Multicore RISC-V cache controller

Master thesis specification

Menthors :

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 global cache controller that connects L2 caches 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. We will verify each phase of pipeline and their functional dependencies. 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.

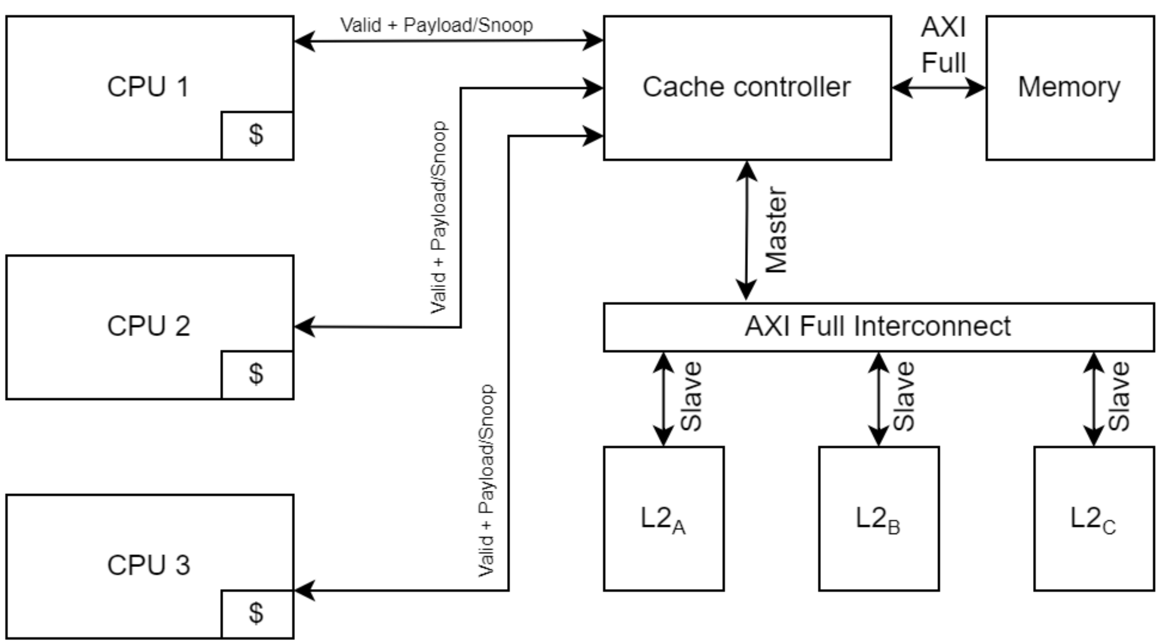

Figure 1 : System Overview

2. Components

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

1. CPU cores (*CPU1*, *CPU2* and *CPU3*) with their local cache memories (*$*).
2. Global cache controller (*Cache controller*)
3. Global memory (*Memory*)
4. AXI Interconnect
5. Shared L2 cache memories ($L2_A$, $L2_B$, $L2_C$)

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

CPU Cores – Each core has RISCV ISA, 5 stages of pipeline (*Instruction fetch*, *Instruction decode*, *Execute*, *Memory access* and *Write back*), local cache memory (L1) and cache controller. Our idea is to implement an instruction set (**maybe *M* or *I* (to be defined...)**). 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 cache controller receives this request, it has to access other L1 caches and look for a requested data there. It this also fails, global cache controller has to look for request data in L2 or main memory.

Global cache 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 caches 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.

AXI Interconnect – Component that connects global cache controller and shared L2 caches. Also does the required routing.

Shared L2 cache memories – Composed of smaller L2 memories in order to reduce latency of data access.

3. Interface

3.1. Interface between CPU Core and Global cache controller

This interface has 5 separate segments:

- Valid + Payload interface to global cache controller, requires address, opcode and data (*in case of non-write command)*
- Valid + Payload for *SNOOP Response*. If it was a *HIT* forward the request data in *payload* and flag a success (*1*), otherwise flag a fail (*0*).
- *SNOOP Request* interface goes from global cache controller to all cores (*number of cores = number of instances of this interface*)
- Data transfer interface from global cache controller to all cores. Fetched data from either L2 or global memory.
- Valid + Payload completion interface. If transaction is completed flag success, otherwise flag an error.

3.2. Interface between global cache controller and L2

For this interface, we decided to use AXI Full, because of its burst mode. We want to fetch a 64 byte cache line from one of slaves. We have one master, global cache controller.

3.3. Interface between global cache controller and main memory

For this interface, we decided to use AXI Full, because of its burst mode. We want to fetch a 64 byte data from the main memory.

4. Commands

4.1. Commands for local cache memories (L1)

- **READ** – This command reads requested data from L1. If it's a *HIT* proceed with an instruction, otherwise it's a *MISS* and CPU Core generates a command for a lower level. Global cache controller takes over the control.
- **READ + UNIQUE –** After global cache controller assures that only CPU that generated this command has a unique data, read it. Idea for this is to be an exclusive state, however we probably won't follow the MOESI protocol exactly. If this instruction is sent, check if cache line is unique and then read it.
- **WRITE –** If there is an empty location in L1, write it. If cache is full, apply LRU, write back that data to L2, and replace it with the new one from the write command.
- **CMO (Cache maintenance operation)**
  - **MI (Make invalid) –** Invalidate data without propagating it to lower level.
  - **CI (Cache invalidate) –** If data was modified (*Dirty*) before invalidating, it must be forwarded to lower level. Otherwise, just invalidate it.

4.2. Commands for L2 caches

- **WRITE –** In this scenario, global cache controller should invalidate this data in all other cores without propagation to lower levels (*SNOOP MI*). For example, if CPU1 generated this command, global cache controller should invalidate this cache line in CPU2 and CPU3 (*if they have this data*). In this situation we assume that this cache line is the newest.
- **READ –** Do a *SNOOP Read.* In this scenario, request data does not exist in neither one L1 cache memory. Read the cache line from L2 and forward it to global cache controller.
- **READ + UNIQUE –** Do a *SNOOP CI.* For example if CPU1 sent this instruction, it fetched cache line X from L2 and modified it resulting in it to become *Dirty*. If then CPU2 requests the same instruction for the same cache line, global cache controllers needs to *SNOOP* on other L1 caches to check if they have that cache line, and invalidate them (*SNOOP CI*), resulting in CPU2 having that cache line clean (*updated*).
- **CMO (Cache maintenance operation)**
  - **MI (Make invalid) –** Do *SNOOP MI.* Just invalidate cache line in L2, without propagation to the global memory.
  - **CI (Cache invalid) –** Do *SNOOP CI.* If the cache line is modified (*Dirty*), invalidate it in L2, and forward it to the global memory.

4.3. Commands for CPU Snooping (from CPU Core to global cache controller)

- **SNOOP READ –** Try to find requested cache line in other L1 memories. If cache line is found, forward it to a global cache controller.
- **SNOOP CI –** Try to find requested cache line in other L1 memories. If the data is *dirty,* invalidate it in L1 you found it in, forward it to the CPU Core that requested it and this CPU Core is obligated to propagate it to lower level.
- **SNOOP MI –** Try to find requested cache line in other L1 memories. If cache line is found, forward it to CPU Core that request it (*via global cache controller*) and invalidate this cache line in all other CPU Cores (*if they have that cache line*).

5. Development process of system

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

5.1. Single cycle RISC-V CPU

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

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 auxillary code. Purpose of that was to make whole process of verification easier. Each auxillary 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 auxillary 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*).

5.2 Bugs found during verification of CPU

1. <u>Wrong implementation of data memory (*sw,sb,sh*)</u> – 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. <u>Wrong implementation of I - opcode in *controller.sv*</u> – 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. <u>Wrong implementation of U - opcode in controller, wrong value of *alu_op*</u> – 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. <u>Missing adder in case of jump or branch</u> – 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. <u>Wrong implementation of STORE half word and STORE byte in reference model</u> – There was not case statement in store half word for checking addr[1], and for store byte as well.

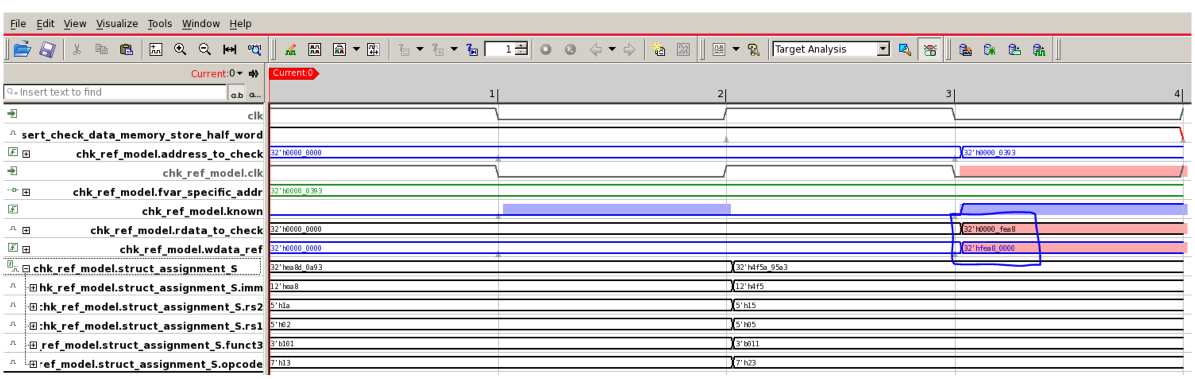

Figure 4 : Load half word error

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.

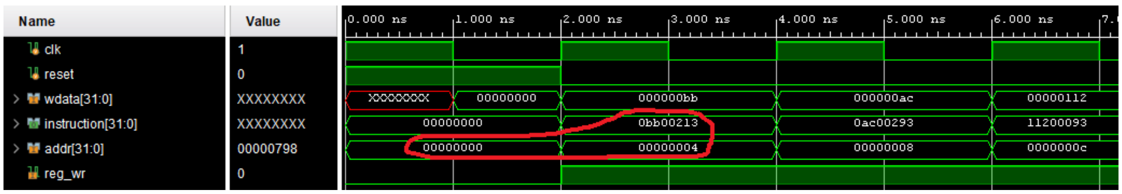

Figure 5: Program counter bug

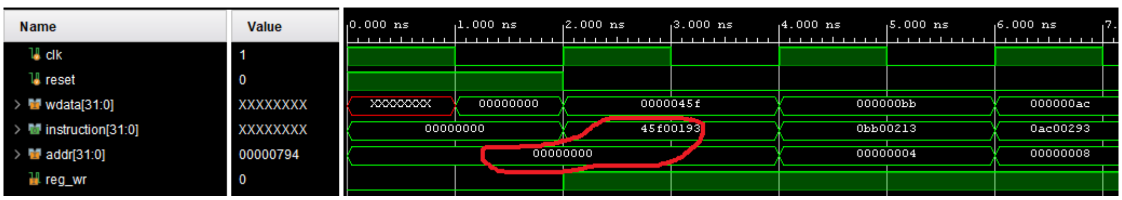

Figure 6: Fixed bug with program counter

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

5.3 Coverage results after testing CPU

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

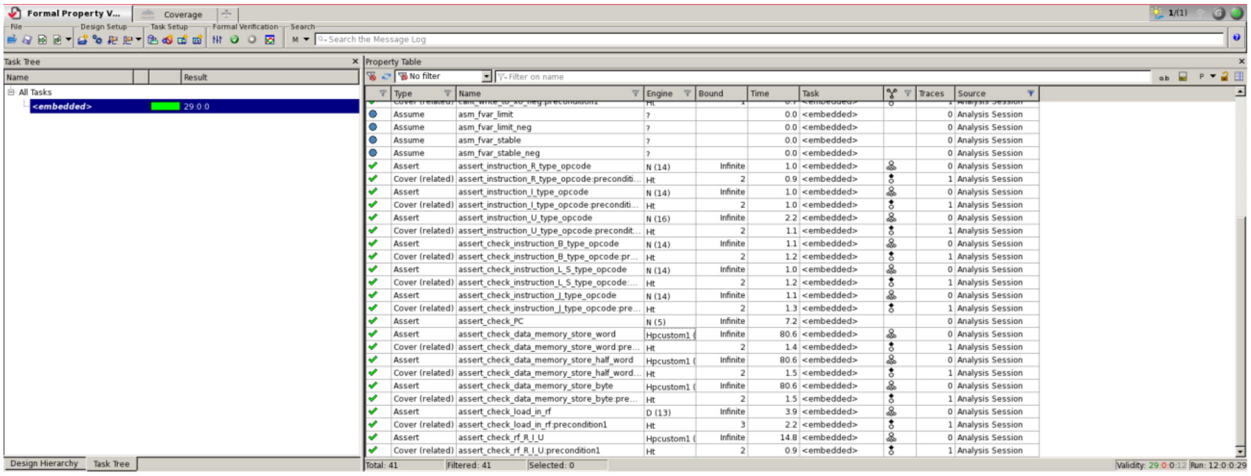

Figure 5: Asserts results

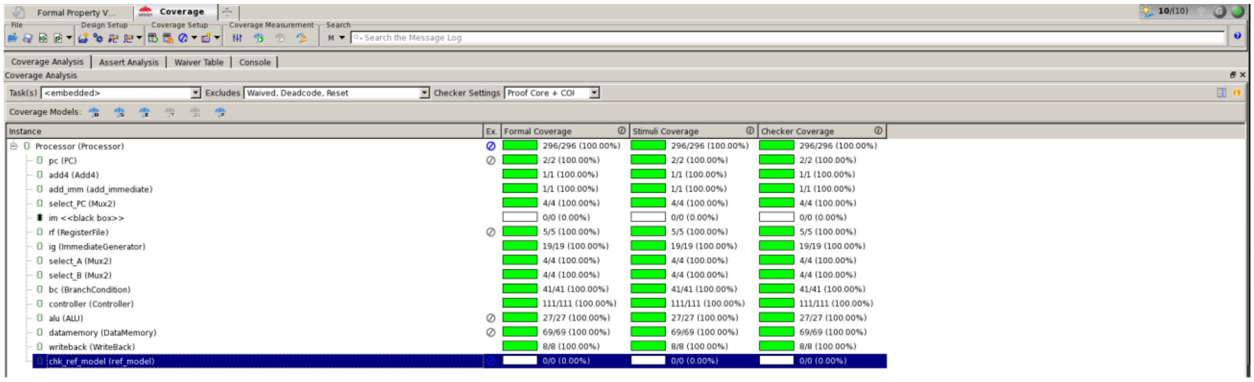

Figure 6: Coverage results

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/