Multicore MESI Based Cache Design HAS

Ver. 2.5.2

1. Introduction

The Design is targeted at developing a DUT comprising of L1 and L2 cache systems which can be utilized for undertaking functional verification. The DUT works in an environment which contains 4 processor stubs for each L1, a bus system to carry out transactions among L1 caches and also between L1 & L2, an arbiter to determine bus access, and a stub for main memory. The DUT implements functional aspects of L1 cache coherency protocols and hence may not contain performance upgrades/components. The basic block representation of the system is as shown below:

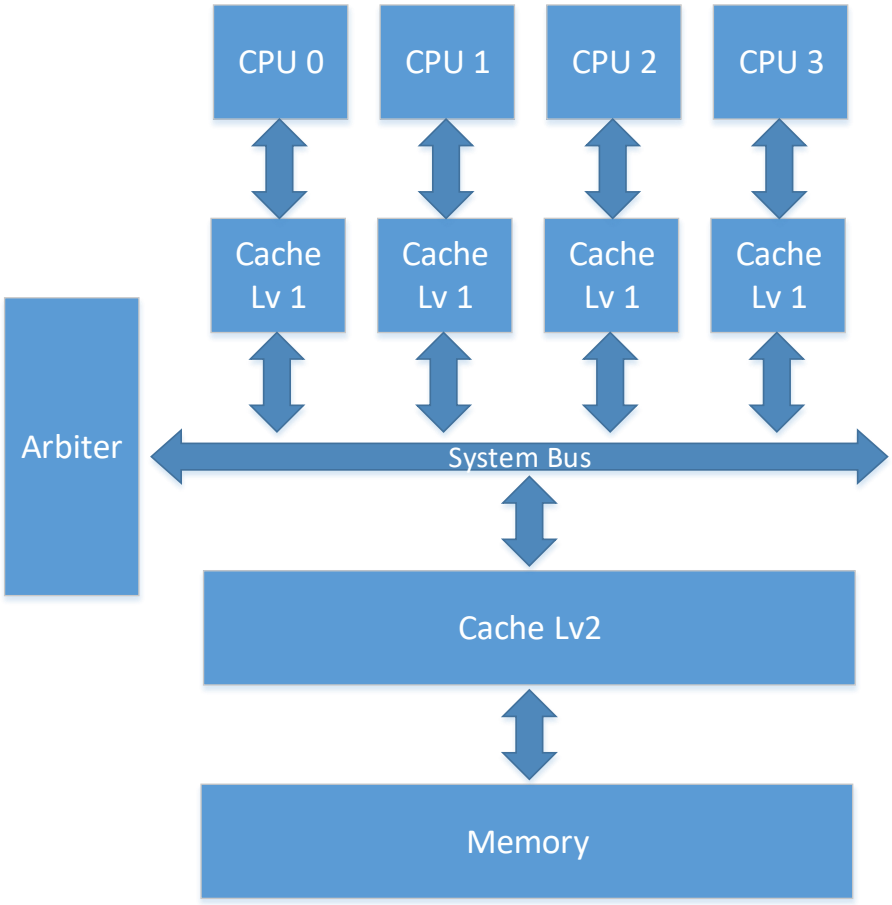

Figure 1-1

The basic design specifications are based on following

1. 32 bit 4 core processor system
2. L1 cache for each processor with Shared L2 memory.
3. Communication between L1 & L2 and among L1s happens through system bus.
4. The Grant/Access of the Bus is decided by an Arbiter
5. Memory is a stub which is assumed to serve all its requests

6. Physically Indexed Physically Tagged Cache system (PIPT) – no TLB
7. Data and instruction caches are separated in L1. But L2 is a unified cache.
8. N-way associative cache: 4-way in L1 and 8-way in L2.
9. MESI based coherency protocol in L1.
10. Pseudo LRU replacement policy.
11. Write-back and write allocate scheme.
12. No write buffers.

2. Level 1 Cache Design

2.1. General Description

The level 1 caches are the first level data storage which interact directly with processors. There are 4 level 1 caches in the design. Each cache provides data service to one processor. All 4 caches have the same structure and function. Every cache has two parts: instruction level cache and data level cache. Both instruction and data level cache is a 256 KB 4-way set associative cache with pseudo-LRU replacement policy. Contents in each data level cache can be shared by all other data level caches and MESI protocol is used to ensure data 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.

2.1. Algorithm Description

2.1.1 Pseudo-LRU Algorithm

The level 1 cache follows a pseudo-LRU algorithm. Each set has a state variable to record its status. The state variable indicates which line should be replaced when there is no free line. Every access to the set will change the state variable. The LRU state - replacement relation and next state transition are shown in table 2-1.

state	replacement
00x	line 0
01x	line 1
1x0	line 2
1x1	line 3

refer to	next state
line 0	11-
line 1	10-
line 2	0-1
line 3	0-0

Table 2-1 (‘x’ means don’t care, ‘-’ means unchanged)

2.1.2 MESI protocol

Each cache line has its own MESI state: Modified, Exclusive, Shared or Invalid. MESI state is updated according to the processor and bus operation. The MESI state transition diagram is shown in Fig. 2-1

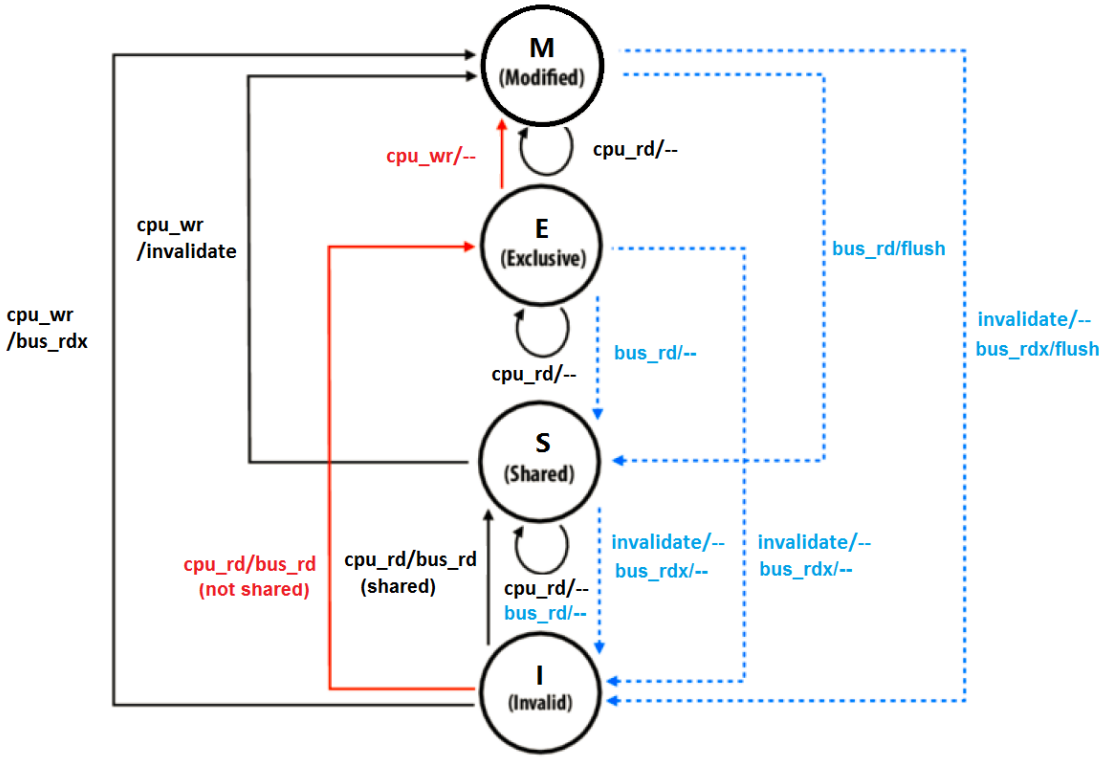

Figure 2-1

The two components of the transition arrows are <For what the request is raised> / <what operation needs to be carried out>. Eg: bus_rdx / flush means When a bus_rdx request is raised for a cache data block from a Modified state, flush operation needs to be carried out along with MESI state change. Flush means writing back the Modified (dirty) block into lower memory. pr_wr – Processor Write Request; pr_rd - Processor Read Request, bus_rd – Request for read on bus ; bus_rdx – Request for read on bus with intention to modify. 'shared' is an additional signal used in this MESI implementation.

2.2. Structures and Implementation

The top level module of level 1 cache is cache_lv1_multicore. It consists of 4 cache_lv1_unicore modules. Each unicore module is responsible for serving the requests of one processor. A cache_lv1_unicore module has data and instruction cache modules named cache_wrapper_lv1_dl and cache_wrapper_lv1_il. Data and instruction cache use different address segments. Any address less than 32'h4000_000 is for instruction cache and the rest is for data cache. In cache_lv1_unicore level there are cmd_switch and req_switch to decide which cache wrapper should respond to the processor requests. The wrapper level the basic cache component consisting of a cache block and cache controller. So the cache_wrapper_lv1_dl has two sub modules: cache_controller_lv1_dl and cache_block_lv1_dl. Similarly, cache_wrapper_lv1_dl has two modules with the same names but has suffix _il.

The cache_controller_lv1_dl encompasses the FSMs of MESI and Pseudo LRU function, which are implemented in mesi_fsm_lv1 module and lru_block_lv1 module respectively.

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 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.

Unlike MESI, pseudo LRU needs LRU state storage units for each set only. Hence, LRU state storage is incorporated inside the LRU block itself with register named lru_var. The LRU functionality has two parts: one to determine the replacement block based on current LRU state and another to update LRU state based on what block being accessed in a set.

The cache_controller_lv1_il has only LRU block. Since instruction cache does not share with others, MESI block is not needed.

The cache_block_lv1_dl consists of a main functional block "main_func_lv1_dl" and other supporting modules. Supporting modules are divided into processor side and snoop side, indicated by "proc" and "snoop" in their names.

The cache storage "cache_var" and cache tag/MESI storage "cache_proc_contr" reside in main functional block. They are implemented based on below examples.

storage location	cache_var	set number	line number
[0]	Data	set 0	line 0
[1]			line 1
[2]			line 2
[3]			line 3
[4]		set 1	line 0
[5]			line 1
[6]			line 2
[7]			line 3
…		…	…

storage location	cache_proc_contr		set number	line number
[0]	Tag	MESI	set 0	line 0
[1]				line 1
[2]				line 2
[3]				line 3
[4]			set 1	line 0
[5]				line 1
[6]				line 2
[7]				line 3
…			…	…

Table 2-1

The actual storage structures Cache_var and Cache_proc_contr are initialied to Invalid state while bringing up the system and hence, the testbench may assume that cache is empty initially and proceed from there.

a) addr_segregator_<proc/snoop>

The addr_segregator_<proc/snoop> units separates index, tag and blk_offset details from the address and place them in index_<proc/snoop>, tag_<proc/snoop> and blk_offset_<proc/snoop>.

b) access_blk_<proc/snoop>_md

Once a request is raised and address is segregated into various components, access_blk_<proc/snoop>_md looks into all the cache lines of the particular cache set to determine if the data block is present in the cache or not. This functional block compares tag_<proc/snoop> with all the cache lines' tag to find if there is a match and confirms that MESI state of that line is not Invalid. To resemble hardware implementation and to support simultaneous comparison of all lines' details, a variable equivalent to the size of number of cache lines in a set is used (One hot usage). The variable is named access_blk_proc [] and one of its bit is assigned 1 if cache data block is present in that line number. For example, 4'b1000 means line 3 is a hit.

c) blk_hit_<proc/snoop>_md

This block just looks at the access_blk_proc [] value and determines if the request is a hit in cache or not. If all bits of Access_blk_proc is '0' then it's not a hit else if any one of the bit is '1' then it is a hit. Accordingly, the signal block_hit_proc is asserted or deasserted.

d) free_block_md

To determine if there is any free block, each lines' MESI state is looked for Invalid state. If any line is Invalid, free_blk is updated with the line details (BLK1 / BLK2 / BLKL3 / BLK4 in 4-way set assoc.) and blk_free flag is also set if any free block is found.
The replacement blocks detail is provided by cache controller to cache block through lru_replacement_proc[] signal.

e) block_to_be_accessed_<proc/snoop>_md

With all these above inputs the blk_access_<proc/snoop> signal is updated in this block with details of the block which is determined to be accessed. For proc side, it is determined by first checking if blk is a hit or not (blk_hit_<proc/snoop>), if hit then based on access_blk_proc[] value blk_access_proc[] is updated, else if there is a free block (by checking blk_free flag) then free_blk value is chosen else if no free block is available then replacement needs to be done, which is given by the cache controller through lru_replacement_proc signal. For snoop side, it does not check blk_free and it does not need replacement. This blk_access_<proc/snoop> is then given to the main functional block for other operations.

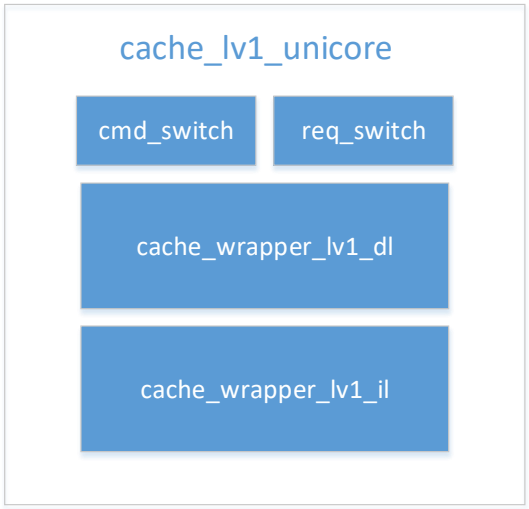

Figure 2-2

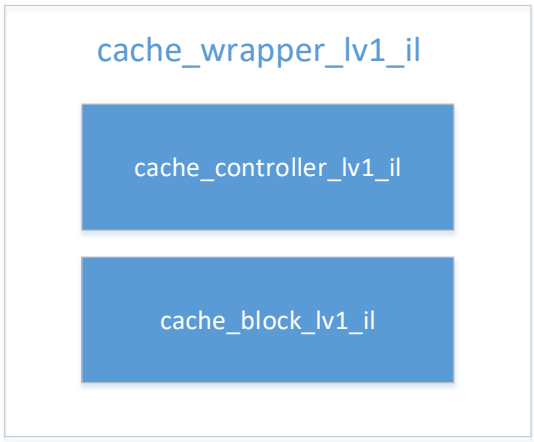

Figure 2-3

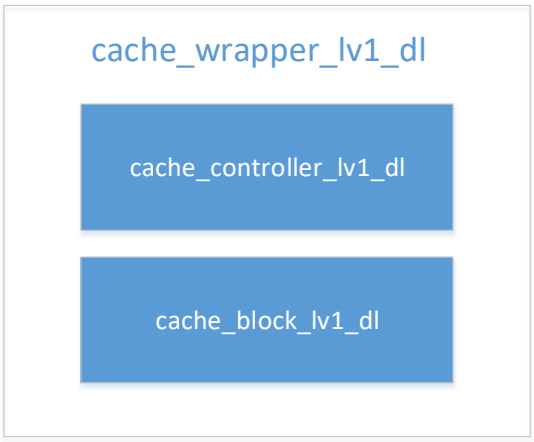

Figure 2-4

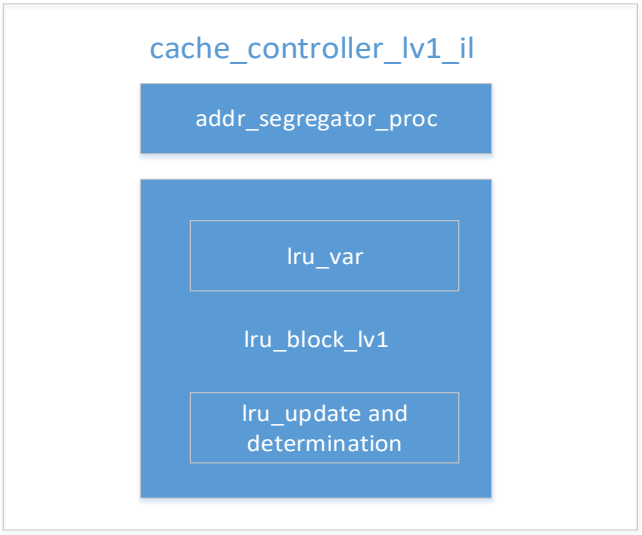

Figure 2-5

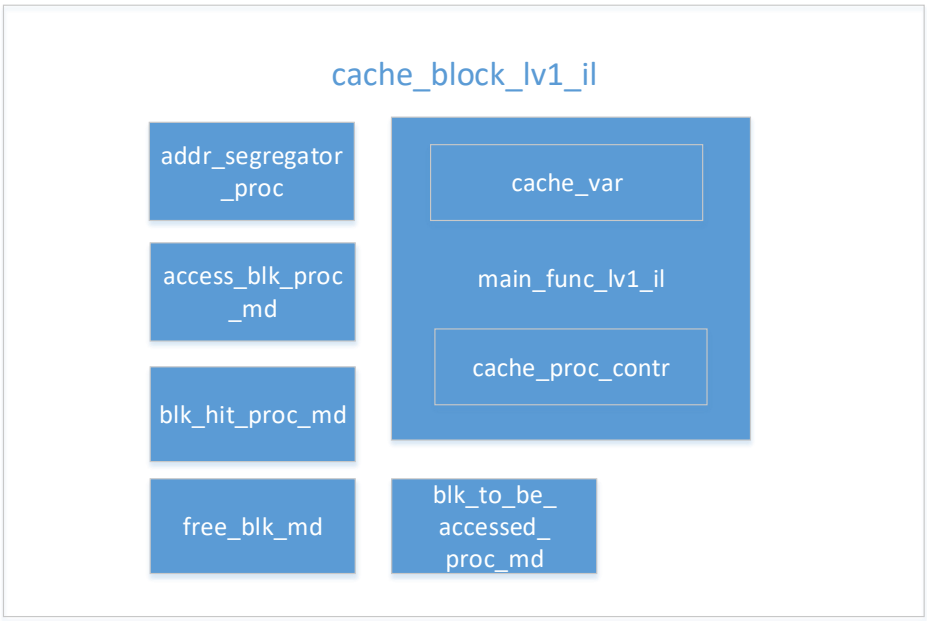

Figure 2-6

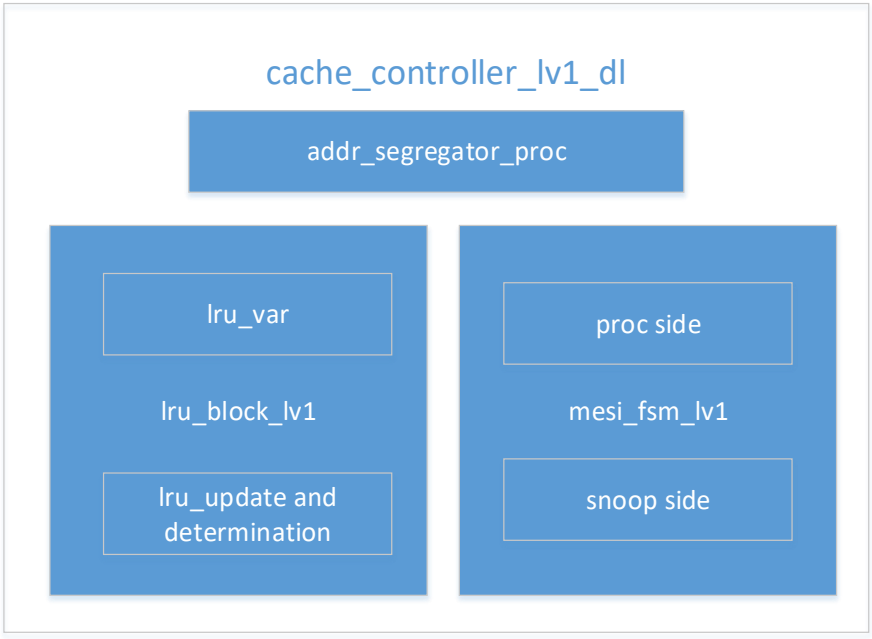

Figure 2-7

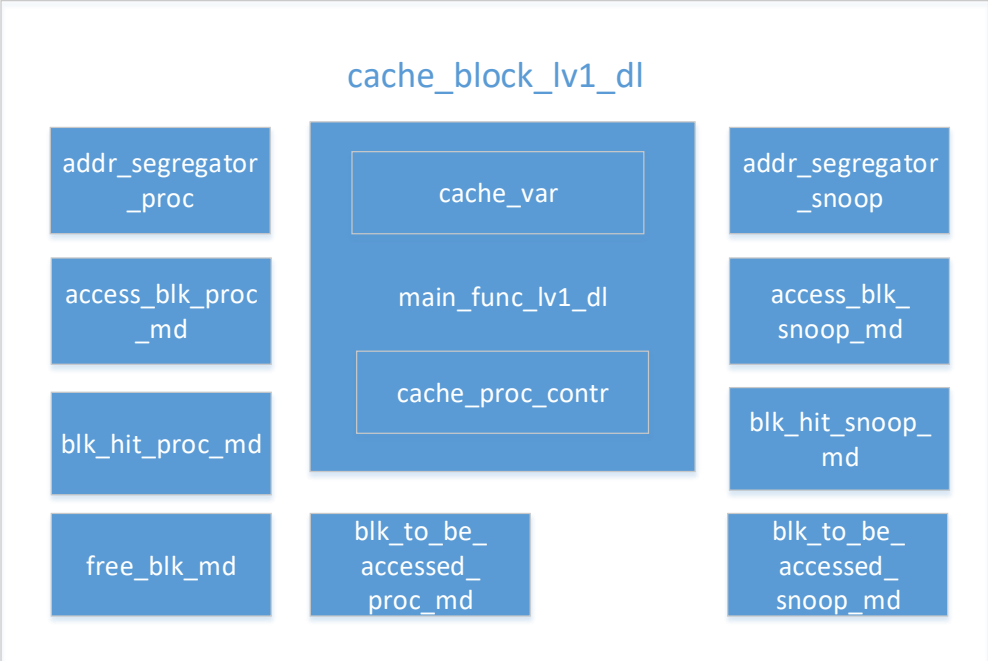

Figure 2-8

The main functional block coordinates the overall functionality of the cache system. It is synchronous to clk signal, i.e. it computes the below mentioned functions only on positive edge of clock.

Before beginning the process the unit deasserts all the signals / flags.

1. Processor Read Hit in L1 Cache

If the block is hit in the cache and its being only read then

- 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

2. Processor Read Miss

If the Block is not hit then the following possibilities can occur

2.1 Free block available in the set

- Bus access is been requested (bus_lv1_lv2_req_proc made high)
- Wait till Access is granted (bus_lv1_lv2_gnt_proc to be made high by arbiter)
- Once access granted, bus_rd and lv2_rd is raised
- Address of the requested data block is put in addr_bus_lv1_lv2

- Wait till level 2 cache / other cache provides the data; communicated by making data_in_bus_lv1_lv2 high. Level 2 cache / other caches put the data in data_bus_lv1_lv2.
- Once data_in_bus_lv1_lv2 becomes high, cache_var [index_proc, blk_access_proc] is updated with value from data_bus_lv1_lv2; cache_proc_contr [index_proc, blk_access_proc] [MESI_range] is updated with updated_mesi_state from CC. Cache_proc_contr [index_proc, blk_access_proc] [Tag_range] is updated with tag_proc.

After this operation, the block will automatically become block hit and previous mentioned Read Hit operation is carried out.

Snoop Side

On the snoop side of the caches with copies of the above requested block, following operations take place (The block is hit in other snooping cache).

Other caches with copies understand that some cache is requesting for the copy with no intention to modify by looking at bus-rd signal (which was made high by above mentioned proc side process)

- cp_in_cache is asserted, telling level 2 to abort bus request.
- The bus access is requested at snoop side (bus_lv1_lv2_req_snoop)
- If already data_in_bus_lv1_lv2 is asserted and own bus request is not granted, then bus_lv1_lv2_req_snoop is deasserted immediately

a) If the copy is in Shared/Exclusive state

- Shared signal is made high
- Data is put in data_bus_lv1_lv2
- data_in_bus_lv1_lv2 is made high indicating data in bus
- Cache_proc_contr [index_proc, blk_access_snoop] [MESI_range] is changed to updated_mesi_state_snoop (Shared state)
- bus_lv1_lv2_req_snoop is deasserted

b)If the copy is in Modified state

- data_bus_lv1_lv2 is loaded with Data from Modified copy
- lv2_wr is asserted to make level 2 cache update its value
- Wait for lv2_wr_done.
- Shared_local is made high. Shared signal will be generated in the bus based on shared_local.

- data_in_bus_lv1_lv2 is made high.
- Cache_proc_contr [index_proc, blk_access_snoop] [MESI_range] is changed to updated_mesi_state_snoop
- bus_lv1_lv2_req_snoop is deasserted

If no other level 1 cache has a copy then level 2 cache provides the data. After these operations, the block will automatically become block hit and Processor Read Hit operation is carried out.

2.2 Processor Read Miss with no free block, replacement needed.

a) If the replacement to be done is Shared/Exclusive then the cache_proc [Index, blk_access_proc] [MESI_range] is made Invalid.

b) If the replacement is in Modified state then the following is carried out.

- bus_lv1_lv2_req_proc is already high (refer aforementioned steps)
- When bus_lv1_lv2_gnt_proc is made high, address of the replacement block is regenerated from its TAG (stored in cache_proc_contr) and Index_proc value and is put in addr_bus_lv1_lv2.
- data_bus_lv1_lv2 is loaded with the dirty data.
- lv2_wr is made high asking level 2 cache to update its value.
- Once lv2_wr_done is made high by level 2 cache, cache_proc [index_proc, blk_access_proc] [MESI_range] is made Invalid.

These set of operations will free that block and which in turn will trigger the free block operations explained above. Then once the above free block operation bring the data from level 2 cache or other level 1 caches it is seen as hit and Processor Read Hit operation is carried out.

3 Processor Write request

- bus_lv1_lv2_req_proc is made high.

3.1 Processor Write Hit in L1 Cache with Shared / Exclusive / Modified state

When the block is hit, then following operations are carried out based on the block's MESI state:

If the cache data block is in Modified / Exclusive state then the following is carried out

- The cache_var is updated with data_bus_cpu_lv1 value.

- cache_proc_contr [index_proc, blk_access_proc][MESI_range] value is updated with updated_mesi_proc value.
- cpu_wr_done is raised.
- bus_lv1_lv2_req_proc is made low

If the Cache data is in Shared condition then the following is carried out

- Once bus_lv1_lv2_gnt_proc is high;
- Address of the block to be invalidated is regenerated from its TAG (stored in cache_proc_contr) and index_proc value and is put in addr_bus_lv1_lv2 bus.
- Signal "invalidate" is made high asking other level 1 caches to make their copy, if present, to be Invalid.
- When all such copies are invalidated, all_invalidation_done is made high.
- The cache_var is then updated with data_bus_lv1_lv2 value.
- cache_proc_contr [index_proc, blk_access_proc][MESI] value is updated with updated_mesi_proc value.
- cpu_wr_done is raised.
- bus_lv1_lv2_req_proc is deasserted

Snoop Side for invalidation request (when block is hit at snoop and invalidate is high)

- cache_proc_contr [index_snoop, blk_access_snoop][MESI_range] made Invalid.
- Signal invalidation_done is made high.
- Signal shared_local is also made high to indicate which all block had the copy in Shared state.

3.2 Processor Write miss

Similar to Read Miss, Write Miss also has two possibilities which are free block/line available and free block/line not available.

(1) Free block available

The following operations are carried out at Proc side of the requesting cache.

If a free line is available. (Processor Write miss in L1 Cache with Shared / Exclusive

/ Modified state)

- bus_lv1_lv2_req_proc is raised.
- Wait till bus_lv1_lv2_gnt_proc to be made high by arbiter.
- Once access granted, bus_rdx and lv2_rd is raised
- Address of the requested data block is put in addr_bus_lv1_lv2.
- Wait till level 2 cache provides the data. Communicated by making data_in_bus_lv1_lv2 high. ATTENTION: data will only be provided by level 2 cache in this case because other level 1 caches will first make their copies Invalid.
- Once data_in_bus_lv1_lv2 becomes high, cache_var [index_proc, blk_access_proc] is updated with value from data_bus_lv1_lv2. cache_proc_contr [index_proc, blk_access_proc] [MESI_range] is updated with updated_mesi_proc. cache_proc_contr [index_proc, blk_access_proc] [Tag_range] is updated with tag_proc.

After this operation, the block will automatically become block hit and previously mentioned Processor Write Hit operation is carried out.

Snoop Side

On the snoop side of the level 1 caches with copies following happens (The block is hit in other snooping cache)

- cp_in_cache is raised.

a) If Copy is in Shared State

- Signal shared_local is made high.
- cache_proc_contr [index_snoop, blk_access_snoop] [MESI_range] value is updated

b) If Copy is in Exclusive state

- cache_proc_contr [index_snoop, blk_access_snoop] [MESI_range] value is updated

c) If Copy is in Modified state

- bus_lv1_lv2_req_snoop is raised to ask for access to the bus.
- Wait for bus_lv1_lv2_gnt_snoop to be high.
- data_bus_lv1_lv2 is loaded with data from Modified copy.
- Signal lv2_wr is asserted to make level 2 cache update its value.

- Wait for lv2_wr_done to be high.
- cache_proc_contr [index_snoop, blk_access_snoop] [MESI_range] value is updated
- bus_lv1_lv2_req_snoop is deasserted.

Only level 2 cache provides the data. It informs the requestor by making data_in_bus_lv1_lv2 high. After this operation, the block will become block hit and previous operation is carried out.

(2) Free block not available, replacement needed.

If the replacement to be done is Shared/Exclusive then the cache_proc [Index_proc, blk_access_proc][MESI_range] is made Invalid.

If the replacement is in Modified state then the following is carried out (as dirty replacement needs to be updated in level 2 cache).

- When bus_lv1_lv2_gnt_proc is made high, address of the replacement block is regenerated from its TAG (stored in cache_proc_contr) and Index_proc value and is put in addr_bus_lv1_lv2.
- data_bus_lv1_lv2 is loaded with the dirty data.
- lv2_wr is made high asking level 2 cache to update its value.
- Once lv2_wr_done is made high by level 2 cache, cache_proc [index_proc, blk_access_proc] [MESI_range] is made Invalid.

2.3. Timing Specifications

Note: “Arbiter delay” in below diagrams is not drawn to scale, namely not necessarily 2 cycles. The number of cycles after which the arbiter will grant the processor depends how many concurrent requests are present and their priority.

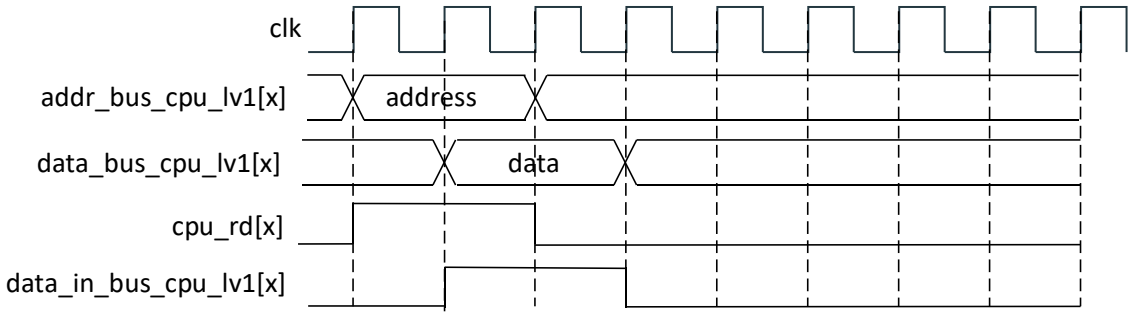

Figure 2.9: read level 1 cache hit

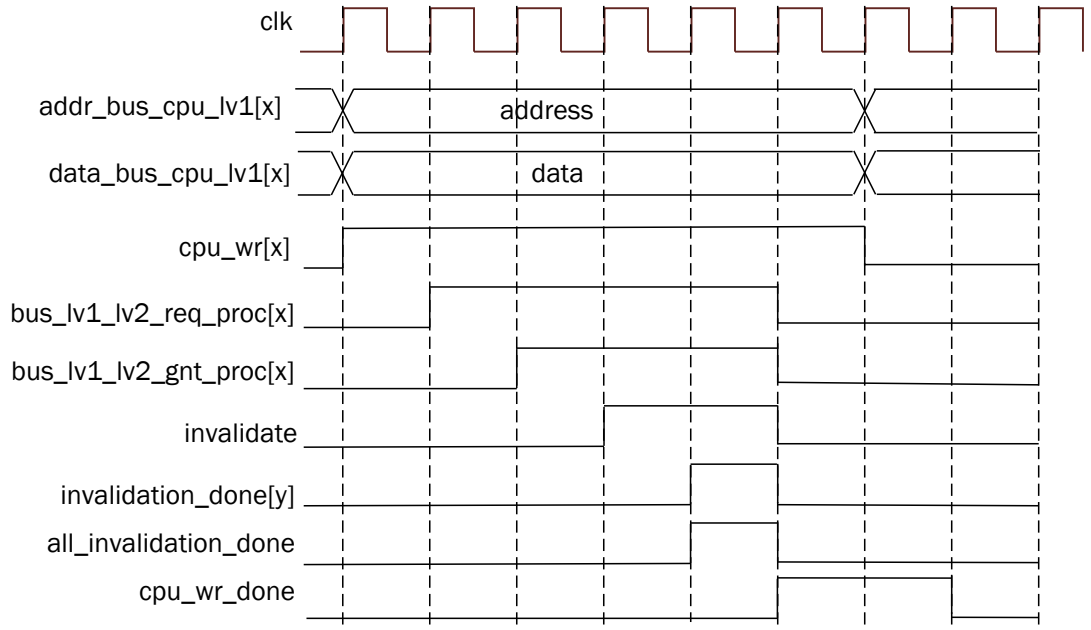

Figure 2.10: write level 1 cache hit

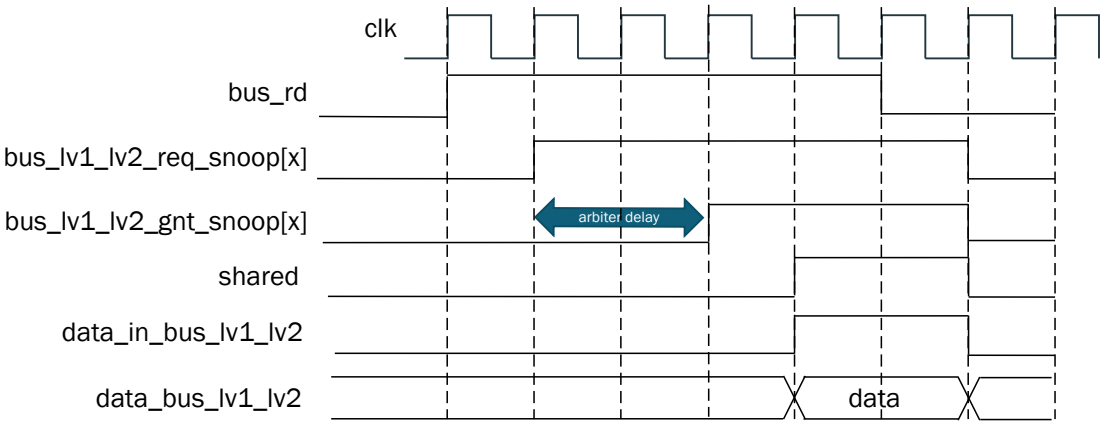

Figure 2.11: bus read, snoop side has copy in Shared/Exclusive state.

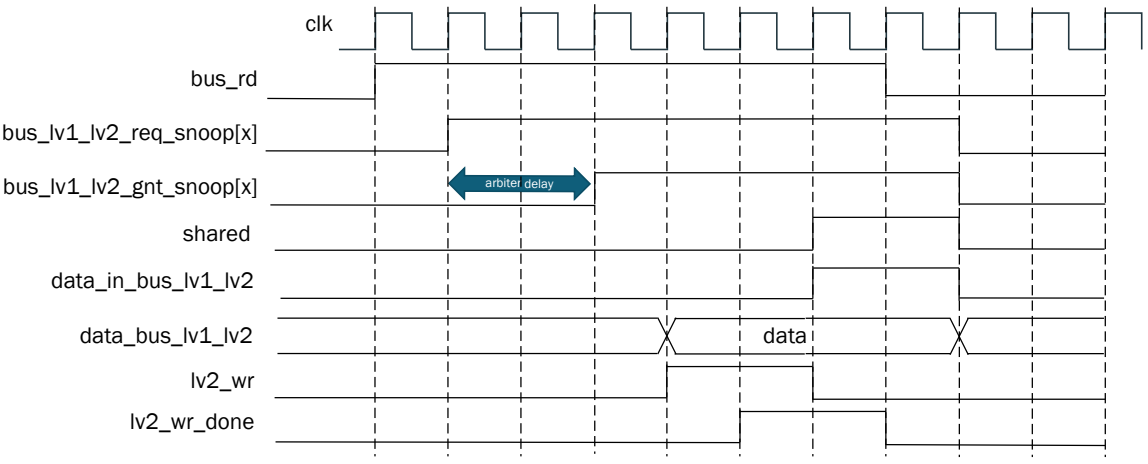

Figure 2.12: bus read, snoop side has copy in Modified state

3 Level 2 Cache Design

3.1. General Description

The level 2 cache is the second level data storage shared by all 4 level 1 caches and interacts directly with the memory. It is a unified cache which does not distinguish instructions from data. The size of level 2 cache is 8 MB. It is an 8-way set associative cache and implements pseudo-LRU replacement policy.

3.2. Algorithm Description

Pseudo-LRU Algorithm

The cache_controller_lv2 will also follow a pseudo-LRU algorithm. Although it is an 8-way associativity scheme instead of the 4-way in level 1 cache, the main idea is the same. The LRU state - replacement relation and next state transition are shown in table 3-1.

Table 1-1

state	replacement
00x0xxx	line 0
00x1xxx	line 1
01xx0xx	line 2
01xx1xx	line 3
1x0xx0x	line 4
1x0xx1x	line 5
1x1xxx0	line 6
1x1xxx1	line 7

refer to	next state
line 0	11-1---
line 1	11-0---
line 2	10--1--
line 3	10--0--
line 4	0-1--1-
line 5	0-1--0-
line 6	0-0---1
line 7	0-0---0

3.3. Structures and Implementation

The top level module of level 2 cache is cache_wrapper_lv2. It is comprised of two blocks: cache_controller_lv2 and cache_block_lv2. The cache_controller_lv2 has a similar structure to cache_controller_lv1_il. There is only one LRU block and no MESI block. The cache_block_lv2 is also similar to cache_block_lv1_il because there is no snoop side or MESI protocol needed. The main functional difference is that cache_block_lv2 allows write operation but cache_block_lv1_il does not. The structure of cache_block_lv2 is shown in Figure xxx.

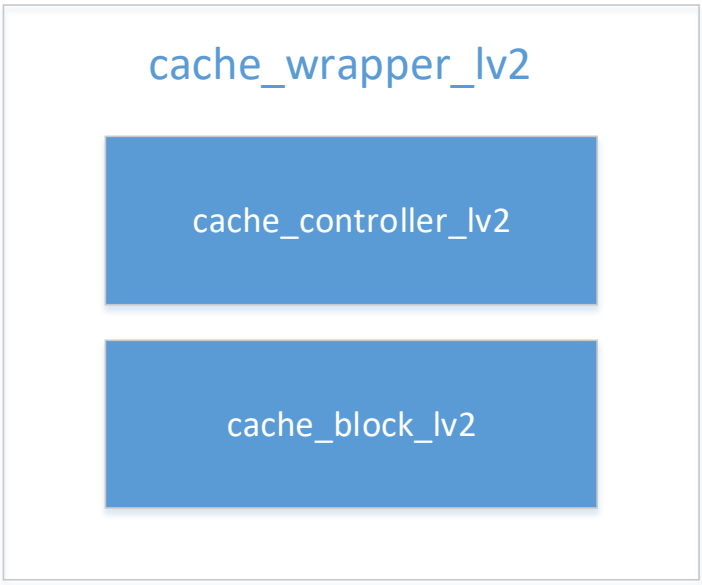

Figure 3-1

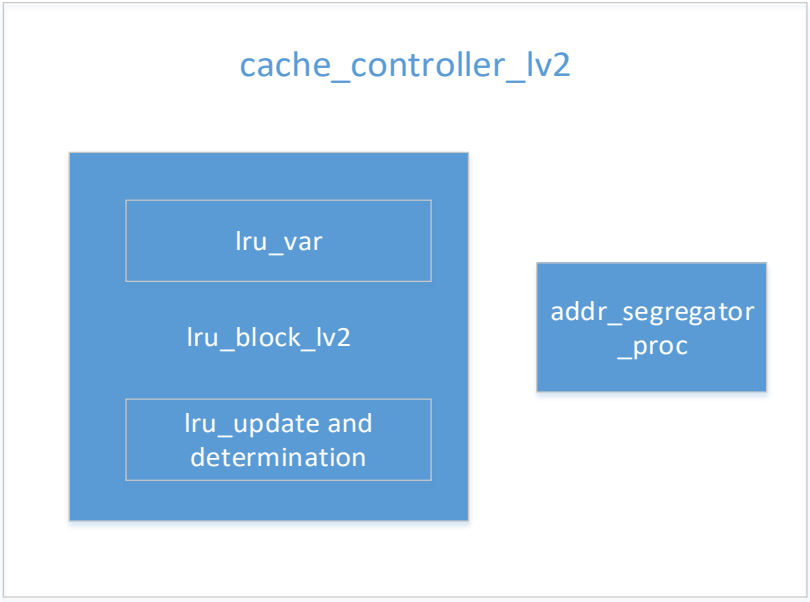

Figure 3-2

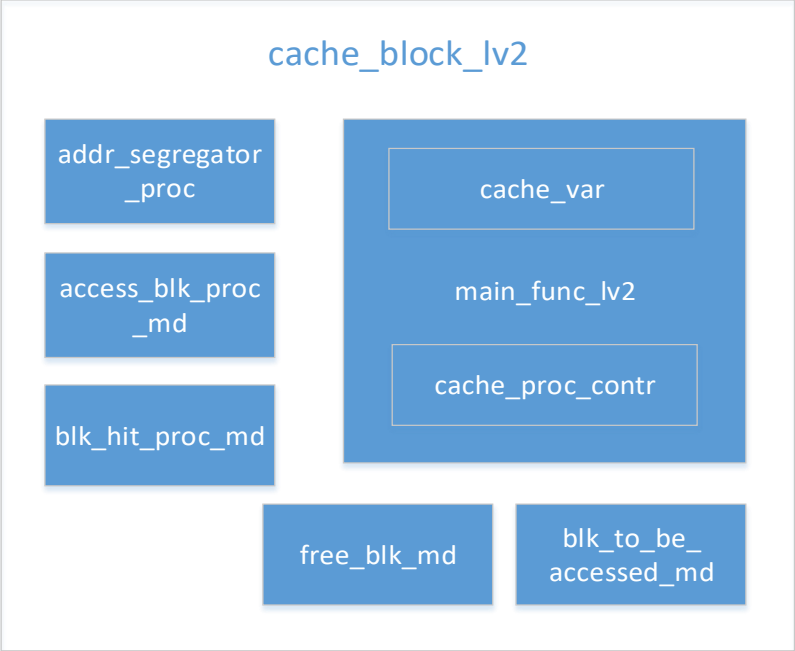

Figure 3-3

Here we only introduce the behavior of the main_func_lv2 block because all other blocks have the same functions as their counterparts in level 1 instruction cache.

(1) Read operation

(a) Read hit

- level 1 raises lv2_rd and loads address in addr_bus_lv1_lv2.
- bus_lv1_lv2_req_lv2 is asserted.
- If bus_lv1_lv2_gnt_lv2 is raised, after certain delay the data is loaded in data_bus_lv1_lv2, data_in_bus_lv1_lv2 is asserted.
- wait lv2_rd to drop, then deassert data_in_bus_lv1_lv2.

(b) Read miss no block replacement

- level 1 raises lv2_rd and loads address in addr_bus_lv1_lv2.
- mem_rd is asserted. Address is loaded in addr_bus_lv2_mem.
- After certain delay, data_in_bus_lv2_mem is asserted. Data is loaded in data_bus_lv2_mem
- mem_rd is deasserted, go to step 2 of (a) and go on.

(c) Read miss with block replacement

- level 1 raises lv2_rd and loads address in addr_bus_lv1_lv2.

- mem_wr is asserted. Address of block to be replaced is loaded in addr_bus_lv2_mem. Its data is loaded in data_bus_lv2_mem.
- wait mem_wr_done to rise, then mem_wr is deasserted.
- go to step 2 of (b) and go on.

In read operation, if cp_in_cache signal is high, then bus_lv1_lv2_req_lv2 will be dropped and no further actions will be taken until cp_in_cache is removed. If cp_in_cache is low, the level 2 cache will follow the above precedures.

(2) Write operation

(d) Write hit

- level 1 raises lv2_wr, loads address in addr_bus_lv1_lv2 and loads data in addr_bus_lv1_lv2
- After certain delay, lv2_wr_done is asserted.
- If lv2_wr is deasserted, lv2_wr_done is deasserted.

(e) Write miss no block replacement

- level 1 raises lv2_wr, loads address in addr_bus_lv1_lv2 and loads data in addr_bus_lv1_lv2
- mem_rd is asserted. Address is loaded in addr_bus_lv2_mem.
- After certain delay, data_in_bus_lv2_mem is asserted. Data is loaded in data_bus_lv2_mem
- mem_rd is deasserted, go to step 2 of (d) and go on.

(f) Write miss with block replacement

- level 1 raises lv2_wr, loads address in addr_bus_lv1_lv2 and loads data in addr_bus_lv1_lv2
- mem_wr is asserted. Address of block to be replaced is loaded in addr_bus_lv2_mem. Its data is loaded in data_bus_lv2_mem.
- wait mem_wr_done to rise, then mem_wr is deasserted.
- go to step 2 of (e) and go on.

3.4. Timing Specifications

Note: “Arbiter delay” and “memory delay” in below diagrams are not drawn to scale. The number of cycles after which the arbiter will grant the processor depends how many concurrent requests are present and their priority. The memory delay is not specified and depends on your implementation.

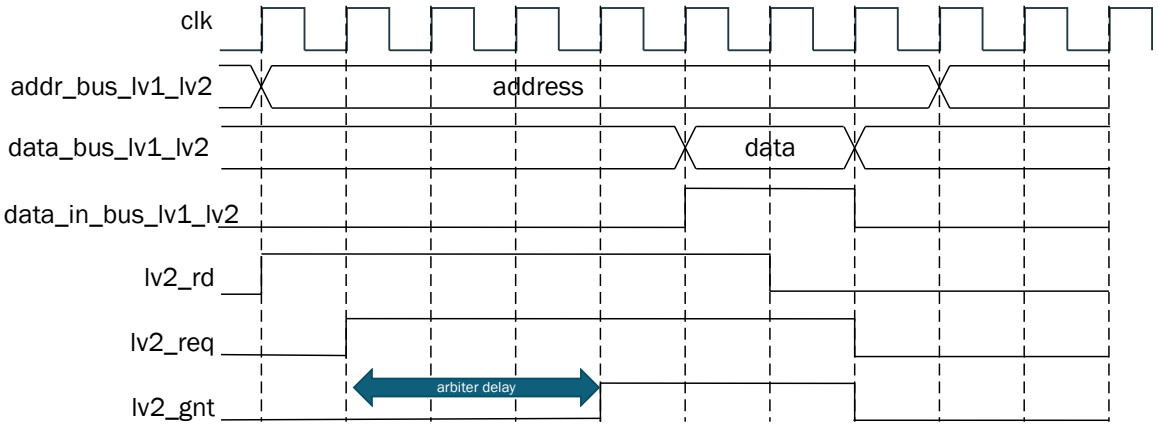

Figure 3-4: read level 2 cache hit

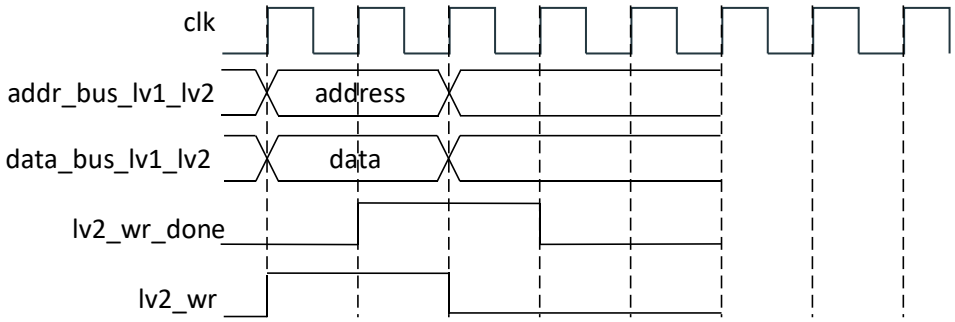

Figure 3-5: write level 2 cache hit

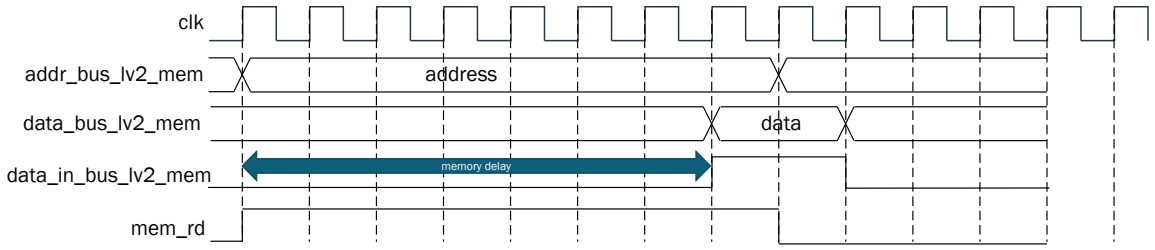

Figure 3-6: read from memory

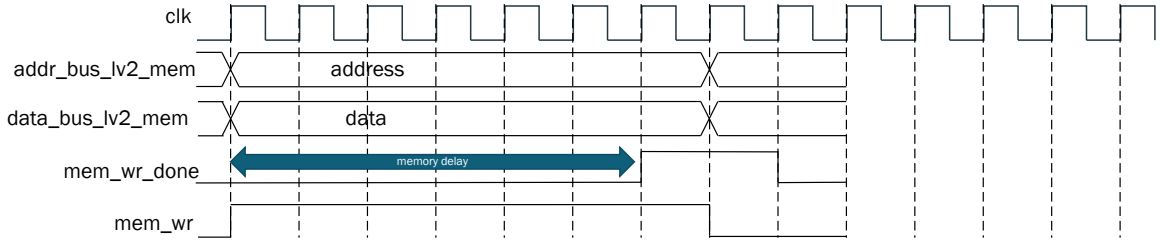

Figure 3-7: write to memory

4 APPENDIX: ARBITER SCHEME

4.1 General Description

An arbiter is a device used in a system to decide, for each cycle, which CPU will be allowed to access that bus. Given only one request, an arbiter promptly permits the corresponding action, delaying any second request until the first action is completed. When an arbiter gets two requests or more at once, it must decide which request to grant first.

NOTE: Arbiter design is not a part of cache design but the verification environment. We provided a sample arbiter with "least recently served" round robin scheme, which is explained in the following sections. You are allowed to use your own arbiter design as long as it satisfies the cache timing specifications. Since complex arbiter scheme improves fairness but could make debugging hard, you may want to choose different schemes for different tests or even manually give grant signals.

4.2 Algorithm Description

The communication is on the 2 side: processor side and snoop side.

4.2.1 Processor Side

Case 1:

One request is raised: When only one request is raised, it will be served on the following clock edge and priority queue will be updated. See Figure 22.

Case 2:

More than one request are raised: The requests will be observed. Least recently served process will be granted the access first. Accordingly, Pointer denoting the head will be moved to the next element in the queue and tail pointer will now point to the number of processor whose request is served. See Figure 23.

Assumption: Queue is already filled in the order. Depends on the requests raised, Least recently served request will be served.

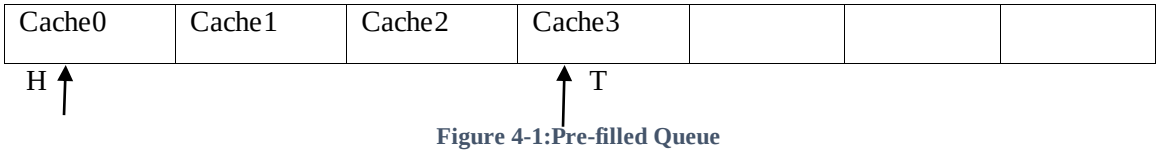

Figure 4-1:Pre-filled Queue

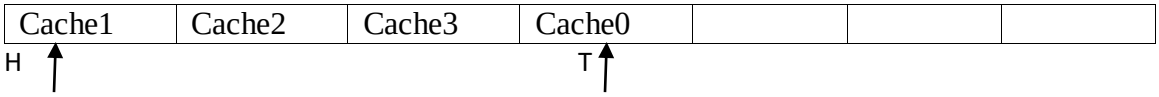

Figure 4-2: Request raised by Cache0

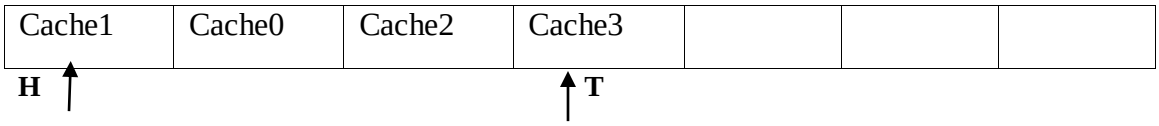

Figure 4-3: Request raised by Cache2 and Cache3

4.2.2 Snoop Side

On the snoop side, access will be granted only if at least 1 processor side request is granted. No snoop side request is granted, if no processor side request is granted beforehand. At a time only one request will be granted based on a fixed priority: cache0 > cache1 > cache2 > cache3 > level 2.

4.3 Structures and Implementation

In the structure of arbiter, top module is arbiter logic, which will control the assertion and de-assertion of grant signals. It contains the sub-modules as FIFO, Processor Side Logic and Snoop Side Logic. FIFO is nothing but the queue which is used for Processor side logic as mentioned in above section 4.2. This structure is shown in Figure X given below:

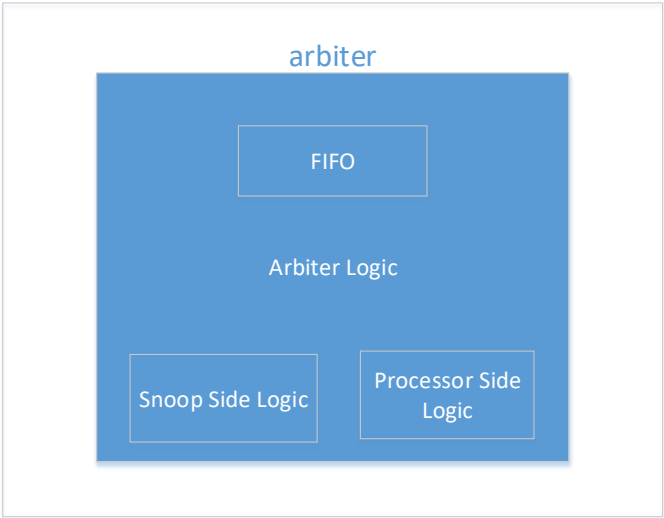

Figure 4-4

4.4 Timing Specifications

simultaneous proc requests and snoop requests,
grant proc0 first, snoop hit in cache2, then grant proc1

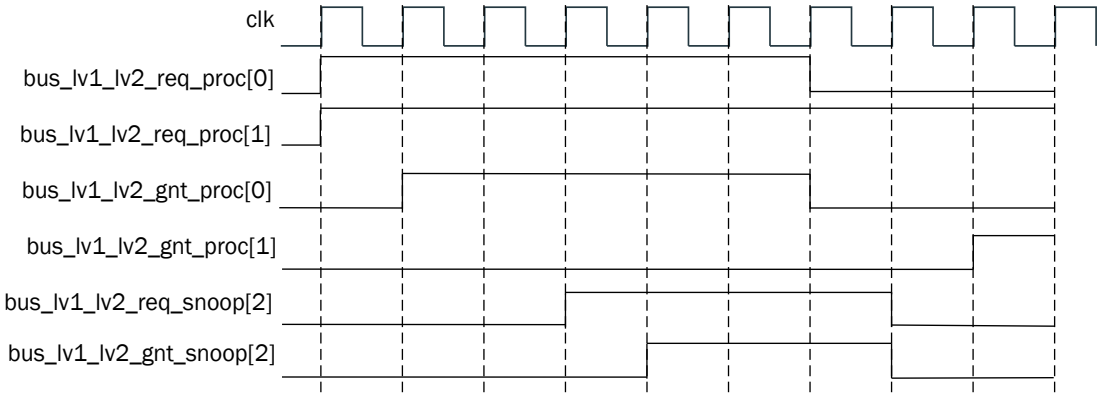

Figure 4-5