Architecting Your Way To Acceleration In UVM

Dean Justus

Paul Lungu

Ciena

October 1, 2015

Ottawa

Agenda

Why accelerate?   Why UVM-A?

Steps To Move Towards UVM-A

Standard UVM architecture

Separate BFMs from driver/monitor

Challenges

Conclusions

Why Accelerate? Why UVM-A?

Why Accelerate?

- Larger ASICs are taking longer to verify
  - Affects time to market
  - Stretches project schedules
  - Tradeoffs between verification coverage and time of tape-out
  - Higher risk of bugs, ECOs

Despite cost of H/W accelerators, acceleration can reduce project risk

Why UVM-A?

- Standard UVM top-level simulation for large ASICs is slow
  - Minimize interaction between the verification components and the DUT
  - Signal toggling driven by the drivers/monitors slows down the simulation
  - Have all the interface signaling toggling inside the HW and have minimal interaction with SW
  - Have most of the packet transaction to signal processing done in HW rather than SW

Steps To Move To UVM-A

Steps To Move To UVM-A

1. Create two separate top-levels (domains)
2. Separate the BFM portion of the driver/monitor from the UVM component
3. Create an hierarchical interface which contains all interfaces in the project

Steps To Move To UVM-A

Create two separate top-levels (domains)

- Single top-level – cannot be synthesized

```
module proj_name_top
  import uvm_pkg::*
  `include “uvm_macros.shv”
  …
  proj_name_top_if itf();
  …
  // assign DUT inputs to interface outputs
  // read DUT outputs into interface inputs
  …
  uvm_config_db#(virtual proj_name_top_if)::set(null,
"uvm_test_top", “proj_name_top_itf_itf", itf);

  proj_dut dut(…); …
endmodule
```

Steps To Move To UVM-A

Create two separate top-levels (domains)

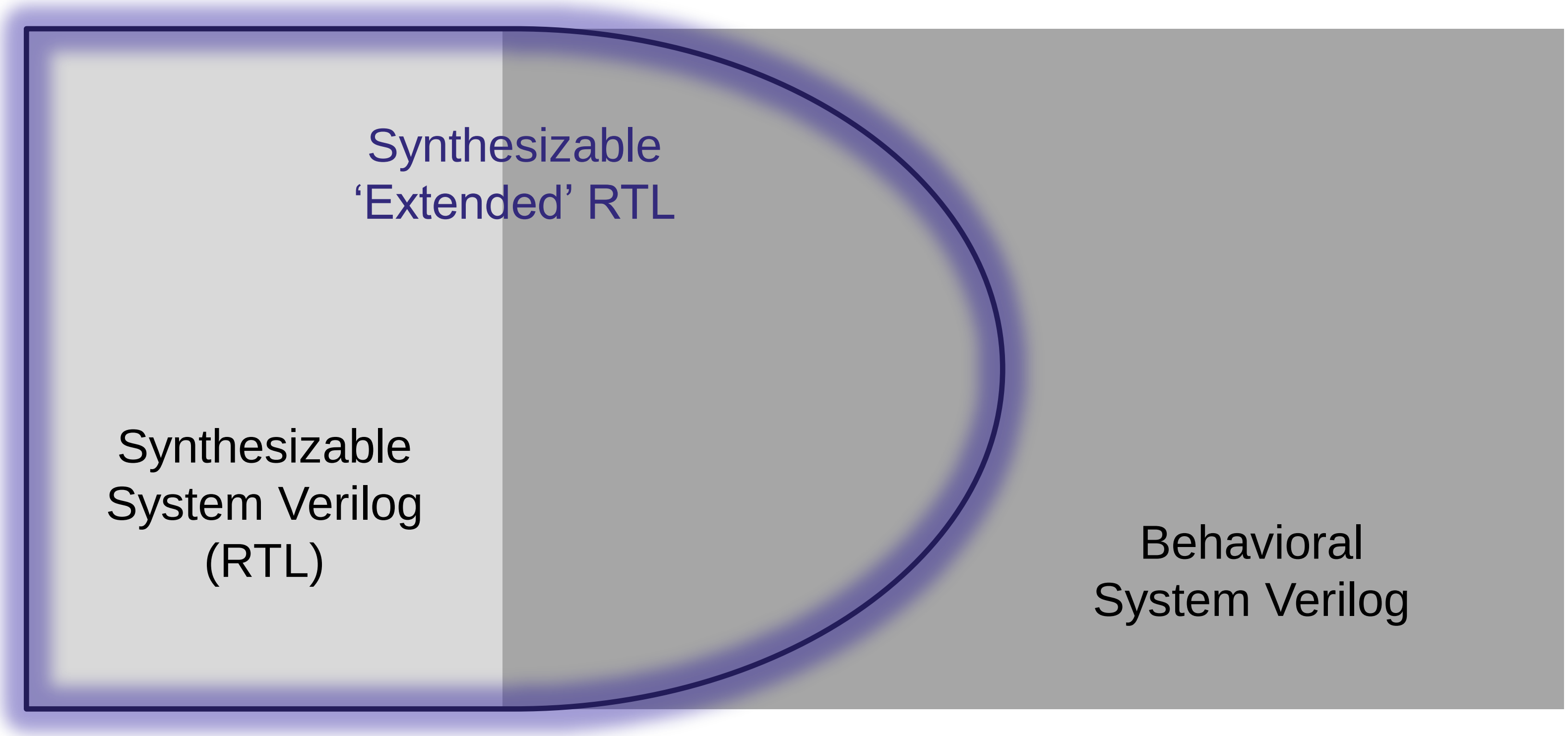
- Create HDL top-level – targeted to acceleration HW
  - HDL contains the DUT, interfaces, driver/monitor BFMs
  - HDL is timed – signal toggling
  - HDL domain is fully synthesizable (in accelerator)
  - HDL TB aspects written with ‘extended’ RTL code

```
module proj_hdl_top
  import proj_params_pkg::*
  proj_top_if itf();
  …
  // do all assignments here
  // read all DUT outputs here

  proj_dut dut(…);
  …
endmodule
```

Steps To Move To UVM-A

Create two separate top-levels (domains)

- ‘extended’ RTL
  - Synthesizable System-Verilog + Synthesizable Behavioral Constructs
  - Available constructs vary by vendor



Steps To Move To UVM-A

Create two separate top-levels (domains)

- Create HVL top-level – targeted to simulator
  - HVL contains the UVM environment
  - HVL is untimed – no signal toggling
  - HVL is not synthesizable
  - HVL communicates with HDL via task calls

```
module proj_hvl_top
  import uvm_pkg::*
  `include “uvm_macros.shv”
  …
  uvm_config_db#(virtual proj_top_if)::set(null,
       "uvm_test_top", “proj_top_itf",
       proj_hdl_top.itf);
 // start UVM tests
 initial
 begin
   run_tests();
 end
endmodule
```

Steps To Move To UVM-A

Separate the BFM portion of the driver/monitor from the UVM component

- BFMs to be written in ‘extended’ RTL within an interface

```
interface spi_driver_bfm_if(spi_if spi_itf);
        task run (input bit [23:0] m_spi_addr,
                  output bit      trans_done);
               send_a_pkt( m_spi_addr, …)
         endtask : run
        task send_a_pkt (input bit [23:0] m_spi_addr, …)
        endtask : send_a_pkt
endinterface : spi_driver_bfm_if
```

Steps To Move To UVM-A

Separate the BFM portion of the driver/monitor from the UVM component

- BFMs reside in HDL top level – drive signal interfaces
- driver/monitor reside in HVL level – transactions only
- BFMs written using ‘extended’ RTL synthesizable constructs
  - Convert transactions into signalling
  - implicit FSMs, initial blocks, named events & waits, unbounded loops, DPI imports & exports, assertions
- ‘extended’ RTL illegal constructs
  - fork-join

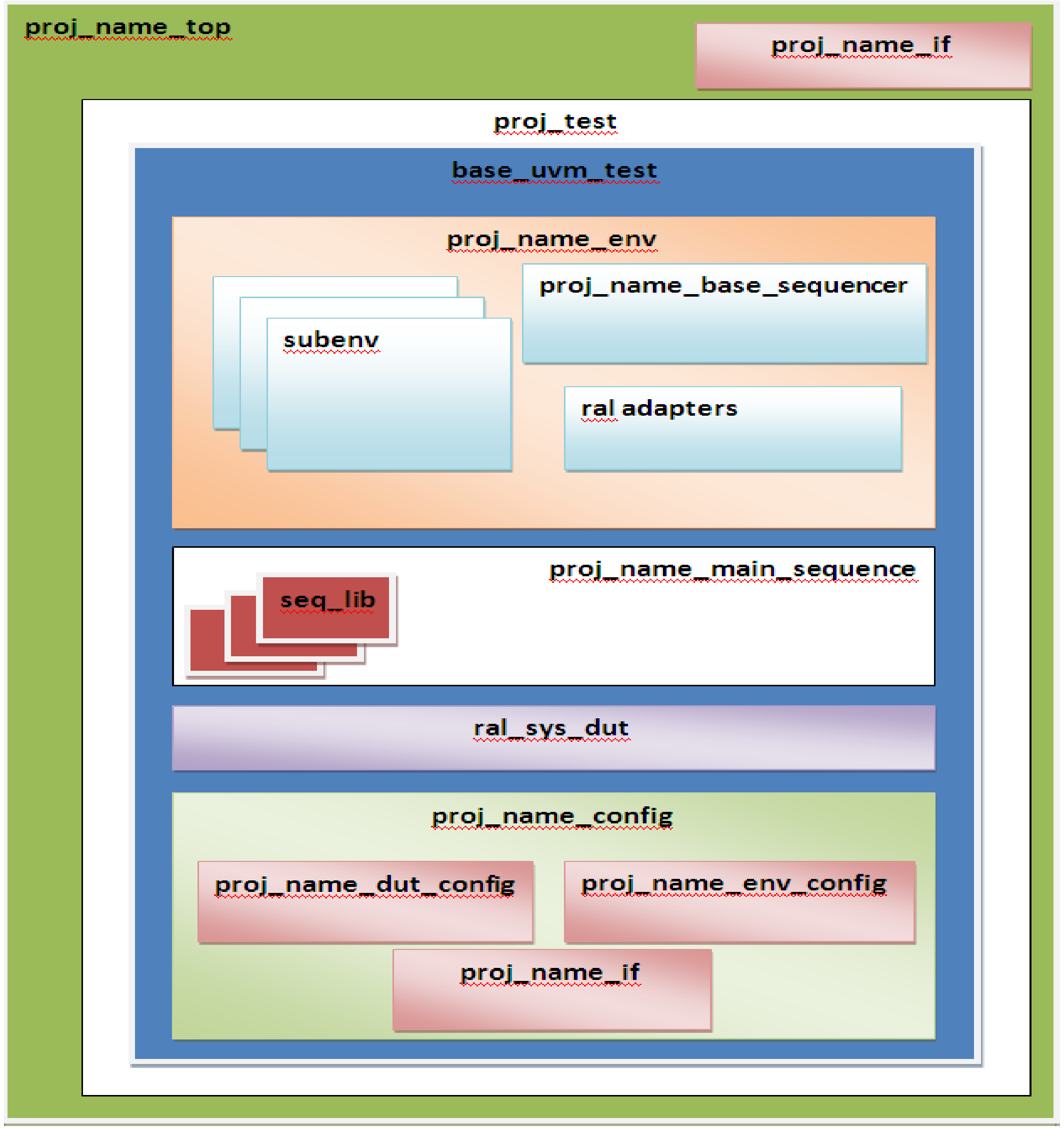
Steps To Move To UVM-A

Create an hierarchical interface which contains all interfaces in the project

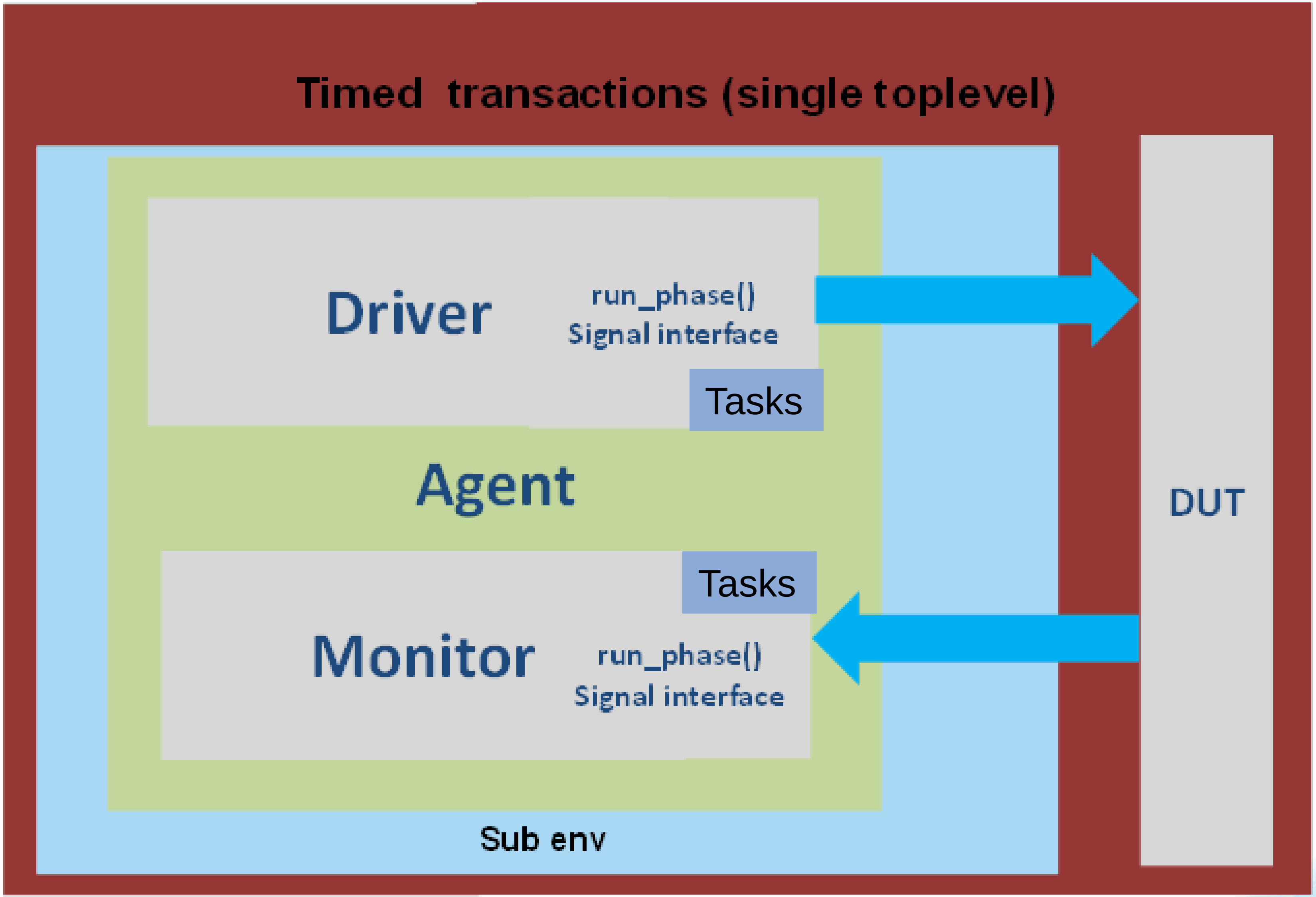
```
interface proj_name_if();

  import proj_name_params_pkg::*;

  clk_if   clk_itf[NUM_CLK_IF]();
  rst_if   rst_itf(.clk(clk_itf[0].clk));
  spi_if   spi_itf[NUM_SPI_IF](.clk(clk_itf[0].clk));
  i2c_if   i2c_itf[NUM_I2C_IF](.i_clk(clk_itf[4].clk),
                                .i_rstn(1'b1));
  ebus_if  ebus_itf[NUM_EBUS_IF](.clk(clk_itf[0].clk));

endinterface
```

Standard UVM Architecture

Standard UVM architecture



Standard UVM architecture



Standard UVM architecture

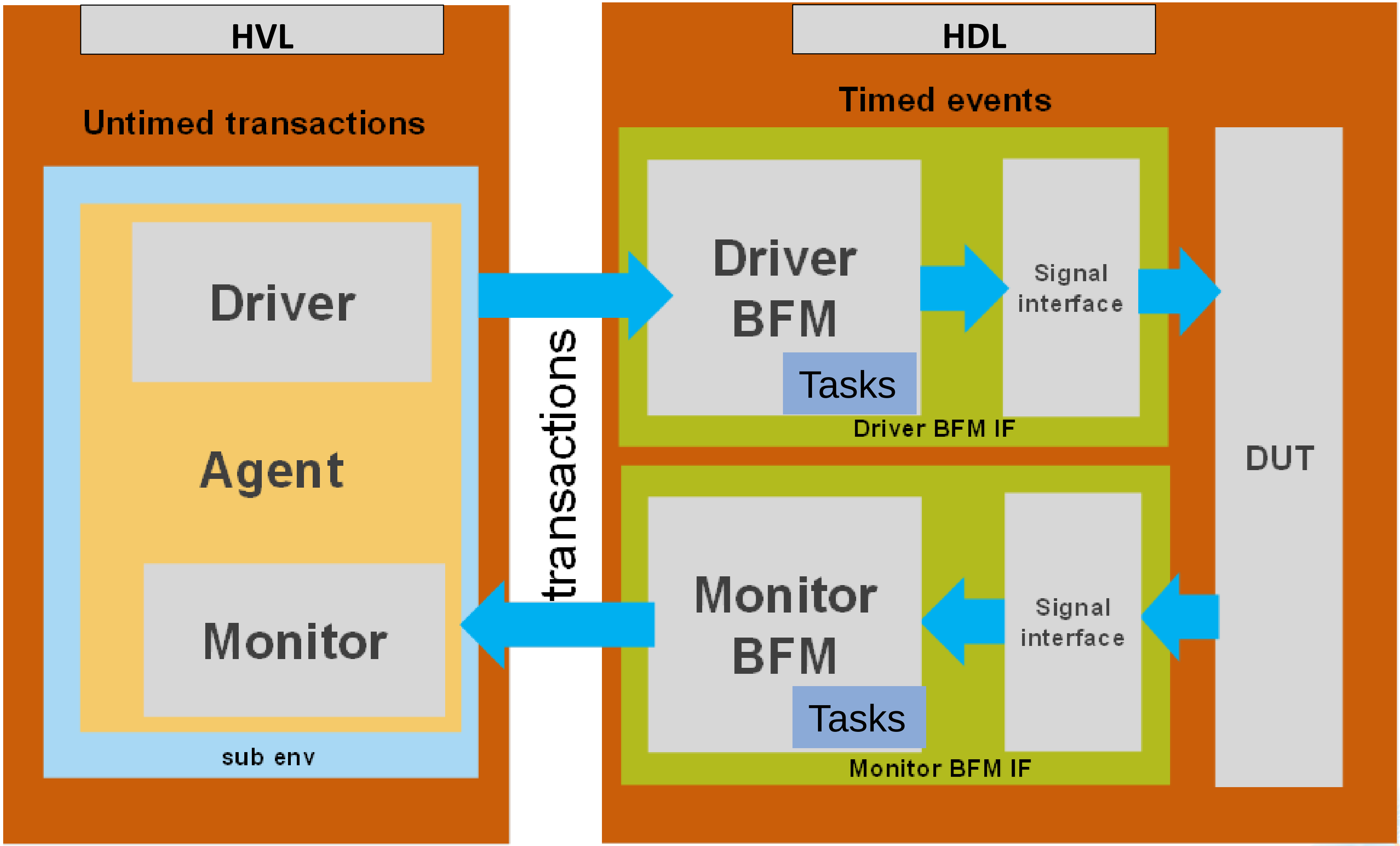
- Standard UVM architecture requires BFMs to be built inside the run_phase() for drivers/monitors
- Signal toggling happens inside the run phase
- Packets are created one at a time and sent to DUT via interface
- Packet manipulation (ie. Headers, FEC, CRC calculation) is done in simulation
- Significant amount of time is spent generating the right packet

Standard UVM architecture

```
class spi_driver extends uvm_driver #(spi_trans);
  `uvm_component_utils(spi_driver)
  spi_config                 m_config;
  spi_trans                  m_trans;
  virtual interface spi_if   vif;
  …
  task run_phase(uvm_phase phase);
    fork
      super.run_phase(phase);
    join_none
    reset_signals();  // to be accellerated
    get_and_drive();  // to be accelerated
  endtask: run_phase
  …
endclass
```

Separate BFMs From driver/monitor

Separate BFMs from driver/monitor



Separate BFMs from driver/monitor

- UVM-A driver/monitor:
  - modified driver is an extension of the original UVM driver
    - class spi_driver_a extends spi_driver;
  - instantiates a virtual interface to the BFM
    - virtual interface spi_driver_bfm_if bfm_vif;
  - receives transactions from the sequencer
    - seq_item_port.get_next_item(m_trans);
  - does not directly drive the protocol interface
  - communicates with the BFM through a task call
    - bfm_vif.run(m_trans.m_spi_addr, …);
    - a protocol is developed for passing variables between driver and BFM
  - has no notion of clocked or timed processes

Separate BFMs from driver/monitor

```
class spi_driver_a extends spi_driver; // create new components
  `uvm_component_utils(spi_driver_a)
   virtual interface spi_driver_bfm_if bfm_vif;
   task run_phase(uvm_phase phase);
   …
    bfm_vif.reset_signals(  is_active,  m_config.m_spi_mode,…); // task call
    forever begin
      seq_item_port.get_next_item(m_trans);
      m_trans = new();
      bfm_vif.run(  m_trans.m_spi_addr,   m_trans.m_spi_data…); // task call
      seq_item_port.item_done();
    end
   …
endclass
```

Separate BFMs from driver/monitor

	UVM	UVM-A
Driver	extended from uvm_driver	extension of existing UVM driver, or, new driver extended from uvm_driver
	instantiates virtual interface to protocol interface	instantiates virtual interface to BFM
	receives transactions from transactor	receives transactions from transactor
	directly drives protocol interface	passes transactions to BFM through task call, but does not directly drive protocol interface
	potentially drives interface using clocked processes	no notion of clocked processes
Monitor	extended from uvm_monitor	extension of existing UVM monitor, or, new monitor extended from uvm_monitor
	instantiates virtual interface to protocol interface	instantiates virtual interface to BFM
	communicates with transactor	communicates with transactor
	captures activity directly from protocol interface	passes transactions to BFM through task call, but does not directly access protocol interface
	potentially monitors interface using clocked processes	no notion of clocked processes

Separate BFMs from driver/monitor

- UVM-A driver/monitor BFM has the following characteristics:
  - is a System Verilog interface
    - interface spi_driver_bfm_if(spi_if spi_itf);
  - the protocol interface is an input
    - (spi_if spi_itf)
  - contains tasks that can be called by the driver
    - task run(input bit [23:0]   m_spi_addr, …);
  - drives the protocol interface
    - Converts transactions into signalling
    - spi_itf.spi_m2s.sck = m_spi_mode[1];
  - has clocked/timed processes

Separate BFMs from driver/monitor

```
interface spi_driver_bfm_if(spi_if spi_itf);
       …
       task run (input bit [23:0]    m_spi_addr,
              …
              output bit            trans_done);
              …
              send_a_pkt( m_spi_addr, …)
              …
        endtask : run
       …
       task send_a_pkt (input bit [23:0]    m_spi_addr, …)
              …
       endtask : send_a_pkt

endinterface : spi_driver_bfm_if
```

Separate BFMs from driver/monitor

```
//add the newly created BFM interfaces to the existing toplevel
interface
       i2c_driver_switch_bfm_if
       i2c_driver_switch_bfm_itf_0(i2c_itf[0]);
       i2c_monitor_bfm_if
       i2c_monitor_bfm_itf_0(i2c_itf[0]);
       rst_driver_bfm_if
       rst_driver_bfm_itf(rst_itf);
       clk_driver_bfm_if
       clk_driver_bfm_itf_0(clk_itf[0]);
       spi_driver_bfm_if
       spi_driver_bfm_itf_0(spi_itf[0]);
       spi_monitor_bfm_if
       spi_monitor_bfm_itf_0(spi_itf[0]);
       ebus_driver_bfm_if
       ebus_driver_bfm_itf_0(ebus_itf[0]);
       ebus_monitor_bfm_if
       ebus_monitor_bfm_itf_0(ebus_itf[0]);
```

Separate BFMs from driver/monitor

```
// env class

class proj_name_env extends uvm_env;
  `uvm_component_utils(proj_name_env)

  proj_name_config    m_config;
  clk_env             m_clk;
  spi_env             m_spi;
  ...
  ral2axi4_master_adapter     ral2axi4_master_adp;
  ral_sys_dut                 RAL; // RAL -    reg_model
  proj_name_base_sequencer    m_base_seqr;
  ...
```

Separate BFMs from driver/monitor

```
  function void build_phase(uvm_phase phase);
  ...
  uvm_config_db#(virtual clk_if)::set(this, "m_clk*", "clk_vif[0]",
       m_config.itf.clk_itf[0]);
  ...
  uvm_config_db#(virtual spi_if)::set(this, "m_spi*", "spi_vif[0]",
       m_config.itf.spi_itf[0]);
  uvm_config_db#(proj_name_params_pkg::axi4_master_if_t)::set(this,
       "m_axi4_master*", "axi4_master_vif",
       m_config.itf.axi4_master_itf);
  ...
  // ...other interfaces excluded for clarity
  endfunction
endclass
```

Separate BFMs from driver/monitor

```
// changes to the env class to accomodate the BFM interfaces


       uvm_config_db#(virtual
       i2c_driver_switch_bfm_if)::set(this, "m_i2c*",
       "i2c_driver_switch_bfm_vif[0]",
       m_config.itf.i2c_driver_switch_bfm_itf_0);


       uvm_config_db#(virtual i2c_monitor_bfm_if)::set(this,
       "m_i2c*", "i2c_monitor_bfm_vif[0]",
       m_config.itf.i2c_monitor_bfm_itf_0);
       ...
       uvm_config_db#(proj_name_params_pkg::axi4_master_driver_bf
       m_if_t)::set(this, "m_axi4_master*",
       "axi4_master_driver_bfm_vif",
       m_config.itf.axi4_master_driver_bfm_itf);
```

Separate BFMs from driver/monitor

```
//create a new base UVM test for acceleration

class base_uvm_a_test extends base_uvm_test;
  `uvm_component_utils(base_uvm_a_test)
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    accelerate_agent("i2c");
    accelerate_agent("rst");
    accelerate_agent("clk");
    accelerate_agent("spi");
    accelerate_agent("ebus");
  ...
  endfunction : build_phase
```

Separate BFMs from driver/monitor

```
//use factory overrides to replace the standard drivers with
acceleration friendly ones


virtual task accelerate_agent(string name);
    case(name)
      "i2c": begin
        factory.set_type_override_by_type(i2c_driver_switch::get_ty
        pe(),i2c_driver_switch_a::get_type(),"*");


        factory.set_type_override_by_type(i2c_monitor::get_type(),
        i2c_monitor_a::get_type(),"*");
      end
      …
endtask
```

Challenges

Challenges

- Use of parameterized classes and interfaces
  - syntax depends on scenario
- Limitations of ‘extended’ RTL vs behavioural code
  - i.e. fork/join
- Converting existing UVM environment to UVM-A
  - Management of UVM/UVM-A parallel models
  - Adapting existing BFM code to ‘extended’ RTL compliant code
- 3rd party VIP for complex protocols
  - Can existing IP be migrated
  - Does vendor have acceleration friendly models

Synopsys Users Group

Conclusions

Conclusions

- UVM-A architecture approach comparable to UVM architecture wrt S/W simulation performance
  - acceptable default methodology
  - UVM-A approach promoted by some UVM consultants
- UVM-A implementable from a reuse architecture
  - UVM Factory Override provides elegant method to utilize both UVM and UVM-A model pairs
  - Still, UVM-A easier to implement ‘from scratch’
- UVM-A BFM development requires little or no ‘UVM’ knowledge (an opinion)
  - Implementable by non-verification professionals
  - Provides team management options

Conclusions

- Decision to target acceleration is a project choice
  - HW cost and effort level vs ‘time to market’ and project schedules

Synopsys Users Group

Thank You