Portland State University

ECE 485/585 - Microprocessor System Design
Winter 2021

Professor: Yuchen Huang

Final Project Report

Natalie Nguyen , Zack Fravel , Megha Jacob , Karla Barraza Lopez

Introduction

Our final project involved the design and implementation of a 32-bit split L1 Cache. The cache consists of an 8-way set associative data cache, a 4-way set associative instruction cache and implements LRU (least recently used) organization policy with a write-back policy with MESI coherence. We implemented the cache in SystemVerilog as described, along with error handling for filtering out invalid processor requests. The following sections describe our design overview, assumptions, details about our LRU caching policy, state change mechanism, file parsing implementation, and verification strategy.

The input to our cache takes the format of:

Command [Integer]   32-bit Address [Hex]

Commands have the following values and represent these functions:

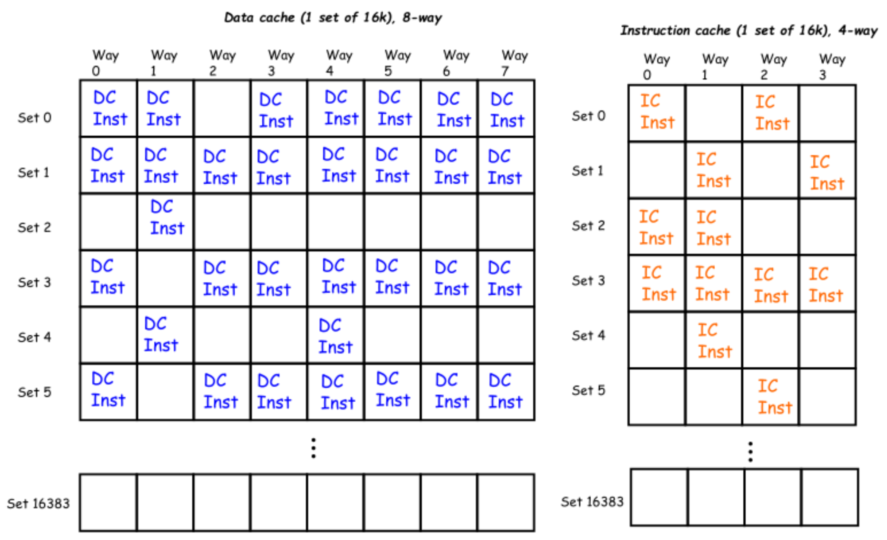
- 0 - Read data request to L1 data cache
- 1 - Write data request to L1 data cache
- 2 - Read request to L1 instruction cache
- 3 - Invalidate command from L2
- 4 - Data request from L2 (in response to snoop)
- 8 - clear the cache and reset all states and statistics
- 9 - print contents and state of cache but allow subsequent activity

When a '9' command is executed, our cache outputs the contents along with combined statistics for the data and instruction cache:

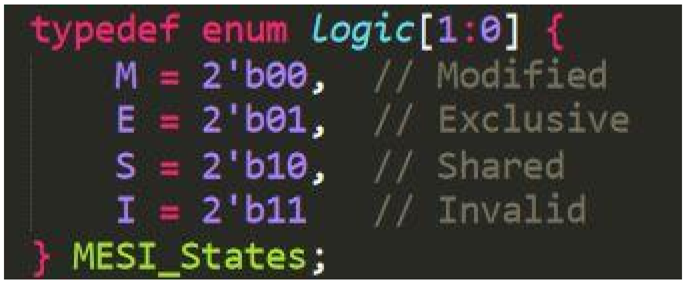
- Number of cache reads
- Number of cache writes
- Number of cache hits
- Number of cache misses
- Cache hit ratio

Summary of Split L1 Cache Design

Since this project is primarily focused on the functionality of our MESI coherence state machine and the cache doesn't have to store data, that allowed us to design and implement our caches as simple 2D arrays as seen on the right. To implement these arrays, we designed a typedef struct that packed all the bits we'd need for

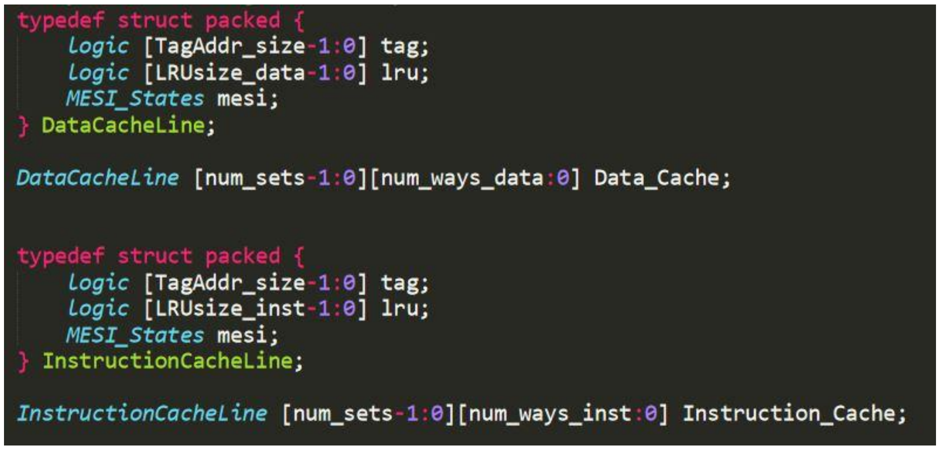


each cache line. Each cache line contains the tag bits for identification, LRU bits, and MESI coherence bits. The picture below shows how we implemented our cache in SystemVerilog.



```
typedef enum Logic[1:0] {
    M = 2'b00,  // Modified
    E = 2'b01,  // Exclusive
    S = 2'b10,  // Shared
    I = 2'b11   // Invalid
} MESI_States;
```

Figure 1: MESI Typedef Enum



```
typedef struct packed {
    Logic [TagAddr_size-1:0] tag;
    Logic [LRUsize_data-1:0] lru;
    MESI_States mesi;
} DataCacheLine;

DataCacheLine [num_sets-1:0][num_ways_data:0] Data_Cache;


typedef struct packed {
    Logic [TagAddr_size-1:0] tag;
    Logic [LRUsize_inst-1:0] lru;
    MESI_States mesi;
} InstructionCacheLine;

InstructionCacheLine [num_sets-1:0][num_ways_inst:0] Instruction_Cache;
```

Figure 2: Cache Line Typedef Struct with Cache Instantiation

Once our structs were defined correctly, all we had to do was instantiate an instance of each as 2D arrays with the number of sets and ways determining our 'rows' and 'columns.' To access one of these entries, you would access it through standard array indexing: [set][way]. For example, accessing way 3 of set 12, is done via cacheName[12][3]. This allowed us to much more easily keep track of what bits were being manipulated in our code and function calls.

Input File Parsing Implementation

Before getting deep into the L1 cache design, we began by implementing and verifying a module that can take our simulus files (.trace) in the format specified by the project description and output the correct mode, command, and address to our cache module. The design is implemented as a single initial block to ensure the process executes sequentially along with a reset task that sends an '8' command to the cache. The implementation takes advantage of the SystemVerilog system functions $valueplusargs, $fopen, $fscanf, and $fclose to read in file names, the desired mode from the command line, and split the strings into different variables to be outputted. Once this was working properly and we decided exactly how our inputs would be coming into the cache, we moved onto digging deep into the actual L1 cache design (detailed in following sections).

Assumptions

These are the assumptions that were made for our split L1 cache design:

- We are not dealing with data and spatial locality. This allowed us to collapse our cache design into two dimensions and simplify our handling of the cache.
- HIT and HITM are always zero. This is because there is no snooping mechanism used in the design since we're only designing one cache.
- When all of the ways on the cache line are not completely filled, if reading data from the cache for the first time and it is a cache miss, the state changes from Invalidate to Exclusive. This is because the states in the cache are initialized with Invalidate.
- When there is a first cache hit from a read operation, the state changes from Exclusive to Shared because we're assuming that the read operation is done by another processor.
- Counter method for LRU uses counting up. For instance, 000 indicates the most recently used or accessed way on the cache line. On the other hand, 111 indicates the least recently used or accessed way on the cache line.
- When the cache line is filled and there is a cache miss, if there is one way that has an Invalidate state on the cache line, it's evicted. If there are more than two ways that have an Invalidate state, we evict the one that has the largest LRU bits out of the invalidated ways. If there are no Invalidate states on the cache line, the way with LRU bits 111 is evicted.
- When n is 3 or 4, it does not affect the number of cache hits and misses.

LRU Counter-Based Method

When there is a cache miss and not every way on the cache line is filled, the tag can be stored in the next empty way on the cache line. The LRU bits assigned for this way is 000 since it is most recently used. Then, the LRU bits for other ways are incremented by one. For instance, if way-1 and way-2 are filled with a tag and there is a cache miss on that cache line, then the new tag is stored in the next empty way which is way-3. The LRU bits for way-3 are 000 and the LRU bits for way-1 and way-2 each get incremented by one.

In other cases, when a way on the cache line is most recently used, the LRU bits for this way becomes 000. If any of the other ways have their LRU bits smaller than the previous LRU bits of the most recently used way, their LRU bits are incremented by one. The LRU bits of the rest stay the same. For instance, if way-1 is most recently used and its current LRU bits are 100. Then, we check the LRU bits of way-2 to way-8. If any of them has their LRU bits smaller than 100 (such as 000, 001, 010, 011), then their LRU bits are incremented by one. Also, LRU bits that are larger than 100 (such as 101, 110, 111) remain the same. Finally, we change the LRU bits of way-1 to 000.

State Change Mechanism (MESI Coherence Implementation)

The state change mechanism is described via the following diagram and also explained in more detail in the MESI state change table as seen in the appendix. The state change is based on two inputs - the command and the hit/miss. The state machine is implemented in SystemVerilog using case statements in a function. First, the input command is checked, followed by a check for hit/miss. Then the next state for instruction cache and data cache is computed based on the present state.

During a line eviction or invalidation, the current state of the line is checked. If the line to be evicted or invalidated is in Modified state, the modified data is written back to L2 before being evicted or invalidated from L1 data cache. Since the instruction cache is read only, it cannot have a Modified state. Instead of transferring actual data between L1 and L2 caches, we displayed messages for writing/reading data to/from L2.

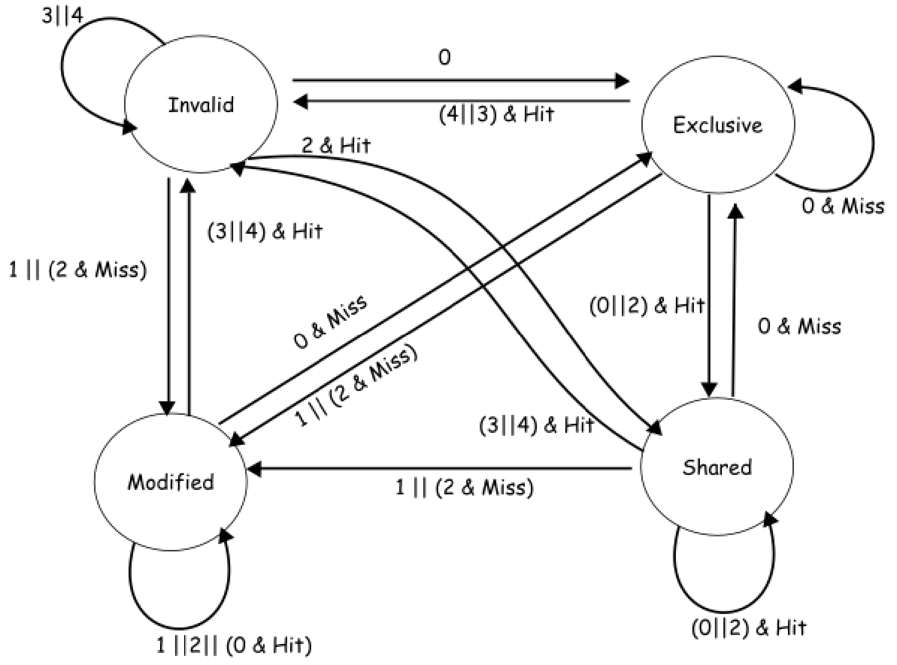
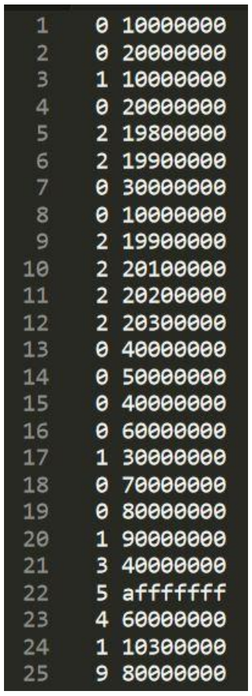


Figure 3: State Diagram for MESI Coherence FSM

Test Plan

Each team member performed unit testing on the modules they were responsible for writing RTL for, along with contributing by creating .trace files to be run for our system tests when checking full behavior of our design. Once our RTL was mostly complete and we had completed our preliminary unit testing, (i.e. file parser, cache, defs, and top level testbench) we were able to test the whole system by running very simple .trace files for things like reset behavior, LRU eviction, MESI state changes, and move onto more complex test after those worked.

On the right, we have included an example of one of our test .trace files (set0.trace) and below we show the expected results we verified (by hand) that we could compare with our outputs to ensure our design was working as we expect.



```
0 10000000
0 20000000
1 10000000
0 20000000
2 19800000
2 19900000
0 30000000
0 10000000
2 19900000
2 20100000
2 20200000
2 20300000
0 40000000
0 50000000
0 40000000
0 60000000
1 30000000
0 70000000
0 80000000
1 90000000
3 40000000
5 afffffff
4 60000000
1 10300000
9 80000000
```

This test contains all typical behaviors that will be found in normal operation (reads, writes, sharing, instruction fetches, L2 requests, and printing).

Expected outputs for split L1 data cache from the test case described:

		Data Cache							
	Set 0	**way 0**	**way 1**	**way 2**	**way 3**	**way 4**	**way 5**	**way 6**	**way 7**
	Tag	100							
0 10000000	**LRU bits**	000							
	MESI	E							
	Tag	100	200						
0 20000000	**LRU bits**	001	000						
	MESI	E	E						
	Tag	100	200						
1 10000000	**LRU bits**	000	001						
	MESI	M	E						
	Tag	100	200						
0 20000000	**LRU bits**	001	000						
	MESI	M	S						
	Tag	100	200	300					
0 30000000	**LRU bits**	010	001	000					
	MESI	M	S	E					
	Tag	100	200	300					
0 10000000	**LRU bits**	000	010	001					
	MESI	M	S	E					
	Tag	100	200	300	400				
0 40000000	**LRU bits**	001	011	010	000				
	MESI	M	S	E	E				
	Tag	100	200	300	400	500			
0 50000000	**LRU bits**	010	100	011	001	000			
	MESI	M	S	E	E	E			
	Tag	100	200	300	400	500			
0 40000000	**LRU bits**	010	100	011	000	001			
	MESI	M	S	E	S	E			

	Tag	100	200	300	400	500	600		
0 60000000	**LRU bits**	011	101	100	001	010	000		
	MESI	M	S	E	S	E	E		
	Tag	100	200	300	400	500	600		
1 30000000	**LRU bits**	100	101	000	010	011	001		
	MESI	M	S	M	S	E	E		
	Tag	100	200	300	400	500	600	700	
0 70000000	**LRU bits**	101	110	001	011	100	010	000	
	MESI	M	S	M	S	E	E	E	
	Tag	100	200	300	400	500	600	700	800
0 80000000	**LRU bits**	110	111	010	100	101	011	001	000
	MESI	M	S	M	S	E	E	E	E
	Tag	100	900	300	400	500	600	700	800
1 90000000	**LRU bits**	111	000	011	101	110	100	010	001
	MESI	M	M	M	S	E	E	E	E
	Tag	100	900	300	400	500	600	700	800
3 40000000	**LRU bits**	111	001	100	000	110	101	011	010
	MESI	M	M	M	I	E	E	E	E
	Tag	100	900	300	400	500	600	700	800
4 60000000	**LRU bits**	111	010	101	001	110	000	100	011
	MESI	M	M	M	I	E	I	E	E
	Tag	100	900	300	103	500	600	700	800
1 10300000	**LRU bits**	111	010	101	000	110	001	100	011
	MESI	M	M	M	M	E	I	E	E

Expected outputs for split L1 instruction cache from the above test case:

	L1 Instruction Cache				
	Set 0	**way 0**	**way 1**	**way 2**	**way 3**
	Tag	198			
2 19800000	**LRU bits**	00			
	MESI	E			
	Tag	198	199		
2 19900000	**LRU bits**	01	00		
	MESI	E	E		

2 19900000	**Tag**	198	199		
	LRU bits	01	00		
	MESI	E	S		
2 20100000	**Tag**	198	199	201	
	LRU bits	10	01	00	
	MESI	E	S	E	
2 20200000	**Tag**	198	199	201	202
	LRU bits	11	10	01	00
	MESI	E	S	E	E
2 20300000	**Tag**	203	199	201	202
	LRU bits	00	11	10	01
	MESI	E	S	E	E

Code Organization

We wanted to ensure anyone who wanted to recreate our results, or create more tests, could easily do so. Our makefile is designed to be run in a linux environment with Mentor Graphics QuestaSim 2019. The project environment contains two folders, /RTL/ and /STIMULUS/, where our source code and .trace files are kept. Details on how to add new tests and run our included tests are described in the appendix along with some guidance when you type 'make help' in the verification environment.

Challenges and Conclusions

Overall once our individual unit tests were completed we had a mostly working design, but still ran into a few challenges finishing up the implementation that we were able to resolve. First was figuring out the right timing for our testbench (i.e. how often a new trace file is transmitted to the design). We settled on every 2 clock cycles since our next-state logic requires an additional cycle to actually set the values in our MESI state bits. Another challenge we overcame was ensuring our cache could handle / detect two identical instructions back-to-back. Initially we were puzzled on how to do this, but realized it could easily be resolved by adding a trigger output to our file parser that goes high when a new line is read and immediately goes back to 0. We then added a trigger input to the cache and made our state-change always block sensitive to the posedge of that trigger signal.

To conclude, our design split L1 cache with MESI coherence correctly implements the specification laid out in the project description assignment. As a team we were able to work together to agree on our design assumptions, overcome challenges in implementation, and fix bugs through a very quick debugging cycle that only lasted a few days once the design work was finished. This concludes our report on our Split L1 (4-way associative instruction, 8-way associative data) cache design with MESI coherence and write-back policy.

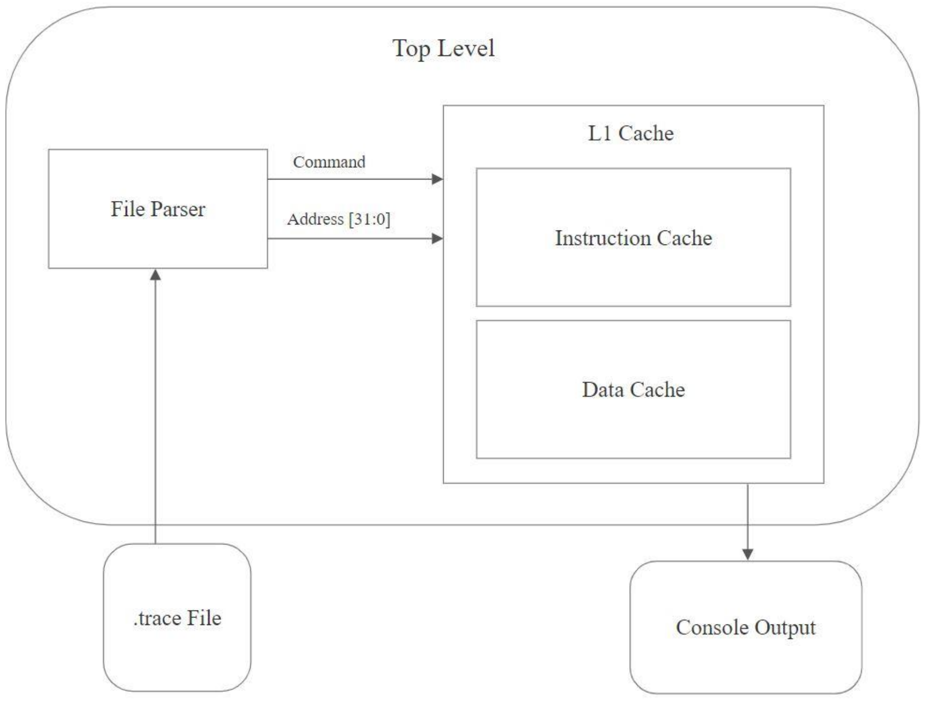
Appendix

I. State Change Table

n	Hit/Miss	PS	NS	Display	Comments
0- Read req to L1 data cache	Hit	**M**	**M**		Ist read -> E state If PS =E, then next read is done by another processor and NS = S Assumes HIT & HITM =0, So when a hit occurs 'I'-state changes to 'E'-state
		E	**S**		
		S	**S**		
		I	**E**		
	Miss	**M**	**E**	Write to L2 <Tag_evicted> Read from L2 <Tag>	
		E	**E**	Read from L2 <Tag>	
		S	**E**	Read from L2 <Tag>	
		I	**E**	Read from L2 <Tag>	
1 – Write req to L1 data cache	Hit	**M**	**M**		First time write is a miss & has a " write through policy" so display 'Write to L2' Other writes are done with "write-back" policy
		E	**M**		
		S	**M**		
		I	**M**	Read for Ownership from L2 <Tag>	
	Miss	**M**	**M**	Write to L2 <Tag> Read for Ownership from L2 <Tag>	
		E	**M**	Read for Ownership from L2 <Tag>	
		S	**M**	Read for Ownership from L2 <Tag>	
		I	**M**	Read for Ownership from L2 <Tag>	
2 – Read req to L1 instr cache	Hit	**M**	**-**		Since Instruction Cache is read only, it cannot have 'M'- state Assumes HIT & HITM =0, So when a hit occurs 'I'-state changes to 'E'-state
		E	**S**		
		S	**S**		
		I	**E**	Read from L2 <Tag>	
	Miss	**M**	**-**		
		E	**E**	Read from L2 <Tag>	
		S	**E**	Read from L2 <Tag>	
		I	**E**	Read from L2 <Tag>	

3 – Invalidate Command frm L2	Hit	**M**	**I**	Return data to L2 <Tag>	
		E	**I**		
		S	**I**		
		I	**I**		
	Miss	**-**	**-**		
4 – Data req frm L2 (RFO)	Hit	**M**	**I**	Return data to L2 <Tag>	
		E	**I**		
		S	**I**		
		I	**I**		
	Miss	**-**	**-**		
8 – Clear cache and reset	-	**M**	**I**	Write to L2 <Tag>	
	-	x	**I**		

II. Block Diagram



III. README

```
----------------------------------------------------------------------------------------------

ECE585 - Microprocessor System Design
Portland State University - Winter 2021

Team 3 (Zack, Megha, Natalie, Karla) - Split L1 MESI Cache Design and Verification

README

----------------------------------------------------------------------------------------------

Our makefile is designed to run on a linux server with mentor-questa-2019 installed.

To ensure mentor-questa-2019 is on your linux machine, run addpkg and make sure
[x] mentor-questa-2019 is checked. Once you have confirmed your simulator is installed,
follow these steps to run our simulations. . .


1. Navigate to 'Source Code' Folder and open a Terminal window.

2. Type 'make help' to see instructions / options for execution and all test names.

3. Type 'make compile' to compile the project.

4. After compilation, the user can run as many tests as desired consecutively
by typing 'make <test_name>_mode<0|1>'.


To create and run custom tests / traces:

        1. Create a set of traces and place them line-by-line in a file
        that has a .trace extension (e.g. example.trace, NOT .txt) and place it in the
        ./STIMULUS/ folder of our verification environment. Make sure the end
        of the file has a '9' command to view the contents and statistics.

        2. After compiling, run the following command for either mode desired:
        Mode 0 - vsim -c +trace="./STIMULUS/example.trace" +mode=0 tb_top -do "run -all;quit"
        Mode 1 - vsim -c +trace="./STIMULUS/example.trace" +mode=1 tb_top -do "run -all;quit"

        3. Alternatively, you can add both of those lines to the Makefile and create two
        targets (example_mode0: and example_mode1:) for more convenient back to back runs.
```