VC Formal Lab
Counter Abstraction

Learning Objectives

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

- Perform counter abstraction
- Find a solution to converge the properties

Lab Duration: 40 minutes

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

Files Location

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

Directory Structure	
FPV/Abstraction/Counter_Abstraction	Lab main directory
README_VCFormal_Counter_Abstraction.pdf	Lab instructions
design/	Verilog RTL code of the Device Under Test (DUT)
sva/	SVA properties to check functionality of the DUT
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 FPV/Abstraction/Counter_Abstraction/run:

```
%cd FPV/Abstraction/Counter_Abstraction/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 packet controller, used in this lab.

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

```
%vi run.tcl
```

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

```
set_fml_appmode FPV
```

6. Specify Formal TB top level module name as Tcl variable:

```
set design packet_ctrl
```

7. Add command to compile DUT and SVA properties:

   The DUT file is located under directory FPV/Abstraction/Counter_Abstraction/design. The assertion and bind files are located under directory FPV/Abstraction/Counter_Abstraction/sva.

```
read_file -sva -format sverilog -top packet_ctrl -vcs
 {../design/packet_ctrl.sv ../sva/packet_ctrl_checker.sv}
```

8. Save run.tcl file and exit editor.

Packet Controller Design Implementation

- This is simple packet controller design.
- After initialization, it takes an input data and creates small data packet and sends them serially at the output port.
- There are 3 interfaces:
  - Clock/reset interface
  - Link interface
  - Packet interface
- Link at the link interface is established based on link_req/link_ack handshake and an internal 16 bit initialization counter.
- After reset, once the initialization counter value reaches 65392, link_req is sent by the packet controller. Link is established once the packet controller receives link_ack.
- link_state transits from Initial -> Busy -> Ready -> Up
  - 2'b00 : Initial
  - 2'b01 : Busy - waiting stabled
  - 2'b10 : Ready - start req-ack handshake (takes 65392 cycles after reset)
  - 2'b11 : Up - ready to send packet
- Once the link is UP and controller receives a pkt_req, it sends pkt_ack and completes the handshake.
- Once the packet interface handshake is done, the controller takes the input data , it creates small data packets and starts sending them at the output.
- pkt_sop is high for the first data packet at the output.
- pkt_out is high for the last data packet at the output.

```
module packet_ctrl
  #(parameter PKT_WIDTH  = 8,
    parameter PKT_LENGTH = 4,
    parameter DATA_WIDTH = (PKT_WIDTH*PKT_LENGTH))

   (input  logic clk,
    input  logic rst,

    output logic [1:0] link_state,
    output logic link_req,
    input  logic link_ack,
    input  logic link_rcv,
    input  logic link_down,

    input  logic pkt_req,
    output logic pkt_ack,
    input  logic [DATA_WIDTH-1:0] data,
    output logic [PKT_WIDTH-1:0]  pkt,
    output logic pkt_sop,
    output logic pkt_eop);
```

Formal Testbench for ALU

In the checker file we have assertions for the following functionalities:

- pkt_eop should go high once all the data packets are sent at the output
- link_state should be 2'b00 after reset
- link_state should transition from 2'b00 to 2'b01
- link_state should transition from 2'b01 to 2'b10
- link_state should transition from 2'b10 to 2'b11

With no abstraction it takes ~50mins to prove all 5 assertions.

Lab Solution

Counter Abstraction Methodology:

- Critical Values
  - Model only important values: 0, 1, max, etc.
  - Requires knowledge of which values are critical.

- Random Increment/Decrement
  - Provide a random input that controls the increments/decrements of the counter
  - All possible counter values remain reachable.
  - Further enhance to make sure that the value always increases on an increment and always decreases on a decrement.
  - If there are multiple counters, they must be kept in sync, constrain the increment/decrement intervals to match.

- Snip the actual counter in the design and bind the abstracted counter to the design

- In this example we model the packet_ctrl.init_counts which is a 16 bit counter

- We use 2 special values based on the FSM used to establish the link in the design
  - BUSY_CNTS (this is SPVALUE1)
  - READY_CNTS (this is SPVALUE2)

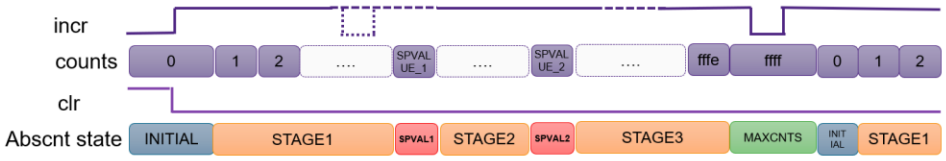
- There would be 7 states in the abstracted counter model to represent the 2 special values and transition between them. Hence it requires 3 bit register to model the FSM for 16 bit counter in our design.

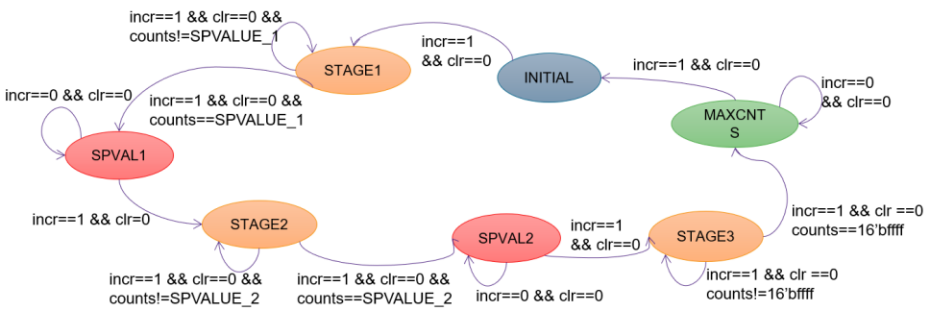
  INITIAL : counts == 0
  STAGE_1 : counts between 1 and SPVALUE_1
  SPVAL_1 : counts == SPVALUE_1
  STAGE_2 : counts between SPVALUE_1+1 and SPVALUE_2
  SPVAL_2 : counts == SPVALUE_2
  STAGE_3 : counts between SPVALUE_2+1 and maximum-1
  MAXCNTS : counts == maximum (all 1s)

- counts is initialized with 16'b0

- counts increments when incr is high and goes to 16b'0 after 16'ffff

- counts goes to 16'b0 if clr is high





In state "STAGE1", its non-deterministic when it transits to state "SPVAL1" This modeling represents any case such as it transits immediately or after long cycles. Similarly, for STAGE2 to SPVAL2 and STAGE3 to MAXCNTS.

Actual design behavior is one of the behavior among the behaviors represented with this modeling

Note: FSM transitions from STAGE* and SPVAL* to INITIAL when 'clr'==1. This is not shown in the state diagram to keep it simple.

Assumptions to model the state transition of abs counter FSM:

```
fvassume -expr { $fell(rst) |-> (cur_state == INITIAL)}
fvassume -expr { clr |=> (cur_state == INITIAL)}
fvassume -expr { (!clr && !incr) |=> $stable(cur_state)}
fvassume -expr { (cur_state == INITIAL && incr)
                 |=> (cur_state == STAGE_1)}
fvassume -expr { (cur_state == STAGE_1 && incr)
                 |=> (cur_state inside {STAGE_1, SPVAL_1})}
fvassume -expr { (cur_state == SPVAL_1 && incr)
                 |=> (cur_state == STAGE_2)}
fvassume -expr { (cur_state == STAGE_2 && incr)
                 |=> (cur_state inside {STAGE_2, SPVAL_2})}
fvassume -expr { (cur_state == SPVAL_2 && incr)
                 |=> (cur_state == STAGE_3)}
fvassume -expr { (cur_state == STAGE_3 && incr)
                 |=> (cur_state inside {STAGE_3, MAXCNTS})}
fvassume -expr { (cur_state == MAXCNTS && incr)
                 |=> (cur_state == INITIAL)}
```

Assumptions to model the count value at a given state of abs counter FSM:

```
fvassume -expr {(cur_state == INITIAL) |-> (counts == 'd0)}
fvassume -expr {(cur_state == STAGE_1)
                 |-> (counts > 'd0 && counts < SPVALUE_1)}
fvassume -expr {(cur_state == SPVAL_1) |-> (counts == SPVALUE_1)}
fvassume -expr {(cur_state == STAGE_2)
                 |-> (counts > SPVALUE_1 && counts < SPVALUE_2)}
fvassume -expr {(cur_state == SPVAL_2) |-> (counts == SPVALUE_2)}
fvassume -expr {(cur_state == STAGE_3) |-> (counts > SPVALUE_2
                 && counts < {BIT_WIDTH{1'b1}})}
fvassume -expr {(cur_state == MAXCNTS)
                 |-> (counts == {BIT_WIDTH{1'b1}})}
```

Summary of Tcl files in solution directory:

1. solution/run_snip.tcl: snip_driver the counter
    - Takes 5s to converge but gives false failure

2. solution/abs_counter.tcl: counter abstraction
    - Takes 2mins to prove all 5 assertions

3. solution/abs_validate.tcl: validate counter abstraction
   prove the assumes used in abstraction as asserts