Training Lab

JasperGold® SVA and Proof

Version 5

1 Overview

This lab focuses on using SystemVerilog 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.

1.1 Lab Objectives

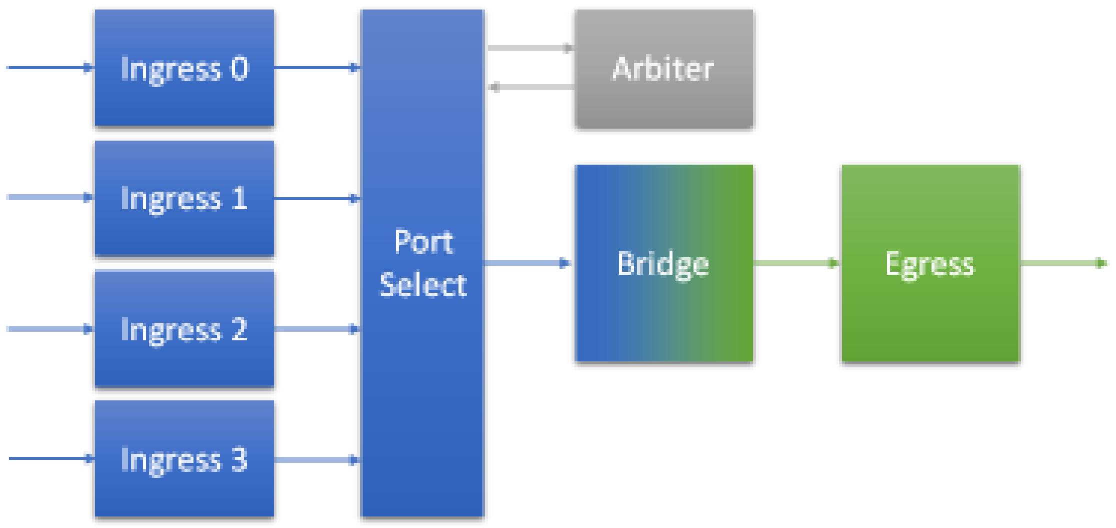
This lab will expose you to the following aspects of JasperGold® Formal Property Verification App:

- Counterexample debug
- Constraint development

1.2 Reference Design

The design used in this lab is a 4-to-1 datapath 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 1-1 Reference Design



1.3 Conventions

The following table lists conventions used in Jasper training materials.

Table 1-1 Jasper Text Conventions

Convention	**Definition**
💡	This icon identifies notes and tips showing different means to achieve the same result, useful features, and so on.
Click	Left-click unless otherwise specified with “right-click.”
Double-click	Left-click two times in rapid succession.
`Courier font`	This style indicates: • Text you will type in GUI fields • Commands and options • File names and paths • Code samples
Italics	User interface items such as button, menu option, and dialog field names.
Menu – Option	GUI command sequence. Example: *Help – Command Reference Manual* Meaning: Click on the *Help* menu and choose the option *Command Reference Manual*.
Blue text	Hyperlinked cross-reference. When you view the PDF version of Jasper labs from a computer screen, click on the blue text to view related information.

2 Environment Setup

2.1 Preparing the Lab

1. Begin this lab in your shell window in a writable location.

2. Unpack the lab package:

```
$ tar -zxf jasper_training.tgz
```

3. Change to this directory

```
$ cd jasper_training
```

2.2 Invoking the Tool

Launch JasperGold Apps with the following command:

```
$ jg lab2.tcl
```

The script `lab2.tcl` contains the commands required to load the design into JasperGold. It follows a typical sequence for most designs:

- Define variables used by the script
- Clear the environment with `clear -all`
- Compile the design with `analyze` and `elaborate` commands
- Specify clock/reset with the `clock` and `reset` commands

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, and user preferences.

Getting Help for the current App as well as the Tcl commands is as easy as clicking on the ? *Help* button.

3 Output Count Verification

This section will cover the verification of a requirement that requires writing glue logic in order to simplify the code for an assertion.

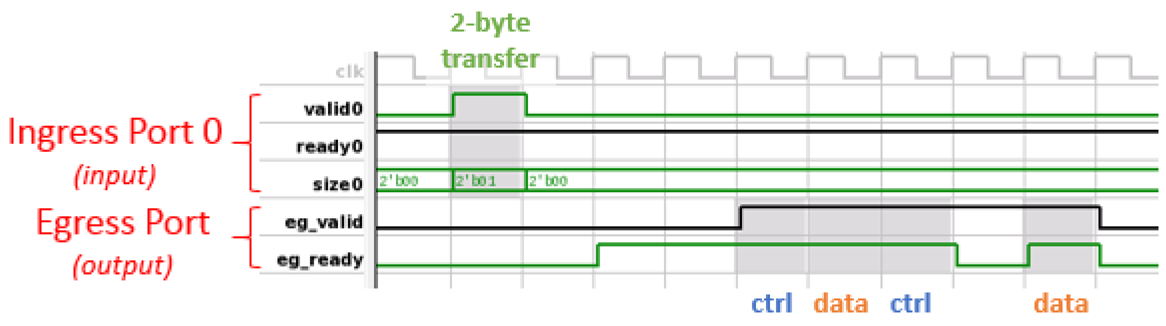
3.1 Description

Requests come into the design on the ingress ports, indicated by signals `valid0..3` and acknowledged by signals `ready0..3`. Each transfer contains a size indicator, signals `size0..3`. The value of these signals is decoded as follows:

- size = `2'b00`: 1-byte transfer
- size = `2'b01`: 2-byte transfer
- size = `2'b10`: 3-byte transfer
- size = `2'b11`: 4-byte transfer

Each of the transfers are serialized at the output, one byte at a time. A transfer for each byte is split into a control word and a data word. For example:

Figure 3-1: Two-byte transfer from ingress 0 port to egress port



The requirement that will be verified is that the design should never generate more requests at the egress port than there were requests in the ingress ports. In the example above, no more than 2 data cycles should be seen at egress, since a single 2-byte request was seen at ingress.

3.2 Specifying assertion

The glue logic for this assertion will consist of a counter `byte_cnt` which:

- Increments for every byte that enters on ingress port
    - Increment amount depends on `size` signal

- Decrements for every byte that exits on egress port
    - Decrement will only happen on data cycles

The design can only hold 64 entries, so the counter needs to have at least 7 bits (count from 0 to 127).

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

The glue logic and assertion will go into the file `props.sva`, which defines a module `props`. At this point the module is already connected to the design (using the `bind` keyword), and contains *some* of the glue logic needed.

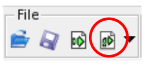
1. Modify the file `props.sva` to complete the glue logic for `byte_count`:

```
always_comb begin
  byte_cnt_nx = byte_cnt;
  // Increment if bytes are valid at ingress
  // << insert increment logic for byte_cnt_nx here >>
  // Decrement after every data word on egress
  // << insert decrement logic for byte_cnt_nx here >>
end
```

The logic should behave as follows:

- For each ingress port, increment according to `size0..3` if a transfer is happening at that port
    - size = `2'b00`: Increment by 1
    - size = `2'b01`: Increment by 2
    - size = `2'b10`: Increment by 3
    - size = `2'b11`: Increment by 4
    - A transfer happens when `valid0..3` and the corresponding `ready0..3` are asserted in the same cycle
- Decrement by 1 for each data word transferred at the egress port
    - A transfer happens when `eg_valid` and `eg_ready` are asserted in the same cycle
    - Use the glue logic `eg_is_data` to detect whether a control or data word is being transferred (only decrement for data words)

(Shortcut available by "**`cp solution/props.sva` ./props.sva**")

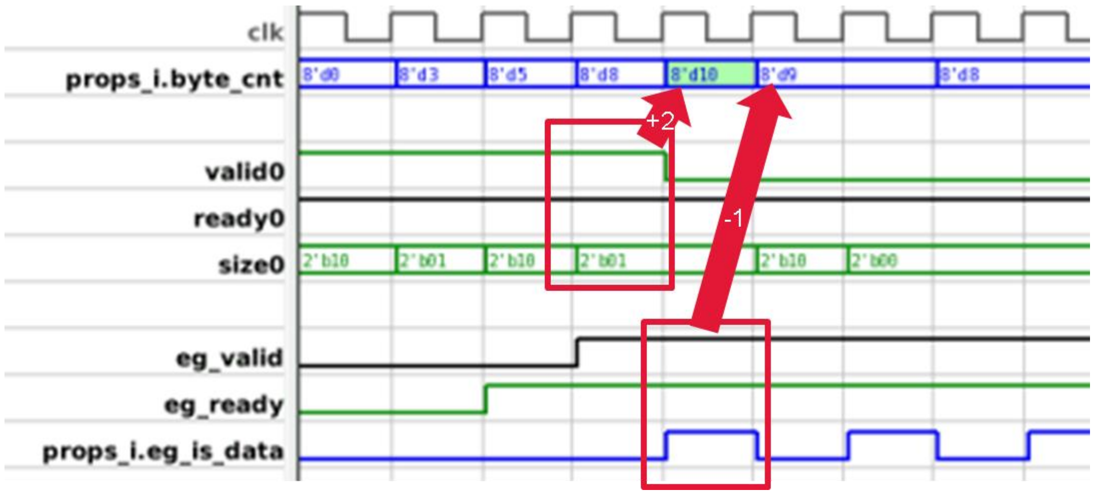
2. Click the  *Source Recent Script* button to reload the Tcl script

3. Use Visualize to inspect the `byte_cnt` behavior, by typing the command:

```
% visualize {props_i.byte_cnt == 10 ##[1:$] props_i.byte_cnt
== 0 [*3]}
```

This command will have Visualize create a waveform where `byte_cnt` reaches 10 and then goes back to 0 for 3 cycles. The waveform should confirm that the logic is behaving correctly and give some insight on how to write the assertion.

Figure 3-2: Exploration of `byte_cnt` logic with egress port activity



Note that when `byte_cnt` is zero then `eg_valid` is also zero. This behavior is expected since the design should not have any byte to output. Now we will write an assertion to confirm that this is always true.

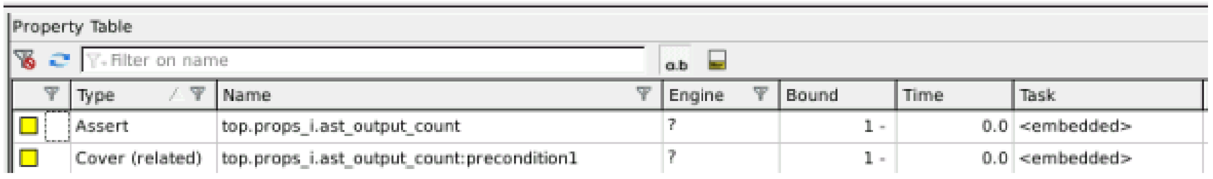
4. Modify the file `props.sva` to include an assertion to check the byte count:

```
ast_output_count: assert property (byte_cnt == 0 |-> !eg_valid);
```

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

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

Figure 3-3: Property Table after adding output count assertion

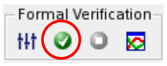

Property Table

Filter on name

	Type	Name	Engine	Bound	Time	Task
🟨	Assert	top.props_i.ast_output_count	?	1 -	0.0	<embedded>
🟨	Cover (related)	top.props_i.ast_output_count:precondition1	?	1 -	0.0	<embedded>

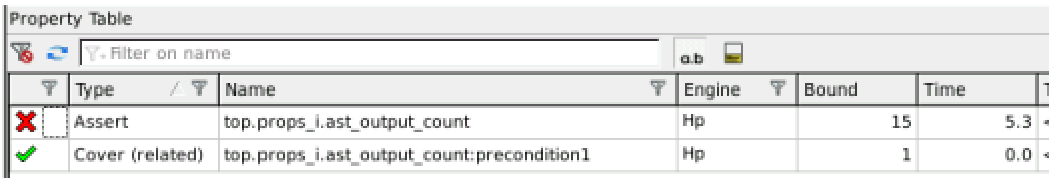
3.3 Running formal proof

Now you will prove this property using the formal engines.

6. Click the  *Prove All* button to run a formal proof

The formal proof should find a counterexample for the assertion `ast_output_count`, which we will debug in the next section.

Figure 3-4: Property Table after initial proof of `ast_output_count`



Property Table

	Type	Name	Engine	Bound	Time
✗	Assert	top.props_i.ast_output_count	Hp	15	5.3
✓	Cover (related)	top.props_i.ast_output_count:precondition1	Hp	1	0.0

3.4 Counterexample debug

The Visualize environment is used for debugging counterexamples. You can use all the techniques covered in the Visualize Lab to understand why an assertion is failing:

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

Out of all these steps, the usage of Why is typically present in every debugging session. The other features are used depending on the complexity of the counterexample.

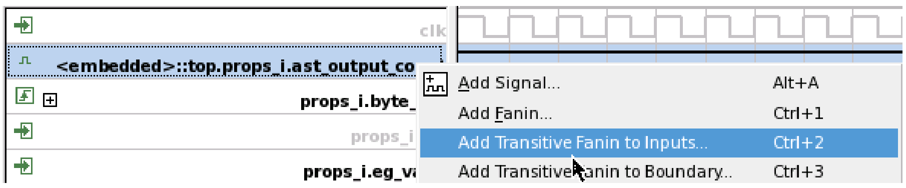
7. Double-click the assertion `ast_output_count` in the Property Table to open its counterexample in Visualize

Since this counterexample was found quickly, it is a good fit for QuietTrace. This will simplify the waveform even before you do any debug.

To highlight the impact of QuietTrace, you will clone this Visualize window and apply QuietTrace on only one of the waveforms.

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

Figure 3-5: Adding inputs connected to `ast_output_count`


clk
<embedded>::top.props_i.ast_output_co
props_i.byte_
props_i
props_i.eg_va
Add Signal... Alt+A
Add Fanin... Ctrl+1
Add Transitive Fanin to Inputs... Ctrl+2
Add Transitive Fanin to Boundary... Ctrl+3

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

9. 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.

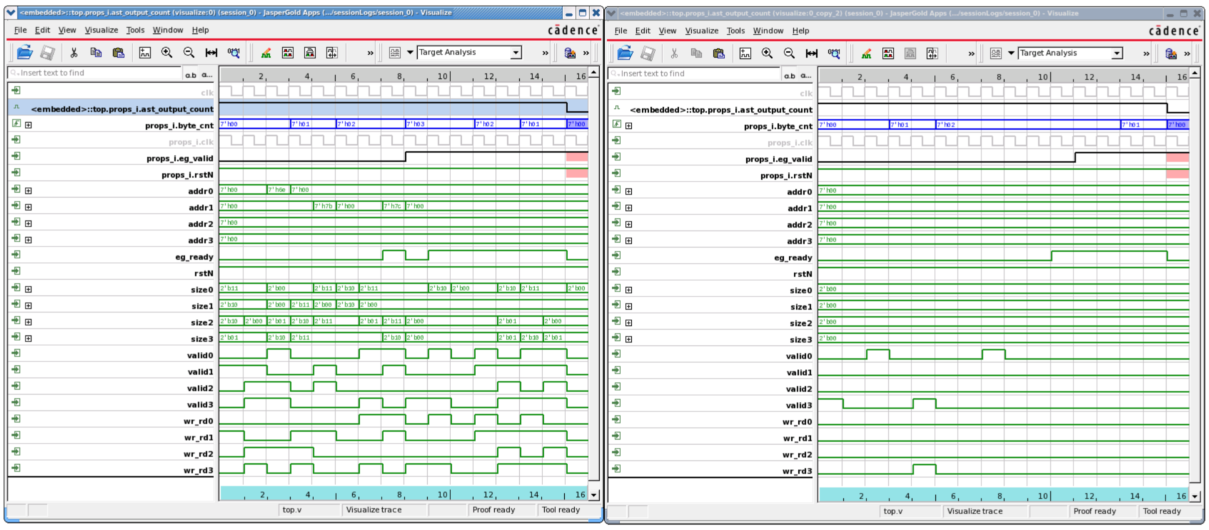
10. Click the *Clone* button to clone the waveform

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

12. 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 3-6: Comparison between non-quiet (left) and quiet (right) counterexample



13. Close the Visualize window where QuietTrace has not been applied.

14. On the quiet waveform (where QuietTrace has been applied), double-click the assertion `ast_output_count` at the last cycle (where it fails) to perform Why on it.

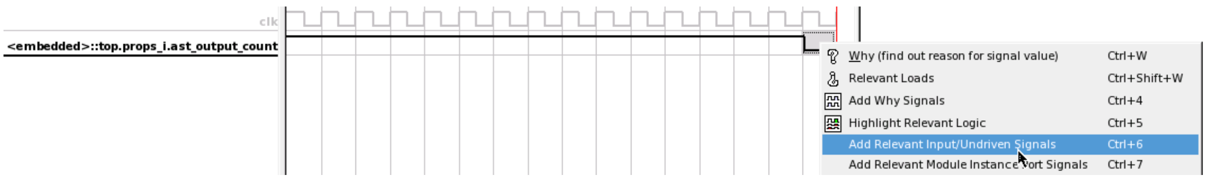
The Why result will show that the assertion is failing because `props_i.byte_cnt` is zero and `eg_valid` is not zero.

Real debugging of this counterexample would demand for additional Why operations to understand why `eg_valid` is high while there are no outstanding requests. This process will be skipped, and we will look at inputs driving the assertion instead.

15. 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)

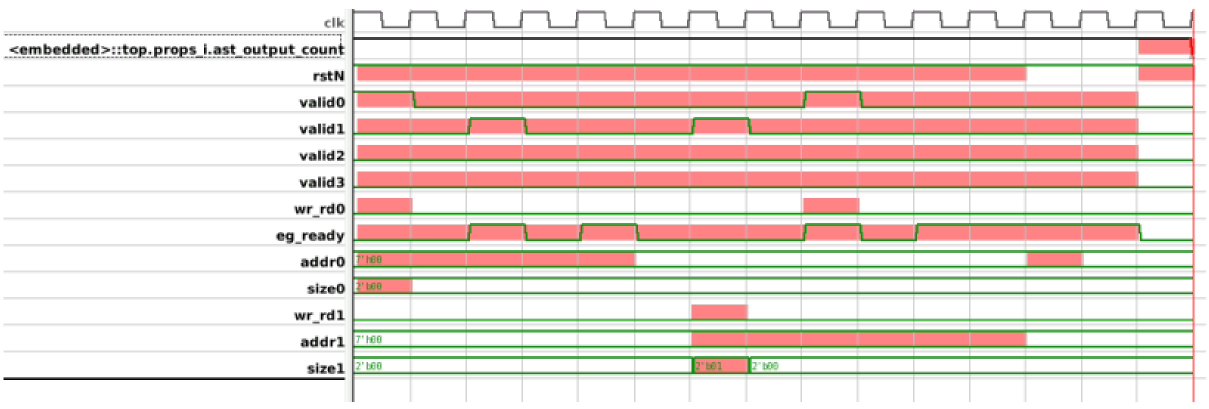
16. Right-click the assertion `ast_output_count` at the last cycle (where it fails) and select *Add Relevant Input/Undriven Signals.*

Figure 3-7: Adding relevant inputs affecting `ast_output_count`


clk
<embedded>::top.props_i.ast_output_count
Why (find out reason for signal value) Ctrl+W
Relevant Loads Ctrl+Shift+W
Add Why Signals Ctrl+4
Highlight Relevant Logic Ctrl+5
Add Relevant Input/Undriven Signals Ctrl+6
Add Relevant Module Instance Port Signals Ctrl+7

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 3-8: Relevant inputs of `ast_output_count`



Further debugging would show that this counterexample exposes a bug in the read logic of the design. QuietTrace and Highlight Relevant Inputs made it easy to see that the `wr_rd` signals were contributing to the failure, as seen above.

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.

3.5 Constraining environment

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`, or 2) in the Tcl file `lab2.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 Tcl 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.

17. Modify the file `lab2.tcl` to include assumptions to disable reads and limit the size:

```
...
# Set up Clocks and Resets
clock clk
reset ~rstN
# Disable reads
assume -name asm_no_read_0 {valid0 |-> wr_rd0}
assume -name asm_no_read_1 {valid1 |-> wr_rd1}
assume -name asm_no_read_2 {valid2 |-> wr_rd2}
assume -name asm_no_read_3 {valid3 |-> wr_rd3}

# Limit the size
assume -name asm_no_size0_4 {size0 != 2'b11}
assume -name asm_no_size1_4 {size1 != 2'b11}
assume -name asm_no_size2_4 {size2 != 2'b11}
assume -name asm_no_size3_4 {size3 != 2'b11}
```

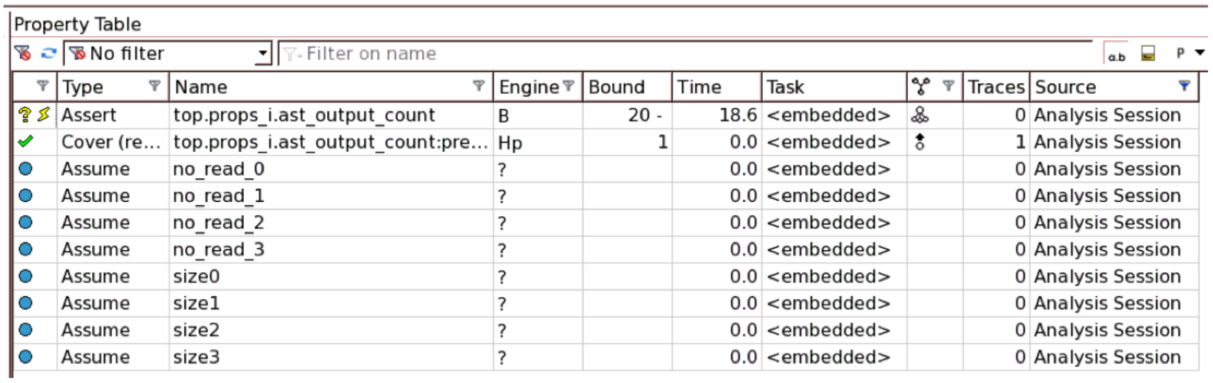
Shortcut available in:

`solution/lab2_solution.tcl`

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

These four assumptions should now appear in the Property Table:

Figure 3-9: Property Table after disabling reads



Property Table

No filter | Filter on name

	Type	Name	Engine	Bound	Time	Task		Traces	Source
?	Assert	top.props_i.ast_output_count	B	20 -	18.6	<embedded>		0	Analysis Session
✓	Cover (re...	top.props_i.ast_output_count:pre...	Hp	1	0.0	<embedded>		1	Analysis Session
●	Assume	no_read_0	?		0.0	<embedded>		0	Analysis Session
●	Assume	no_read_1	?		0.0	<embedded>		0	Analysis Session
●	Assume	no_read_2	?		0.0	<embedded>		0	Analysis Session
●	Assume	no_read_3	?		0.0	<embedded>		0	Analysis Session
●	Assume	size0	?		0.0	<embedded>		0	Analysis Session
●	Assume	size1	?		0.0	<embedded>		0	Analysis Session
●	Assume	size2	?		0.0	<embedded>		0	Analysis Session
●	Assume	size3	?		0.0	<embedded>		0	Analysis Session

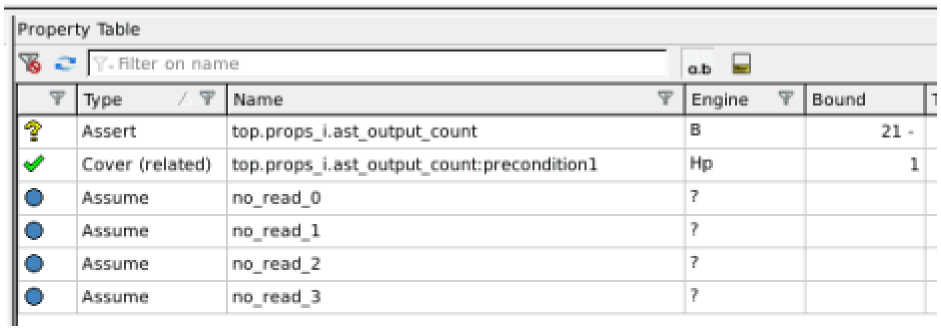
19. Click the *Prove All* button to run a formal proof

The 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.

20. Click the *Stop All Jobs* button to stop the formal proof

The property will be left in the undetermined state (icon), with a proof bound of about 16-25 cycles, depending on how long you let it run:

Figure 3-10: Property Table after stopping proof



Property Table

	Type	Name	Engine	Bound
?	Assert	top.props_i.ast_output_count	B	21 -
✓	Cover (related)	top.props_i.ast_output_count:precondition1	Hp	1
●	Assume	no_read_0	?	
●	Assume	no_read_1	?	
●	Assume	no_read_2	?	
●	Assume	no_read_3	?	

3.6 Summary

The development of this assertion uncovered a design bug. After adding a temporary constraint to work around it, now formal is not able to make the assertion fail anymore. 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 21 cycles or more, which means that the proof bound (number of cycles that the assertion is guaranteed to hold) is 20 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 can be done as a separate exercise.