11/3/2024

Multicore RISC-V cache controller

Master thesis specification

Mentors:

Vuk Vranjković

Tivadar Mako

Students:

Ivan Milin E1-79/2023

Petar Stamenković E1-11/2023

Contents

1. System Overview

This document is a specification of a Master thesis project that includes 2 faculty subjects:

1. Advanced microprocessor systems
2. Formal methods of verification and design

It is a cooperation between faculty of technical sciences in Novi Sad and VeriestS.

With first subject we will cover the design of individual RISCV core with its own cache controller and L1 cache alongside with a bus controller that connects global cache controller, L2 cache and main memory with already mentioned cores. We will use Vivado Design Suite and code our design using Verilog HDL.

With second subject we will cover the verification using Jasper Gold formal tool and SystemVerilog language. Our approach will be *bottom-up* as we will try to verify each developed module during design phase. Our idea is to verify each module with 2 different points of view, firstly with a simple testbench during design phase and secondly using formal tool to verify complex scenarios and increase the coverage.

Figure 1 represents the top diagram of our system.

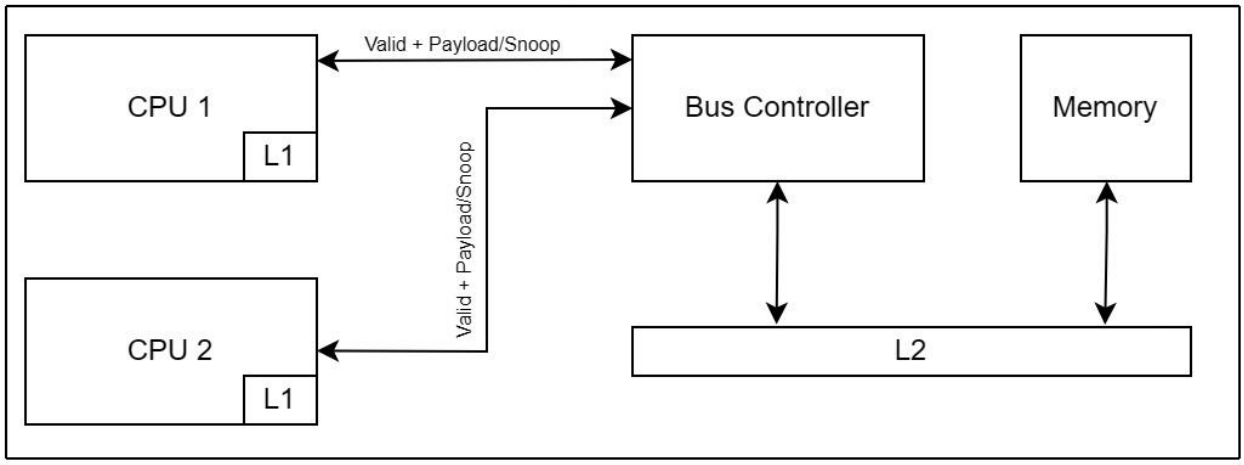

Figure 1 : System Overview

2. Components

As you can see on the figure 1, our system is composed of following components.

1. CPU cores (*CPU1 and CPU2*) with their local cache memories (*L1*).
2. Global bus controller (*Bus controller*)
3. Global memory (*Memory*)
4. L2 cache with its controller (*L2*)

This paragraph will give you a more information about each component.

CPU Cores – Each core is single cycle and has RISCV ISA, local cache memory (L1) and cache controller with **MESI** cache coherence protocol. Our idea is to implement following instruction sets: ***R, I, S, B, U, J, L***. Then we will generate a binary text file from an assembly code using RIPES simulator [1]. We will test our system by loading contents of the binary file into design and viewing the waveforms from Vivado simulator.

Also, each CPU has a feature for accessing data from its local cache memory. If requested data does not exist in local cache memory we flag a *MISS* and send a *SNOOP Request*. When global bus controller receives this request, it has to access other L1 caches and look for a requested data there. It this also fails, global bus controller has to look for request data in L2 or main memory.

Global bus controller – This is a key component of our system and it is responsible for:

- Accepting request from CPU Core. Only one core can be served at once, others have to wait.
- In case of local cache *SNOOP MISS*, scan through L2 cache looking for a requested data. Routing logic will be implemented.
- Sends a *SNOOP Request* if requested data does not exist in local cache.
- If neither L1 nor L2 have the requested data, look for it in global memory.
- Maintain the coherence of caches.

Global memory – Lowest level of memory hierarchy. Biggest capacity but also biggest latency.

L2 cache memory and controller – Two-way set-associative memory with 512 sets. Used as a second level of memory hierarchy, in case of L1 misses.

3. Interface

3.1. Interface between CPU Core and Global bus controller

This interface has 5 separate segments:

- Ports dedicated to sending data to bus controller.
- Ports dedicated to receiving data from bus controller.
- Ports dedicated to arbitraging in case both cores want to access bus at the same time.
- Standard memory ports.
- Ports dedicated to send data to L2.

3.2. Interface between global bus controller and L2

- Ports dedicated to sending data to L2.
- Ports dedicated to inform L1 memories in case of *HIT/MISS.*
- Standard memory ports.

3.3. Interface between L2 and main memory

- Ports dedicated to sending evicting data from L2 to data memory.
- Ports dedicated to receiving data from data memory in case of L2 miss.

4. Development process of system

In this paragraph we will briefly describe development of the whole system divided in three phases:

1. Single cycle core without local data cache subsystem
2. Single cycle core with local data cache subsystem (*L1*)
3. Full system (*with bus controller, L2 and data memory*)

4.1. Single cycle RISC-V CPU – phase 1

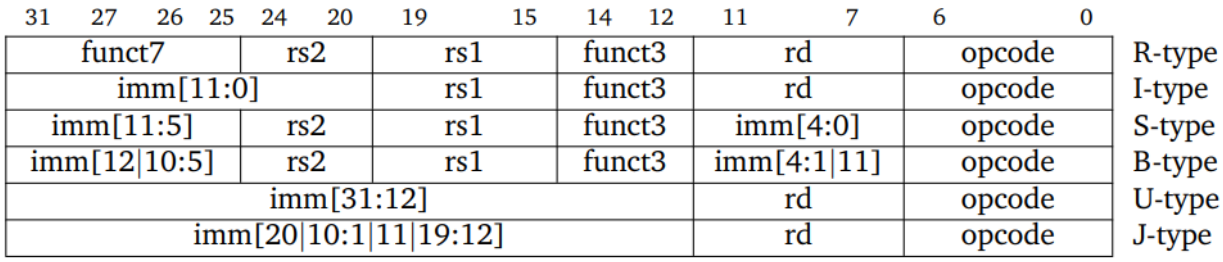

Figure 2 : Instruction formats

This table represents format of instruction sets and bit position of following fields:

1. *opcode* – is a 7 bit field which indicates type of instruction
2. *funct3*, *funct7* – fields that select the operation of instruction type
3. *rs1, rs2* – fields that represent addresses of registers whose values will be used in operation
4. *rd* – filed that represent address of register where result of operation will be stored
5. *imm (constant value)* – instead of using value from register, value can be hardcoded in instruction

During this phase we have implemented single cycle CPU with following instruction extensions from the table above.

1. **R extension** – Both operands come from register file and the result of operation is written back in register file on address stored in filed *rd*.
   R Instructions implemented: *add, sub, xor, or, and, sll, srl, sra, slt, sltu*.
2. **I extension** – One operand comes from register file and another is hardcoded into instruction filed *imm,* result is stored back in register file on address stored in filed *rd.*
   I Instructions implemented: *addi, xori, ori, andi, slli, srli, srai, slti, sltui*.
3. **S extension** – Value in register file from address *rs2* is stored in data memory on address whose value is sum of values from register file on address *rs1* and *imm.*
   S Instructions implemented: *sb (store byte), sh (store half word), sw (store word)*.
4. **L extension** – Value from data memory on address *rs1 + imm* is stored in *rd* in register file.
   L Instructions implemented: *lb (load byte), lh (load half word), lw (load word)*.
5. **B extension** – Depending on *funct3,* values from register *rs1* and *rs2* will be compared and aftermath is incrementing value of program counter with immediate value or with 4.
   B Instructions implemented: *beq, bne, blt, bge, bltu, bgeu*.

6. **U extension** - The final operation result is related to the 20-bit *imm*, and the result is written back to the *rd* register.
   U Instruction implemented: *lui*.
7. **J extension** - The format of this instruction is very similar to U-type, it only have *rd* and *imm* and opcode.
   J Instruction implemented: *jal*.

All these instructions where firstly verified in Vivado Suit, initially we wrote some simple RISC-V assembly code, translated it in machine code using Ripes Simulator.
Then we loaded this code as stimuli for testbench to check if values in Vivado match with values from Ripes.

Stimuli is stored folder *instruction_tests*, and Vivado testbench is stored in folder *tb.*

Once we completed this, we started formally verifying developed CPU with JasperGold tool. As an introduction to tool, we verified module Controller, Branch Condition and Immediate Generator with unit level reference model, all necessary files for those three modules are stored in folder *verif* separated in three subfolders *branch_checking*, *controller_checking* and *immediate_checking.*

When all asserts passed and bugs were fixed, in folder *verif* in subfolder *reference_model* we started developing reference model of whole CPU as a system level abstraction.

To increase readability, we divided some code in separate files.
File *defines.sv* stores values of opcodes represented in binary, *struct.sv* is used to define structures whose variables were used to make easier debug process and better vision in waveform.

In file *ref_model.sv* firstly we needed to constraint input of stimuli, for this we used verification directive *assume*.

Because our system uses both clock edges we had to create properties for assumes for both positive and negative edge of clock.

- Assumes present in our code:

1. *assume_opcodes*: This assume restricts seven lowest bits of instruction to adequate opcodes for our system (image *Figure 3*)
2. *assume_load_rs2_not_NULL*: If instruction opcode is LOAD this assume restrict that *rd* cannot be ZERO because register x0 is hardcoded to zero, and sum of values from *rs1* and *imm* should be smaller than 1024 since that is our data memory size.
3. *assume_store_less_than_1024*: If instruction opcode is STORE this assume restrict that sum of values from *rs1* and *imm* should be smaller than 1024 since that is our data memory size.
4. *assume_cant_write_to_x0*: This assume does not allow to write result of R, I and U instruction in x0 because register x0 is hardcoded to zero.
5. *assume_fvar_limit,* assume_fvar_stable: This assumes restrict value of free variable to be smaller than 1024 and keeps value of it through whole verification process

- In order to verify system we used auxiliary code. Purpose of that was to make whole process of verification easier. Each auxiliary code maps to a certain module in order to develop properties for asserts easier.

At the end whole purpose of reference model is to compare if there is any mismatch between expected values (*calculated aux code of ref_model*) and real values that appear in CPU.

Almost all auxiliary code used in this reference model is using both combinational and sequential logic and is adapted for both positive and negative edges (*data in register file and in data memory is stored on negative edge*).

4.2. Bugs found during verification of CPU – phase 1

1. Wrong implementation of data memory (*sw,sb,sh*) – Wrong data was stored on wrong address in data memory.

```
always_comb begin
        write_data = 'b0;
        if (wr_en) begin                // S-type instruction
            //read_data_from_memory = memory[addr[31:2]];

                case (mask)
                        3'b000: begin   // Store byte
                                case (addr[1:0])
                                        //0: write_data = {24'b0, wdata[7:0]};
                                        //1: write_data = {24'b0, wdata[15:8]};
                                        //2: write_data = {24'b0, wdata[23:16]};
                                        //3: write_data = {24'b0, wdata[31:24]};

                                        0: write_data = (memory[addr[31:2]] & 32'hFFFFFF00) | {24'b0, wdata[7:0]};
                                        1: write_data = (memory[addr[31:2]] & 32'hFFFF00FF) | {16'b0, wdata[7:0], 8'b0};
                                        2: write_data = (memory[addr[31:2]] & 32'hFF00FFFF) | {8'b0, wdata[7:0], 16'b0};
                                        3: write_data = (memory[addr[31:2]] & 32'h00FFFFFF) | {wdata[7:0], 24'b0};
                                        default : write_data = 0;
                                endcase
                        end

                        3'b001: begin   // Store halfword
                                case (addr[1])
                                    //0: write_data = {16'b0, wdata[15:0]};
                                        //1: write_data = {16'b0, wdata[31:16]};

                                        0: write_data = (memory[addr[31:2]] & 32'hFFFF0000) | {16'b0, wdata[15:0]};
                                        1: write_data = (memory[addr[31:2]] & 32'h0000FFFF) | {wdata[15:0],16'b0};
                                        default : write_data = 0;
                                endcase
                        end
                        3'b010: begin   // Store word
                                write_data = wdata;
                        end
                        // This is changed - did not exist before
                        default : write_data = 0;
                endcase
        end
end
```

Figure 3: Fixed store block in DataMemory.sv

Commented section in code is wrong implementation. For example if you wanted to store half word on two lowest bytes, half word was stored on its location but data on other two bytes was cleared on zero. Same problem was with store half word on higher two bytes, two lowest bytes was cleared on zero. If you wanted to store byte on lowest byte of the word, this byte would be stored but other three bytes was cleared on zero, same problem occurred with storing byte on other locations in a 4 byte word. In order to fix this problem before storing data in data memory, we had to read whole word, and depending on type of store we had to mask bytes that is not relevant for our store and to keep their values instead of clearing them on zero.

2. Wrong implementation of I - opcode in *controller.sv* – In if statement there was a wrong value of *funct7 (Mismatch from reference card – funct7 was 0x02 instead of 0x20)*

Before: 3'b101: begin if (func7 == 7'b0000010) alu_op <= 6; else alu_op <= 5; end

After: 3'b101: begin if (func7 == 7'b0100000) alu_op <= 6; else alu_op <= 5; end

3. Wrong implementation of U - opcode in controller, wrong value of *alu_op* – ALU was missing state intended for *LUI* operation so we added state 10 and in Controller we set *alu_op* to 10.

Changes in controller.sv for U-type instruction: alu_op = 10;

Changes in ALU.sv: 10: C = B;

4. Missing adder in case of jump or branch – Output of immediate generator was directly connected to ALU instead of dedicated adder and address wasn't calculated correctly. We added adder and connected mentioned output to adder input.

Changes in Processor.sv: add_immediate add_imm(.in1(index), .in2(B_i), .out(add_imm_s));

5. Wrong implementation of STORE half word and STORE byte in reference model – There was not case statement in store half word for checking addr[1], and for store byte as well.

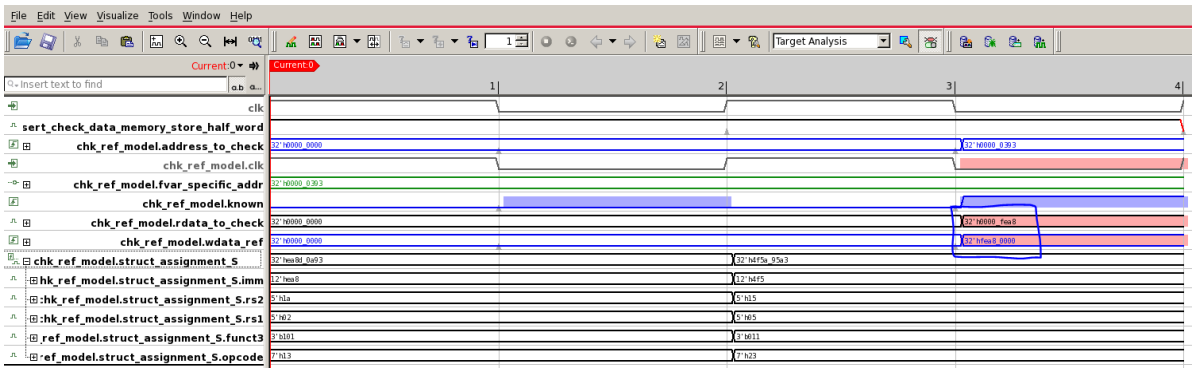

Figure 4 : Load half word error

6. Instruction was not properly read from instruction memory – To convert assembly in machine code we used Ripes simulator. Simulator also shows how machine code is executed on its architecture and we also used this as a reference model to compare register values with our register values. While executing code stored in *store_load_instrution_test.txt* we saw in waveform that first instruction which is *addi x3, x0, 1119* is not executed in our design.

Bug was in file *Add4.sv*, counter would start incrementing its values in restart-routine and it caused that first instruction was not executed because program counter wouldn't point on first location in instruction memory. We added in if-statement check for restart so counter will start incrementing its values after restart is finished.

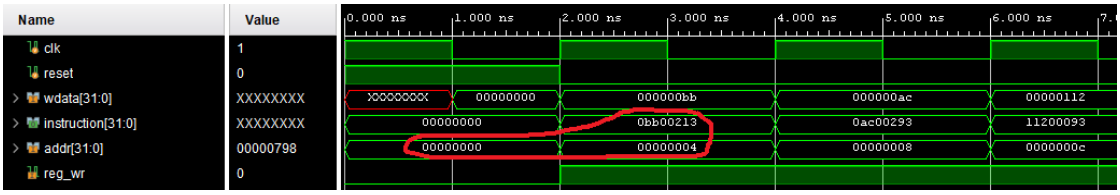

Figure 5: Program counter bug

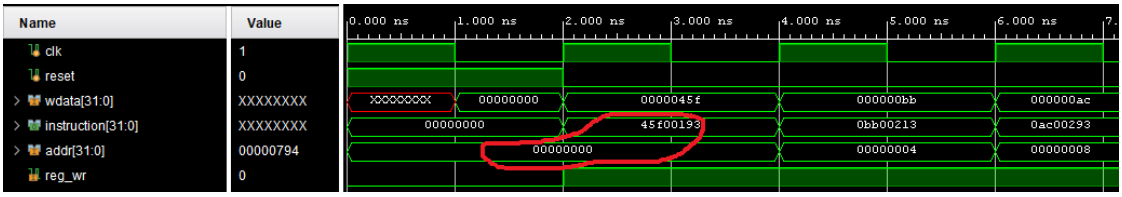

Figure 6: Fixed bug with program counter

7. One cycle delay between writing data in cache memory and data memory – Write through policy requires that upon STORE instruction, we write data in both cache and data memory in the same cycle. Address port for data memory was delayed one cycle compared to the address port for cache memory and this caused mismatch.

- *Opcode 23 is opcode for STORE instruction.*
- *dmem_address is output from cache memory that sends address to data memory to write on.*
- *dm_address is assigned signal in cache that represents address in cache to write on.*

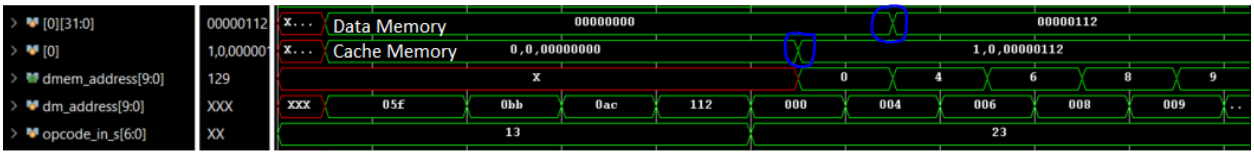

Figure 7: One cycle delay upon storing data in both cache and data memory

4.3. Coverage results after testing CPU – phase 1

Once all asserts passed, we used JasperGold Coverage App. On figure 3. and figure 4. for you can see passed asserts and coverage results.

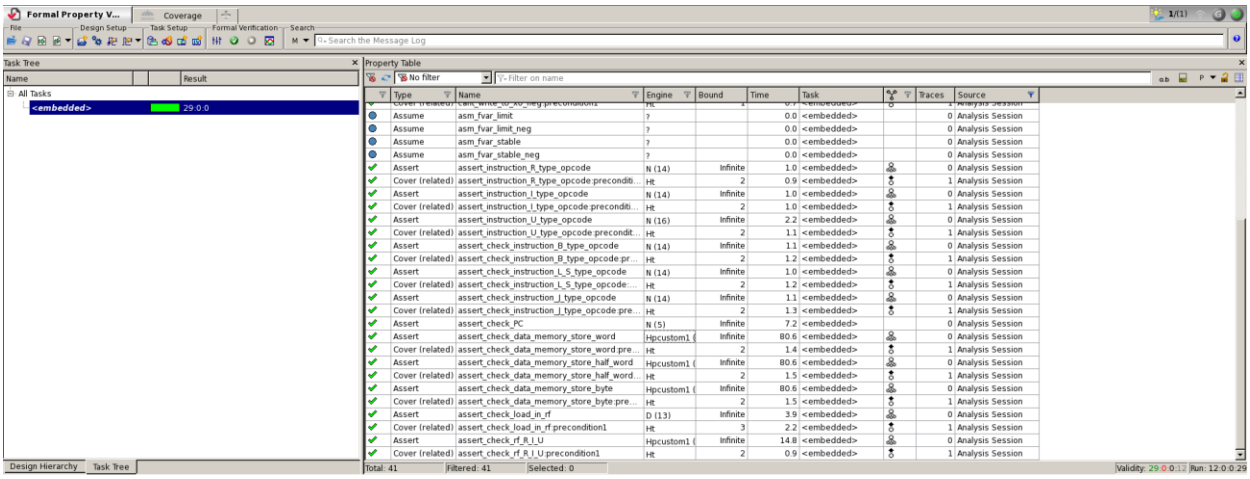

Figure 5: Asserts results

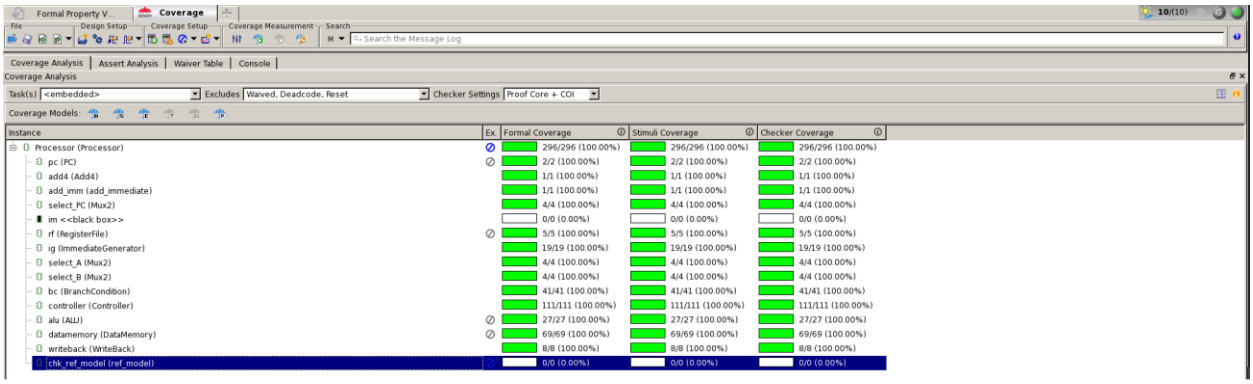

Figure 6: Coverage results

4.4. Single cycle with local data cache subsystem development – phase 2

After previously designed and verified core from phase 1, we move on to phase 2 by adding a simple local data cache subsystem. Due to our core being a single cycle variant, we do not expect any significant performance upgrades. This is just an implementation of a concept in order to understand basics of cache and its relation to the RISC-V core.

The structure of cache is ***direct-mapped*** cache. For write policy we use ***write-back***.

Cache memory has 256 locations. To address cache lines we use lowest 8 bits of address (*alu_out*) for index and other 24 bits for tag. Each location width is 58 bit and divided in 3 parts:

- Two highest bits represent a **MESI state**.
- Next 24 bits represent a **tag**.
- Rest represent an actual 32 bit **data** to be store or loaded.

Cache is placed in parallel with data memory module and we distinguish two scenarios accessing memory, LOAD and STORE instructions.

- When STORE instruction occurs, we use *write-through* policy and write incoming data to both data memory and cache memory at the same time on a **negative** clock edge. When the STORE happens on a location that already has some data, we simply overwrite it with new data.
- When LOAD instruction occurs, subsystem has 2 scenarios, load **hit** and load **miss**. If data in cache is valid and tags match, hit happens and data from cache is instantly (*on negative edge of the same cycle*) loaded in register file. In case of a load miss, new *WAIT_WRITE* state is introduced that stalls the core for 1 cycle in order to fetch the required data from data memory. If load happens to the already filled location, data is overwritten again.

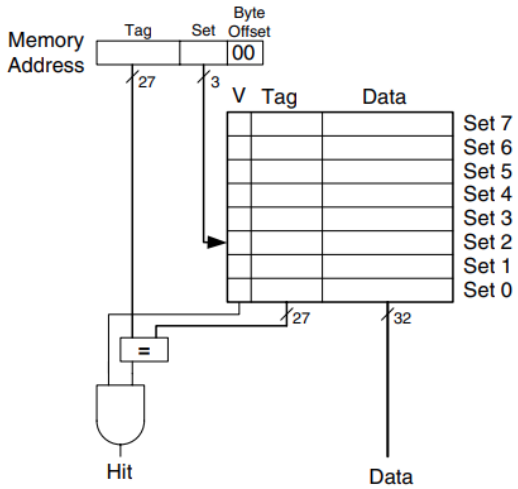

Figure 8: Direct mapped cache [4]

4.5. Bugs found during verification of CPU – phase 2

1. Wrong implementation of FSM – Initially we had states IDLE, MISS and WAIT WRITE. In Vivado, problem did not occur due to specific test sequence we set. However in formal tool, we realized that one state is unnecessary and we reduced FSM to only two states. Now, we have MAIN state where we check if STORE or LOAD happened and we transition to WAIT_WRITE state only upon load miss.

2. Half word and byte storing – When we needed to check if the same half word is correctly stored in both data memory and cache memory on same location, property was comparing grey box data memory signal with cache memory. Signal *rdata_to_check* in reference model did not do adequate masking (*read only half of the word and filled rest of it with zeros*), so mismatch occurred. Same issue occurred for storing a byte.

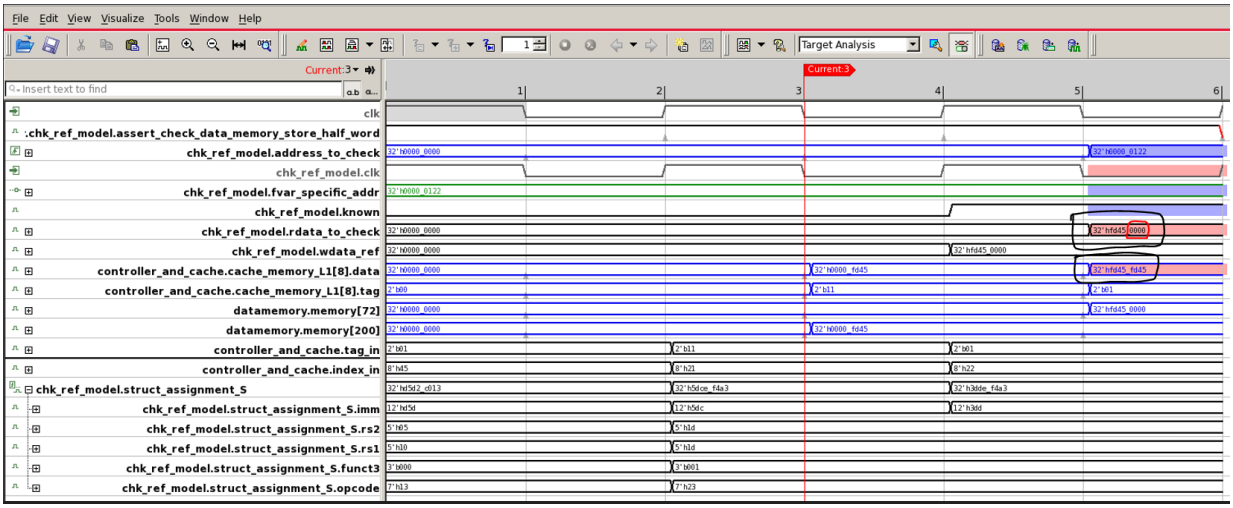

Figure 9: Half word mismatch

3. Delay issues due to a possibly wrong property – New property instance started on cycle 4 due to present skew even though it shouldn't have. Property now uses function *$past* and this problem is avoided.

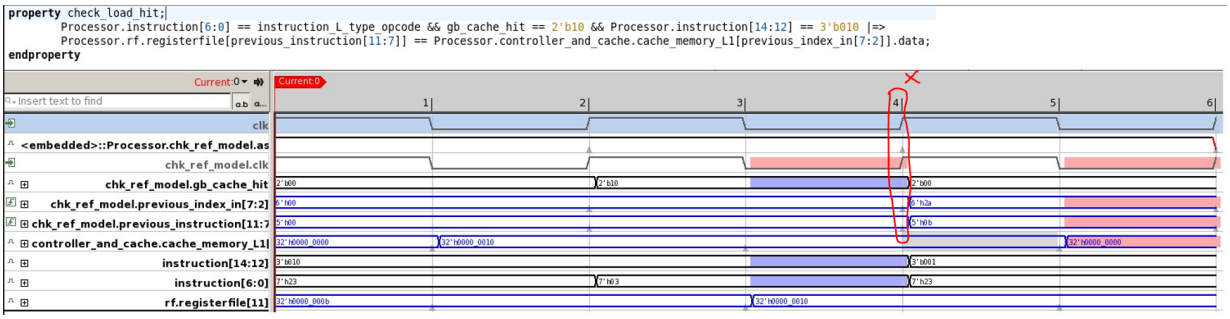

Figure 10: Skew issue

4. LOAD miss scenario issue – Cache subsystem had combinational logic for STORE and for LOAD instructions. STORE instruction has variations of store word, store half word and store byte, so 3 mask values. LOAD instruction has same variations including signed half word and signed byte, so 5 mask values. In case of LOAD miss and signed variants, after stalling, fetched data has to be **stored** in cache but instruction is still **load**, and so combinational logic for **store** was missing those 2 mask values for signed byte and signed half word. In CEX (*counter example*), we see that upon signed half word we enter *default* branch which sets data to 0 instead of actual value.

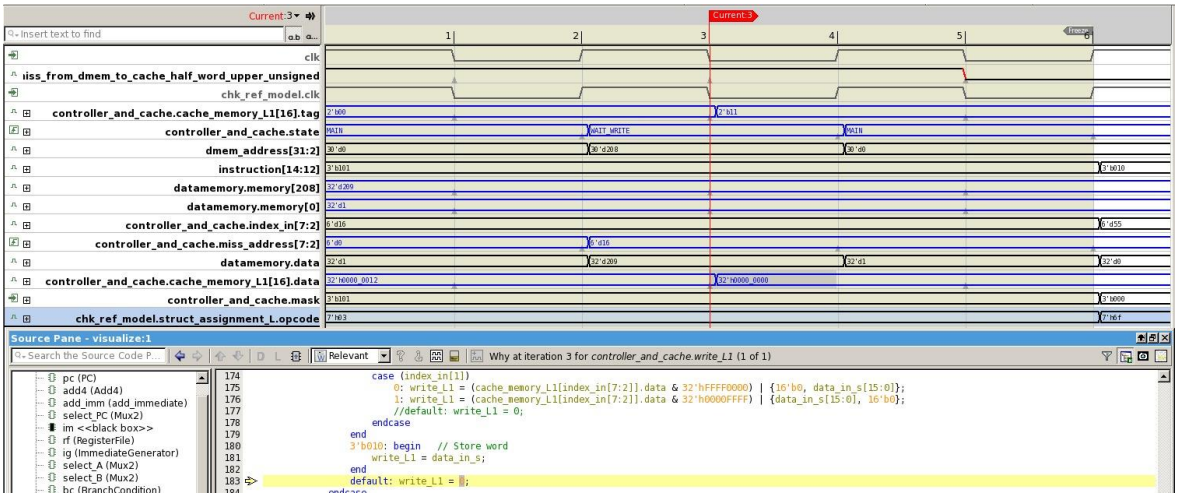

Figure 11: Load miss bug – Signed half word

4.6. Coverage results after testing CPU – phase 2

Once all asserts passed, we used JasperGold Coverage App. On figure 12 and figure 13 for you can see passed asserts and coverage results.

Property Table

No filter | Filter on name

Type	Name	Engine	Bound	Time	Task
Assert	assert_instruction_R_type_opcode	N (14)	Infinite	0.4	<embedded>
Assert	assert_instruction_I_type_opcode	N (14)	Infinite	0.4	<embedded>
Assert	assert_instruction_U_type_opcode	N (16)	Infinite	0.8	<embedded>
Assert	assert_check_instruction_B_type_opcode	N (14)	Infinite	0.4	<embedded>
Assert	assert_check_instruction_L_S_type_opcode	N (14)	Infinite	0.4	<embedded>
Assert	assert_check_instruction_J_type_opcode	N (14)	Infinite	0.5	<embedded>
Assert	assert_check_PC	N (14)	Infinite	0.9	<embedded>
Assert	assert_check_data_memory_store_word	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_data_memory_store_half_word_upper	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_data_memory_store_half_word_lower	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_data_memory_store_byte0	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_data_memory_store_byte1	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_data_memory_store_byte2	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_data_memory_store_byte3	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_word	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_half_word_lower_unsigned	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_half_word_upper_unsigned	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_half_word_lower_signed	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_half_word_upper_signed	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte0_unsigned	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte1_unsigned	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte2_unsigned	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte3_unsigned	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte0_signed	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte1_signed	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte2_signed	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_hit_byte3_signed	Hpcustom1 (	Infinite	132.7	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_word	Hpcustom1 (	Infinite	164.4	<embedded>

Property Table

No filter | Filter on name

Type	Name	Engine	Bound	Time	Task
Assert	assert_check_load_miss_from_cache_to_rf_word	Hp (5)	Infinite	195.4	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_half_word_upper_signed	Hpcustom1 (	Infinite	164.4	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_half_word_lower_signed	Hpcustom1 (	Infinite	164.4	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_half_word_upper_signed	Hp (5)	Infinite	195.4	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_half_word_lower_signed	Hp (5)	Infinite	195.4	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_half_word_upper_unsigned	Hpcustom1 (	Infinite	164.4	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_half_word_lower_unsigned	Hpcustom1 (	Infinite	164.4	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_half_word_upper_unsigned	Hpcustom1 (	Infinite	173.4	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_half_word_lower_unsigned	Hpcustom1 (	Infinite	173.4	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte0_signed	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte1_signed	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte2_signed	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte3_signed	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte0_unsigned	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte1_unsigned	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte2_unsigned	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_dmem_to_cache_byte3_unsigned	Hp (4)	Infinite	106.0	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte0_unsigned	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte1_unsigned	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte2_unsigned	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte3_unsigned	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte0_signed	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte1_signed	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte2_signed	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_load_miss_from_cache_to_rf_byte3_signed	Hp (5)	Infinite	270.1	<embedded>
Assert	assert_check_state_transition_MAIN_WAIT_WRITE	PRE (1)	Infinite	0.0	<embedded>
Assert	assert_check_state_transition_WAIT_WRITE_MAIN	N (14)	Infinite	0.3	<embedded>
Assert	assert_check_rf_R_I_U	Hp (2)	Infinite	20.5	<embedded>

Figure 12: Asserts for phase 2

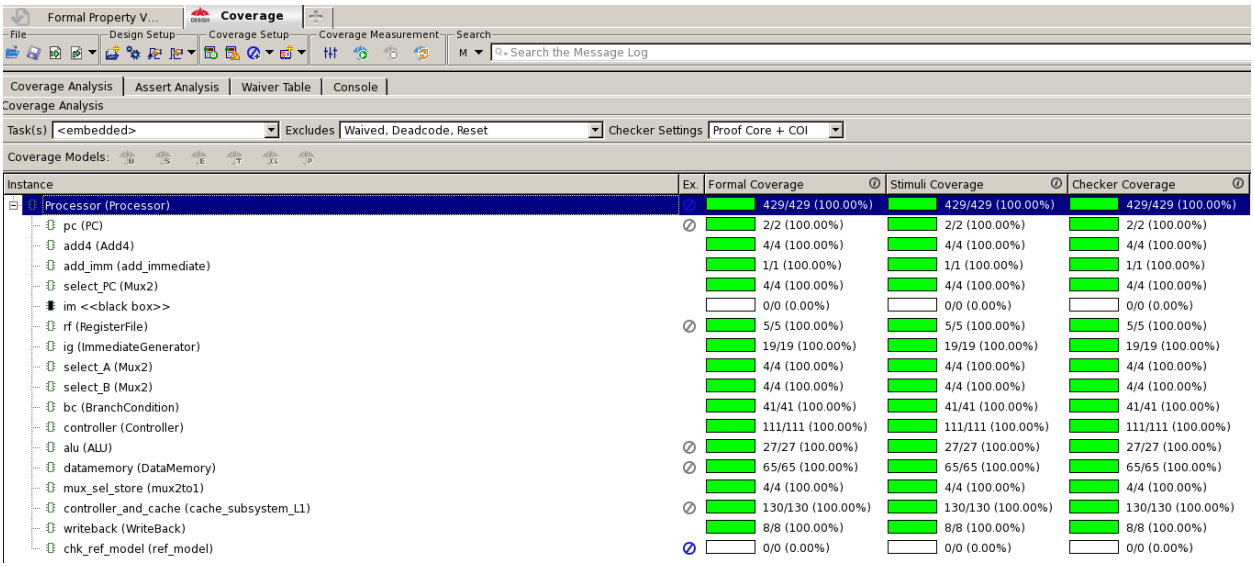

Figure 13: Coverage for phase 2

4.7. Full system development – phase 3

This phase describes the whole system overview. **Data memory** from core was moved out, and now it represents third level of memory hierarchy. We load from that memory in case of both L1 and L2 miss. It doesn't have a *cache* structure, it is just a simple memory implementation. For testing purposes it will have 1024 locations, but once it's verified we will increase its capacity to 256k lines. Final design has following components:

- Two RISCV single cycle cores (*figure 16*) with individual L1 cache subsystem (cache memory + controller). They are *direct-mapped* and use a *MESI cache coherence protocol* (*figure 14*).
- Bus controller that handles requests from cores, maintain MESI states and transfers data between L1 memories and L2 memory.
- L2 cache subsystem (cache memory + controller) that is 2-way *set-associative* with 512 sets (1024 lines, 4 times bigger than L1). It uses LRU (*Least recently used*) as a replacement policy.
- Data memory mentioned above.

Full system elaborated design is shown on figure 15.

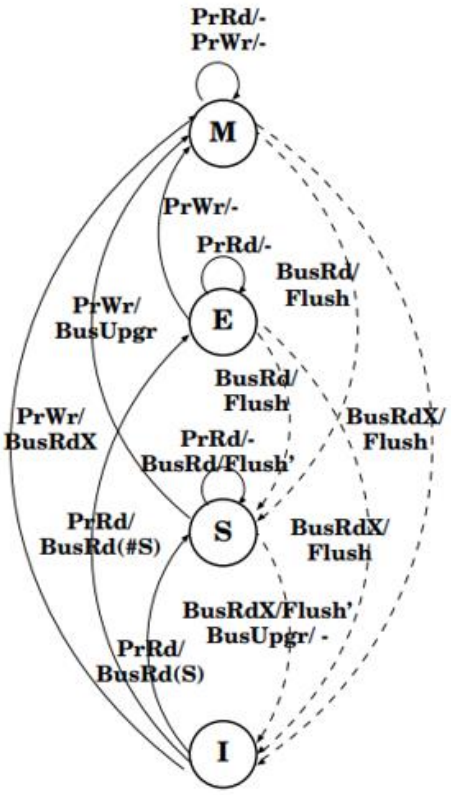

Figure 14: MESI protocol [5]

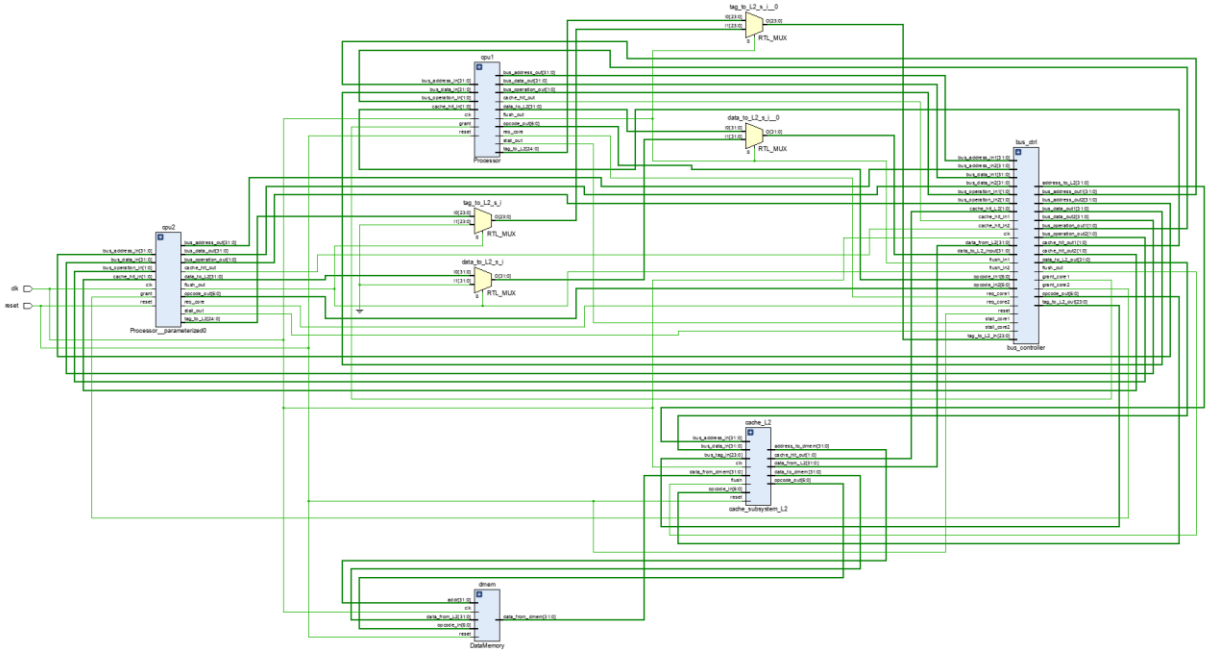

Figure 15: Elaborated system design

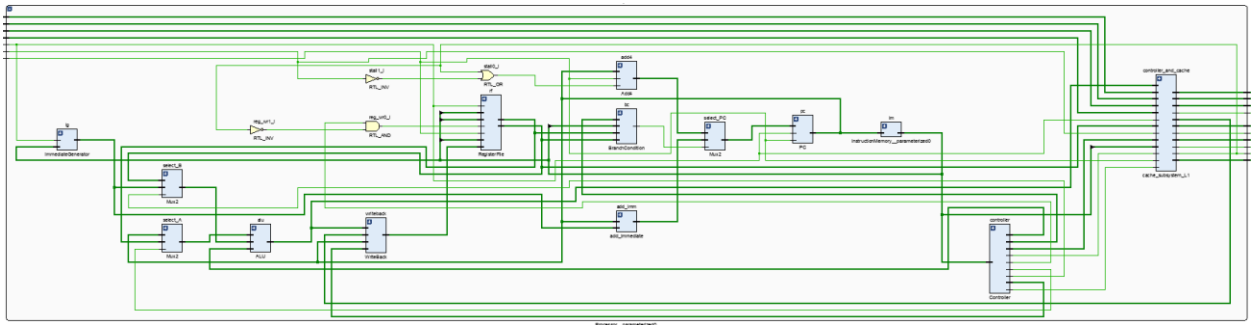

Figure 16: Elaborated core design

4.8. Bugs found during design process – phase 3

Using RIPES simulator we translated assembly code into machine code and used some instruction sequences to test design. Those can be found in folder *instruction_tests*.

1. Wrong value of signal *bus_operation_in* – Default value in *always_comb* dedicated for processor side of MESI protocol was set to 0 which represents a bus operation *BusRd*. This is fixed by changing it to 2'b11 which means *no operation* on bus.

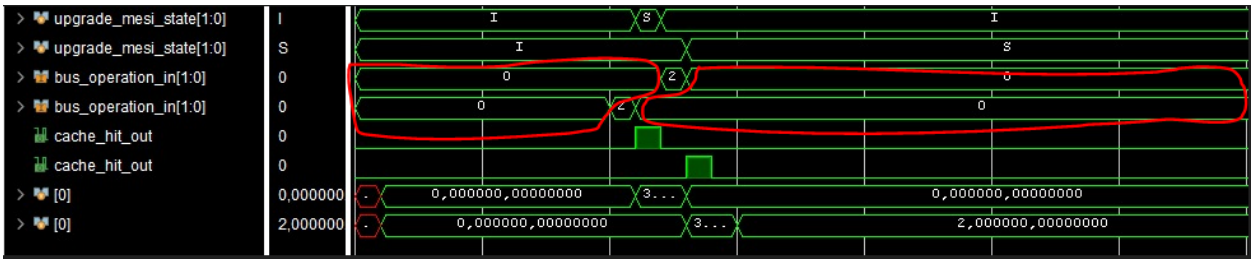

Figure 17: Wrong default value for bus_operation_in

2. Wrong setting of cache hit signal in L1 – *Cache_hit_in* is a signal that informs a core that requested data, if other one has it. For checking we used grey-box signal *tag_in* that went 'X' once all instructions were finished which resulted in wrong cache hit information. We replaced it with *bus_address[31:8]* and bug was removed.

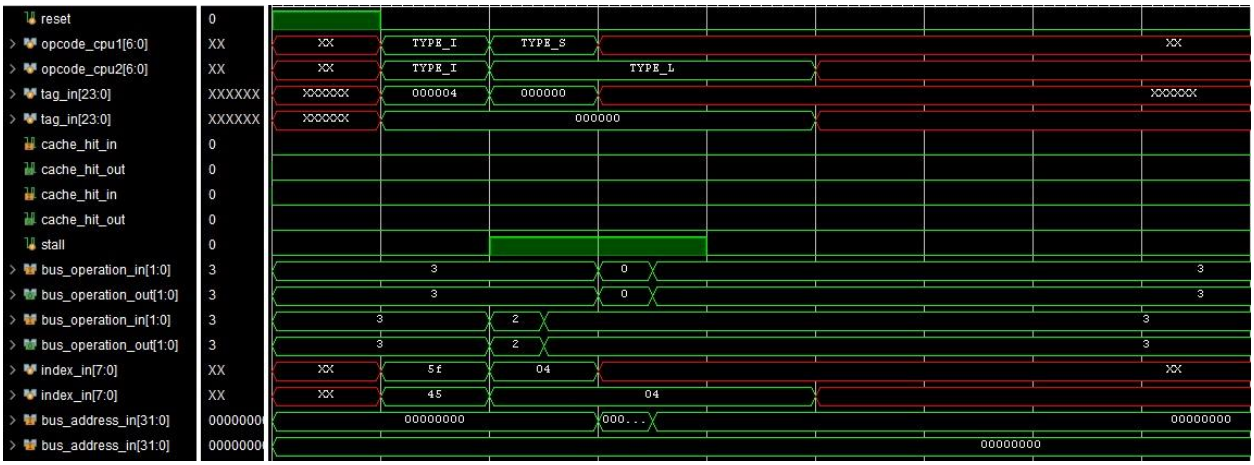

Figure 18: Cache hit signal doesn't rise

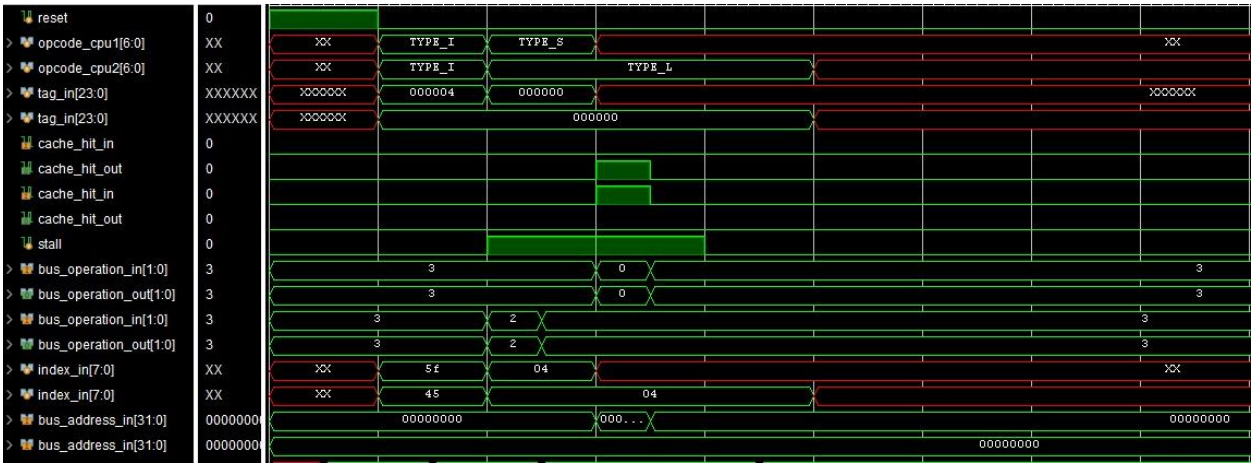

Figure 19: Cache hit signal asserts – bug resolved

3. Register file write issue – Writing in register file was allowed even though *stall* signal was active and cache hit signal was 0, which should not be the case. Bug was resolved by modifying following line in *Processor.sv* file.

RegisterFile rf (.clk(clk), .reset(reset), .reg_wr(reg_wr && !stall),…

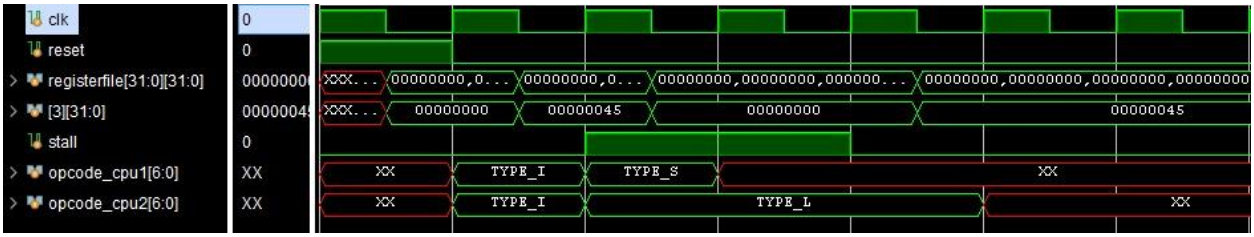

Figure 20: Stall signal is active and store happens – Bug

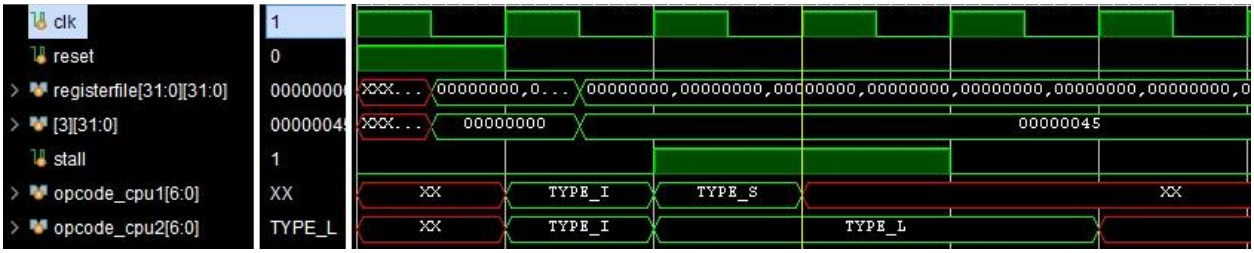

Figure 21: Bug resolved – No stores during stall

4. L2 load miss issue – In the load miss scenario in L2, where both LRU bit of both ways is 0 (*initial*), address to data memory was not forwarded and thus the data was not fetched. By simply adding address in this if block, bug was resolved.

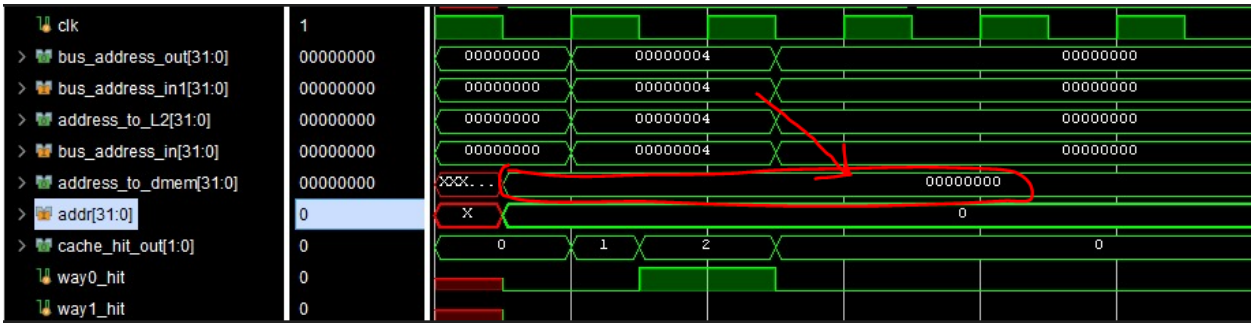

Figure 21: Address to data memory not set

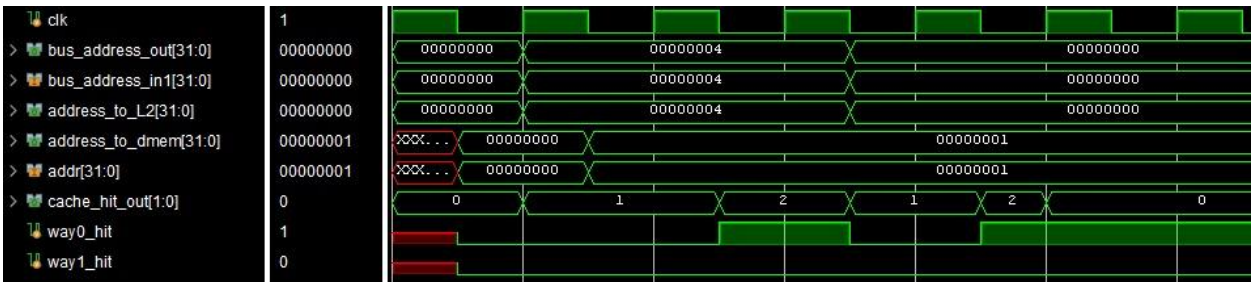

Figure 22: Address forwarded to Data memory

References

[1] https://github.com/mortbopet/Ripes/releases
[2] https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf
[3] https://fraserinnovations.com/risc-v/risc-v-instruction-set-explanation/
[4] "Digital Design and Computer Architecture (Second Edition)", Sarah L. Harris, David Harris
[5] T. Suh, "*Integration and evaluation of cache coherence protocols for multiprocessor socs*", 2006.