Hardware Formal Verification

Coverage Closure and BugHunt Project Report

Phase III

Final Report

Yuxiang Chen(yc3096), Ao Li(al3483)

Columbia University

Fall 2016

TABLE OF CONTENTS

I. OVERVIEW

This report describes the third phase of the *Coverage Closure and BugHunt* Project. We aim to apply formal verification skills learned in the previous two phases to much complex designs using commercial tools Mentor Graphics Questa Formal and Cadence Jasper. The project is divided into three tasks, the first task is to formally verify two medium designs called fifo_transport_single.sv and fifo_transport_double.sv with the instantiation of fifo design from phase II using Questa Formal. The second task is to formally verify a ALU design with a comprehensive formal verification approach and carry out using the Questa Formal tool. The last task is to reproduce task 1 using Cadence Jasper. The report will thoroughly describe the approaches to each task, and the source code can be found from www.columbia.edu/~yc3096/fv.zip.

II. FIFO BUG HUNTING

In first task, we use cover properties and a set of assertions to formally verify two relatively large design, *fifo_transport_single.sv* and *fifo_transport_double.sv*, which use a combined read/write ctrl signal. The first DUT uses a single instantiation of the faulty *fifo.sv* design from phase II, and the second design uses two instantiations.

Part 1: fifo_transport_single.sv

The *fifo_transport_single* uses a single instantiation of the "faulty" fifo design and combined read/write signal. The combined signal *in_readwrite_ctrl* is decoded into *in_read_ctrl* and *in_write_ctrl*, and passed to the fifo instantiation. The assignments below ensures that only and at least one control signal will be set to high.

```
assign  in_read_ctrl = !in_readwrite_ctrl;
assign  in_write_ctrl = in_readwrite_ctrl;
```

Below is a set of coverage cases declared in the 4-deep, synchronous reset FIFO design(*fifo.sv*). Since the first design only have a depth of 4, we expect only from *fifo_num_entries_1* to *fifo_num_entries_4* to be covered.

```
// uncovered cases
fifo_num_entries_7: cover property (number_of_current_entries == 7);
fifo_num_entries_6: cover property (number_of_current_entries == 6);
fifo_num_entries_5: cover property (number_of_current_entries == 5);
// covered cases
fifo_num_entries_4: cover property (number_of_current_entries == 4);
fifo_num_entries_3: cover property (number_of_current_entries == 3);
fifo_num_entries_2: cover property (number_of_current_entries == 2);
fifo_num_entries_1: cover property (number_of_current_entries == 1);
```

Below is a set of assertions that declared in the 4-deep, synchronous reset FIFO design(*fifo.sv*). Our main purpose in designing these set of assertions is to find corner cases that can lead to FIFO overflow and underflow. We categorized the assertions to reset, full-condition, and empty-condition assertions, and we have three additional interesting FIFO assertions listed in the last category.

Synchronous reset assertion:

1. When the FIFO is reset, the **empty** flag should be set, **full** flag, **write_ptr**, **read_ptr**, and **num_of_current_entries** should all be set to zero.

```
// Reset Assertions
// 1. when the FIFO is reset, the empty flag should be set, full flag,
//    write_ptr, read_ptr, and num_of_current_entries should all be set to zero.
reset_flags_ptrs: assert property (@(posedge clk)
        rst |=> (write_ptr == 0 && read_ptr == 0 && out_is_empty == 1 &&
        out_is_full == 0 && number_of_current_entries == 0));
```

FIFO full condition assertions:

2. When the FIFO is full, the **full** flag should be set to 1.
3. When the FIFO is full, the **empty** flag should be set to 0.
4. When the FIFO is has N-1 entries, and **write** occurs, then **full** flag should be set to 1.
5. When FIFO is full, the **write** control should be zero.

```
// FIFO full condition assertions:
// 2. when the FIFO is full, the full flag should be asserted
full_fifo_condition: assert property (@(posedge clk)
        disable iff(rst) (number_of_current_entries > (ENTRIES  - 1'b1)) |-> out_is_full);

// 3. when the FIFO is full, FIFO must not be empty
fifo_full_not_empty_condition: assert property (@(posedge clk)
        disable iff(rst)  out_is_full |-> !out_is_empty );

// 4. when the FIFO has N-1 entries and in_write_ctrl is asserted and in_read_ctrl is
//    not asserted, the full flag should be asserted
fifo_go_full_condition: assert property (@(posedge clk)
        disable iff(rst) ((number_of_current_entries == (ENTRIES-1'b1)) &&
        (in_read_ctrl == 0)&&(in_write_ctrl == 1)) |=> out_is_full );

// 5. when the FIFO is full, in_write_ctrl should be 0
fifo_full_no_write_condition: assert property (@(posedge clk)
        disable iff (rst) out_is_full |-> !in_write_ctrl );
```

FIFO empty condition assertions:

6. When the FIFO is empty, the **empty** flag should be 1.
7. When the FIFO is empty, the **full** flag should be set to 0.
8. When the FIFO is has 1 entry, and **read** occurs, then **empty** flag should be set to 1.
9. When FIFO is empty, the **read** control should be zero.

```
// FIFO empty condition assertions:
// 6. when the FIFO is empty, the empty flag should be asserted
empty_fifo_condition: assert property (@(posedge clk)
        disable iff(rst) (number_of_current_entries == 0) |-> out_is_empty);

// 7. when the FIFO is empty, FIFO must not be full
fifo_empty_not_full_condition: assert property (@(posedge clk)
        disable iff(rst)  out_is_empty |-> !out_is_full);

// 8. when the FIFO has only one entry and in_read_ctrl is asserted and in_write_ctrl is
//    not asserted, the empty flag should be asserted
fifo_go_empty_condition: assert property (@(posedge clk)
        disable iff(rst) ((number_of_current_entries == 1)&&(in_read_ctrl == 1)
        &&(in_write_ctrl == 0)) |=> out_is_empty );

// 9. when the FIFO is empty, in_read_ctrl should be 0
fifo_empty_no_read_condition: assert property (@(posedge clk)
        disable iff(rst) out_is_empty |-> !in_read_ctrl );
```

FIFO other cases

10. When the FIFO is NOT full, the **full** and **empty** flags should be set to 0.
11. Write pointer should not change list in_write_ctrl is 0.
12. Read pointer should not change whilst in_read_ctrl is 0.

```
// FIFO other cases
// 10. when the FIFO is neither full nor empty, the full flag and empty flag should not be asserted
fifo_no_empty_no_full_condition: assert property (@(posedge clk)
        disable iff(rst) (((number_of_current_entries > 0) &&
        (number_of_current_entries < ENTRIES))|-> (!out_is_full && !out_is_empty)));

// 11. write pointer should not change whilst in_write_ctrl is zero
fifo_no_write_ptr_change_condition: assert property (@(posedge clk)
        disable iff(rst)(!in_write_ctrl |=> $stable(write_ptr)));

// 12. read pointer should not change whilst in_read_ctrl is zero
fifo_no_read_ptr_change_condition: assert property (@(posedge clk)
        disable iff(rst)(!in_read_ctrl |=> $stable(read_ptr)));
```

We declared two assumptions to the top-level design(*fifo_transport_single*) to constrain the input control signal: *no push to (but pop out of) the queue whenever full flag set and no pop out of(but push to) the queue whenever empty flag set.* These two constraints are similar to the two assumptions added in phase II. It's important that we only applied the constraints to the top-level input, which is *in_readwrite_ctrl*.

```
default clocking c0 @(posedge clk); endclocking
fifo_assume_empty:  assume property (@(posedge clk)
                    (out_is_empty && ~out_is_full) |-> in_readwrite_ctrl);
fifo_assume_full: assume property  (@(posedge clk)
                    (~out_is_empty && out_is_full)|-> !in_readwrite_ctrl);
```

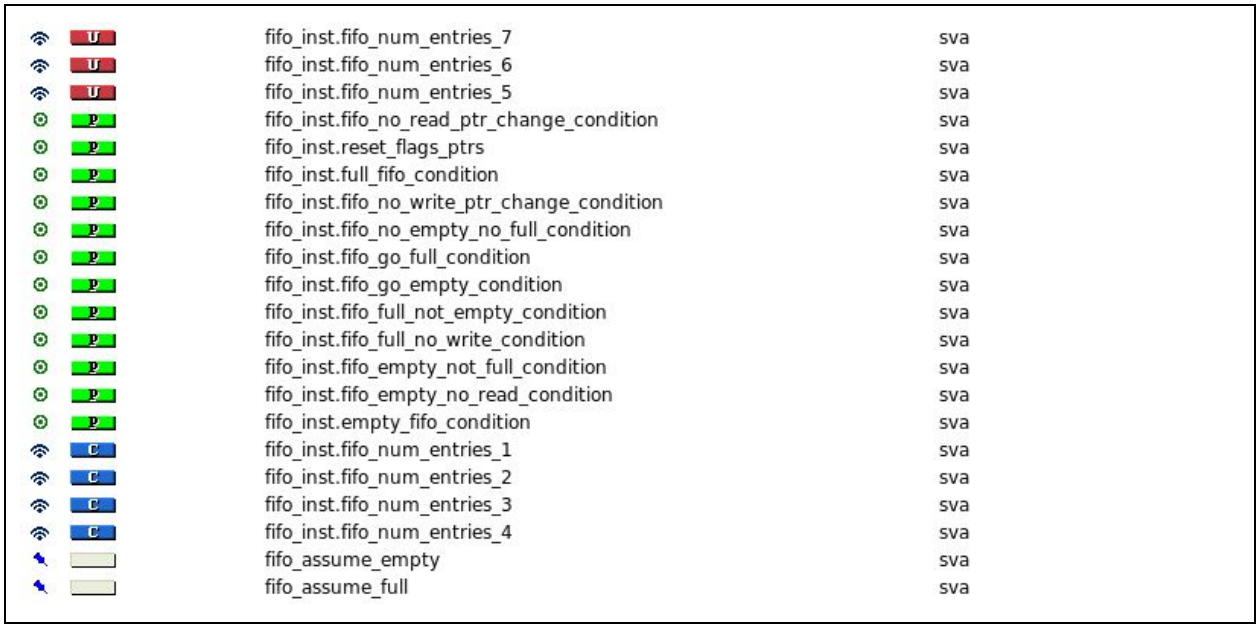
We set the FPV questa environment using the Makefile created previously with minor changes. A screenshot of the Makefile is shown below.

```
run: clean compile_sva formal debug

###### Define Variables #####################################################
VLIB = ${QHOME}/modeltech/plat/vlib
VMAP = ${QHOME}/modeltech/plat/vmap
VLOG = ${QHOME}/modeltech/plat/vlog
TOP_LEVEL = fifo_transport_single.sv.orig.sv
SECOND_LEVEL = fifo.sv
###### Compile Design ########################################################
compile_sva:
    $(VLIB) work
    $(VMAP) work work
    $(VLOG) ./src/vlog/${TOP_LEVEL}
    $(VLOG) ./src/vlog/${SECOND_LEVEL}
compile_ovl:
    $(VLIB) work
    $(VMAP) work work
    $(VLOG) -sv ./src/vlog/${TOP_LEVEL} \
        +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
###### Run Formal Analysis ###################################################
formal:
    qverify -c -od Output_Results -do "\
        do qs_files/directives.tcl; \
        formal compile -d fifo_transport_single\
            -target_cover_statements; \
        formal verify -init qs_files/design.init \
        -effort unlimited -timeout 5m; \
        exit"

###### Debug Results #########################################################
debug:
    qverify Output_Results/formal_verify.db &

###### Clean Data ############################################################
clean:
    qverify_clean
    \rm -rf work modelsim.ini *.wlf *.log replay* transcript *.db
    \rm -rf Output_Results *.tcl
```

To run the formal tool, we first

```
make run
```

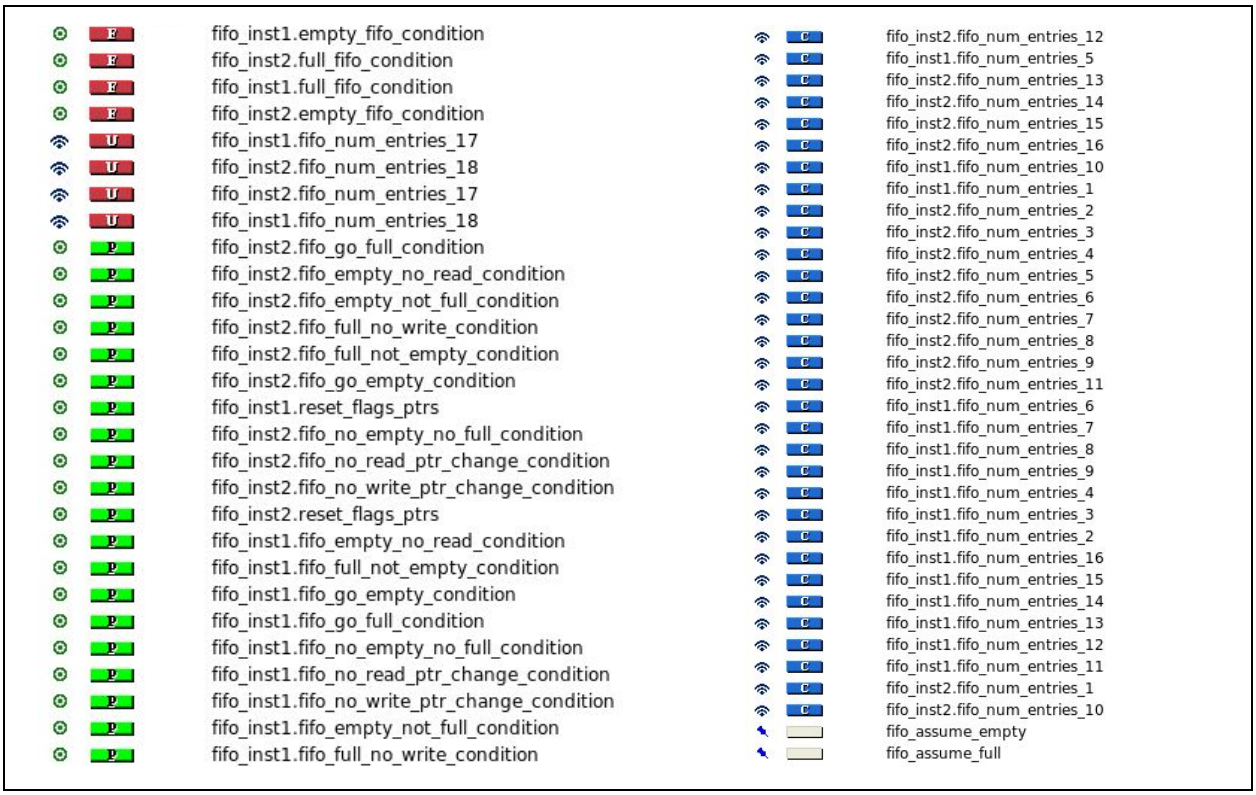
in command line, and the questa will clean the previous work, compile the design and run formal verification. The FPV result is shown as below. It's shown that all properties are either covered and proved even though we included the faulty FIFO. Note that the first three cases are uncovered because it's a 4-deep FIFO.



The first transport FIFO design is working despite the bug(in *fifo.sv*) because the top-level combined read/write control signal ensured that at least one of write and read control will be asserted high, and thus will never enter the faulty *else if* state discovered in phase II.

Part 2: fifo_transport_double.sv

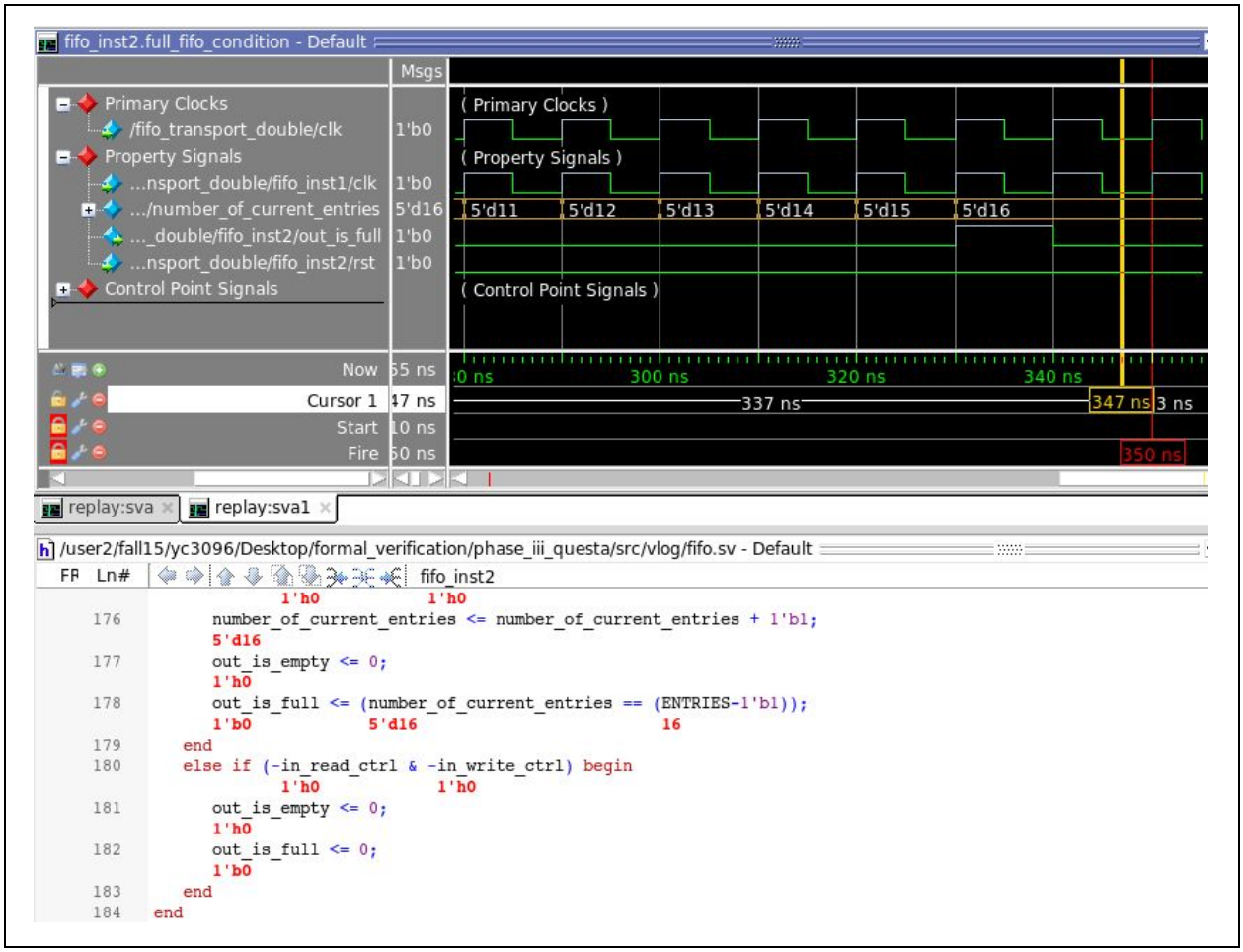
The second step is to verify *fifo_transport_double*, which uses two instantiation of the "faulty" 16-deep synchronized FIFO design and a combined read/write signal. The design concatenate the two FIFO and doubled the depth by adding additional control logic on the top-level. To verified the second design, we declared the same set of assertions from part 1, and same two assumptions applied to input(*in_readwrite_ctrl*) on the top-level design.



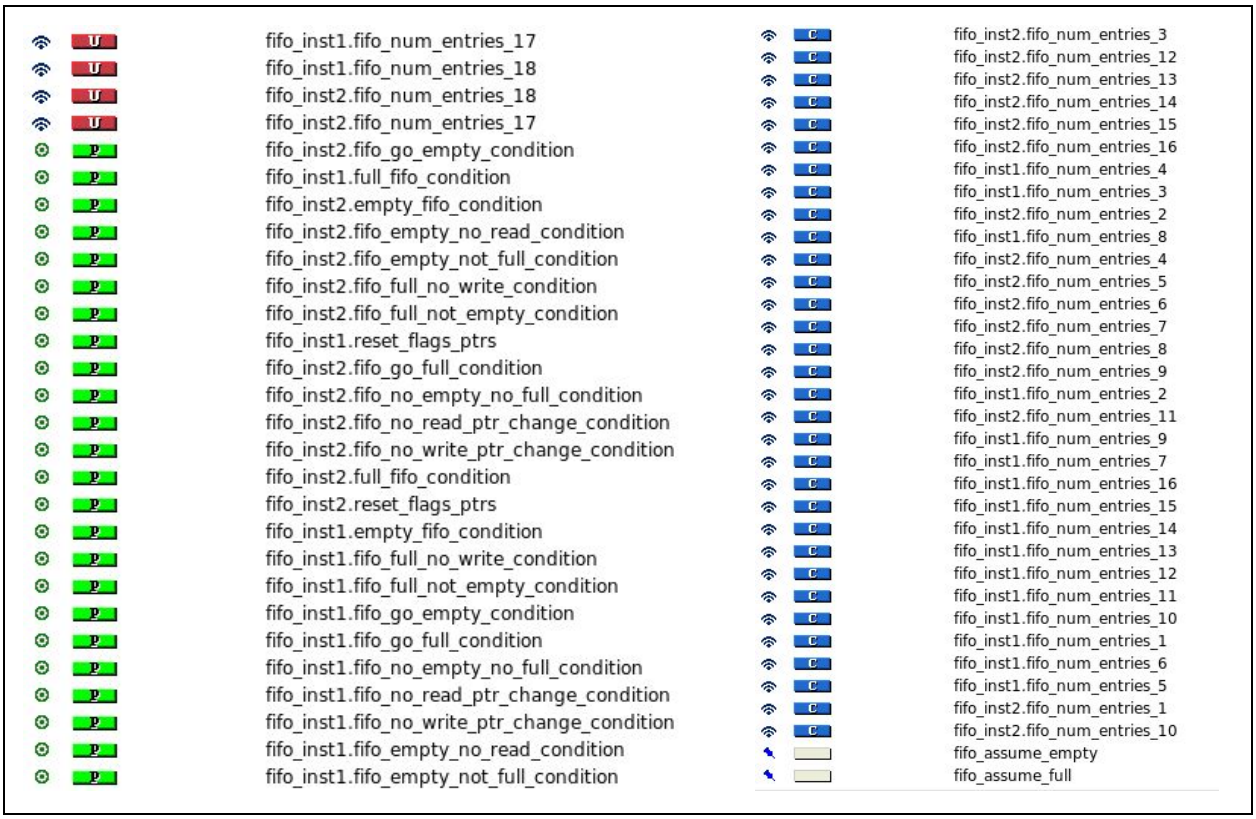
The screenshot above shows that the FPV found two firings for each FIFO. By analyzing the *full_fifo_condition* property, we found that the failure is caused when the small FIFO is full, and the write and read control signals to the submodule are both set to 0, thus the FIFO entered the unexpected faulty state while both *out_is_empty* and *out_is_full* are reset to 0.

The waveform visualizer below shows the counterexample illustrating that the *full* flag is not set when the FIFO is full. After reset, the write control is set for 16 cycles, and then the *full* flag is asserted, which caused the write control to pull down in the next cycle. Since the read and write control to the submodule is not directly generated from the *in_readwrite_ctrl* from the top level, it is possible that both of them set to 0, which leads to the last *else if* statement in the faulty FIFO. Therefore, the *full* flag is pulled down even though the FIFO is still full.

The another firing, *empty_fifo_condition* is also caused by this bug. After filling the FIFO, the read control asserted for a number of cycles until the FIFO is empty, empty flag asserted, and then both read and write control is set to 0, which also leads to the least *else if* statement in the faulty FIFO, so the *empty* flag pulled down even though the FIFO is still empty.



We removed the two assignment lines, re-verified the design using the same set of assertions, and now all assertions are proved as shown below. Note that *fifo_num_entries_17* and *fifo_num_entries_18* are uncovered, because both FIFO are 16-deep.



III. EFFECTIVE PFV FOR ALU VERIFICATION

In this task, we are going to dive into more advanced design verification techniques, bug hunting FPV, which aims to target complex scenarios and discover subtle corner cases that are unlikely to be hit through simulation. We are also going to explore practical issues in setting up an robust FPV environment and effectively using it on our model for bug hunting and full proof. We illustrate this process on an arithmetic logic unit(ALU) design example from the website *formalverificationbook.com*

Part A: Latency Check

Our first step is to check whether the latency of all operations in our top-level ALU design is fixed. As we scan through the RTL code, we found that the proper latency for majority cases are 3 clock cycles. Our formal approach is to add assertions to verify all operations(both logical and arithmetic) have a latency of 3 clock cycles.

Based on the given top-level RTL, we have to apply additional assumptions so that behaviors are restricted to checking operations of the choice. We add two assumptions exhibited as followings:

Assumptions

1. Assume the defeaturee bit is driven to be 1. When this bit is 0 and src2 is 0, the clock for arithmetic module is gated. As this bit will be driven to 1 in most real-lie usages, we can safely deprioritize cases where this bit is 0.

```
defeature_assume: assume property (@(posedge Clk) (defeature_addck));
```

2. Assume all feeding operations are valid and are either arithmetic or logical operation

```
legal_opcode: assume property(@(posedge Clk) valid_arithmetic_op (uopcode) || valid_logical_op(uopcode));
```

Coverage & Assertions

To check the latency of each operation, we first use cover property to check whether all potentially operations can be used in our environment. The FPV tool shows that our initial covers on these are successful.

```
genvar j;
generate for (j= OPADD;  j<= OPCMP; j++) begin: cover_all_operations
   cover_arithmetic: cover property (@(posedge Clk) ((uopcode == j) ##3 (resultv != 0) ) );
end
endgenerate

genvar k;
generate for (k= OPAND;  k<= OPXOR; k++) begin: logical_latency
   cover_logical: cover property (@(posedge Clk) ((uopcode == k) ##3 (resultv != 0) ) );
end
endgenerate
```

Then we assert properties to verify when valid operation and uopv arrive, a valid output signal appears in three cycles. The assertion is proved from formal tool.

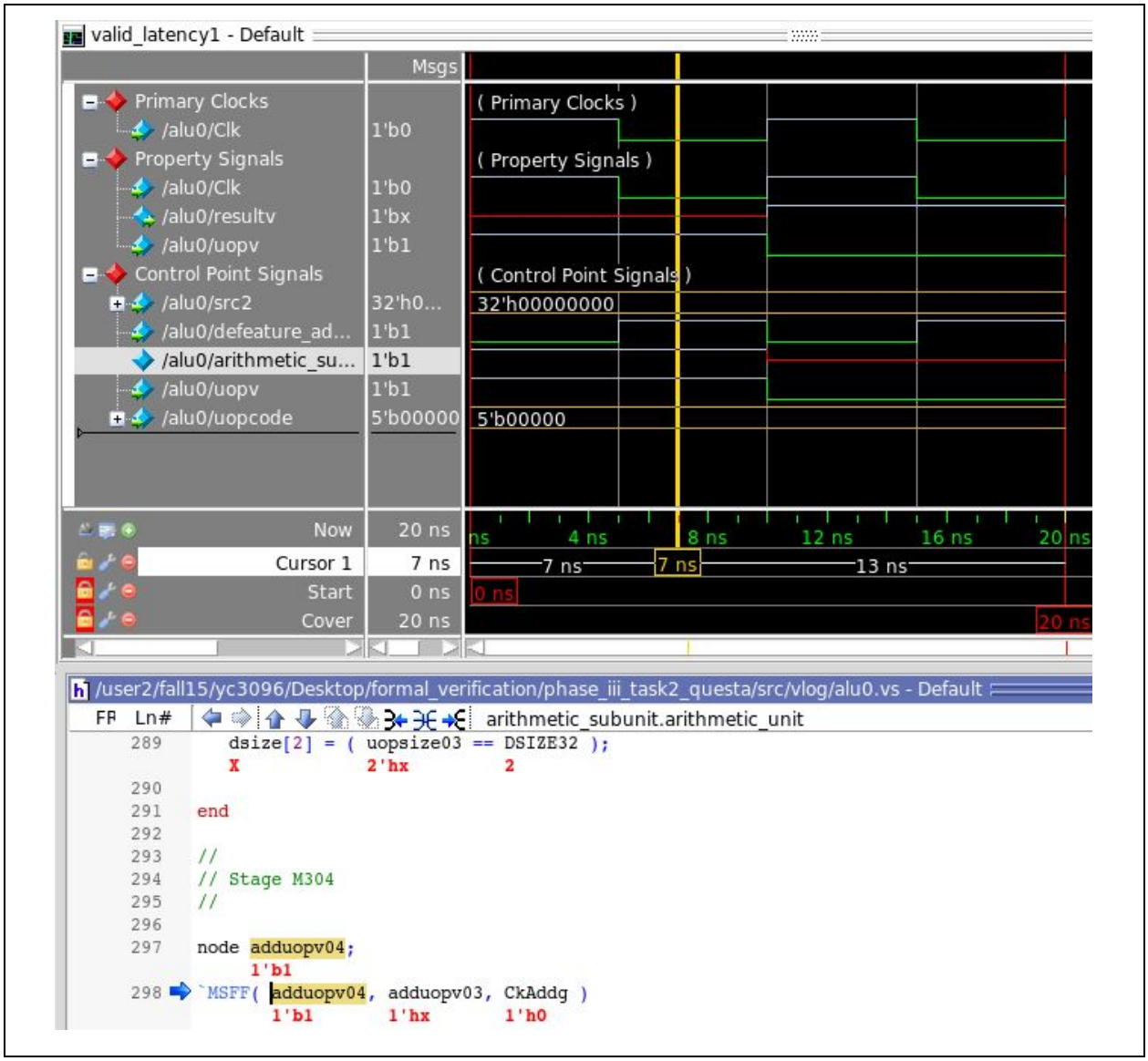
```
valid_latency3: assert property (@(posedge Clk) ((uopv) |-> ##3 resultv ));
```

We further investigated the possibilities by checking whether the valid result signal will appear after two clock cycles.

```
cover_latency1: cover property (@(posedge Clk) ((uopv) |-> ##1 resultv ));
cover_latency2: cover property (@(posedge Clk) ((uopv) |-> ##2 resultv ));
```

defeature_assume		sva
legal_opcode		sva
cover_all_operations[10].cover_arithmetic	C	sva
cover_all_operations[11].cover_arithmetic	C	sva
cover_all_operations[8].cover_arithmetic	C	sva
cover_all_operations[9].cover_arithmetic	C	sva
cover_latency1	C	sva
cover_latency2	C	sva
logical_latency[0].cover_logical	C	sva
logical_latency[1].cover_logical	C	sva
logical_latency[2].cover_logical	C	sva
valid_latency3	P	sva

After formal verifying, these two properties are covered, and the counterexample result is a bit surprising, as we don't expect *resultv* sets to high in two cycles. Bringing up the waveform, we are surprised to see that the *adduopv04* node connected to the flip-flop in arithmetic unit has been always high since reset, hence the cover passes. The reason for the valid bit is high is that the MSFF is not resettable - the flop output can be come out with any value starting from beginning.



Therefore, we can formally prove that all operations have the latency of 3 clock cycles, however, due to the non-resettable issue of the flip-flop, the *resultv* signal could rise earlier than expected. To fix the issue, we need to make the flop resettable. We will explain this in Part D in details.

Part B: Liveness Property

In this section, we exercise writing liveness properties, one that passes and another one that fails.

a. Passed Liveness Property

Assumption:

1. scan debugger is disabled
2. Opcodes are either arithmetic or logical operations
3. Clock gating is disabled

Assertion: when a valid operation arrives, there will be a result eventually.

```
//Assumption:
Legal_logical_opcode: assume property(@(posedge Clk) (valid_arithmetic_op(uopcode))|||(valid_logical_op(uopcode)));
clock_gating_disable: assume property(@(posedge Clk) (defeature_addck));
disbale_dft_scan:     assume property(@(posedge Clk) (dfts1ovrd == 1'b0));
assume property (@(posedge Clk) (uopv==1 ));
//Assertion:
passed_condition:  assert property(@(posedge Clk)(uopv) |=> s_eventually resultv);
```

Verification: the liveness property is proved

			Name	Type	Radius	Clocks	Time
⊙	P	∞	upov_condition	sva		Clk	
			Legal_logical_opcode	sva		Clk	
			assume__0	sva		Clk	
			clock_gating_disable	sva		Clk	
			disbale_dft_scan	sva		Clk	

b. Failed Liveness Property

Assumption:

1. 1. scan debugger is disabled
2. Opcodes are either arithmetic or logical operations
3. Clock gating is enabled

Assertion: when a valid operation arrives, there will be a result eventually.

```
//Assumption:
Legal_logical_opcode: assume property(@(posedge Clk)(valid_arithmetic_op(uopcode) )|||(valid_logical_op(uopcode)));
clock_gating_disable: assume property(@(posedge Clk) (!defeature_addck));
disbale_dft_scan:     assume property(@(posedge Clk) (dfts1ovrd == 1'b0));
//Assertion:
failed_condition:     assert property(@(posedge Clk)((uopv)|=> s_eventually resultv));
```

This liveness property failed, because in this environment, we didn't disable the clock gating, therefore it is possible that the formal tool find a counterexample when clock is gated, and the *resultv* never set to high.

⊙	F	∞	failed_condition	sva	3	Clk	40ns
			Legal_logical_opcode	sva		Clk	
			clock_gating_disable	sva		Clk	
			disbale_dft_scan	sva		Clk	

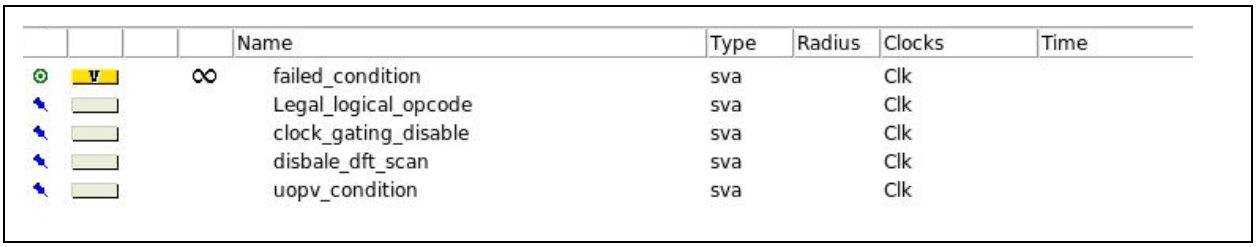
Part C: Vacuous Property

An important of quality checks is to make sure that we do not have any proofs passing vacuously. In this section, we exercise writing a vacuous property from the ALU design that will lead to a vacuous truth

Assumption & Assertion:

```
Legal_logical_opcode: assume property(@(posedge Clk) (valid_arithmetic_op (uopcode))|||(valid_logical_op (uopcode)));
clock_gating_disable: assume property(@(posedge Clk) (!defeature_addck));
disable_dft_scan:     assume property(@(posedge Clk) (dfts1ovrd == 1'b0));
uopv_condition:       assume property(@(posedge Clk) (uopv==0));
//Assertion:
vacuous_condition:    assert property(@(posedge Clk)((uopv)|=> s_eventually resultv));
```

The above property is proved to be vacuously. We say that an implication $p \rightarrow q$ holds vacuously if p is always false. In te case above, our operation valid bit is assumed to be 0, so it's impossible for uopv to hold ture and resultv eventually hold false. So the implication is tautology.



Part D: Comprehensive Formal Verification Approach

FPV Plan for Arithmetic Block

Goals	Verify the correct behavior of the arithmetic unit, in the absence of unusual activity, such as clock-gating.
Properties	Create cover points that replicate each waveform in the spec that illustrates arithmetic unit behavior. Assume all operations are arithmetic, blackboxing the logic unit, add an assumption that the logical subunit valid signal *logresultv* is always 0. Assume clock gating is off. a. Cover each arithmetic operation(ADD, SUB, CMP), alone and back-to-back with another arbitrary operation. Assume only arithmetic operations and src2 is nonzero. b. Cover cases of each operation above with specific data that exercise all bits. Assume dft scan is disabled. For example: i. opcode: ADD, src1=32'h77777777, src2=32'h88888888. ii. opcode: SUB, src1=32'hFFFFFFFE, src2=32'hFFFFFFFF. iii. opcode: MUL, src1=32'h0000FFFF, src2=32'h000010001. iv. opcode: CMP, src1=32'h0, src2=32'hFFFFFFFF. c. Assert that when a valid operation arrives, a valid output appears in three clock cycles. d. Assert that each operation generates the expected results, given specific data. e. Create a reference model, and check that the result in the real RTL matches our reference model.
Complexity Staging	Initial stages: Blackbox logical subunit, set DSIZE to 8. Disable DFT and clock gating. Stages for improving verification quality if time permits: a. Allow all DSIZE values b. Allow DFT functionality c. Allow clock-gating
Exit Criteria	We exercise all covers and prove our arithmetic unit assertions for correct ALU functionality, under the overconstraints we have specified above.

The syntax to blackbox the logic unit, and constraint the logic subunit result valid *logresultv* is shown below:

```
#blackbox logic unit
netlist blackbox logical_subunit
formal netlist constraint logresultv  1'b0
```

We would start with defining cover properties for typical activities and interesting combinations of basic behaviors in test A and B. Then we define assertions to prove targets(arithmetic operation) according by the specification in test C and D. Lastly, we created a shadow reference model that calculate the core results of the logic, and then compare the result generated by the two units.

Test A: Coverages for all Arithmetic Operations

Cover each arithmetic operation(ADD, SUB, MUL, CMP), alone and back-to-back with another arbitrary operation. Assume only arithmetic operations and src2 is nonzero.

Assumption:

```
Legal_logical_opcode: assume property(@(posedge Clk) (valid_arithmetic_op (uopcode) ));
src2_value: assume property  (@(posedge Clk) (src2!=0 ));
```

Cover Property:

```
genvar k;
generate for (k= OPADD;  k<= OPCMP; k++) begin: g1
    arithmetic_alone:cover  property (@(posedge Clk)(uopv & uopcode == k)  ##3 resultv )  ;
    arithmetic_back2back:cover property (@(posedge Clk)(uopv & uopcode == k) ##1 (uopv)##2 resultv ##1 resultv);
   end
endgenerate
```

The result shows that all opcodes can potentially be used in our environment. We have examined the waveform to confirm the fact.

C	g1[8].arithmetic_alone	sva	4	Clk
C	g1[8].arithmetic_back2back	sva	5	Clk
C	g1[9].arithmetic_alone	sva	4	Clk
C	g1[9].arithmetic_back2back	sva	5	Clk
C	g1[10].arithmetic_alone	sva	4	Clk
C	g1[10].arithmetic_back2back	sva	5	Clk
C	g1[11].arithmetic_alone	sva	4	Clk
C	g1[11].arithmetic_back2back	sva	5	Clk
	Legal_arithmetic_opcode	sva		Clk
	src2_value	sva		Clk

Test B: Coverages with Specific Data

Cover cases of each operation above with specific data that exercise all bits. Assume dft scan is disabled.

Assumption:

```
Legal_logical_opcode: assume property(@(posedge Clk) (valid_arithmetic_op (uopcode) ));
src2_value: assume property  (@(posedge Clk) (src2!=0 ));
```

We designed a set of inputs for each operation to exercise all bits in arithmetic unit.

Coverages:

```
valid_adder: cover property (@(posedge Clk)(
    (uopv & uopcode == OPADD) ##1(src1 == 32'h77777777 & src2 ==32'h88888888) ##2(resultv)));
valid_sub: cover property (@(posedge Clk)(
    (uopv & uopcode == OPSUB) ##1(src1 == 32'hFFFFFFFE & src2 ==32'hFFFFFFFF) ##2(resultv)));
valid_mul: cover property (@(posedge Clk)(
    (uopv & uopcode == OPMUL) ##1(src1 == 32'h0000FFFF & src2 ==32'h00010001) ##2(resultv)));
valid_cmp: cover property (@(posedge Clk)(
    (uopv & uopcode == OPCMP) ##1(src1 == 32'h0 & src2 ==32'hFFFFFFFF) ##2(resultv)));


valid_back2back_adder: cover property (@(posedge Clk)(
    ((uopv & uopcode == OPADD) ##1 (uopv & uopcode == OPADD & uopv & src1 == 32'h77777777 & src2 ==32'h88888888)
    ##1 (uopv & src1 == 32'h77777777 & src2 ==32'h88888888) ##1 resultv ##1 resultv)
));
valid_back2back_sub: cover property (@(posedge Clk)(
    ((uopv & uopcode == OPSUB) ##1 (uopv & uopcode == OPSUB & uopv & src1 == 32'hFFFFFFFE & src2 ==32'hFFFFFFFF)
    ##1 (uopv & src1 == 32'hFFFFFFFE & src2 ==32'hFFFFFFFF) ##1 resultv ##1 resultv)
));
valid_back2back_mul: cover property (@(posedge Clk)(
    ((uopv & uopcode == OPMUL) ##1 (uopv & uopcode == OPMUL & uopv & src1 == 32'h0000FFFF & src2 ==32'h00010001)
    ##1 (uopv & src1 == 32'h0000FFFF & src2 ==32'h00010001) ##1 resultv ##1 resultv)
));
valid_back2back_cmp: cover property (@(posedge Clk)(
    ((uopv & uopcode == OPCMP) ##1 (uopv & uopcode == OPCMP & uopv & src1 == 32'h00000000 & src2 ==32'hFFFFFFFF)
    ##1 (uopv & src1 == 32'h00000000 & src2 ==32'hFFFFFFFF) ##1 resultv ##1 resultv)
));
```

The result shows that both scenarios with our specific data alone and back-to-back with another operations give cover points.

Properties

Name	Type	Radius	Clocks	Time
valid_adder	sva	4	Clk	50ns
valid_back2back_adder	sva	5	Clk	60ns
valid_back2back_cmp	sva	5	Clk	60ns
valid_back2back_mul	sva	5	Clk	60ns
valid_back2back_sub	sva	5	Clk	60ns
valid_cmp	sva	4	Clk	50ns
valid_mul	sva	4	Clk	50ns
valid_sub	sva	4	Clk	50ns
Legal_logical_opcode	sva		Clk	
src2_value	sva		Clk	

Test C: Operations with Valid Latency

Assert that when a valid operation arrives, a valid output appears in three clock cycles.

```
genvar p;
generate for (p= OPADD;  p<= OPCMP; p++) begin: arithmetic3
   arithmetic_assert_3_cycles: assert property(@(posedge Clk)((uopv & uopcode==p) |-> ##3 resultv));
end
endgenerate
```

We receive proofs for the assertions above.

arithmetic3[8].arithmetic_assert_3_cycles	sva	Clk
arithmetic3[9].arithmetic_assert_3_cycles	sva	Clk
arithmetic3[10].arithmetic_assert_3_cycles	sva	Clk
arithmetic3[11].arithmetic_assert_3_cycles	sva	Clk

We also tested cases when clock gating is enabled, when a valid operation arrives, a valid result will not show up in three cycles.

```
Legal_logical_opcode: assume property (@(posedge Clk) (valid_arithmetic_op (uopcode) ));
clock_gating_enable:  assume property (@(posedge Clk) (!defeature_addck));
disbale_dft_scan:     assume property (@(posedge Clk) (dfts1ovrd == 1'b0));
src2_value:           assume property (@(posedge Clk) (src2==0 ));
```

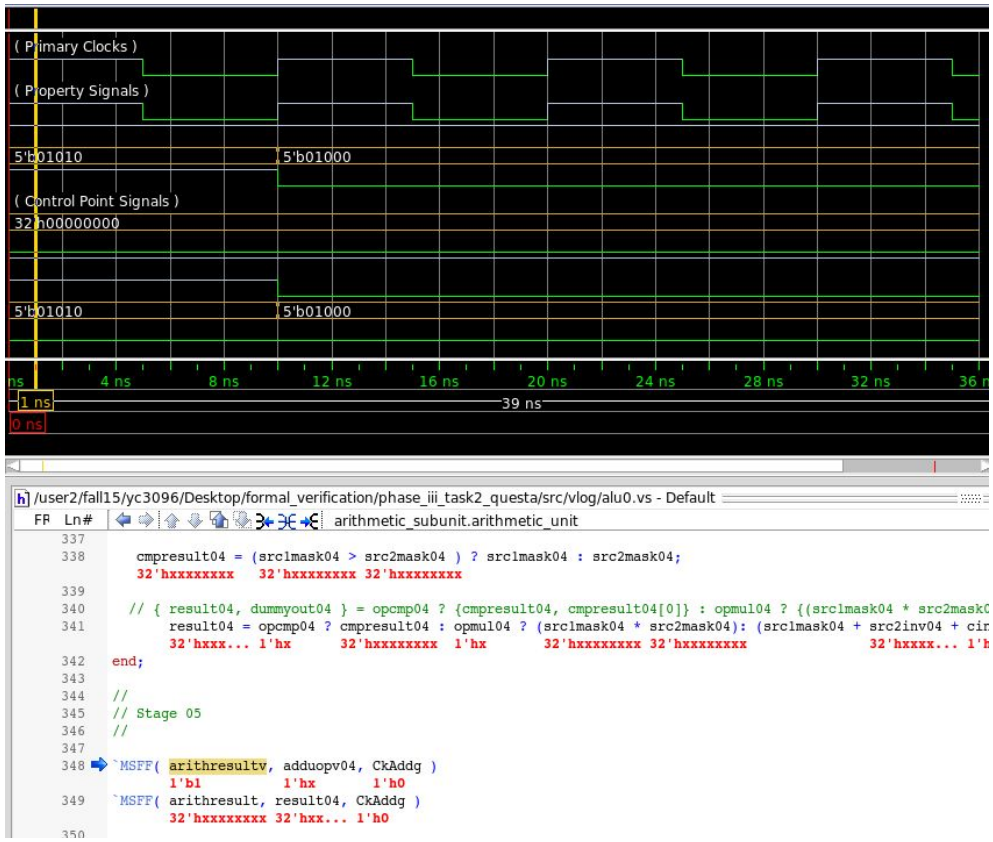
Assertion:

```
genvar m;
generate for (m= OPADD;  m<= OPCMP; m++) begin: g1
    Page176_cover_arithmeticm: assert property (@(posedge Clk) (uopv & (uopcode == m) |-> ##3 !resultv));
end
endgenerate
```

The FPV result shows firings for all four arithmetic operations.

Legal_logical_opcode	sva		Clk		
clock_gating_enable	sva		Clk		
disbale_dft_scan	sva		Clk		
src2_value	sva		Clk		
g1[10].Page176_cover_arithmeticm	sva	4	Clk		40ns
g1[11].Page176_cover_arithmeticm	sva	4	Clk		40ns
g1[8].Page176_cover_arithmeticm	sva	4	Clk		40ns
g1[9].Page176_cover_arithmeticm	sva	4	Clk		40ns

Bringing up the waveform, and tracing back from the *resultv* signal, we found the *arithresultv* node connected to the flip-flop in arithmetic unit has been always high since reset, hence the assertion fires. The reason for the valid bit is high is again that the MSFF is not resettable - the flop output can be come out with any value starting from beginning.



This issue could lead to more firings in further verification, therefore, we added asynchronous reset to the MSFF. The new RTL for MSFF is shown as below. Then we modified the corresponding RTL for ALU, which takes a rst bit as input, and passes it to it's submodules.

```
// A basic Flip-Flop with asynchronous reset
`define MSFF(q,i,clock,rst)   \
   always_ff @(posedge clock or posedge rst) \
      begin                   \
        if (rst) begin        \
            q <= 0;           \
        end else begin        \
            q <= i;           \
        end                   \
      end
```

After we made the flop resettable and completed these modifications to the ALU to our enviroment and rerun, we saw a pass to the assertions. We also went back to the previous test, and saw proves as well.

Legal_logical_opcode	sva	Clk	1
clock_gating_enable	sva	Clk	1
disbale_dft_scan	sva	Clk	1
src2_value	sva	Clk	1
g1[10].Page176_cover_arithmetic	sva	Clk	4
g1[11].Page176_cover_arithmetic	sva	Clk	4
g1[8].Page176_cover_arithmetic	sva	Clk	4
g1[9].Page176_cover_arithmetic	sva	Clk	4

Test D: Operations with Expected Result

Then we verify that each arithmetic operation generates the expected results, given specific data.

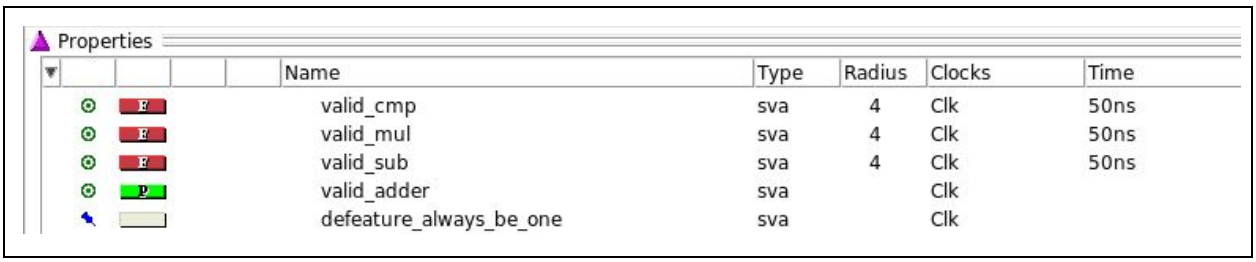
Assumption:

```
clock_gating_disable: assume property(@(posedge Clk) (defeature_addck));
```

Assertions:

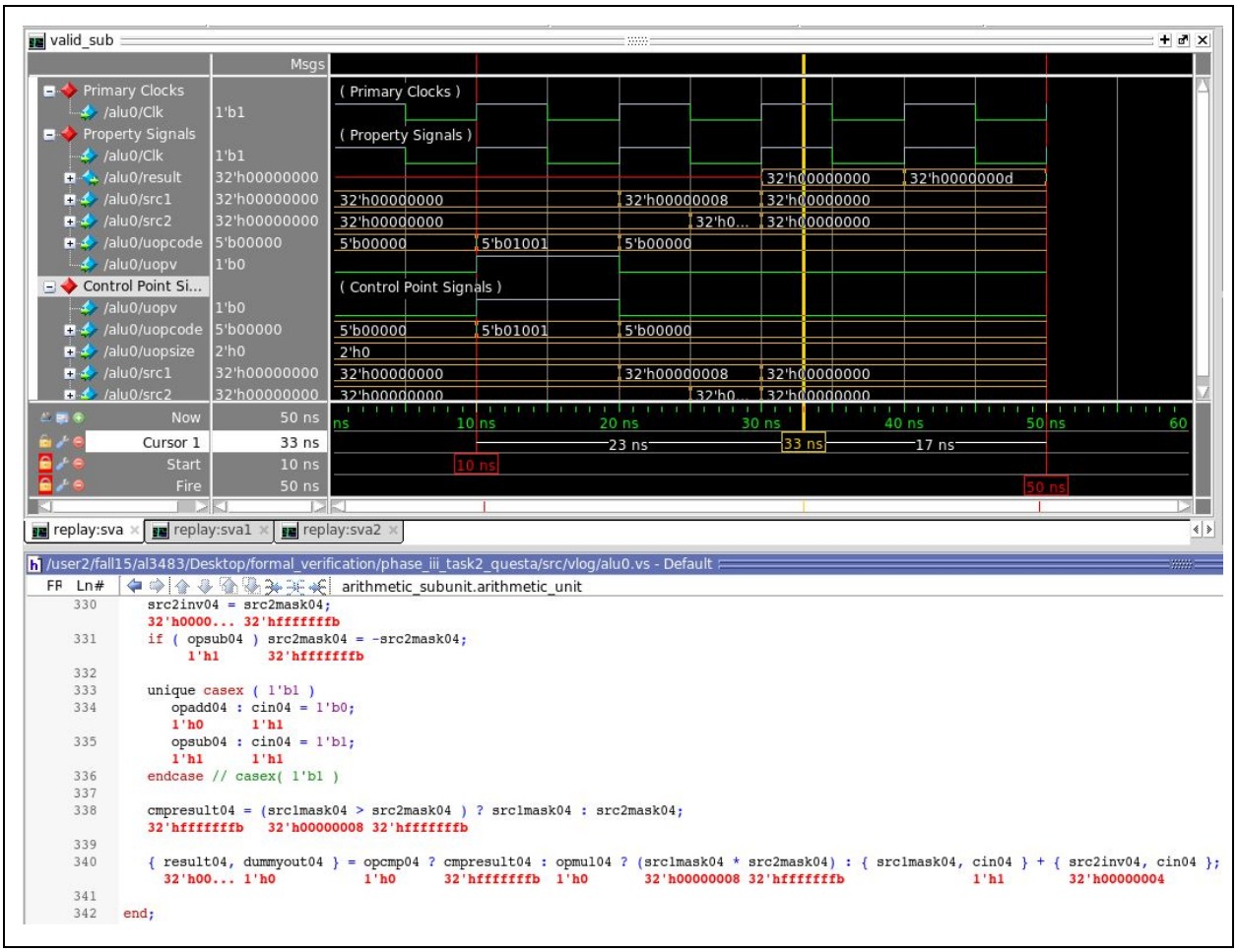
```
valid_adder: assert property (@(posedge Clk)(
    (uopv & uopcode == OPADD) ##1(src1 == 32'h8 & src2 ==32'h4) |-> ##2(result == 32'hC)));
valid_sub: assert property (@(posedge Clk)(
    (uopv & uopcode == OPSUB) ##1(src1 == 32'h8 & src2 ==32'h4) |-> ##2(result == 32'h4)));
valid_mul: assert property (@(posedge Clk)(
    (uopv & uopcode == OPMUL) ##1(src1 == 32'h8 & src2 ==32'h4) |-> ##2(result == 32'h20)));
valid_cmp: assert property (@(posedge Clk)(
    (uopv & uopcode == OPCMP) ##1(src1 == 32'h8 & src2 ==32'h4) |-> ##2(result == 32'h8)));
```

The result shows three unexpected firings for SUB, MUL and CMP operations. We look into each counterexample and find out why the design is not behaving sanely.



Subtraction Firing:

The waveform below shows that when a valid SUB opcode arrives and the two specified operands arrive at the next cycle, then we get an unexpected result of 32'hd(should be 32'h4). We trace back to line 340, and find that although *opsub04* is correctly set, the uninverted *src2inv04* is incorrectly assigned to *result04*.



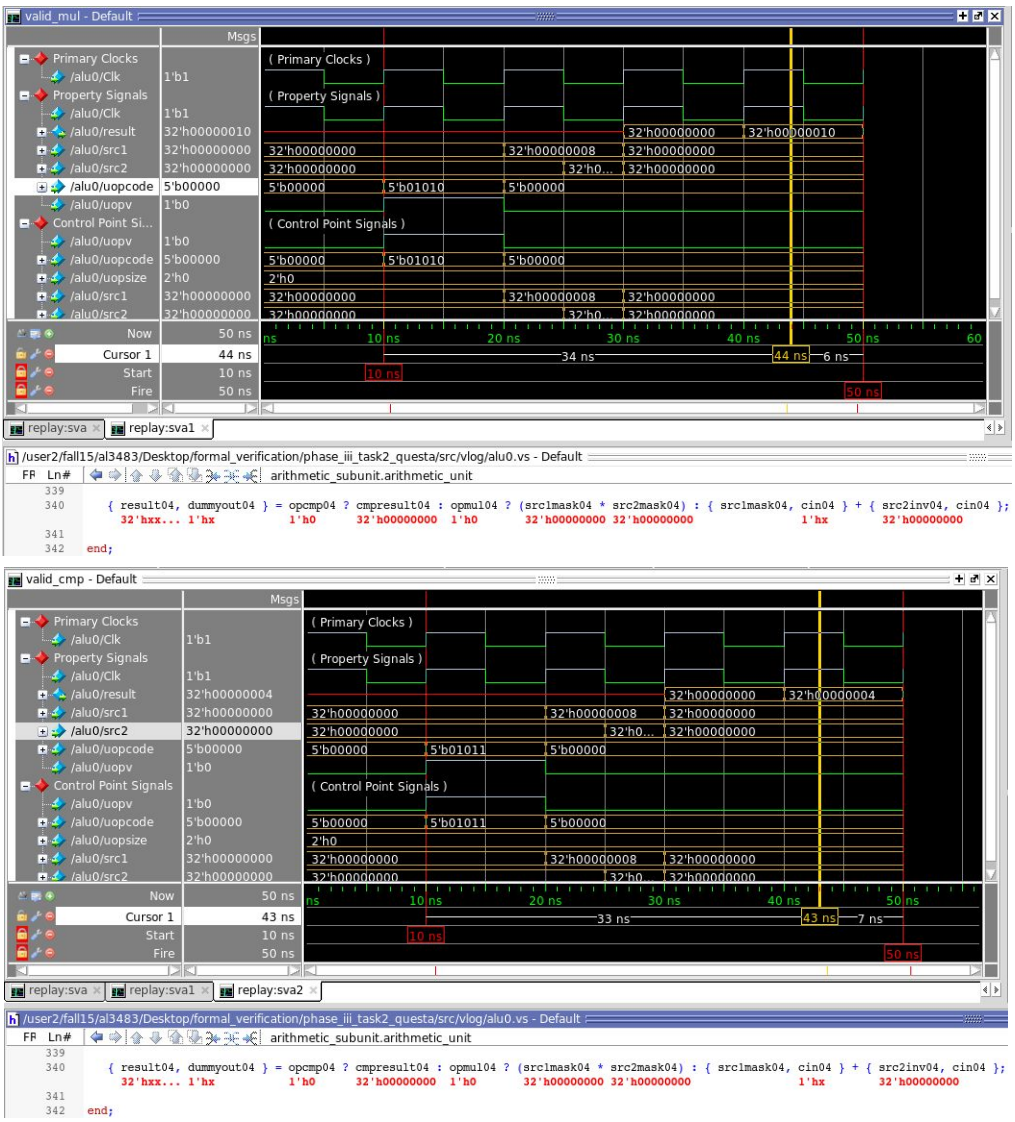

```
330     src2inv04 = src2mask04;
        32'h0000... 32'hfffffffb
331     if ( opsub04 ) src2mask04 = ~src2mask04;
             1'h1       32'hfffffffb
332
333     unique casex ( 1'b1 )
334        opadd04 : cin04 = 1'b0;
           1'h0      1'h1
335        opsub04 : cin04 = 1'b1;
           1'h1      1'h1
336     endcase // casex( 1'b1 )
337
338     cmpresult04 = (src1mask04 > src2mask04 ) ? src1mask04 : src2mask04;
        32'hfffffffb   32'h00000008 32'hfffffffb
339
340     { result04, dummyout04 } = opcmp04 ? cmpresult04 : opmul04 ? (src1mask04 * src2mask04) : { src1mask04, cin04 } + { src2inv04, cin04 };
          32'h00... 1'h0           1'h0      32'hfffffffb  1'h0       32'h00000008 32'hfffffffb                1'h1        32'h00000004
341
342 end;
```

To fix the issue, we change the assignment in line 331 as following:

```
if ( opsub04 ) src2inv04 = ~src2mask04;
```

Multiplication & Comparator Firing:

The waveforms below show that when a valid MUL or CMP opcode arrives and the two specified operands arrive at the next cycle, then we get an unexpected result of 32'h10(MUL) or 32'h04 (CMP), however, the expected result should be 32'h20 and 32'h8 respectively. We follow the same steps as the last task, and found that *src1mask04*, *src2mask04* and *cmpresult04* have the correct values, but the multiplied or compared results are not properly assigned to *result04*. The issue is that the last bit of the multiplied or compared result is assigned to the *dummyout04*. This dummy bit is useful to eliminate the cin bit from ADD/SUB operation, but will cause bug when the operation is multiplication and comparison.



To fix the issue, we change the assignment in line 340 as following:

```
result04 = opcmp04 ? cmpresult04 : opmul04 ? (src1mask04 * src2mask04): (src1mask04 + src2inv04 + cin04) ;
```

We rerun the FPV tool, and all three firings are fixed with the modifications above.

We verify another set of data that exercise all bits. Each operation generates the expected result in three cycles.

```
valid_adder: assert property (@(posedge Clk)(
    (uopv & uopcode == OPADD & uopsize == DSIZE32) ##1(src1 == 32'h77777777 & src2 == 32'h88888888) |-> ##2(result == 32'hFFFFFFFF)));
valid_sub: assert property (@(posedge Clk)(
    (uopv & uopcode == OPSUB & uopsize == DSIZE32) ##1(src1 == 32'hFFFFFFFE & src2 == 32'hFFFFFFFF) |-> ##2(result == 32'hFFFFFFFF)));
valid_mul: assert property (@(posedge Clk)(
    (uopv & uopcode == OPMUL & uopsize == DSIZE32) ##1(src1 == 32'h0000FFFF & src2 == 32'h00010001) |-> ##2(result == 32'hFFFFFFFF)));
valid_cmp: assert property (@(posedge Clk)(
    (uopv & uopcode == OPCMP & uopsize == DSIZE32) ##1(src1 == 32'h00000000 & src2 == 32'hFFFFFFFF) |-> ##2(result == 32'hFFFFFFFF)));
```

valid_adder	sva	Clk
valid_cmp	sva	Clk
valid_mul	sva	Clk
valid_sub	sva	Clk

Test E: Reference Model

Next we would like to create a reference model, calculating the expected result for each operation without including any of the complexities of a real design such as pipelining, scan, or debug logic.

The reference model for arithmetic looks like this:

```
module arithmetic_reference (Clk, uopv, uopcode, uopsize, src1, src2, arithresultv_ref, arithresult_ref, rst);

input node Clk;
input node rst;
input node uopv;
input node [3:0] uopcode;
input node [2:0] uopsize;
input node [31:0] src1;
input node [31:0] src2;
output node arithresultv_ref;
output node [31:0] arithresult_ref;
node opadd, opsub, opmul, opcmp;
node uopv1, isuop1;
node [3:0] uopcode1;
node [2:0] uopsize1;
node [31:0] result1, result2, resultv1, resultv2;
node isuop;
always_comb isuop = ( uopcode == OPADD | uopcode == OPSUB | uopcode == OPMUL | uopcode == OPCMP);

`MSFF( uopcode1, uopcode, Clk, rst)
`MSFF( uopsize1, uopsize, Clk, rst)
`MSFF( resultv1, uopv & isuop, Clk, rst)

always_comb begin
   opadd = ( uopcode1 == OPADD );
   opsub = ( uopcode1 == OPSUB );
   opmul = ( uopcode1 == OPMUL );
   opcmp = ( uopcode1 == OPCMP );
end

always_comb begin
   unique casex ( 1'b1 )
      opadd : result1 = (src1 + src2);
      opsub : result1 = (src1 - src2);
      opmul : result1 = (src1 * src2);
      opcmp : result1 = (src1 > src2 ) ? src1:src2;
   endcase
end
`MSFF( result2, result1, Clk, rst)
`MSFF( resultv2, resultv1, Clk, rst)
`MSFF( arithresult_ref, result2, Clk, rst)
`MSFF( arithresultv_ref, resultv2, Clk, rst)
endmodule
```

Once the above module is instantiated to the top-level ALU module, we can add the following assertion to check that the result is the read RTL matches our reference model. We first assume that clock gating is disabled, and the maximum data value is 8 bits.

```
assume_dsize8: assume property (@(posedge Clk) (uopsize == DSIZE08));
assume_src1_size8: assume property (@(posedge Clk) (src1 < 256));
assume_src2_size8: assume property (@(posedge Clk) (src2 < 256));
Legal_arithmetic_opcode: assume property (@(posedge Clk) (valid_arithmetic_op (uopcode) ));

genvar l;
generate for (l= OPADD;  l<= OPCMP; l++) begin: eq1
    assert_equi_check1: assert property ((uopv & uopcode == l)|-> ##3(arithresult_ref == result));
    assert_equi_check2: assert property ((uopv & uopcode == l)|-> ##3(arithresultv_ref == resultv));
end
endgenerate
```

The two assertions prove that the two units should generate the same outputs, give complete coverage of the data space of all arithmetic operations.

assume_dsize8	sva		Clk
assume_src1_size8	sva		Clk
assume_src2_size8	sva		Clk
eq1[8].assert_equi_check1	sva		Clk
eq1[8].assert_equi_check2	sva		Clk
eq1[9].assert_equi_check1	sva		Clk
eq1[9].assert_equi_check2	sva		Clk
eq1[10].assert_equi_check1	sva		Clk
eq1[10].assert_equi_check2	sva		Clk
eq1[11].assert_equi_check1	sva		Clk
eq1[11].assert_equi_check2	sva		Clk
Legal_arithmetic_opcode	sva		Clk

As the time permits, we tried DSIZE with 16 bits, however, we ran into inconclusive issue with the timeout of 30 mins. We received proofs for ADD, SUB and CMP, but the assertion for multiplication obtained an inconclusive result as the time runs out.

```
assume_dsize16: assume property (@(posedge Clk) (uopsize == DSIZE16));
assume_src1_size16: assume property (@(posedge Clk) (src1 < 65536));
assume_src2_size16: assume property (@(posedge Clk) (src2 < 65536));
Legal_arithmetic_opcode: assume property (@(posedge Clk) (valid_arithmetic_op (uopcode) ));

genvar l;
generate for (l= OPADD;  l<= OPCMP; l++) begin: eq1
    assert_equi_check1: assert property ((uopv & uopcode == l)|-> ##3(arithresult_ref == result));
    assert_equi_check2: assert property ((uopv & uopcode == l)|-> ##3(arithresultv_ref == resultv));
end
endgenerate
```

assume_dsize16	sva		Clk
assume_src1_size16	sva		Clk
assume_src2_size16	sva		Clk
eq1[8].assert_equi_check1	sva		Clk
eq1[8].assert_equi_check2	sva		Clk
eq1[9].assert_equi_check1	sva		Clk
eq1[9].assert_equi_check2	sva		Clk
eq1[10].assert_equi_check1	sva	3	Clk
eq1[10].assert_equi_check2	sva		Clk
eq1[11].assert_equi_check1	sva		Clk
eq1[11].assert_equi_check2	sva		Clk
Legal_arithmetic_opcode	sva		Clk

Furthermore, this reference model only covers a subset data sizes and arithmetic operations due to the state complexity. If time permits, we can relax the constraint(DSIZE and clock gating) to further expand the scope of our verification.

FPV Plan for Logical Block

Goals	Verify the correct behavior of the logical unit, in the absence of unusual activity, such as DFT scan.
Properties	Create cover points that replicate each waveform in the spec that illustrates logical unit behavior. Assume all operations are logical operation, blackboxing the arithmetic unit, add an assumption that the arithmetic subunit valid signal *arithresultv* is always 0. a. Cover each logical operation(AND, OR, XOR), alone and back-to-back with another arbitrary operation. Assume only logical operations. b. Cover cases of each operation above with specific data that exercise all bits. Assume dft scan is disabled. For example: c. Assert that when a valid operation arrives, a valid output appears in three clock cycles, with DFT disabled and enabled. DFT disabled: i. opcode: AND, src1=32'hFFFFFFFF, src2=32'hFFFFFFFF. ii. opcode: OR, src1=32'hFFFFFFFF, src2=32'h00000000. iii. opcode: XOR, src1=32'hFFFFFFFF, src2=32'h00000000. DFT enabled: iv. opcode: AND, dftdata=32'hFFFFFFFF, src2=32'hFFFFFFFF. v. opcode: OR, dftdata=32'hFFFFFFFF, src2=32'h00000000. vi. opcode: XOR, dftdata=32'hFFFFFFFF, src2=32'h00000000. d. Assert that each operation generates the expected results, given specific data, with DFT disabled and enabled. e. Create a reference model, and check that the result in the real RTL matches our reference model.
Complexity Staging	Initial stages: Blackbox logical subunit, set DSIZE to 8. Disable DFT and clock gating. Stages for improving verification quality if time permits: a. Allow all DSIZE values b. Allow DFT functionality
Exit Criteria	We exercise all possible corner cases for different operation codes to prove our logical block's results are valid and working correctly.

The syntax to blackbox the arithmetic unit, and constraint the logic subunit result valid *arithresultv* is shown below:

```
# Ignored, blackbox
netlist blackbox arithmetic_subunit
netlist constant arithresultv 1'b0
```

Similar to the previous plan, we would to start with defining cover properties for typical activities and interesting combinations of basic logical behaviors in test A, B and C. Then we define assertions to proof targets(arithmetic operation) according by the specification in test C and D. Lastly, we created a shadow reference model that calculate the core results of the logic, and then compare the result generated by the two units.

Test A: Coverages for all Logical Operations

Cover each logical operation(AND, OR, XOR), alone and back-to-back with another arbitrary operation. Assume only logical operations.

```
genvar k;
generate for (k= OPAND;  k<= OPXOR; k++) begin: g1
    arithmetic_alone: cover  property (@(posedge Clk)(uopv & uopcode == k)  ##3 resultv );
    arithmetic_back2back: cover property (@(posedge Clk)(uopv & uopcode == k) ##1 (uopv)##2 resultv ##1 resultv);
   end
endgenerate
```

The result shows that all opcodes can potentially be used in our environment. We have examined the waveform to confirm the fact.

Properties

Name	Type	Radius	Clocks	Time
g1[0].logical_alone	sva	4	Clk	50ns
g1[0].logical_back2back	sva	5	Clk	60ns
g1[1].logical_alone	sva	4	Clk	50ns
g1[1].logical_back2back	sva	5	Clk	60ns
g1[2].logical_alone	sva	4	Clk	50ns
g1[2].logical_back2back	sva	5	Clk	60ns

Test B: Coverages with Specific Data

Cover cases of each operation above with specific data that exercise all bits. Assume dft scan is disabled.

```
valid_and: cover property (@(posedge Clk)(
    (uopv & uopcode == OPAND  & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 ==32'hFFFFFFFF) ##2(resultv)));
valid_or: cover property (@(posedge Clk)(
    (uopv & uopcode == OPOR & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 ==32'h00000000) ##2(resultv)));
valid_xor: cover property (@(posedge Clk)(
    (uopv & uopcode == OPXOR & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 ==32'h00000000)  ##2(resultv)));

valid_back2back_and: cover property (@(posedge Clk)(
    ((uopv & uopcode == OPAND & uopsize == DSIZE32)
    ##1 (uopv & uopcode == OPAND & uopv & src1 == 32'hFFFFFFFF & src2 ==32'hFFFFFFFF)
    ##1 (uopv & src1 == 32'hFFFFFFFF & src2 ==32'hFFFFFFFF) ##1 resultv ##1 resultv)));
valid_back2back_or: cover property (@(posedge Clk)(
    ((uopv & uopcode == OPOR & uopsize == DSIZE32)
    ##1 (uopv & uopcode == OPOR & uopv & src1 == 32'hFFFFFFFF & src2 ==32'h00000000)
    ##1 (uopv & src1 == 32'hFFFFFFFF & src2 ==32'h00000000) ##1 resultv ##1 resultv)));
valid_back2back_xor: cover property (@(posedge Clk)(
    ((uopv & uopcode == OPXOR & uopsize == DSIZE32)
    ##1 (uopv & uopcode == OPXOR & uopv & src1 == 32'hFFFFFFFF & src2 ==32'h00000000)
    ##1 (uopv & src1 == 32'hFFFFFFFF & src2 ==32'h00000000) ##1 resultv ##1 resultv)));
```

The result shows that both scenarios with our specific data alone and back-to-back with another operations give cover points.

valid_and	sva	4	Clk	50ns
valid_back2back_and	sva	5	Clk	60ns
valid_back2back_or	sva	5	Clk	60ns
valid_back2back_xor	sva	5	Clk	60ns
valid_or	sva	4	Clk	50ns
valid_xor	sva	4	Clk	50ns

Test C: Operations with Valid Latency

Assert that when a valid operation arrives, a valid output appears in three clock cycles.

```
genvar p;
generate for (p= OPAND;  p<= OPXOR; p++) begin: logical3
   logical_assert_3_cycles: assert property(@(posedge Clk)((uopv & uopcode==p) |-> ##3 resultv));
end
endgenerate
```

Result showing that all logical operations will generate a result with a latency of 3 cycles.

logical3[0].logical_assert_3_cycles	sva	Clk
logical3[1].logical_assert_3_cycles	sva	Clk
logical3[2].logical_assert_3_cycles	sva	Clk

Test D: Operations with Expected Result

Then we verify that each logical operation generates the expected results, given specific data with DFT disabled and enabled.

```
valid_and_dft_disabled: assert property (@(posedge Clk)(
    (uopv & uopcode == OPAND & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 == 32'hFFFFFFFF & dfts1ovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)
));
valid_or_dft_disabled: assert property (@(posedge Clk)(
    (uopv & uopcode == OPOR & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 == 32'h00000000 & dfts1ovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)
));
valid_xor_dft_disabled: assert property (@(posedge Clk)(
    (uopv & uopcode == OPXOR & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 == 32'h00000000 & dfts1ovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)
));
valid_and_dft_enabled: assert property (@(posedge Clk)(
    (uopv & uopcode == OPAND & uopsize == DSIZE32)
    ##1(dftdata == 32'hFFFFFFFF & src2 == 32'hFFFFFFFF & dfts1ovrd == 1'b1) |-> ##2(result == 32'hFFFFFFFF)
));
valid_or_dft_enabled: assert property (@(posedge Clk)(
    (uopv & uopcode == OPOR & uopsize == DSIZE32)
    ##1(dftdata == 32'hFFFFFFFF & src2 == 32'h00000000 & dfts1ovrd == 1'b1) |-> ##2(result == 32'hFFFFFFFF)
));
valid_xor_dft_enabled: assert property (@(posedge Clk)(
    (uopv & uopcode == OPXOR & uopsize == DSIZE32)
    ##1(dftdata == 32'hFFFFFFFF & src2 == 32'h00000000 & dfts1ovrd == 1'b1) |-> ##2(result == 32'hFFFFFFFF)
));
```

The result shows that in either mode, we see expected result in three cycles.

valid_and_dft_disabled	sva	Clk
valid_and_dft_enabled	sva	Clk
valid_or_dft_disabled	sva	Clk
valid_or_dft_enabled	sva	Clk
valid_xor_dft_disabled	sva	Clk
valid_xor_dft_enabled	sva	Clk

Test E: Reference Model

Similar to what we did to the arithmetic unit, we would like to create a reference model for the logical unit, calculating the expected result for each operation without including any of the complexities of a real design such as pipelining.

The reference model for logical unit looks like this:

```
module logical_reference (Clk, rst, uopv, uopcode, uopsize, src1, src2, logresultv_ref, logresult_ref, dfts1ovrd, dftdata);

    input node Clk;
    input node rst;
    input node uopv;
    input node [3:0] uopcode;
    input node [2:0] uopsize;
    input node [31:0] src1;
    input node [31:0] src2;
    input node dfts1ovrd;
    input node [31:0] dftdata;
    output node logresultv_ref;
    output node [31:0] logresult_ref;

    node opand, opor, opxor;
    node uopv1, isuop1;
    node [3:0] uopcode1;
    node [2:0] uopsize1;
    node [31:0] src1_d, result1, result2, resultv1, resultv2;
    node isuop;
    always_comb begin src1_d = dfts1ovrd ? dftdata : src1; end
    always_comb begin isuop = ( uopcode == OPAND | uopcode == OPOR | uopcode == OPXOR); end

    `MSFF( uopcode1, uopcode, Clk, rst)
    `MSFF( uopsize1, uopsize, Clk, rst)
    `MSFF( resultv1, uopv & isuop, Clk, rst)

    always_comb begin
       opand = ( uopcode1 == OPAND );
       opor = ( uopcode1 == OPOR );
       opxor = ( uopcode1 == OPXOR );
    end

    always_comb begin
       unique casex ( 1'b1 )
          opand: result1 = (src1_d & src2);
          opor: result1 = (src1_d | src2);
          opxor: result1 = (src1_d ^ src2);
       endcase
    end
    `MSFF( result2, result1, Clk, rst)
    `MSFF( resultv2, resultv1, Clk, rst)
    `MSFF( logresult_ref, result2, Clk, rst)
    `MSFF( logresultv_ref, resultv2, Clk, rst)
endmodule
```

Once the above module is instantiated to the top-level ALU module, we can add the following assertion to check that the result is the read RTL matches our reference model. We first assume that DFT scan is disabled, and the maximum data value is 8 bits.

We obtained proofs for all assertions under the constraints above. Then we changed the uopsize to DSIZE16 and DSIZE32, along with relaxed src1, src2 and dftdata values. All cases give complete proofs. DSIZE = 8

```
assume_src1_size08: assume property (@(posedge Clk) (src1 < 256));
assume_src2_size08: assume property (@(posedge Clk) (src2 < 256));
assume_dftdata_size08: assume property (@(posedge Clk) (dftdata < 256));
assume_dsize08: assume property (@(posedge Clk) (uopsize == DSIZE08));
genvar l;
generate for (l= OPAND;  l<= OPXOR; l++) begin: eq1
    assert_equi_check_result: assert property ((uopv & uopcode == l)|-> ##3(logresult_ref == result));
    assert_equi_check_resultv: assert property ((uopv & uopcode == l)|-> ##3(logresultv_ref == resultv));
end
endgenerate
```

DSIZE = 8

		assume_dftdata_size8	sva	Clk	1
		assume_dsize8	sva	Clk	1
		assume_src1_size8	sva	Clk	1
		assume_src2_size8	sva	Clk	1
	P	eq1[0].assert_equi_check_result	sva	Clk	4
	P	eq1[0].assert_equi_check_resultv	sva	Clk	4
	P	eq1[1].assert_equi_check_result	sva	Clk	4
	P	eq1[1].assert_equi_check_resultv	sva	Clk	4
	P	eq1[2].assert_equi_check_result	sva	Clk	4
	P	eq1[2].assert_equi_check_resultv	sva	Clk	4
		Legal_arithmetic_opcode	sva	Clk	1

DSIZE = 16

```
assume_src1_size16: assume property (@(posedge Clk) (src1 < 65536));
assume_src2_size16: assume property (@(posedge Clk) (src2 < 65536));
assume_dftdata_size16: assume property (@(posedge Clk) (dftdata < 65536));
assume_dsize016: assume property (@(posedge Clk) (uopsize == DSIZE16));
genvar l;
generate for (l= OPAND;  l<= OPXOR; l++) begin: eq1
    assert_equi_check_result: assert property ((uopv & uopcode == l)|-> ##3(logresult_ref == result));
    assert_equi_check_resultv: assert property ((uopv & uopcode == l)|-> ##3(logresultv_ref == resultv));
end
endgenerate
```

		assume_dftdata_size16	sva	Clk	1
		assume_dsize16	sva	Clk	1
		assume_src1_size16	sva	Clk	1
		assume_src2_size16	sva	Clk	1
	P	eq1[0].assert_equi_check_result	sva	Clk	4
	P	eq1[0].assert_equi_check_resultv	sva	Clk	4
	P	eq1[1].assert_equi_check_result	sva	Clk	4
	P	eq1[1].assert_equi_check_resultv	sva	Clk	4
	P	eq1[2].assert_equi_check_result	sva	Clk	4
	P	eq1[2].assert_equi_check_resultv	sva	Clk	4
		Legal_arithmetic_opcode	sva	Clk	1

DSIZE = 32

```
assume_dsize32: assume property (@(posedge Clk) (uopsize == DSIZE32));
genvar l;
generate for (l= OPAND;  l<= OPXOR; l++) begin: eq1
    assert_equi_check_result: assert property ((uopv & uopcode == l)|-> ##3(logresult_ref == result));
    assert_equi_check_resultv: assert property ((uopv & uopcode == l)|-> ##3(logresultv_ref == resultv));
end
endgenerate
```

		assume_dsize32	sva	Clk	1
	P	eq1[0].assert_equi_check_result	sva	Clk	4
	P	eq1[0].assert_equi_check_resultv	sva	Clk	4
	P	eq1[1].assert_equi_check_result	sva	Clk	4
	P	eq1[1].assert_equi_check_resultv	sva	Clk	4
	P	eq1[2].assert_equi_check_result	sva	Clk	4
	P	eq1[2].assert_equi_check_resultv	sva	Clk	4

FPV Plan for Overall ALU Block

Goals	Verify the correct behavior of the overall ALU with different operation mode, including all DSIZEs, clock-gating and scan/debug feature.
Properties	Create assertions that set various kinds of inputs environment mode to illustrates overall ALU unit behavior by monitoring result valid bit. a. Assume clock gating is defeatured. i. Assert that when a valid operation arrives, each operation will generate valid output bit in three clock cycles. ii. Assert that when a valid operation is not arrived, the valid output will not arrive in three cycles. iii. Assert that when opcodes are not valid (neither arithmetical nor logical operations), the valid output will not arrive in three clock cycles. iv. Assert that each operation generates the expected results, given specific data, with DFT disabled. b. Assume clock gating is relaxed. i. Cover cases of when valid arithmetic operation arrives, the valid output does not arrive in three cycles. ii. Assert that when a valid logical operation arrives, the valid output arrives in three cycles. c. Assume both clock gating and dft scan/debug enabled: i. Create a reference model, and check that the result in the real RTL matches our reference model.
Complexity Staging	Since we have already verified each individual blocks, we would like to test the entire block with much relaxed constraints. Initial stages: set DSIZE to 8. Defeature clock gating and DFT scan. Stages for improving verification quality if time permits: 1. Allow DFT functionality 2. Allow all DSIZE values
Exit Criteria	We exercise all possible corner cases for different operation codes to prove our entire ALU block's results are valid and working correctly.

Lastly, we remove the blackboxes, and test the entire ALU unit by relaxing some of the initial compromises we made earlier. Below is the actual assumptions and assertions wrote in systemverilog.

Test A: Clock Gating & DFT Scan Disabled
We first tested the mode when clock gating is disabled, and DFT is also shut off.

```
clock_gating_disable:assume property  (@(posedge Clk) (defeature_addck));
no_upov_condition: assert property (@(posedge Clk)
                   disable iff(rst)(uopv&(valid_arithmetic_op(uopcode)||valid_logical_op(uopcode)))|-> ##3 resultv);
non_upov_condition: assert property (@(posedge Clk)
                   disable iff(rst)(!uopv) |-> ##3 !resultv);
no_valid_opcode_condition: assert property(@(posedge Clk)
                   disable iff(rst)(!valid_logical_op(uopcode))&&(!valid_arithmetic_op (uopcode))|->##3!resultv);

valid_adder: assert property (@(posedge Clk)disable iff(rst)(
    (uopv & uopcode == OPADD & uopsize == DSIZE32)
    ##1(src1 == 32'h77777777 & src2 == 32'h88888888 & dftslovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)));
valid_sub: assert property (@(posedge Clk)disable iff(rst)(
    (uopv & uopcode == OPSUB & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFE & src2 == 32'hFFFFFFFF & dftslovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)));
valid_mul: assert property (@(posedge Clk)disable iff(rst)(
    (uopv & uopcode == OPMUL & uopsize == DSIZE32)
    ##1(src1 == 32'h0000FFFF & src2 == 32'h00010001 & dftslovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)));
valid_cmp: assert property (@(posedge Clk)disable iff(rst)(
    (uopv & uopcode == OPCMP & uopsize == DSIZE32)
    ##1(src1 == 32'h00000000 & src2 == 32'hFFFFFFFF & dftslovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)));
valid_and_dft_disabled: assert property (@(posedge Clk)disable iff(rst)(
    (uopv & uopcode == OPAND & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 == 32'hFFFFFFFF & dftslovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)
));
valid_or_dft_disabled: assert property (@(posedge Clk)disable iff(rst)(
    (uopv & uopcode == OPOR & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 == 32'h00000000 & dftslovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)
));
valid_xor_dft_disabled: assert property (@(posedge Clk)disable iff(rst)(
    (uopv & uopcode == OPXOR & uopsize == DSIZE32)
    ##1(src1 == 32'hFFFFFFFF & src2 == 32'h00000000 & dftslovrd == 1'b0) |-> ##2(result == 32'hFFFFFFFF)
));
```

The result below shows that all assertions are proved with no counterexample.

clock_gating_disable	sva		Clk
no_upov_condition	sva		Clk
no_valid_opcode_condition	sva		Clk
non_upov_condition	sva		Clk
valid_adder	sva		Clk
valid_and_dft_disabled	sva		Clk
valid_cmp	sva		Clk
valid_mul	sva		Clk
valid_or_dft_disabled	sva		Clk
valid_sub	sva		Clk
valid_xor_dft_disabled	sva		Clk

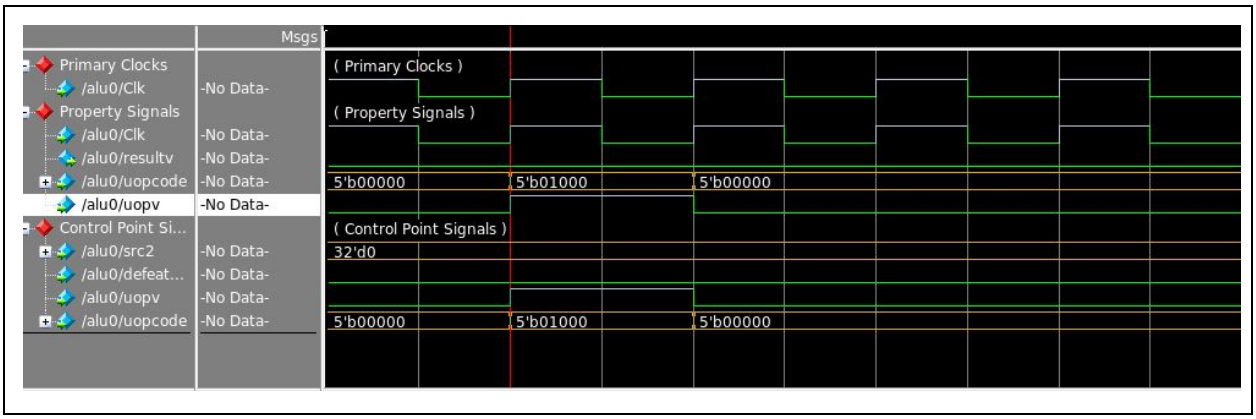
Test B: Operations with Defeature Clock Relaxed

Assume clock gating is relaxed. Cover cases of when valid arithmetic operation arrives, the valid output does not arrive in three cycles. Also add assertions that when a valid logical operation arrives, the valid output arrives in three cycles.

```
assume_valid_operation: assume property (@(posedge Clk) (valid_arithmetic_op(uopcode) || valid_logical_op(uopcode) ));
arithmetic_resultv: cover property (@(posedge Clk)(uopv & valid_arithmetic_op(uopcode) |-> ##3 !resultv));
logical_resultv: assert property (@(posedge Clk)(uopv & valid_logical_op(uopcode) |-> ##3 resultv));
```

The result shows that the coverage property are covered and assertions are proved. Look into the coverage case, the no result happens when src2 is 0, and the clock gating signal is ON. The coverage point is expected.

logical_resultv	sva		Clk	
arithmetic_resultv	sva	4	Clk	
assume_valid_operation	sva		Clk	



Test C: Reference Model

Lastly, we combined two reference models in previous two tasks, and generated a shadow model for the entire ALU block. This model includes all 7 operations and DFT scan, however, the clock gating feature is skipped as the specification for this feature remains unclear.

We started with data size of 8 bits, with several assumptions such as assuming valid logical or arithmetic operations, operands smaller than 8 bits value, and disabled clock gating. we added the following assertion to check that the result is the read RTL matches our reference model.

```
assume_dsize8: assume property (@(posedge Clk) (uopsize == DSIZE08));
assume_src1_size8: assume property (@(posedge Clk) (src1 < 256));
assume_src2_size8: assume property (@(posedge Clk) (src2 < 256));
clock_gating_defeatured:  assume property (@(posedge Clk) (defeature_addck));
assume_valid_operation: assume property (@(posedge Clk) (valid_arithmetic_op(uopcode) || valid_logical_op(uopcode) ));

genvar l;
generate for (l= OPADD;  l<= OPCMP; l++) begin: eq1
    assert_equi_check1: assert property ((uopv & uopcode == l)|-> ##3(result_ref == result));
    assert_equi_check2: assert property ((uopv & uopcode == l)|-> ##3(resultv_ref == resultv));
end
endgenerate
```

The two assertions prove that the two ALU units will generate the same outputs, give complete coverage of the data space of all arithmetic and logical operations with DSIZE08 and defeatured clock gating.

P	eq1[10].assert_equi_check1	sva	Clk
P	eq1[10].assert_equi_check2	sva	Clk
P	eq1[11].assert_equi_check1	sva	Clk
P	eq1[11].assert_equi_check2	sva	Clk
P	eq1[8].assert_equi_check1	sva	Clk
P	eq1[8].assert_equi_check2	sva	Clk
P	eq1[9].assert_equi_check1	sva	Clk
P	eq1[9].assert_equi_check2	sva	Clk
	assume_dsize8	sva	Clk
	assume_src1_size8	sva	Clk
	assume_src2_size8	sva	Clk
	assume_valid_operation	sva	Clk
	clock_gating_defeatured	sva	Clk

DSIZE = 16:

I	eq1[10].assert_equi_check1	sva	3	Clk
P	eq1[10].assert_equi_check2	sva		Clk
P	eq1[11].assert_equi_check1	sva		Clk
P	eq1[11].assert_equi_check2	sva		Clk
P	eq1[8].assert_equi_check1	sva		Clk
P	eq1[8].assert_equi_check2	sva		Clk
P	eq1[9].assert_equi_check1	sva		Clk
P	eq1[9].assert_equi_check2	sva		Clk
	assume_dsize16	sva		Clk
	assume_src1_size16	sva		Clk
	assume_src2_size16	sva		Clk
	assume_valid_operation	sva		Clk
	clock_gating_defeatured	sva		Clk

To further improving our PFV coverages, we tried DSIZE16 with a much longer runtime(3 hours). However, we still run into inconclusive issue for the multiplication due to enormous state space and design complexity. The good thing is that we are able to get proofs for all other operations(ADD, SUB, CMP, AND, OR and XOR) in DSIZE16 and DSIZE32.

Part E: Comparing Two alu0 Instances

In this section, we create a high-level module named *alu_cmp*, and instantiate two instances of *alu0* to the top module. Then we assert properties to check whether the outputs *result* and *resultv* are the same from both instances respectively .

The high-level module is looking like this:

```
module alu_cmp(Clk, uopv, uopcode, uopsize, src1, src2, defeature_addck, dfts1ovrd, dftdata, rst);
input node Clk, uopv;
input node [4:0] uopcode;
input node [1:0] uopsize;
input node [31:0] src1;
input node [31:0] src2;
input node dfts1ovrd;
input node defeature_addck;
input node [31:0] dftdata;
input node rst;
node        resultv_1;
node [31:0] result_1;
node        resultv_2;
node [31:0] result_2;
alu0 alu0_1(
                    .Clk ( Clk ), .uopv ( uopv ), .uopcode ( uopcode ),
                    .uopsize ( uopsize ), .src1 ( src1 ), .src2 ( src2 ),
                    .resultv(resultv_1), .result(result_1), .defeature_addck(defeature_addck),
                    .dfts1ovrd(dfts1ovrd), .dftdata(dftdata), .rst(rst)
            );

alu0 alu0_2(
                    .Clk ( Clk ), .uopv ( uopv ), .uopcode ( uopcode ),
                    .uopsize ( uopsize ), .src1 ( src1 ), .src2 ( src2 ),
                    .resultv(resultv_2), .result(result_2), .defeature_addck(defeature_addck),
                    .dfts1ovrd(dfts1ovrd), .dftdata(dftdata), .rst(rst)
            );

        default clocking c0 @(posedge Clk); endclocking
        genvar l;
        generate for (l= OPADD;  l<= OPCMP; l++) begin: eq1
            assert_equi_check1: assert property ((uopv & uopcode == l)|-> ##3(result_1 == result_2));
            assert_equi_check2: assert property ((uopv & uopcode == l)|-> ##3(resultv_1 == resultv_2));
        end
        endgenerate

endmodule // alu_comp
```

We obtained four firings as the *result1* and *result2* different from each other, Bringing up the waveform, The counterexample shows that both defeature bit and source2 are 0, and an arithmetic operation asserted at the second cycle. Since the clock is gated, neither logical nor arithmetic unit would be selected. We obtain floating results, thus the two results could be different.

⊙	F	eq1[10].assert_equi_check1	sva	4	Clk
⊙	F	eq1[11].assert_equi_check1	sva	4	Clk
⊙	F	eq1[8].assert_equi_check1	sva	4	Clk
⊙	F	eq1[9].assert_equi_check1	sva	4	Clk
⊙	P	eq1[10].assert_equi_check2	sva		Clk
⊙	P	eq1[11].assert_equi_check2	sva		Clk
⊙	P	eq1[8].assert_equi_check2	sva		Clk
⊙	P	eq1[9].assert_equi_check2	sva		Clk

This is due to a design issue with the MUX_2_1. The MUX output a floating output when *sa* and *sb* are both set to 0. We generally don't want this to happen. Therefore, to fix the problem, we add a default case that set out to 0.

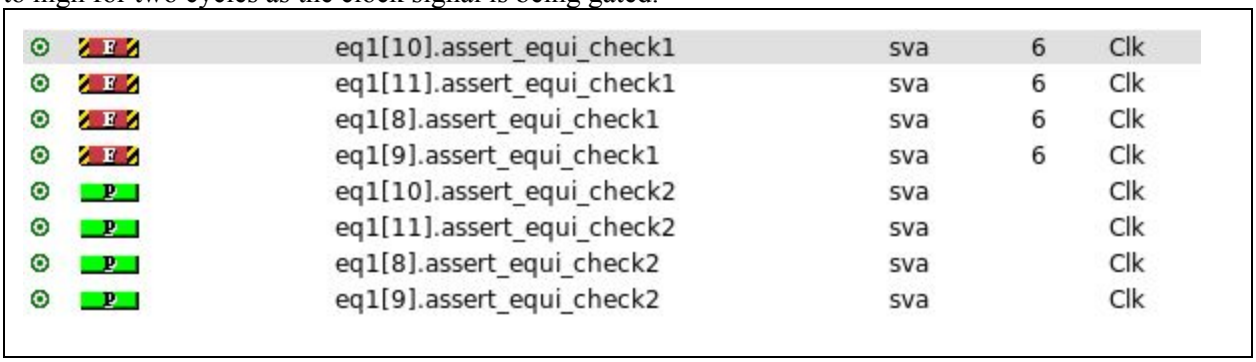
Before:

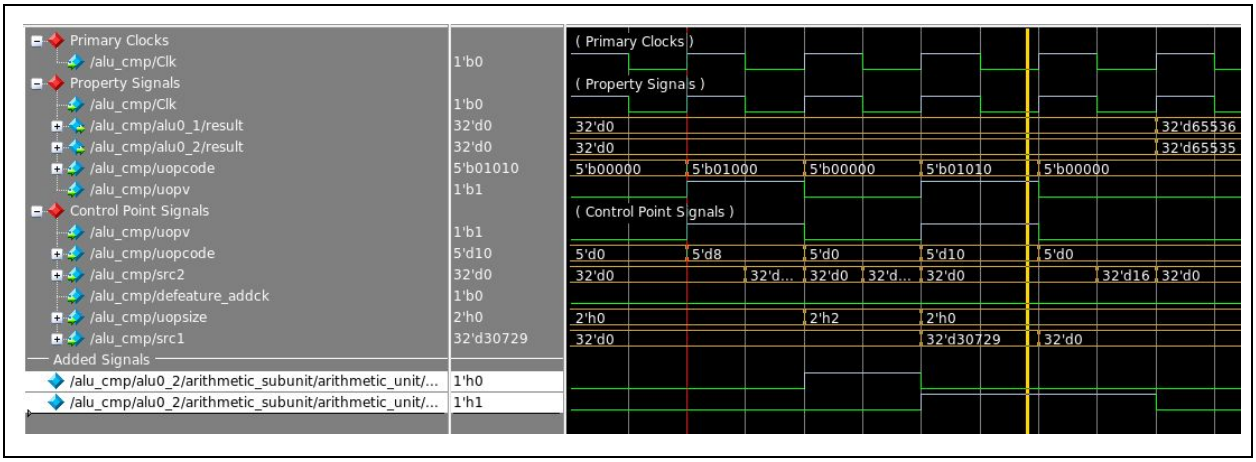
```
// A 2-to-1 mux
`define MUX2_1(out,sa,a,sb,b)         \
  unique casex ( 1'b1 )               \
            sa : out = a;             \
            sb : out = b;             \
         endcase // casex( 1'b1 )
```

After:

```
// A 2-to-1 mux
`define MUX2_1(out,sa,a,sb,b)         \
  unique casex ( 1'b1 )               \
            sa : out = a;             \
            sb : out = b;             \
            default : out = 0;        \
         endcase // casex( 1'b1 )
```

With the modified MUX2, we rerun the assertion, we obtain another failure. Bringing up the waveform, and tracing back from both instances, we see that after *addopov03* set high, *addopov04 and resultv* are set to high for two cycles as the clock signal is being gated.





In this case, the floating variable *cin* in the two instances are set to different values, which leads to different arithmetic results. Although it is not a valid arithmetic operation, a false *arithmetic_resultv* is still passed to the alu0, and *arithmetic_result* is selected as the output for the ALU unit.

```
unique casex ( 1'b1 )
   opadd04 : cin04 = 1'b0;
   1'h0      1'h0
   opsub04 : cin04 = 1'b1;
   1'h0      1'h0
endcase // casex( 1'b1 )
```

```
unique casex ( 1'b1 )
   opadd04 : cin04 = 1'b0;
   1'h0      1'h1
   opsub04 : cin04 = 1'b1;
   1'h0      1'h1
endcase // casex( 1'b1 )
```

To fix the floating issue, we add default case for *cin* same as what we did to the MUX2, then we obtain proof for all assertions.

P	eq1[10].assert_equi_check1	sva
P	eq1[10].assert_equi_check2	sva
P	eq1[11].assert_equi_check1	sva
P	eq1[11].assert_equi_check2	sva
P	eq1[8].assert_equi_check1	sva
P	eq1[8].assert_equi_check2	sva
P	eq1[9].assert_equi_check1	sva
P	eq1[9].assert_equi_check2	sva

In general, when we compare two instances of a design, we can find design problems that caused by floating gates. Because the design can go to multiple states with the same inputs. These are usually unwanted situations.

Part F: Inconclusive Result & Blackboxes

Inconclusive analysis happens when the formal analysis timed out before proving or disproving the assertion. As the complexity of the design increases, it takes much longer time for the checker to exhaustively reach a conclusive result. Blackboxing is one of the approaches to reduce design state space and complexity.

If we obtained a proof for the design with blackboxed modules, the proof is valid for the target unit. However, we should consider this as one proof apart from many other proofs in our comprehensive FPV plan. On the other hand, if we obtained a counterexample, it could either mean that we have a true bug in our design or we need to add more assumptions related to the blackbox to rectify some false positive situations.

When choosing the potential blackbox candidates, it is very important to think about the data flow through the design and what the effects of a blackbox might be. Since the outputs of a blackbox become free variables, we also need to carefully consider adding new assumptions, so that the blackbox does not appear to be producing faulty signals for the other modules.

Part G: Blackbox Vs. Abstractions

While abstraction techniques simplify the state space of the design by reducing the size of logic, blackboxes help to minimize the logic complexity by ignoring submodules that is irrelevant for formal verification. We have used both techniques in our formal verification project. The example of blackbox is blackboxing the logical unit while verifying arithmetic unit while verifying the ALU design. This technique requires making new assumptions such as setting *logresultv* to 0. An example of abstraction is

to reduce the size of counter from 32 bits to 8 bits in FIFO verification, which allow the tool to find the overflow issue at a much lower bound.

IV. CADENCE JASPER TOOL

Our last task is to reproduce task 1 using another popular commercial tool, Cadence Jasper.
The biggest difference in term of running the code is to use tcl file rather than Makefile. The tcl file looks like this:

```
# ----------------------------------------
#  Copyright (C) 2014 Jasper Design Automation, Inc. All Rights
#  Reserved.  Unpublished -- rights reserved under the copyright
#  laws of the United States.
# ----------------------------------------

# Analyze design under verification files
set ROOT_PATH ../designs/reference_design/verilog_sva
set RTL_PATH ${ROOT_PATH}/source/design
set PROP_PATH ${ROOT_PATH}/source/properties

analyze -sv \
  ${RTL_PATH}/fifo_transport_single.sv.orig.sv
analyze -sv \
  ${RTL_PATH}/fifo.sv

# Analyze property files
#analyze -sv \
#  ${PROP_PATH}/bindings.sva \
#  ${PROP_PATH}/v_fifo.sva

# Elaborate design and properties
elaborate -top fifo_transport_single

# Set up Clocks and Resets
clock clk
reset rst
# Get design information to check general complexity
get_design_info

# Prove properties
# 1st pass: Quick validation of properties with default engines
#set_max_trace_length 10
#prove -all
#
# 2nd pass: Validation of remaining properties with different engine
set_max_trace_length 50
set_prove_per_property_time_limit 150m

set_engine_mode {K I N}
prove -all

# Report proof results
report
```

We wrote a bash script, named run.sh. To run the script, execute:

```
bash run.sh
```

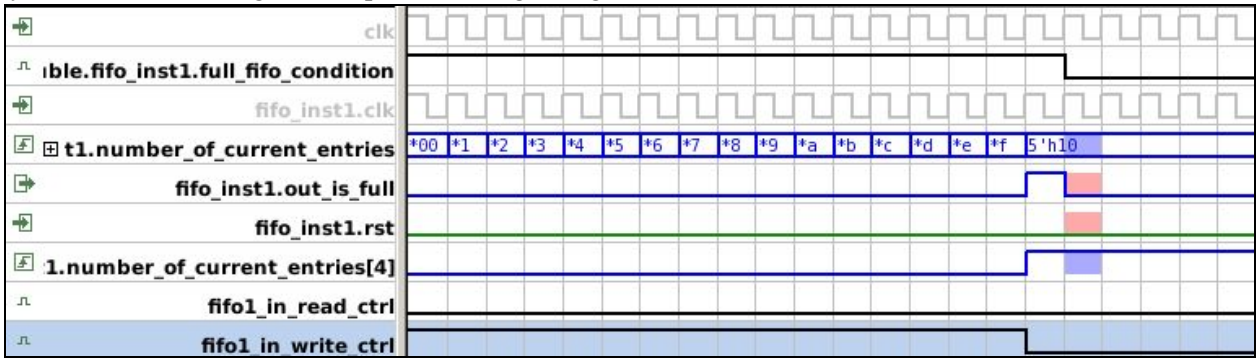
The Jasper FPV tool pops up, we obtain a pass for our set of assertions in *fifo_transport_single*. Also, we will not reach 5, 6 and 7 entries in the FIFO as we expected

```
---[ <embedded> ]-------------------------------------------------------------------------------------------------------------------------------
[1]   fifo_transport_single.fifo_assume_empty:precondition1                                  covered        I          1      0.001 s
[2]   fifo_transport_single.fifo_assume_full:precondition1                                   covered        K          5      0.002 s
[3]   fifo_transport_single.fifo_inst.fifo_num_entries_7                                     unreachable    N   Infinite      0.001 s
[4]   fifo_transport_single.fifo_inst.fifo_num_entries_6                                     unreachable    N   Infinite      0.001 s
[5]   fifo_transport_single.fifo_inst.fifo_num_entries_5                                     unreachable    N   Infinite      0.002 s
[6]   fifo_transport_single.fifo_inst.fifo_num_entries_4                                     covered        K      1 - 5      0.002 s
[7]   fifo_transport_single.fifo_inst.fifo_num_entries_3                                     covered        K      1 - 4      0.002 s
[8]   fifo_transport_single.fifo_inst.fifo_num_entries_2                                     covered        K      1 - 3      0.002 s
[9]   fifo_transport_single.fifo_inst.fifo_num_entries_1                                     covered        K      1 - 2      0.002 s
[10]  fifo_transport_single.fifo_inst.full_fifo_condition                                    proven         N   Infinite      0.000 s
[11]  fifo_transport_single.fifo_inst.full_fifo_condition:precondition1                      covered        K      1 - 5      0.002 s
[12]  fifo_transport_single.fifo_inst.fifo_full_not_empty_condition                          proven         N   Infinite      0.001 s
[13]  fifo_transport_single.fifo_inst.fifo_full_not_empty_condition:precondition1            covered        K      1 - 5      0.002 s
[14]  fifo_transport_single.fifo_inst.fifo_go_full_condition                                 proven         PRE Infinite      0.000 s
[15]  fifo_transport_single.fifo_inst.fifo_go_full_condition:precondition1                   covered        K      1 - 4      0.002 s
[16]  fifo_transport_single.fifo_inst.fifo_full_no_write_condition                           proven         I   Infinite      0.000 s
[17]  fifo_transport_single.fifo_inst.fifo_full_no_write_condition:precondition1             covered        K      1 - 5      0.002 s
[18]  fifo_transport_single.fifo_inst.empty_fifo_condition                                   proven         N   Infinite      0.001 s
[19]  fifo_transport_single.fifo_inst.empty_fifo_condition:precondition1                     covered        I          1      0.001 s
[20]  fifo_transport_single.fifo_inst.fifo_empty_not_full_condition                          proven         N   Infinite      0.001 s
[21]  fifo_transport_single.fifo_inst.fifo_empty_not_full_condition:precondition1            covered        I          1      0.001 s
[22]  fifo_transport_single.fifo_inst.fifo_go_empty_condition                                proven         PRE Infinite      0.000 s
[23]  fifo_transport_single.fifo_inst.fifo_go_empty_condition:precondition1                  covered        N          2      0.000 s
[24]  fifo_transport_single.fifo_inst.fifo_empty_no_read_condition                           proven         N   Infinite      0.000 s
[25]  fifo_transport_single.fifo_inst.fifo_empty_no_read_condition:precondition1             covered        I          1      0.001 s
[26]  fifo_transport_single.fifo_inst.fifo_no_empty_no_full_condition                        proven         N   Infinite      0.000 s
[27]  fifo_transport_single.fifo_inst.fifo_no_empty_no_full_condition:precondition1          covered        K      1 - 2      0.002 s
[28]  fifo_transport_single.fifo_inst.fifo_no_write_ptr_change_condition                     proven         PRE Infinite      0.000 s
[29]  fifo_transport_single.fifo_inst.fifo_no_write_ptr_change_condition:precondition1       covered        K      1 - 5      0.002 s
[30]  fifo_transport_single.fifo_inst.fifo_no_read_ptr_change_condition                      proven         PRE Infinite      0.000 s
[31]  fifo_transport_single.fifo_inst.fifo_no_read_ptr_change_condition:precondition1        covered        I          1      0.001 s
%%
```

Then we tried *fifo_transport_double,* which uses two instantiation of the faulty 16-deep synchronized FIFO design and a combined read/write signal. We received several firings.

✖	Assert	fifo_transport_double.fifo_inst1.full_fifo_condition	N	1 - 18	
✖	Assert	fifo_transport_double.fifo_inst1.empty_fifo_condition	I	35	
✖	Assert	fifo_transport_double.fifo_inst2.full_fifo_condition	K	34	
✖	Assert	fifo_transport_double.fifo_inst2.empty_fifo_condition	N	1 - 2	
✖	Cover	fifo_transport_double.fifo_inst1.fifo_num_entries_18	N (15)	Infinite	
✖	Cover	fifo_transport_double.fifo_inst1.fifo_num_entries_17	N (5)	Infinite	
✖	Cover	fifo_transport_double.fifo_inst2.fifo_num_entries_18	N (21)	Infinite	
✖	Cover	fifo_transport_double.fifo_inst2.fifo_num_entries_17	N (13)	Infinite	

We bring up the waveform for the first firing. The counterexample shows that the the FIFO Transport keeps adding values to the the first FIFO until it's full and it stops writing. Then both read and write control are set to zero, which reset the full signal to zero, enen though FIFO1 is still filled full with 16 entrities. As we mentioned in the first task, this bug is caused by the last *else if* statment added in the *fifo.sv*. After removing the last part, all firings are gone.



V. CONCLUSION

In this project, we first explored the basic methods, SystemVerilog assertions and line coverages of formal verification through small FIFO design exercies. we also explored the limitations of formal verification in some scenarios that could lead to incomplete verification. Then, we studied two advanced design verification techniques, bug hunting and full proof FPV techniques from Chapter 6 of Formal Verification textbook[1], and applied these techniques to a relatively complex ALU design. Our main tasks are accomplished Mentor Questa, however, we also exploited another commercial industry-standard tools, Cadence's JasperGold in some tasks.

In the phase of ALU verification, we verified each individual block using line coverages, assertions, and reference models. However, due to the complexity of multiplication operation, we were not able to obtain a prove for this operation beyond DSIZE08. Also, because lacking of specification, we were not able to understand what the *defeature_clock* bit means(it shut down some of the flops when src2=0 and defeature_clock=0). Therefore, we were not able to verify the correct behavior of the alu0 while it set to zero. However, we found several bugs with this bit drive to 1, including flop resetability issue, MUX output floating issue and incorrect arithmetic operation issue(SUB, MUL and CMP).

With increasing design complexity, verification becomes a very important but costly step of the design flow. As traditional simulation based testing cannot guarantee sufficient coverages, we found formal verification approach appears to be very cost effective in many cases. We particularly found it useful in the process of RTL design, since testbench creation usually is a very slow and error prone task.

VI. COLLABORATIONS

We found the topic to be very attractive and useful, thus both of us worked extremely hard and collaboratively on this project. The detail contributions are listed below:

Name	Yuxiang Chen(yc3096)	Ao Li(al3483*)*
Contributions	50%	50%
Details	1. Read user manual, search for relative papar and techniques 2. PFV Plan Adjustment 3. SVA Implementation 4. Debug Firing 5. Report Composing 6. Final Report Polishing and Adjustment	1. Read Textbook, search for relative techniques 2. PFV Planning 3. SVA Implementation 4. Debug Firings 5. Report Composing

VII. ACKNOWLEDGMENT

We would like to thank our professor Michael Theobald for his excellent course Formal Verification of Hardware and Software Systems, Your extensive knowledge, enthusiasm and organization made the class a pleasure to attend and study for. Thank you for completing our Columbia experience. We really appreciate it.

Meanwhile, we also would like to say thank you to our hardworking TAs, Xinhao and Tiezheng for their efforts before the midterm. It is your efforts that make our learning progress to be smooth.

VIII. REFERENCES

[1] E. Seligman, T. Schubert, and M. V. Achutha Kiran Kumar, *Formal verification: An essential Toolkit for modern VLSI design*. United States: Morgan Kaufmann Publishers In, 2015.

[2] C. E. Cummings, "SystemVerilog Assertions Design Tricks and SVA Bind Files," SNUG, 2009.

[3] "SystemVerilog assertions part-i," in *ASIC World*, 2014. [Online]. Available: http://www.asic-world.com/systemverilog/assertions1.html. Accessed: Dec. 20, 2016.

[4] Doulos, “Developing & Delivering KnowHow,” *SystemVerilog Assertions Tutorial*. [Online]. Available: https://www.doulos.com/knowhow/sysverilog/tutorial/assertions/. [Accessed: 20-Dec-2016].