Statically Dynamic or Dynamically Static?
Exploring the power of classes and enumerations in SystemVerilog Assertions for reusability and scalability

Sachin Scaria - Intel
Sreenu Yerabolu - Intel
Don Mills – Microchip (Presenter)
Abstract

- Usage of class data type for SoC level assertions (SV)
  - Modular code development
  - Reusable code
  - Reduce human errors

- Can be extended to complex data structures
Motivation

• Reusable assertions across IPs/projects/revisions
• Minimal changes while scaling up
• Ease of plugging into the Testbench
Case Study Attributes

- SoC with IPs across interconnect fabrics
- IPs have common rules to adhere to
- IP design rules can be enhanced
- An added feature needed to be verified
All the interfaces starting with IF will be compliant to a standard protocol consisting of two links
1) Serial
2) Parallel

Assertions were required to be connected to each IF interface. This will help in enforcing the security compliance, when responses are sent from the IP.
IP Packet Handling

- IP has pre-configured key values
- Matches incoming packet/command keys
- Responds with Success/Unsuccessful response
A Sophisticated Scenario

Keys from Master 1
K1, K3, K4

Keys from Master 2
K2, K4, K5

Keys common between the masters
K4

Valid Keys to an IP

IP has to respond with a specific set of valid key in the response based on the master number

IP
Traditional Approach

- One file per interface
- More than one assertion module for compilation
  - But uses a similar templated code
  - Changes mostly in the keys defined for the IP
- Assertion modules proportional to number of IPs
  - Each IP will require one unique module
Proposed Approach

• Type over-ride using parameterized class types
• Encapsulating enum “Key” values using virtual classes
  – Allows for unique scopes for each set of keys
• Single assertion file and multiple instances
  – Configuration changes based on the instantiation
    • Thus key values for the instance are modified
Implementation Details
Sample Code - 1

virtual class `a`;
    typedef enum logic [7:0] {
        K1 = 8'h01,
        K2 = 8'h08,
        K3 = 8'h09
    } valid_keys;
endclass

Valid Keys in `a` are
    K1(01), K2(08), K3(09)

virtual class `b`;
    typedef enum logic [7:0] {
        K3 = 8'h9
    } valid_keys;
endclass

Valid Keys in `b` is
    K3(09)
Virtual Classes

- Declared outside the assertion block
- Encapsulate enumerations (keys)
- Virtual classes are used to restrict accidental access
- Class members access using "::" scope resolution operator
Sample Code - 2
Module, IP checks

Base Concepts

module ip_check #(type T1 = int, type T2 = int)
    logic [7:0] local_enum;
    string my_string;

typedef T1::valid_keys key_g1_enum_t;
typedef T2::valid_keys key_g2_enum_t;
key_g1_enum_t key_g1_enum;
key_g2_enum_t key_g2_enum;

initial begin
    local_enum = 8'h8;
    key_g1_enum = key_g1_enum_t'(local_enum);
    my_string = key_g1_enum.name() ;
    $display ("value is %s",my_string);
    local_enum = 8'h9;
    key_g2_enum = key_g2_enum_t'(local_enum);
    my_string = key_g2_enum.name() ;
    $display ("value is %s",my_string);
end
endmodule:ip_check

Instantiation

module tb_ip_check ();
    ip_check #(.T1(a), .T2(b)) dut_ip1_check () ;
endmodule

Log report

# value is K2
# value is K3

Valid Keys in b is K3(09)

Valid Keys in a are K1(01), K2(08), K3(09)
module ip_check #(type T1 = int, type T2 = int)  
  (my_vif vif);

typedef T1::valid_keys key_g1_enum_t;
typedef T2::valid_keys key_g2_enum_t;
key_g1_enum_t        key_g1_enum;
key_g2_enum_t        key_g2_enum;

//assuming vif.key is declared in the vif interface
logic valid1_key;
assign key_g1_enum = key_g1_enum_t'(vif.key);
assign valid1_key = (key_g1_enum.name () != "");

logic valid2_key;
assign key_g2_enum = key_g2_enum_t'(vif.key);
assign valid2_key = (key_g2_enum.name () != "");

sequence check_key_g1 ;
  @(posedge vif.clk)
  ((vif.cmd_start) && (vif.grp == 2'b0) && (valid1_key));
endsequence

sequence check_key_g2 ;
  @(posedge vif.clk)
  ((vif.cmd_start) && (vif.grp == 2'b1) && (valid2_key));
endsequence

// assertion for grp1
property check_grp1_cmpl;
  logic [7:0] local_tag;
  @(posedge vif.clk)
  disable iff (!vif.rst_b)
  (check_key_grp1.triggered, local_tag = vif.tag) ->
  (vif.opcode == 2'b0) && (vif.resp_tag == local_tag);
  //opcode 2'b0 -> successful operation
endproperty

// assertion for grp2
property check_grp2_cmpl;
  logic [7:0] local_tag;
  @(posedge vif.clk)
  disable iff (!vif.rst_b)
  (check_key_grp2.triggered, local_tag = vif.tag) ->
  (vif.opcode == 2'b0) && (vif.resp_tag == local_tag);
  //opcode 2'b0 -> successful operation
endproperty
endmodule:ip_check
• Would do the same code to verify “valid” Master
• Assists with coverage per the assertion
  – Cover the enums
  – Easy to maintain the code for coverage
• bind ip_check into each IP instance
  – The following syntax is not complete
    bind ip_check #( .T1(a), .T2(b) ) dut_ip1_check () ;
Tool Support

- Verified with one of the “big 3” simulators - Questa
- Enhancements are filed with remaining vendors
References

• SNUG Silicon Valley, 2016, "Thank you VGEN - Complex SoC Assertion Qualification Turnaround Time from Hours to Minutes", Sachin, Sreenu

