to verify the functions 8, 9, 10, and 11 in section 3.
26. **assert_valid_bus_lv1_lv2_req_proc:** bus_lv1_lv2_req_proc should be followed by bus_lv1_lv2_gnt_proc acc to HAS. This assertion is used to verify the functions 11 in section 3.
27. **assert_valid_bus_rd_rdx:** bus_rd and bus_rdx should not be asserted simultaneously. This assertion is used to verify the functions 8, 9, 10 and 11 in section 3.
28. **assert_simult_snoop_check**: All Processors are snooping simultaneously which should not happen as one processor should always request. This assertion is used to verify the functions 10 and 11 in section 3.
29. **assert_valid_inv_gnt:** Invalidate should not be asserted without proccessor getting grant in previous cycle. This assertion is used to verify the functions 9 in section 3.
30. **assert_valid_shared_data_in_bus:** If shared is asserted, data_in_bus_lv1_lv2 should be high. This assertion is used to verify the functions 9 in section 3.
31. **assert_valid_data_in_data_bus:** If data is put in data_bus_lv1_lv2, data_in_bus_lv1_lv2 should be made high immediately. This assertion is used to verify the functions 3, 4, 5, 6 and 8 in section 3.
32. **assert_valid_des_invalidate:** After all_invalidation_done is asserted , deassert invalidate. This assertion is used to verify the functions 9, and 11 in section 3.
33. **assert_data_in_bus_rdx_rd:** lv2_rd or bus_rdx or bus_rd should be deasserted after data_in_bus_lv1_lv2 being asserted. This assertion is used to verify the functions 3, 4, 5, 6 and 11 in section 3.
34. **assert_valid_addr_lv2_rd_wr:** Address should not be invalid when lv2_rd/wr request is processed. This assertion is used to verify the functions 3, 4, 5, 6 and 11 in section 3.
35. **assert_valid_proc_bus_gnt:** System bus grant should not be given to multiple processors at the same time. This assertion is used to verify the functions 3, 4, 5, 6 and 8 in section 3.
36. **assert_valid_snoop_bus_gnt:** System bus grant should not be given to multiple snooping processors at the same time. This assertion is used to verify the functions 11 in section 3.
37. **assert_bus_lv1_lv2_req_lv2:** bus_lv1_lv2_req_lv2 should not be asserted without lv2_rd or lv2_wr being asserted in previous cycle. This assertion is used to verify the functions 3, 4, 5, 6 and 11 in section 3.
38. **assert_bus_lv1_lv2_req_gnt_proc:** bus_lv1_lv2_gnt_proc should be asserted only if previous bus_lv1_lv2_req_proc. This assertion is used to verify the functions 3, 4, 5, 6 and 11 in section 3.
39. **assert_bus_lv1_lv2_req_gnt_snoop:** bus_lv1_lv2_gnt_snoop should be asserted only if previous bus_lv1_lv2_req_snoop. This assertion is used to verify the functions 3, 4, 5, 6 and 11 in section 3.
40. **assert_invalidate_all_invalidation_done:** all_invalidation_done should be asserted only if previous invalidate. This assertion is used to verify the functions 9 in section 3.
41. **assert_shared_bus_gnt:** shared should be asserted only if previous bus_lv1_lv2_gnt_proc. This assertion is used to verify the functions 9 and 11 in section 3.
42. **assert_invalidate_bus_gnt:** invalidate should be asserted only if previous bus_lv1_lv2_gnt_proc. This assertion is used to verify the functions 9 in section 3.
43. **assert_bus_rd_bus_gnt:** bus_rd should be asserted only if previous bus_lv1_lv2_gnt_proc. This assertion is used to verify the functions 3, 4, 5, 6 and 8 in section 3.

44. **assert_bus_rdx_bus_gnt:** bus_rdx should be asserted only if previous bus_lv1_lv2_gnt_proc. This assertion is used to verify the functions 3, 4, 5, 6 and 11 in section 3.
45. **assert_bus_lv2_req_gnt:** bus_lv1_lv2_gnt_lv2 should be asserted only if previous bus_lv1_lv2_req_lv2. This assertion is used to verify the functions 3, 4, 5, 6 and 11 in section 3.
46. **assert_data_in_bus_req_gnt:** data_in_bus_lv1_lv2 should be asserted only if previous bus_lv1_lv2_gnt_lv2 or bus_lv1_lv2_gnt_snoop. This assertion is used to verify the functions 4, 6 and 11 in section 3.

5. Section 5: Design Functionality Coverage

Coverage describes the scope of the verification problem. In order to prove the completeness of our verification effort, we will develop and track the following different types of coverages.

1. Code Coverage.
   1.1. Block Coverage
   1.2. Expression Coverage and
   1.3. Toggle Coverage
2. Functional Coverage.

We plan to achieve the following target percentage of coverage for each coverage type mentioned above for Coverage Closure as shown in the below table:

S.No	Coverage Type	Target Coverage %
1	**Code Coverage**	90%
	1.1. Expression Coverage	85%
	1.2. Block Coverage	90%
	1.3. Toggle Coverage	90%
2	**Functional Coverage**	95%

Table 1: Target Coverages

We want all the possible cases to be hit for the coverage closure. So we try to include the coverpoints like

1. To check if all the 4 L1 caches are covered
2. All the 4 cpu requests are served
3. Both read and write requests are served
4. Cache miss and hit cases for each cache
5. We can cross the cases in 1-4 to create cross bins to increase the coverage bins further.

Once we reach the Target coverage as specified in table 1, we will have the coverage closure.

The coverpoints included in the monitor files are as follows:

Cpu_monitor:

1. REQUEST: Request of operation either rd or wr
2. CACHE_TYPE: DCache or ICache
3. DATA:
4. ADDR

CROSS COVERAGE

1. X_REQUEST_CACHE_TYPE : cross REQUEST, CACHE_TYPE

We make sure that the cpu_wr on Icache is made illegal and thus excluded from the coverage.

System_bus_monitor:

1. REQUEST_TYPE
2. REQUEST_PROCESSOR
3. REQUEST_ADDRESS

4. READ_DATA
5. BUS_REQ_SNOOP
6. REQ_SERV_BY
7. CP_IN_CACHE
8. WRITE_DATA
9. REQUEST_WRITE
10. EVICTION_ADDRESS
11. EVICTION_DATA

CROSS COVERAGE

1. X_PROC_REQ_TYPE: cross REQUEST_TYPE, REQUEST_PROCESSOR
2. X_PROC_ADDRESS: cross REQUEST_PROCESSOR, REQUEST_ADDRESS
3. X_PROC_DATA: cross REQUEST_PROCESSOR, READ_DATA
4. X_PROC_CP_IN_CACHE: cross REQUEST_PROCESSOR, CP_IN_CACHE
5. X_PROC_EVICT_DATA: cross REQUEST_PROCESSOR, EVICTION_DATA
6. X_PROC_EVICT_ADDR: cross REQUEST_PROCESSOR, EVICTION_ADDRESS
7. X_PROC_WRITE_DATA: cross REQUEST_PROCESSOR, WRITE_DATA
8. X_PROC_SERVICE: cross REQUEST_PROCESSOR, REQ_SERV_BY

Coverage Achieved:

The screenshot of the coverage achieved is attached herewith. As our target coverage as mentioned in table 1 achieved, we initiated the coverage closure.

Exclusions made:

Following exclusions are made during coverage analysis

a. Assertions for arbiter (1.2.2)
b. Assertions for Lv2-memory (1.2.4)
c. Address Segmentation (1.3)
d. Functional coverage for LRU (1.9.2)
e. Functional Coverage for MESI (1.9.3)
f. Functional Coverage for Arbiter (1.9.1)

All the above-mentioned exclusions are saved in the refinement file named ProjectVerificationPlanSpring2023.vpRefine

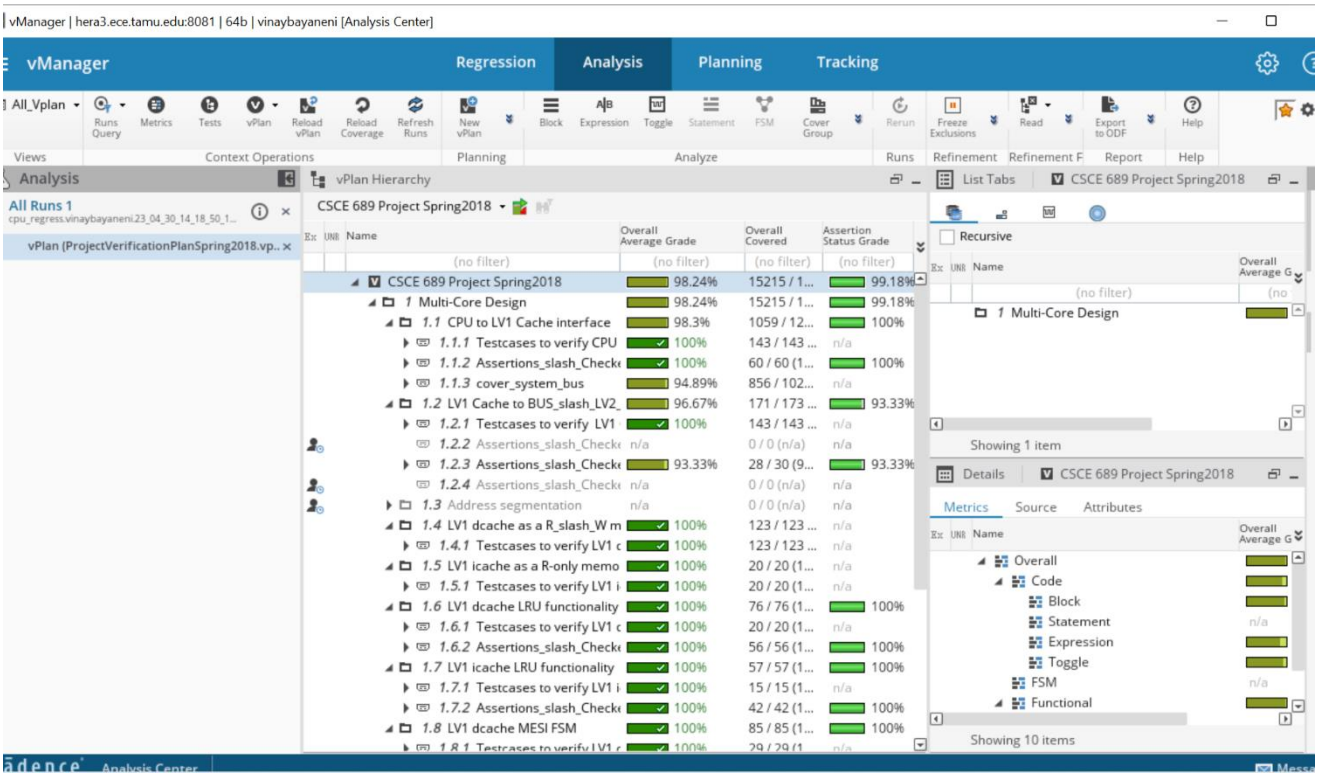

Figure 3: Coverage Analysis Part 1

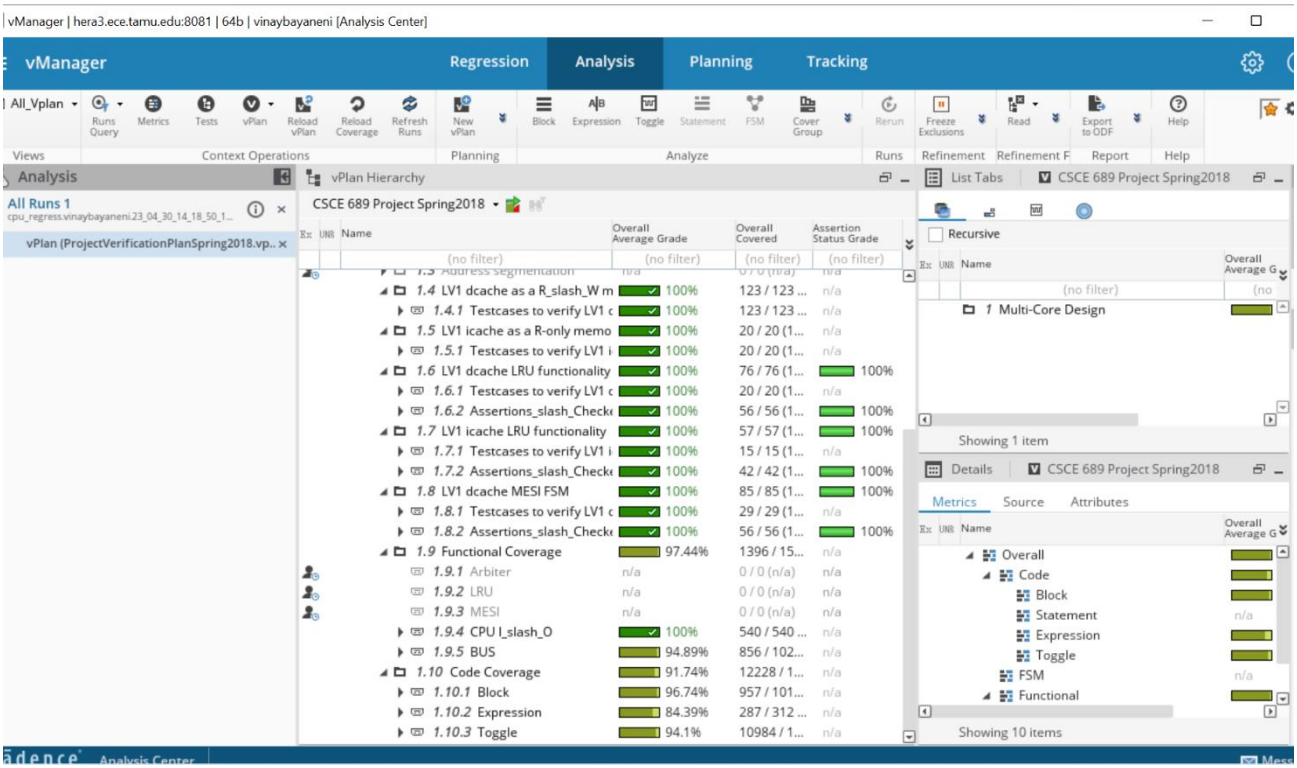

Figure 4: Coverage Analysis Part 2