
# Architecting Your Way To Acceleration In UVM

Paul Lungu, Dean Justus

Ciena
Ottawa, Canada

www.ciena.com

**ABSTRACT**

*Today's ASICs are increasingly complex and, with the number of gates commonly exceeding 100 million the simulation as we know it reaches limits in terms of performance. The normal evolution would be to accelerate the DUT at hardware speed. However, the continuous interaction between the verification environment and the DUT implemented in an accelerator impacts the maximum performance one can achieve. The solution would be to synthesize parts of the verification environment responsible for signal toggling into the same accelerator along with the DUT. This has the potential to significantly reduce the overall simulation overhead due to simulator/accelerator interaction. This is an old trick but doing so in UVM is quite new. This paper will discuss in detail the path to architect an UVM environment for acceleration (UVM-A). A seamless methodology to switch from a standard UVM environment to a fully emulation friendly UVM-A environment will be discussed.*

## Table of Contents

## Table of Figures



## 1. Introduction

Technological advances have resulted in more complex designs as ASICs have improved and are able to accommodate ever-increasing volumes of logic. This increases the complexity of the design in many ways, but also impacts the verification schedule. Systems on a Chip (SoC) larger than 100M+ gates will require the use of sophisticated simulation techniques and verification methodologies. Bugs, if not found at early stages of the design process can get very expensive to fix later on. Therefore, it is imperative to ensure the verification process is closed before tape out. To achieve this it is important to have all the tests in the verification plan completed and passing. This requires a speed up in the verification process. Maximizing reuse between projects or within the same project, as well as achieving a faster simulation turnaround in order to reduce the time spent to run and debug tests can achieve this.

Normally, the test debug process involves numerous iterations that can extend the verification process, increasing the schedule. One way of achieving reusability is to adopt an advanced verification methodology, and UVM has become, in the last several years, the methodology of choice for many companies. However, given an ever increasing size of the design, UVM alone cannot be relied upon to speed up the verification cycle, especially when used at the top level of an SoC or system. Verification at this level has to involve other methods which usually run the simulation at hardware speed. The methodology of choice to solve this issue is to simulate the Design Under Test (DUT) in an emulator. Emulators can run at millions of clock cycles per second, by far exceeding the simulation speeds realized in software, which can at best reach thousands of clocks per second. As many of us know, this is an old trick, but having an emulator work within a UVM environment is quite new. However, the reusability gains provided by using

UVM have to be maintained while architecting for emulation, and there are a few aspects that a user needs to be aware of. Once the methodology and architecture have been selected for emulation the same code should be used for simulations. The user has to ensure that the test cases selected for emulation only can be run on a simulator as well without changing the code. All the classes, randomization techniques and constraints used in UVM should be used for emulation, allowing the user to virtually run any of the tests on an emulator. In reality only a reduced number of test cases (which usually run longer in simulation) will be selected to run on an emulator.

This paper will present a few architectural choices needed to create a UVM environment capable of running on both an emulator and a simulator. A clear way of migrating from an existing standard UVM environment to an emulation friendly one will be presented. The current verification methodology involves using a company specific library of components. The paper will discuss in detail the process of creating a new library of components with synthesizable driver/monitor BFMs for emulation starting from the existing library that is not emulation friendly. A simple but elegant method of replacement of the existing drivers and monitors with a pair of separated untimed and timed drivers/monitors using factory overrides will be presented. This approach will create a non-invasive methodology which allows both emulation friendly and non-emulation friendly components to live together and give the verification primes choices when deciding which way to go at different stages in the project. Based on our research, adoption of this methodology should give the user increases in speed of a few hundred to a few thousand times compared with regular simulation speeds.

The paper will introduce a new methodology name, UVM for acceleration (UVM-A) that refers to the UVM adapted to work for acceleration. The paper will also contain references to emulation friendly environments that basically mean that UVM-A concepts are applied. The words "acceleration" and "emulation" will be used in the paper with the same meaning.

We first deployed this methodology on a Xilinx FPGA. VCS-MXi 2014-03. SP1 was used to run simulations. No accelerator box was targeted. All simulations were carried out using a simulator.

## 2. What is UVM-A?

UVM-A stands for Universal Verification Methodology for Acceleration and the term has been introduced recently to refer to the requirements needed for a verification environment to work with a HW accelerator. While accelerating designs in an accelerator box is an old topic, using UVM as the verification environment is new and the topic needs special attention.

The main requirement when architecting such a verification environment is to ensure it still works in a standard simulation environment with no additional changes.

To reduce the amount of interaction between accelerator and simulator the methodology requires a dual top-level solution. One top-level will be used to instantiate everything that is consuming time and the other top-level will be used for the verification code that is untimed. In other words, one emulated/accelerated domain and one simulated domain will be created using this method. The emulated/accelerated domain or HDL domain will contain the DUT, interfaces, and synthesizable BFMs used to drive signals to/from the DUT and any other models that are timed. The simulation domain or HVL domain will contain the entire UVM infrastructure necessary to run simulations.

Another major requirement for UVM-A is to separate the BFM timed portion of the drivers and monitors from the UVM untimed portion of these components, move these to the HDL domain while the UVM untimed portion remains in HVL domain. The timed portion of the drivers/monitors will have to be modified into a synthesizable form creating an "acceleratable environment". The BFM along with the DUT will run at hardware acceleration speeds, reducing the simulation time due to the interaction between the two domains.

The HVL domain will simply be required to issue transactions to the box, and these transactions will consume no time in the simulator. The transaction once received by the HW accelerator will be sent to the BFM and executed. The execution time is minimal since it is done in HW.

The main requirement for the code sitting in the HDL domain is that it must be synthesizable by the HW box targeted in this application. Hence, the code will have to obey the rules of the box. The vast majority of accelerators from the main vendors allow the user to write code in an extended RTL (X-RTL) superset of constructs which allows some behavioural code to be present inside the modules sent to the box for synthesis. Each box will have its own rules but all the boxes accept most behavioural constructs. The following is a list of some behavioural constructs universally accepted by HW accelerators:

- implicit FSMs,
- initial blocks,
- named events & waits,
- unbounded loops,
- DPI imports & exports,
- assertions.

Details with these constructs will be given in the subsequent chapters.

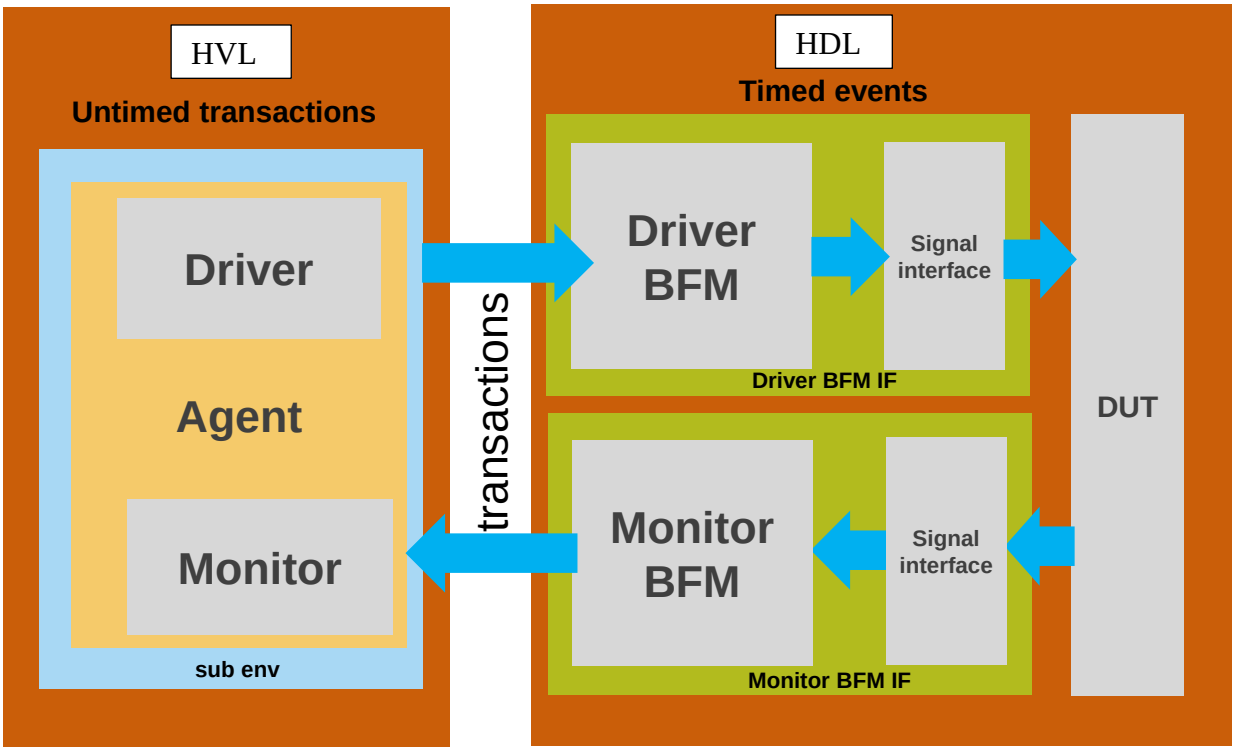


**Figure 1 UVM-A architectural diagram**

Figure 1 above explains the domain separation and gives a clear separation of what is executed in simulation and what is done inside the emulator. The portions of the driver and monitor remaining in the HVL domain will control the BFMs from the HDL domain via task calls. Only transaction request will cross the boundary between HDL and HVL domains. No signal toggling will occur in this case at the boundary between the two domains.

## 3. Starting from a standard UVM architecture

Figure 2 below shows the standard agent architecture that serves as the starting point. At the beginning of the project the user normally doesn't know if acceleration is needed or not. It really depends on the length of the longest-running test case. This is usually decided upon towards the last phase of the project, once the environment is already built. For this reason the proposed

UVM-A architecture doesn't necessarily require that the user start with acceleration in mind. This decision may not be needed that early in the program.

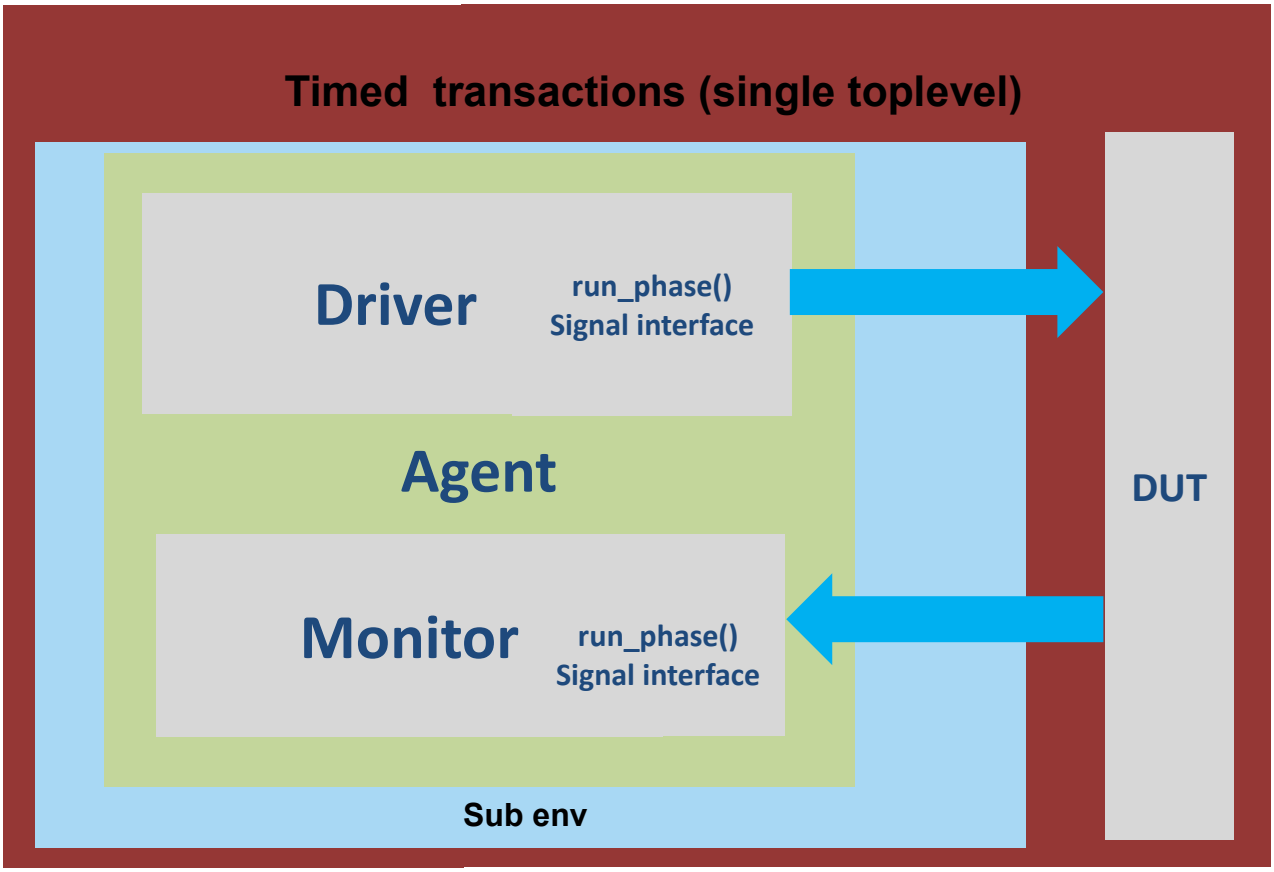


**Figure 2 Standard UVM agent architecture**

As can be seen above, the standard UVM architecture requires standard agents with drivers and monitors having their respective BFMs implemented inside the run_phase() task. The virtual signal interface is part of the driver and monitor for all the protocols used. The transactions inside the driver and monitor run_phase() all are timed transactions. The signal toggling is driven from inside the run_phase(). This consumes most of the simulation time in a standard UVM environment architecture. For clarity Figure 3 shows the whole UVM standard architecture based on multiple subenvironments. Each agent is wrapped in a subenvironment and this makes it easier at the toplevel.

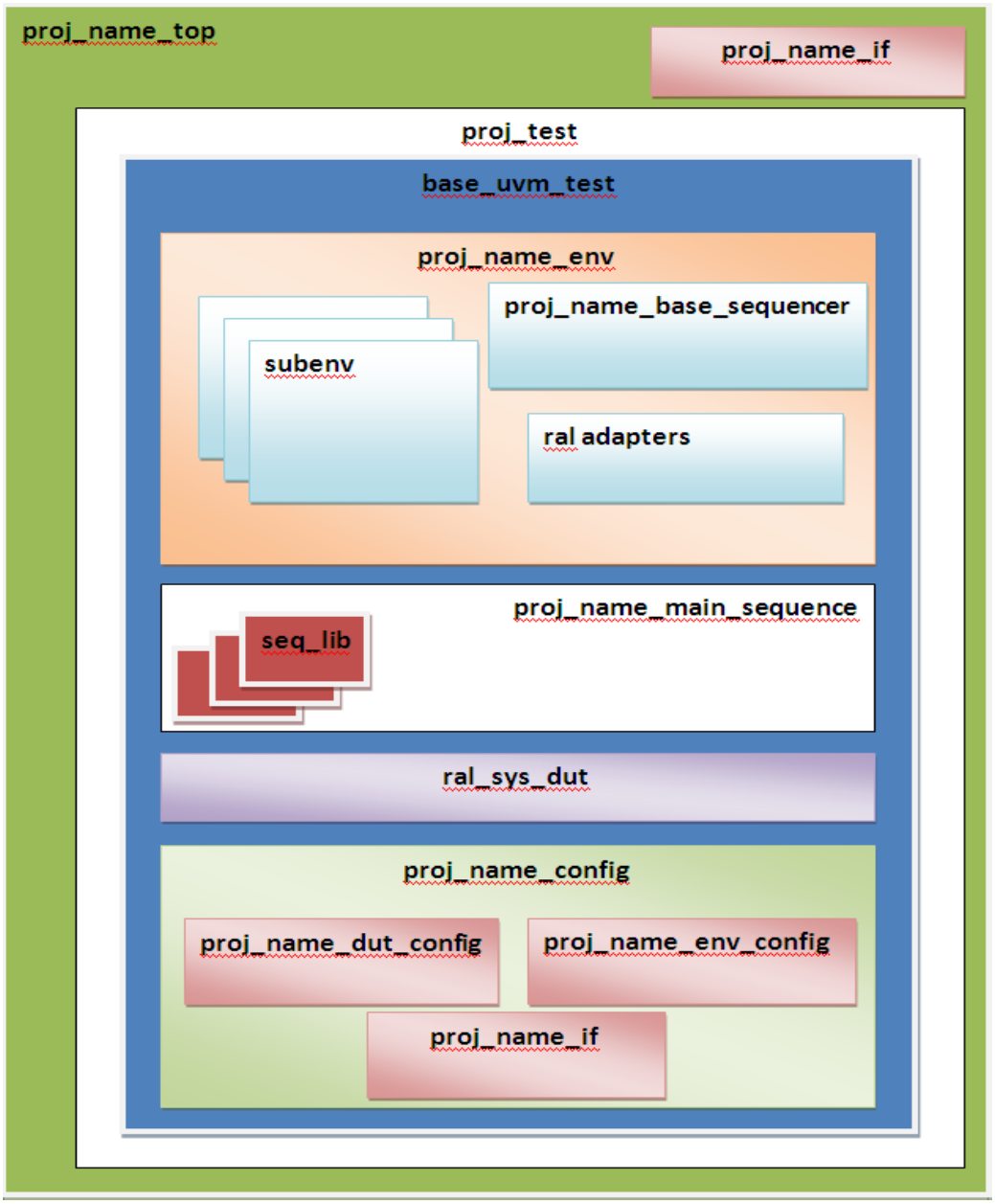


**Figure 3. Standard UVM toplevel environment**

## 3.1 Use of the hierarchical interfaces

The standard architecture makes use of a hierarchical interface at the top-level. This works like a container for all the interfaces used in the verification. A code example of such a hierarchical interface is shown in Figure 4. From the single top-level the interface handle is sent to the verification world using uvm_config_db as seen in Figure 4. The handle is then passed to the hierarchical interface from the config class since the config class was chosen to carry over this info inside the verification environment. The base_uvm_test is used as an intermediary step to

assign the handle to the interface from the config class. As seen in Figure 5, the use of the hierarchical interface reduces the amount of coding needed at top-level and it creates a place where everything interface related can be dealt with. This will help later on when UVM-A specific code will be added.

```
1  interface proj_name_if();
2
3    import proj_name_params_pkg::*;
4
5    clk_if                                    clk_itf[NUM_CLK_IF]();
6    rst_if                                     rst_itf(.clk(clk_itf[0].clk));
7    axi4_master_if #(AXI4_USER_WIDTH,
8                  AXI4_ID_WIDTH,
9                  AXI4_M_ID_WIDTH,
10                 AXI4_ADDR_WIDTH,
11                 AXI4_M_ADDR_WIDTH,
12                 AXI4_NUM_LANES,
13                 AXI4_DATA_WIDTH)     axi4_master_itf();
14   pcie_rp_env_if                      rp_env_itf();
15   pcie_rp_dut_if                      rp_dut_itf();
16   misc_if                             misc_itf(.clk(clk_itf[0].clk));
17   spi_if                              spi_itf[NUM_SPI_IF](.clk(clk_itf[0].clk));
18   i2c_if                              i2c_itf[NUM_I2C_IF](.i_clk(clk_itf[4].clk), .i_rstn(1'b1));
19   ebus_if                             ebus_itf[NUM_EBUS_IF](.clk(clk_itf[0].clk));
20
21 endinterface
```

**Figure 4 Top-level hierarchical interface**

As seen in the code above, all the interfaces used in the project can be instantiated hierarchically inside the top-level interface. It is recommended to create a 'misc' interface (see line 16 in Figure 3) that will be the wrapper around everything else we need to ensure that the verification environment is untimed. Delays of any type can be added to tasks within the 'misc' interface, which can then be called from the sequences or any other place where tests are written.

## 3.2 Single top-level

As discussed above, the standard UVM architecture requires a single top-level. Both the verification environment and the DUT are instantiated at this level. Since both the verification environment and the DUT consume time in a similar way, all the signal toggling between the two consume simulation time which translates into long verification cycles, especially when top-level or system level simulations are executed. Figure 5 below shows how the hierarchical interface is instantiated at the top-level along with the DUT and the verification environment. This is the standard way in UVM.

```
1.  `timescale 1ns/1ps
2.  module proj_name_top;

3.  import uvm_pkg::*;
4.  import proj_name_test_pkg::*;
5.  import proj_name_params_pkg::*;
6.  `include "uvm_macros.svh"

7.  proj_name_if   itf();

8.  // assign clocks and resets
9.  assign itf.axi4_master_itf.clk_reset.clock   = dut.gms_clk;
10. assign itf.axi4_master_itf.clk_reset.reset_n = itf.rst_itf.async_n_rst;

11. //assign PCIe dut interface
12. assign itf.rp_dut_itf.sys_clk_p   = itf.clk_itf[0].clk;
13. assign itf.rp_dut_itf.sys_clk_n   = ~itf.clk_itf[0].clk;
14. assign itf.rp_dut_itf.sys_rst_n   = itf.rst_itf.async_n_rst;

15. bit run_cmn_setup_vif = setup_common_vif();

16. function bit setup_common_vif();
17. uvm_config_db#(virtual proj_name_if)::set(null, "uvm_test_top", "proj_name_itf", itf);
18. return 1;
19. endfunction

20. // Run the UVM test
21. initial begin
22. run_test();
23. end

24. proj_name # (
        a. .PARAM1          (`PARAM1)
            i. ) dut (
        b. .RESET_N(itf.rst_itf.async_n_rst),
        c. .PM_TICK(itf.clk_itf[4].clk),
        d. .CLK_IN_P( itf.clk_itf[3].clk),
        e. .CLK_IN_N(~itf.clk_itf[3].clk)
        f. ...  );

25. endmodule
```

**Figure 5. Single Top-level**

As seen in the code example above, everything is instantiated in the unique top-level. The top-level interface is sent via the config database (line 17) to the top-level UVM test. The base UVM test will access the hierarchical interface from the config_db and will distribute it down

the hierarchy to the top-level environment. This will be explained in the subsequent chapters. The environment (i.e. drivers, monitors) is in direct contact with the DUT signals via the interfaces. All of this happens at simulation speed, where every clock tick which has the potential to slow down the simulation significantly.

## 3.3 Standard drivers/monitors in the library

At the start of a project the verification environment will either be derived from an existing environment or it will be developed from scratch. If the environment is a reuse environment following a UVM methodology, then it will of course contain standard UVM components, including drivers and monitors derived from the uvm_driver and uvm_monitor classes. However, do note that when starting a test bench from scratch, the decision could be made to architect the test bench in such a way that it is acceleration-ready, rather than simply architecting it using a standard methodology.

Regardless of the direction the test bench architecture is taking, starting with a standard driver and monitor, designed as they are in any standard UVM environment, is not in and of itself an issue as aspects of these classes are necessary in the acceleration model and the classes will be extended into the acceleration version. And for the purposes of this discussion it is assumed that the test bench is starting from an existing UVM environment which is not targeted for acceleration. A library of standard, reusable UVM components has been created with required standard protocols in place in the test environment along with any project specific proprietary protocols. The intent is to have commonalities among projects and to maintain a similar architecture. The standard architecture requires that every agent or array of agents be embedded into a sub-environment, which is a reusable piece throughout the project. For the SPI model discussed, refer to Section 9.1, Standard UVM Environment SPI Models for an example of the driver/monitor pair. Note that the driver in this case:

- is extended from uvm_driver
  - class spi_driver extends uvm_driver #(spi_trans);
- instantiates a virtual interface
  - virtual interface spi_if vif;
- receives transactions from the transactor
  - seq_item_port.get_next_item(m_trans);
- directly drives the interface
  - i.e. vif.spi_m2s.sck = m_config.m_spi_mode[1];

Meanwhile, the monitor:

- is extended from uvm_monitor
  - class spi_monitor extends uvm_monitor;
- instantiates a virtual interface
  - virtual interface spi_if vif;
- captures activity directly from the interface
  - i.e. @(posedge vif.spi_m2s.cs);

When following this paradigm, the goal will be to leave this existing test environment mostly undisturbed in moving towards an acceleration ready environment. The only piece of the existing environment that will need to be modified is the driver and monitor.

Note that, whether to start a test bench design with a standard UVM architecture as opposed to an acceleration-ready methodology is an architecture decision that depends on the needs of the project and on whether an acceleration target is anticipated or not. In saying this, the point is not to advocate for starting with UVM-A vs UVM. However, it should be understood in the architecture decision process that converting drivers and monitors into an acceleration-ready form may require significant effort, which may or may not affect the architecture decision. This will be discussed further in Section 4, Moving towards UVM-A.

### 3.4 Architecting for standard UVM

Some projects may not need acceleration until close to the end of the project when very long top-level tests are written. Having a flexible methodology that can be applied in flight is helpful in the sense that the project can be started using a standard library of components and have most of the tests written using the regular methodology. Once the project reaches a certain level of development and the simulations are getting longer at the top-level, an accelerator can be used. But the methodology will have to be changed to use the UVM-A library of components that are developed separately. The subsequent chapters will explain the standard methodology using a standard library of components, and then how the UVM-A library has been developed and how it can be used in an existing project. In conclusion, you can start the project using standard UVM and worry about UVM-A later.

### 3.5 Standard UVM Base Test

As mentioned in 3.2 the single top-level module is responsible to send the project hierarchical top-level interface to the top-level UVM base test. The example below shows how the test retrieves the interface from the config_db and initializes the hierarchical interface from the config class. The methodology consists in passing the interface via the config class since this class is passed via the config_db anyway.

```
1.  class base_uvm_test extends uvm_test;
2.  `uvm_component_utils(base_uvm_test)

3.  proj_name_env           m_env;
4.  proj_name_config        m_config;
5.  proj_name_main_seq      m_seq;
6.  ral_sys_dut             RAL;
7.  function void build_phase(uvm_phase phase);
8.  super.build_phase(phase);
9.  ...
10. m_config     = proj_name_config::type_id::create("m_config", this);
11. if ( !uvm_config_db#(virtual proj_name_if)::get(this, "", "proj_name_itf", m_config.itf))
12. `uvm_fatal(get_type_name(), "Can't retrieve proj_name_if!");

13. RAL = ral_sys_dut::type_id::create("RAL",this);
14. RAL.build();
15. RAL.reset();
16. RAL.lock_model();

17. uvm_config_db#(proj_name_config)::set(this, "m_env*", "config", m_config);
18. uvm_config_db#(ral_sys_dut)::set(null, "*", "ral_sys_dut", RAL);
19. m_seq = proj_name_main_seq::type_id::create("m_seq");
20. m_env     = proj_name_env::type_id::create("m_env", this);
21. endfunction : build_phase
22. ...
23. endclass
```

**Figure 6. Standard base UVM test**

Figure 6 above shows how the build_phase() takes care of creating the config object (line 10) and subsequently passing the handle of the top-level interface (line 11). The standard base UVM test also creates the RAL object (line 13) the main sequence (line 19) and the UVM environment (line 20).

## 3.6 Standard env

One of the members of the UVM base test above is the top-level environment. This is a class which normally constructs all the agents used in the verification environment. The proposed standard methodology assumes that all the agents are encapsulated inside of a sub-environment. Inside the agent sub-environment the scoreboard is instantiated along with the agent, the coverage class, and the protocol interface. Figure 7 shows how the top-level environment gets the config class from the config_db and then extracts the interface, distributing it further to the sub-environments.

```
class proj_name_env extends uvm_env;
  `uvm_component_utils(proj_name_env)

  proj_name_config                              m_config;
  clk_env                                       m_clk;
  proj_name_params_pkg::axi4_master_env_t       m_axi4_master;
  spi_env                                       m_spi;
...
  ral2axi4_master_adapter                       ral2axi4_master_adp;
  ral_sys_dut                                   RAL; // RAL - reg_model
  proj_name_base_sequencer                      m_base_seqr; function void
  ...
  function void build_phase(uvm_phase phase);

   if ( !uvm_config_db#(ral_sys_dut)::get(this, "", "ral_sys_dut", RAL)) begin
    `uvm_fatal(get_type_name(), "Can't retrieve RAL!");
  end

  if ( !uvm_config_db#(proj_name_config)::get(this, "", "config", m_config)) begin
    `uvm_fatal(get_type_name(), "Can't retrieve environment config!");
  end

  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
```

**Figure 7. Standard UVM environment**

Only pieces from the top-level environment will be presented in this paper. The goal here is to see what it takes to move to UVM-A later on. All of the objects created in the base UVM test are sent via config_db and are retrieved by the top-level environment class. This two-step approach of passing the config and RAL classes was used in the standard methodology to ensure reusability.

# 4. Moving towards UVM-A

Based on the environment described in the previous chapters, moving to an acceleration friendly environment is simply a matter of following the steps described below.

1. First, we need to separate the BFMs from the respective drivers and monitors and create separate classes for the driver and monitor. The BFMs for the drivers and monitors will be written in a task using synthesizable code. The tasks will be a member of an interface. This will be discussed in detail in 4.3.
2. Second, we need to create two top-levels. One top-level is for the synthesizable code, containing the DUT and the top-level hierarchical interface. The other is for untimed code that basically contains everything which is non-synthesizable plus the run_tests() construct to start the UVM test.
3. The next step will be to add the above BFM interfaces into the hierarchical interface. This is detailed in 0 below.
4. After the BFM interfaces are instantiated into the top-level hierarchical interfaces, the newly added interfaces will have to be passed down the hierarchy to the respective driver and monitor (which extend from the standard driver and monitor) and have the new BFM interface added as a member. This is detailed in 0.
5. Another necessary step in the process is to create another UVM base test specific to acceleration, and this class will extend from the standard one. This is detailed in chapter 0.
6. The last step is to use factory overrides to replace the standard drivers and monitors with the acceleration friendly versions that contain the BFMs separated into synthesizable tasks. This can be done in the newly created UVM base test for the accelerated tests. In using this method, the overrides apply to all accelerated test cases or on a per test case basis, in which case only a selected number of tests extending from the UVM base test for acceleration will have certain drivers and monitors overwritten. Some test cases may not require all the drivers and monitors replaced.

Once all of these steps are implemented the verification environment will communicate with the synthesizable top-level via calls in zero time. All simulation time is spent inside the synthesizable task that gets called from the hierarchical interface. The following description basically shows how the time spent in verification is actually 0 while everything moves at hardware acceleration speeds inside the box by the calls made from simulator to accelerator. On a timeline the sequence looks like this:

1. UVM test requires an action by calling a task via the hierarchical interface.
2. The task, once called, starts advancing the time based on a clock, and this is done in acceleration rather than simulation.
3. Once the task is completed the handle gets back to simulation, which will require an other action and so on.

### 4.1 Changes to the hierarchical interface

All of the drivers and monitors will have to be replaced with their equivalent from the acceleration friendly library. That means that all will have the BFMs removed from the run_phase() and instead the BFM will be implemented in a task that is part of an interface specific for that driver or monitor. The content of the BFM interface will be detailed in chapter 4.4 but this chapter will just show how to instantiate the newly created BFM interfaces into the existing top-level hierarchical interface.

```
1.  i2c_driver_switch_bfm_if          i2c_driver_switch_bfm_itf_0(i2c_itf[0]);
2.  ...
3.  i2c_monitor_bfm_if                i2c_monitor_bfm_itf_0(i2c_itf[0]);
4.  ...
5.  rst_driver_bfm_if                 rst_driver_bfm_itf(rst_itf);
6.  clk_driver_bfm_if                 clk_driver_bfm_itf_0(clk_itf[0]);
7.  ...
8.  spi_driver_bfm_if                 spi_driver_bfm_itf_0(spi_itf[0]);
9.  spi_monitor_bfm_if                spi_monitor_bfm_itf_0(spi_itf[0]);
10. axi4_master_driver_bfm_if #(AXI4_USER_WIDTH,
                i.  AXI4_ID_WIDTH,
                ii. AXI4_M_ID_WIDTH,
                iii. AXI4_ADDR_WIDTH,
                iv. AXI4_M_ADDR_WIDTH,
                v.  AXI4_NUM_LANES,
                vi. AXI4_DATA_WIDTH)
                                      i. axi4_master_driver_bfm_itf(axi4_master_itf);
11. ebus_driver_bfm_if                ebus_driver_bfm_itf_0(ebus_itf[0]);
12. ebus_monitor_bfm_if               ebus_monitor_bfm_itf_0(ebus_itf[0]);
```

**Figure 8. UVM-A top-level hierarchical interface**

As seen in the code above, arrays of interfaces have to be instantiated inside the hierarchical interface along with the existing signal interfaces. This new addition is minimal in terms of coding and it helps in making the environment aware of the new synthesizable code added. Extra care is necessary when adding the parameterized interface (line 10). The syntax used to compile this interface is especially challenging so additional care is required while the code is written.

## 4.2 Create dual top levels

Another important step in modifying the existing test environment to work in an acceleration friendly scenario is creating a synthesizable portion next to a portion that contains untimed code running the UVM test. A two top-level approach is needed, where one top-level (named proj_name_hdl_top here) will contain the DUT along with the hierarchical interface, while the other top-level (named proj_name_hvl_top here) contains the run_tests() construct that starts the UVM test. The code below shows how these top levels are created and what they contain. The HDL top-level gets the DUT and the hierarchical interface from the existing single top-level while the HVL top-level will get the run_tests() construct. It is recommended to move the setting of the hierarchical interface into the config_db into the HVL portion since this is a non-synthesizable construct for all the boxes.

```
`timescale 1ns/1ps

module proj_name_hdl_top;
// copy the existing single toplevel and remove all the non-synthesizable
//code from the single toplevel
endmodule
```

**Figure 9. HDL top-level**

```
`timescale 1ns/1ps

module proj_name_hvl_top;

// Place all non-synthesizable. code here

  import uvm_pkg::*;
  import proj_name_test_pkg::*;
  import proj_name_params_pkg::*;
  `include "uvm_macros.svh"

...

  // Run the UVM test
  initial begin
    $display("Initiate UVM RunTest.");
    run_test();
  end
endmodule
```

**Figure 10. HVL top-level**

## 4.3 Separate BFMs from drivers/monitors

Assuming that the test environment is starting from a standard UVM environment, the steps to modify the existing driver and monitor models would be as follows. Refer to section 9.2, UVM-A Environment SPI Models for examples of the modified driver/monitor and the accompanying BFMs.

First, the driver and monitor will need to be split into driver/BFM and monitor/BFM pairs. This means that procedural code that existed in the driver/monitor will need to be transferred and potentially translated into a form suitable for the BFM. The driver/monitor will still communicate with the transactor/sequencer. However, these commands will need to be formed into a 'protocol' that can be passed to tasks in the new BFM.

In terms of the structure of the driver, the following is noted.

- 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 transactor
  - 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

In terms of the structure of the monitor, the following is noted.

- modified monitor is an extension of the original UVM monitor
  - class spi_monitor_a extends spi_monitor;
- instantiates a virtual interface to the BFM
  - virtual interface spi_monitor_bfm_if bfm_vif;
- communicates with the transactor
  - though not really in this case
- does not directly drive the protocol interface
- communicates with the BFM through a task call
  - bfm_vif.run( m_config.m_spi_mode, …);
  - a protocol is developed for passing variables between driver and BFM
- has no notion of clocked or timed processes

In summary, here is how the UVM driver/monitor compares to the UVM-A driver/monitor.

| | UVM | UVM-A |
|---|---|---|
| **Driver** | extended from uvm_driver | extension of existing UVM driver,<br>or,<br>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,<br>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,<br>or,<br>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,<br>but does not directly access protocol interface |
| | potentially monitors interface using clocked processes | no notion of clocked processes |

The next step is to create the new test bench elements which are the driver and monitor BFMs. Essentially, the BFMs contain the sequence/protocol/algorithms that were contained within the original driver/monitor for communicating with the interface.

Specifically, the driver 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
  - i.e. (spi_if spi_itf)
- contains tasks that can be called by the driver
  - i.e. task run( input bit [23:0] m_spi_addr, …);
- drives the protocol interface
  - i.e. spi_itf.spi_m2s.sck = m_spi_mode[1];
- has clocked/timed processes

Specifically, the monitor BFM has the following characteristics.

- is a System Verilog interface
  - interface spi_monitor_bfm_if(spi_if spi_itf);
- the protocol interface is an input
  - i.e. (spi_if spi_itf)
- contains tasks that can be called by the driver
  - task run( input bit [1:0] m_spi_mode, …);
- communicates with the protocol interface
  - i.e. @(posedge spi_itf.clk);
- has clocked/timed processes

It is important to note that the BFMs must be synthesizable within the target accelerator equipment. Therefore these must be written using constructs that are supported by the accelerator vendor. The vendor documentation will provide guidelines on acceptable constructs. Some vendors may refer to this as X-RTL or Extended RTL. X-RTL is a superset of constructs containing Verilog/System-Verilog synthesizable code and 'behavioural' type constructs that the accelerator may support. The supersets of constructs generally offer more flexibility than standard RTL. However, it should be noted that, generally, constructs such as 'fork/join' will not be synthesizable. This can present challenges in translating original UVM driver procedures into a BFM acceptable form.

The goal in building the driver/BFM and monitor/BFM pairs is to ensure that the 'behavioural' side of the environment runs in zero time, while the BFM/DUT side of the environment serves as the 'timed' or 'clocked' area of the test bench. Furthermore, for efficiency, the aim is to minimize transactions between the driver/monitor and the BFMs so that productivity gains from the accelerator can be maximized.

The final step is to stitch these pieces into the environment. Steps to do this include:

- add BFMs to design interface
- declare new BFM elements in environment
- do factory overrides of driver/monitor in test/sequence

Most of the changes to the environment to support acceleration will be transparent to the overall environment. This means that it is possible to flip back and forth between acceleration ready and non-acceleration ready environments easily. This is the beauty and elegance of the factory override mechanism in UVM. The only elements that really change are the driver and monitor. If factory overrides for the driver/monitor are in place for a particular test case / sequence, then the instantiated BFMs will be employed. However, if the driver/monitor factory overrides are not present, then the instantiated BFMs and the associated interface connections will simply be ignored and the test environment will perform as it did before any acceleration related additions were made.

Whether the test environment is targeting acceleration hardware or not, the acceleration ready methodology will function in a pure non-accelerated (aka SW) verification environment. Performance will not be noticeably different from that realized from a standard UVM environment.

Therefore, a verification environment can be architected to be acceleration ready out of the box even if it is unclear whether the project is targeting acceleration or not. By architecting the environment to be acceleration ready, the project can be 'future-proofed' should needs change as the project progresses or as the environment is reused. To repeat the point made in Section 3.3 *Standard drivers/monitors in the library*, the objective is not to advocate for starting with UVM-A vs UVM. However, working towards an acceleration-ready form at the outside may save effort with respect to writing and managing BFMs.

### 4.4 Changes to the env

Since all BFMs are instantiated as virtual interfaces in the new drivers and monitors, they have to be accessed from the top-level environment via the config_db class. The code to configure the hierarchy is shown below in Figure 11. Basically, the user just has to add a few lines of code following the existing code in the UVM environment to include 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_bfm_if_t)::set(this,
"m_axi4_master*", "axi4_master_driver_bfm_vif",
m_config.itf.axi4_master_driver_bfm_itf);

// not all the interfaces are present in this example
```

**Figure 11. UVM-A top-level environment changes**

### 4.5 UVM-A Test Case

The methodology recommends the creation of a separate UVM base test from which all the acceleration friendly tests will extend from. This UVM-A base test will extend from the standard UVM test as shown in Figure 12. As seen below, a task is used to accelerate each agent and the argument is for the agent to become 'acceleration friendly'. Factory overrides are used to replace the standard drivers and monitors with the acceleration friendly versions from the library.

```
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("axi4_master");
    accelerate_agent("ebus");
  endfunction : build_phase

  virtual task accelerate_agent(string name);
    case(name)
      "i2c": begin
  factory.set_type_override_by_type(i2c_driver_switch::get_type(),i2c_driver_switch_a::get_type(),"*");
        factory.set_type_override_by_type(i2c_monitor::get_type(),i2c_monitor_a::get_type(),"*");
      end
      "rst":  begin
        factory.set_type_override_by_type(rst_driver::get_type(),rst_driver_a::get_type(),"*");
      end
      "clk": begin
        factory.set_type_override_by_type(clk_driver::get_type(),clk_driver_a::get_type(),"*");
      end
      "axi4_master":  begin

factory.set_type_override_by_type(axi4_master_driver#()::get_type(),axi4_master_driver_a#()::get_typ
e(),"*");
      end
      "spi":  begin
        factory.set_type_override_by_type(spi_driver::get_type(),spi_driver_a::get_type(),"*");
        factory.set_type_override_by_type(spi_monitor::get_type(),spi_monitor_a::get_type(),"*");
      end
      "ebus": begin
        factory.set_type_override_by_type(ebus_master::get_type(),ebus_driver_a::get_type(),"*");
        factory.set_type_override_by_type(ebus_slave::get_type(),ebus_monitor_a::get_type(),"*");
       end
    endcase
...
```

**Figure 12. Base UVM-A test and factory overrides**

Using this methodology there is no change needed to the existing standard drivers or monitors. The only downside is that the BFM portions for both standard and acceleration friendly versions have to be maintained at the same time. In the case where only one driver and monitor pair has to be replaced per test, factory overrides are the preferred option rather than overriding all the components in one place. A mix of standard and acceleration friendly versions can be used in the verification environment at any time but this approach is not recommended.

The code above shows in details how the override works for both parameterized and non-parameterized classes. Additional care is required when dealing with parameterized classes. It is particularly challenging to figure out the right syntax in System Verilog for a particular scenario specific to a project.

## 5. Methodology challenges

The development of the synthesizable BFMs has been carried out based on a set of standard behavioural constructs (X-RTL) which can be synthesized by the majority of the acceleration tools. However, we still have to prove the transactors built are indeed synthesizable in a particular box. Replacing existing UVM code with synthesizable code was particularly challenging since not all of the constructs used in UVM have equivalents within the range of synthesizable constructs. Some of the choices for the constructs we made earlier did not work and the code had to be re-written to match the spec.

### 5.1 Advantages of using factory overrides

One of the main accomplishments of using this methodology was to reach the conclusion that UVM-A can be deployed after the fact with no existing code having to be touched or replaced to make that happen. This is a huge advantage from past experiences when usually important changes to the code structures had to be made close to the end of the project in order to accommodate some changes such as this. Most of the success of this methodology is attributed to the factory overrides concept in UVM which allows the user to seamlessly overwrite classes and change behaviour without changing the existing code base. The examples given in the paper show how this has been done and we were impressed by the fact that the concept worked right out of the box. This is a ‘management dream’ to have such a feature deployed late in the project with no effect to the already existing code.

In order to have this flexibility the architecture of the standard UVM environment has to allow factory overrides to change behaviour and a strong requirement is to have everything embedded in classes rather than tasks. Building a separate UVM-A library of components was very helpful in achieving this kind of flexibility. However, one of the main disadvantages of using two sets of libraries for drivers and monitors is that the verification team has to maintain both sets of transactors while they implement the same protocol.

# 6. Conclusions

This methodology has been developed to give flexibility to ASIC verification primes and allow freedom during the project development. The UVM-A methodology has to be developed on top of a solid company-wide or project specific standard methodology, and it can gradually become the de-facto methodology, replacing the standard where drivers/monitors BFMs are written inside the run_phase() of each transactor. Using two top levels and completely separating the timed and untimed domains brings more clarity into the verification environment. The coding style with respect to writing sequences has to be changed to push the delays into the timed HDL domain and leave the untimed constructs inside HVL domain, but doing so will create a standardized way of writing tests.

We noticed benefits while developing the methodology, but more development is needed to ensure that all the existing transactors are 'acceleration friendly'. The goal to move to a UVM-A methodology is difficult to reach but not impossible. We plan to change more transactors and continue building the UVM-A library of components to match the existing UVM library. One downside of this exercise is that, for each protocol, the verification team will have to maintain two sets of BFMs: one for the standard methodology and one for the 'acceleration friendly' methodology.

During the development of the methodology we noticed significant opportunities in terms of using designers who have limited UVM knowledge to develop synthesizable BFMs. The next section details our experience of working with designers to achieve the goal of having a complete UVM-A methodology at Ciena.

## 6.1 An Interesting Side-Effect (An Opinion)

The past 15-20 years has seen significant evolution in the area of ASIC/FPGA verification. Fifteen to twenty years ago, verification was largely carried out using Verilog or VHDL test benches without a sense of a standard methodology. In this period it was common for individuals in design teams to carry out both design and verification roles depending on the needs and schedule of the project. The skills required to fill both roles, mainly knowledge of Verilog or VHDL, were largely paralleled and it was relatively easy for an engineer to switch between both roles. But since that period, verification has evolved to a point where the industry has standardized on agreed-upon methodologies. This has provided benefits as designs have become increasingly more complex. But one drawback is that the methodologies have themselves become complex, to the point that fulfilling a verification engineer role has become a speciality. It is now very challenging for a design engineer to transfer into a verification role because the skill sets are now far less paralleled. Working in verification now requires extensive knowledge and experience related to the verification methodologies, which designers rarely have the opportunity to obtain. This has, among other things, made it more difficult to manage a design team as it is nearly impossible to move resources between design and verification roles as in years past and still expect reasonable efficiencies.

In employing the acceleration-ready methodology the structure of the driver/monitor and BFM pairs are largely methodology-agnostic and more closely resemble a form similar to straight System Verilog and RTL. Furthermore, the driver/monitor to BFM/DUT relationship in this methodology removes some of the abstraction that is incorporated into the UVM methodology. This

allows a designer who is not a verification expert to get a better view of the test bench and to see better how the test environment relates to the DUT. The benefit is that engineers who are primarily designers are now more capable of efficiently participating in verification activities, such as BFM development, as these types of tasks require little or no true UVM knowledge, but rather simply require Verilog/System Verilog related skill sets that are in line with RTL design tasks.

## 7. Acknowledgments

The authors would like to thank Gregory King (Ciena) and Bryan Morris (Verilab) for all the valuable feedback and for the time they took to review this paper.

## 8. References

*[1] Verification Methodology Manual for SystemVerilog, Janick Bergeron, Eduard Cerny, Alan Hunter*

*[2] IEEE Standard for SystemVerilog - Std 1800-2005 (Nov. 08/2005)*

*[3] "SystemVerilog Event Regions, Race Avoidance & Guidelines", Clifford E. Cummings, SNUG-2006.*

[4] "UVM & Emulation Methodology Guidelines", Synopsys Confidential

[5] Universal Verification Methodology (UVM) User's Guide, Accellera (release 1.1d)

[6] From Simulation To Emulation – A Fully Reusable UVM Framework, Anoop Saha. Mentor Graphics Corp.

[7] Synopsys Solvnet

[8] "A methodology for vertical Reuse of functionmal verification from subsystem to SoC level with seamless SoC emulation", Pranav Kumar, Digvijaya Pratap SINGH, ST Microelectronics, Greater NOIDA, INDIA

# 9. Appendix

## 9.1 Standard UVM Environment SPI Models

SPI_DRIVER

```
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;
  int                      m_driver_id;

  function new(string name = "spi_driver", uvm_component parent);
    super.new(name, parent);
    trans_driven_ap = new("trans_driven_ap", this);
  endfunction : new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(spi_config)::get(this, "", "config", m_config))
      `uvm_error(get_type_name(), "spi_config not set for this component")
  endfunction : build_phase

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if(!uvm_config_db#(virtual spi_if)::get(this, "", $psprintf("spi_vif[%0d]",m_driver_id), vif))
      `uvm_error(get_type_name(),{$psprintf("virtual interface must be set for:
",get_full_name(),".spi_vif[%0d]",m_driver_id)})
  endfunction: connect_phase

  task run_phase(uvm_phase phase);
    fork
      super.run_phase(phase);
    join_none
    // Initialize Signals Here
    reset_signals();
    // Drive Signals Here
    get_and_drive();

  endtask: run_phase

  function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    //place holder for reports

  endfunction: report_phase

  task reset_signals();
    fork
      begin
        if (m_config.is_active == UVM_ACTIVE) begin
          vif.spi_m2s.sck = m_config.m_spi_mode[1];
          ...
```

```
      end
        ...
    end
    begin
      if (m_config.m_continuous_clk == 1'b1) begin
        forever begin
          repeat (m_config.m_t_half_clk) @(posedge vif.clk);
          vif.spi_m2s.sck = ~vif.spi_m2s.sck;
        end
      end
    end
  join_any;
endtask : reset_signals

//---------------------------------
// get_and_drive
//---------------------------------
task get_and_drive();
  forever
    begin
    // get the reset transaction
    seq_item_port.get_next_item(m_trans);
    `uvm_info(get_type_name(), {"Got a transaction "}, UVM_HIGH);
    if (m_config.m_spi_burst_ena)
      send_a_burst_pkt(m_trans.m_spi_opcode, m_trans.m_spi_addr, m_trans.m_spi_burst_data);
    else
      send_a_pkt(m_trans.m_spi_opcode, m_trans.m_spi_addr, m_trans.m_spi_data);
    seq_item_port.item_done();
    `uvm_info(get_type_name(), {"Done transaction "}, UVM_HIGH);
  end
endtask : get_and_drive


task send_a_pkt(bit[7:0] opcode, bit [23:0] addr, bit [31:0] data);
  `uvm_info(get_type_name(), {"Got a pkt "}, UVM_HIGH);
  m_trans = new();
  m_trans.m_spi_opcode = opcode;
  m_trans.m_spi_addr = addr;
  m_trans.m_spi_data = data;
  send_opcode();
  send_address();
  if (m_trans.m_spi_opcode[0]==m_config.m_opcode_polarity) begin
    send_write_data(m_trans.m_spi_data);
  end
  else begin
    receive_read_data(m_trans.m_spi_data);
    -> m_config.m_rd_data_ready;
  end
  // Write the transaction to the analysis port
  trans_driven_ap.write(m_trans);
  `uvm_info(get_type_name(), {"Finished pkt "}, UVM_HIGH);
endtask : send_a_pkt

task send_a_burst_pkt(bit[7:0] opcode, bit [23:0] addr, bit [7:0] data[$]);
  ...
endtask : send_a_burst_pkt
```

```
  task send_opcode();
    ...
  endtask : send_opcode;

  task send_address();
    ...
  endtask : send_address;

  MORE TASKS
    ...
  END MORE TASKS

  ...
endclass : spi_driver
```

SPI_MONITOR

```
class spi_monitor extends uvm_monitor;
  `uvm_component_utils(spi_monitor)

  uvm_analysis_port #(spi_trans) trans_collected_ap_port;

  spi_config                  m_config;  // handle to cfg
  virtual interface spi_if      vif;  // handle to vif
  spi_trans                   m_new_trans; // collected spi trans
  int                       m_monitor_id;

  function new (string name = "spi_monitor", uvm_component parent);
    super.new(name, parent);
    trans_collected_ap_port = new("trans_collected_ap_port", this);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(spi_config)::get(this, "", "config", m_config))
      `uvm_error(get_type_name(), "spi_config not set for this component")
  endfunction : build_phase

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if(!uvm_config_db#(virtual spi_if)::get(this, "", $psprintf("spi_vif[%0d]",m_monitor_id), vif))
      `uvm_error(get_type_name(),{$psprintf("virtual interface must be set for:
",get_full_name(),".spi_vif[%0d]",m_monitor_id)})
  endfunction : connect_phase

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
      if (m_config.is_active == UVM_PASSIVE)
        collect_transactions(phase);
      @(posedge vif.clk);
    end
  endtask : run_phase

  virtual task collect_transactions(uvm_phase phase);
    // Create a new transaction
    m_new_trans = spi_trans::type_id::create("m_new_trans",,get_full_name());

    // Ignore any transaction when the chip_select isn't asserted
    //wait_for_sample_clk();
    vif.spi_s2m.miso = 1'b0;
    wait_for_start();
    capture_opcode();
    capture_address();
    fork
      begin
        while (1) begin
          if (m_new_trans.m_spi_opcode[0] == m_config.m_opcode_polarity) begin
            capture_write_data();
          end else
            send_read_data();
```

```
      end
    end
    begin
      wait_for_done();
    end
  join_any
  disable fork;
endtask : collect_transactions

//---------------------------------
// wait for chip_select to start
//---------------------------------
task wait_for_start();
  @(posedge vif.clk)
  if (m_config.m_chip_select_polarity)
    @(posedge vif.spi_m2s.cs);
  else
    @(negedge vif.spi_m2s.cs);
endtask : wait_for_start

virtual task capture_opcode();
  ...
endtask : capture_opcode

virtual task capture_address();
  ...
endtask : capture_address

virtual task capture_write_data();
  ...
endtask : capture_write_data

virtual task send_read_data();
  ...
endtask : send_read_data

virtual task wait_for_done();
  ...
endtask : wait_for_done

 ...

endclass: spi_monitor
```

INTERFACE

```
`include "misc_if.sv"

interface proj_name_if();

  clk_if                        clk_itf[NUM_CLK_IF]();
  rst_if                        rst_itf(.clk(clk_itf[0].clk));
  axi4_master_if #(AXI4_USER_WIDTH,
                AXI4_ID_WIDTH,
                AXI4_M_ID_WIDTH,
                AXI4_ADDR_WIDTH,
                AXI4_M_ADDR_WIDTH,
                AXI4_NUM_LANES,
                AXI4_DATA_WIDTH)     axi4_master_itf();
  misc_if                        misc_itf(.clk(clk_itf[0].clk));
  spi_if                        spi_itf[NUM_SPI_IF](.clk(clk_itf[0].clk));
  spi_driver_bfm_if                spi_driver_bfm_itf_0(spi_itf[0]);
  spi_monitor_bfm_if                spi_monitor_bfm_itf_0(spi_itf[0]);

endinterface

//-------------------------------------------------------------------------------
```

## 9.2 UVM-A Environment SPI Models

SPI_DRIVER_A

```
class spi_driver_a extends spi_driver;
  `uvm_component_utils(spi_driver_a)

  virtual interface spi_driver_bfm_if bfm_vif;

  function new(string name = "spi_driver_a", uvm_component parent);
    super.new(name, parent);
    `uvm_info(get_type_name(), $psprintf("SPI_DRIVER_A: new constructor."), UVM_HIGH)
  endfunction : new

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info(get_type_name(), $psprintf("SPI_DRIVER_A: Connect phase."), UVM_HIGH)
    if(!uvm_config_db#(virtual spi_driver_bfm_if)::get(this, "",
$psprintf("spi_driver_bfm_vif[%0d]",m_driver_id), bfm_vif))
      `uvm_fatal(get_type_name(),{$psprintf("virtual interface must be set for:
",get_full_name(),".spi_driver_bfm_vif[%0d]",m_driver_id)})
  endfunction: connect_phase

  task run_phase(uvm_phase phase);
    bit trans_done;
    bit success;
    bit is_active;
    bit data_err;

    `uvm_info(get_type_name(), $psprintf("SPI_DRIVER_A: Run phase."), UVM_HIGH)

    if (m_config.is_active == UVM_ACTIVE) begin
      is_active = 1'b1;
    end
    else begin
      is_active = 1'b0;
    end

    bfm_vif.reset_signals(  is_active,
                            m_config.m_spi_mode,
                            m_config.m_chip_select_polarity,
                            m_config.m_continuous_clk,
                            m_config.m_t_half_clk);

    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,
                        m_trans.m_spi_opcode,
                        m_config.m_spi_mode,
                        m_config.m_rd_data,
                        m_config.m_chip_select_polarity,
                        m_config.m_continuous_clk,
                        m_config.m_t_half_clk,
```

```
                    m_config.m_spi_addr_width,
                    data_err,
                    trans_done);
      if (data_err == 1'b0) begin
        `uvm_info(get_type_name(), {"SPI Driver_a: Read back correct data."}, UVM_HIGH);
      end else begin
        `uvm_error(get_type_name(), {"SPI Driver_a: Read back incorrect data - THAT'S BAD!!!!"});
      end
      seq_item_port.item_done();
    end
  endtask: run_phase

endclass: spi_driver_a
```

SPI_MONITOR_A

```
class spi_monitor_a extends spi_monitor;
  `uvm_component_utils(spi_monitor_a)

  virtual interface spi_monitor_bfm_if  bfm_vif;  // handle to vif

  function new(string name = "spi_monitor_a", uvm_component parent);
    super.new(name, parent);
    `uvm_info(get_type_name(), $psprintf("spi_MONITOR_A: new constructor."), UVM_HIGH)
  endfunction : new

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if(!uvm_config_db#(virtual spi_monitor_bfm_if)::get(this, "",
$psprintf("spi_monitor_bfm_vif[%0d]",m_monitor_id), bfm_vif))
      `uvm_fatal(get_type_name(),{$psprintf("virtual interface must be set for:
",get_full_name(),".spi_monitor_bfm_vif[%0d]",m_monitor_id)})
  endfunction : connect_phase

  task run_phase(uvm_phase phase);
    bit trans_done;
    `uvm_info(get_type_name(), $psprintf("spi_MONITOR_A: In run_phase."), UVM_HIGH)

    if (m_config.is_active == UVM_PASSIVE) begin
        bfm_vif.run(   m_config.m_spi_mode,
                       m_config.m_mem_array,
                       m_config.m_chip_select_polarity,
                       m_config.m_spi_addr_width,
                       m_config.m_opcode_polarity,
                       trans_done);
    end
  endtask : run_phase

endclass: spi_monitor_a
```

SPI_DRIVER_BFM_IF

```
interface spi_driver_bfm_if(spi_if spi_itf);

 import spi_verif_pkg::*;

 // synthesiable i2c xactor code goes in here
 bit          trans_done;
 logic [31:0]  m_rd_data_local;
 bit          data_err_local;

initial
 begin
  // reset signals
   reset_signals(1'b0, 2'b00, 1'b0, 1'b0, 25);
 end

 task reset_signals(   input bit is_active,
                       input bit [1:0] m_spi_mode,
                       input bit m_chip_select_polarity,
                       input bit m_continuous_clk,
                       input int m_t_half_clk);
   if (is_active) begin
     spi_itf.spi_m2s.sck = m_spi_mode[1];
     ...
   end
   else begin
     spi_itf.spi_m2s.sck = 1'bz;
     ...
   end

   if (m_continuous_clk == 1'b1) begin
     forever begin
       repeat (m_t_half_clk) @(posedge spi_itf.clk);
       spi_itf.spi_m2s.sck = ~spi_itf.spi_m2s.sck;
     end
   end
 endtask : reset_signals

 //---------------------------------
 // run
 //---------------------------------
 task run( input bit [23:0]    m_spi_addr,
           input bit [31:0]    m_spi_data,
           input bit [7:0]     m_spi_opcode,
           input bit [1:0]     m_spi_mode,
           input logic [31:0]  m_rd_data,
           input bit           m_chip_select_polarity,
           input bit           m_continuous_clk,
           input int           m_t_half_clk,
           input int           m_spi_addr_width,
           output bit          data_err,
           output bit          trans_done);
   trans_done = 0;
   data_err_local = 1'b0;
   // get the reset transaction
```

```
   $display("[%0d]: SPI Driver_bfm_if: Got a transaction",$time);
   send_a_pkt( m_spi_addr,
               m_spi_data,
               m_spi_opcode,
               m_spi_mode,
               m_rd_data,
               m_chip_select_polarity,
               m_continuous_clk,
               m_t_half_clk,
               m_spi_addr_width,
               data_err_local);
   $display("[%0d]: SPI Driver_bfm_if: Done transaction",$time);
   data_err = data_err_local;
   trans_done = 1;
  endtask : run


  task send_a_pkt(  input bit [23:0]    m_spi_addr,
                    input bit [31:0]    m_spi_data,
                    input bit [7:0]     m_spi_opcode,
                    input bit [1:0]     m_spi_mode,
                    input logic [31:0]  m_rd_data,
                    input bit           m_chip_select_polarity,
                    input bit           m_continuous_clk,
                    input int           m_t_half_clk,
                    input int           m_spi_addr_width,
                    output bit          data_err);
   $display("[%0d]: SPI Driver_bfm_if: Got a pkt",$time);
   send_opcode(m_spi_opcode, m_spi_mode, m_chip_select_polarity, m_continuous_clk, m_t_half_clk);
   send_address(m_spi_addr, m_spi_mode, m_chip_select_polarity, m_continuous_clk, m_t_half_clk,
m_spi_addr_width);
    if (m_spi_opcode[0]==1'b1)
      send_write_data(m_spi_data, m_spi_mode, m_chip_select_polarity, m_continuous_clk,
m_t_half_clk);
    else
      receive_read_data(m_spi_mode, m_rd_data, m_chip_select_polarity, m_continuous_clk,
m_t_half_clk, data_err);
   $display("[%0d]: SPI Driver_bfm_if: Finished pkt",$time);
  endtask : send_a_pkt

  task send_opcode( input bit [7:0] m_spi_opcode,
                    input bit [1:0] m_spi_mode,
                    input bit       m_chip_select_polarity,
                    input bit       m_continuous_clk,
                    input int       m_t_half_clk);
    ...
  endtask : send_opcode

  task send_address(input bit [23:0]    m_spi_addr,
                    input bit [1:0]     m_spi_mode,
                    input bit           m_chip_select_polarity,
                    input bit           m_continuous_clk,
                    input int           m_t_half_clk,
                    input int           m_spi_addr_width);
    ...
  endtask : send_address
```

```
  task send_write_data( input bit [31:0]    m_spi_data,
                        input bit [1:0]     m_spi_mode,
                        input bit           m_chip_select_polarity,
                        input bit           m_continuous_clk,
                        input int           m_t_half_clk);
    ...
  endtask : send_write_data

  task receive_read_data(   input bit [1:0]     m_spi_mode,
                            input logic [31:0]  m_rd_data,
                            input bit           m_chip_select_polarity,
                            input bit           m_continuous_clk,
                            input int           m_t_half_clk,
                            output bit          data_err);
    ...
  endtask : receive_read_data


endinterface : spi_driver_bfm_if
```

SPI_MONITOR_BMF_IF

```
interface spi_monitor_bfm_if(spi_if spi_itf);

  import spi_verif_pkg::*;

  // synthesiable spi xactor code goes in here
  bit          m_read_done;
  bit [23:0]    m_spi_addr;
  bit [31:0]    m_spi_data;
  bit [7:0]     m_spi_opcode; // bit 0 is read/write


  task run( input bit [1:0]     m_spi_mode,
            inout bit [31:0]    m_mem_array[MAX_MEM_SIZE],  //1024
            input bit           m_chip_select_polarity,
            input int           m_spi_addr_width,
            input bit           m_opcode_polarity,
            output bit          trans_done);

    // Create a new transaction
    forever begin
      // Ignore any transaction when the chip_select isn't asserted
      spi_itf.spi_s2m.miso = 1'b0;
      wait_for_start(m_chip_select_polarity);
      capture_opcode(m_spi_mode);
      capture_address(m_spi_addr_width, m_spi_mode);
      if (m_spi_opcode[0] == m_opcode_polarity) begin
        capture_write_data(m_mem_array, m_spi_mode);
        $display("[%0d]: SPI Monitor_bfm_if: OPCODE Polarity set to: %0b",$time, m_opcode_polarity);
      end else begin
        send_read_data(m_mem_array, m_spi_mode);
      end
      wait_for_done(m_chip_select_polarity);
    end
  endtask: run

  task wait_for_start(input bit m_chip_select_polarity);
    @(posedge spi_itf.clk)
    if (m_chip_select_polarity)
      @(posedge spi_itf.spi_m2s.cs);
    else
      @(negedge spi_itf.spi_m2s.cs);
    $display("[%0d]: SPI Monitor_bfm_if: Start detected",$time);
  endtask : wait_for_start

  task capture_opcode(input bit [1:0] m_spi_mode);
    ...
  endtask : capture_opcode

  task capture_address( input int m_spi_addr_width,
    ...
  endtask : capture_address

  task capture_write_data(  inout bit [31:0] m_mem_array[MAX_MEM_SIZE],
    ...
```

```
  endtask : capture_write_data

  task send_read_data(  inout bit [31:0] m_mem_array[MAX_MEM_SIZE],
                        input bit [1:0] m_spi_mode);
    ...
  endtask : send_read_data

  task wait_for_done(input bit m_chip_select_polarity);
    ...
  endtask : wait_for_done

  ...

endinterface: spi_monitor_bfm_if
```