Project Report:

Formal Verification of a 32 - bit core based on RISC – V Architecture

Yuqi Lin (yl5334) Xi Wang (xw2935)

Department of Electrical Engineering, Columbia University

ENGIE 6863: FORMAL VERIF HW SW SYSTEMS

Prof. Michael Theobald

December 23rd, 2023

Contents

Overview

This final report is mainly focused on the formal verification of 32-bit Risc-V architecture. This report will include all the work we have done in the last several weeks. The first part of this report is the FIFO task, which includes the basic property assertion and model checking methods, such as OVL and Golden model. The second part of this report will introduce our formal verification of 32-bit Risc-V architecture. We will first introduce the structure of our verified design and then illustrate how we gradually prove and change this design using formal verification method. The last part is the summary of our whole project.

Work Contribution

We have divided this project in several part.

Data collection: Yuqi Lin, Xi Wang

Property: Yuqi Lin, Xi Wang

Coding: Yuqi Lin

Formal Verification tools and Debug: Yuqi Lin, Xi Wang

Report: Yuqi Lin, Xi Wang

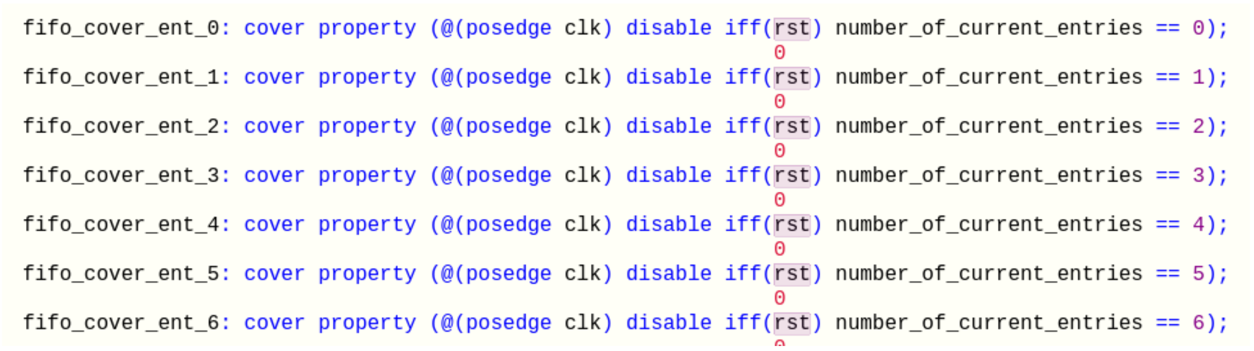
Synchronous FIFO Formal Verification

We completed a formal verification on the synchronous FIFO in out project status II. This FIFO project introduced us with the basic model checking method, including the property assertion, OVL and golden model equivalence checking. The following part will be divided into three tasks to illustrate how we gradually proved the FIFO structure.

Task I

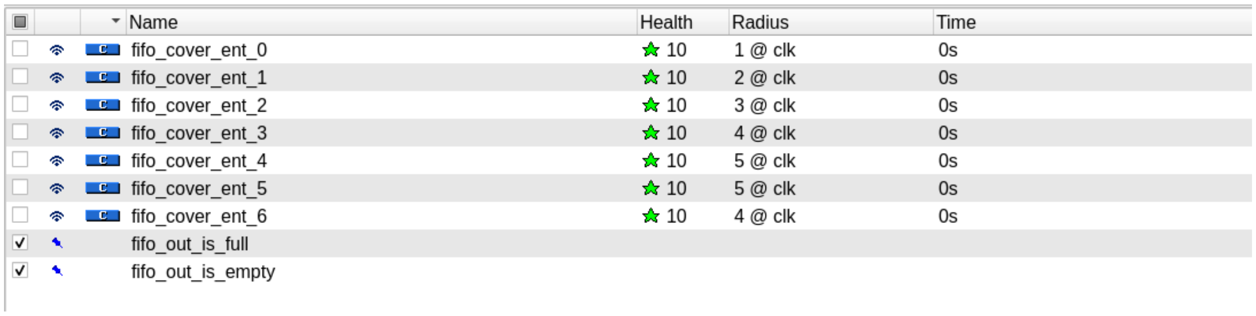
A.

We add the cover property in Fig. 1 using the cover properties command and get the result.



```
fifo_cover_ent_0: cover property (@(posedge clk) disable iff(rst) number_of_current_entries == 0);
                                                            0
fifo_cover_ent_1: cover property (@(posedge clk) disable iff(rst) number_of_current_entries == 1);
                                                            0
fifo_cover_ent_2: cover property (@(posedge clk) disable iff(rst) number_of_current_entries == 2);
                                                            0
fifo_cover_ent_3: cover property (@(posedge clk) disable iff(rst) number_of_current_entries == 3);
                                                            0
fifo_cover_ent_4: cover property (@(posedge clk) disable iff(rst) number_of_current_entries == 4);
                                                            0
fifo_cover_ent_5: cover property (@(posedge clk) disable iff(rst) number_of_current_entries == 5);
                                                            0
fifo_cover_ent_6: cover property (@(posedge clk) disable iff(rst) number_of_current_entries == 6);
```

Fig .1. FIFO cover property code



	Name	Health	Radius	Time
	fifo_cover_ent_0	★ 10	1 @ clk	0s
	fifo_cover_ent_1	★ 10	2 @ clk	0s
	fifo_cover_ent_2	★ 10	3 @ clk	0s
	fifo_cover_ent_3	★ 10	4 @ clk	0s
	fifo_cover_ent_4	★ 10	5 @ clk	0s
	fifo_cover_ent_5	★ 10	5 @ clk	0s
	fifo_cover_ent_6	★ 10	4 @ clk	0s
✓	fifo_out_is_full			
✓	fifo_out_is_empty			

Fig .2. The result of cover property

B.

The number of 5 and 6 are covered is undesirable since we are using a 4-deep ring FIFO. The biggest number of number_of_current_entries should be 7.

C.

We check the waveform of the mistake in Fig .3. We can find that the number changed from 0 to 7, which is unexpected. However, we can easily find the problem. The size of the variable is 7, it must be 0-1 to get the 7. Also, we find that the out_is_empty is 1 when we start to read, which is a problem. Therefore, we find that the FIFO read when the FIFO is empty.

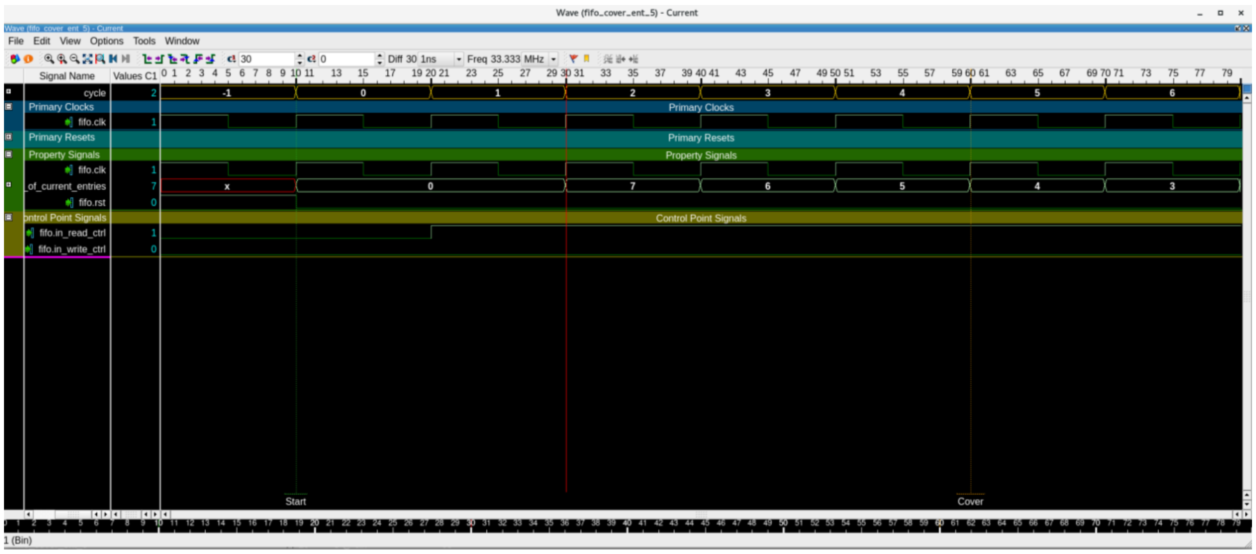
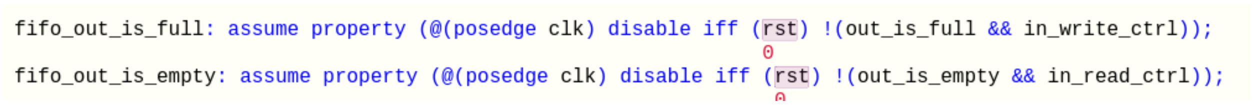


Fig .3. Wave form of Counterexample

D.

We add 2 assumptions in the design to prevent the read when FIFO is empty and the write when FIFO is full.



```
fifo_out_is_full: assume property (@(posedge clk) disable iff (rst) !(out_is_full && in_write_ctrl));
                                                                0
fifo_out_is_empty: assume property (@(posedge clk) disable iff (rst) !(out_is_empty && in_read_ctrl));
                                                                 0
```

Fig .4. FiFO assumption

E, F, F, H, I

The bug still exists. The waveform is in Fig .5. We can find that the out_is_empty will enter a unexpected 0. We check the code and find that the last condition is unnecessary. The last condition will set the out_is_empty to 0 in an unexpected way. We comment the last condition in Figure 4 and rerun the verification. At last, we can finally get the expected result where the 5 and 6 is ucovered.

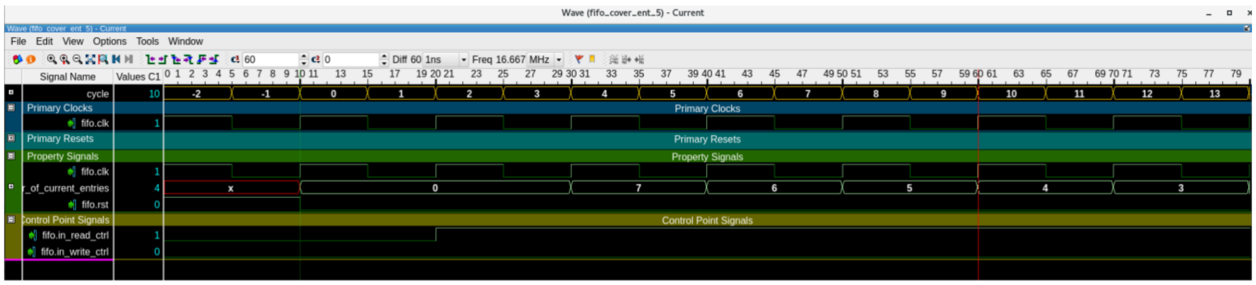
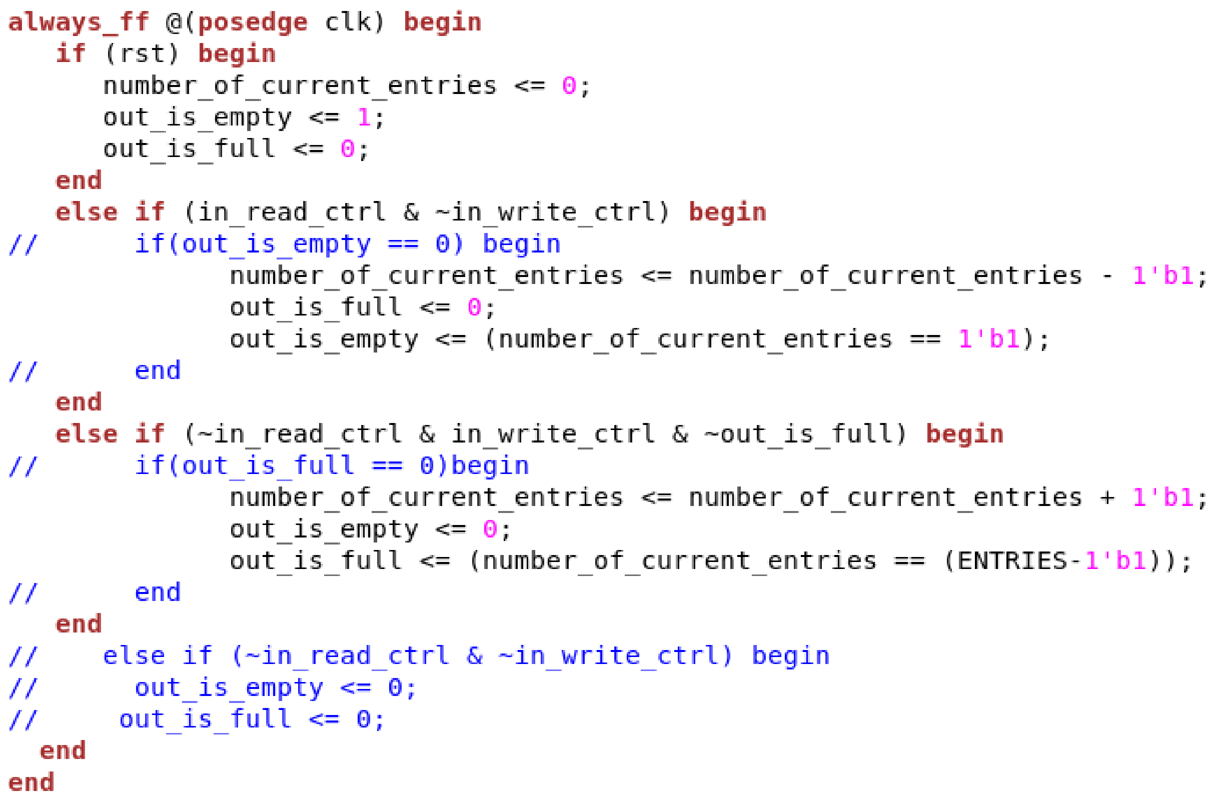
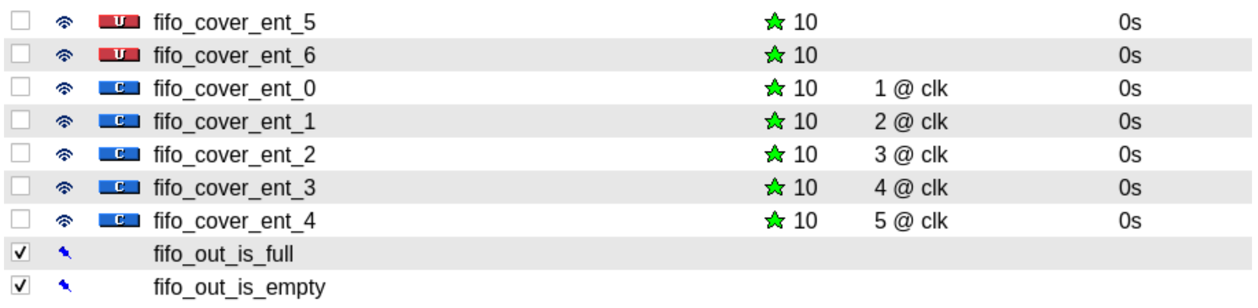


Fig .5. Waveform of Counterexample



```
always_ff @(posedge clk) begin
   if (rst) begin
      number_of_current_entries <= 0;
      out_is_empty <= 1;
      out_is_full <= 0;
   end
   else if (in_read_ctrl & ~in_write_ctrl) begin
//      if(out_is_empty == 0) begin
            number_of_current_entries <= number_of_current_entries - 1'b1;
            out_is_full <= 0;
            out_is_empty <= (number_of_current_entries == 1'b1);
//      end
   end
   else if (~in_read_ctrl & in_write_ctrl & ~out_is_full) begin
//      if(out_is_full == 0)begin
            number_of_current_entries <= number_of_current_entries + 1'b1;
            out_is_empty <= 0;
            out_is_full <= (number_of_current_entries == (ENTRIES-1'b1));
//      end
   end
//    else if (~in_read_ctrl & ~in_write_ctrl) begin
//      out_is_empty <= 0;
//     out_is_full <= 0;
  end
end
```

Fig .6. Debug FIFO code

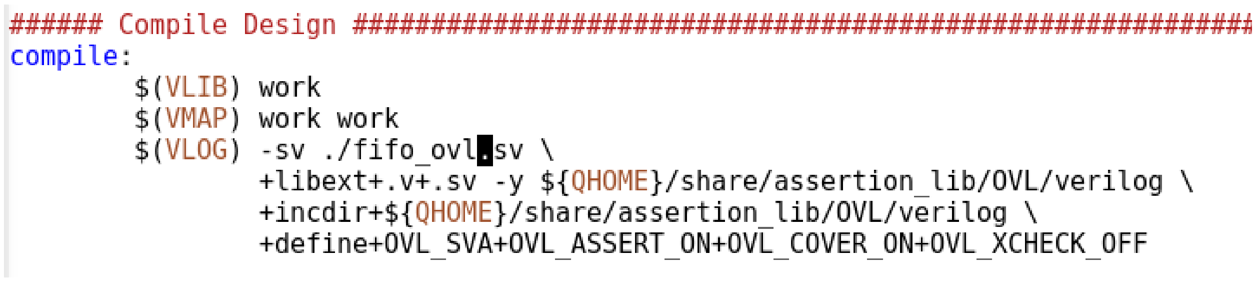


☐		U	fifo_cover_ent_5	★ 10		0s
☐		U	fifo_cover_ent_6	★ 10		0s
☐		C	fifo_cover_ent_0	★ 10	1 @ clk	0s
☐		C	fifo_cover_ent_1	★ 10	2 @ clk	0s
☐		C	fifo_cover_ent_2	★ 10	3 @ clk	0s
☐		C	fifo_cover_ent_3	★ 10	4 @ clk	0s
☐		C	fifo_cover_ent_4	★ 10	5 @ clk	0s
✓			fifo_out_is_full			
✓			fifo_out_is_empty			

Fig .7. Verification after debug

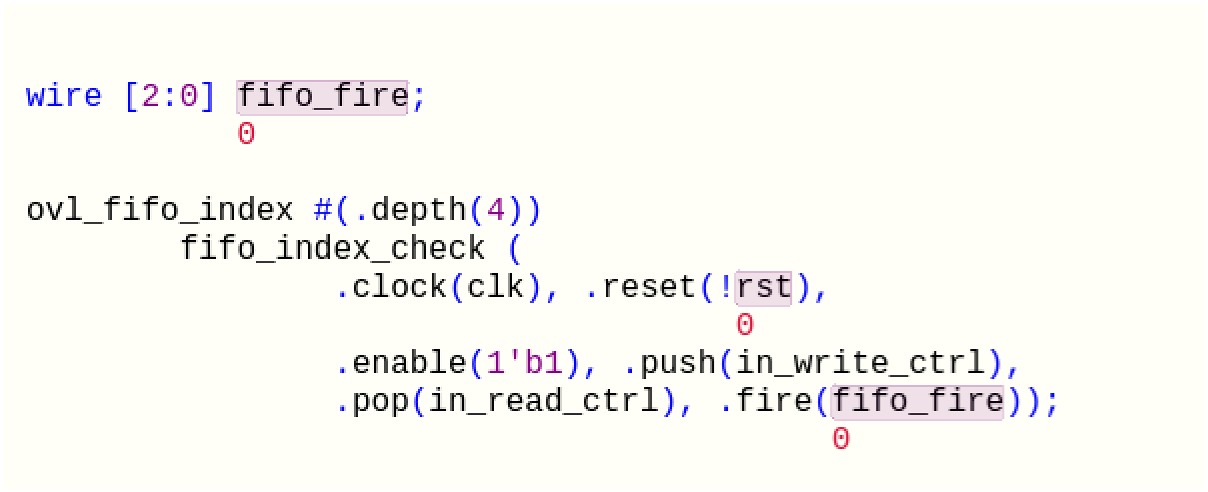
Task 2

We find the use of the ovl_FIFO_checker from the Path in the makefile. Than we add the path in the Makefile to the sv file as Fig .8. And we use the ovl_FIFO_index to write the check attributes in Fig .9.



```
###### Compile Design ##########################################################
compile:
	$(VLIB) work
	$(VMAP) work work
	$(VLOG) -sv ./fifo_ovl.sv \
		+libext+.v+.sv -y ${QHOME}/share/assertion_lib/OVL/verilog \
		+incdir+${QHOME}/share/assertion_lib/OVL/verilog \
		+define+OVL_SVA+OVL_ASSERT_ON+OVL_COVER_ON+OVL_XCHECK_OFF
```

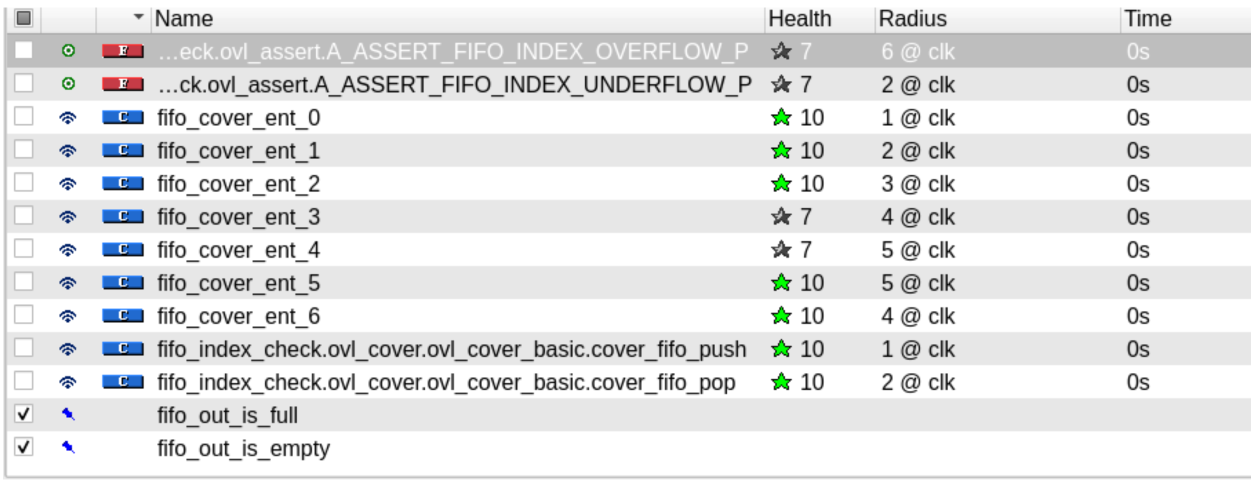
Fig .8 OVL Source Code



```
wire [2:0] fifo_fire;
           0

ovl_fifo_index #(.depth(4))
        fifo_index_check (
                .clock(clk), .reset(!rst),
                                     0
                .enable(1'b1), .push(in_write_ctrl),
                .pop(in_read_ctrl), .fire(fifo_fire));
                                          0
```

Fig .9 OVL Model Code

We run the verification and get the verification result in figure 10. Figure 11 shows the counterexample of the design. We know that the ovl_FIFO_index can only check the overflow and underflow of the design. We can find that in the 70ns of the design, the FIFO is empty while the read is still set to 1, which will bring out the same problem as task 1. We change the code as task 1 and get the final right answer in figure 12.



	Name	Health	Radius	Time
	…eck.ovl_assert.A_ASSERT_FIFO_INDEX_OVERFLOW_P	7	6 @ clk	0s
	…ck.ovl_assert.A_ASSERT_FIFO_INDEX_UNDERFLOW_P	7	2 @ clk	0s
	fifo_cover_ent_0	10	1 @ clk	0s
	fifo_cover_ent_1	10	2 @ clk	0s
	fifo_cover_ent_2	10	3 @ clk	0s
	fifo_cover_ent_3	7	4 @ clk	0s
	fifo_cover_ent_4	7	5 @ clk	0s
	fifo_cover_ent_5	10	5 @ clk	0s
	fifo_cover_ent_6	10	4 @ clk	0s
	fifo_index_check.ovl_cover.ovl_cover_basic.cover_fifo_push	10	1 @ clk	0s
	fifo_index_check.ovl_cover.ovl_cover_basic.cover_fifo_pop	10	2 @ clk	0s
✓	fifo_out_is_full			
✓	fifo_out_is_empty			

Fig .10 Result of OVL checking

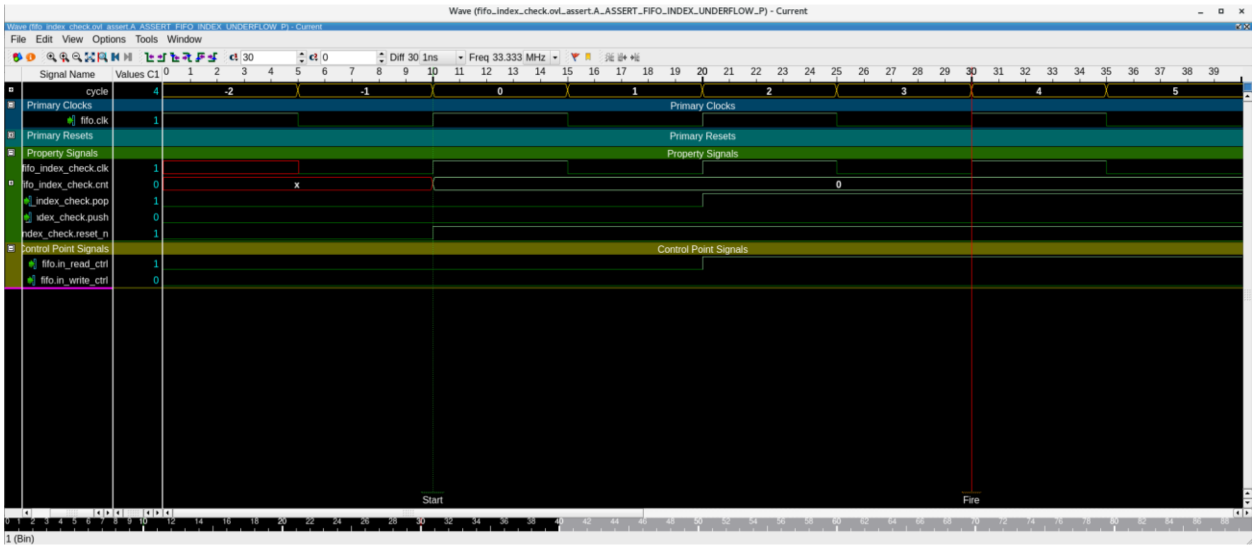
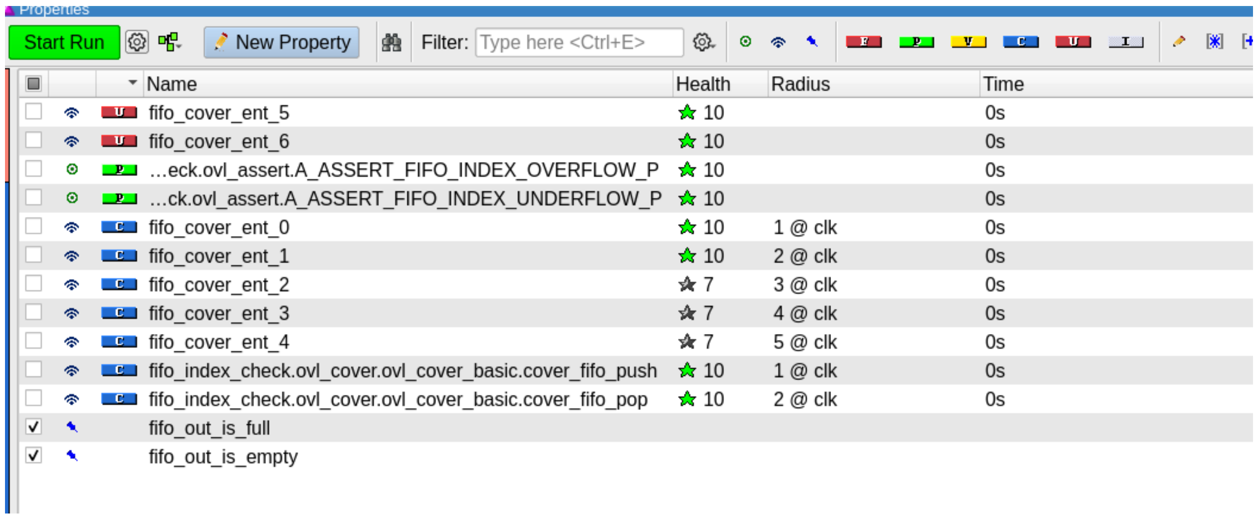


Fig .11 Waveform of Counterexample

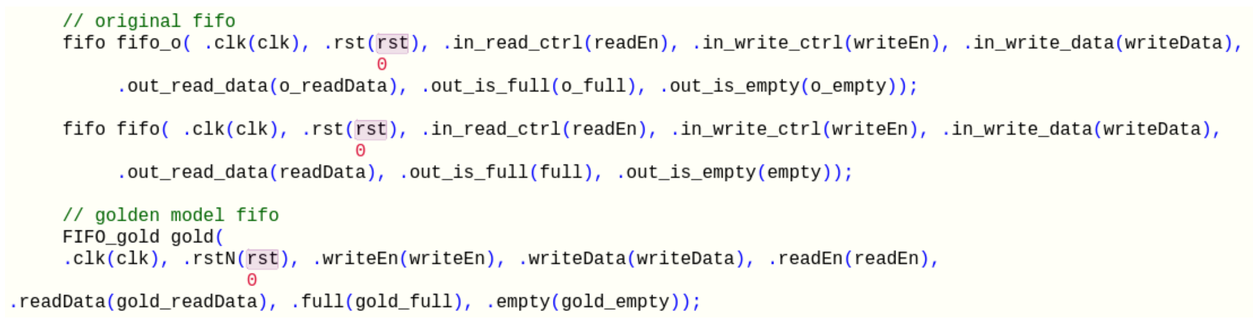


			Name	Health	Radius	Time
☐		U	fifo_cover_ent_5	★ 10		0s
☐		U	fifo_cover_ent_6	★ 10		0s
☐		P	...eck.ovl_assert.A_ASSERT_FIFO_INDEX_OVERFLOW_P	★ 10		0s
☐		P	...ck.ovl_assert.A_ASSERT_FIFO_INDEX_UNDERFLOW_P	★ 10		0s
☐		C	fifo_cover_ent_0	★ 10	1 @ clk	0s
☐		C	fifo_cover_ent_1	★ 10	2 @ clk	0s
☐		C	fifo_cover_ent_2	★ 7	3 @ clk	0s
☐		C	fifo_cover_ent_3	★ 7	4 @ clk	0s
☐		C	fifo_cover_ent_4	★ 7	5 @ clk	0s
☐		C	fifo_index_check.ovl_cover.ovl_cover_basic.cover_fifo_push	★ 10	1 @ clk	0s
☐		C	fifo_index_check.ovl_cover.ovl_cover_basic.cover_fifo_pop	★ 10	2 @ clk	0s
✓			fifo_out_is_full			
✓			fifo_out_is_empty			

Fig .12. Overall result of FIFO Verification

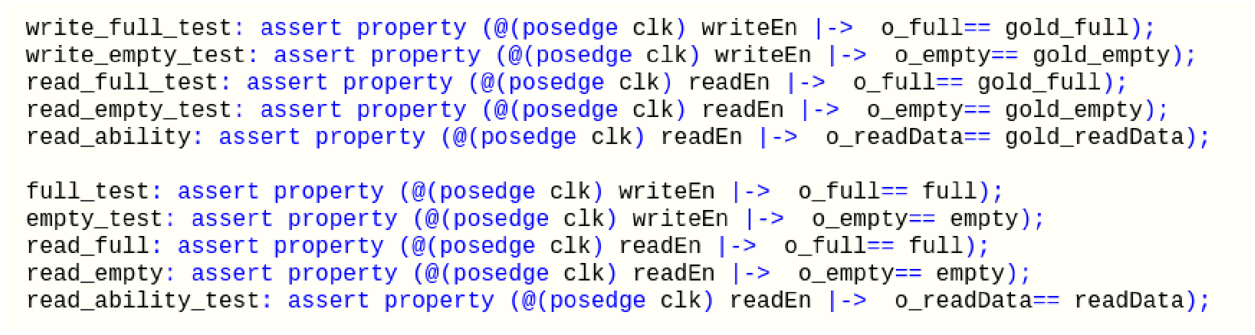
Task III

In task 3, we find a golden FIFO systemVerilog Online (FIFO, 2023). We module 3 different FIFO. The first 2 is defined using the Code provided by professor. The last 1 is defined in the golden code model. The code is in Fig .13. We checked the property in Fig .14, however, we found that the output of our original code and golden code is completely different. But our 2 module defined in the same code have the same output instead of readData. This is a big problem and we have not come up with the solution to deal with this problem. This problem should be further changed in the future. The result is in Fig .15.



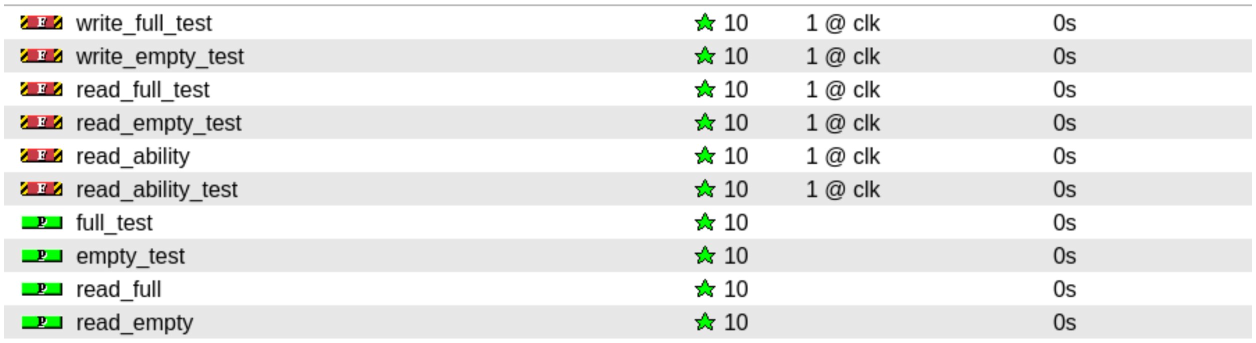
```
    // original fifo
    fifo fifo_o( .clk(clk), .rst(rst), .in_read_ctrl(readEn), .in_write_ctrl(writeEn), .in_write_data(writeData),
                                  0
        .out_read_data(o_readData), .out_is_full(o_full), .out_is_empty(o_empty));

    fifo fifo( .clk(clk), .rst(rst), .in_read_ctrl(readEn), .in_write_ctrl(writeEn), .in_write_data(writeData),
                                0
        .out_read_data(readData), .out_is_full(full), .out_is_empty(empty));

    // golden model fifo
    FIFO_gold gold(
    .clk(clk), .rstN(rst), .writeEn(writeEn), .writeData(writeData), .readEn(readEn),
                       0
.readData(gold_readData), .full(gold_full), .empty(gold_empty));
```

Fig .13. FIFO Definition Code



```
write_full_test: assert property (@(posedge clk) writeEn |->  o_full== gold_full);
write_empty_test: assert property (@(posedge clk) writeEn |->  o_empty== gold_empty);
read_full_test: assert property (@(posedge clk) readEn |->  o_full== gold_full);
read_empty_test: assert property (@(posedge clk) readEn |->  o_empty== gold_empty);
read_ability: assert property (@(posedge clk) readEn |->  o_readData== gold_readData);

full_test: assert property (@(posedge clk) writeEn |->  o_full== full);
empty_test: assert property (@(posedge clk) writeEn |->  o_empty== empty);
read_full: assert property (@(posedge clk) readEn |->  o_full== full);
read_empty: assert property (@(posedge clk) readEn |->  o_empty== empty);
read_ability_test: assert property (@(posedge clk) readEn |->  o_readData== readData);
```

Fig .14. Property assertion



F	write_full_test	★ 10	1 @ clk	0s
F	write_empty_test	★ 10	1 @ clk	0s
F	read_full_test	★ 10	1 @ clk	0s
F	read_empty_test	★ 10	1 @ clk	0s
F	read_ability	★ 10	1 @ clk	0s
F	read_ability_test	★ 10	1 @ clk	0s
P	full_test	★ 10		0s
P	empty_test	★ 10		0s
P	read_full	★ 10		0s
P	read_empty	★ 10		0s

Fig .15. Property Checking Result

RISC-V Formal Verification

Our formal verification design of RISC-V (https://github.com/NAvi349/riscv-proc/tree/main, 2023) came from Github in the SystemVerilog format. The structure of the RISC-V is shown in Fig .16. Our design mainly focuses on the data of the RISC-V. The datapath includes 5 stages: Instruction Fetch – MUX -ALU – Execution – Memory. To prove the success of this design, we will mainly focus on the functionality of each stage. Then, we will check the liveness property and safety property of the datapath to ensure the correctness of the data. The working procedure of the pipeline is that, the instruction Fetch will generate the instruction which is in control of the

MUX and ALU. When the instruction arrive at the MUX, the 3 to 1 mux will have change the 3 data based on the instruction and send it to ALU. The ALU will do the calculation in the instruction and send the output to the execution stage. Finally, the data will move to the memory from execution stage to memory stage.

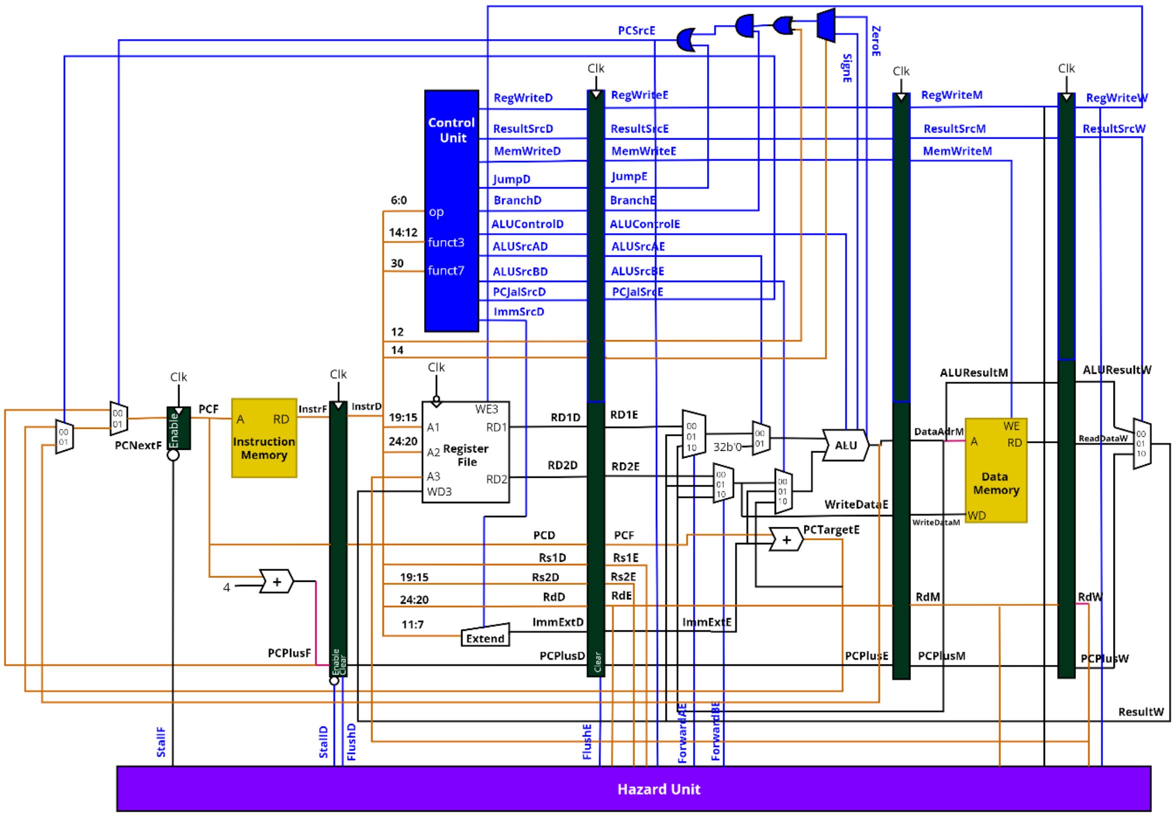


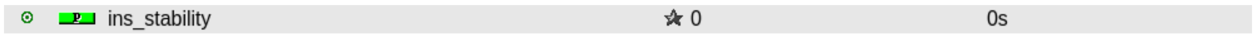
Fig .16. RISC-V pipeline

Instruction Fetch

Instruction checking

In the instruction Fetch stage, the most important task is that we need to ensure the instruction generation is successful. Also, we need to make sure that our definition of instruction is complete, which means that there will not be an instruction that the whole datapath could not understand. To deal with this problem, we use the safety property to check that the instruction will not over the limit of our instruction range.

```
ins_stability : assert property (@(posedge clk) ins |-> (ALUControl>=0 & ALUControl <=15));
```



```
⊙ P ins_stability                    ☆ 0          0s
```

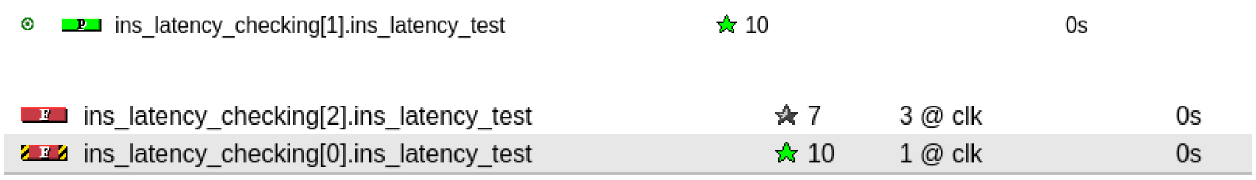
Fig .17. Test of instruction stability

The result in Fig .17 is pass, which means that the instruction is fully decoded, which means that we will not have an undetermined stage.

Instruction Latency Checking

We also need to check the instruction latency property. We want to check how many cycles the instruction will need to propagate after its generation. We need to do a lot of latency checking in the datapath to ensure the data moving.

```
genvar j;
    generate for (j=0; j<=2; j++)begin: ins_latency_checking
                          0
        ins_latency_test : assert property (@(posedge clk) ((clear&enable) |-> ##j (IF.InstrD == 0)));
    end
    endgenerate
```



```
⊙ P ins_latency_checking[1].ins_latency_test        ☆ 10                   0s

F ins_latency_checking[2].ins_latency_test              ☆ 7      3 @ clk          0s
F ins_latency_checking[0].ins_latency_test              ☆ 10     1 @ clk          0s
```

Fig .18. Test of Instruction Latency Checking

The result shows that the latency of the instruction is 1 cycle. And the other 2 instructions latency are all failed. We can learn from this result that the instruction will have a 1 clock cycle latency.

Mux (OVL Checking)

The functionality of our mux in this design can be illustrated in Fig .19. The Mux in this datapath has the functionality of choosing the data input ALU. The data will come from the memory, the execution stage, or the ALU. The ALU will enable us to do some complex operations such as multiplication and division.

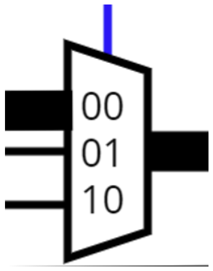
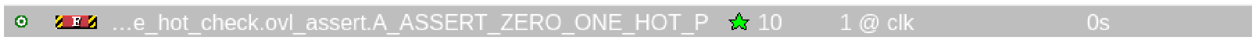


Fig .19. The Structure of Mux

OVL, open verification library, is a open library, which provides us with a lot of function which enables us to do the formal verification much more easily. We have used the ovl_fifo in the FIFO checking. Now, we found that the Mux structure is perfect for using the ovl_zero_one_hot. We use the ovl_zero_one_hot to check the property and get the result.

```
// Mux checking
ovl_zero_one_hot #(.width(2))
    ovl_zero_one_hot_check (
            .clock(clk), .reset(!rst),
                                 0
            .enable(1'b1), .test_expr(mux_select), .fire(fire_mux));
```



```
...e_hot_check.ovl_assert.A_ASSERT_ZERO_ONE_HOT_P   10   1 @ clk   0s
```

Fig .20. Output of OVL Result

We found that our result is wrong, which is quite clearly because our input instruction can be any number. The input of the mux might be 11 since the overall of the input instruction. To change this, we want to use the assumption to limit the input instruction so that it will not have the situation that the input of MUX is 11. We add the assumption and run the formal verification again. However, we found that the assumption is not able to constraint the output of the file. The output is still wrong. When we check the waveform generated by the formal verification tools, we found that the input instruction still have the unwilling input. This is a problem we should deal with in the future.

```
ovl_check: assume property (@(posedge clk) insF[8:7] != 11);
                                           0->1 0
```

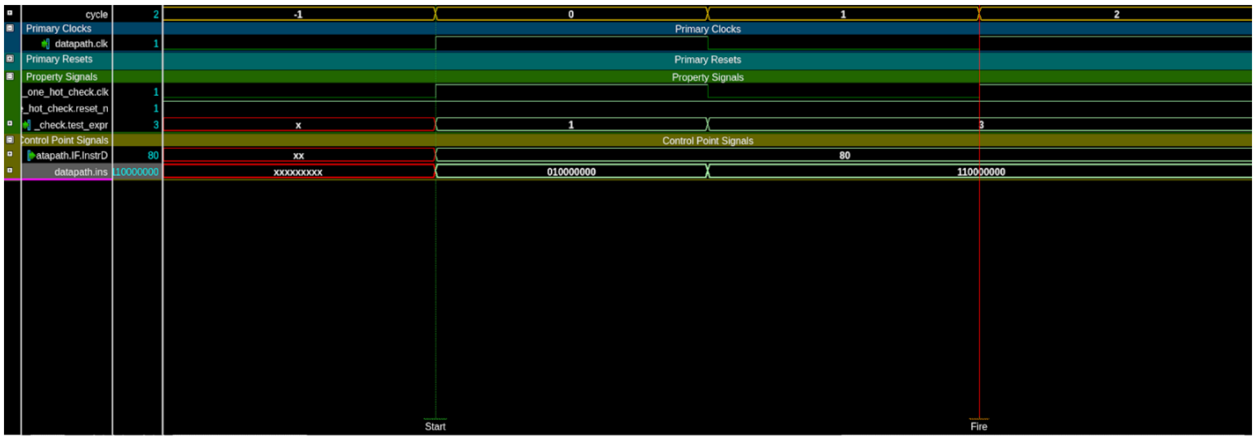


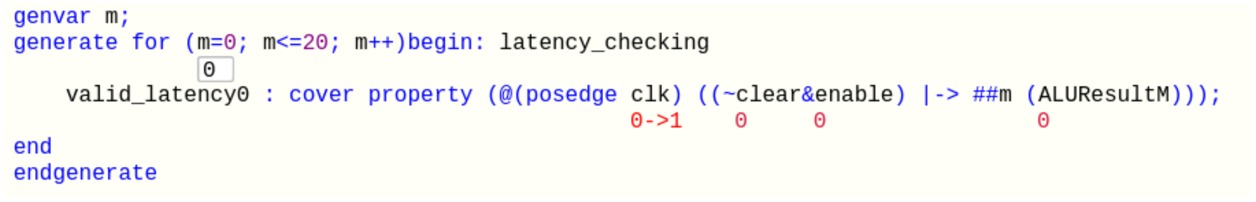
Fig .21. Waveform of the unwilling Data of Instruction

ALU

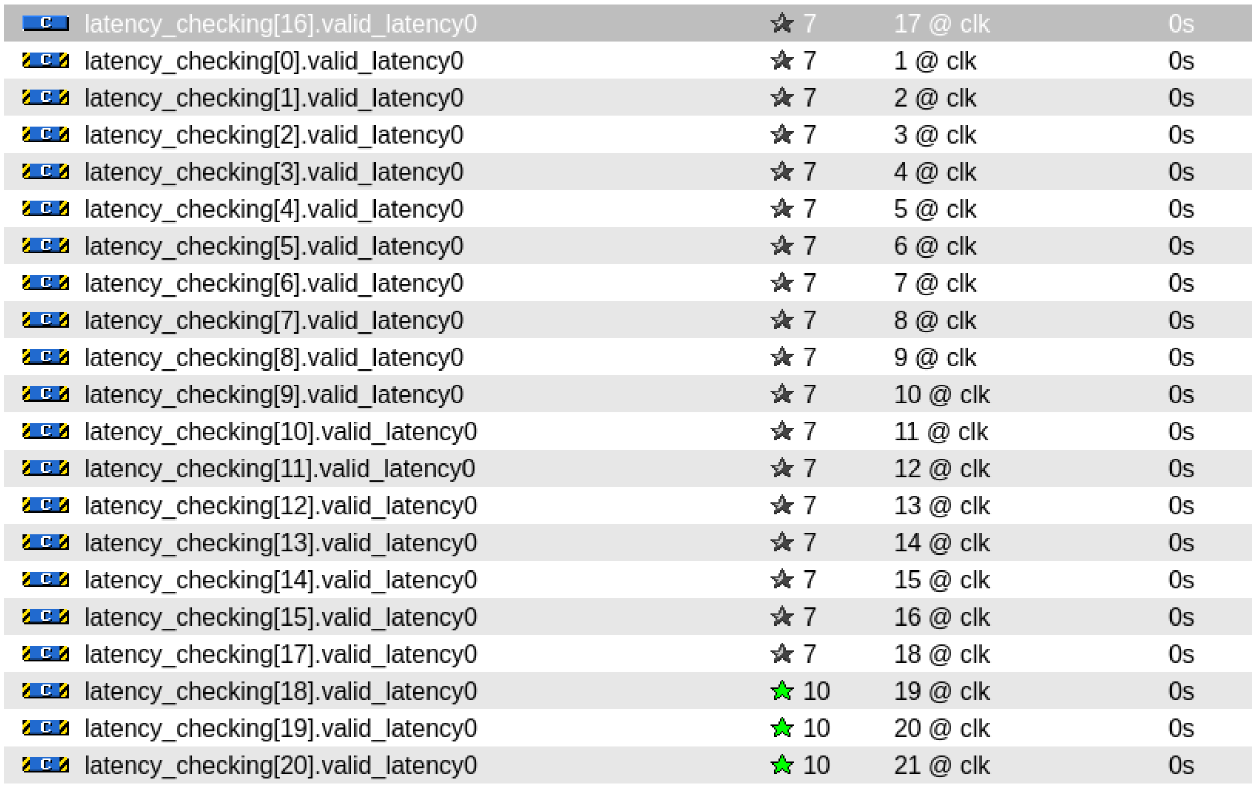
ALU is the most important part in this ALU, which is also the most complicated part. We need to check many aspects of the ALU to ensure that the ALU is not only functional and do the expected arithmetic operation.

ALU Valid Checking

We first do the ALU valid Checking of the ALU. We want to know that how many cycles the ALU needed to generate the output in the memory. We use the cover property to check the dataflow and check if it will go the memory when the instruction is enable.



```
genvar m;
generate for (m=0; m<=20; m++)begin: latency_checking
               0
    valid_latency0 : cover property (@(posedge clk) ((~clear&enable) |-> ##m (ALUResultM)));
                                                 0->1    0      0                 0
end
endgenerate
```



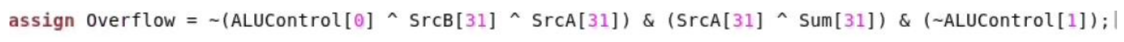
latency_checking[16].valid_latency0	★ 7	17 @ clk	0s
latency_checking[0].valid_latency0	★ 7	1 @ clk	0s
latency_checking[1].valid_latency0	★ 7	2 @ clk	0s
latency_checking[2].valid_latency0	★ 7	3 @ clk	0s
latency_checking[3].valid_latency0	★ 7	4 @ clk	0s
latency_checking[4].valid_latency0	★ 7	5 @ clk	0s
latency_checking[5].valid_latency0	★ 7	6 @ clk	0s
latency_checking[6].valid_latency0	★ 7	7 @ clk	0s
latency_checking[7].valid_latency0	★ 7	8 @ clk	0s
latency_checking[8].valid_latency0	★ 7	9 @ clk	0s
latency_checking[9].valid_latency0	★ 7	10 @ clk	0s
latency_checking[10].valid_latency0	★ 7	11 @ clk	0s
latency_checking[11].valid_latency0	★ 7	12 @ clk	0s
latency_checking[12].valid_latency0	★ 7	13 @ clk	0s
latency_checking[13].valid_latency0	★ 7	14 @ clk	0s
latency_checking[14].valid_latency0	★ 7	15 @ clk	0s
latency_checking[15].valid_latency0	★ 7	16 @ clk	0s
latency_checking[17].valid_latency0	★ 7	18 @ clk	0s
latency_checking[18].valid_latency0	★ 10	19 @ clk	0s
latency_checking[19].valid_latency0	★ 10	20 @ clk	0s
latency_checking[20].valid_latency0	★ 10	21 @ clk	0s

Fig .22. Outcome of latency checking

This is just a roughly checking. We will check this property again in our liveness property. However, we can still found from this result that we still need to deal with the warning. Most of the warning happen because the reset and dataflow is not clear. So a lot of data is empty in this situation. We still need to put much more limitation on it, which will be done in the liveness property.

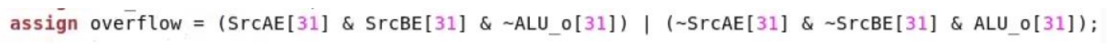
Overflow verification

In our RISC-V code, we noticed that there is a sub-module called signed adder in our ALU block. Also, there is an overflow sign bit in that code. We noticed that it is written in RTL code by:



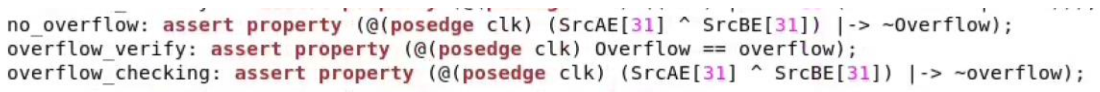
```
assign Overflow = ~(ALUControl[0] ^ SrcB[31] ^ SrcA[31]) & (SrcA[31] ^ Sum[31]) & (~ALUControl[1]);
```

This is a little bit strange since we know that the overflow bit should be related to the carry of the highest and second highest positions in mathematics. So that must be related to both two operands of adders. But we can easily see from the above code that it just related to one operand A in the right side of the and function. Based on this, we write a new overflow function ourselves to see if the overflow bit is matched.



```
assign overflow = (SrcAE[31] & SrcBE[31] & ~ALU_o[31]) | (~SrcAE[31] & ~SrcBE[31] & ALU_o[31]);
```

This overflow bit we write is given by whether it is a negative number when adding two positive signed numbers or appear a positive number when adding two negative signed numbers, which is the mathematical definition of overflow.



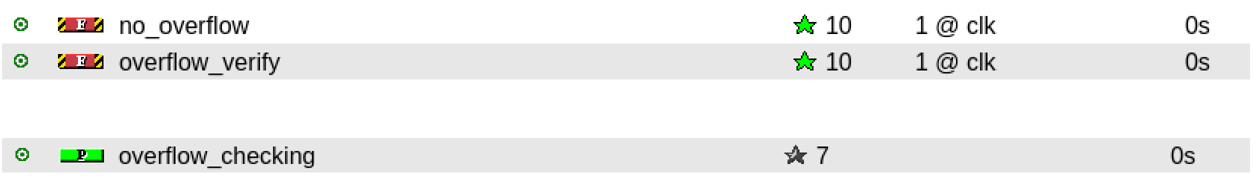
```
no_overflow: assert property (@(posedge clk) (SrcAE[31] ^ SrcBE[31]) |-> ~Overflow);
overflow_verify: assert property (@(posedge clk) Overflow == overflow);
overflow_checking: assert property (@(posedge clk) (SrcAE[31] ^ SrcBE[31]) |-> ~overflow);
```

Then we make three assumptions to verify the code's correctness.

The first one is to verify the overflow function of the original bit.

The second one is to verify if the overflow function of two methods is always the same.

The third one is to finally make sure our method is correct to verify overflow.



F	no_overflow	10	1 @ clk	0s
F	overflow_verify	10	1 @ clk	0s
P	overflow_checking	7		0s

Fig .23. Overflow Verification

We can see from this picture that our new method to test overflow of ALU is always true, but the result of the 2 methods is not always the same. Also, the original is wrong in testing overflow and we need to change it into our correct new one.

Below is a counter example given by FVT.

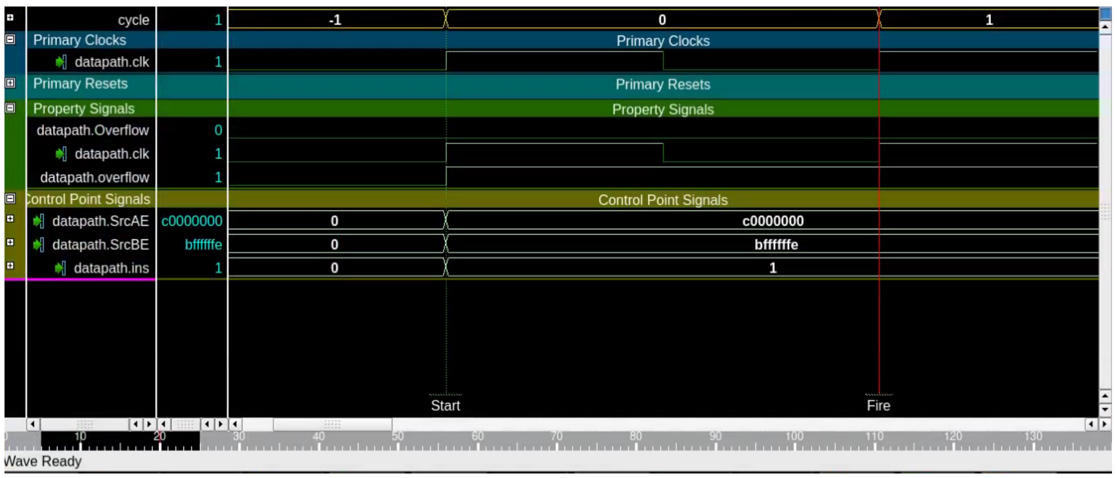


Fig .24. Fire Waveform

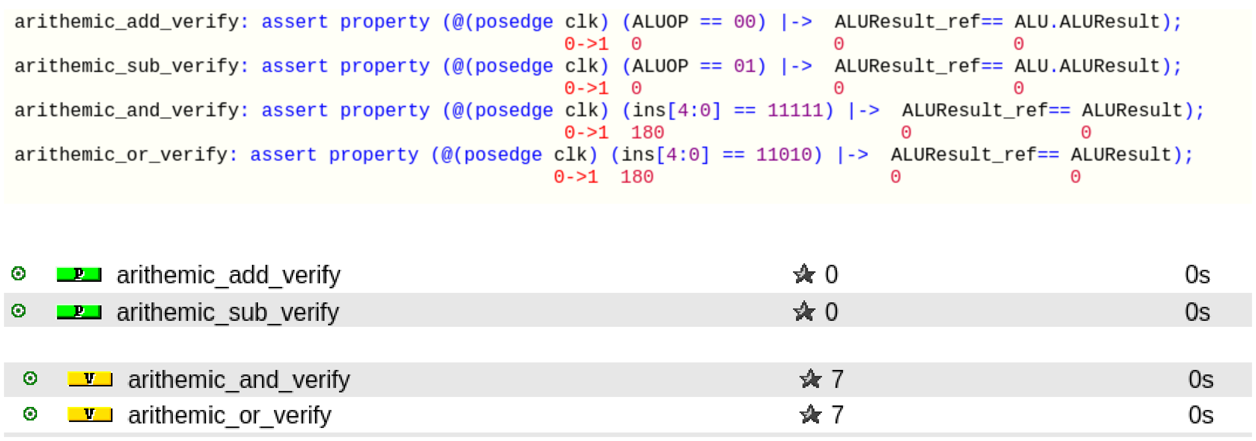
We can see from the graph that when two signed inputs of adder are 0xc0000000 and 0xbffffffe, the original code method gave us a 0 which means there is no overflow, while the new code we write ourselves gave us a 1 which means there might exist overflow. We can simply verify that c is 1100, b = 1011. There is no carry-in from the former 7bits of the 2 hexadecimal inputs so we can just focus on b plus c. For the 1100 plus 1011 should be 0111 with the first bit being directly truncated. In this case, two negative number add together and get a positive result so overflow should be 1. This prove that our new code about overflow is correct and the original one of RISC-V is wrong. The reason might be the designer have not thought about testing the function of adder in signed number or just write a too simple testbench to prove it work. This manifests the value of the significance of using formal verification.

ALU functionality (Reference Code Equivalence Checking)

In the ALU functionality checking, we will use a golden code which only contains the basic arithmetic operation. We will check the input and output of the two model to verify if they will have the same output when they have the same input and doing the same operation. The following is the code for our reference ALU.

```
module alu_reference(input logic [31:0] SrcA,
                        input logic [31:0] SrcB,
                        input logic [1:0] ALUControl ,
                        output logic  [31:0] ALUResult,
                        output logic Zero, Sign, Overflow);

//logic [31:0] Sum;

//assign Sum = SrcA + (ALUControl[0] ? ~SrcB : SrcB) + ALUControl[0];  // sub using 1's complement
//assign Overflow = ~(ALUControl[0] ^ SrcB[31] ^ SrcA[31]) & (SrcA[31] ^ Sum[31]) & (~ALUControl[1]);
assign Overflow = (SrcA[31] & SrcB[31] & ~ALUResult[31]) | (~SrcA[31] & ~SrcB[31] & ALUResult[31]);

assign Zero = ~(|ALUResult);
assign Sign = ALUResult[31];


always_comb
                casex (ALUControl)
                                2'b00: ALUResult = SrcA + SrcB;
                                2'b01: ALUResult = SrcA - SrcB; // and
                                2'b10: ALUResult = SrcA | SrcB; // or
                                2'b11: ALUResult = SrcA & SrcB; // sll, slli
                                default: ALUResult = 32'bx;
                endcase


endmodule
```

We make some assertions to check the output of the reference code and our original code.



```
arithemic_add_verify: assert property (@(posedge clk) (ALUOP == 00) |->  ALUResult_ref== ALU.ALUResult);
                                                     0->1  0               0                0
arithemic_sub_verify: assert property (@(posedge clk) (ALUOP == 01) |->  ALUResult_ref== ALU.ALUResult);
                                                     0->1  0               0                0
arithemic_and_verify: assert property (@(posedge clk) (ins[4:0] == 11111) |->  ALUResult_ref== ALUResult);
                                                     0->1  180               0                0
arithemic_or_verify: assert property (@(posedge clk) (ins[4:0] == 11010) |->  ALUResult_ref== ALUResult);
                                                    0->1  180               0                0
```

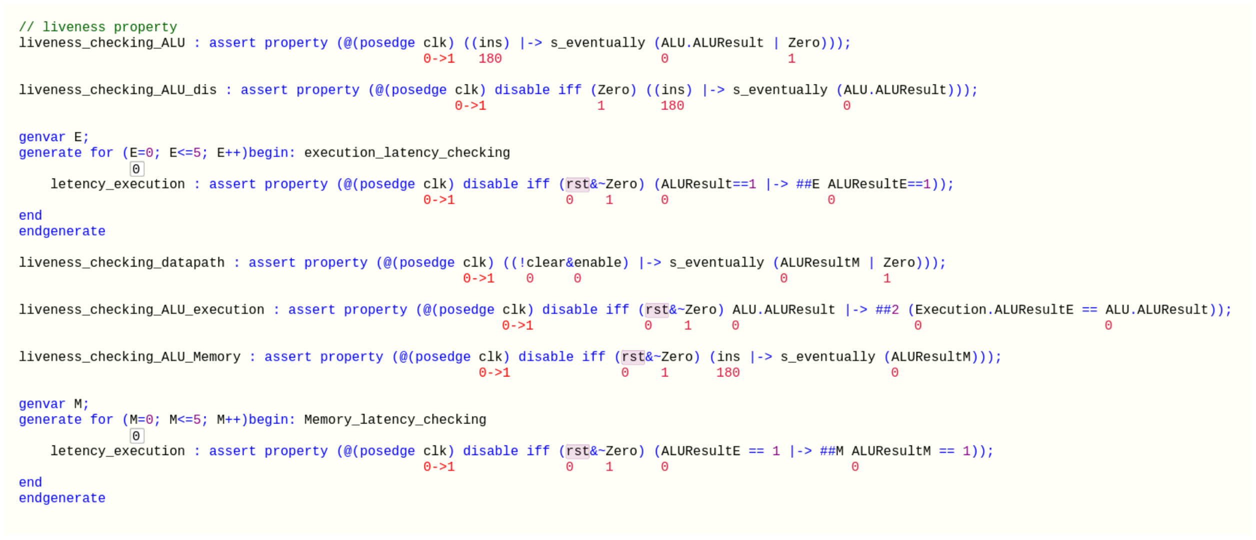
```
P  arithemic_add_verify     ★ 0    0s
P  arithemic_sub_verify     ★ 0    0s

V  arithemic_and_verify     ★ 7    0s
V  arithemic_or_verify      ★ 7    0s
```

Fig .25. Result of Equivalence Checking

We can found from the Fig . . that the add and subtraction is totally right and pass the verification. However, the And and Or operation is only vacuous proof. We think of the reason and found that it is hard for us to control the data of instruction. We might to strictly limit the number of the instruction that FVT cannot find the example to prove our property. FVT cannot also found the counter example, which make it vacuous proof. We still need to deal with it in the future.

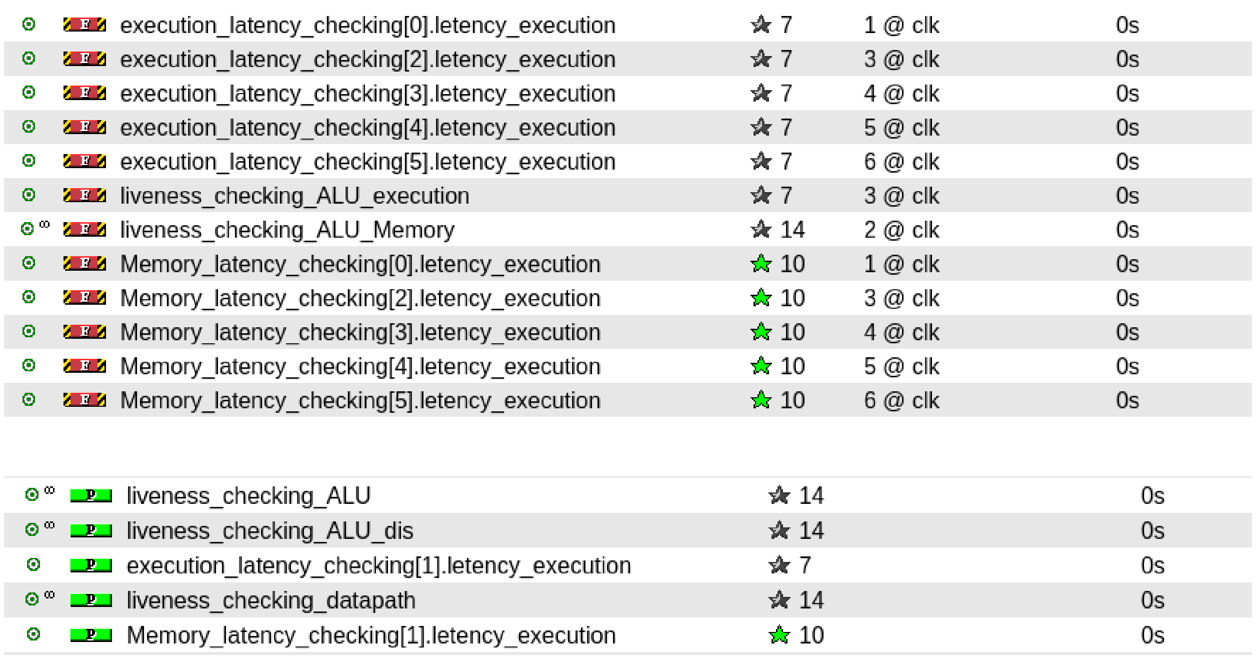
Liveness Property Checking

In this part, we will check the data flow of the whole design. We have learned a lot from this liveness property checking. We learn how to use the disable iff to set the unwilling data to 0. We also learn how to use the generate and genvar to do loop verification. In this part, we left a lot of property which might not be right. We just want to put here to remind us of how to correctly assert the assumption.



```
// liveness property
liveness_checking_ALU : assert property (@(posedge clk) ((ins) |-> s_eventually (ALU.ALUResult | Zero)));
                                                   0->1   180                    0               1

liveness_checking_ALU_dis : assert property (@(posedge clk) disable iff (Zero) ((ins) |-> s_eventually (ALU.ALUResult)));
                                                       0->1              1       180                    0

genvar E;
generate for (E=0; E<=5; E++)begin: execution_latency_checking
               0
    letency_execution : assert property (@(posedge clk) disable iff (rst&~Zero) (ALUResult==1 |-> ##E ALUResultE==1));
                                                   0->1              0    1      0                    0
end
endgenerate

liveness_checking_datapath : assert property (@(posedge clk) ((!clear&enable) |-> s_eventually (ALUResultM | Zero)));
                                                        0->1    0     0                         0            1

liveness_checking_ALU_execution : assert property (@(posedge clk) disable iff (rst&~Zero) ALU.ALUResult |-> ##2 (Execution.ALUResultE == ALU.ALUResult));
                                                             0->1              0    1     0                      0                       0

liveness_checking_ALU_Memory : assert property (@(posedge clk) disable iff (rst&~Zero) (ins |-> s_eventually (ALUResultM)));
                                                          0->1              0    1      180                   0

genvar M;
generate for (M=0; M<=5; M++)begin: Memory_latency_checking
               0
    letency_execution : assert property (@(posedge clk) disable iff (rst&~Zero) (ALUResultE == 1 |-> ##M ALUResultM == 1));
                                                   0->1              0    1      0                       0
end
endgenerate
```

We have the output in the following image.



⊙	F	execution_latency_checking[0].letency_execution	☆ 7	1 @ clk	0s
⊙	F	execution_latency_checking[2].letency_execution	☆ 7	3 @ clk	0s
⊙	F	execution_latency_checking[3].letency_execution	☆ 7	4 @ clk	0s
⊙	F	execution_latency_checking[4].letency_execution	☆ 7	5 @ clk	0s
⊙	F	execution_latency_checking[5].letency_execution	☆ 7	6 @ clk	0s
⊙	F	liveness_checking_ALU_execution	☆ 7	3 @ clk	0s
⊙ ∞	F	liveness_checking_ALU_Memory	☆ 14	2 @ clk	0s
⊙	F	Memory_latency_checking[0].letency_execution	★ 10	1 @ clk	0s
⊙	F	Memory_latency_checking[2].letency_execution	★ 10	3 @ clk	0s
⊙	F	Memory_latency_checking[3].letency_execution	★ 10	4 @ clk	0s
⊙	F	Memory_latency_checking[4].letency_execution	★ 10	5 @ clk	0s
⊙	F	Memory_latency_checking[5].letency_execution	★ 10	6 @ clk	0s

⊙ ∞	P	liveness_checking_ALU	☆ 14	0s
⊙ ∞	P	liveness_checking_ALU_dis	☆ 14	0s
⊙	P	execution_latency_checking[1].letency_execution	☆ 7	0s
⊙ ∞	P	liveness_checking_datapath	☆ 14	0s
⊙	P	Memory_latency_checking[1].letency_execution	★ 10	0s

Fig .26. Result of Liveness Property Checking

We can learn from our final property checking that the dataflow latency from ALU to Execution and the latency from Execution to Memory all need a 1 cycle latency. We can learn from the liveness property that what is the correct way to define a liveness property. We also learn the use of s_eventually to check that our design will finally succeed, which is the liveness property.

Summary

We have successfully complete the formal verification of the datapath. Although there is some small part we still need to change in the future, this procedure give us a basic intuition on how to do the formal verification and how to do the analysis. We first check the functionality and property checking of each component in the datapath. Second, we check the safety and liveness properties of the dataflow, including the latency of each operation. We also use a lot of formal verification method in this design. For example, cover and assertion property to do the model checking, the assumption property, the OVL verification and the Reference code Equivalence checking. The project enable us to have a deeper understanding of the formal verification.

Reference

(https://circuitcove.com/design-examples-fifo/, 2023)

(https://github.com/NAvi349/riscv-proc/tree/main, 2023)

Appendix

datapath.sv : sv file for the RISC-V Datapath

datapath.pptx : Slice for the Presentation

Makefile: makefile for datapath.sv