Confidential

JasperGold RTL Design Bring-Up Rapid Adoption Kit

August 2019

cadence®

Contents

Objectives of this RAK

- Describe a methodology for initial bring-up of RTL designs using formal verification instead of a simulation testbench
- Show features of JasperGold that can help with RTL design bring-up
- Demonstrate the types of static analysis that can be performed, including clock and reset analysis
- Become familiar with a methodology that can help give you confidence that your design is functioning as expected
- Learn features of JasperGold Visualize™ that help you explore design behavior

Overview of JasperGold RTL Design Bring-Up

- Problem
  - New designs require some initial testing to make sure that they are ready to hand off to the verification team or integrate with other blocks.
  - Developing a unit testbench can involve days or even weeks of work to develop.
  - This unit testbench is often not reusable and does not provide any additional value to the verification team.
  - The unit testbench needs to be maintained and (hopefully) have matching documentation.
  - Some teams use automated simulation testbenches, but it usually requires high effort to use it because of a lack of modeling of interfaces.

- Solution
  - JasperGold can easily be used to quickly and thoroughly explore design behaviors without any simulation testbench.
  - Often, it takes less than a few minutes to compile a design and start exploring behaviors.
  - Iterative experiments are rapid, without the bulk or complexity of a testbench.
  - There is no “throwaway” work. Properties created at the block level can be handed off to the other designers or the verification team for use in both simulation and formal.
  - In a team setting, properties developed at the inputs and outputs of of your block can be shared for block-level design bring-up or verification of the connecting block.
  - Once users become more familiar with JasperGold, they can use the same environment with other JasperGold apps to check things such as deadcode, FSM deadlocks, X-propagation, and more.

Prerequisites

- Experience
  - This RAK assumes that you have basic experience with Verilog® and SystemVerilog SVA syntax.
  - It is recommended that you become familiar with the following material before starting this RAK:
    - Article (20416736) JasperGold Apps RTL Development Design Exploration Use Model https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1Od000000050TTEAY
    - Article (20417069) Design Exploration Guidelines https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1Od000000050YqEAI

- Licenses
  - This RAK uses the JasperGold Formal Property Verification App (JG-FPV)
  - One of the following license schemes will be required to launch JG-FPV:
    - jasper_fpv
    - jasper_fao
    - jasper_papp
    - jasper_fpv_opt && (incisive_formal_verifier || incisive_enterprise_verifier)
  - In addition, you will need one of the following interactive licenses:
    - (jasper_interactive || jasper_pint || jasper_interactive_opt)
  - Consult with your CAD team to see if you have one of the above licenses

Setup and Confirm Tool Installation

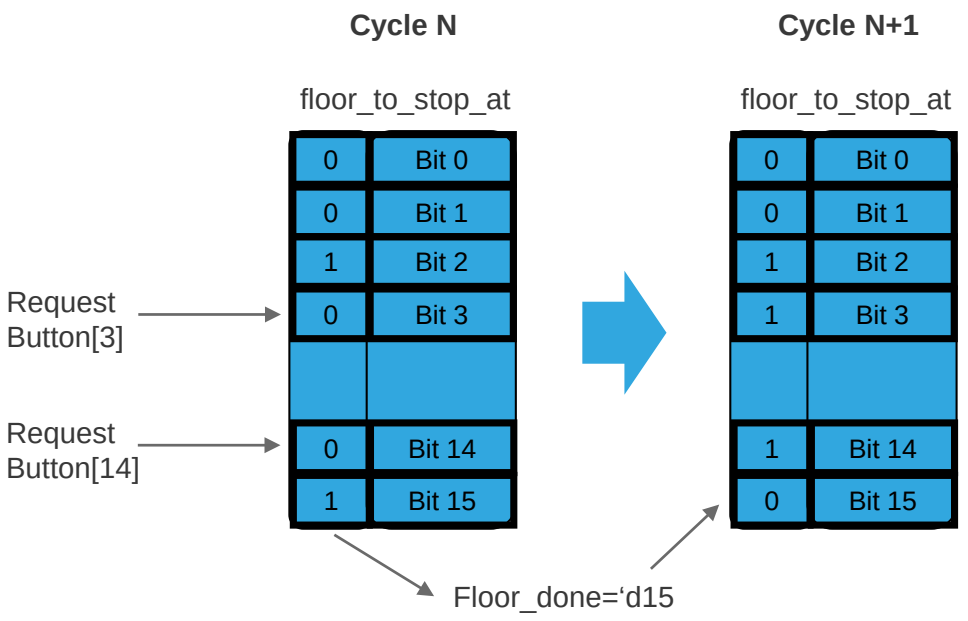
- Confirming your license and JasperGold version
  - Type the following in your shell:
  - `% jg –version`
  - `% jg -fpv -batch`

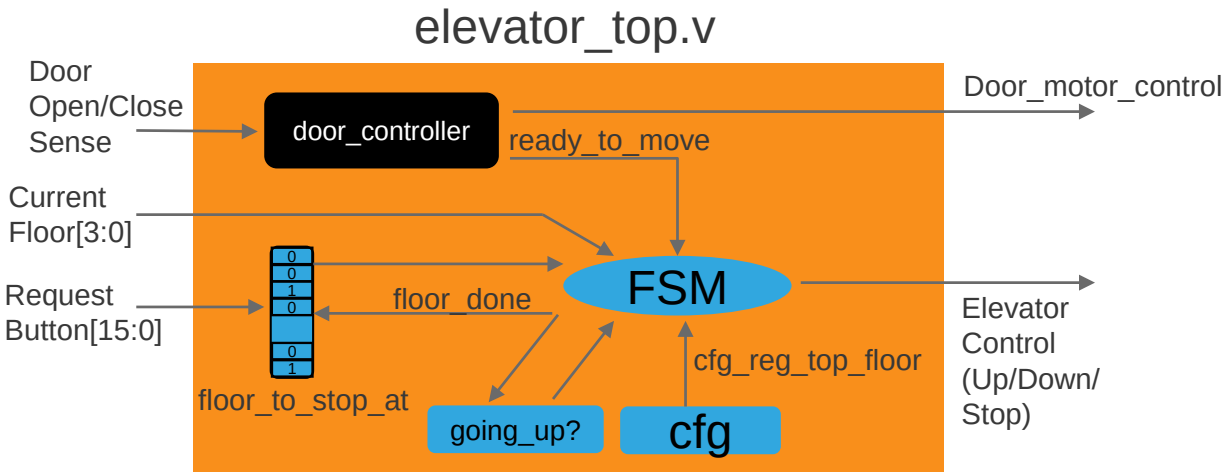
- The commands above should display the JasperGold version. You should be running with 2019.06FCS or newer.

- Unpack the RAK distribution package in your work area. This document and all files needed to run the example are included in that package.

Design Overview

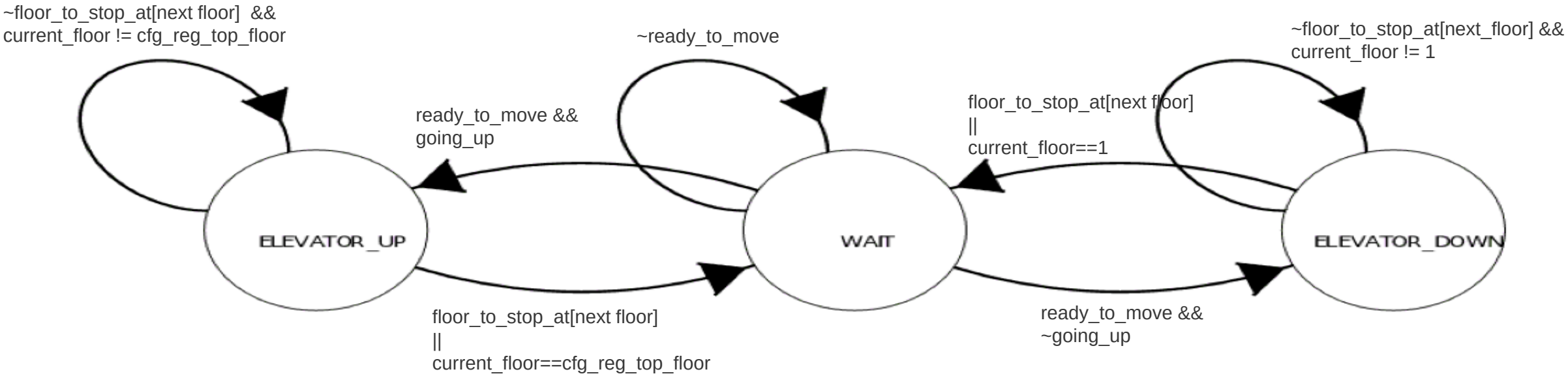
- The elevator controller is responsible for determining when the elevator should move up, move down, or stop.
- There is a single request input (`request_button`) for each floor. Once the `request_button` for any floor is high, then it is set in the `floor_to_stop_at` array.
- The FSM decides if the elevator goes UP or DOWN or WAITs.
- When the `floor_to_stop_at` is reached, the `floor_done` indicates which bit to clear. The elevator then stays in WAIT until `ready_to_move` goes high again.
- The `going_up` flag toggles from high to low if the `floor_done` vector equals the `cfg_reg_top_floor`. It toggles from low to high when it equals 1.
- The `door_controller` instance (black boxed) provides the `ready_to_move` signal that tells the FSM when the doors have closed and the elevator is able to move again.





Design Overview (continued)

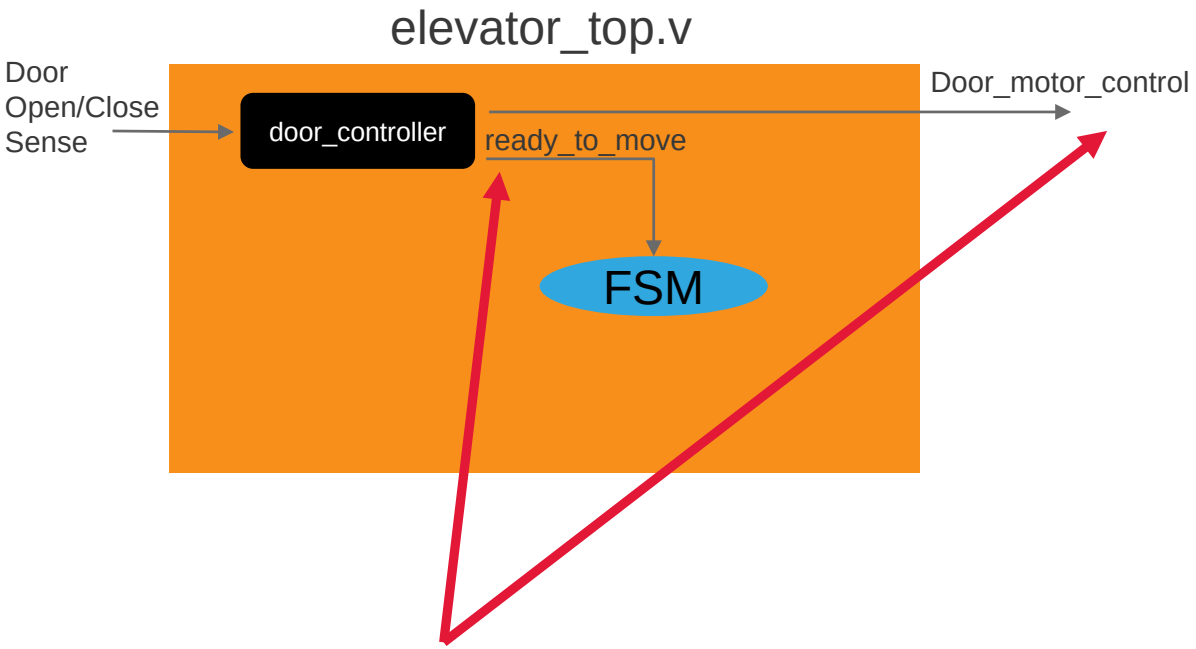
- The Finite State Machine (FSM) for the `elevator_top` module is shown below, with the conditions simplified to help you get familiar with the design.



Black Boxing

How black boxing can help you

- As mentioned in the previous slide, the `door_controller` is black boxed.
- Our objective is to verify the `elevator_top` module, even though we do not have the RTL available for the `door_controller` module.
- In JasperGold, we can compile the `elevator_top` and indicate that we want to black box the `door_controller`.
- This means that JasperGold can wiggle each black box output any way that it wants in ANY cycle!
- We will show how we can constrain this behavior later in the RAK.
- You might want to consider black boxing large or complex elements in your design. JasperGold will automatically black box large memory arrays and multipliers by default.
- For more details on black boxing commands, look at the Tcl Help for the `elaborate` command or review the following video:
  - Article (20467784) An Introduction To Elaborating Designs in JasperGold (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V000007MnuKUAS



Because `door_controller` is black boxed, `ready_to_move` and `door_motor_control` can take ANY value in ANY cycle!

Compiling the RTL

- We need to compile our design in JasperGold. Code should be fully synthesizable Verilog, SystemVerilog, or VHDL.
- There are some non-synthesizable constructs, such as “initial” blocks that will compile, but will be ignored by JasperGold.
- This design is all Verilog and located in a single file called `elevator_top.v`.
- Notice that we have 1 clock called `clk` and 1 active-high reset called `rst`.

```
module elevator_top (
  input clk,
  input rst,

  // Configuration logic
  input        cfg_rnw,
  input [31:0] cfg_addr,
  input [15:0] cfg_wr_data,
  input [15:0] cfg_rd_data,

  // Elevator sense and control
  input [15:0] request_button,    // 1 button for each floor to indicate a floor to stop at
  input  [3:0] current_floor,     // 4-bit value indicating which floor we are at
  output [1:0] elevator_control,  // 2'b00=ELEV_MOTOR_STOP, 2'b01=ELEV_MOTOR_UP, 2'b10=ELEV_MOTOR_DOWN

  // Door sense and control
  input        door_open_sense,
  input        door_close_sense,
  output       door_motor_control // 2'b00=DOOR_STOP, 2'b01=DOOR_OPEN, 2'b10=DOOR_CLOSE
);
```

Compiling the RTL

Setup Tcl to load your design into JasperGold

- With your preferred text editor, create a new file called `run_elevator_top.tcl`.
- This file will be used to compile the design and tell JasperGold what our clock and reset look like.
- The `elaborate` command contains the switch `-bbox_m door_controller` to tell JasperGold that we are black boxing this module.
- Look in the Tcl Help for additional capabilities of the 'analyze' and 'elaborate' commands.
- Copy the text at the right (or from `<RAK_install>/solution/compile_rtl.tcl`) to create your Tcl file
- For more information about `analyze` and `elaborate`, see the following videos:
  - Article (20467785) Introduction to the Analyze Command for JasperGold (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V000007MnuoUAC
  - Article (20467784) An Introduction To Elaborating Designs in JasperGold (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V000007MnuKUAS

run_elevator_top.tcl

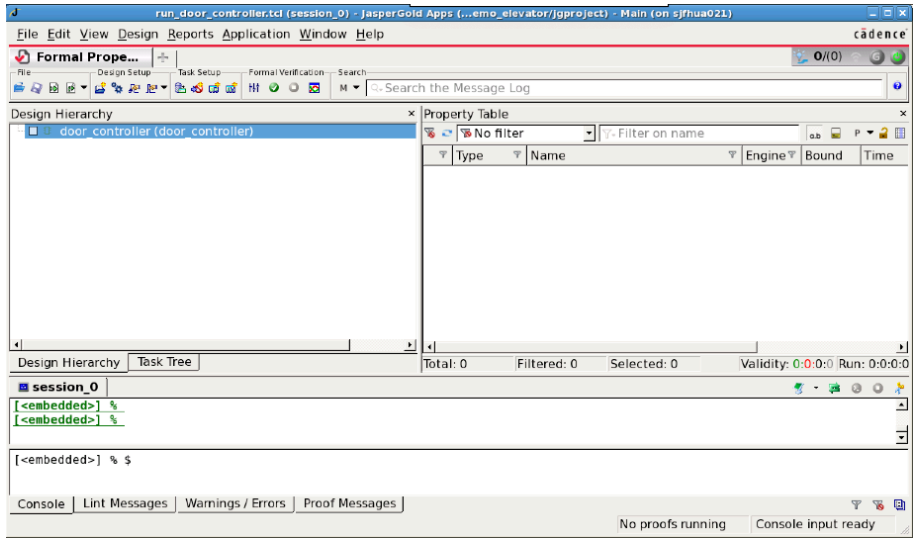
```
# Clear environment
clear -all

# Compile HDL files
analyze -sv elevator_top.v
elaborate -top elevator_top -bbox_m door_controller
```

Launch JasperGold

- We can first launch JasperGold from our terminal:

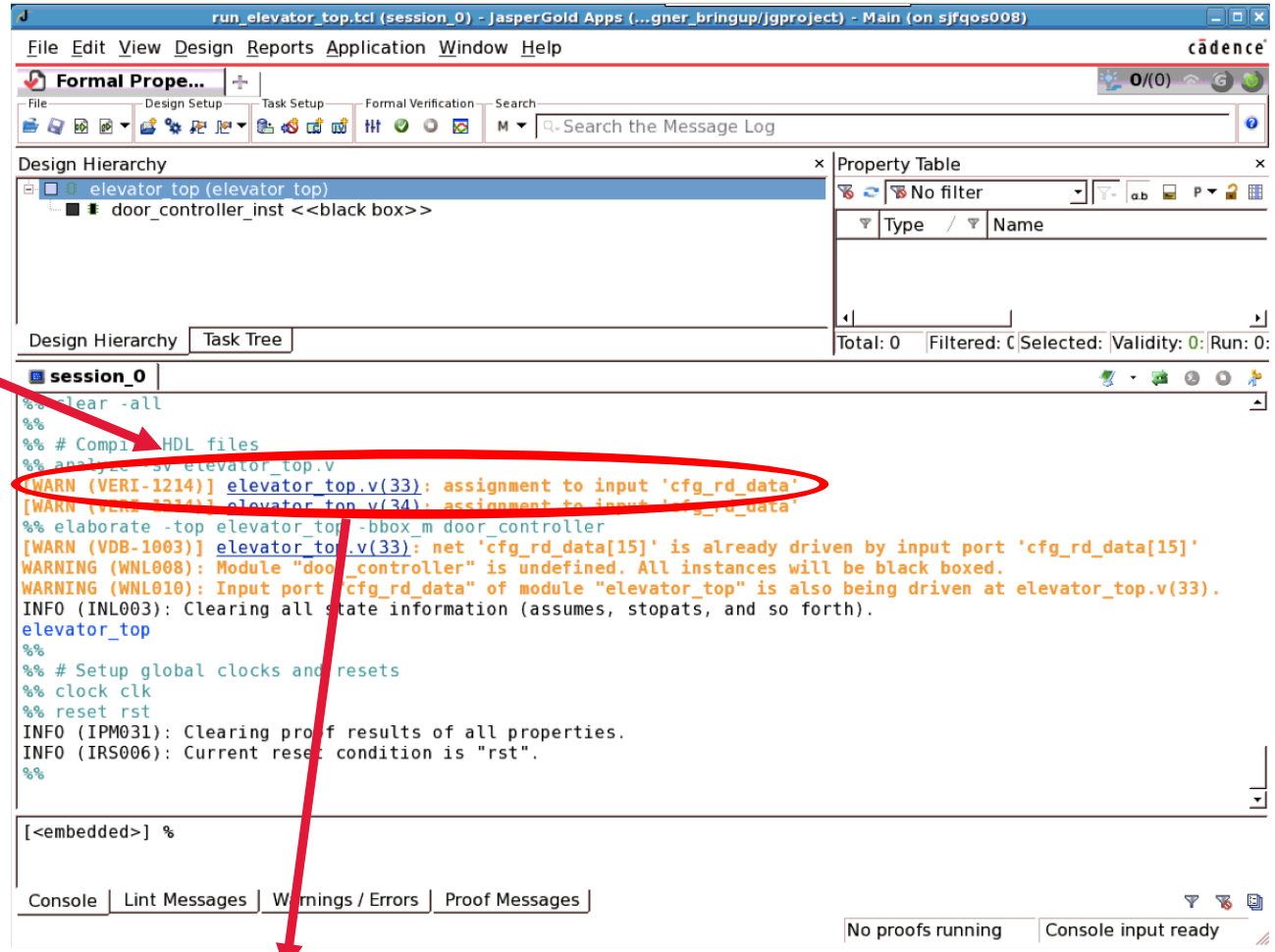
```
% jg run_elevator_top.tcl
JasperGold Apps 2019.06 FCS 64 bits 2019.04.01 18:13:48 PDT
```

- You will soon see the JasperGold GUI with the design loaded.



Static Analysis – Analyze/Elaborate

- After launching JasperGold, you will see compilation warnings in the message log.
- Let's debug the first VERI-1214 warning.
- This warning says we are assigning a value to an input called `cfg_rd_data`.
- We can click on the link to RTL line 33 (blue) to bring up the Source Browser, which highlights the line of RTL where we have the problem.
- This is definitely a problem since `cfg_rd_data` is incorrectly declared as an input!
- TIP: You can right-click in the *Source Code Pane* and choose *Edit* to bring up an editor. To configure which editor you prefer, you can go to the JasperGold menu bar and choose *Edit -> Preferences -> External Tools*. Here you can enter the command used to launch your favorite editor.

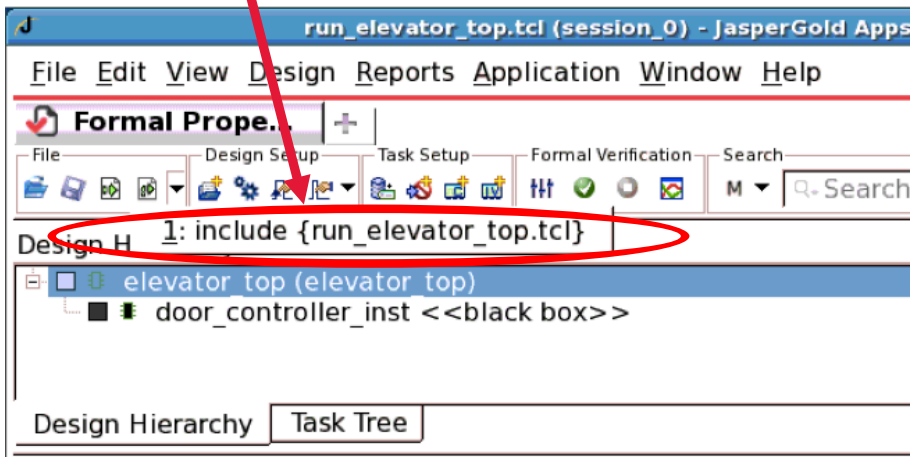



```
33 assign cfg_rd_data[15:4] = 'd0;
34 assign cfg_rd_data[3:0]  = cfg_reg_top_floor;
```

Static Analysis – Analyze/Elaborate (continued)

- Edit the `elevator_top.v` to change `cfg_rd_data` from an input to an output.

- Then, reload the Tcl file by going to the toolbar, pressing the small down arrow, and choosing the `run_elevator_top.tcl` script.

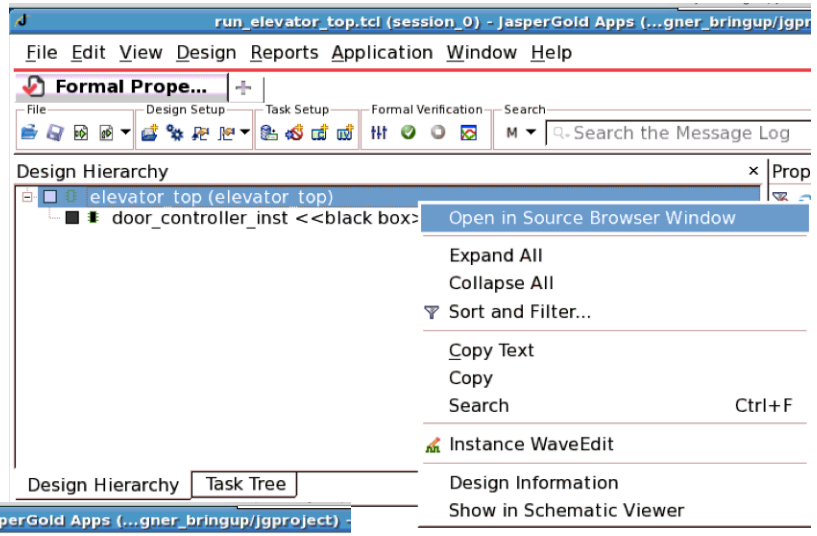


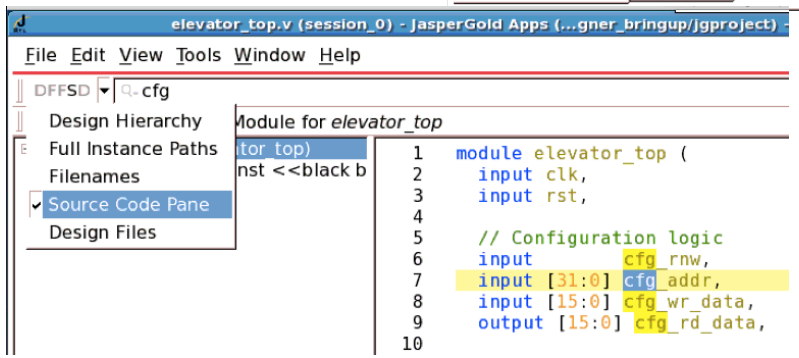

```
module elevator_top (
  input clk,
  input rst,

  // Configuration logic
  input        cfg_rnw,
  input [31:0] cfg_addr,
  input [15:0] cfg_wr_data,
  output [15:0] cfg_rd_data,

  // Elevator sense and control
  input [15:0] request_button,
  input  [3:0] current_floor,
  output [1:0] elevator_control,

  // Door sense and control
  input        door_open_sense,
  input        door_close_sense,
  output       door_motor_control
);
```

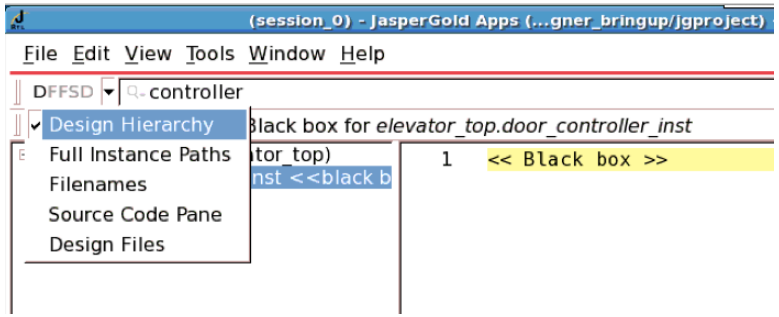
elevator_top.v

Static Analysis – Hierarchy

- After re-launching JasperGold, right-click on `elevator_top` in the hierarchy and choose *Open in Source Browser Window*.
- In the search bar, type `cfg` and choose *Source Code Pane* from the drop-down menu.
- Notice that you can also search the Design Hierarchy, Instance Paths, and so forth. In large designs, this can be very useful.





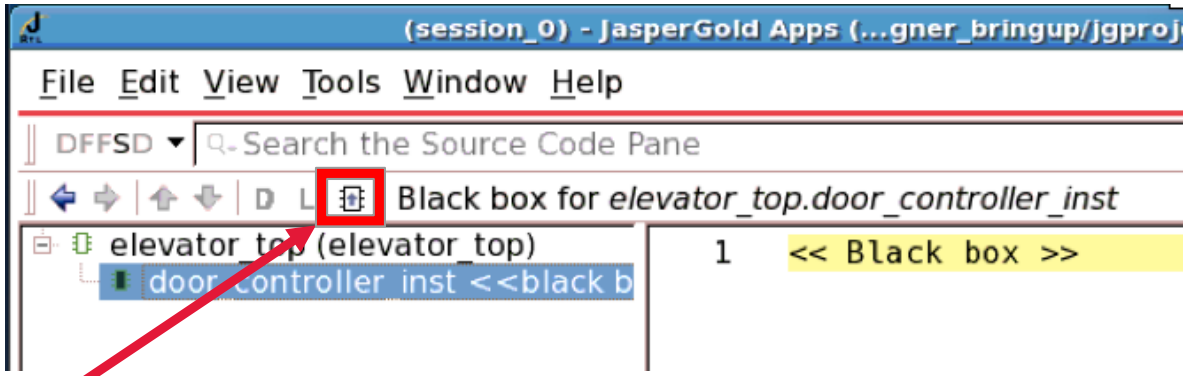


Static Analysis – Hierarchy (continued)

- In the Source Browser, we can see the `door_controller` is a black box. Click on it to select it.

- To see the instantiation of the `door_controller_inst`, click on the *Show Instantiation* button. This shows you the instantiation in the Source Browser.

- Close the Source Browser window.




```
108 door_controller door_controller_inst (
109     .clk(clk),
110     .rst(rst),
111     .ready_to_open(state==WAIT),
112     .request(|request_button),
113     .door_open_sense(door_open_sense),
114     .door_close_sense(door_close_sense),
115     .door_motor_control(door_motor_control),
116     .ready_to_move(ready_to_move)
117 );
```

Static Analysis – Clock and Reset

- Before we can run JasperGold on our design, we need to define the clock and reset.

- For simple designs, you can use the following Tcl commands in the JasperGold Console:

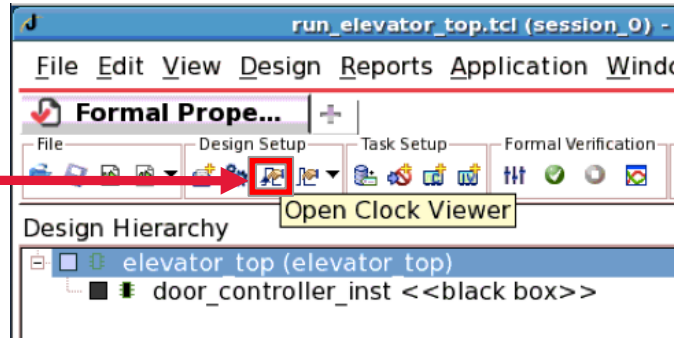
```
clock <your_clock>
reset <your_reset>
```

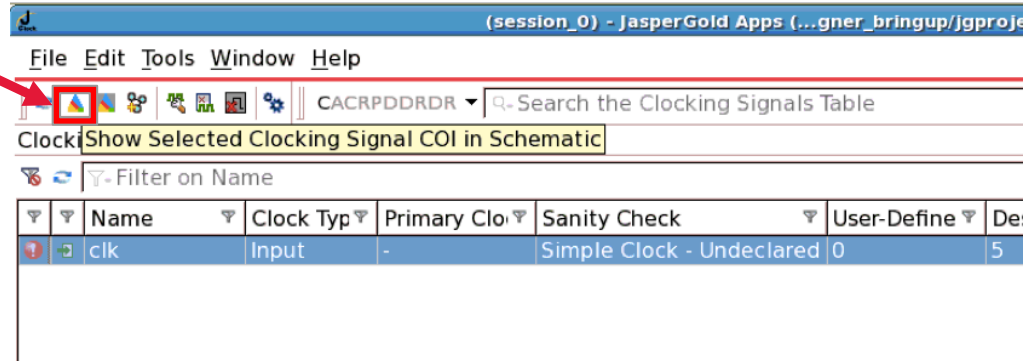
- For more information, see these video links for clock and reset:
  - Article (20468192) Introduction To JasperGold Clock Command (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V00000679pbUAA
  - Article (20468194) Specifying Reset for JasperGold (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V00000679plUAA

- In the next section, we will show you how to use the clock viewer and reset analysis to help you declare clock and reset.

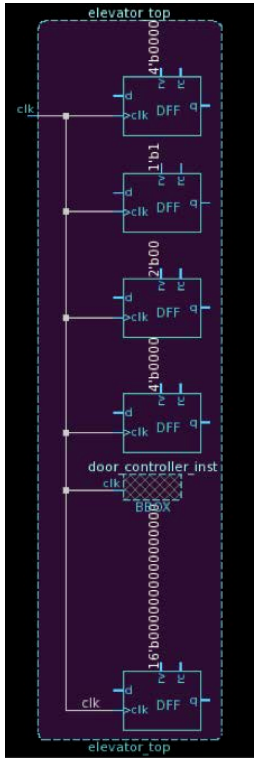
Static Analysis – Clock Viewer

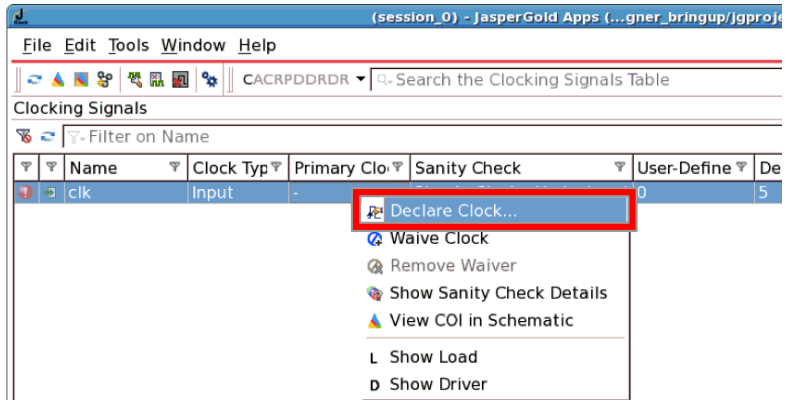
- Let's use the Clock Viewer to help us identify and declare a clock. In the toolbar, click on the Clock Viewer button.
- The table shows we have only 1 clock. Click on the button shown at the right to bring up a schematic of the clock cone-of-influence.
- The schematic can be useful to help you understand which clocks you need to declare.
- In this design, the clock is simple. In more complex designs, it can help to identify undriven clocks, registers driven on different clock edges, gated clocks, clock dividers, and more.
- Close the schematic.
- Right-click on the `clk` signal in the table and choose *Declare Clock*.
- We do not have any special clock requirements, so just press *OK* in the dialog that comes up.
- You can now close the Clock Viewer window.

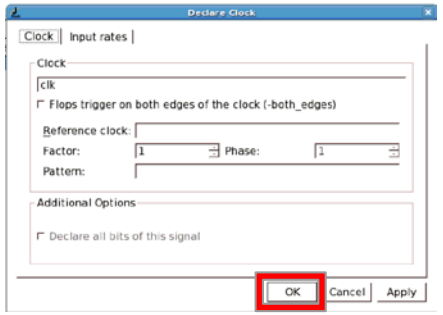




Clock Schematic

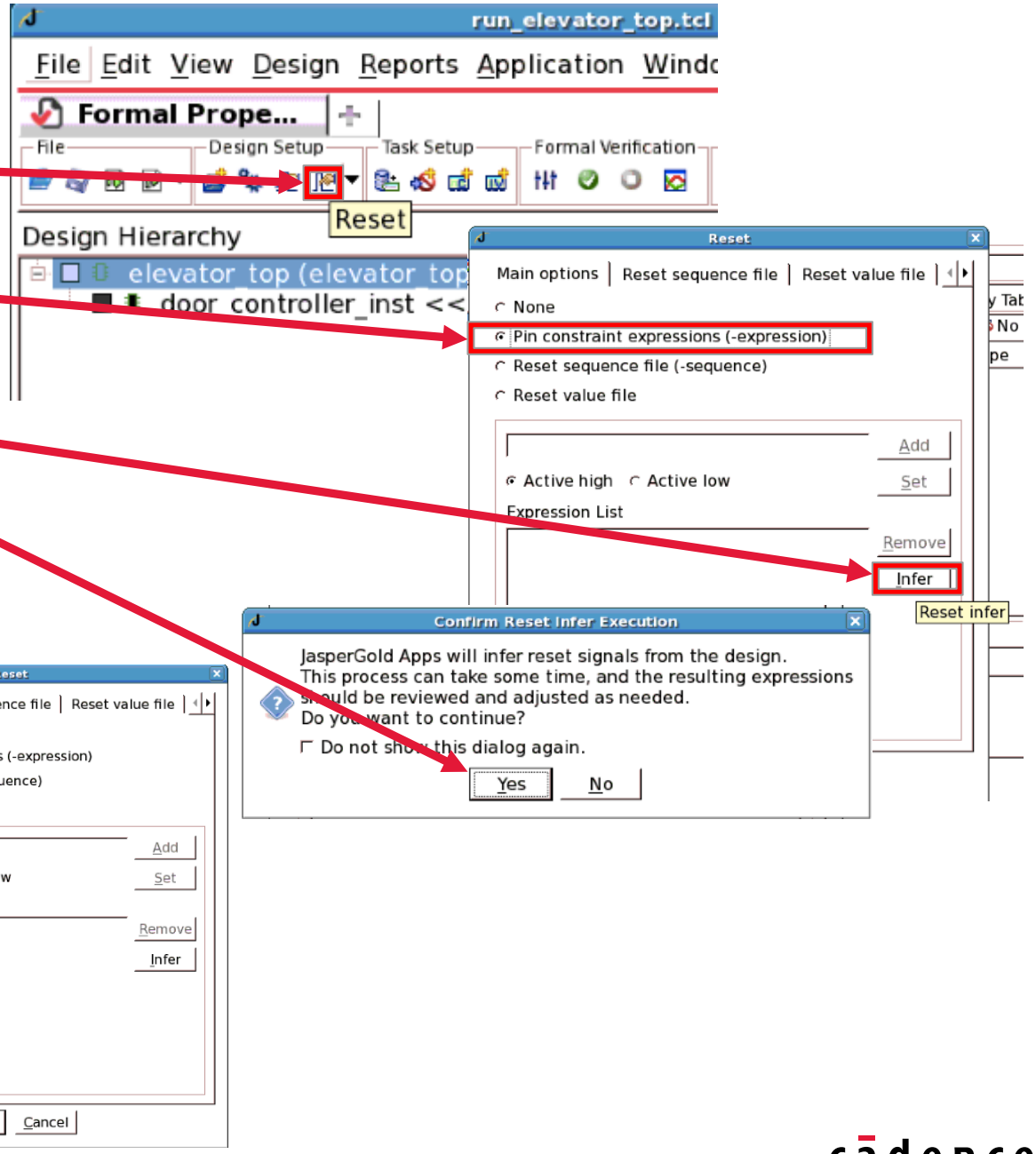






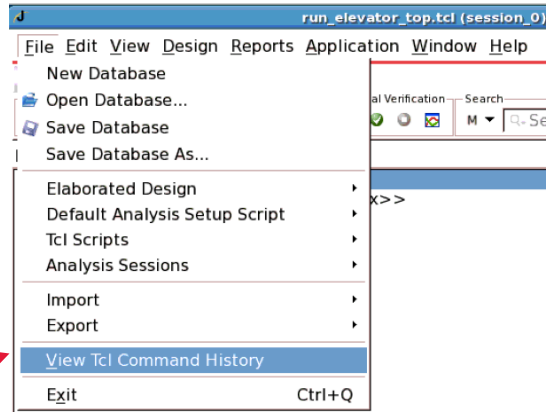
Static Analysis – Reset Analysis

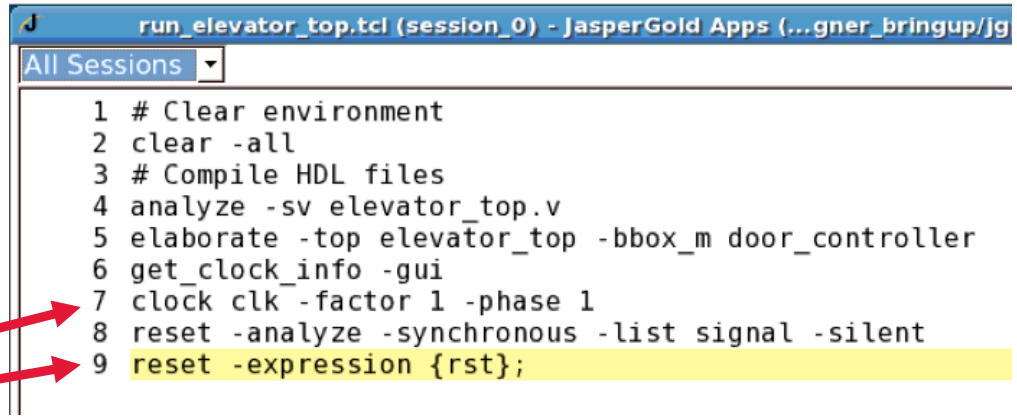
- Click on the *Reset* button in the toolbar.
- Choose *Pin constraint*, which will allow JasperGold to drive the reset pin and reset all of our flops.
- Next, choose *Infer* to find out which pin controls our reset. Press *Yes* to proceed.
- You will see that one reset pin called `rst` was found.
- Press *OK* to finish.
- In more advanced designs, reset analysis can identify sync/async reset conditions, multiple top-level reset pins feeding flops, the number of flops and properties using the reset, and more.
- For more information, see this link about reset analysis:
  - Article (20417121) Reset Analysis and Verification with JasperGold Apps https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1Od000000050ZgEAI

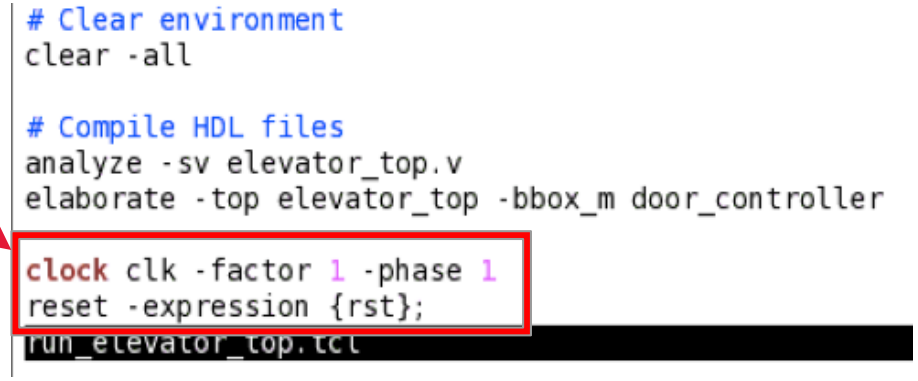


Editing Your Tcl File

- Now that we have identified the clock and reset, we need to add the clock and reset commands to our Tcl file.
- We can re-source this Tcl file at any time to instantly set up our formal environment.
- On the *File* menu, choose *View Tcl Command History*.
- This gives us a list of all of the commands we have run so far and gives us an easy way to copy the commands.
- Copy the clock and reset commands and paste them at the bottom of your `run_elevator_top.tcl` file in your text editor.
- Close the Tcl History window and save the file in your editor.







Design Query

- JasperGold offers some powerful ways to help you query the design.
- In the Console, we can use `get_design_info` to get a summary of the design. The numbers in parentheses are the “bit-blasted” counts.
- Some other switches that you can pass are shown on the right.
- The command returns a Tcl list, which can be used to pass on to other Tcl commands.
- For more information and examples, you can type `help get_design_info -gui` in the Console.

```
[<embedded>] % get_design_info
Statistics [for instance "elevator_top"]
----------------------------
# Flops:          5 (27) (0 property flop bits)
# Latches:        0 (0)
# Gates:          107 (759)
# Nets:           126
# Ports:          12
# RTL Lines:      178
# RTL Instances:  1
# Embedded Assumptions: 0
# Embedded Assertions:  0
# Embedded Covers:      0
27
```

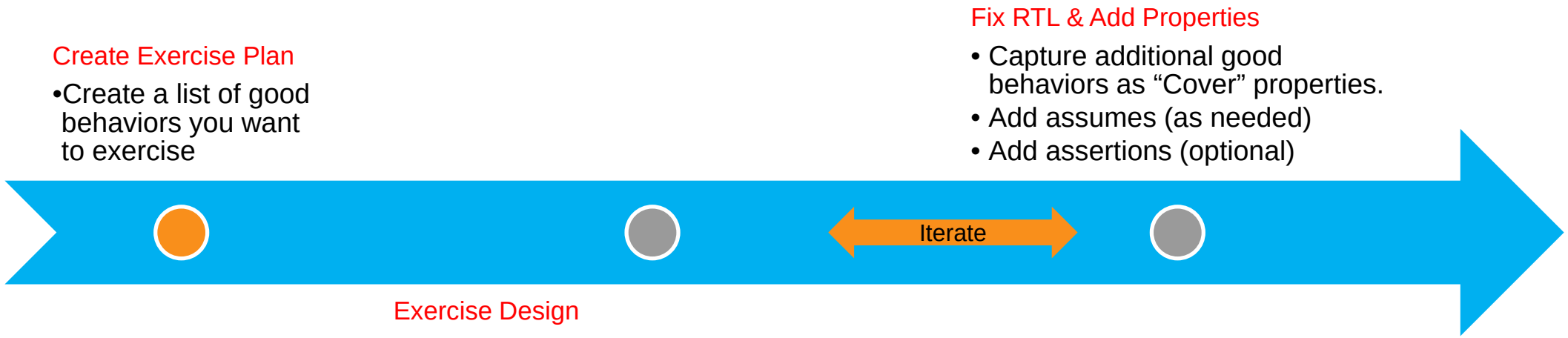
```
[<embedded>] % get_design_info -instance door_controller_inst
Statistics [for black boxed instance "door_controller_inst"]
----------------------------
# Ports:          8
0
```

```
[<embedded>] % get_design_info -list fsm
List of FSMs: 1 (2)
------------------
state (2)

state
```

```
[<embedded>] % get_design_info -list signal -filter cfg -regexp
List of Signal: 5 (69)
------------------
cfg_addr (32)
cfg_rd_data (16)
cfg_reg_top_floor (4)
cfg_rnw (1)
cfg_wr_data (16)

cfg_addr cfg_rd_data cfg_reg_top_floor cfg_rnw cfg_wr_data
```

RTL Design Bring-Up Flow

- To be successful with RTL design bring-up in JasperGold, you need to first create a Plan consisting of the good behaviors you want to exercise. We want to see that these good behaviors can be exercised, and we want to study these waveforms.
- As we exercise the design, we will capture additional behaviors as properties that can be handed off to other simulation and formal teams.
- We iterate between exercising and capturing properties until we are able to observe all good behaviors in our Plan.

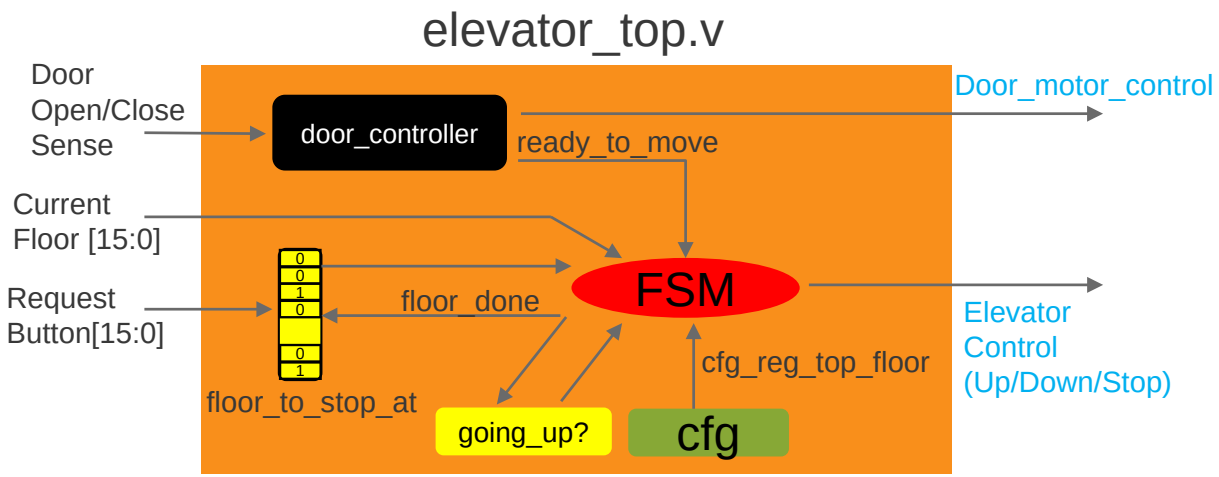


Create Exercise Plan

- Let's start with planning out the good behaviors we want to exercise.
- We want to check some parts of the design for simple reachability. For example, we can check that all FSM states can be reached.
- We also have functional sequences we can exercise. In this case, we want to observe that the elevator can go all the way up and down and that the top floor can be reprogrammed.
- Rather than telling JasperGold what we want to do with the inputs, we tell it what we want to observe. JasperGold will generate the input stimulus required to show the behavior.
- We have color-coded our plan below against the block diagram to show which tests are exercising each part of the design.

❑ *Reachability*
- ❑ All FSM states and transitions are reachable
- ❑ All valid output states are reachable
- ❑ Configuration bits can be programmed

❑ *Functional Sequences*
- ❑ Elevator goes all the way up and down twice (UP → Down → Up → Down)
- ❑ Elevator goes up, then stops, then continues going up, then goes down (UP → Wait → Up → Down)
- ❑ While going up, we reprogram the top floor (UP → CFG → UP → CFG → Down)



Create Exercise Plan (continued)

- Edit `elevator_top.v` and copy the SVA cover properties as shown. Paste them at the bottom of the RTL, just above `endmodule`. You can also copy these directly from `<RAK_install>/solution/sva_cover_properties.txt`.
- We can see below how each of these cover properties maps to our Exercise Plan.

Add covers for good behaviors

Exercise Plan

- ❑ *Reachability*
  - ❑ All FSM states and transitions are reachable
  - ❑ All valid output states are reachable
  - ❑ Configuration bits can be programmed
- ❑ *Functional Sequences*
  - ❑ Elevator goes all the way up and down twice (UP → Down → Up → Down)
  - ❑ Elevator goes up, then stops, then continues going up, then goes down (UP → Wait → Up → Down)
  - ❑ While going up, we reprogram the top floor (UP → CFG → UP → CFG → Down)

```
// ***********************************************************************
// Coverage (Reachability)
// ***********************************************************************
cov_elevator_top_fsm_WAIT:          cover property (@(posedge clk) state==WAIT);
cov_elevator_top_fsm_ELEVATOR_UP:   cover property (@(posedge clk) state==ELEVATOR_UP);
cov_elevator_top_fsm_ELEVATOR_DOWN: cover property (@(posedge clk) state==ELEVATOR_DOWN);
cov_elevator_top_fsm_WAIT_to_WAIT:  cover property (@(posedge clk) state==WAIT ##1 state==WAIT);
cov_elevator_top_fsm_WAIT_to_UP:    cover property (@(posedge clk) state==WAIT ##1 state==ELEVATOR_UP);
cov_elevator_top_fsm_WAIT_to_DOWN:  cover property (@(posedge clk) state==WAIT ##1
state==ELEVATOR_DOWN);
cov_elevator_top_fsm_UP_to_WAIT:    cover property (@(posedge clk) state==ELEVATOR_UP ##1 state==WAIT);
cov_elevator_top_fsm_UP_to_UP:      cover property (@(posedge clk) state==ELEVATOR_UP ##1
state==ELEVATOR_UP);
cov_elevator_top_fsm_DOWN_to_WAIT:  cover property (@(posedge clk) state==ELEVATOR_DOWN ##1
state==WAIT);
cov_elevator_top_fsm_DOWN_to_DOWN:  cover property (@(posedge clk) state==ELEVATOR_DOWN ##1
state==ELEVATOR_DOWN);


cov_elevator_top_out_stop:          cover property (@(posedge clk) elevator_control==ELEV_MOTOR_STOP);
cov_elevator_top_out_up:            cover property (@(posedge clk) elevator_control==ELEV_MOTOR_UP);
cov_elevator_top_out_down:          cover property (@(posedge clk) elevator_control==ELEV_MOTOR_DOWN);


cov_elevator_top_cfg_all_zero:      cover property (@(posedge clk) cfg_rd_data==4'b0000);
cov_elevator_top_cfg_all_ones:      cover property (@(posedge clk) cfg_rd_data==4'b1111);


// ***********************************************************************
// Coverage (Functional Sequences)
// ***********************************************************************
cov_elevator_top_up_down_up_down:   cover property (@(posedge clk) state==ELEVATOR_UP   ##[1:$]
                                      state==ELEVATOR_DOWN ##[1:$]
                                      state==ELEVATOR_UP   ##[1:$]
                                      state==ELEVATOR_DOWN);


// This cover shows requests coming in while we are going up. So we have to stop along the way to the highest floor
cov_elevator_top_up_wait_up_wait_down:cover property (@(posedge clk) state==ELEVATOR_UP ##5
                                      state==WAIT       ##6
                                      state==ELEVATOR_UP ##3
                                      state==WAIT       ##[1:$]
                                      state==ELEVATOR_DOWN);


// This cover shows the upper floor being re-programmed as we are going up
cov_elevator_top_up_cfg_up_cfg_down:  cover property (@(posedge clk) state==ELEVATOR_UP ##[10:$]
                                      ~$stable(cfg_reg_top_floor) ##[1:$]
                                      state==ELEVATOR_UP ##[1:$]
                                      ~$stable(cfg_reg_top_floor) ##[1:$]
                                      state==ELEVATOR_DOWN);
```

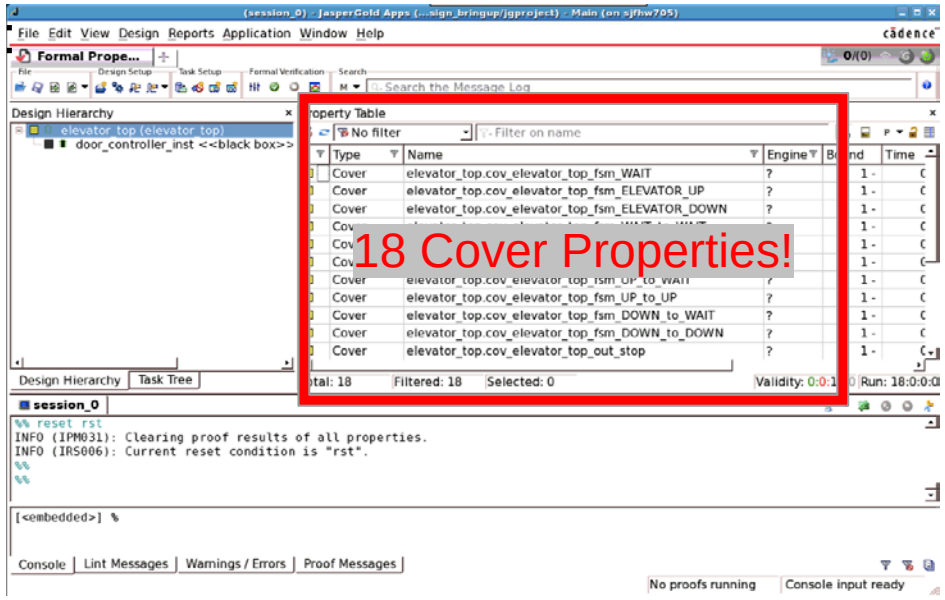
Exercise Design

- Now that we have our “good behavior” cover properties, let’s reload the `run_elevator_top.tcl` script, which will recompile our design to include these properties.
- Type the following into the JasperGold Console:

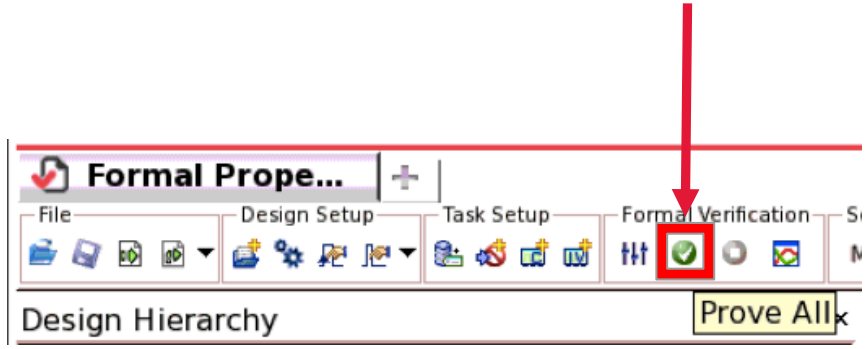
```
[<embedded>] % include run_elevator_top.tcl
```

Console | Lint Messages | Warnings / Errors | Proof Messages

- You will now see that we have 18 cover properties in the design



- Click on *Prove All* in the toolbar:



Exercise Design (continued)

- You will see that green checkmarks show up next to each property, indicating that JasperGold was able to cover that property.

- If we type `fsm` in the property filter box, we can see that all of our FSM states and transitions have been reached:

Property Table

No filter | fsm

	Type	Name	Engine	Bound	Time
✓	Cover	elevator_top.cov_elevator_top_fsm_WAIT	N	1	
✓	Cover	elevator_top.cov_elevator_top_fsm_ELEVATOR_UP	Hp	6	
✓	Cover	elevator_top.cov_elevator_top_fsm_ELEVATOR_DOWN	Hp	3	
✓	Cover	elevator_top.cov_elevator_top_fsm_WAIT_to_WAIT	Hp	2	
✓	Cover	elevator_top.cov_elevator_top_fsm_WAIT_to_UP	Hp	6	
✓	Cover	elevator_top.cov_elevator_top_fsm_WAIT_to_DOWN	Hp	3	
✓	Cover	elevator_top.cov_elevator_top_fsm_UP_to_WAIT	Hp	7	
✓	Cover	elevator_top.cov_elevator_top_fsm_UP_to_UP	Hp	7	
✓	Cover	elevator_top.cov_elevator_top_fsm_DOWN_to_WAIT	Hp	4	
✓	Cover	elevator_top.cov_elevator_top_fsm_DOWN_to_DOWN	Hp	4	

Note: You may see different information here depending on the tool version and settings. This RAK will not be covering formal engines.

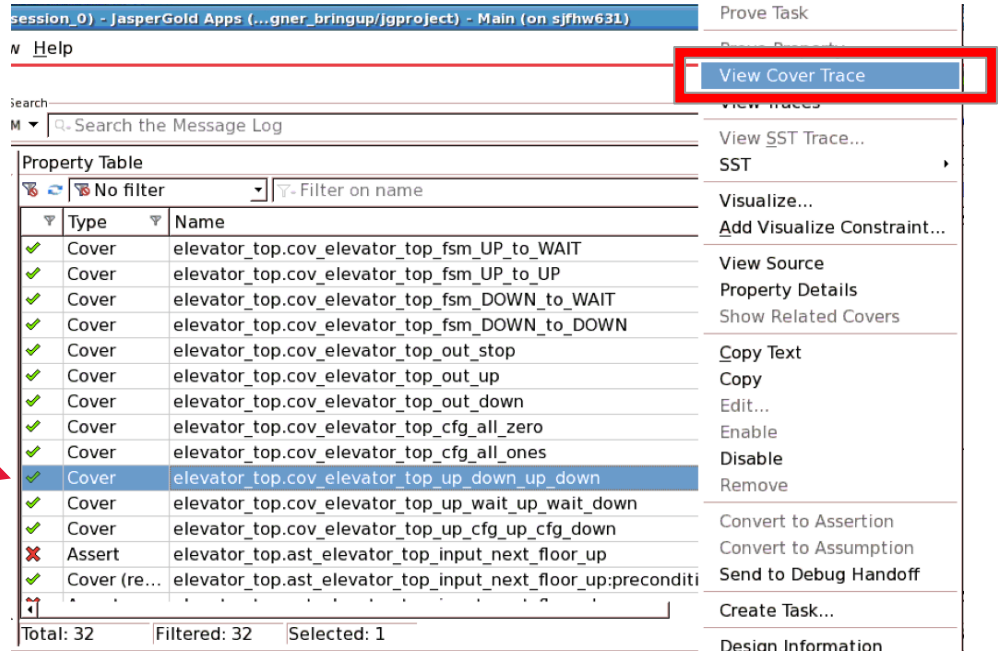
- ✓ *Reachability*
  - ✓ All FSM states and transitions are reachable
  - ✓ All valid output states are reachable
  - ✓ Configuration bits can be programmed
- *Functional Sequences*
  - Elevator goes all the way up and down twice (UP → Down → Up → Down)
  - Elevator goes up, then stops, then continues going up, then goes down (UP → Wait → Up → Down)
  - While going up, we reprogram the top floor (UP → CFG → UP → CFG → Down)

- By browsing/filtering the *Property Table*, we can quickly see that all of our reachability requirements have been met!
- Since we used a good naming convention for our properties, searching for properties is easier.

Exercise Design (continued)

- Let's take a look at the waveform for one of the functional sequences.
- We want to inspect the waveform to confirm that the behavior is reached in a "reasonable" way. This might not be the only way it can be reached, but it is often the shortest way.
- Right-click on `elevator_top.cov_elevator_top_up_down_up_down` and choose *View Cover Trace.*

- ❑ *Reachability*
  - ❑ All FSM states and transitions are reachable
  - ❑ All valid output states are reachable
  - ❑ Configuration bits can be programmed
- ❑ *Functional Sequences*
  - ❑ Elevator goes all the way up and down twice (UP → Down → Up → Down)
  - ❑ Elevator goes up, then stops, then continues going up, then goes down (UP → Wait → Up → Down)
  - ❑ While going up, we reprogram the top floor (UP → CFG → UP → CFG → Down)

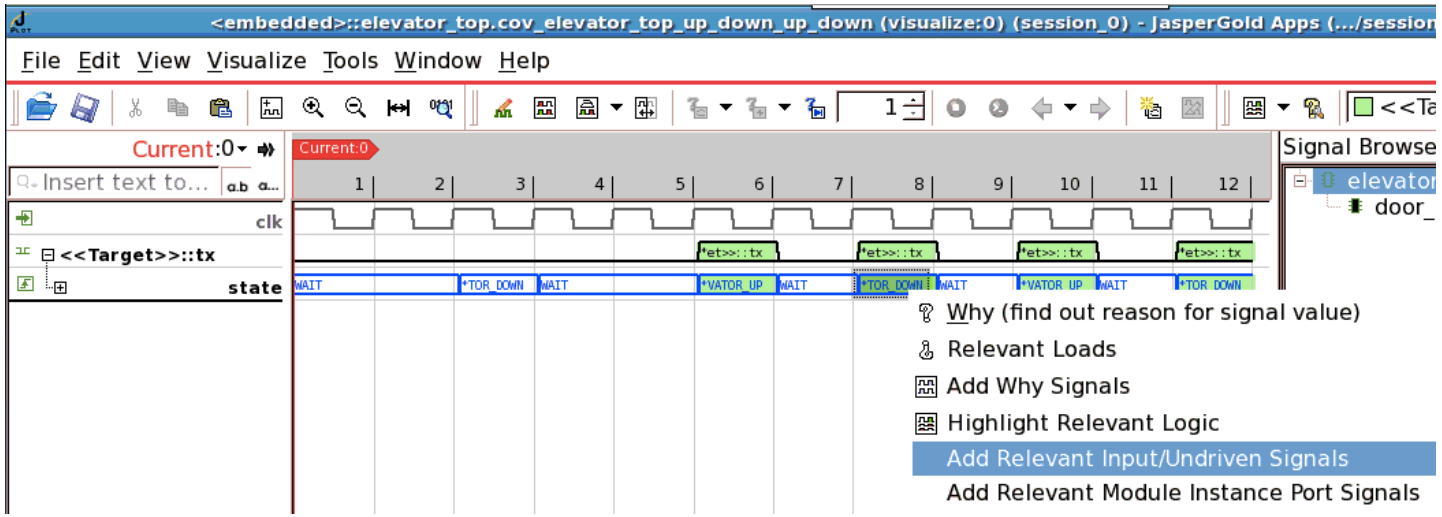


Exercise Design (continued)

“Why” analysis

Add relevant/undriven signals

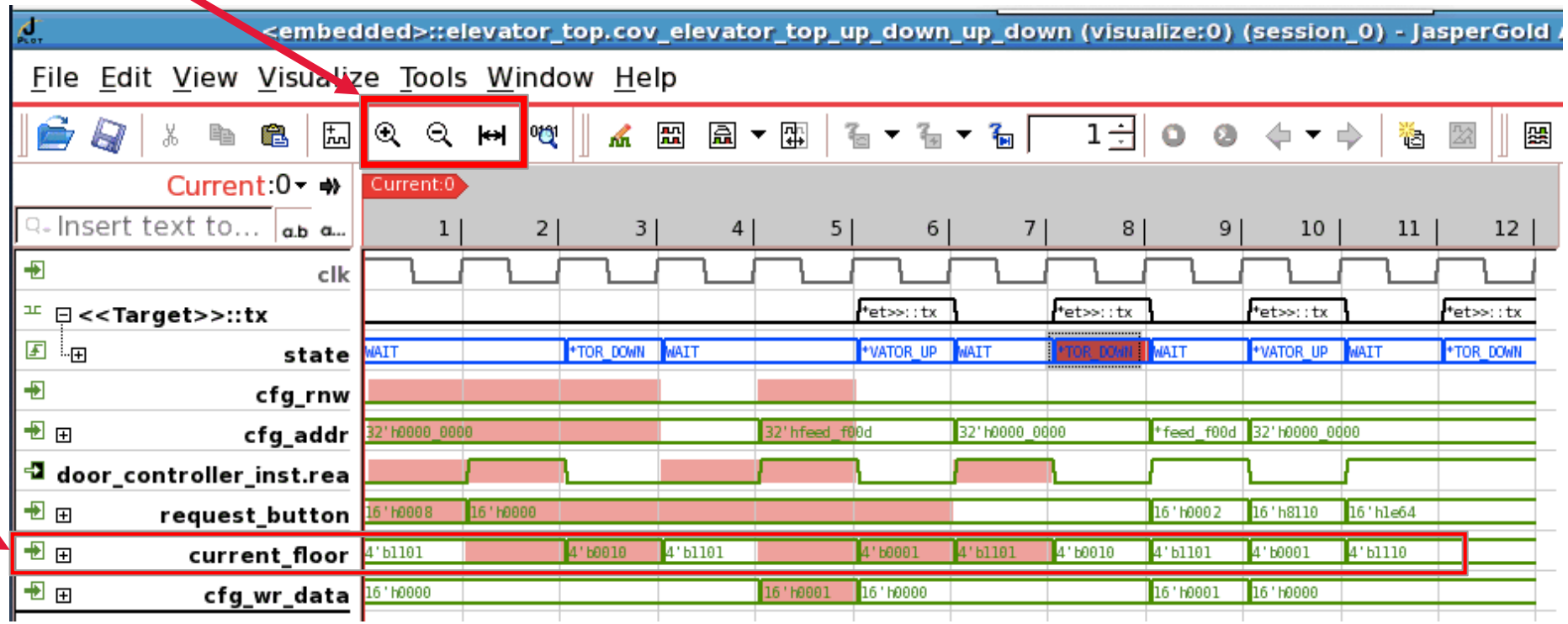
- The Visualize window launches, showing the trace where the cover property is reached.
- Something does not look right. We see that the FSM is transitioning too quickly. It is toggling between `DOWN` and `UP` several times within only 12 cycles!
- Let’s see why it is immediately going back `DOWN` in cycle 8.
- Right-click on the value of `state` in cycle 8 and choose *Add Relevant Input/Undriven Signals*.



Exercise Design (continued)

- We can see that Visualize has provided us with the values at the primary inputs automatically!
- What we see is that the `current_floor` input is not behaving legally. It is jumping randomly from floor to floor!
- Use the zoom buttons on the toolbar to zoom in to see what is going on.

Not Legal!



Add Properties

"Assume-Guarantee"

Converting assert to assume

- We check with the design architect who says that the current floor input comes from the output of a 'sensor decoding' block. This block should never skip floors from cycle to cycle. In addition, it should only change if the elevator control is indicating that the motors are moving the elevator up or down!

- This seems like something we want to guarantee. If any upstream module (or simulation) violates this behavior, we need to know about it.

- Since this behavior should be guaranteed, we should first write it as an SVA assertion that compiles with our RTL. Then, if we are ever testing this module as the top level (which we are doing here), we can convert it to an assumption using a special command in JasperGold. ***This methodology is sometimes called "Assume-Guarantee."*** We are assuming behavior at the inputs to our block, and this behavior can be verified in other formal testbenches, and even in simulation.

- Another way to do this is to have each of these properties be an `assert property <expression>` by default and have an `` `ifdef ASSUME_INPUTS `` macro choose an `assume property <expression>` version of the same property. You want to make it an assert by default, because if this module is ever instantiated somewhere else, we do not want assumptions on signals that are not the top level.

Copy from <RAK_install>/solution/sva_input_properties.txt to bottom of elevator_top.v, above "endmodule"

```
// ****************************************************************************
// Assume or Assert behavior at the input to the elevator controller
// Note: We will convert these to assumptions in the TCL script, if necessary
// ****************************************************************************
ast_elevator_top_input_next_floor_up:          assert property (@(posedge clk)
elevator_control==ELEV_MOTOR_UP    |=> current_floor==($past(current_floor) + 1'b1))
else $display ("ERROR: motor up and current floor not incrementing");

ast_elevator_top_input_next_floor_down:        assert property (@(posedge clk)
elevator_control==ELEV_MOTOR_DOWN  |=> current_floor==($past(current_floor) - 1'b1))
else $display ("ERROR: motor down and current floor not decrementing");

ast_elevator_top_input_current_floor_behavior:  assert property (@(posedge clk)
elevator_control== ELEV_MOTOR_STOP |=> $stable(current_floor)) else $display
("ERROR: current floor changed when motor was stopped");
```

Copy from <RAK_install/solution/assert_to_assume.tcl to bottom of run_elevator_top.tcl

```
assume -from_assert <embedded>::elevator_top.ast_elevator_top_input_next_floor_up
assume -from_assert <embedded>::elevator_top.ast_elevator_top_input_next_floor_down
assume -from_assert <embedded>::elevator_top.ast_elevator_top_input_current_floor_behavior
```

Re-Exercise Design with Added Properties

- We can now exercise the design again with our new assumptions.
- Type the following into the JasperGold Console:

```
[<embedded>] % include run_elevator_top.tcl
```

Console | Lint Messages | Warnings / Errors | Proof Messages

- If you scroll through the *Property Table*, you will now see our 3 new assumptions:

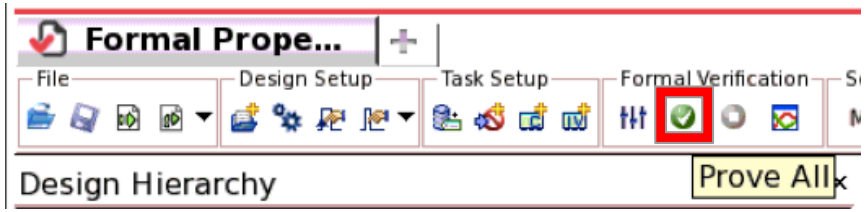
Property Table

No filter | Filter on name

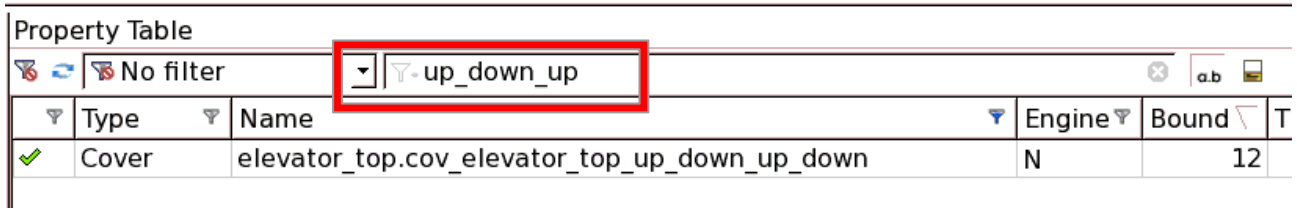
Type	Name
Cover (re...	elevator_top.ast_door_controller_output_ready_to_move:precondition1
Assert	elevator_top.ast_elevator_top_output_up_behav
Cover (re...	elevator_top.ast_elevator_top_output_up_behav:precondition1
Assert	elevator_top.ast_elevator_top_output_dn_behav
Cover (re...	elevator_top.ast_elevator_top_output_dn_behav:precondition1
Assert	elevator_top.ast_elevator_top_ready_to_open
Cover (re...	elevator_top.ast_elevator_top_ready_to_open:precondition1
Assume	elevator_top.ast_elevator_top_input_next_floor_up:assume
Assume	elevator_top.ast_elevator_top_input_next_floor_down:assume
Assume	elevator_top.ast_elevator_top_input_current_floor_behavior:assume

Re-Exercise Design (continued)

- Use the prove button again to prove all the properties.



- In the *Property Table* filter bar, type up_down_up_down to search for the property we previously looked at:

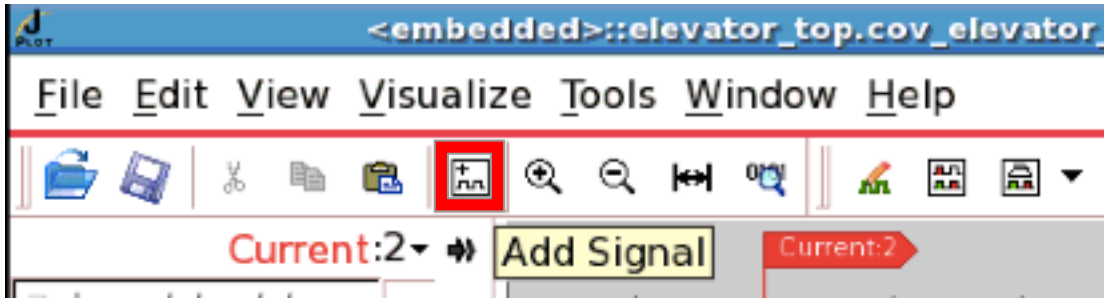


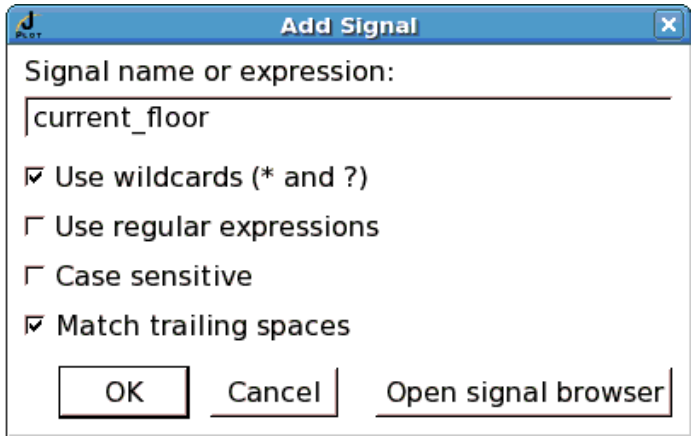
- Double-click on the property to open the trace, which will now have all of our assumptions applied.

Re-Exercise Design (continued)

Add signal

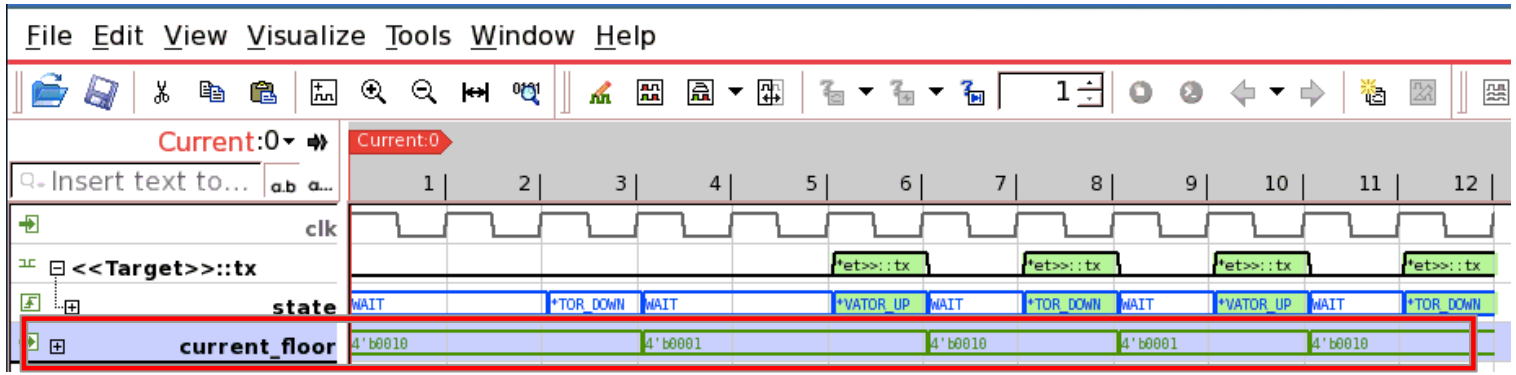
- Let's click on the *Add Signal* button and add `current_floor` to confirm that our assumption fixes the floor behavior.





- As we can see, `current_floor` looks good. It is incrementing and decrementing by 1.

- But why is `state` going `DOWN` in cycle 3 and back `UP` in cycle 6 and `DOWN` again in cycle 8?

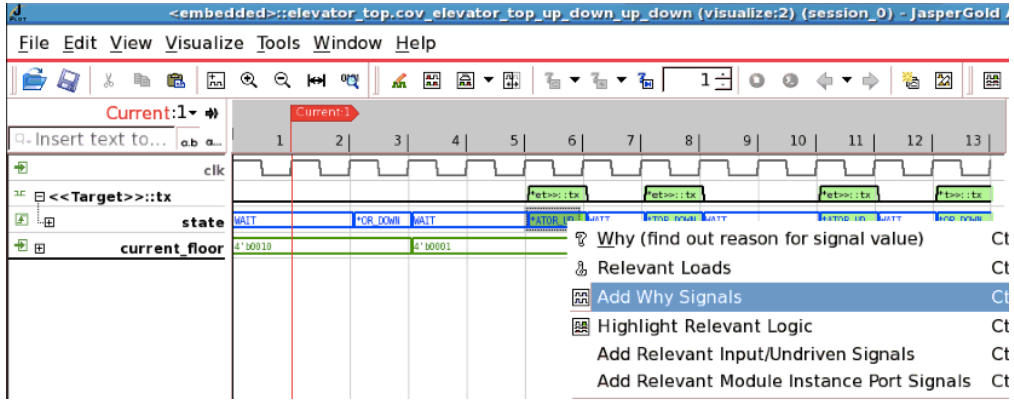


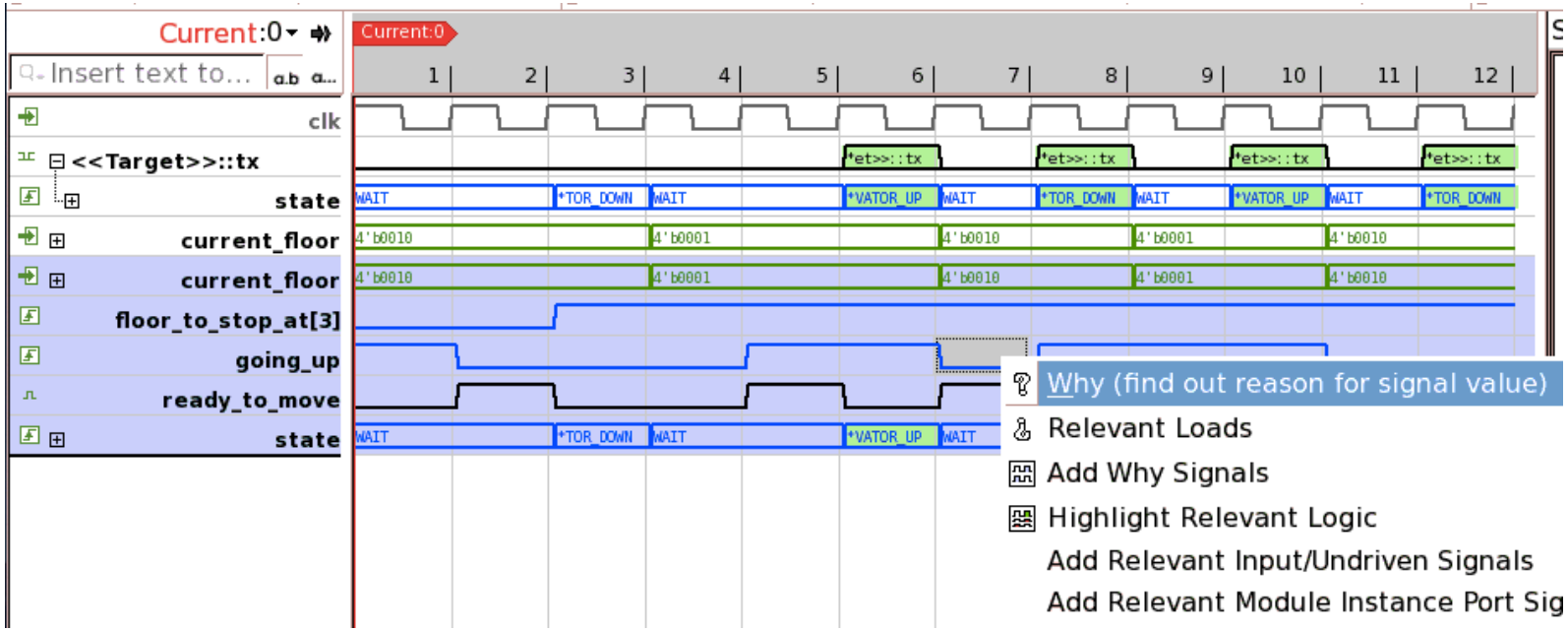
Re-Exercise Design (continued)

Add Why signals

Why analysis

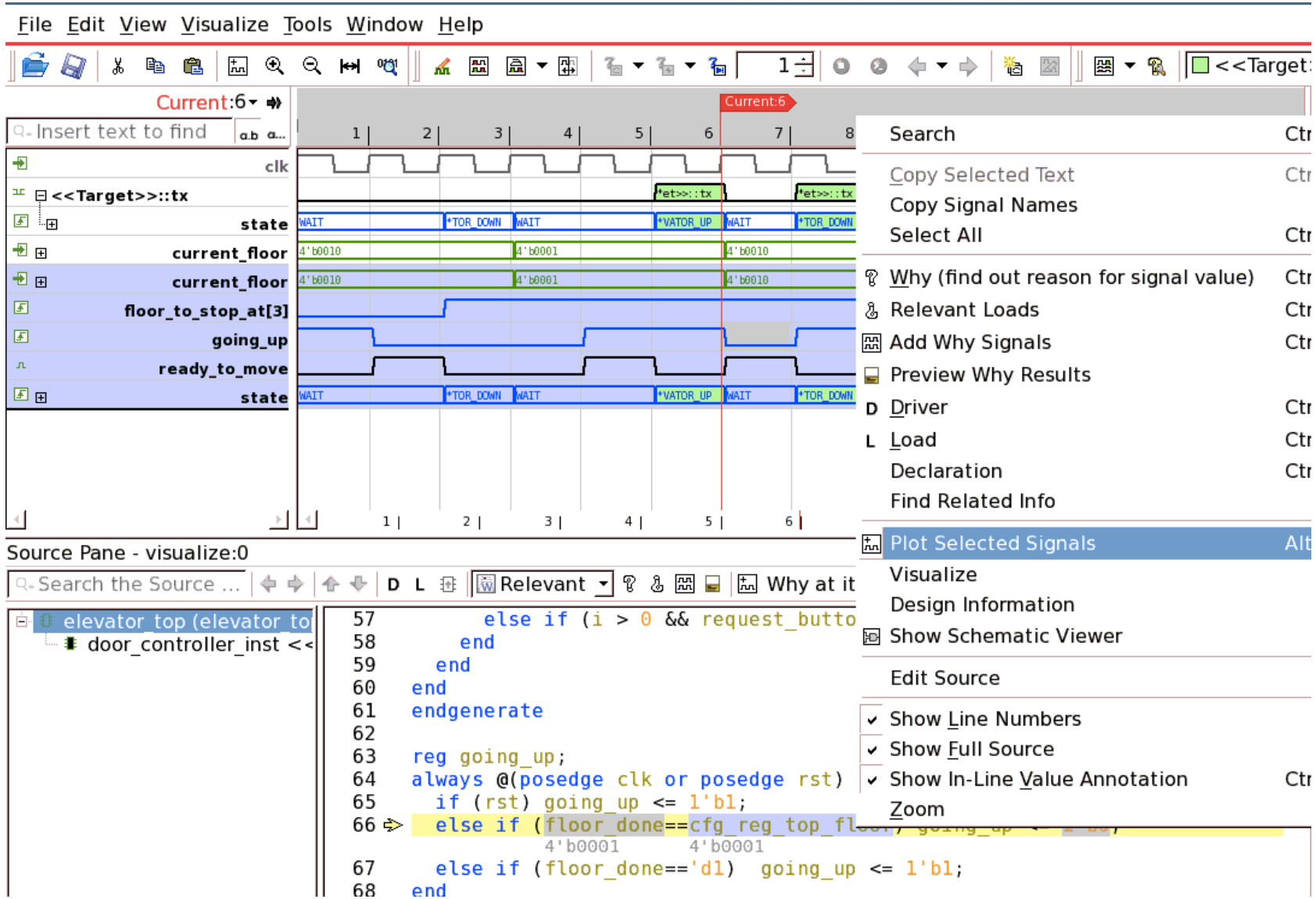
- Right-click on `state` in cycle 6 and choose *Add Why Signals.*
- The signals that contribute to `state` changing in cycle 6 are added automatically.
- We know from the design description that the `going_up` flag toggles when it equals `cfg_reg_top_floor`.
- Let's see if this is what is happening. Right-click on `going_up` in cycle 7 and choose *Why.*





Re-Exercise Design (continued)

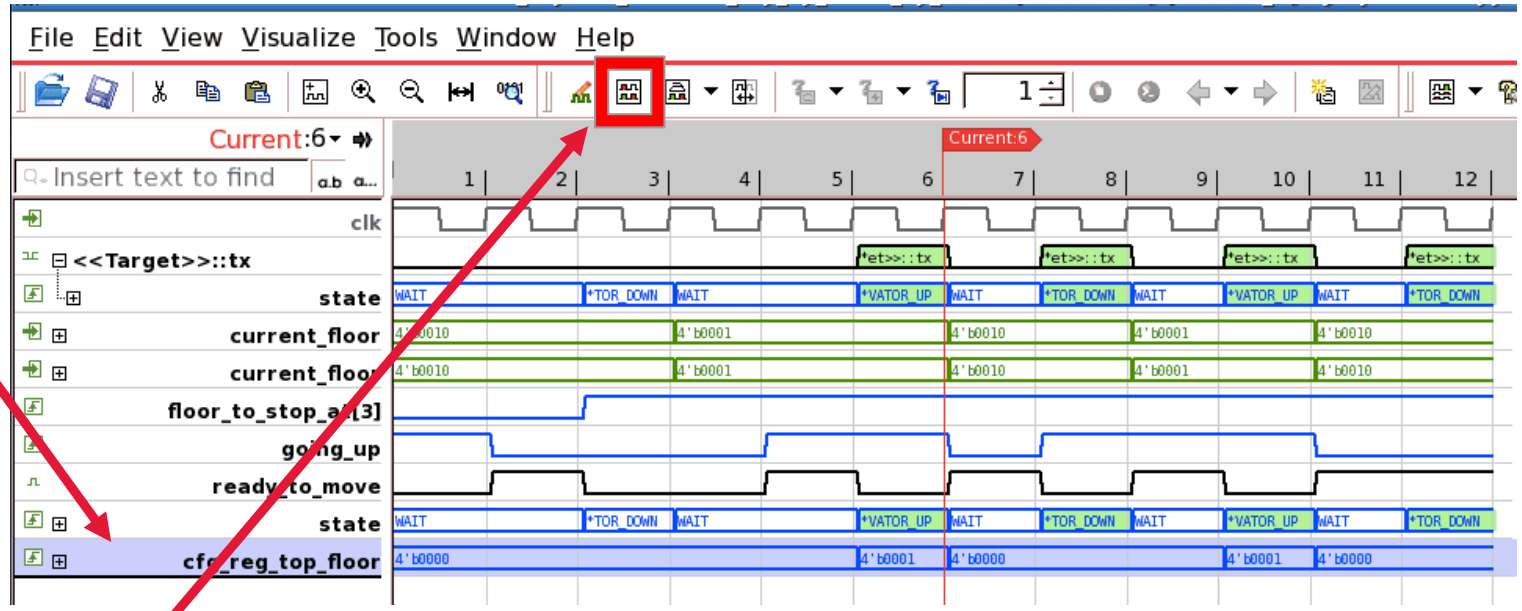
- After choosing *Why* on a signal in a given cycle, the source code pane appears.
- The source code is annotated with the signal values contributing to why `going_up` went low in cycle 7.
- We can see that it is pointing to line 66, and that `cfg_reg_top_floor` is equal to `4'b0001`.
- This is why `going_up` toggles low!
- Let's plot the value of `cfg_reg_top_floor`.
- In the source code pane, right-click on `cfg_reg_top_floor` and choose *Plot Selected Signals*.
- You can optionally just drag any signal from the source browser to the waveform.

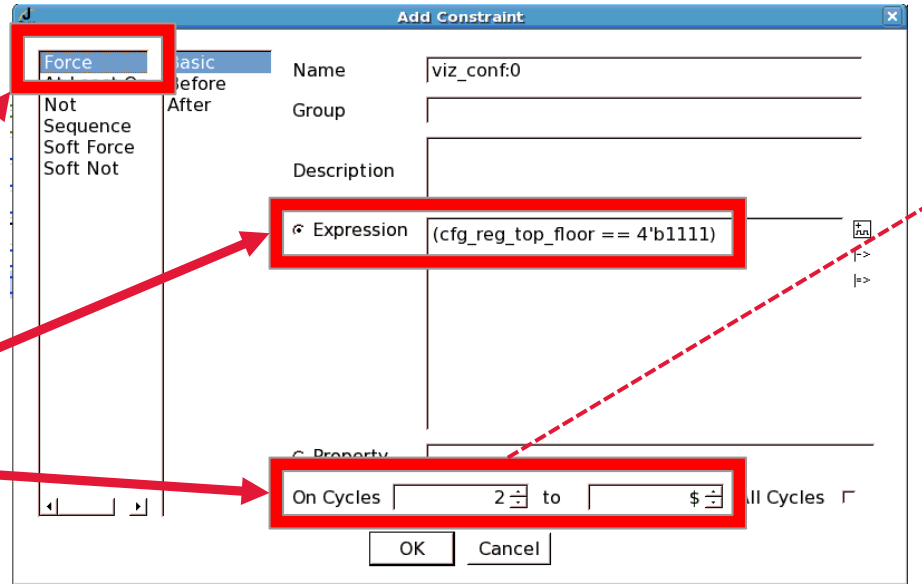


Re-Exercise Design (continued)

Add Visualize constraint

- JasperGold is driving the configuration logic so that `cfg_reg_top_floor` changes periodically.
- This might be realistic, but let's say we wanted to see what happens if we force the value of `cfg_reg_top_floor` to be `4'b1111`?
- We do not want to add an SVA constraint because that is a dangerous over-constraint!
- Since this is just a quick experiment, we can add the constraint so that it only exists inside this Visualize window.
- Click on the value of `cfg_reg_top_floor` in any cycle.
- Now click on the *Add Constraint* button in the toolbar.
- In the Add Constraint dialog, choose `Force`, set the value to `4'b1111`, and specify *On Cycles* with 2 to $. Then press *OK*.





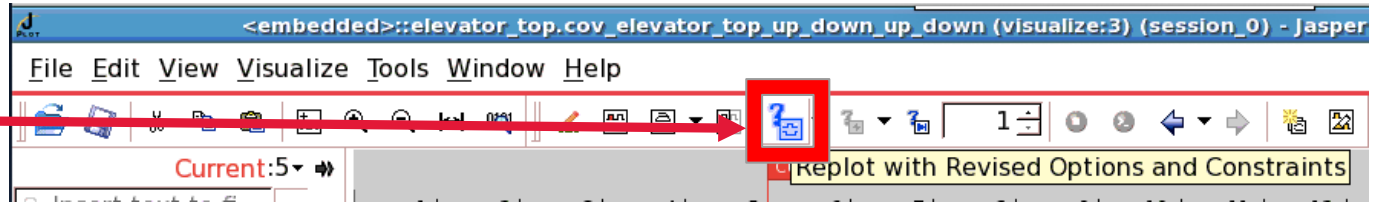
Why did we choose 2 instead of 1 as the starting cycle?

Because "cfg_reg_top_floor" is a flop that MUST take on its reset value of 4'b0000 in cycle 1

cadence

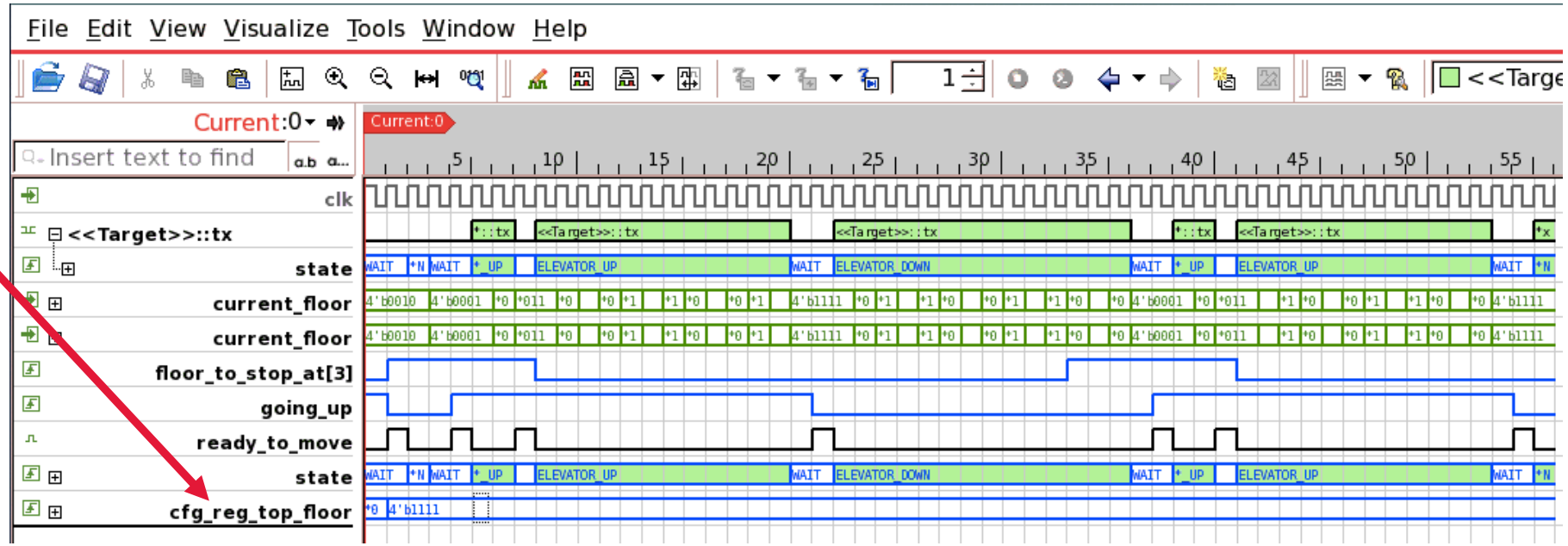
Re-Exercise Design (continued)

- Press the *Replot* button to replot with our new constraint.



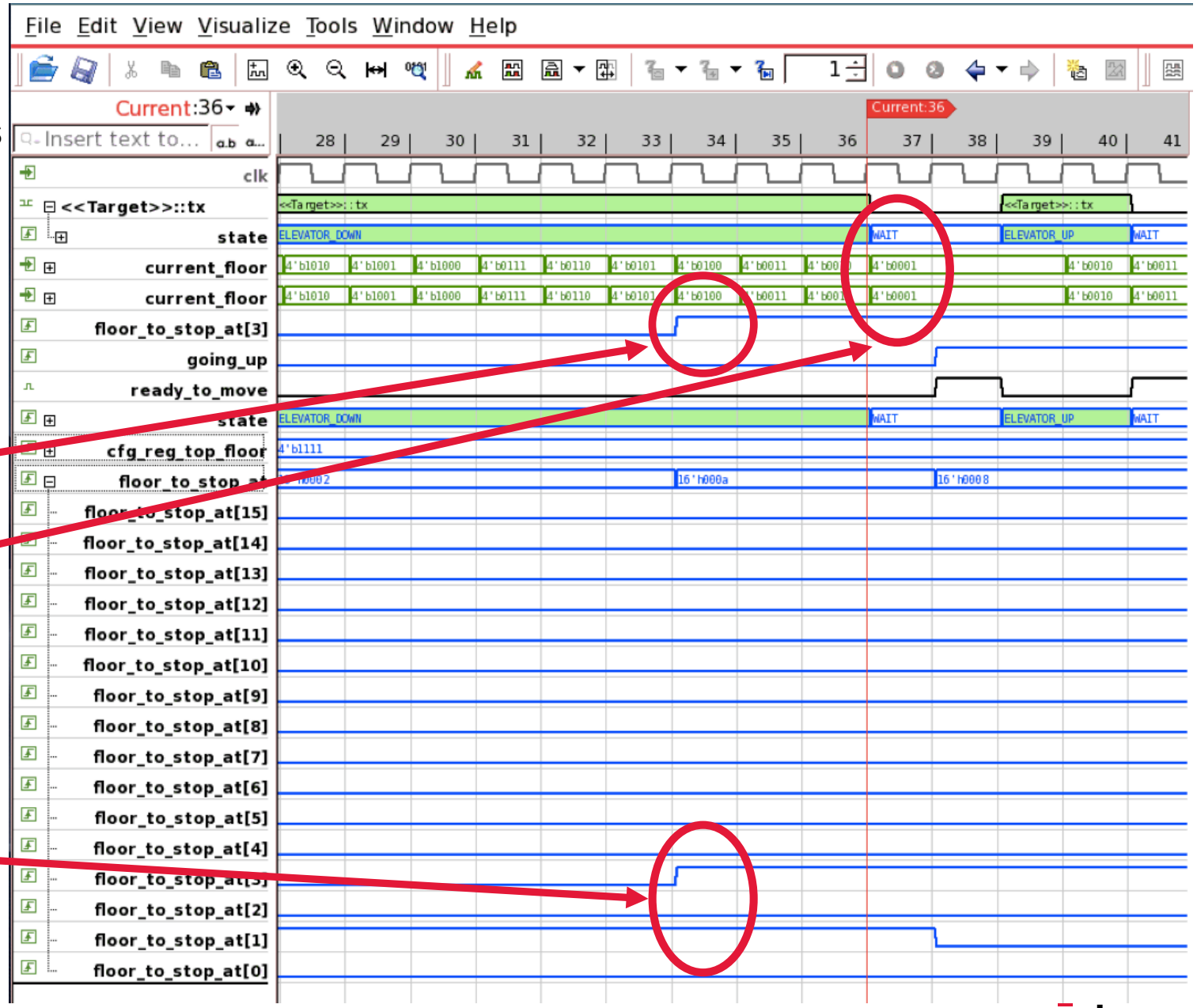
- We can see that `cfg_reg_top_floor` is programmed to be `4'b1111` from 1 to $

- Now our trace is much longer, which makes sense since the elevator has to go all the way up to floor 15 and back down to 1!



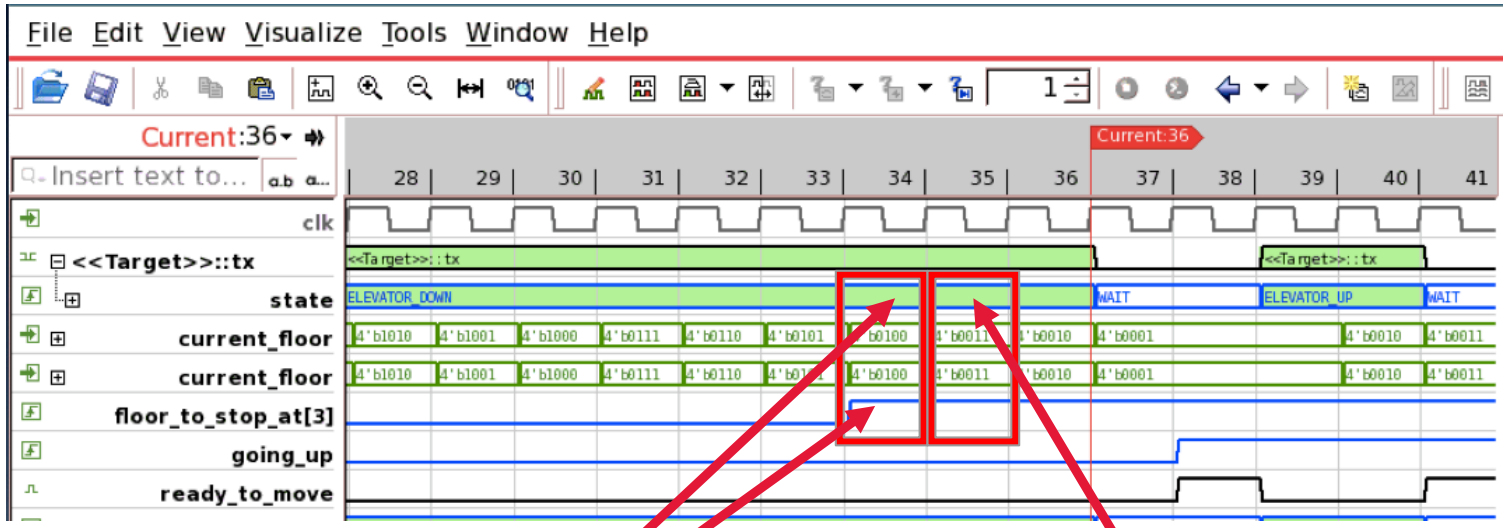
Re-Exercise Design (continued)

- Closer inspection of the waveform reveals that something is wrong.
- From the design specification, we know that when `floor_to_stop_at[current_floor]` is true, we should stay at that floor (`state==WAIT`).
- We can see that `floor_to_stop_at[3]` is set.
- But instead of stopping at floor 3, `current_floor` goes all the way back to floor 1 before we go to state `WAIT`!
- This is a bug!
- If we add the entire `floor_to_stop_at` vector to the waveform, we can clearly see that we should stop at at both floor 3 and floor 1.



Re-Exercise Design (continued)

- This is a good opportunity to add an assertion property to check that this behavior never happens.
- Let's use the bad behavior in the waveform as a guide.
- In cycle 34, we know we need to stop at floor 3 on cycle 35. So `state` should equal WAIT in cycle 35.
- As shown on the right, we can write what we want to check in English.
- Then we will write the SVA for it.



English

"IF `state` is equal to ELEVATOR_DOWN AND `floor_to_stop_at[current_floor-1]` is true… …THEN `state` should be WAIT on the next cycle."

SVA

```
Assert (state==ELEVATOR_DOWN && floor_to_stop_at[current_floor-1]  |=>  state==WAIT)
```

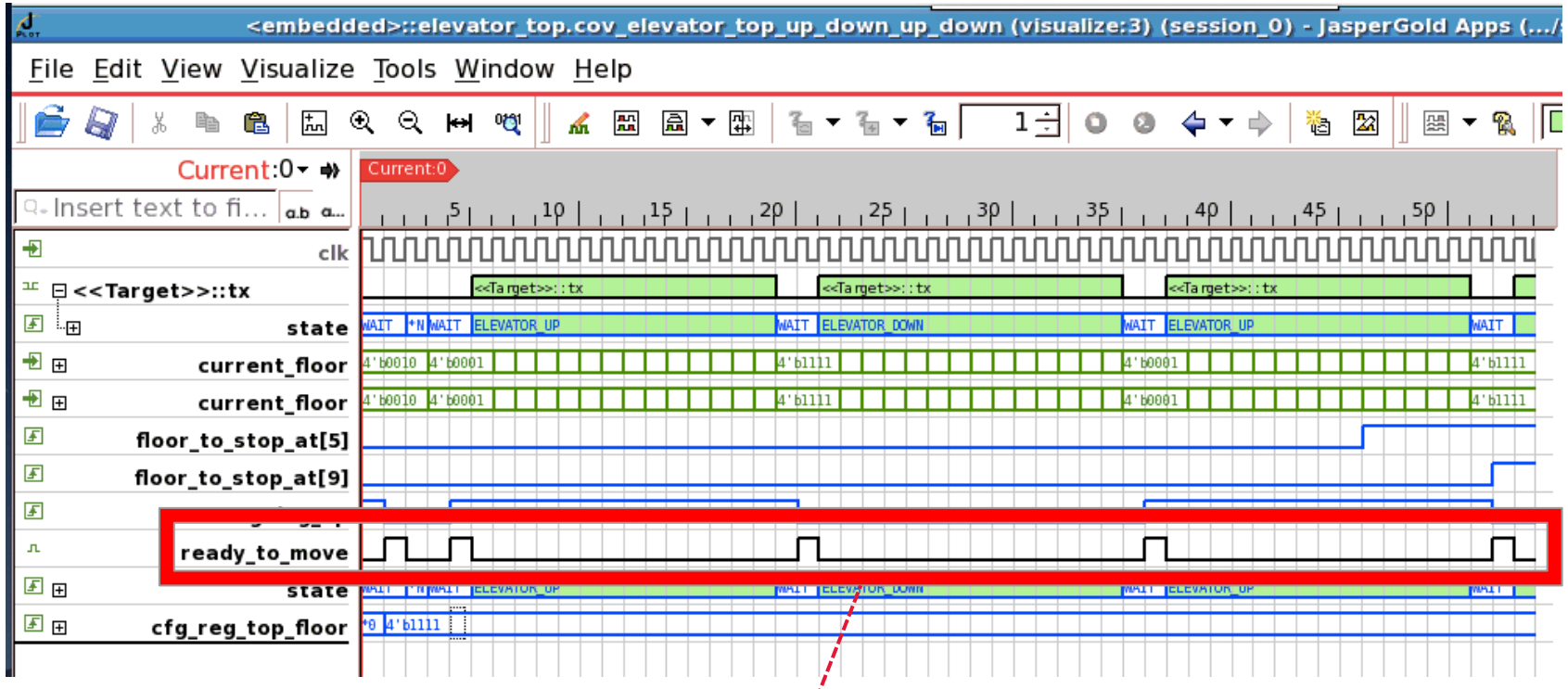
Re-Exercise Design
(continued)

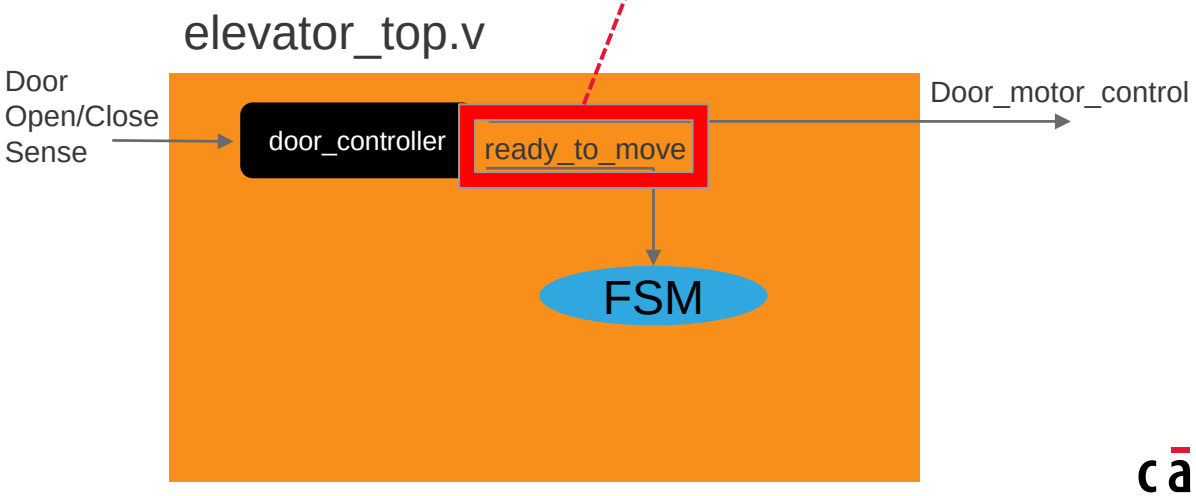
- You'll see below that we actually have two SVA checkers. One for ELEVATOR_UP and one for ELEVATOR_DOWN
- Edit `elevator_top.v` and copy the SVA checker properties as shown. Paste them at the bottom of the RTL, just above `endmodule`. You can also copy these directly from `<RAK_install>/solution/sva_checker_properties.txt`.

```
// ****************************************************************************
// Assert correct behavior of the elevator controller
// ****************************************************************************
ast_elevator_top_stop_at_next_floor_down:      assert property (@(posedge clk) state==ELEVATOR_DOWN && floor_to_stop_at[current_floor-1] |=> state==WAIT)
else $display ("ERROR: did not stop at floor correctly when going down");

ast_elevator_top_stop_at_next_floor_up:        assert property (@(posedge clk) state==ELEVATOR_UP   && floor_to_stop_at[current_floor+1] |=> state==WAIT) else
$display ("ERROR: did not stop at floor correctly when going up");
```

Re-Exercise Design (continued)

- As we inspect this waveform, we also see that `ready_to_move` wiggles too fast.
- This signal comes from the `door_controller`, which is black boxed.
- We check the architectural specification, and find out that when `state==WAIT` the `ready_to_move` signal from the `door_controller` should go high between 1 and 20 cycles later.
- Since we are not responsible for the `door_controller` logic, we should add an assertion at the output of the `door_controller` that verifies this behavior.
- Then, since we are black boxing the `door_controller` , we need to convert this assertion to an assumption using a Tcl command.
- This assumption will allow JasperGold to wiggle the `ready_to_move` signal according to the specification while we test our logic.
- In the next slide, we will fix our RTL and add this property on the `ready_to_move` signal.





Fix RTL and Add Properties

- When we inspect the RTL for the `floor_to_stop_at` bug on the previous page, we find out that we have a typo in the `ELEVATOR_DOWN` state. Fix the following line of RTL:

```
ELEVATOR_DOWN:
  //if (floor_to_stop_at[current_floor+1'b1]) begin // This is the BUG
  if (floor_to_stop_at[current_floor-1'b1]) begin   // This is the FIX!
    // Need to stop at next floor
    state <= WAIT;
    floor_done <= current_floor-1'b1;
  end else if (current_floor==1) begin
    state <= WAIT;
  end else state <= ELEVATOR_DOWN;
```

- Next, we add a property to the `ready_to_move` signal as follows, so that if we black box the `door_controller`, the property is an assumption. If the `door_controller` is not black boxed, it is an assertion.

Copy from <RAK_install>/solution/sva_door_controller_properties.txt to bottom of elevator_top.v, above “endmodule”

```
// ******************************************************************************
// Assume or Assert behavior at the output of the door controller
// Note: Here we are showing one way to convert assumptions to assertions using a
`define
// ******************************************************************************
`ifdef DOOR_CONTROLLER_IS_BBOXED
  asm_door_controller_output_ready_to_move:     assume property (@(posedge clk)
state==WAIT |-> ##[1:20] ready_to_move) else $display ("ERROR: ready_to_move
failed to occur in the required time");
`else
  ast_door_controller_output_ready_to_move:     assert property (@(posedge clk)
state==WAIT |-> ##[1:20] ready_to_move) else $display ("ERROR: ready_to_move
failed to occur in the required time");
`endif
```

Add the following (in red) to run_elevator_top.tcl

```
# Clear environment
clear -all


# Compile HDL files
analyze -sv elevator_top.v +define+DOOR_CONTROLLER_IS_BBOXED
elaborate -top elevator_top -bbox_m door_controller

clock clk
reset rst

assume -from_assert <embedded>::elevator_top.ast_elevator_top_input_next_floor_up
assume -from_assert
<embedded>::elevator_top.ast_elevator_top_input_next_floor_down
assume -from_assert
<embedded>::elevator_top.ast_elevator_top_input_current_floor_behavior
```

Re-Exercise Design with Added Properties

- Once again, we can exercise the design.
- Type the following into the JasperGold Console:

```
[<embedded>] % include run_elevator_top.tcl
```

Console | Lint Messages | Warnings / Errors | Proof Messages

- If we search for `ready_to_move` with the *Property Table* filter, you will now see our new assumption on the `ready_to_move` signal. In addition, JasperGold has created an automatic cover property that checks to ensure that the precondition of our assumption can be covered!

Property Table

No filter | ready_to_move

Type	Name	Engine	Bound	Time
Assume	elevator_top.asm_door_controller_output_ready_to_move	?		
Cover (related)	elevator_top.asm_door_controller_output_ready_to_move:precondition1	?	1 -	

Conclusion and Next Steps

- We have demonstrated how to go through the process of exercising both reachability and Functional sequences.
- Before we are finished with the overall RTL design bring-up flow, the other two functional sequences should also be exercised, and any failing assertions should be debugged.
- While inspecting these functional sequences and assertion failures, there might be some more bugs you can identify!
- The design bring-up flow gives us confidence that our design can perform basic "good" behaviors. And along the way, we might find some bugs.
- We have not fully verified the behavior against the design specification (with a full set of assertions) and have not checked the formal environment for completeness or overconstraints.

- *Reachability*
  - All FSM states and transitions are reachable
  - All valid output states are reachable
  - Configuration bits can be programmed
- *Functional Sequences*
  - Elevator goes all the way up and down twice (UP → Down → Up → Down)
  - Elevator goes up, then stops, then continues going up, then goes down (UP → Wait → Up → Down)
  - While going up, we reprogram the top floor (UP → CFG → UP → CFG → Down)

- This concludes the RAK!
- For more information about more powerful JasperGold Visualize features, see the following links:
  - Article (20418830) Visualize; Freeze and Add Constraint (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1Od0000000511NEAQ&pageName=ArticleContent
  - Article (20418377) Visualize RTL Signal and Highlight Relevant Logic (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1Od000000050u4EAA&pageName=ArticleContent
  - Article (20418347) Visualize Features: Clone, QuietTrace and Highlight Difference (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1Od000000050taEAA&pageName=ArticleContent
  - Article (20485614) Custom Markers and Vertical Annotation in the Visualize Window (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V0000091BqGUAU&pageName=ArticleContent
  - Article (20466839) JasperGold Visualize GUI Features https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V000007Mh9EUAS&pageName=ArticleContent
- Additional JasperGold Apps are available to help you with producing high quality RTL code. For more information, please see these links:
  - Article (20471679) Introduction to JasperGold® Superlint App (RAK) – JasperGold front-end https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V000006Aia8UAC
  - Article (20468285) Overview of JasperGold X-Propagation Verification App (Video) https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V0000067A1SUAU&pageName=ArticleContent

cādence®