DESIGN & VERIFICATION DVCON 2014 CONFERENCE & EXHIBITION

SystemVerilog Interface Cookbook

Rockwell Automation

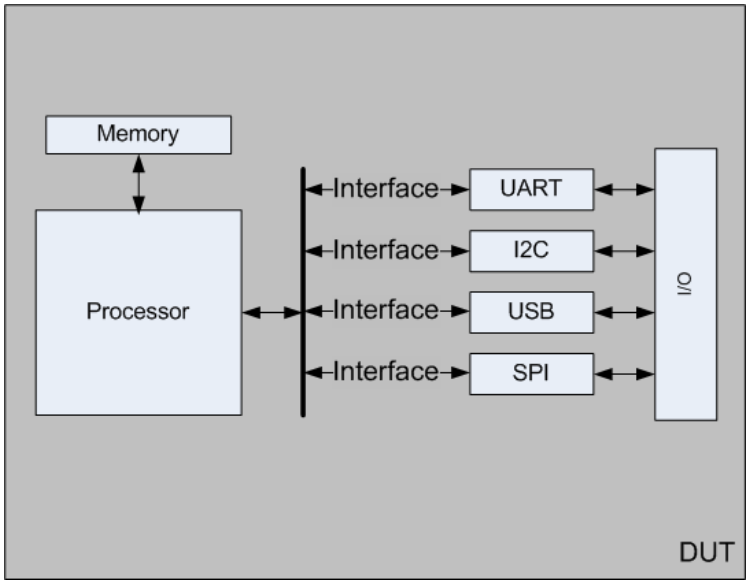
Allen-Bradley • Rockwell Software

Agenda

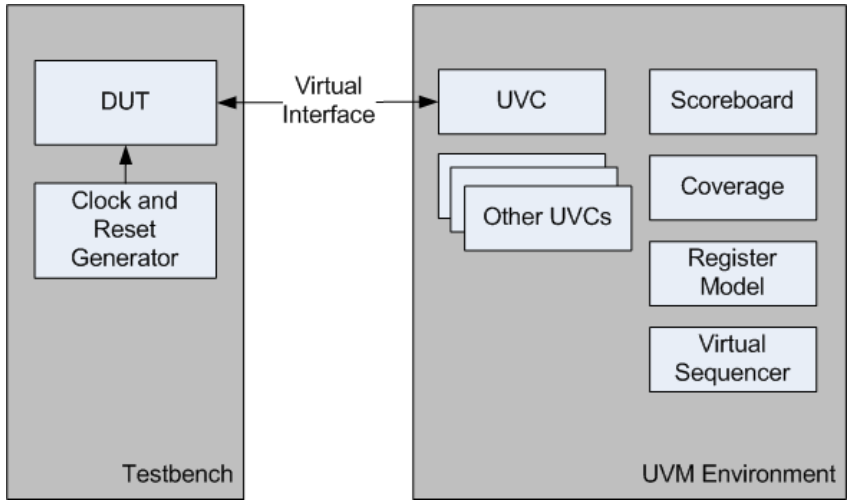
- Motivation
  - No comprehensive resource documenting most common SOC usage scenarios
- VHDL/Verilog BFM Reuse
- Parameterized VIP
- Register backdoor access
- Hardware/Software Flow Control

Using Interfaces for Basic Connections

Connect blocks within the DUT:

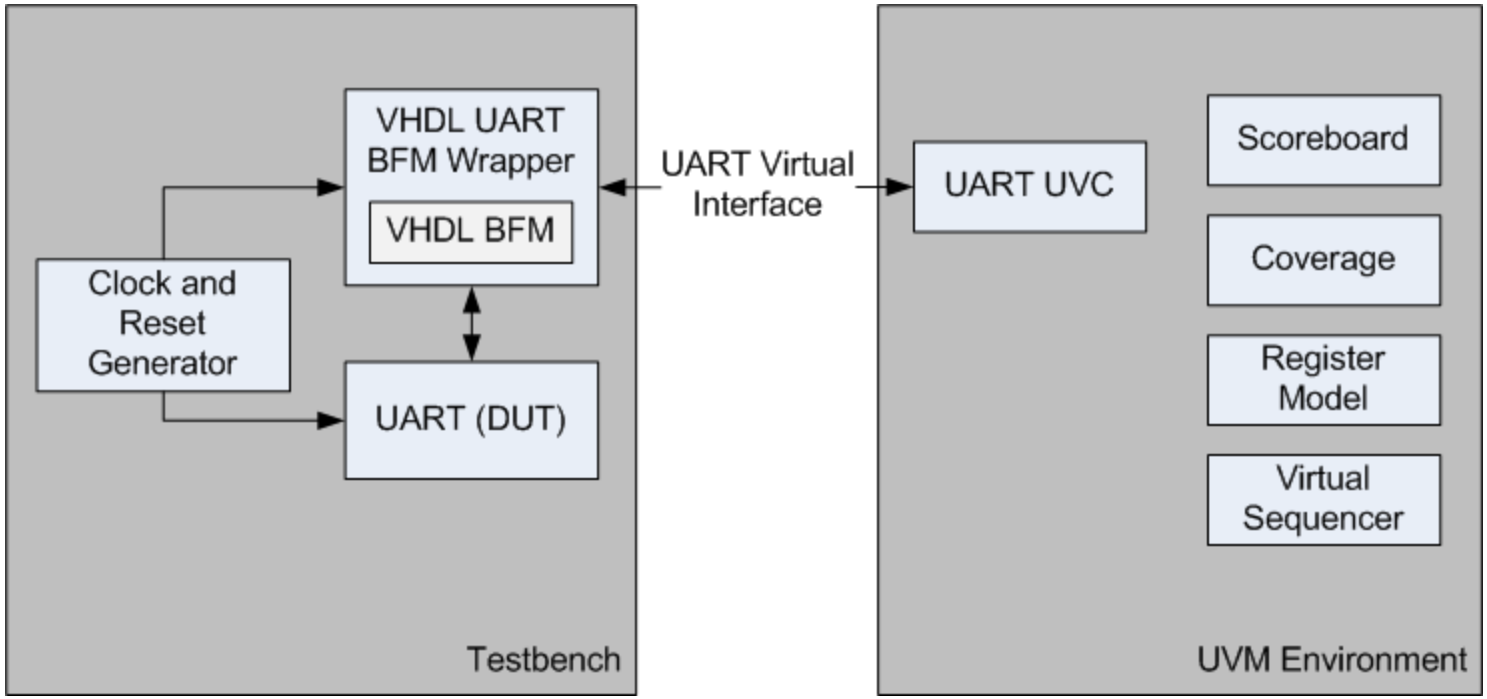


Connect the static world to the dynamic world through virtual interfaces:



Integrating a Legacy VHDL BFM into a UVM Environment

Only one possible approach:



VHDL UART BFM Wrapper has signals to interface the legacy BFM to the DUT and signals to connect to a virtual interface.

Integrating a Legacy VHDL BFM into a UVM Environment

```
entity uart_bfm_wrapper is
  port(
        -- Inputs from UVC
        write_data   : in std_logic_vector (7 downto 0);
        rw_addr      : in std_logic_vector (2 downto 0);
        r_wn         : in std_logic;
        num_clks     : natural;
        reset_start  : in std_logic;
        rw_start     : in std_logic;

        -- Outputs for DUT
        addr         : out std_logic_vector (2 downto 0);
        clk          : out std_logic;
        -- other outputs for DUT...

        -- Outputs for UVC
        read_data    : out std_logic_vector (7 downto 0);
        reset_done   : out std_logic;
        rw_done      : out std_logic
  );
end uart_bfm_wrapper;
```

BFM wrapper responds to control signals with BFM procedure calls.

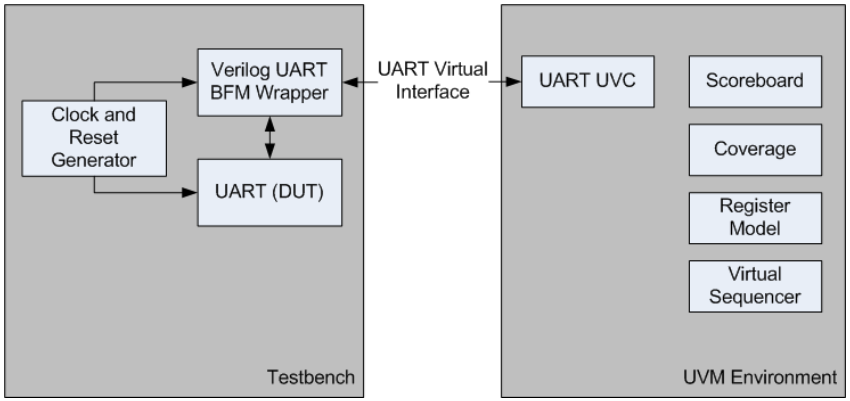
```
architecture beh of uart_bfm_wrapper is
  begin

    uart_bfm_inst : uart_bfm
      port map (
        -- Decompose records
        reg_rw.addr   => addr,
        reg_rw.cs     => cs_internal,
        reg_rw.datin  => datin,
        reg_rw.nrw    => nrw,
        -- Connect other BFM I/O...
    );

    main: process begin

      if (reset_start = '1') then
          reset_done <= '0';
          reset(num_clks => num_clks);
          wait until rising_edge(rst_internal);
          reset_done <= '1';
      elsif (rw_start = '1') then
          rw_done <= '0';
          if (r_wn = '1') then
              read(rw_addr);
          else
              write(write_data, rw_addr);
          end if;
          wait until rising_edge(cs_internal);
          rw_done <= '1';
      end if;

    end process main;
end beh;
```

Integrating a Legacy VHDL BFM into a UVM Environment

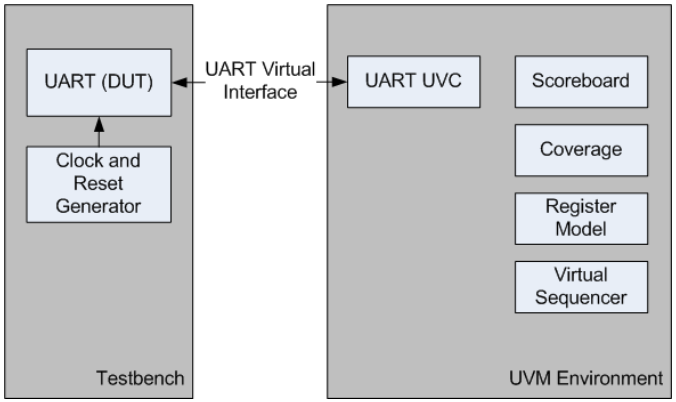
```
virtual task drive_dut();

  vif.write_data <= req_txn.write_data;
  vif.rw_addr    <= req_txn.rw_addr;
  vif.num_clks   <= req_txn.num_clks;

  if (req_txn.trans_type == RESET) begin

      vif.start_reset   <= 1'b1;
      vif.start_rw      <= 1'b0;
      @(posedge vif.reset_done);
      vif.start_reset <= 1'b0;

  end else begin // Transaction is RW

      vif.start_reset<= 1'b0;
      vif.start_rw    <= 1'b1;
      if (req_txn.trans_type == READ) begin
          vif.r_wn <= 1'b1;
      end else begin
          vif.r_wn <= 1'b0;
      end
      @(posedge vif.rw_done);
      req_txn.read_data = vif.read_data;
      vif.start_rw    <= 1'b0;

  end
endtask: drive_dut
```

uvm_driver drives the control signals contained in the SystemVerilog interface

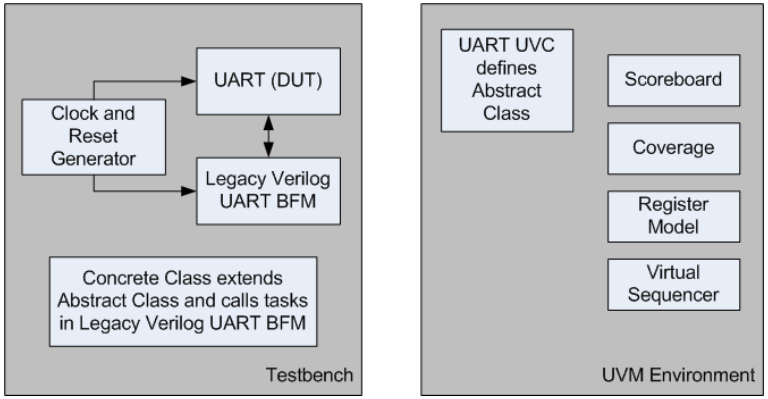
Integrating a Legacy Verilog BFM into a UVM Environment



Option 1: Just like VHDL



Option 2:
Convert the BFM into a SystemVerilog interface



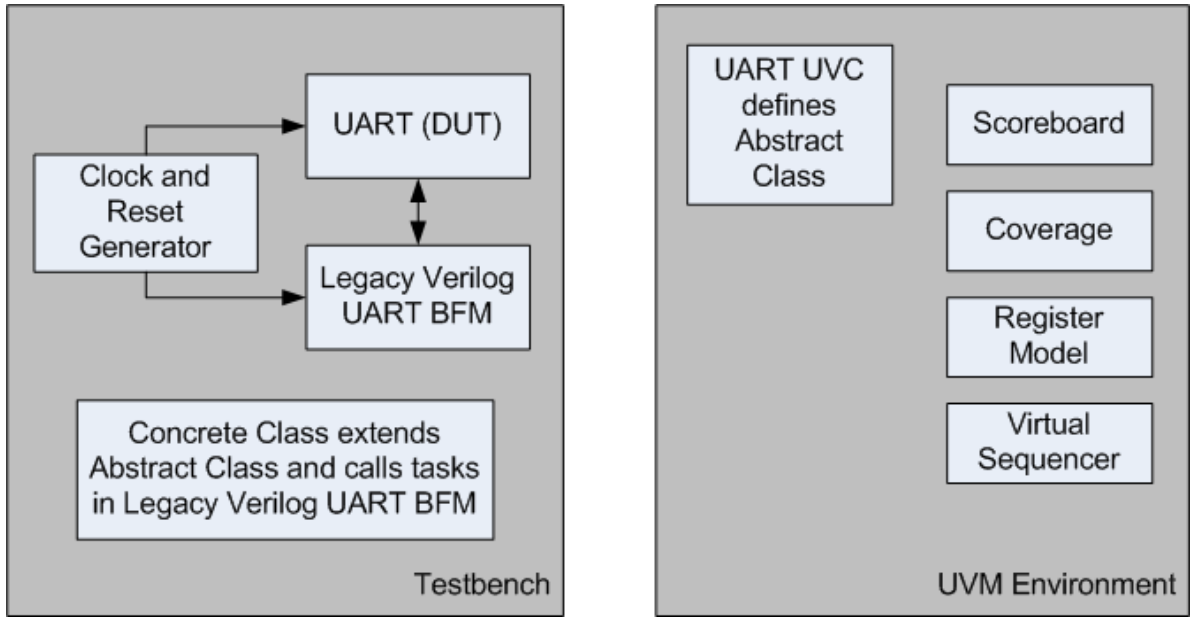
Option 3:
Use abstract and concrete classes

Integrating a Legacy Verilog BFM: Using Abstract & Concrete Classes

- Base class*
  - Prototype for the subclasses
- Abstract Class*
  - A base class that cannot be instantiated
  - Keyword *virtual*
- Concrete Class*
  - Can be instantiated
  - Must override any virtual methods of the abstract class from which it derives

* IEEE 1800-2012 SystemVerilog Language Reference Manual

Integrating a Legacy Verilog BFM: Using Abstract &Concrete Classes



Key Concept:
There is NO Virtual Interface

Integrating a Legacy Verilog BFM: Using Abstract &Concrete Classes

virtual makes this class abstract

pure virtual task must be implemented by the derived class

```
virtual class uart_driver extends uvm_driver #(uart_seq_item, uart_seq_item);

  // factory registration macro
  `uvm_component_utils(uart_driver)

  // internal components
  uart_seq_item    req_txn;

  function new (string name = "uart_driver", uvm_component parent = null);
    super.new(name,parent);
  endfunction: new

  function void build_phase (uvm_phase phase);
    super.build_phase(phase);
  endfunction: build_phase

  pure virtual task uart_write(logic [7:0] bit_data, logic [2:0] addr_w);
  pure virtual task uart_read(logic [2:0] addr_r, output logic [7:0] read_data);
  pure virtual task uart_reset(integer  num_clocks);

endclass: uart_driver
```

Integrating a Legacy Verilog BFM: Using Abstract &Concrete Classes

```
module uart_tb;

  // Instantiate the BFM and the DUT

  // Declare the concrete driver class

  initial begin
    uart_driver::type_id::set_type_override(uart_driver_concrete::get_type());
    run_test();
  end
endmodule: uart_tb
```

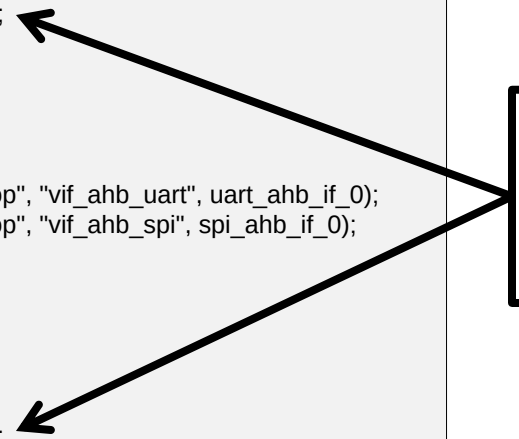
Abstract class is overridden at runtime

```
class uart_driver_concrete extends uart_driver;

  `uvm_component_utils(uart_driver_concrete)

  task uart_write(input logic [7:0] bit_data, input logic [2:0] addr_w);
      uart_bfm_inst.uart_write(bit_data, addr_w);
  endtask: uart_write

  task uart_read(input logic [2:0] addr_r, output logic [7:0] read_data);
      uart_bfm_inst.uart_read(addr_r);
      read_data = datout;
  endtask: uart_read

  task uart_reset(input integer  num_clocks);
      uart_bfm_inst.uart_reset(num_clocks);
  endtask: uart_reset

  task run_phase(uvm_phase phase);
      forever begin
          seq_item_port.get_next_item(req_txn);
          @(posedge clk_in) #1;
          case(req_txn.trans_type)
              RESET: uart_reset(req_txn.num_clks);
              READ: uart_read(req_txn.rw_addr, req_txn.read_data);
              WRITE: uart_write(req_txn.write_data, req_txn.rw_addr);
          endcase
          seq_item_port.item_done();
      end
  endtask: run_phase
endclass: uart_driver_concrete
```

Concrete class has ability to call the BFM tasks

Parameter Propagation

```
interface ahb_vip_if  #(parameter  NUM_MASTERS,
                                   NUM_SLAVES,
                                   ADDRESS_WIDTH,
                                   WDATA_WIDTH,
                                   RDATA_WIDTH)
                        (input clk, input reset);
  logic [ADDRESS_WIDTH-1:0] haddr;
  logic [WDATA_WIDTH-1:0]   hwdata;
  logic [RDATA_WIDTH-1:0]   hrdata;
  ...
endinterface: ahb_vip_if


module top_tb ();
    typedef virtual ahb_vip_if #(1,10,32,32,32) vif_ahb_t;

    vif_ahb_t  uart_ahb_if_0();
    vif_ahb_t    spi_ahb_if_0();

    initial begin
      uvm_config_db #(vif_ahb_t)::set(null, "uvm_test_top", "vif_ahb_uart", uart_ahb_if_0);
      uvm_config_db #(vif_ahb_t)::set(null, "uvm_test_top", "vif_ahb_spi", spi_ahb_if_0);
      run_test();
    end
endmodule


virtual class test_base extends uvm_test;
    typedef virtual ahb_vip_if #(1,10,32,32,32) vif_ahb_t;

    virtual function void build_phase(uvm_phase phase);
      ...
      uvm_config_db #(vif_ahb_t)::get(this, "", "vif_ahb_uart", m_ahb_master_agent_cfg[0].vif)
      uvm_config_db #(vif_ahb_t)::get(this, "", "vif_ahb_spi", m_ahb_master_agent_cfg[1].vif))
  endfunction: build_phase
endclass: test_base
```



Knowledge of the parameters is needed in many places

Parameter Propagation: Using Abstract & Concrete Classes

```
class uart_driver#(integer ID=0)
    extends uvm_driver #(uart_seq_item, uart_seq_item);

    uart_if_base vif;

    function void build();
        uvm_object tmp;

        super.build();

        get_config_object({"uart_if_",ID},tmp,0);
        $cast(vif,tmp);
    endfunction

endclass: uart_driver
```

```
interface uart_uvc_if
    #(DATA_WIDTH=8, ADDR_WIDTH=3)
    (input clk, input reset);

    logic  [DATA_WIDTH-1:0]   datout;
    logic  [ADDR_WIDTH-1:0]  addr;
    logic  [DATA_WIDTH-1:0]   datin;

    class uart_if_concrete extends uart_if_base;
        task uart_write;
            input integer bit_data;
            input integer addr_w;
        begin
            addr = addr_w;
            nrw = # (CLK_PRD) 1;
            datin = bit_data;
            cs = #(CLK_PRD) 0;
            cs = #(5*CLK_PRD) 1;
            nrw = #(CLK_PRD) 0;
            # CLK_PRD;
        end
        endtask
    endclass

    uart_if_concrete concrete_if;

    function uart_if_base create_concrete_if(string name);
        concrete_if = new(name);
        return concrete_if;
    endfunction

endinterface: uart_uvc_if
```

Parameter Propagation: Using Abstract & Concrete Classes

Knowledge of the parameters is needed in only one location

```
module uart_tb;

  uart_uvc_if #(  .DATA_WIDTH(32),
                  .ADDR_WIDTH(4)) uart_if_0(.clk(clk_in), .reset(rst_in));
  uart_uvc_if #(  .DATA_WIDTH(16),
                  .ADDR_WIDTH(2)) uart_if_1(.clk(clk_in), .reset(rst_in));
  dut   dut_inst ( // DUT port connections...);

  initial begin
     set_config_object("*","uart_if_0",uart_if_0.create_concrete_if("uart_if_0"),0);
     set_config_object("*","uart_if_1",uart_if_1.create_concrete_if("uart_if_1"),0);
     run_test();
  end
endmodule: uart_tb
```

Parameter Propagation: Using Abstract & Concrete Classes

```
class uart_driver#(integer ID=0)
    extends uvm_driver #(uart_seq_item, uart_seq_item);

    uart_if_base vif;

    function void build();
        uvm_object tmp;

        super.build();

        get_config_object({"uart_if_",ID},tmp,0);
        $cast(vif,tmp);
    endfunction

endclass: uart_driver
```

```
interface uart_uvc_if
    #(DATA_WIDTH=8, ADDR_WIDTH=3)
    (input clk, input reset);

    logic  [DATA_WIDTH-1:0]   datout;
    logic  [ADDR_WIDTH-1:0]  addr;
    logic  [DATA_WIDTH-1:0]   datin;

    class uart_if_concrete extends uart_if_base;
        task uart_write;
            input integer bit_data;
            input integer addr_w;
        begin
            addr = addr_w;
            nrw = # (CLK_PRD) 1;
            datin = bit_data;
            cs = #(CLK_PRD) 0;
            cs = #(5*CLK_PRD) 1;
            nrw = #(CLK_PRD) 0;
            # CLK_PRD;
        end
        endtask
    endclass

    uart_if_concrete concrete_if;

    function uart_if_base create_concrete_if(string name);
        concrete_if = new(name);
        return concrete_if;
    endfunction

endinterface: uart_uvc_if
```

Backdoor Access

Bind module
For VHDL DUT

VHDL Backdoor interface
read/write tasks

```
module vhdl_dut_tb();
  top_wrapper      dut_wrapper ();

  bind dut_wrapper.dut.u_top.u_regs vhdl_backdoor_module
        vhdl_backdoor_module_i (.*, .status(status_rd_ff));

endmodule

interface vhdl_reg_backdoor_if(input [31:0] status);
  import uvm_pkg;

  task backdoor_read (uvm_reg_item rw);
    rw.value[0]=status;
  endtask

  task backdoor_write (uvm_reg_item rw);
   `uvm_error("status_reg_backdoor", "Backdoor write to VHDL not
allowed!")
  endtask

endinterface : vhdl_reg_backdoor_if

module vhdl_backdoor_module(input wire [31:0] status );
   import uvm_pkg::*;

   vhdl_reg_backdoor_if vh_status_reg_if(.*);

   initial begin
      uvm_config_db #(virtual vhdl_reg_backdoor_if)::set(null,"*",
           "vif_vh_reg_backdoor", vh_status_reg_if);
   end

endmodule : vhdl_backdoor_module
```

Backdoor Access

User defined backdoor

```
class status_reg_backdoor extends uvm_status_reg_backdoor;
      `uvm_object_utils(status_reg_backdoor)
  // Task to write via backdoor

  virtual status_reg_backdoor_if status_reg_backdoor_vif;

  virtual task read(uvm_reg_item rw);
    do_pre_read(rw);
    status_reg_backdoor_vif.backdoor_read(rw);
    rw.status = UVM_IS_OK;
    do_post_read(rw);
  endtask

  virtual task write(uvm_reg_item rw);
    do_pre_write(rw);
    status_reg_backdoor_vif.backdoor_write(rw);
    rw.status = UVM_IS_OK;
    do_post_write(rw);
  endtask
endclass : status_reg_backdoor
```

SystemVerilog Backdoor interface

```
interface reg_backdoor_if( );
  import uvm_pkg::*;
  task backdoor_read (uvm_reg_item rw);
   rw.value[0]=top_tb.dut_wrapper.dut.u_regs.status_rd_data_ff;
  endtask

  task backdoor_write (uvm_reg_item rw);
    top_tb.dut_wrapper.dut.u_regs.status_rd_data_ff= rw.value[0];
  endtask

endinterface : reg_backdoor_if
```

Hardware/Software Flow Control with Interfaces

User defined backdoor

```
interface firmware_if();
  task backdoor_write (logic [31:0] address, logic[31:0] data, logic [2:0] fw_cmd);
      top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[66:64] = fw_cmd;
      top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[63:32] = data;
      top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[31:0]   = address;
  endtask
  task backdoor_read (logic [31:0] address, logic[31:0] data, logic [2:0] fw_cmd);
     fw_cmd  = top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[66:64];
     data       = top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[63:32];
     address = top_tb.dut_wrapper.dut.u_top.u_proc_sys.u_mem.u_mailbox[31:0];
  endtask
endinterface: firmware_if

class firmware_driver extends uvm_driver #(firmware_item, firmware_item);
  virtual firmware_if       vif_fw;
  virtual event_if          vif_event;
    task run_phase(uvm_phase phase);
      forever begin
        seq_item_port.get_next_item(req_txn);
        case(req_txn.trans_type)
          READ: vif_fw.backdoor_read(req_txn.address, req_txn.data,req_txn.fw_cmd);
          WRITE: vif_fw.backdoor_write(req_txn.address, req_txn.data,req_tx.fw_cmd);
        endcase
        seq_item_port.item_done();
      end
    endtask: run_phase
endclass: firmware_driver
```

Hardware/Software Flow Control with Interfaces

Monitor triggers events

Sequences can use events for execution control

```
class firmware_monitor extends uvm_monitor;
 firmware_item           mon_txn, t;
 virtual event_if        event_vif;

 task monitor_dut();
   forever begin
     @(event_vif.message) begin
       case (event_vif.message)
         boot_done: begin
           -> event_vif.boot_done_ev;
           `uvm_info({get_type_name(),":monitor_dut"},"Boot Done", UVM_NONE)
         end
         cpu_instr_start: begin
           -> event_vif.cpu_instr_start_ev;
           `uvm_info({get_type_name(),":monitor_dut"},"CPU Instruction Start", UVM_NONE)
         end
         cpu_instr_finish: begin
           -> event_vif.cpu_instr_finish_ev;
           `uvm_info({get_type_name(),":monitor_dut"},"CPU Instruction Finished", UVM_NONE)
         end
         cpu_isr_start: begin
           -> event_vif.cpu_isr_start_ev;
           `uvm_info({get_type_name(),":monitor_dut"},"CPU Interrupt Subroutine Start", UVM_NONE)
         end
         cpu_isr_finish: begin
           -> event_vif.cpu_isr_finish_ev;
           `uvm_info({get_type_name(),":monitor_dut"},"CPU Interrupt Subroutine Finished", UVM_NONE)
         end
         default : begin
           -> event_vif.warning_ev;
           `uvm_info({get_type_name(),":monitor_dut"},$psprintf("Invalid CPU message : %h",
                     firmware_vif.message), UVM_NONE)
         end
       endcase
     end
   end
 endtask: monitor_dut
```

Conclusion