

This RAK demonstrates JasperGold FPV App. This lab focuses on using System Verilog Assertions (SVA) to perform verification using JasperGold Formal Property Verification App. The SVA code that you will write should show how to perform meaningful verification while using just very basic operators in the language. With the code loaded into the tool, you will see the different possible outcomes as a formal proof runs on the properties, such as full proofs, undetermined results, and counterexamples.

1. Setting up JG:

First you need to setup your environment. Set the path to pick the JG executables as shown below:

```
% set path = (<jaspergold_install_dir >/bin $path)
```

2. Preparing the Lab:

- Begin this lab in your shell window in a writable location.
- Copy the example directory from the tool installation directory:

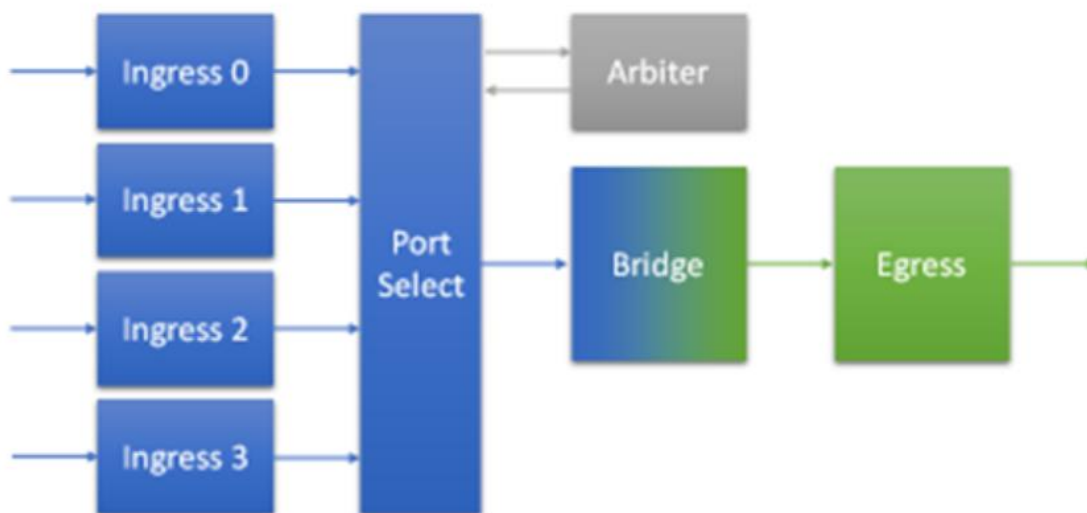
```
$cp -r <jaspergold_install_dir >  
/doc/example_jaspergold_apps/designs/reference_design/verilog_sva .
```
- Change to this directory

```
$ cd verilog_sva
```

3. Reference Design:

The design used in this lab is a 4-to data path multiplexer with arbitration. Data is entering on Ingress port 1 to 4 and leaving on Egress port. The data flow is controlled with a request-grant handshake.

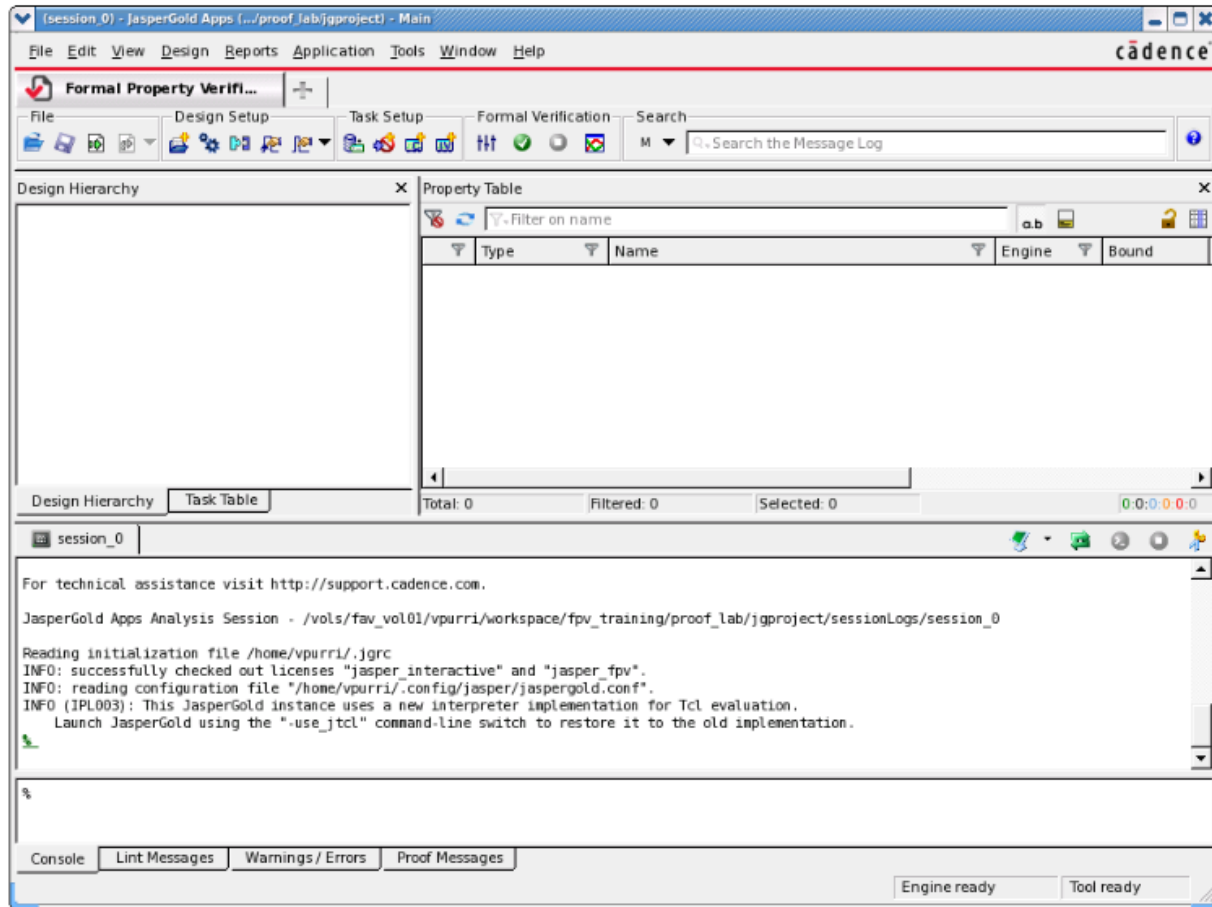
Figure 1 Reference Design



4. Invoking the Tool

Launch JasperGold Apps with the following command: `$jg`

Figure 2 JasperGold Apps Console



See the JasperGold Apps User Guide for a detailed description of the JasperGold Apps console. You will see that a directory called `jgproject` is automatically created in your working directory. It contains information such as log and command files, saved sessions e.t.c.

5. Analyzing Design Files

The first operation consists of analyzing design and property files. You can use the GUI to trigger these commands, type them directly on the command line, or simply source a Tcl script.


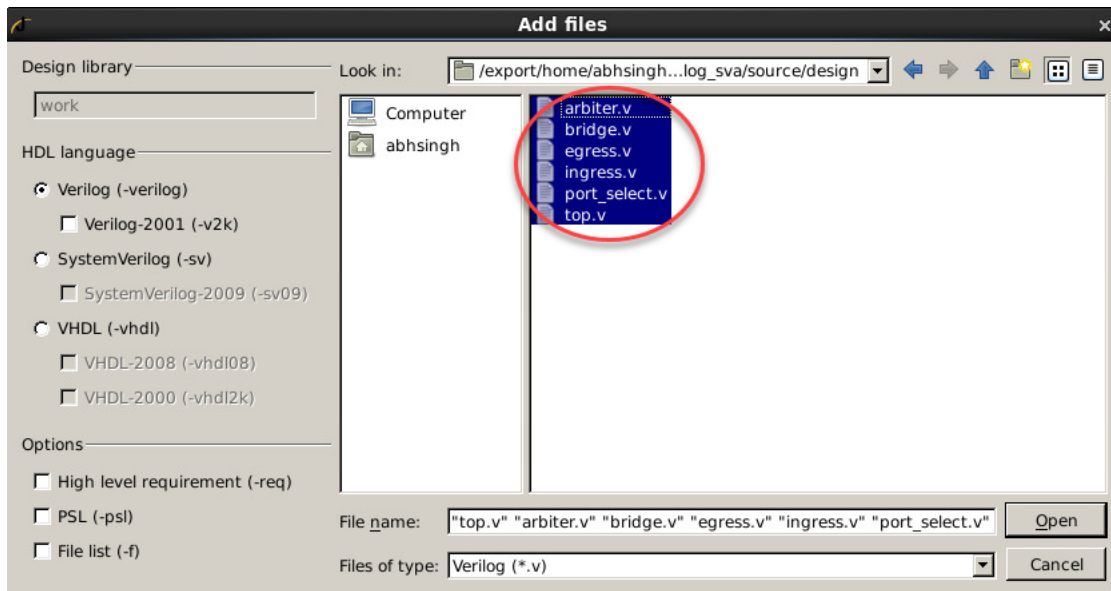
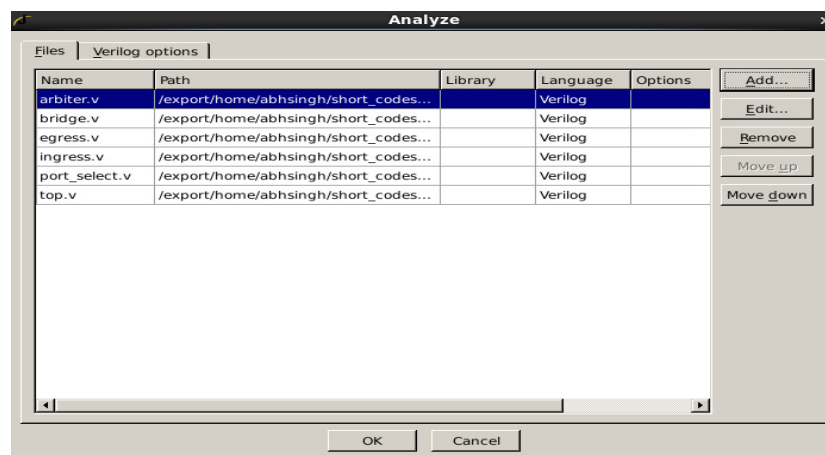
Click the Analyze wizard button. Click the  Analyze wizard button then click Add. The Add files dialog appears as below (see Figure 3). Go to the source/design directory, click the Verilog or VHDL radio button, depending on the language that you selected for this lab, select all the files in the directory and click Open.

Figure 3 Adding Design Files



The Add files dialog closes, and the Analyze dialog is active (see Figure 4).

Figure 4 Analyzing Design and Property Files



Click ok. JasperGold Apps analyzes the design and property files. While parsing, the tool displays info, warning, and error messages if applicable.

6. Elaborating Design Hierarchy

The next procedure consists of elaborating the design hierarchy. At the same time, JasperGold Apps identifies, and extracts properties embedded in the design and associated verification modules.


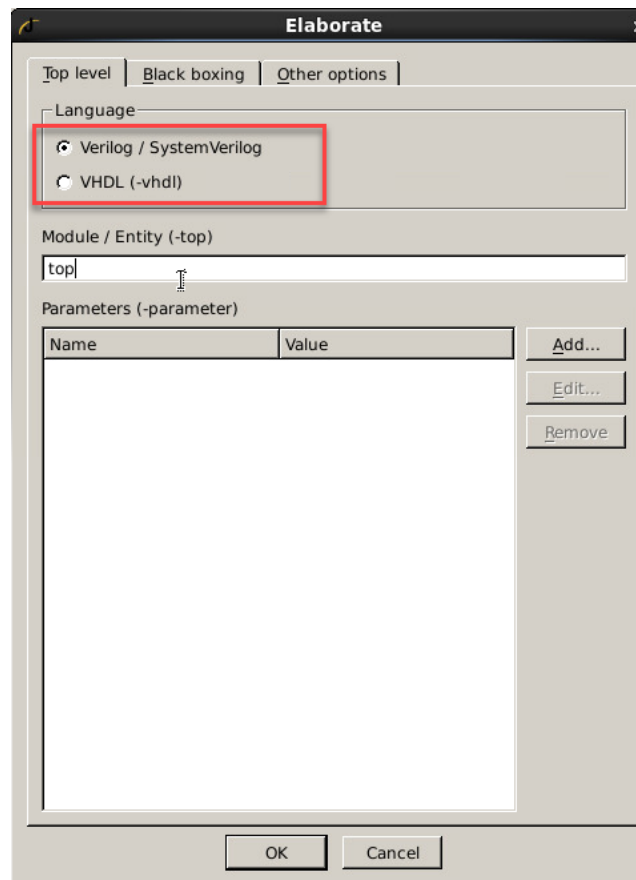
Click the  Elaborate wizard button. The Elaborate dialog appears (see Figure 5). Type top in the Module / Entity field, click the Verilog or VHDL radio button and click ok.

Figure 5 Elaborating Dialog

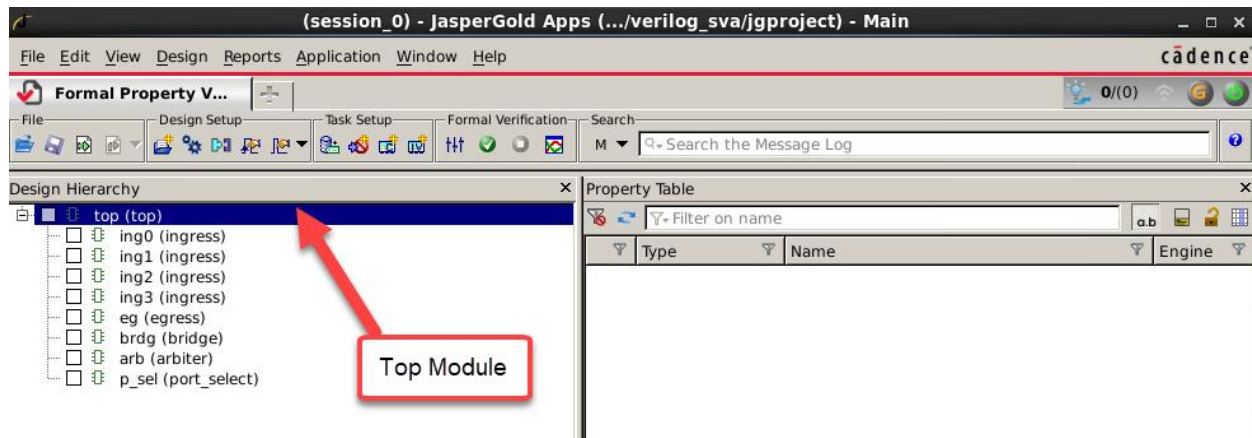


JasperGold Apps elaborates the design hierarchy, synthesizes the netlist, and extracts the embedded properties.

- **Reviewing the design information**

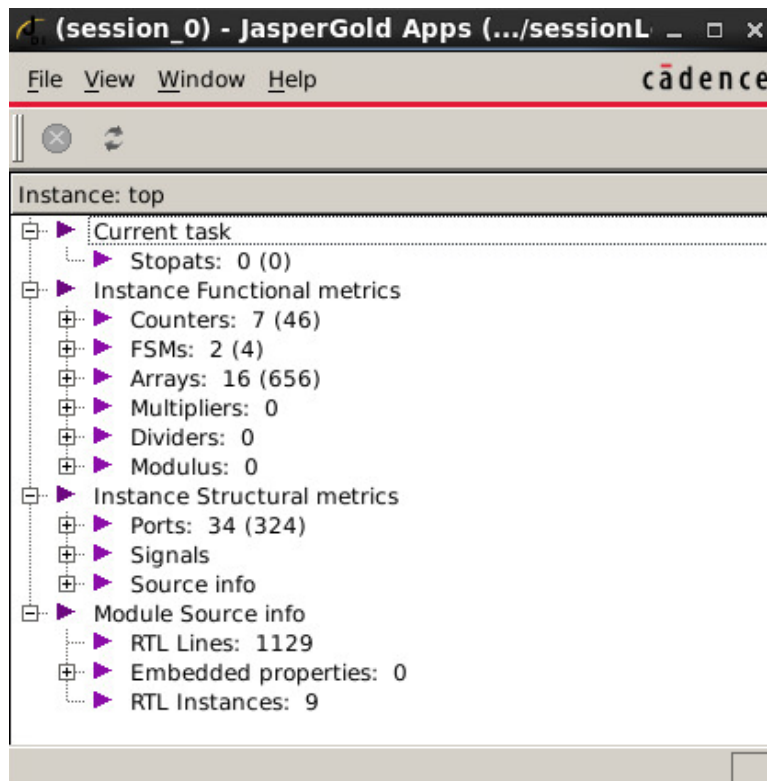
In this step we review the design information of the analyzed RTL

Figure 6 Main Console after loading the design



Click on the design module top in the Design Hierarchy pane. Right-click and choose Design Information. The Design Information window appears (see Figure 7).

Figure 7 Design Information



Click on the + icons to expand Module Source info, Instance Structural metrics, and Instance Functional metrics. This allows to inspect the details of the design loaded into the tool. From here you could start the visualization of the design (e.g. showing a transition into a certain FSM state), after you performed the reset analysis (next section)

7. Specifying the Clock

In this step, you will specify the clock for the design. The tool will help you with that.

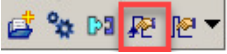
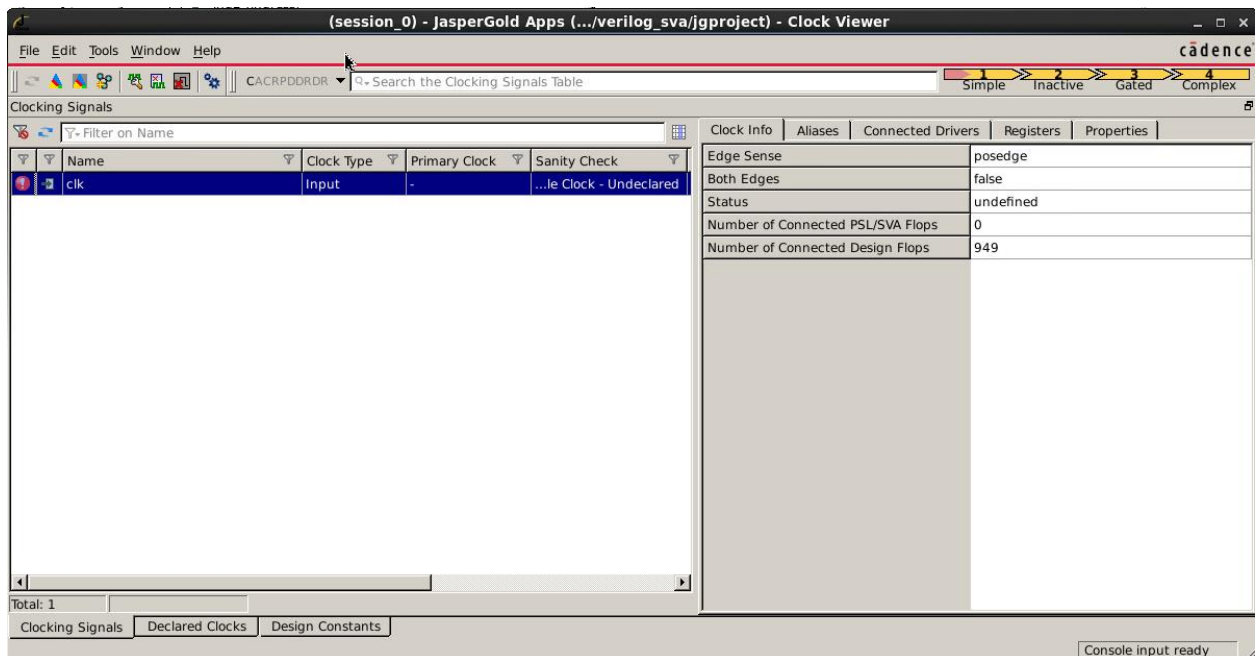
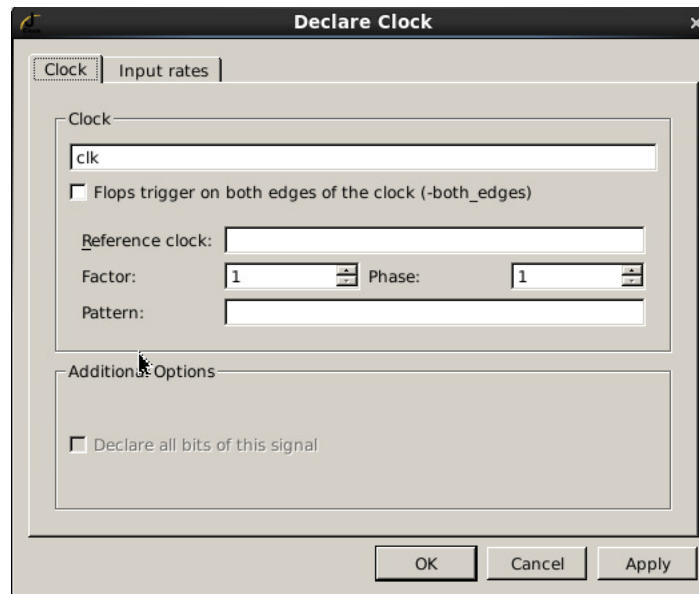
Click the  Clock Viewer button. The Clock Viewer will open. This GUI shows you all the clocking signals (i.e. signals used as clocks) in the design, together with information about how many properties and flops they clock, and whether they are connected to a declared clock.

Figure 8: Clock Viewer GUI



The design only has one clocking signal clk, which has to be declared as clock. For that we will use the GUI. Right-click clk and select Declare Clock. The clock declaration dialog appears.

Figure 9 Clock Declaration GUI



Click ok. As an alternative to the GUI, you could type the following command to specify the clock:

```
% clock clk
```

After specifying the clock, the Clock Viewer GUI will be updated, showing that all clocks have been declared

Note: Before performing any formal verification task, you must declare a clock in JasperGold. This can be done in many different ways:

Use `clock clk` to specify an input as clock

Use `clock -none` to specify that no signals shall be used as clocks

Use `clock -infer` to automatically infer a signal as clock

8. Specifying the Reset

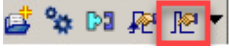
Next, you will specify the reset expression for JasperGold Apps to initialize the design. The tool will also help you with that.

On the command line in the JasperGold console, type the following:

```
% reset -analyze -synchronous
```

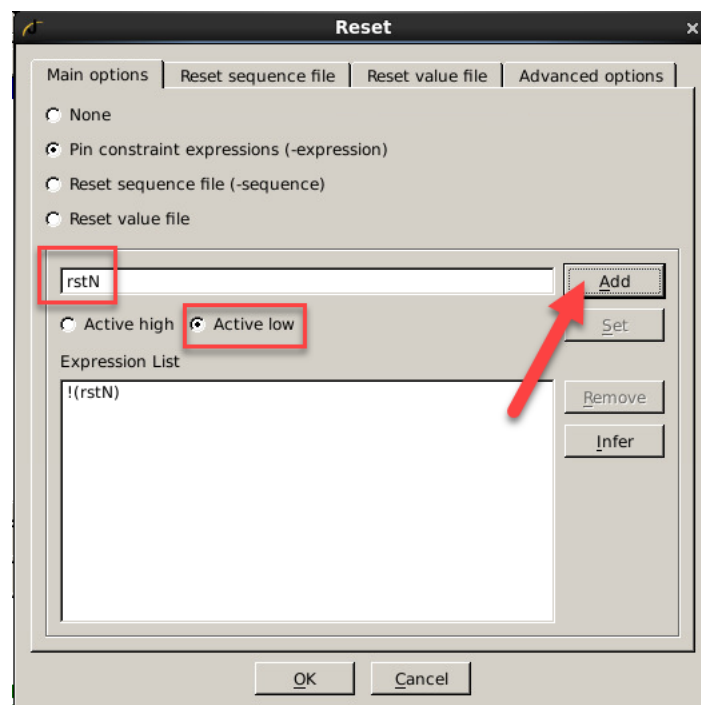
This command analyzes the design's reset logic and returns information about synchronous and asynchronous reset signals. For this design, you can see that there is only one reset signal and the signal name is negated to show that it is active-low:

Signal Name: ~rstN
Type: Simple reset - driven by input
Design flops connected: 70 (813)
PSL/SVA flops connected: 0

Click the  Reset wizard button. The Reset dialog appears.

Select Pin constraint expressions (-expression). Type the proposed reset signal rstN, Select Active low and click Add.

Figure 10 Specifying the Reset Expression



Click ok.

As an alternative to the GUI, you could type the following command to specify the reset expression:


```
Verilog:% reset ~rstN  
VHDL: % reset {not rstN}
```


9. Tcl Script

You can do all the above steps (5,6,7,8) by writing a simple Tcl script which will load the design files, compile the design, and set up clocks and resets. Use your favorite text editor and create a file run.tcl with the following contents:

```
# Clear the environment
clear -all
# Analyze design files
analyze -verilog [glob source/design/*.v]
analyze -sv [glob source/properties/*.sva]
# Elaborate design and properties
elaborate -top top
# Set up Clocks and Resets
clock clk
reset ~rstN
```

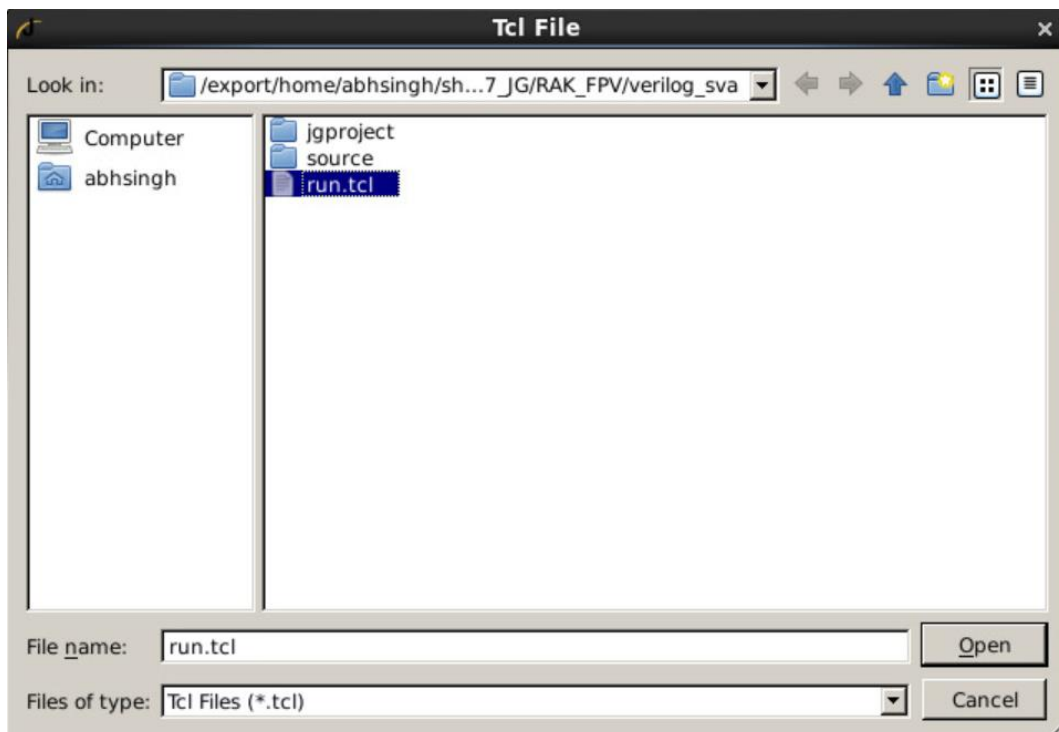
- **Load the design and assertions using tcl:**

In the JasperGold window, click the  Source button. Select and open run.tcl

NOTE: You can also load a Tcl script as you launch JG by running it as:

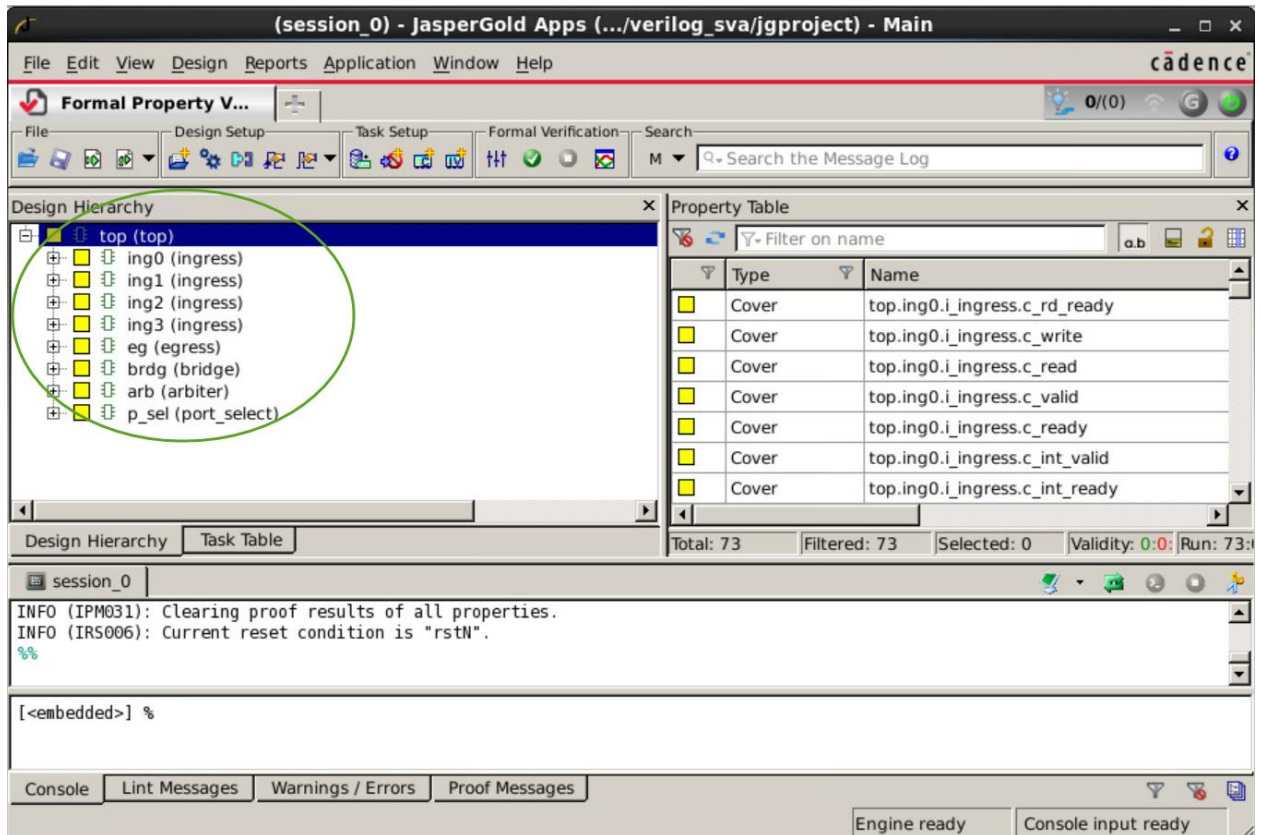
```
jg run.tcl
```

Figure 11: Source Tcl File Dialog



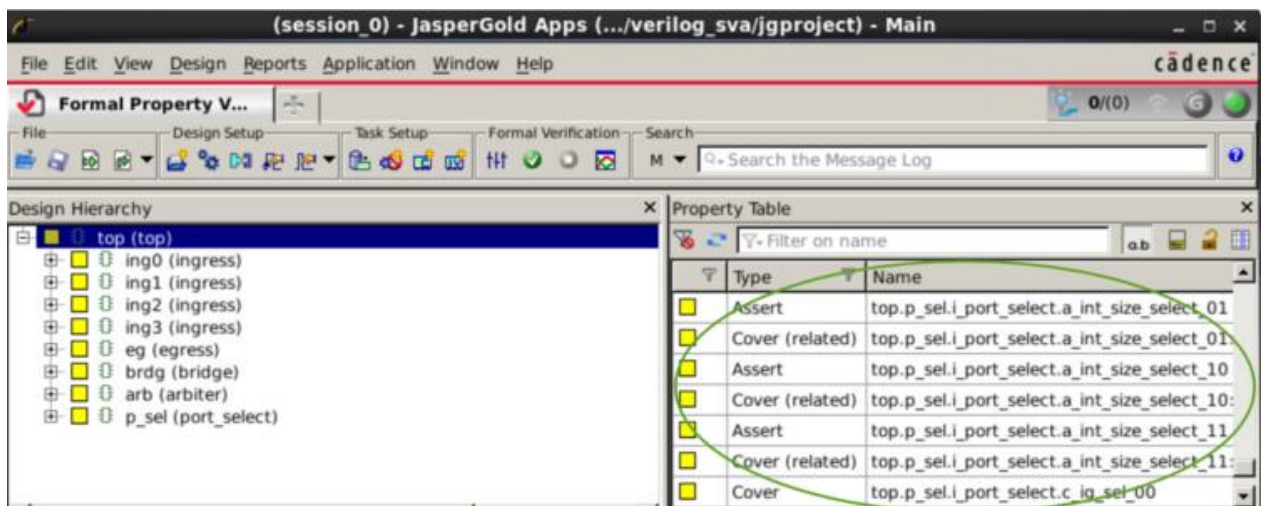
After JasperGold loads the design, you should see the design hierarchy on the left pane

Figure 12: JasperGold Apps Console after loading design



The assertion (and a cover for its precondition) also appears in the Property Table:

Figure 13: Property Table



10. Design Constraints

Sometimes design requires constraint on some signals (e.g. side band signals) to a fixed value. You can write those constraints either in the design or your run.tcl script. Here is the example:

```
% assume {ing2.int_valid == 1'b0}  
% assume {ing3.int_valid == 1'b0}
```

If you apply, then these two constraints will appear in the Property Table with the names `assume:0` and `assume:1`. We are not using them in this lab.

11. Running formal proof

Now you will prove the properties using JasperGold's formal engines.


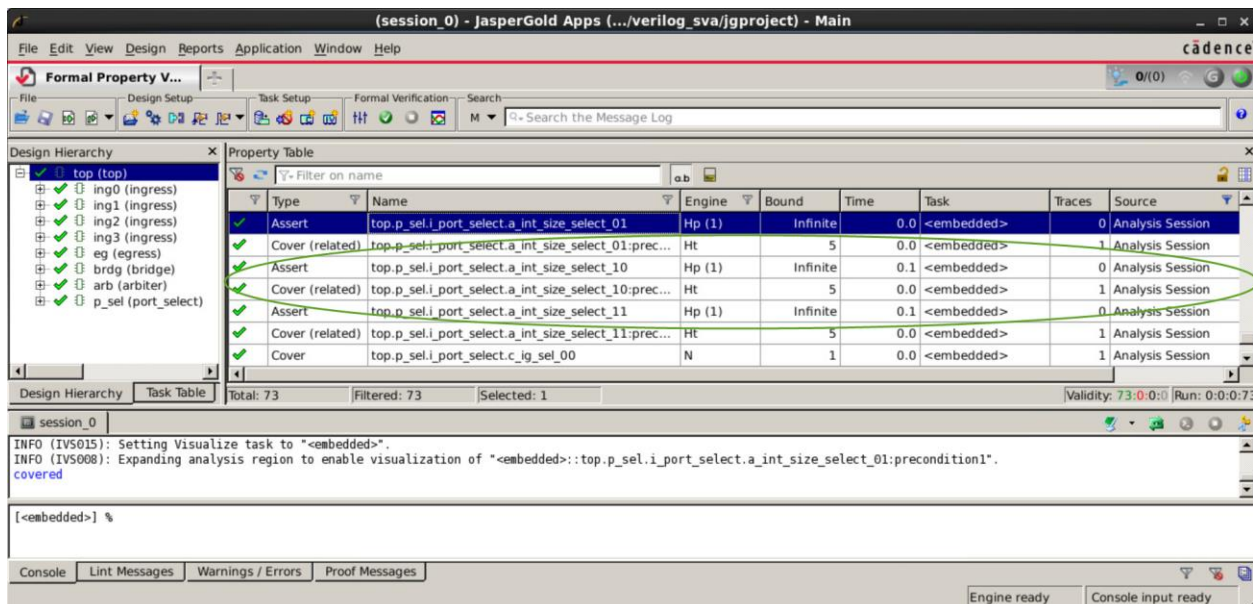

Click the  Prove All button to run a formal proof.

Figure 14: Property Table after proving assertion



The assertions were fully proven, indicated by the  icon. This means that the assertion is always true in this environment and can never be violated for all possible stimulus driven into the design. In formal verification, this is the best possible result.

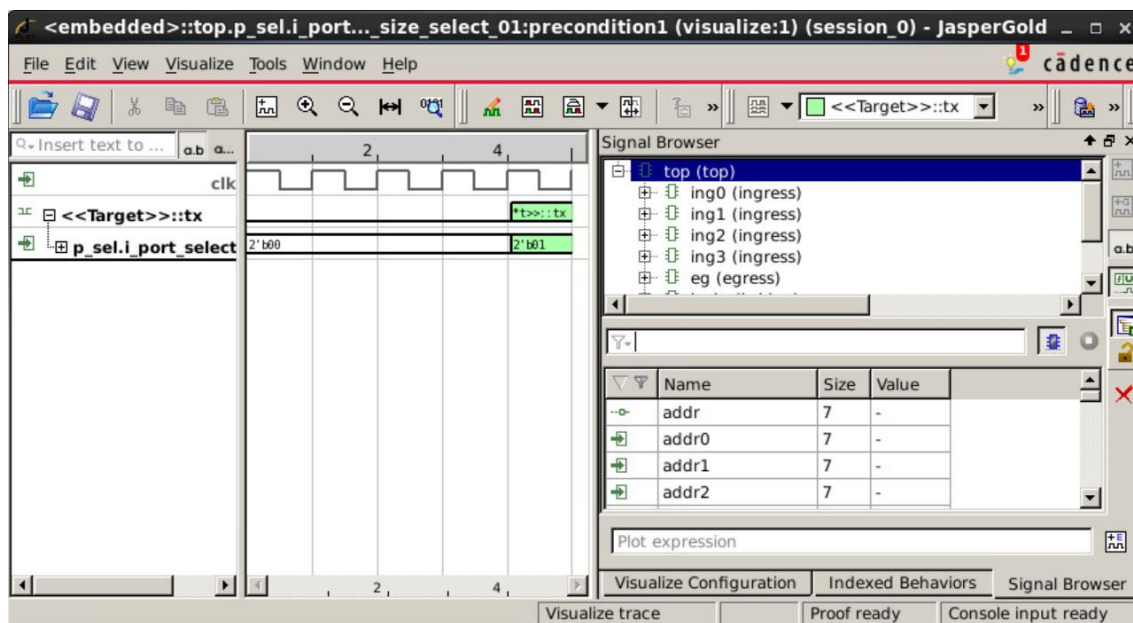
For fully proven assertions, no waveform is available. Double-clicking on the assertion shows a message to confirm that:

Figure 15: Message informing user that waveform is not available



This behavior is also true for unreachable covers, and undetermined assertions or covers. The cover property in the environment captures the precondition for the assertion. You can see that it is also passing, indicating that the formal engines were able to find a stimulus that hits this cover in 5 cycles. A waveform showing this is available via double-click or through the option View Cover Trace on the right-click menu.

Figure 16: Precondition cover waveform for output sequence assertion



A passing precondition cover is also a good sign. It means that formal is able to reach a condition where the assertion triggers, confirming that you are actually testing the design.

It is common to stumble upon unreachable precondition covers as a formal testbench is being developed. This means that the corresponding assertion is not triggering at all and has to be investigated. The most common cause is when the environment is over-constrained (i.e. assumptions are too restrictive), mistakes/typos while coding the assertion, or simply a design bug (e.g. deadcode).

12. Adding new properties

As above, all the properties are passing so we will add some new properties to see the behavior of failing properties. Use your favorite text editor and create a file props.sva with the following contents:

```
module props (
    input clk, rstN,
    input eg_valid, eg_ready,
    input valid0, valid1, valid2, valid3,
    input ready0, ready1, ready2, ready3,
    input [1:0] size0, size1, size2, size3
);
    reg eg_is_odd;

    always @(posedge clk or negedge rstN)
        if (!rstN) eg_is_odd <= 1'b1; // first is odd
        else if (eg_valid && eg_ready) eg_is_odd <= ~eg_is_odd;

// Byte count logic
    reg [6:0] byte_cnt, byte_cnt_nx;
    always @(posedge clk or negedge rstN)
        if (!rstN) byte_cnt <= 0;
        else byte_cnt <= byte_cnt_nx;
    always_comb begin
        byte_cnt_nx = byte_cnt;
// Increment if bytes are valid at ingress
        if (valid0 && ready0) byte_cnt_nx = byte_cnt_nx + (1 << size0);
        if (valid1 && ready1) byte_cnt_nx = byte_cnt_nx + (1 << size1);
        if (valid2 && ready2) byte_cnt_nx = byte_cnt_nx + (1 << size2);
        if (valid3 && ready3) byte_cnt_nx = byte_cnt_nx + (1 << size3);
// Decrement after every data word on egress
        if (eg_valid && eg_ready && !eg_is_odd) byte_cnt_nx =
            byte_cnt_nx-1;
        end
        ast_output_count: assert property (
            byte_cnt == 0 // if there are no outstanding bytes...
            |-> // then in the same cycle
            !eg_valid // eg_valid must be low
        );
    endmodule

// Connect module to design
    bind top props props_i (.*);
```

The assertion will simply check that if the counter is zero, then the egress port should not output any bytes.

Modify the file run.tcl to load the file props.sva:

```
# Clear the environment
clear -all
# Analyze design files
analyze -verilog [glob source/design/*.v]
# Analyze SVA file
analyze -sv props.sva
# Elaborate design and properties
elaborate -top top
...
```



Click the Source Recent Script button to reload the Tcl script. The formal proof should find a counterexample for the assertion ast_output_count.

Figure 17 Property Table after initial proof of ast_output_count

Property Table					
		Filter on name		a.b	
▼	Type	Name	▼	Engine	▼ Bound
✓	Assert	top.props_i.ast_output_sequence		N (7)	Infinite
✓	Cover (related)	top.props_i.ast_output_sequence:precondition1		Ht	5
✗	Assert	top.props_i.ast_output_count		B	16
✓	Cover (related)	top.props_i.ast_output_count:precondition1		Hp	1

13. Counterexample(CEX) Debug

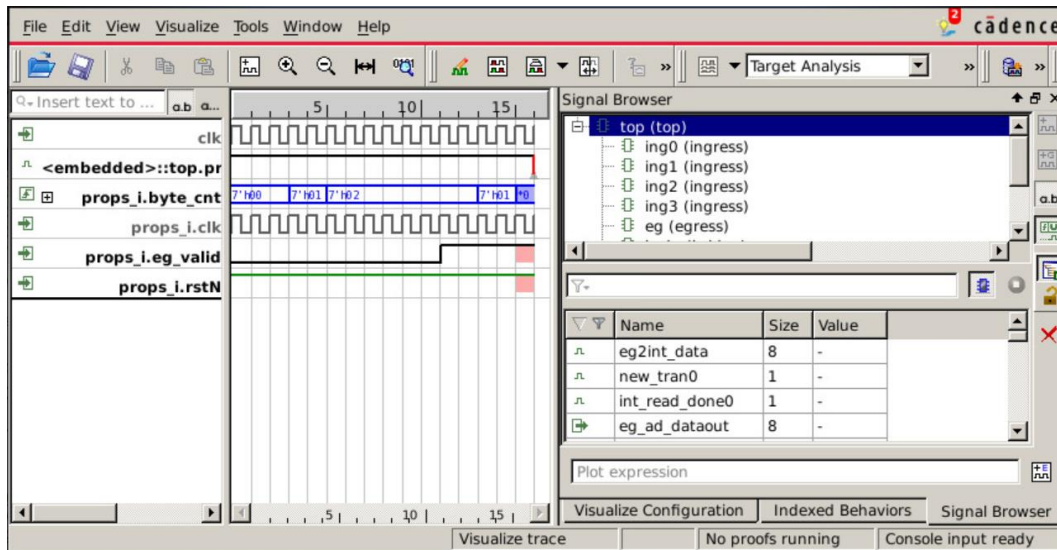
The Visualize environment is used for debugging counterexamples. The debugging flow is as follows:

- Use 'Why' to understand assertion failure
- Use Highlight Relevant Inputs to identify root cause
- Use QuietTrace to get a simplified counterexample

Out of all these steps, the usage of 'Why' is typically present in every debugging session. Debugging CEX using 'Why' method is also known as root cause analysis. The other features are used depending on the complexity of the counterexample. In the current folder

Double-click the assertion ast_output_count in the Property Table to open its counterexample in Visualize

Figure 18: Counterexample from Property Table

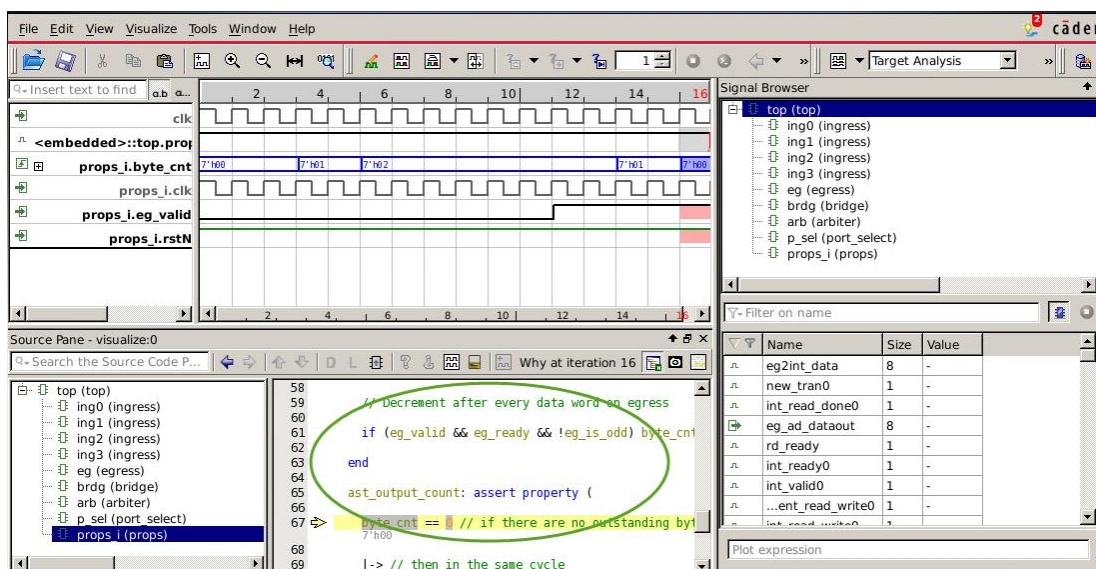


Use Why to understand assertion failure:

This counterexample will do 'Why' operations to understand why eg_valid is high while there are no outstanding requests.

Double-click the assertion ast_output_count at the last cycle (where it fails) to perform Why on it. The Why result will show that the assertion is failing because props_i.byte_cnt is zero and eg_valid is not zero.

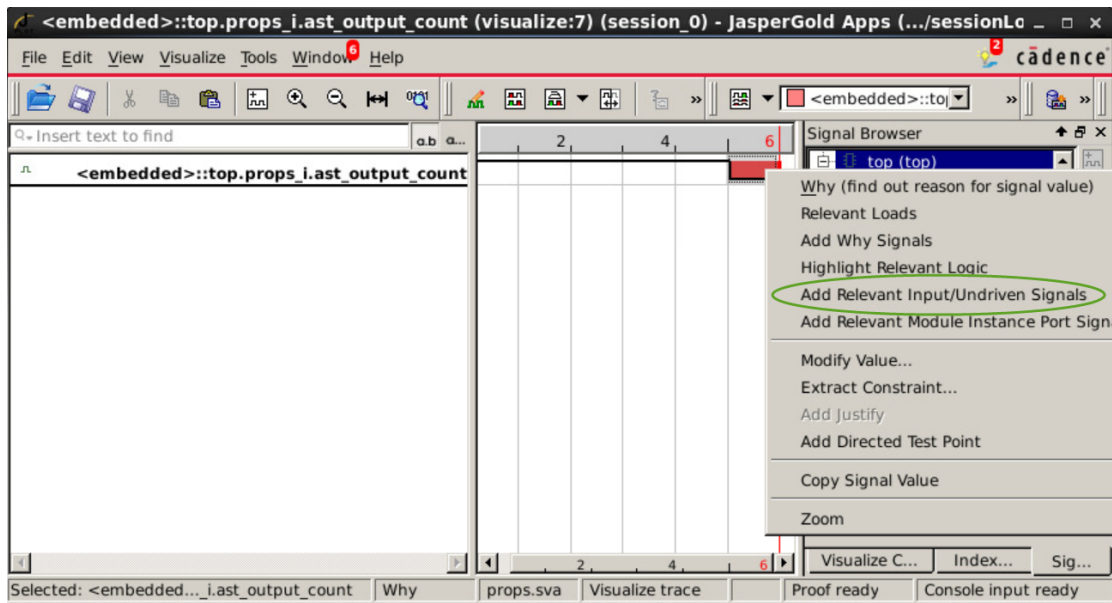
Figure 19: Using Why to debug assertion



Use Highlight Relevant Inputs to identify root cause

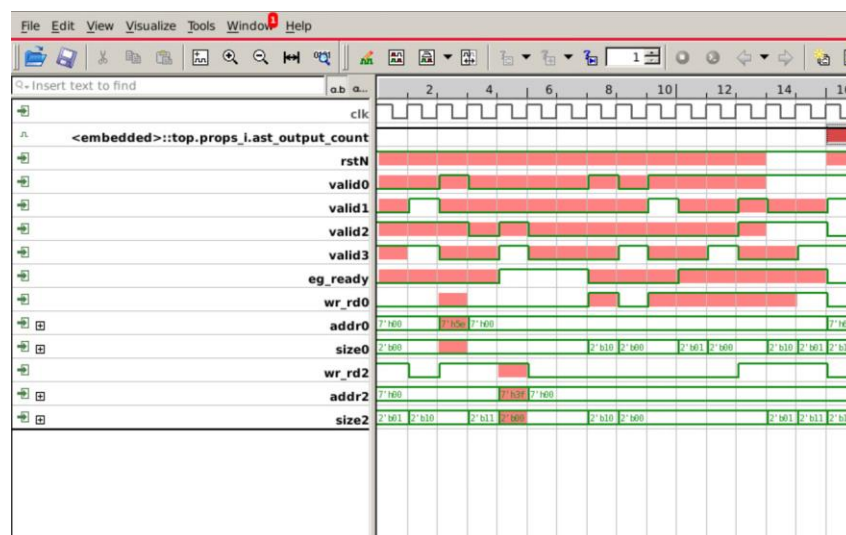
Now, remove all signals from the Visualize window, except for `ast_output_count` (Tip: select the first signal after `ast_output_count`, hold the Shift key, then select the last signal in the waveform, hit the Delete key). Right-click the assertion `ast_output_count` at the last cycle (where it fails) and select Add Relevant Input/Undriven Signals.

Figure 20: Adding relevant inputs affecting `ast_output_count`



Only relevant inputs contributing to `ast_output_count` will be plotted, with highlighting to tell you the exact cycles where they are relevant. Note that this is different than the previous step using Add Transitive Fanin to Inputs, as that method adds all inputs, instead of only the relevant ones.

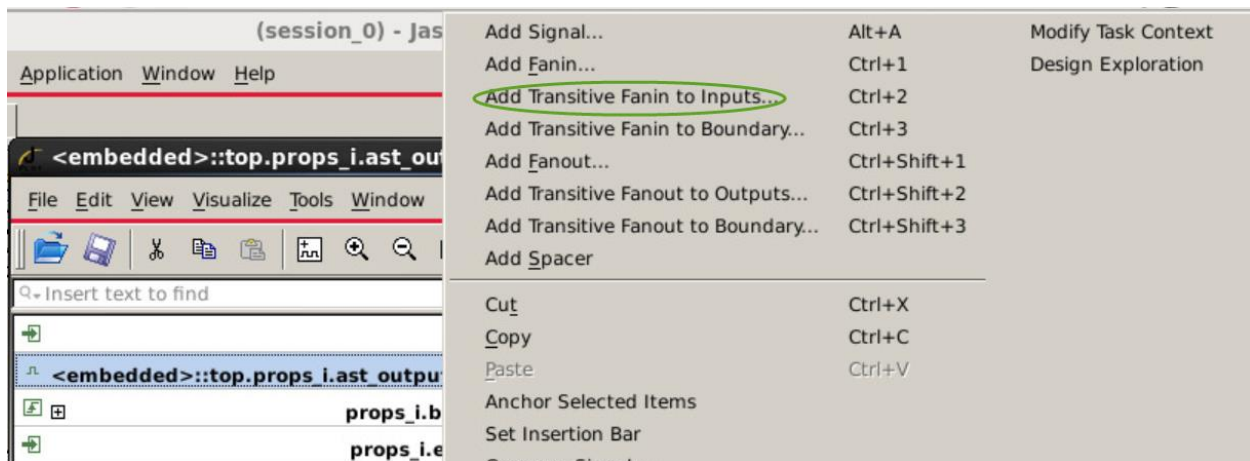
Figure 21: Relevant inputs of `ast_output_count`



Use QuietTrace to get a simplified counterexample:

In Visualize, right-click the assertion `ast_output_count` and select Add Transitive Fanin to Inputs...


Figure 22: Adding inputs connected to `ast_output_count`





The Add Transitive Fanin dialog will show up. It contains all the inputs that are structurally connected to the assertion.

Click OK .

All the inputs will be plotted. Notice all the activity on them. Now you will apply QuietTrace on this counterexample, which should change this stimulus.

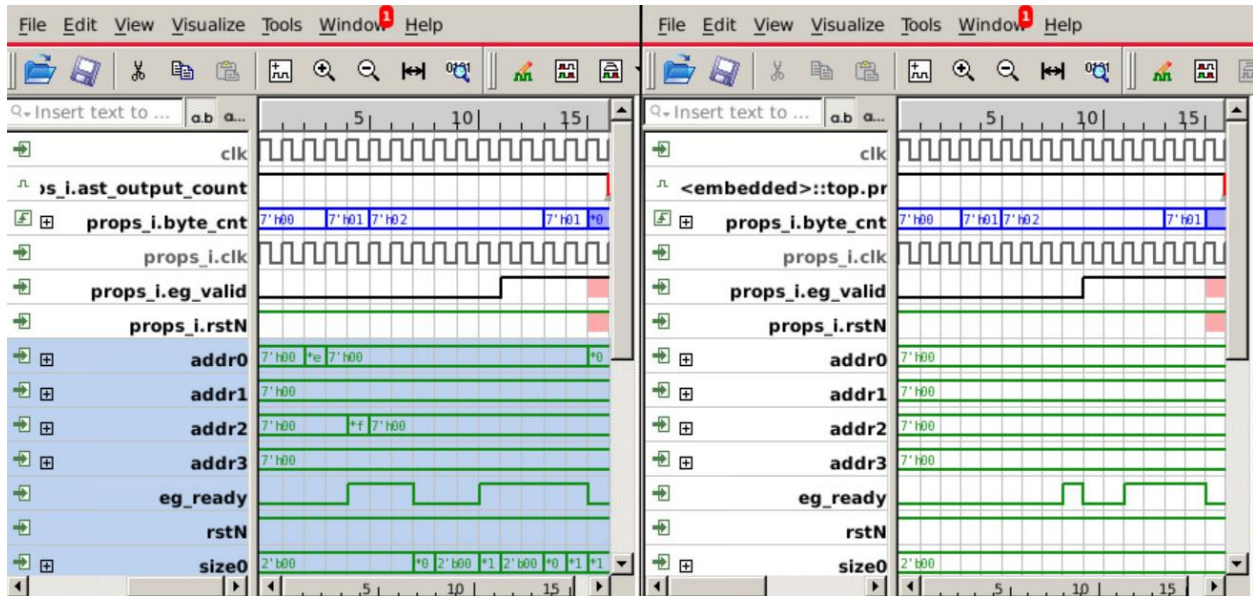
Click the  Clone button to clone the waveform

On the new Visualize window that just appeared, click the  QuietTrace button to enable QuietTrace

Press F5 or click  Replot to apply the new configuration to the waveform

Comparing the two waveforms side by side clearly shows the impact of QuietTrace: the stimulus with QuietTrace is much simpler than without.

Figure 23: Comparison between non-quiet (left) and quiet (right) counterexample



When you enable QuietTrace and Replot, the waveform will get updated automatically as soon as QuietTrace is done. In case you start to debug the original/non-quiet waveform, this update might interrupt your debugging process. Cloning the waveform and applying QuietTrace on the clone is useful to such interruption, as the update will happen on the cloned waveform only, not the original one.

QuietTrace and Highlight Relevant Inputs will make it easy to see that the wr_rd signals would be contributing to the failure. This will simplify the waveform even before you do any debug.

To move past this design bug, you will overconstrain the environment to avoid reads altogether. That will allow searching for bugs in the write logic while the read logic is waiting to be fixed by a designer.

14. Adding More Constraints During Debug

You will now add an assumption to disable reads at the ingress interfaces, to work around the design bugs found in the previous section.

There are two options on where to put this assumption:

1. In the SVA file props.sva
2. In the Tcl file run.tcl

Both would work, but we will follow this guideline to decide:

- Tie-off assumptions go in the Tcl file
e.g. disable scan mode, disable test mode, set static operation mode
- Temporary assumptions go in the SVA file
e.g. temporarily disable a feature or interface for directed testing

- Permanent assumptions go in the SVA file
e.g. protocol rules

Since this assumption is temporary, it will go in the Tcl file. Modify the file run.tcl to include assumptions to disable reads:

```
...
# Set up Clocks and Resets
clock clk
reset ~rstN
# Disable reads
assume -name no_read_0 {valid0 |-> wr_rd0}
assume -name no_read_1 {valid1 |-> wr_rd1}
assume -name no_read_2 {valid2 |-> wr_rd2}
assume -name no_read_3 {valid3 |-> wr_rd3}
# Prove all properties
prove -all
```

Click the Source Recent Script button to reload the Tcl script. These four assumptions should now appear in the Property Table:

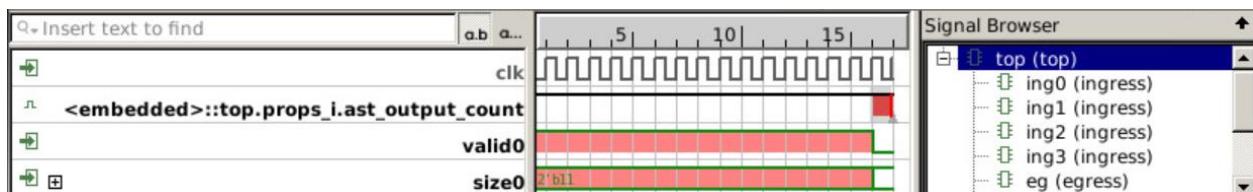
Figure 24: Property Table after disabling reads

Type	Name	Engine	Bound
Assert	top.props.l.ast_output_count	B	17
Assume	no_read_3	?	
Assume	no_read_2	?	
Assume	no_read_1	?	
Assume	no_read_0	?	

Total: 8 Filtered: 8 Selected: 1

As the formal proof finishes, it will find a new counterexample for ast_output_count. Debugging that counterexample shows a different issue now: the size signals are not properly constrained:

Figure 25: Result of Highlight Relevant Inputs on ast_output_count



The specification says that their legal values are only 2'b00, 2'b01, and 2'b10. However, formal is driving values 2'b11, and they are causing the design to misbehave and violate the assertion. In this case this is not a design bug, but rather an environment bug, because formal is not driving the size signals correctly

You now need to add another assumption to properly constrain the size signals. Since this is a permanent protocol constraint, let's add it to the SVA file this time. Modify the file props.sva to include assumptions to constrain the size:

```
module props (  
    input clk, rstN,  
    input eg_valid, eg_ready,  
    input valid0, valid1, valid2, valid3,  
    input ready0, ready1, ready2, ready3,  
    input [1:0] size0, size1, size2, size3  
);  
    ...  
  
    // Size cannot be 2'b11  
    asm_size0: assume property (valid0 |-> size0 != 2'b11);  
    asm_size1: assume property (valid1 |-> size1 != 2'b11);  
    asm_size2: assume property (valid2 |-> size2 != 2'b11);  
    asm_size3: assume property (valid3 |-> size3 != 2'b11);  
  
endmodule  
// Connect module to design  
bind top props props_i (.*);
```

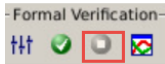
Modify the file run.tcl to run a formal proof automatically after the design is loaded and include settings for time limit.

```
.....  
# Prove all properties  
prove -all -time_limit 3m  
.....
```



Click the Source Recent Script button to reload the Tcl script

The formal proof will run for a long time now, up to the time limit specified in the environment. You can stop the proof so we can move to the next step.



Click the stop All Jobs button to stop the formal proof. The property will be left in the undetermined state, with a proof bound of about 18-23 cycles, depending on how long you let it run:

Figure 26: Property Table after stopping proof

Type	Name	Engine	Bound
? Assert	top.props.i.ast_output_count	B	23 -
✓ Cover (related)	top.props.i.asm_size3:precondition1	Hp	1
● Assume	top.props.i.asm_size3	?	
✓ Cover (related)	top.props.i.asm_size2:precondition1	Hp	1
● Assume	top.props.i.asm_size2	?	

The development of this assertion uncovered one design bug and one environment bug. After adding a temporary constraint to work around the design bug and a protocol constraint to fix the environment bug. However, in this case the assertion is not fully proven within the time limit either.

Having assertions in undetermined state (? icon) is very common in formal testbenches, and it is also called “bounded proof”, as the tool guarantees that the assertion holds for up to a number of cycles – this number is shown in the Bound column. For example, in the figure above, the tool is showing that any failure for the assertion would take 23 cycles or more, which means that the proof bound (number of cycles that the assertion is guaranteed to hold) is 22 cycles.

There are several techniques and methodologies to sign off on formal verification using bounded proofs. The most important one is to have cover properties in the environment to check whether critical design behavior is covered within the proof bound. This will be covered in the next section. Another important sign-off criteria for formal testbenches is to validate the constraints being used. Since we did add two sets of constraints in this section, validating them is necessary. This will also be covered.

Note: No CEX found till this time limit. If you want to debug it further, follow the above steps.

15. Adding cover properties

Similar to simulation, coverage is also very important in formal. The simplest and easiest way to assess coverage is to include cover properties in the testbench, and make sure that the formal engines are hitting them.

JasperGold will automatically create cover properties for every assertion written using the implication operators \rightarrow and \Rightarrow . Those cover properties will target the left-hand side of the implication, and are called precondition covers:

Figure 27: Precondition cover

Type	Name	Engine	Bound
✓ Cover (related)	top.props.i.ast_output_sequence:precondition1	N	5
✓ Assert	top.props.i.ast_output_sequence	N (9)	Infinite
✓ Cover (related)	top.props.i.ast_output_count:precondition1	Hp	1
? Assert	top.props.i.ast_output_count	B	23 -
✓ Cover (related)	top.props.i.asm_size3:precondition1	Hp	1

In addition to these covers, it is recommended to add more cover properties, targeting interesting/important events in the design.

Modify the file props.sva to include the following cover properties:

```
module props (
    input clk, rstN,
    input eg_valid, eg_ready,
    input valid0, valid1, valid2, valid3,
    input ready0, ready1, ready2, ready3,
    input [1:0] size0, size1, size2, size3
);
    ...

    // Covers for interface
    cov_eg_valid_odd: cover property (eg_valid && eg_ready && eg_is_odd);
    cov_eg_valid_even: cover property (eg_valid && eg_ready && !eg_is_odd);
    // Covers for internal signals
    cov_eg_state_ADDR: cover property (top.eg.cur_state == `EG_ADDR);
    cov_eg_state_DATA: cover property (top.eg.cur_state == `EG_DATA);
    cov_fifo_8: cover property ($countones(top.brdg.fifo_entry_valid)==8);
    cov_fifo_16: cover property ($countones(top.brdg.fifo_entry_valid)==16);

endmodule
// Connect module to design
bind top props props_i (.*);
```

Note: You can reference internal design signals directly from the SVA file through a hierarchical reference (also called external module reference). In the code above, this was used to access internal signals without having to add them to the port list of the module.



Click the Source Recent Script button to reload the Tcl script. All the covers that have been added will be hit by formal, except for one:

Figure 28: Precondition cover

Type	Name	Engine	Bound
✓ Cover	top.props_i.cov_fifo_8	Ht	10
✗ Cover	top.props_i.cov_fifo_16	Hp (1)	Infinite
✓ Cover	top.props_i.cov_eg_valid_odd	Ht	5
✓ Cover	top.props_i.cov_eg_valid_even	Ht	6
✓ Cover	top.props_i.cov_eg_state_DATA	Ht	6

The cover that was not hit refers to an internal FIFO having all its entries valid. The formal engines concluded that it is not possible to use all 16 locations of the FIFO, hence making the cover unreachable. This is another design bug, related to how the FIFO pointers were coded, making it impossible for it to be truly full.

16. Validating constraints

One of the biggest challenges with formal verification is to develop constraints properly. In case constraints are incorrect, the environment might be over-constrained, which means that formal will not be testing all the stimulus that you'd like. As a result, bugs may be missed because the stimulus to hit them was not considered.

Adding cover properties, as seen in the previous section, helps detect over-constraints. In case a cover is unreachable, it could be because the stimulus to hit it has been constrained away by the current set of assumptions.

Another method to verify constraints is to use the `check_assumptions` command. It will perform two checks:

- `noConflict`: Check that constraints do not conflict with each other, by making sure that it is possible to generate waveforms of infinite length.
- `noDeadEnd`: Check that it is always possible to extend waveforms from any reachable state.

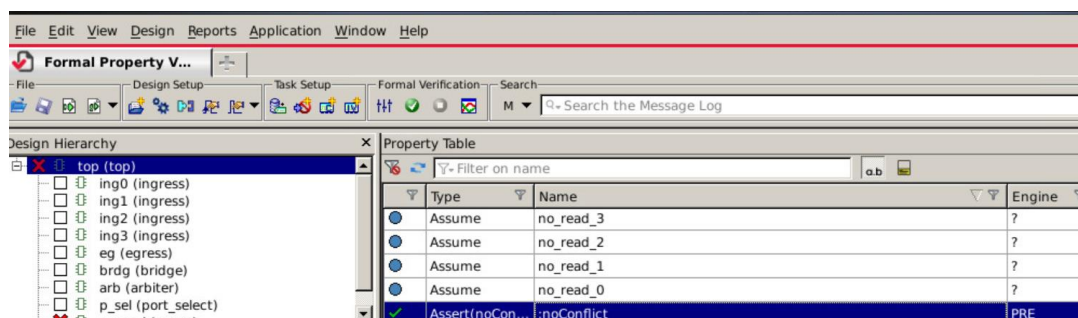
Modify the file `run.tcl` to include the `check_assumptions` command:

```
...
# Disable reads
assume -name no_read_0 {valid0 |-> wr_rd0}
assume -name no_read_1 {valid1 |-> wr_rd1}
assume -name no_read_2 {valid2 |-> wr_rd2}
assume -name no_read_3 {valid3 |-> wr_rd3}
# Check assumptions
check_assumptions
# Prove all properties
prove -all
```

Click the  Source Recent Script button to reload the Tcl script.

The assertions created by `check_assumptions` will be proven, showing that there are no conflicts in this environment.

Figure 29: Proof result for `check_assumptions` properties



Note: This RAK is verified on JasperGold version 2018.06.

References:

- JasperGold Apps Command Reference Manual
- JasperGold Apps User Guide