

Synopsys[®] VC Formal Tutorial

Formal Register Verification (FRV)



Portland State University

©2023

Disclaimer:

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contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

The design example in this tutorial is not supplied. You need to use your own files to follow the tutorial steps and instructions

Table of Contents

1. Introduction	3
1.1 About and Usage of FRV	4
1.2 Design Files	5
1.3 TCL File	6
2. Application Setup	8
2.1 Invoking VC Formal GUI	9
2.1.1 User Interface Details	9
2.1.2 Loading TCL File	10
2.2 Invoking VC Formal Along with TCL File	11
3. Application Usage	12
3.1 Detecting Errors	14
3.2 Resolving Errors	18
3.3 Restarting VC Formal	20
Appendix	21
<i>Table 1.1. FRV App Checker Parameter Types</i>	21

1. Introduction

VC Formal uses TCL (Tool Command Language) scripts to tell it what to do. The script can define the app within VCFormal that we want to use, the files we are working on, how we are mapping them, the clock cycles, and more. Instead of diving into the details of TCL scripts, we will use templates. One would then simply need to modify those templates to interact with VC Formal as needed in their project.

To begin, you need to set up the VNCserver as shown in the previous tutorials and open the Linux GUI. For better practices and to keep everything organized, we create a main folder for the design we want to analyze, and in this case, I called mine “FRV_Example”. **Don’t use spaces when naming the files and folders.**

Inside that folder, we create two folders: one is Design, where you put the actual Verilog or SystemVerilog designs we want to analyze, and the other is Run, where we put the TCL that will run the VCFormal FRV analysis for us.

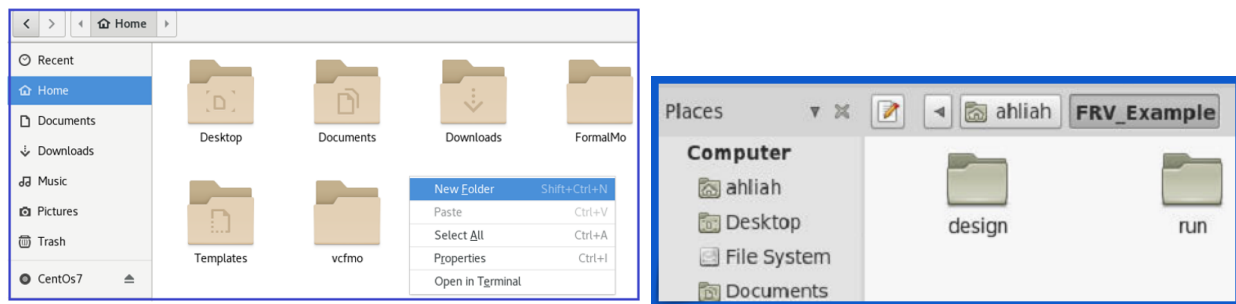
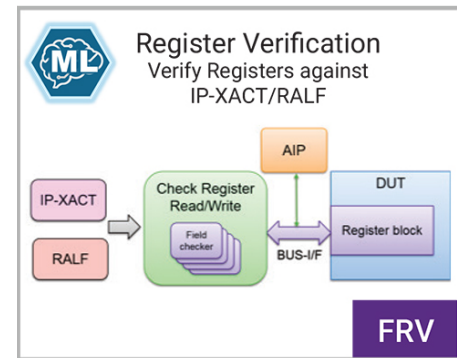


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of FRV

Formal register verification is used to perform formal verification checks for registers in a design. It cuts down the time of performing directed simulation tests by formally verifying register behavior of a design configuration.

Intended register behavior is provided by the user via IP-XACT (.xml file) or RALF (.ralf file) format. This register specification file will be a separate file that compliments your main design file(s) (SV or V file). It will be where you define all registers and/or reg-fields.



During the verification process, FRV will determine if register intent is consistent by checking read/write transactions. Once FRV is initiated, a checker file will be produced. The checker file is what VC Formal uses in order to check on your specified registers. To ensure FRV performance, an additional file is needed to “bind” (SV file) this checker file and your register specification file.

1.2 Design Files

In the Design folder, you'll put the design files. The TCL file will be put in the Run folder. We suggest you name your folders with lowercase letters, as VC Formal is case-sensitive.

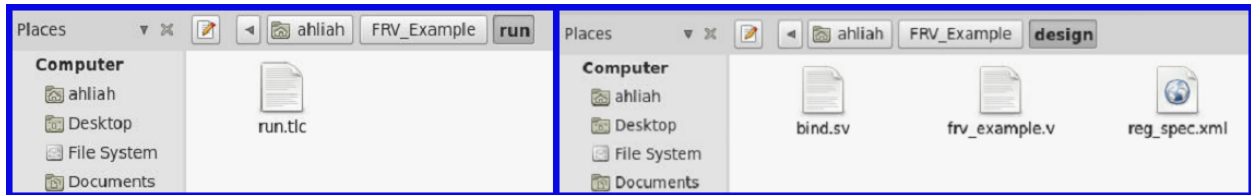


Figure 1.2. Showing the SystemVerilog and TCL files in the correct folder locations.

Shown left of Figure 1.2, the design folder should hold **3 files**:

- A SystemVerilog or Verilog file containing your main module
- A .xml or .ralf file containing your register specifications
- A SystemVerilog file that will “bind” main module file to checker file

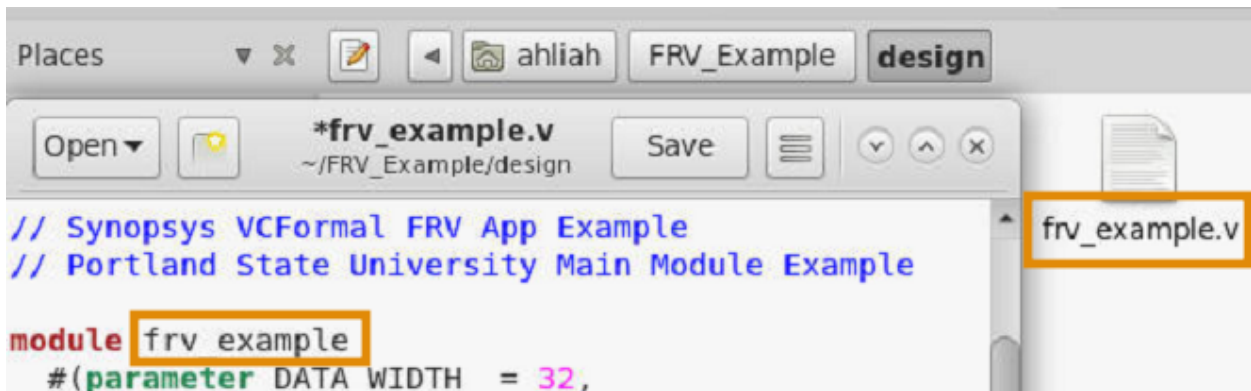


Figure 1.3. Example of the main design module we are using in this tutorial.

In order for VC Formal to map your design, you need to acknowledge and **keep note of your exact module name AND your file name**, as highlighted in Figure 1.3 above. Although I chose to keep the same name for both in this example, it is important to know the difference. This will come in handy when you create your TCL file.

The register specification file is where you define all registers in your main design file in XML or RALF format.

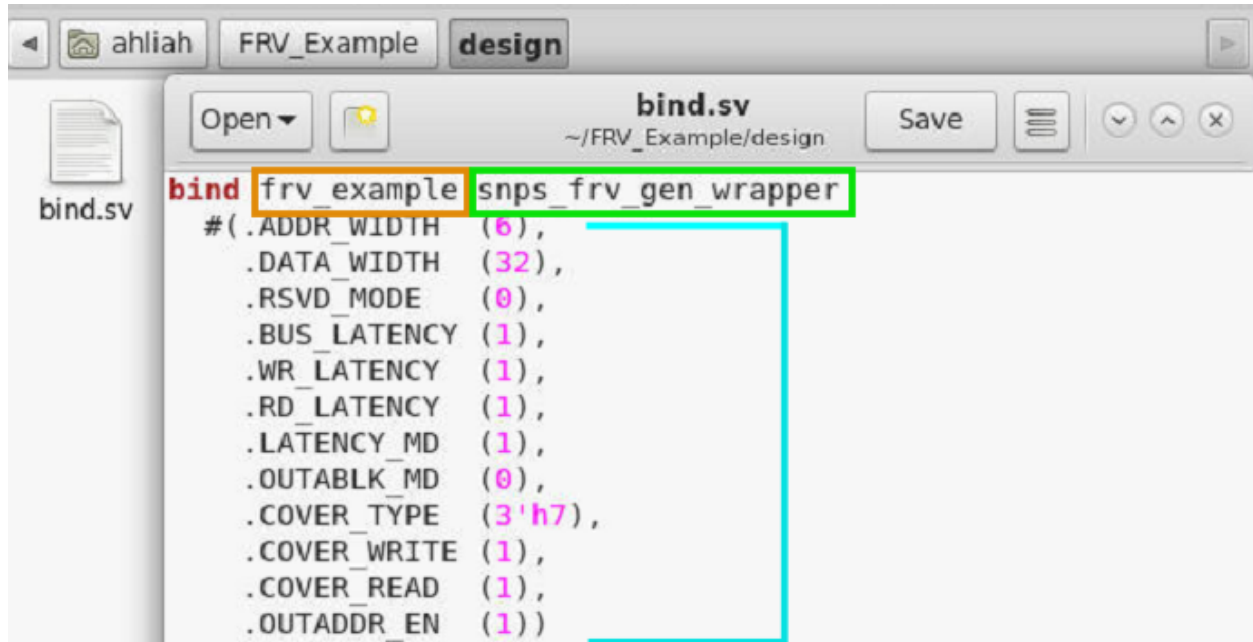


Figure 1.4. Example of the bind file we are using in this tutorial.

The “bind” file will bind the checker file (generated after running TCL file) to your design.

Things to note from Figure 1.4:

- The orange highlight is our design module name
- The green highlight is a generic bus wrapper (recommended)
- The blue highlight encapsulates the checkers parameters (see Appendix Table 1.1)

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (full script in *Figure 1.6*), which you can use as a template for your functional checks on VC Formal.

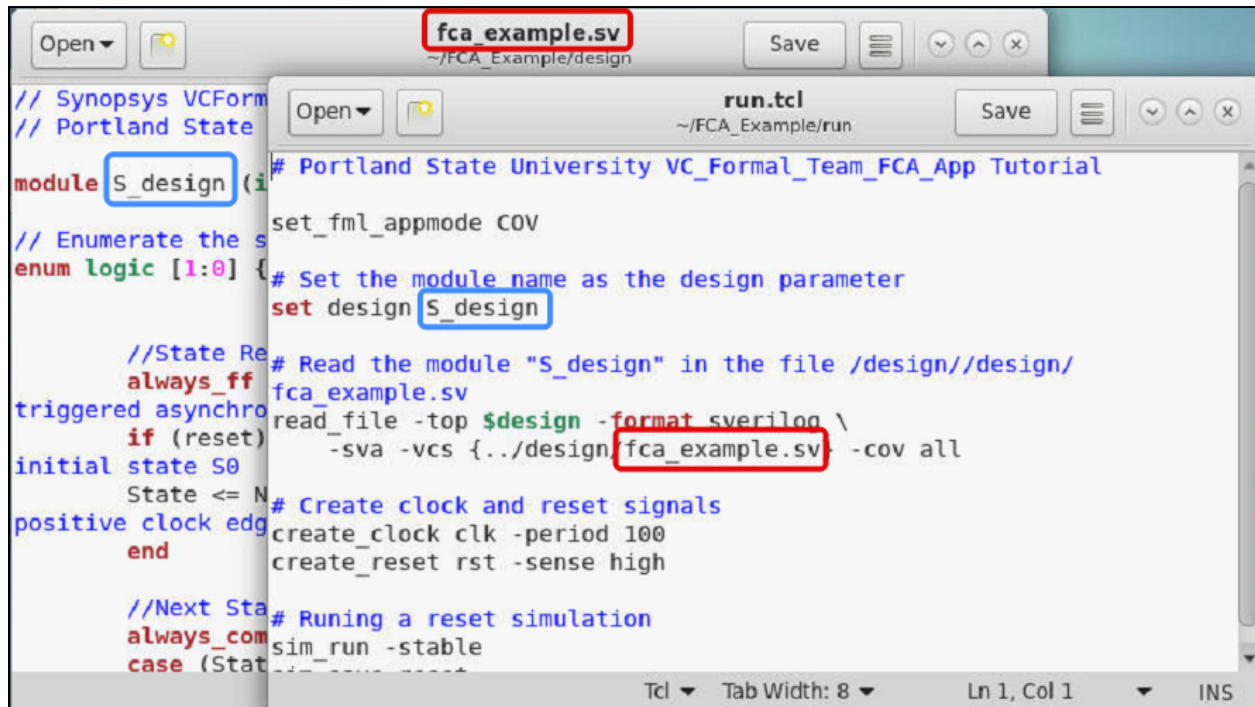


Figure 1.5. Difference between the design file name and module name.

In *Figure 1.5*, both the Design file and TCL files are shown side by side. This is an example from another VC Formal application; this is **NOT** a TCL file for the FRV app.

- ❖ The blue highlights are to demonstrate the **module name**.
- ❖ The red highlights are to demonstrate the **design file name**.

These will **NOT** be the same for every user. You will need to keep track of your own module and design file names in order for your TCL file to work correctly when you try to run in VC Formal.

```

# Portland State University VC_Formal_Team_FRV_App_Tutorial

set_fml_appmode FRV

# Set the module name as the design parameter
set design frv_example 1

# Read register specification file then generate checker file
frv_load -ipxact ../design/reg_spec.xml -auto_load 2

# Read the module "FRV_design" and bind it to checker file
read_file -top $design -format sverilog \
    -sva -vcs "../design/frv_example.v" ../design/bind.sv 3 4

# Create clock and reset signals
create_clock clk -period 100
create_reset rst -sense high

# Running a reset simulation
sim_run -stable
sim_save_reset

```

Figure 1.6. Annotated TCL template file.

- (1) Instruction that sets the appmode to FRV in VC Formal.
- (2) “frv_load” identifier that perform FRV by taking in the register specification file and verifying specified registers in the file.
- (3) Design file location so VC Formal knows where to find the file.
- (4) Bind file location so VC Formal knows where to find the file.

As mentioned before, VC Formal is case sensitive, and in order for it to map your designs, you will need to use the same module name in the TCL script (3) and the module name in the design file.

Any file name (ex. frv_example.sv) can be named however you want.

2. Application Setup

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

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

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

```
$vcf -gui
```

OR

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

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

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

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

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

2.1 Invoking VC Formal GUI

To proceed with invoking VC Formal:

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

vcf -gui

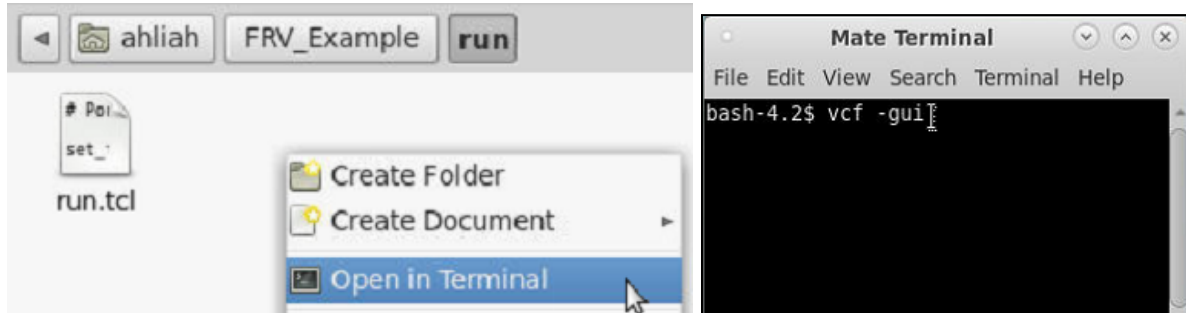


Figure 2.1. Invoking VC Formal in the terminal.

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

You should then see the VC Formal GUI as shown in *Figure 2.2* below. You can toggle between showing Targets, Constraints, or both Targets and Constraints window by clicking the blue box icon in the upper right-hand of the application **(1)**.

It may look like any of these three icons: 

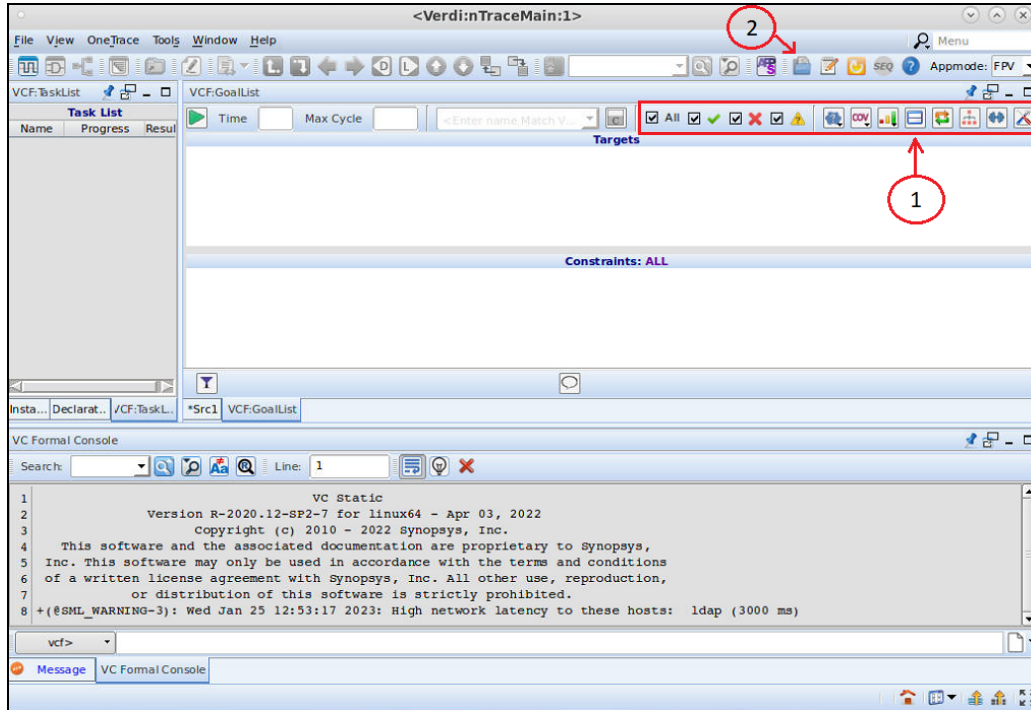


Figure 2.2. VC Formal GUI introductory screen.

Then load a TCL script by clicking on the  icon **(2)** as shown in Figure 2.2.

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

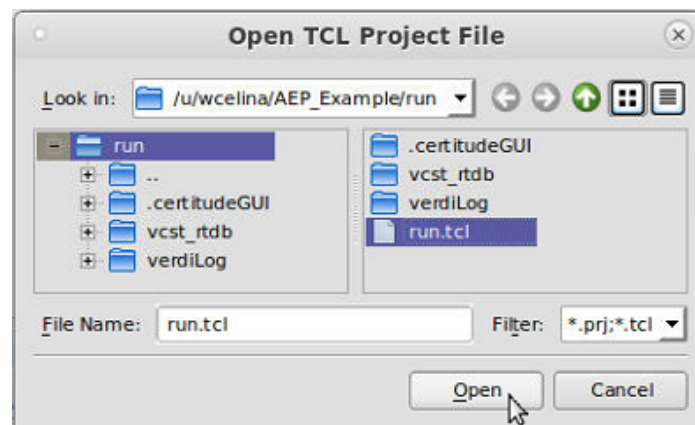


Figure 2.3. Selecting TCL file.

2.2 Invoking VC Formal Along with TCL File:

To proceed with invoking VC Formal:

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

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

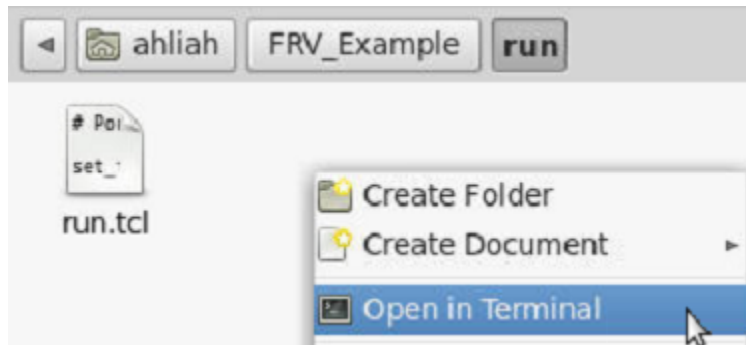


Figure 2.4. Opening terminal in the 'run' folder.

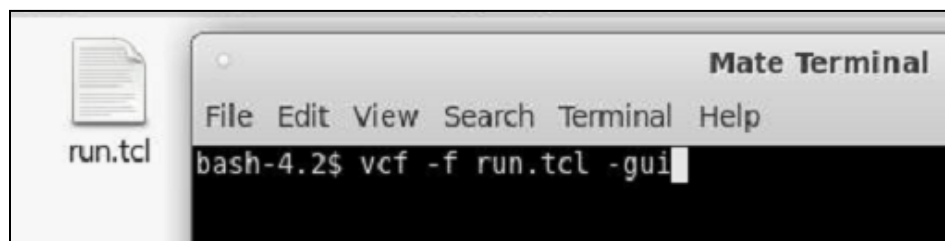


Figure 2.5. Invoking VC Formal and TCL script in the terminal.

And that's it!

3 Application Usage

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

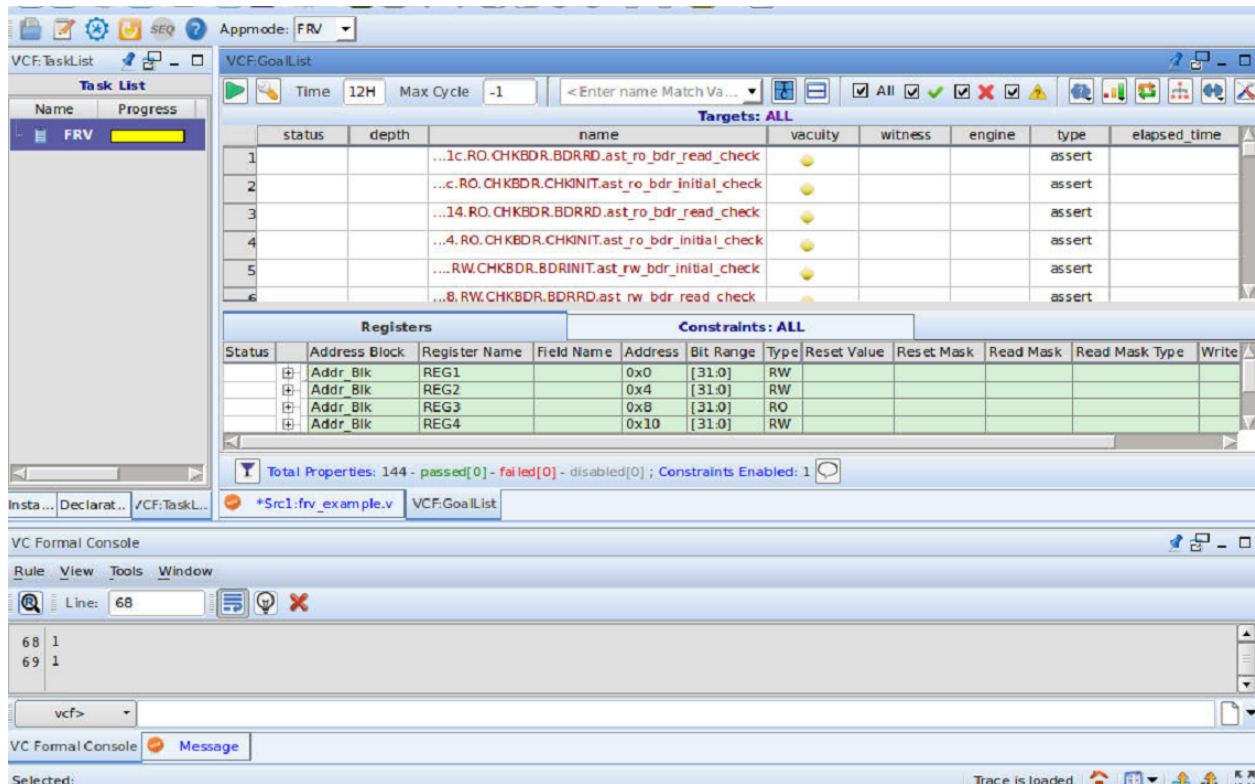


Figure 3.1. Screen after loading TCL script.

Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the *VCF:GoalList* tab.

3.1 Detecting Errors

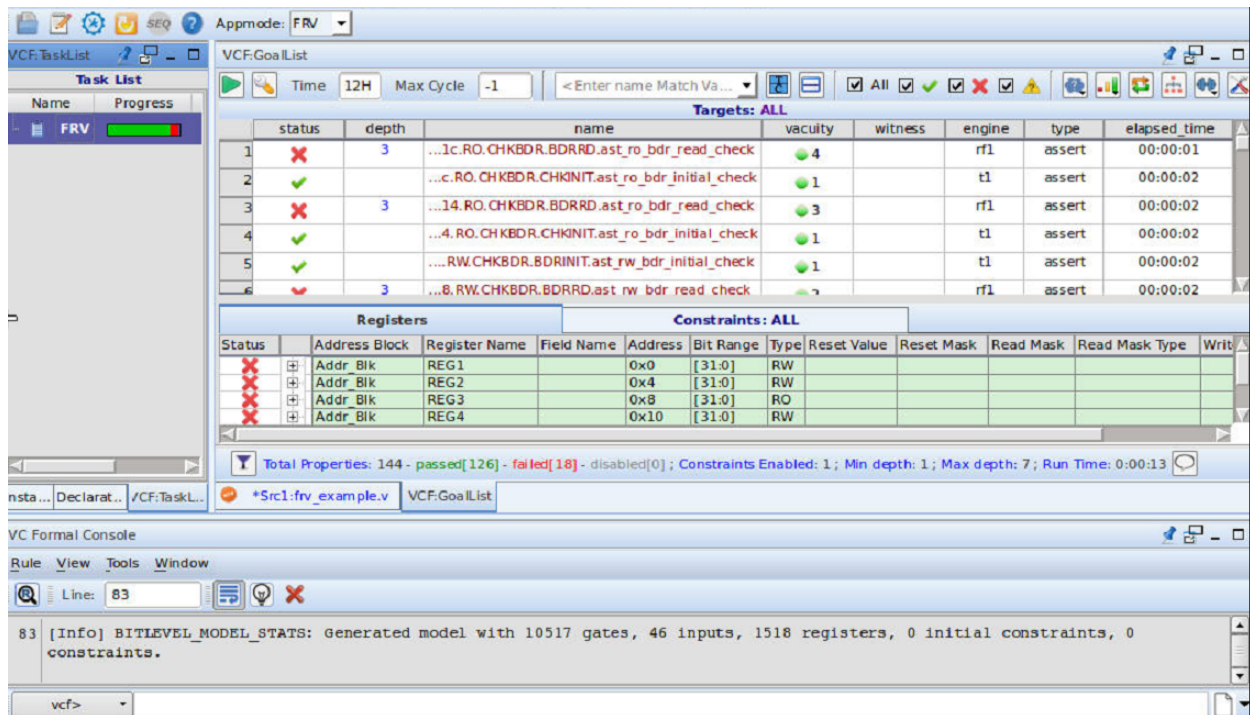


Figure 3.3. Screen after running TCL script and the status given = ✗.

In Figure 3.3 above, under the VCF: GoalList tab, we see icons. VC Formal gives this “Proven” status icon for the part of our design that ran through FPV successfully.

We also see one icon; VC Formal gives this “Falsified” status icon for the part of our design that failed FPV.

Under the Registers tab, the same statuses can be seen. Other information such as Address block, Register Name, Field Name, Address, Bit Range, and Type display the default register attributes from this analysis. The other columns can provide further information but it must be declared in your register specification file. These include:

- Reset Value
- Reset Mask
- Read Mask (and type)
- Write Mask (and type)
- Backdoor

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

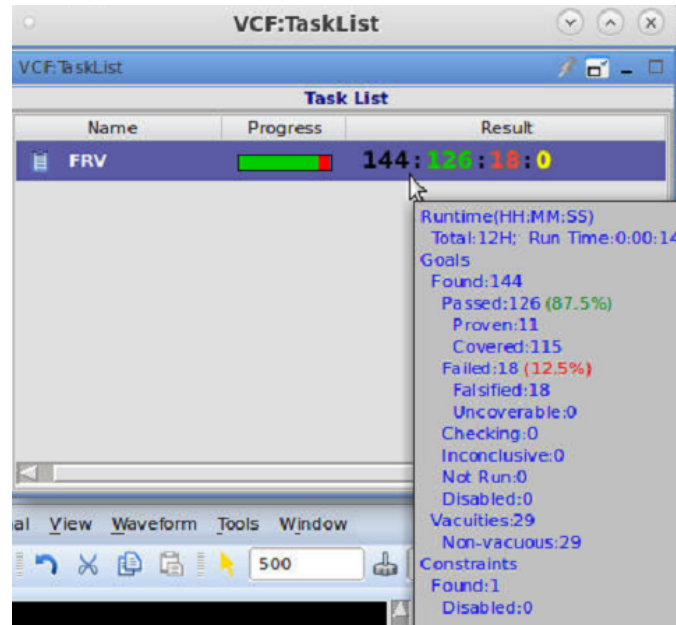


Figure 3.4. Results of the analyzed script.

We can start by looking at the register data in the FRV report. Use the command below in the VC Formal Console window to see the report:

```
frv_report
```

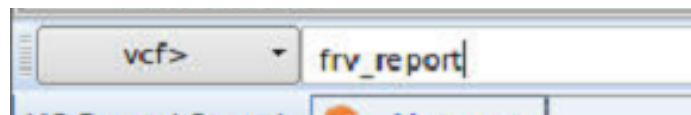


Figure 3.5. VC Formal Console window.

```

86 vcf>frv_report
87 Register Summary:
88   Component      : frv_example
89   MemoryMap/System: Example_Code
90   Address Block   : Addr_Blk
91   Base Address    : 0x0
92   Register Count  : 7
93   Assertions      : 29/11/18/0
94   Covers          : 115/115/0/0
95 Register List: 0x0 : frv_example/Example_Code/Addr_Blk
96   ID      Register/field Name      Address/bit offset      S:
97   Asserts/Covers
98 -----
99   4      REG1                      0x00                      3:
100  16/16/0/0

```

Figure 3.6. VC Formal Console showing a report of register data.

The FRV report will show the Register Summary of your design.

- *Component* is the design module under evaluation.
- *MemoryMap/System*, *Address Block*, *Base Address*, and *Register Count* are all provided in the register specification file.
- *Assertions* are in the order of “Total Asserts” - “Proven Asserts” - “Falsified Asserts” and “Unknown Asserts”.
- *Covers* are in the order of “Total Covers” - “Covered Covers” - “Uncoverable Covers” and “Unknown Covers”.

Under the *Register Summary* will consist of a *Register List* with a table of identified registers.

3.2 Resolving Errors

To see the register where this fault is resulting, we go to the column in between *Status* and *Address Block* within the *Registers* tab and click on the “Plus” icon.

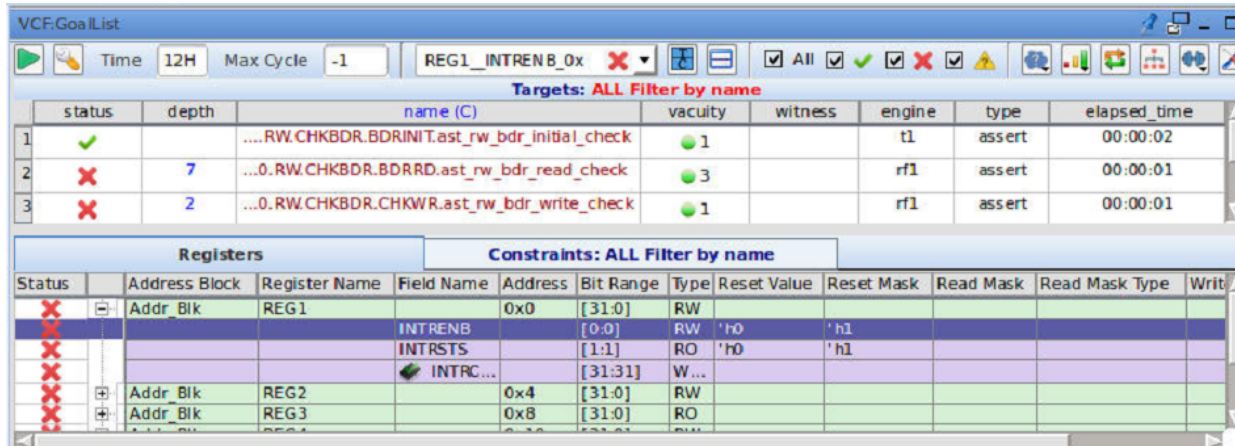


Figure 3.7. Under “Registers”, click on “Plus” icon.

We will then be given the field names within the address block that is failing in our design.

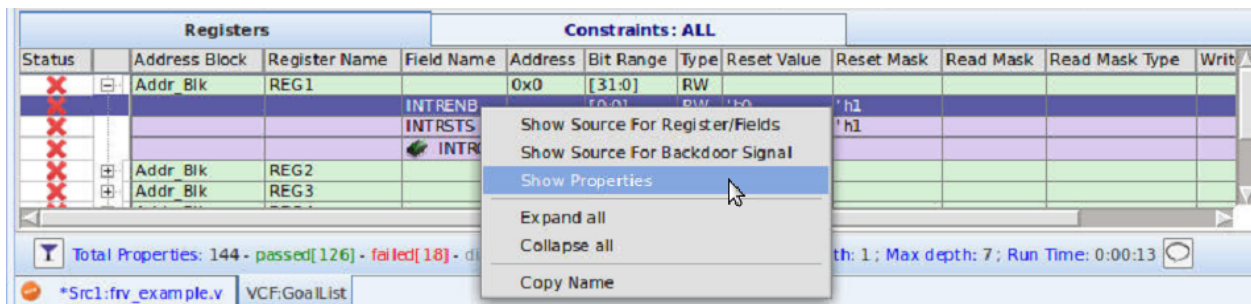


Figure 3.8. Field names under the address block that is causing failures.

Right click on the “Field Name” of choice that failed FRV and click “Show Properties”. This will show/filter all targets that are associated with the field name you selected (shown in Figure 3.7).

Double click on a failed status icon associated with the filtered *Targets*. This will generate a waveform tab.

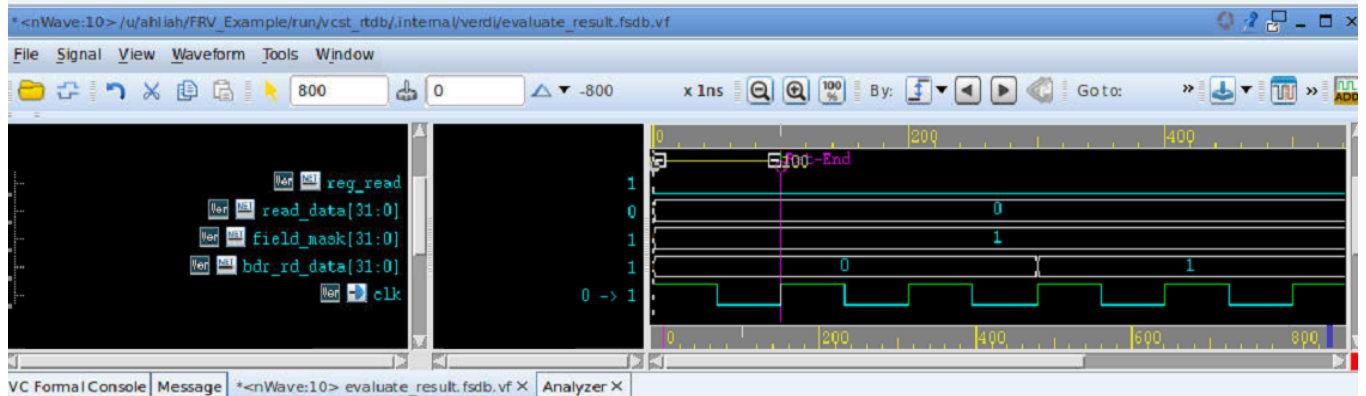


Figure 3.9. Waveform window with general traces provided by FPV.

General traces provided by FPV:

- `reg_read` - Goes “high” to indicate register read is complete.
- `read_data` - Indicates read data driven by your design.
- `field_mask` - Indicates the data bits that correspond to the register field that is checked
- `bdr_rd_data` - Indicates the expected data within this register field (unbolded part will be different name for everybody). If this trace shows a different assertion than `reg_data`, it will fail.

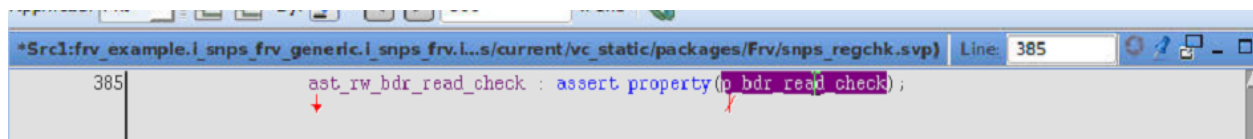


Figure 3.10: Source of failure corresponding with property being investigated.

Double click on a property name and it will take you to the source within design (shown in Figure 3.10). You can then begin back-tracing. This will eventually take you to the line where the signal is being driven and you can identify whether or not this logic is that to be expected; revealing the root-cause. This will follow similar tracing techniques laid out in other VC Formal tutorials.

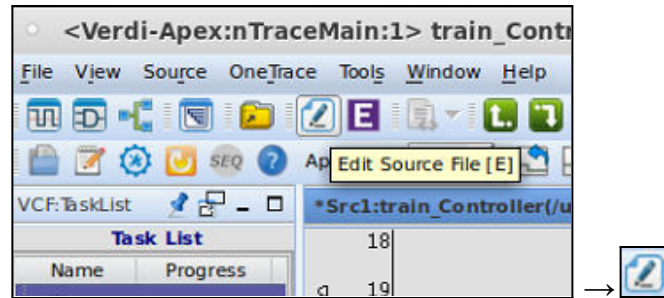


Figure 3.11. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes.

Next, to complete our changes, follow the steps below to restart VC Formal.

3.3 Restarting VC Formal

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

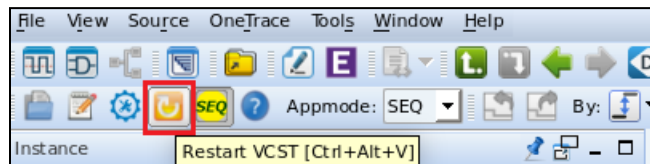


Figure 4.1. Location of the restart button in VC Formal window.

After the application and TCL file is loaded, click the green play button and we should see no errors, every status within the GoalList and Registers windows embodying the green checkmark.

Appendix

Parameter Name	Description	Default Value
ADDR_WIDTH	Address bit width.	32
DATA_WIDTH	Data bit width.	32
RSVD_MODE	Specify how to check reserved and/or unmapped registers and/or fields • 0: no check when ACCESS_TYPE==ACS_RSVD . • 1: 0 should be read when ACCESS_TYPE==ACS_RSVD and CHECK_INIT==0 . • 2: 1 should be read when ACCESS_TYPE==ACS_RSVD and CHECK_INIT==0.	1
COMPOSITE	Specify type of assertions per field for initial value and after write • 1: Generate one common assertion (better convergence). • 0: Generate two individual assertions.	1
BUS_LATENCY	Effective only in generic wrapper: Latency from read enable signal to read data valid on bus.	1
WR_LATENCY	Write Latency from bus write data propagates to internal field.	1
RD_LATENCY	Read Latency from internal field to bus read data.	1
COVER_TYPE	When COVER_WRITE==1 • 3'bx1: cover write between write to read window • 3'bx1x: cover write before first write. • 3'b1xx: cover write before first read. When COVER_READ==1 • 3'bx1: cover read between write to read window • 3'bx1x: cover read before first write. • 3'b1xx: cover read before first read.	3'h7
COVER_WRITE	Specify enable or disable write access related cover properties • 0: Disable all write access related cover properties. • 1: Enable all write access related cover properties.	1
COVER_READ	Specify enable or disable read access related cover properties • 0: Disable all read access related cover properties. • 1: Enable all read access related cover properties.	1

OUTADDR_EN	Specify enable or disable out of address range access related cover properties • 0: Disable out of address range access related cover properties. • 1: Enable out of address range access related cover properties.	1
LATENCY_MD	Specify Latency handling • 0: RD_LATENCY parameters are used only in backdoor checks. • 1: RD_LATENCY parameters are used in all checks.	0
OUTABLK_MD	Specify how to check out of address range accesses • 0: check read data should be 0 for out of addressBlock range. • 1: check read data should be 1 for out of addressBlock range. • other: no check for out of addressBlock range.	2
SNPS_AS_ABS	Synopsys Non-Deterministic Abstractions • 0: check all possible scenarios. • 1: check only specific scenario in each step.	0

Table 1.1. FRV App Checker Parameters. Names, default values, and descriptions of all offered.