

# **Synopsys VC Formal Tutorial**

## **Formal Property Verification App (FPV)**



**Portland State University**

©2023

## Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

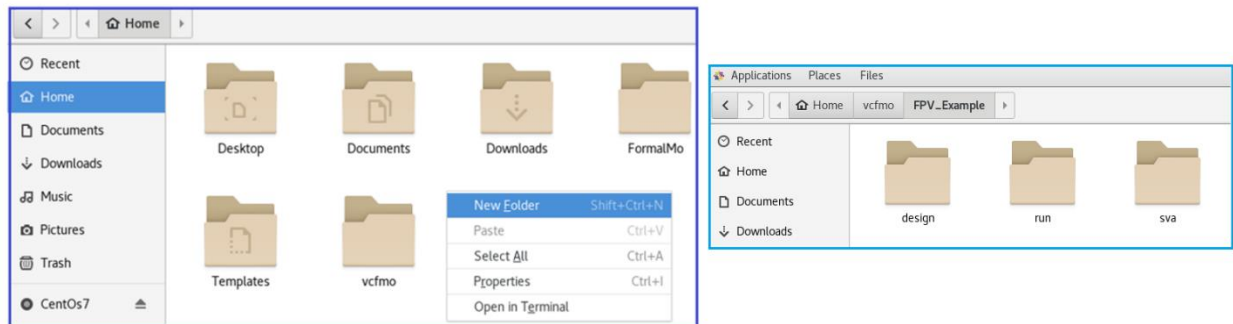
## Table of Contents

Introduction .....	3
About and Usage of FPV.....	4
Design Files .....	5
TCL File .....	7
Application Setup .....	9
Invoking VC Formal GUI .....	10
User Interface Details .....	10
Loading TCL File .....	11
Invoking VC Formal Along with TCL File .....	12
Running Files .....	13
Detecting Errors .....	14
Looking into the falsified properties .....	16
Resolving Errors .....	19
Restarting VC Formal .....	21
Appendix .....	22
SystemVerilog Assertions .....	22

## Introduction

VC Formal utilizes TCL (Tool Command Language) scripts to direct its actions. These scripts can specify various aspects of the analysis, such as the app to be used, the files to be analyzed, clock cycles, and more. However, instead of delving into the intricacies of writing TCL scripts, we can utilize pre-made templates. By modifying these templates, we can interact with VC Formal in a manner tailored to our specific project needs.

To begin, it is necessary to set up the VNCserver, as outlined in the previous tutorials, and open the Linux GUI. For organizational purposes, it is recommended to create a main folder for the design to be analyzed. In this example, the folder is named "FPV\_Example", but it is important to avoid using spaces when naming files and folders. Within this folder, two subfolders are created: "Design" for storing the Verilog or SystemVerilog designs to be analyzed, and "Run" for storing the TCL script that will run the VC Formal FPV analysis, and "sva" for the .sva assertion file.



*Figure 1. Creating folders to organize design and TCL files.*

## About and Usage of the FPV App

The FPV application is utilized for verifying a DUT by proving properties. These properties can be created by users or provided by commercial AIPs for interface protocol or function blocks. The app uses a range of powerful VC Formal engines to thoroughly prove or disprove a property. If an assertion fails, the FPV app will generate a counter-example to demonstrate a violating trace. However, there may be cases where it is impossible to definitively prove or disprove a property. In these cases, the app will provide a bounded proof result indicating that no falsification can be found up to a specific depth from the initial state. Proof results for a set of assertions mean that given the given constraints, it is impossible to falsify the properties.

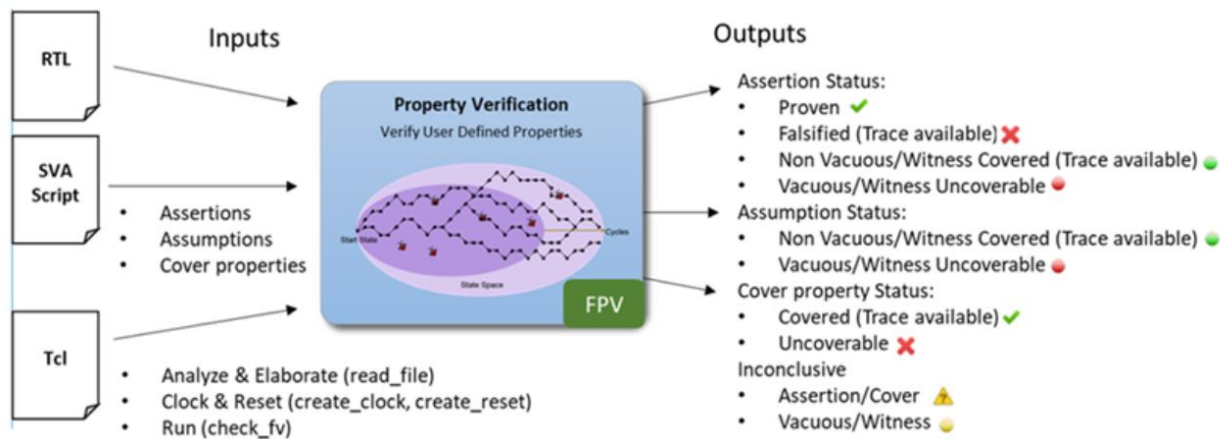


Figure 2. Inputs and outputs of the FPV app.

The FPV application requires several inputs, including:

- RTL design files in Verilog/SystemVerilog/VHDL formats
- Assertions, Assumptions, and cover properties written in an SVA file, or an inline SVA in the SystemVerilog design file.
- TCL file

The FPV app would then output the status of properties such as assertion status, assumption status, and cover status. The assertion status could be proven, falsified, vacuous, witness-coverable, uncoverable, or inconclusive.

## Design Files

The SystemVerilog design file should be placed in the Design folder, while the TCL file belongs in the Run folder. It's recommended to name the folders with lowercase letters since VC Formal is case-sensitive. Additionally, ensure that there are no spaces in the names of the folders or files.

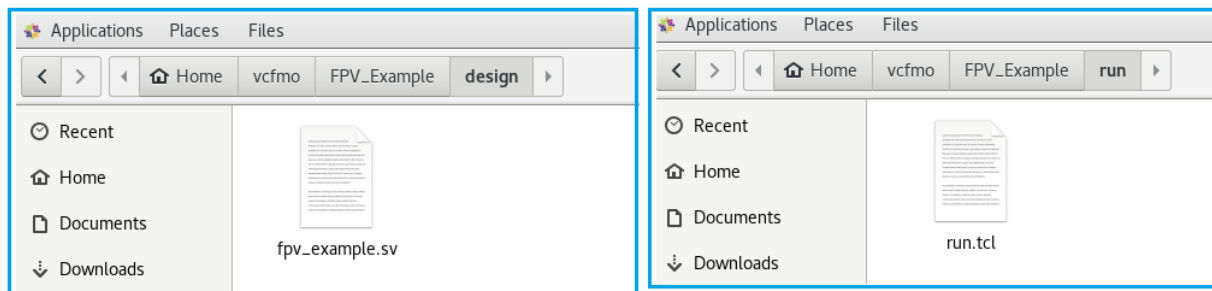


Figure 3. Showing the SystemVerilog and TCL files in the correct folder locations.

### Two places to write the SystemVerilog Assertions

The first way is to create .sva file in the SVA folder, create a module, and write the SVA assertions in it. You'll then need to create another binder file and bind your SV design and the SVA assertion module together.

The second way is to write inline SVA assertions inside your SV code. This is usually much faster, and possibly easier to do, though it's not the most practical when writing many assertions.

For simplicity we will go through how to write inline assertions within your SV design itself. In this case, you won't need to put anything inside the "sva" folder.

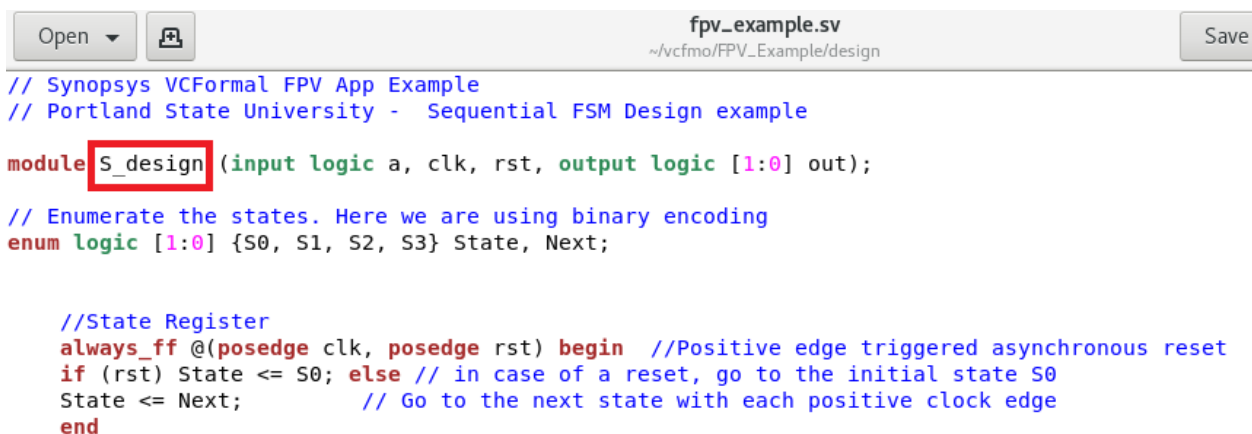


Figure 4. Example of the design we are using in this tutorial.

In order for VC Formal to map your design, you need to acknowledge and keep note of your exact module name, as highlighted in *Figure 3* above. This will come in handy when you create your TCL file. To make things simple, we suggest naming your design file after its corresponding module name (more on this in the TCL File section below).

## TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 5*), which you can use as a template for your functional checks on VC Formal.

```

# Portland State University VC_Formal_Team_FPV_App Tutorial

# Select FPV as the VC Formal App mode
set_fml_appmode FPV ①

# Set the module name as the design parameter
set design S_design ②

# Read the module "S_design" in the file /design/fpv_example.sv
# "+define+INLINE_SVA" allows us to write SVA properties and assertions within the .sv design
itself
read_file -top $design -format sverilog -sva \
  -vcs {../design/fpv_example.sv +define+INLINE_SVA} ③ ④

# Create clock and reset signals
# clk and rst should reflect the name of the clock and reset signals in your design
create_clock clk -period 100 ⑤
create_reset rst -sense high ⑥

# Running a reset simulation
sim_run -stable
sim_save_reset
  
```

*Figure 5. Annotated TCL template file.*

1. Instruction that sets the appmode to FPV in VC Formal.
2. Name of the main module as established in the Design file.
3. Location of the design file should be specified for VC Formal to locate it.
4. “+define+INLINE\_SVA” lets VC Formal know that we will be using inline SVA in our design.
5. Name of the clock signal. You can specify the period you’d like to use. (You can keep it as 100)
6. Name of the reset signal in your design, and whether it’s active high or low.

As mentioned before, VC Formal is case AND character sensitive, and in order for it to map your designs, you will need to use the same module name in the TCL script (**2**) and the module name in the design file. For example, we can see that our module name in the TCL script is the same as the module name in the design file in *Figure 3*.

The file name (fpv\_example.sv) can be named however you want.

## Writing the assertions

Now to use the FPV app, we need to write the assertions we want to test in our design. The assertions are written in SVA (SystemVerilog Assertions). In this tutorial, we will write inline SVA assertions.

In your sv design file, scroll down just before endmodule to write the assertions.

You'll need to use "``ifdef INLINE_SVA`" before you write the assertions, and "``endif`" once you're done writing the assertion. Below is an example of 4 Inline assertions written at the bottom of my fpv\_example.sv file, just before "endmodule"

```

// Selecting the state output
always_comb begin
  case (State)
    S0:    out=2'b00;
    S1:    out=2'b01;
    S2:    out=2'b10;
    S3:    out=2'b11;
    default: out=2'b00;
  endcase
end

// inline SVA (included within the design)
`ifdef INLINE_SVA
// Check if the output is coded using onehot encoding
onehot_out: assert property(
  @(posedge clk) disable iff (rst) $onehot(out));

// Check if the output at State S0 is 2'b00
check_out_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 |-> out==2'b00));

// Check if the output at State S3 is 2'b01
check_out_S3: assert property(
  @(posedge clk) disable iff (rst)
  (State==S3 |-> out==2'b01));

// Check if we get next state S1 the next clock cycle
// if we're in S0 and the input a is high
check_state_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 && a) |-> ##1 (State==S1));
`endif

endmodule

```

Saving file "/home/ghonim/vcfmo/FPV\_Example/design/fpv\_e... SystemVerilog Tab Width: 8 Ln 63, Col 10 INS

Figure 6. Inline SVA assertions in the design sv file

An Appendix is attached to help you write basic SVA assertions.



## Application Setup

There are multiple ways to invoke the VC Formal GUI and load the TCL script.

In this tutorial, we will show two methods for doing so. When using either method, it is important that you open the terminal within the appropriate “Run” folder associated with the FXP app.

- [Invoke VC Formal GUI](#), then manually load the TCL script in the application:

```
$vcf -gui
```

**OR**

- [Invoke VC Formal GUI and TCL script](#) in one command:

```
$vcf -f run.tcl -gui      or      $vcf -f run.tcl -verdi
```

‘run.tcl’ is the name of the TCL file we are using. If your file name differs from this, you will need to change it accordingly in this command.

The “-gui” switch opens VC Formal in the GUI, and it’s equivalent to the switch “-verdi”.

We will go through both of these methods in the following sections.

## Invoking VC Formal GUI

To proceed with invoking VC Formal:

- 1) Inside the “run” folder, right-click the whitespace and choose “Open in Terminal”
- 2) In the terminal, type in the command:

```
vcf -gui
```

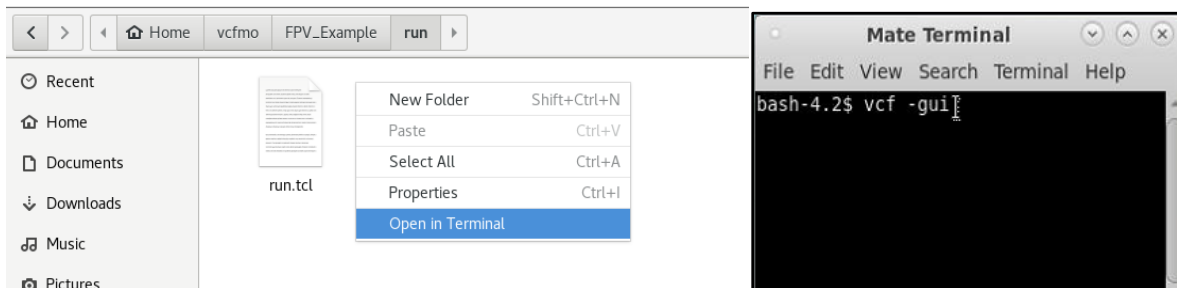


Figure 7. Invoking VC Formal in the terminal.

Note: You may have to wait a few seconds for the program to start up.

You should then see the VC Formal GUI as shown in *Figure 8* below. You can toggle between showing Targets, Constraints, or both Targets and Constraints window by clicking the blue box icon in the upper right-hand of the application (**1**).

It may look like any of these three icons:

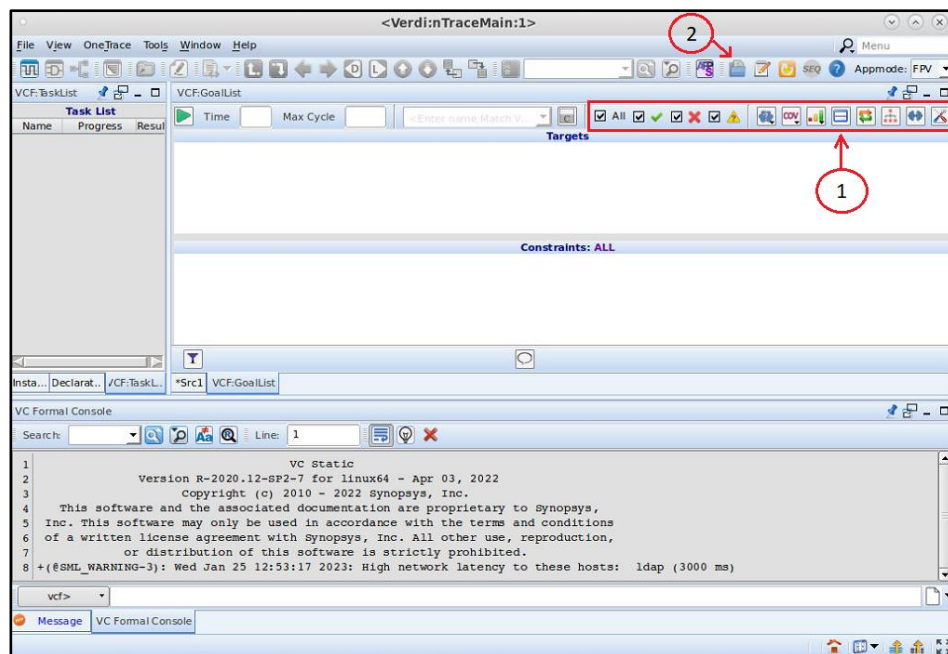

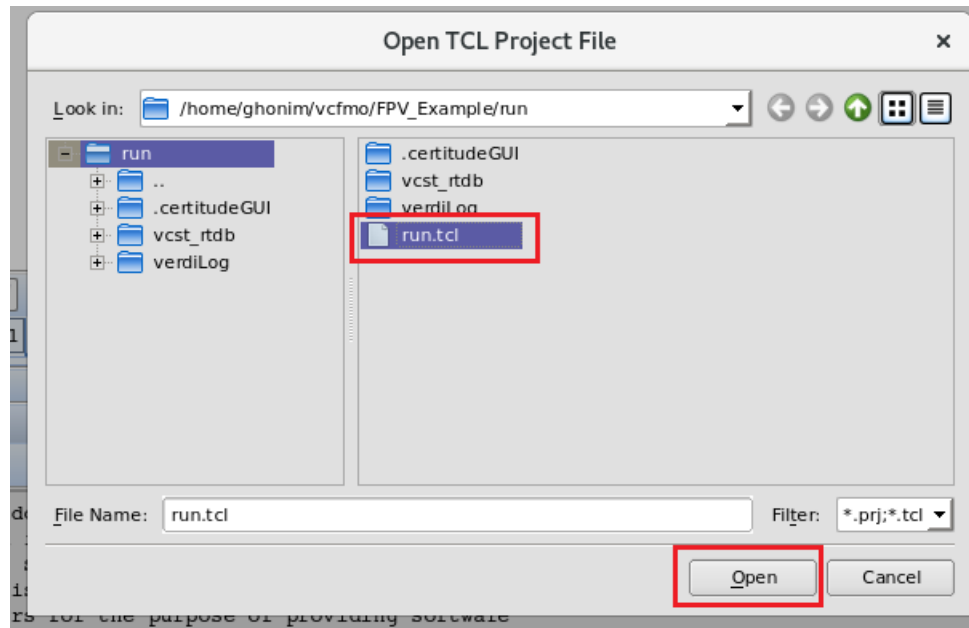


Figure 8. VC Formal GUI introductory screen.

Then load a TCL script by clicking on the  icon **(2)** as shown in *Figure 8*.

Next, select the “run.tcl” file we have in the “run” folder:



*Figure 9. Selecting TCL file.*

## Invoking VC Formal Along with TCL File:

To proceed with invoking VC Formal:

- 1) Inside the “run” folder, right-click the whitespace and choose “Open in Terminal”
- 2) In the terminal, type in the command:

```
vcf -f run.tcl -gui
```

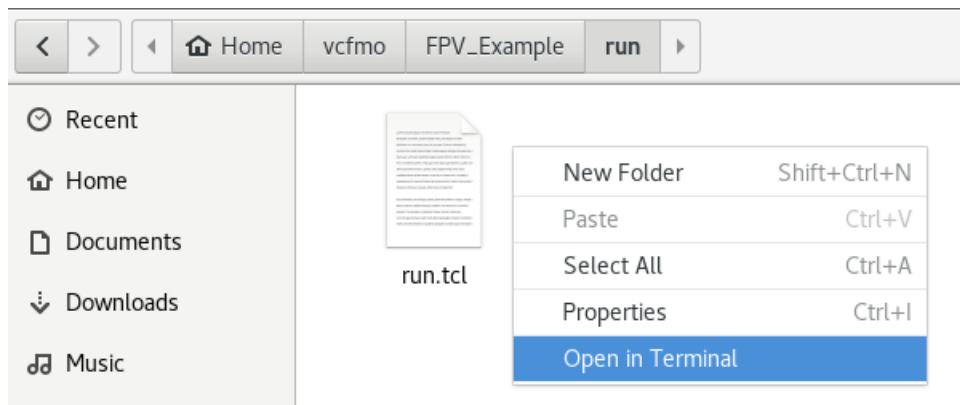


Figure 10. Opening terminal in the ‘run’ folder.

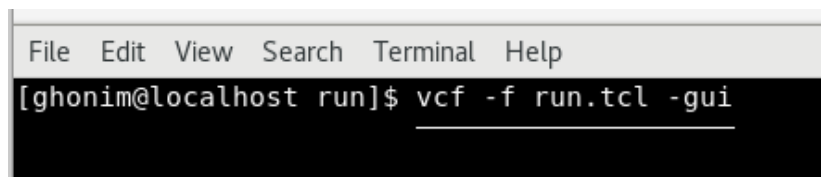


Figure 11. Invoking VC Formal and TCL script in the terminal.

## Running Files

After successfully invoking VC Formal GUI and loading the TCL script in the above methods, your screen should have contents in the *VCF:GoalList* tab and look something like this:

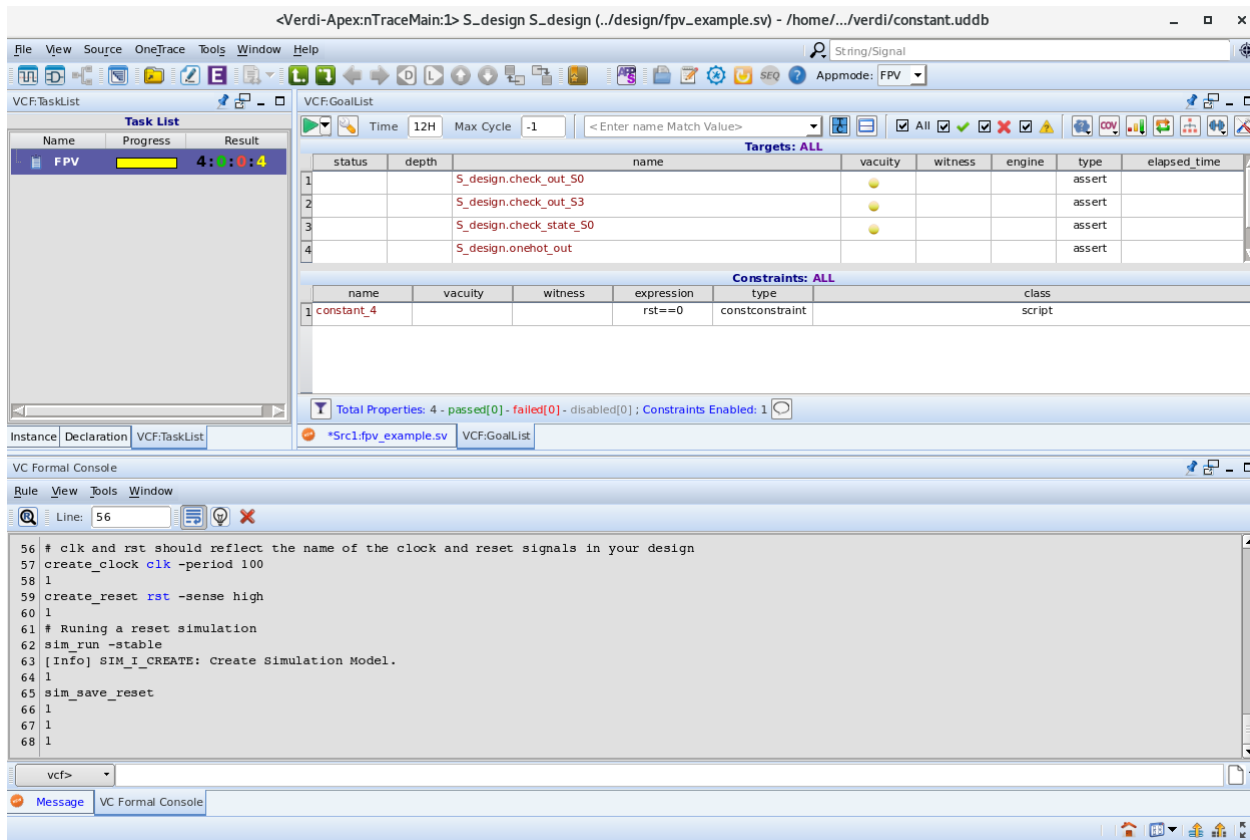



Figure 12. Screen after loading TCL script.

You should see all of your outputs listed here. Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the *VCF:GoalList* tab.

## Detecting Errors

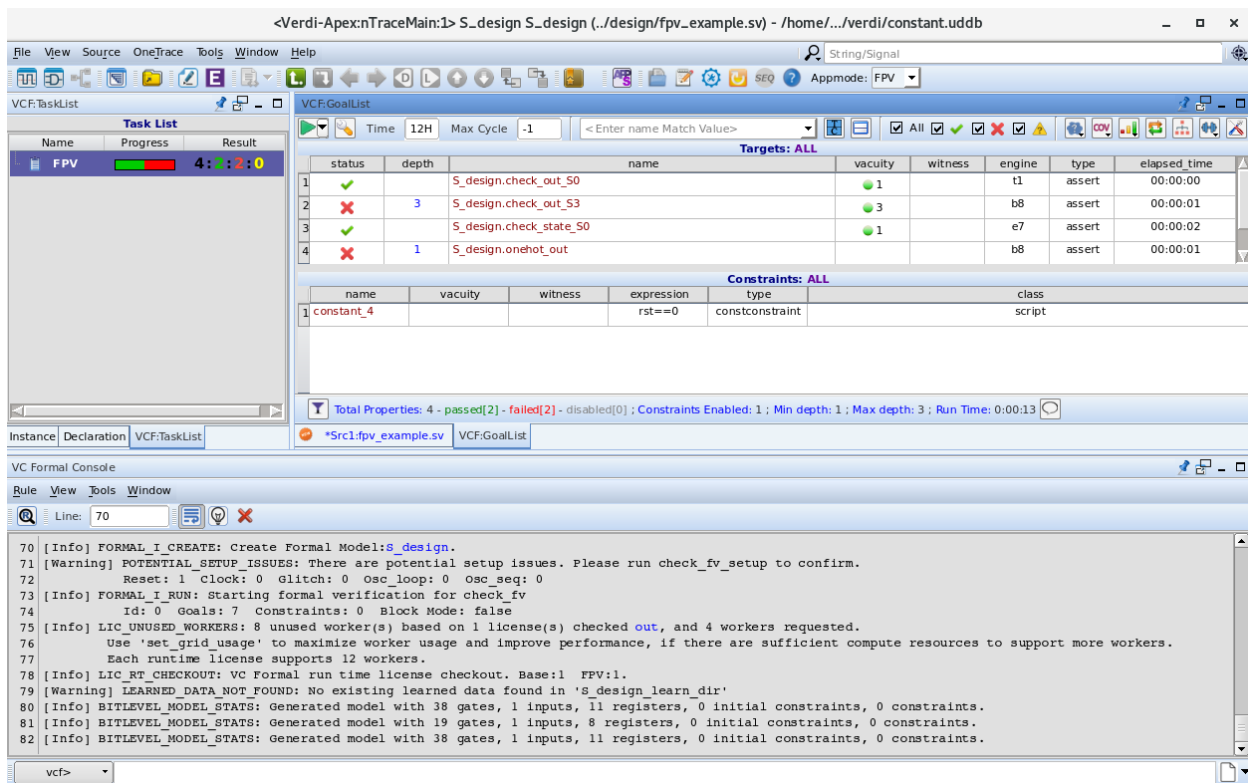


Figure 13. Screen after running TCL script and the status given = ✗.

In Figure 11 above, we see two icons, meaning that two of the assertions are proven. In our design, the “check\_out\_S0” property means that the output of S0 is indeed 2'b00, and the “check\_state\_S0” assertion means that indeed when we’re in State S0, and the input a is high, the next state is going to be S1 as we specified in our inline assertions.

We also have two icons, meaning that two of the assertions were falsified. In our design, the “onehot\_out” property means that signal “out” has onehot encoding, meaning that we only have exactly one bit high “1” at a time, which clearly false in our design since we’re using binary encoding, and the signal out can be 2'b11, and 2'b00. VC Formal falsified this property, as expected.

The “check\_out\_S3” property indicates that the output signal “out” of state S3 is 2'b01, which is clearly incorrect since we’re setting that output to be 2'b11 in our design, hence the property is falsified.

On the left under *Task List*, we can see that VC Formal was given one task by the FPV app. You can hover over the numbers under *Result* to see the results in detail. You may need to alter the tab size by dragging the side panels or scrolling to the left:

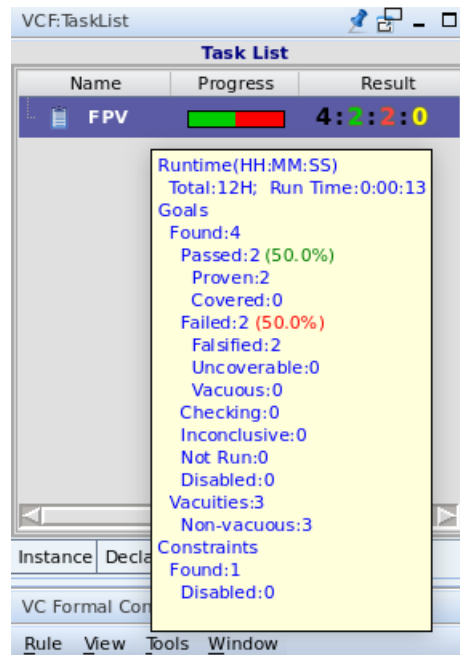



Figure 14. Results of the analyzed script.

## Looking into the falsified properties

To see what caused the falsified properties to be falsified, double-click on the first  (row 2) from *Figure 13*. You should then see a generated waveform like the one shown below:

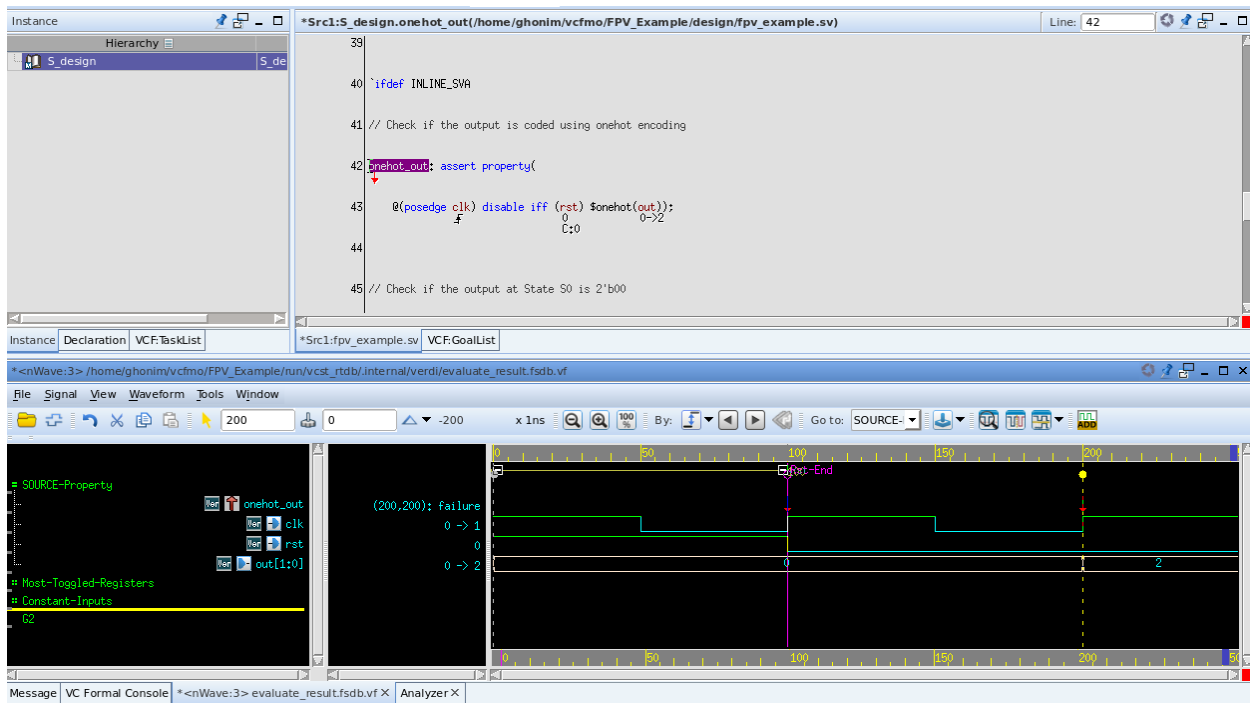


Figure 15. Examining the failed FPV check in our design.

If you double click on the `out[1:0]` signal, you'll see instances where both `out[1]` and `out[0]` are 0's or 1's, which is not correct onehot encoding.

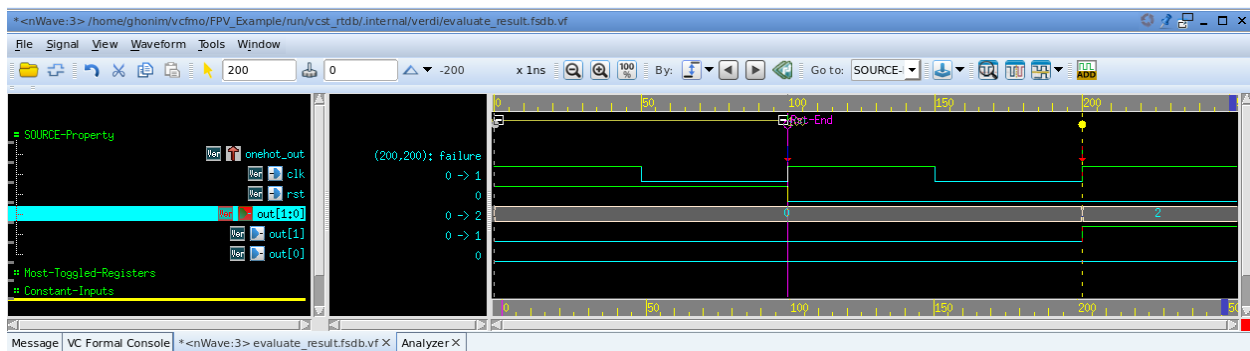



Figure 16. The `out[1:0]` signal expanded.



To check the other falsified property, we can double-click on the second  (row 4) from Figure 13. You should then see a generated waveform like the one shown below:

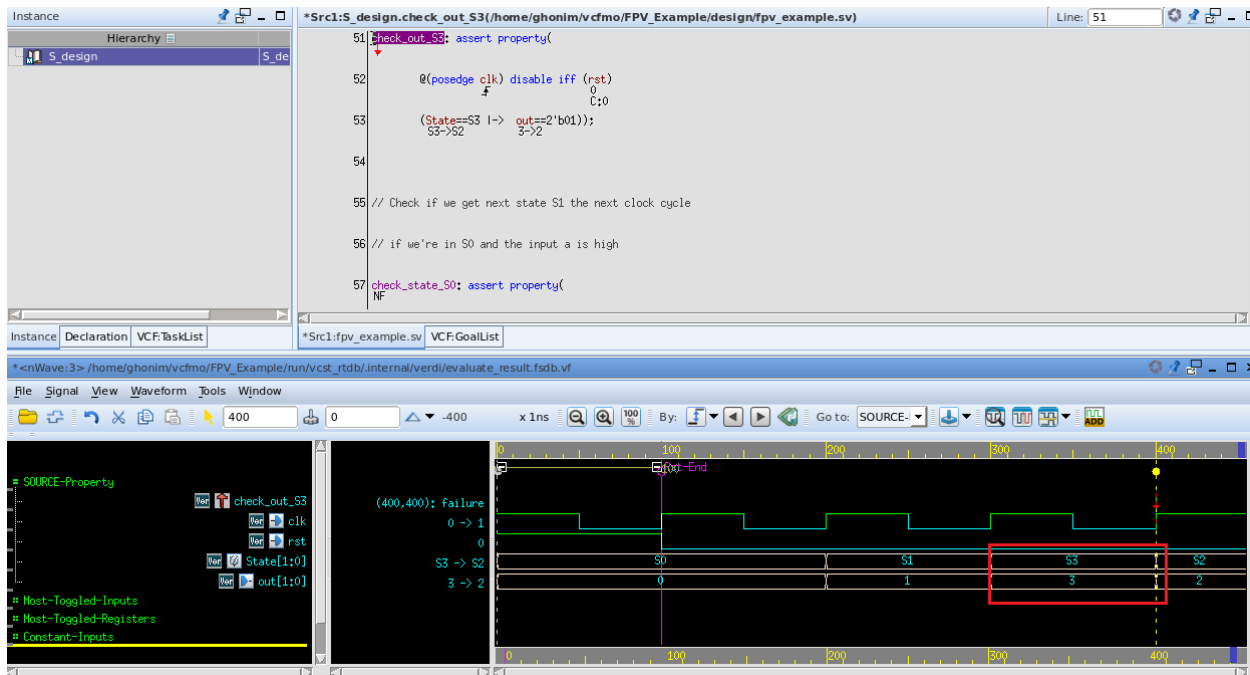


Figure 17. We see here that State S3 has an output of 3 “11”, and not 1 “01” as specified in our assertion, hence it’s falsified.

To gain better understanding of those proved, or falsified signals we can look at the COI “Cone of Influence” digital diagrams.

Right click on the property which you want to see its COI, and click on “Show COI Schematic”.

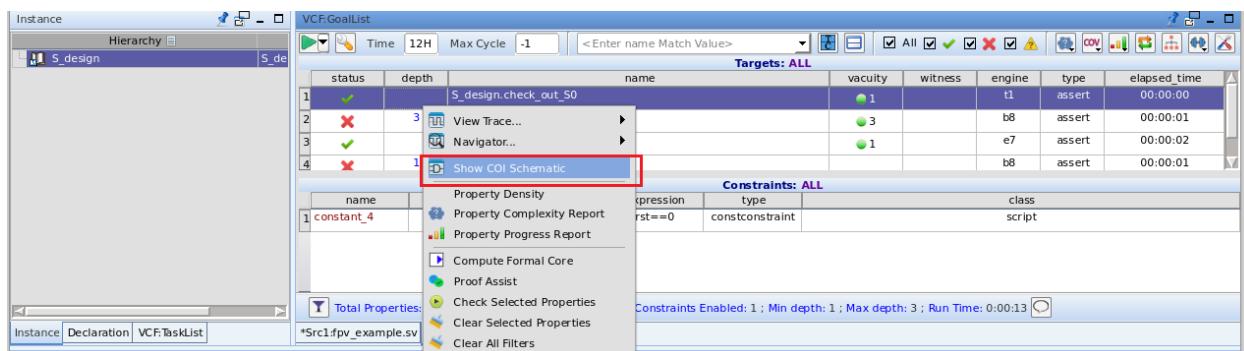


Figure 18. Showing the Cone Of Influence Schematic

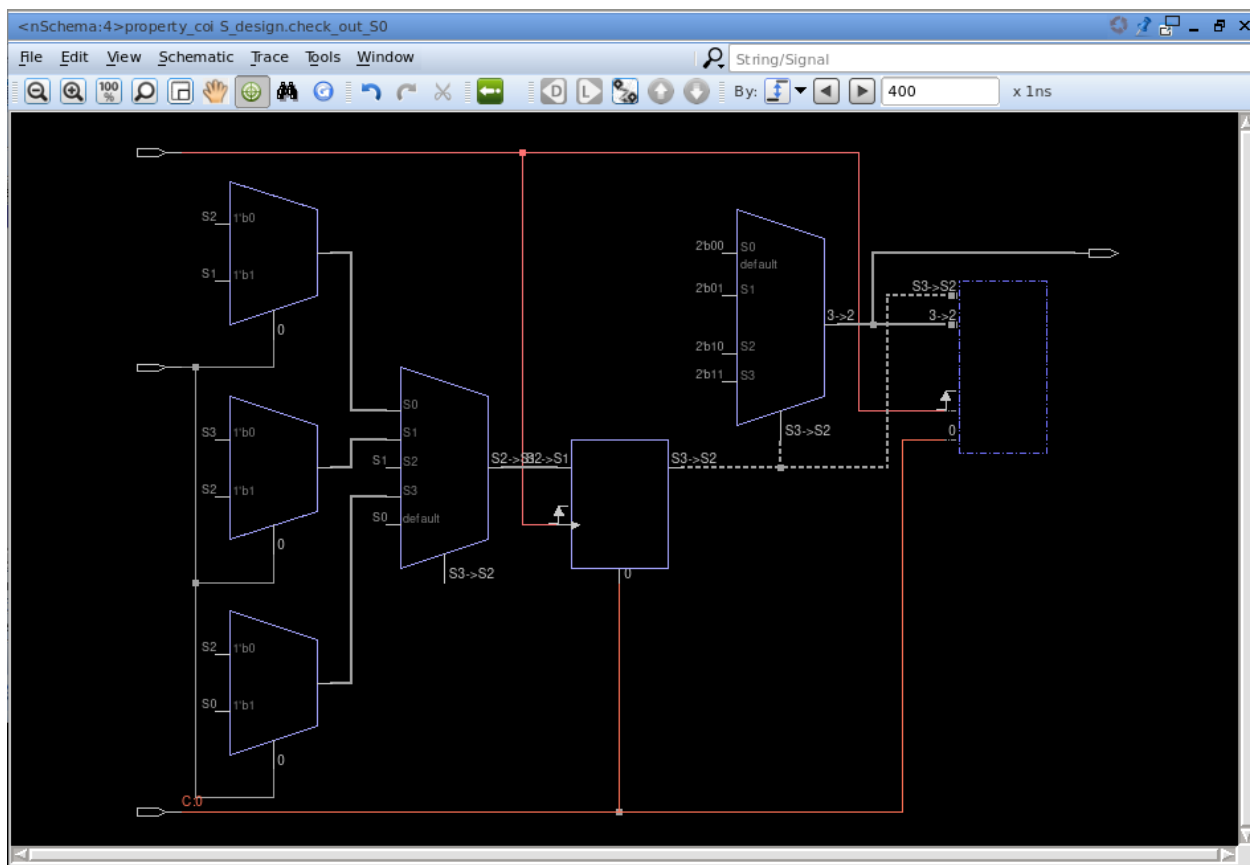


Figure 19. COI Schematic.

And you can do this with the different assertions to help you see the signals and factors affecting the property we're verifying.

## Resolving Errors

To resolve the falsified properties, we need to modify our code such that it behaves as the properties specify, otherwise, we need to rewrite the assertions such that they work for our design.

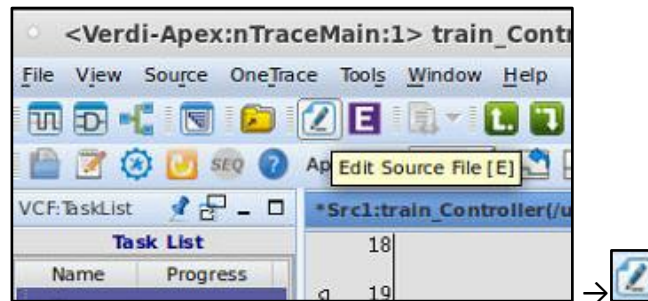


Figure 20. To make changes to the design file in VC Formal, click on "Edit Source File". Don't forget to save your changes.

You can also open the file in any text editor to make the changes, and restart VC Formal to re-run the analysis.

In our case, we modified the output signal such that we have onehot encoding, and the output of state S3 is 2'b01, and we also modified the assertion for the output of S0 to be 2'b01 to maintain onehot encoding.

```

//Setting the state outputs
always_comb begin
  case (State)
    S0: out=2'b01;
    S1: out=2'b01;
    S2: out=2'b10;
    S3: out=2'b01;
    default: out=2'b01;
  endcase
end

// inline SVA (included within the design)

`ifdef INLINE_SVA
// Check if the output is coded using onehot encoding
onehot_out: assert property(
  @(posedge clk) disable iff (rst) $onehot(out));

// Check if the output at State S0 is 2'b00
check_out_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 |-> out==2'b01));

// Check if the output at State S0 is 2'b01
check_out_S3: assert property(
  @(posedge clk) disable iff (rst)
  (State==S3 |-> out==2'b01));

// Check if we get next state S1 the next clock cycle
// if we're in S0 and the input a is high
check_state_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 && a) |-> ##1 (State==S1));


`endif
endmodule

```

Figure 21. Updated fpv\_example.sv file.

Now we run the VC Formal FPV app analysis again, we can do this by closing VC Formal and running it again in the same way, or by click the restart button after updated the SV source file.

## Restarting VC Formal

We can restart VC Formal by clicking on . If you invoked VC Formal along with the TCL script in the one-line command, then VC Formal will automatically load the same TCL file again. If you invoked VC Formal without the TCL script, then you will have to manually load the script again.

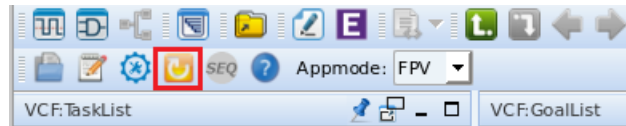


Figure 22. Location of the restart button in VC Formal window.

Click on the green run button again to get the formal property verification results:

**Task List**

Name	Progress	Result
FPV	4:00:00	100.0%

Runtime(HH:MM:SS)  
Total:12H: Run Time:0:00:07  
Goals  
Found:4  
Passed:4 (100.0%)  
Proven:4  
Covered:0  
Failed:0 (0.0%)  
Falsified:0  
Uncoverable:0  
Vacuous:0  
Checking:0  
Inconclusive:0  
Not Run:0  
Disabled:0  
Vacuities:3  
Non-vacuous:3  
Constraints  
Found:1  
Disabled:0

**Goal List**

status	depth	name	vacuity	witness	engine	type	elapsed_time
✓	1	S_design.check_out_S0	1		t1	assert	00:00:01
✓	3	S_design.check_out_S3	3		t1	assert	00:00:01
✓	1	S_design.check_state_S0	1		rp1	assert	00:00:01
✓		S_design.onehot_out			t1	assert	00:00:01

**Constraints: ALL**

name	vacuity	witness	expression	type	class
constant_4			rst==0	constconstraint	script

Total Properties: 4 - passed[4] - failed[0] - disabled[0] ; Constraints Enabled: 1 ; Run Time: 0:00:07

\*Src1fpv\_example.sv VCF:GoalList

Figure 23 .Now we see that all the properties have been proved!

## Appendix: SystemVerilog Assertions (SVA)

---

Assertions are an important tool used to verify the behavior of a design and provide functional coverage information. This involves determining whether the design is working correctly and assessing the quality of the test. Assertions can be checked either dynamically through simulation or statically using a separate tool known as a property checker. This tool is used to confirm if the design meets its specification but may require certain assumptions about the design's behavior to be defined.

In SystemVerilog, there are two types of assertions: immediate (`assert`) and concurrent (`assert property`). Concurrent assertions include coverage statements (`cover property`) and assume property statements, which have the same syntax as concurrent assertions. Another statement called `expect` is utilized in testbenches and checks that some specified activity occurs. All these statement types use sequences and properties to describe the design's temporal behavior over time, defined by one or more clocks.

### Example:

*// Check if the output is coded using onehot encoding (this is just a comment)*

```
onehot_out: assert property(
    @(posedge clk) disable iff (rst) $onehot(out));
```

The first thing you do is to name your property or assertion, you do this by writing that name with no spaces, when putting a colon.

```
Name_of_the_property: assert property( ....);
```

If you are writing an immediate assertion, you use “`assert`”, otherwise, if it’s a concurrent assertion, you should use “`assert property`” and open the brackets where the whole assertion goes. After the brackets, you must put a semicolon.

Now, inside the assertion if you’re writing an immediate assertion, there’s no need to use a clock or reset signals, you write a statement as if it’s an if statement in a `always_comb` block. Immediate assertions are rarely used.

The most common assertions are concurrent assertions, and below are some examples:

```
// Check if the output at State S0 is 2'b00
check_out_S0: assert property(
    @(posedge clk) disable iff (rst)
    (State==S0 |-> out==2'b01));
```

This property takes place at every positive clock edge, and it's disabled (won't happen) if and only if the reset signal (rst) is high, this assertion is disabled (we're not expecting it to hold if we have a reset signal) we're assuming this reset signal is positive edge triggered, as we specified in our design.

If however reset is not asserted, at every clock edge the FPV app will check which state we're in, and if we're in State S0, this implies that the output is 2'b01. We're specifying that this is a property that needs to hold. If it doesn't hold, then something is wrong with our design, or the way we wrote this assertion.

|-> Overlapping Implication operator. The RHS is evaluated at/from the same cycle the LHS is true.

|=> Non-overlapping Implication operator. The RHS is evaluated one clock cycle after the LHS is true.

For example:

```
// if both a and b are high, then ack is high after 2 clock cycles from this cycle.
```

```
Check_ack: assert property (@posedge (clk) disable iff (rst)
    (a && b ) |-> ##2 ack);
```

```
// if both a and b are high, then ack is high after 2 clock cycles from this cycle.
```

Check\_ack: assert property (@posedge (clk) disable iff (rst)

(a && b ) | => ##1 ack);

Those expressions above are equivalent. There is no need to use both the |-> and | => operators, as one of them would be sufficient to convey the assertion meaning as long as we specify the correct number of cycles. To maintain consistency, we recommend using the |-> Overlapping operator.

### Some SVA operators

Operator	Semantics
\$onehot	Returns true if the expression has exactly one bit set to 1. Otherwise, it returns false.
\$onehot0	Returns true if there are zero or one bits set to 1 in the expression.
\$isunknown	Returns true if any bit in the expression is 'X' or 'Z'.
\$stable	Returns true if expression's value remained unchanged; false otherwise.
\$rose	Returns true if the LSB of the expression changes to 1, otherwise false.
\$fell	Returns true if the LSB of the expression changes to 0, otherwise false.
\$past(expression, number of cycles)	This function returns the value of an expression from a specified number of cycles ago.

### Some SVA operators

##n Delay operator, delay n number of cycles.

## [m:n] Delay this fixed time interval from m to n

!, |, && Boolean operators

not, or, and Property operators.

### More SVA examples used in our design:

// Check if the output at State S0 is 2'b00

check\_out\_S0: assert property(

@(posedge clk) disable iff (rst)

(State==S0 |-> out==2'b01));



```
// Check if the output at State S0 is 2'b01
check_out_S3: assert property(

    @(posedge clk) disable iff (rst)
    (State==S3 |-> out==2'b01));

// Check if we get next state S1 the next clock cycle
// if we're in S0 and the input a is high
check_state_S0: assert property(

    @(posedge clk) disable iff (rst)
    (State==S0 && a) |-> ##1 (State==S1)
```