VC Formal Lab
Formal Register Verification (FRV) App Setup and Standard Usage

Learning Objectives

In this VC Formal lab, you will use a generic bus controller example to learn to do the following:

- Set up and compile the design
- Load Register Verification checks into VC Formal
- Set up clocks and resets
- Establish initial state for formal
- Run FRV checks
- Debug failures

Familiarity with the SystemVerilog Assertion (SVA) language and knowledge of basic formal verification concepts are required for this lab.

Lab Duration:
30 minutes

Files Location

All files for this VC Formal lab are in directory:
$VC_STATIC_HOME/doc/vcst/examples/FRV/

Directory Structure	
FRV	Lab main directory
README_VCFormal_FRV.pdf	Lab instructions
design/	rtl/ : Verilog RTL code of the Device Under Test (DUT) xml/ : Register definition in XML format ralf/ : Register definition in RALF format
sva/	Formal testbench files
run/	Run directory
solution/	Solution directory

Resources

The following resources are available for in-depth guidance regarding VC Formal usage, commands, and variables.

VC Formal User Guide:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/VC_Formal_UG.pdf

VC Formal Apps Quick References Guides:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/

VC Formal Apps Tcl Templates:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/vcf_tcl_templates/

Prepare your Environment

1. Set environment variable pointing to your VC Formal installation directory:

```
%setenv VC_STATIC_HOME /tools/synopsys/vcstatic
```

2. Add path $VC_STATIC_HOME/bin to the PATH environment variable.

3. Change your working directory to FRV/run:

```
%cd FRV/run
```

Now you are ready to begin the lab.

Create a run.tcl Setup File

VC Formal has a Tcl-based command interface. It is common to start with a Tcl file to set up and compile a design. In this step, you will create a VC Formal Tcl file for the DUT, a bus controller, used in this lab.

The DUT files and file list are located under FRV/design.

4. Open file run.tcl (any arbitrary name is ok to use) using any text editor:

```
%vi run.tcl
```

5. Add command to enable FRV App mode (default when starting VC Formal):

```
set_fml_appmode FRV
```

6. Try using 6.a or 6.b below for your lab, not both.

   a. If using IPXACT, enter the following commands to load the register specification

```
frv_load -ipxact $SRC_DIR/xml/axi4lite_dmac.xml -auto_load
```

   b. If using RALF, enter the following commands to load the register specification

```
frv_load -ral $SRC_DIR/ralf/axi4lite_dmac.ralf \
          -top axi4lite_dmac_ralf -auto_load
```

7. Add command to compile DUT and SVA properties:

```
read_file -top axi4lite_dmac -format sverilog -sva \
  -vcs "-f $SOL_DIR/filelist.f $SVA_DIR/bind_frv.sv"
```

8. Enter clock definition

```
create_clock CLK -period 100
```

9. Enter reset definition and initialize

```
create_reset RSTN -sense low

sim_run -stable
sim_save_reset
```

10. Save run.tcl file and exit editor

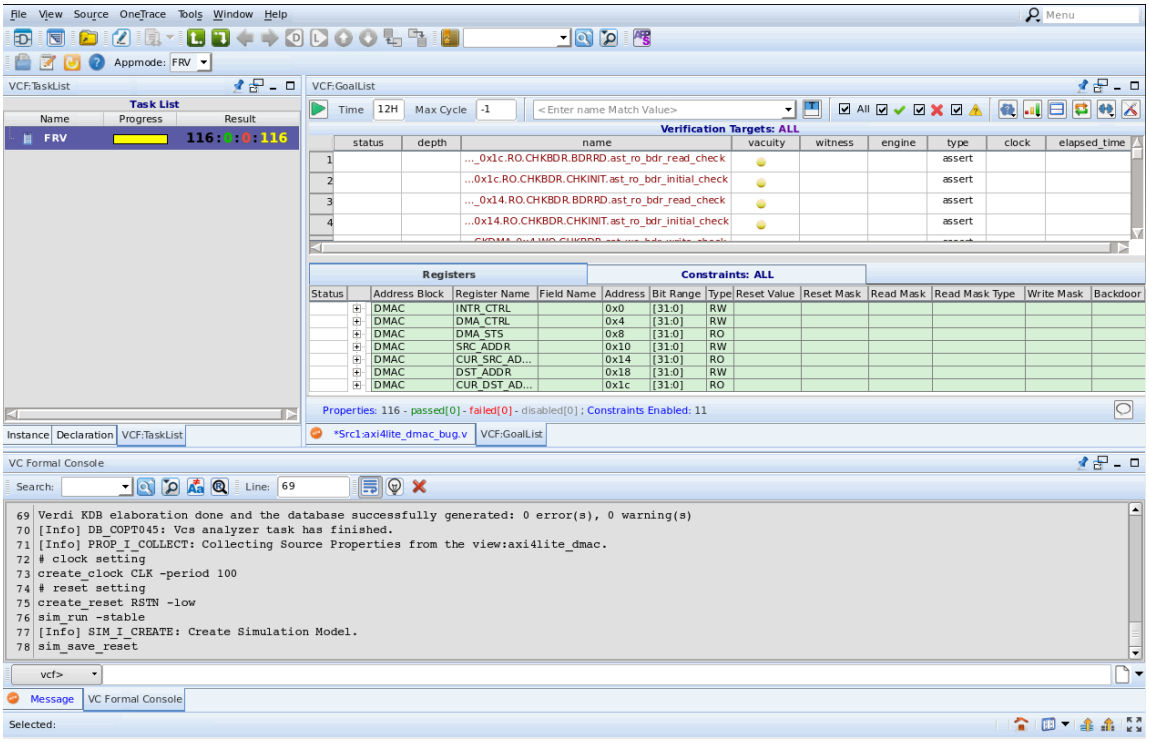
VC Formal can be run in three modes: interactive Verdi GUI mode, interactive without Verdi GUI using shell mode, and non-interactive batch mode. Verdi GUI mode is generally recommended for FRV.

Start VC Formal FRV in Verdi GUI Mode

11. Start the tool in Verdi GUI mode

```
%vcf -f run.tcl -verdi
```

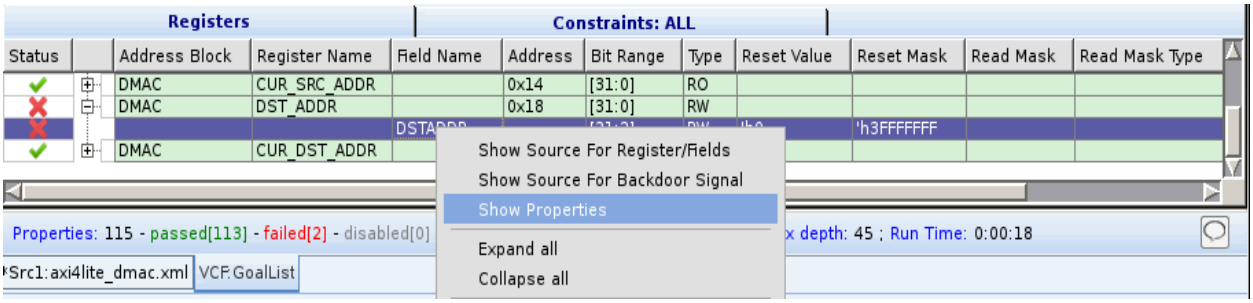
The GUI starts in VC Formal, with icons, tables, tabs, and windows tailored for FRV. The App mode is set to FPV by default. When the FRV app mode variable is set, it will start the GUI in FRV mode.



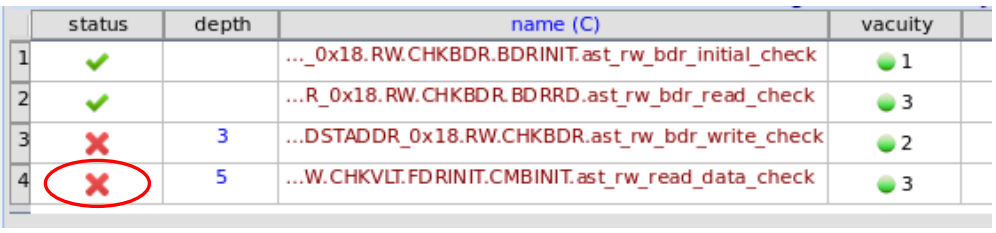
12. Start property checking by clicking on  run. This may take a few minutes to run.

Debug Failures

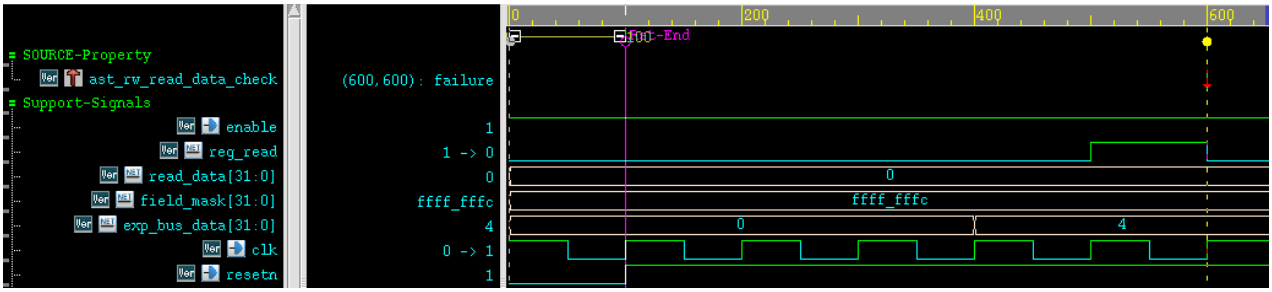
13. Click on icon of the failing address block to view the field name associated
14. Right-click on the Field name that is failing and select “Show Properties”, OR double-click on the red-cross in the Status column



15. This will show properties associated with the register. In the picture below, there are 4 properties associated with it, of which 2 are failing.



16. The failures can be debugged in the same way as FPV. Double-click on the red-cross in the Status column for one of failures; the trace should come up.



- ‘reg_read’ shows a register read occurring
- ‘read_data’ shows read data from design
- ‘field_mask’ shows which bits correspond to the register field being checked
- ‘exp_bud_data’ shows expected read data for this register field
- This trace shows read data is expected to be 4, but design drives 0

17. Double click on property name to go to the source

```
*Src1:axi4lite_dmac.i_snps_frv_generic.i_snps_frv.i_axi4lite_dmac....ta_check(/slowfs/vgfvcae1/hiroshin/frvtest/fr
363                 (* snps_frv_property *)
364                 ast_rw_read_data_check : assert property(p_read_data_check);
365             end : CMBINIT
```

```
l:axi4lite_dmac.i_snps_frv_generic.i_snps_frv.i_axi4lite_dmac....ta_check(/slowfs/vgfvcae1/h
236   property p_read_data_check;
237      @(posedge clk) disable iff (!resetn)
                 f                   1
238        (enable && reg_read)
            1          t
239        |-> ((read_data & field_mask) == (exp_bus_data & field_mask));
                 0            fffffffc            4                fffffffc
240   endproperty
```

18. Start back-tracing from ‘read_data’ signal, and repeat back-trace

```
*Src1:axi4lite_dmac(/slowfs/vgfvcae1/hiroshin/frvtest/FRV_u
115   assign reg_rd   = RGCSEL & ~RGWRITE;
               0         0         0
116   assign decerr   = (RGADDR>='h20);
               0          18
117   assign RGBUSY   = s_active;
               0         0
118   assign RGRDATA  = o_rdata;
               0         0
```

17. Verdi will take you to the line where this signal is being driven. This register field is for destination address. Double-click ‘dst_addr’ to see logic driving the signal

```
Src1:axi4lite_dmac(/slowfs/vgfvcae1/hiroshin/frvtest/FRV_updated/design/rtl/axi4lite_dmac_bug.v)
124         if (reg_rd) begin
                0
125            case (RGADDR)
                   18
126              REG_INTR : o_rdata <= {{(DATA_WIDTH-2){1'b0}}, o_intr, intr_en};
                            0                                  0       0
127              REG_CTRL : o_rdata <= {s_active, {(DATA_WIDTH-TRANS_LEN-1){1'b0}}, set_length};
                            0          0                                            0
128              REG_STTS : o_rdata <= {s_active, {(DATA_WIDTH-TRANS_LEN-1){1'b0}}, dst_length};
                            0          0                                            0
129              REG_SRCA : o_rdata <= {src_addr, 2'b0};
                            0          0
130              REG_CSRC : o_rdata <= {cur_srca, 2'b0};
                            0          0
131              REG_DSTA : o_rdata <= {dst_addr, 2'b0};
                            0          0
132              REG_CDST : o_rdata <= {cur_dsta, 2'b0};
                            0          0
```

18. Back-trace will take you to the root-cause of the bug

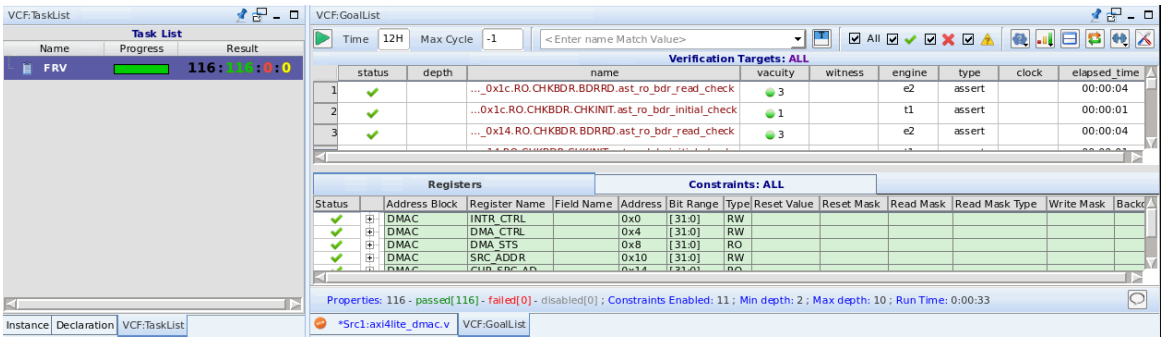
```
*Src1:axi4lite_dmac(/slowfs/vgfvcae1/hiroshin/frvtest/FRV_updated/design/rtl/axi4lite_dmac_bug.v)
158              end
159              REG_SRCA : src_addr <= RGWDATA[XADDR_WIDTH-1:2];
                            0          1
160              //REG_DSTA : dst_addr <= RGWDATA[XADDR_WIDTH-1:2];
161              REG_DSTA : dst_addr <= {RGWDATA[XADDR_WIDTH-1:3], 1'b0}; // bug
                            0          0
162            endcase // case (ADDR)
```

19. The failures are caused by bugs in the RTL. The fixed RTL is available in the solution’s directory. Go to solution directory and execute

```
%vcf -f run_ipxact.tcl -verdi (or run_ralf.tcl)
```

20. Confirm all assertions are proven with fixed RTL

a. IPXACT Solution



b. RALF Solution

The RALF solution has 2 assertions less than the IPXACT solution because of differences in IPXACT and RALF formats. In RALF, there is no way of specifying “write-only + 1-to-clear”. So, we need to disable the equivalent FRV backdoor read check, “ast_rw_bdr_read_check” for this example. If you don’t, you will see it as a CEX which is anticipated.