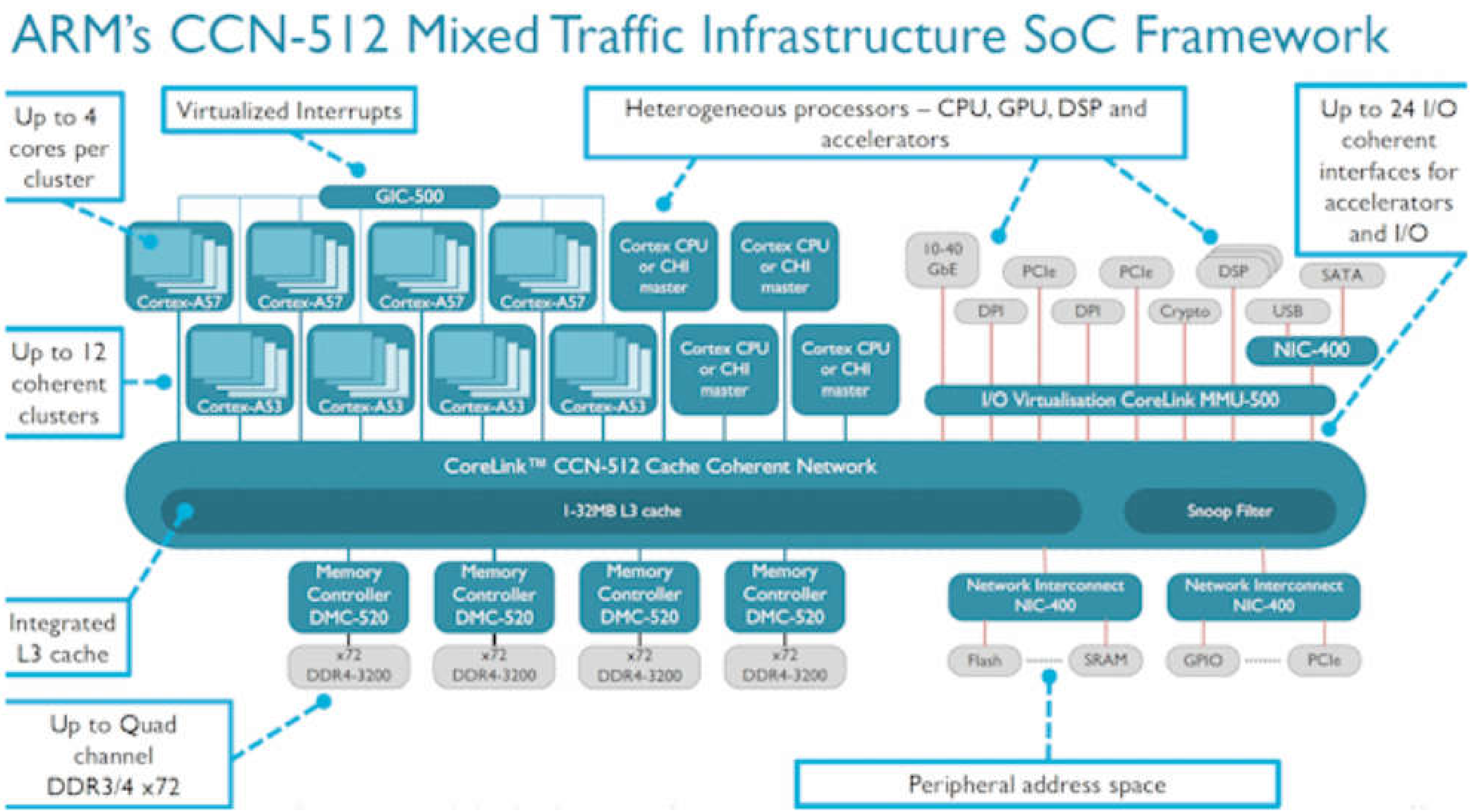
UVM (Universal Verification Methodology)
For SW Engineers

Tuan Nguyen-viet

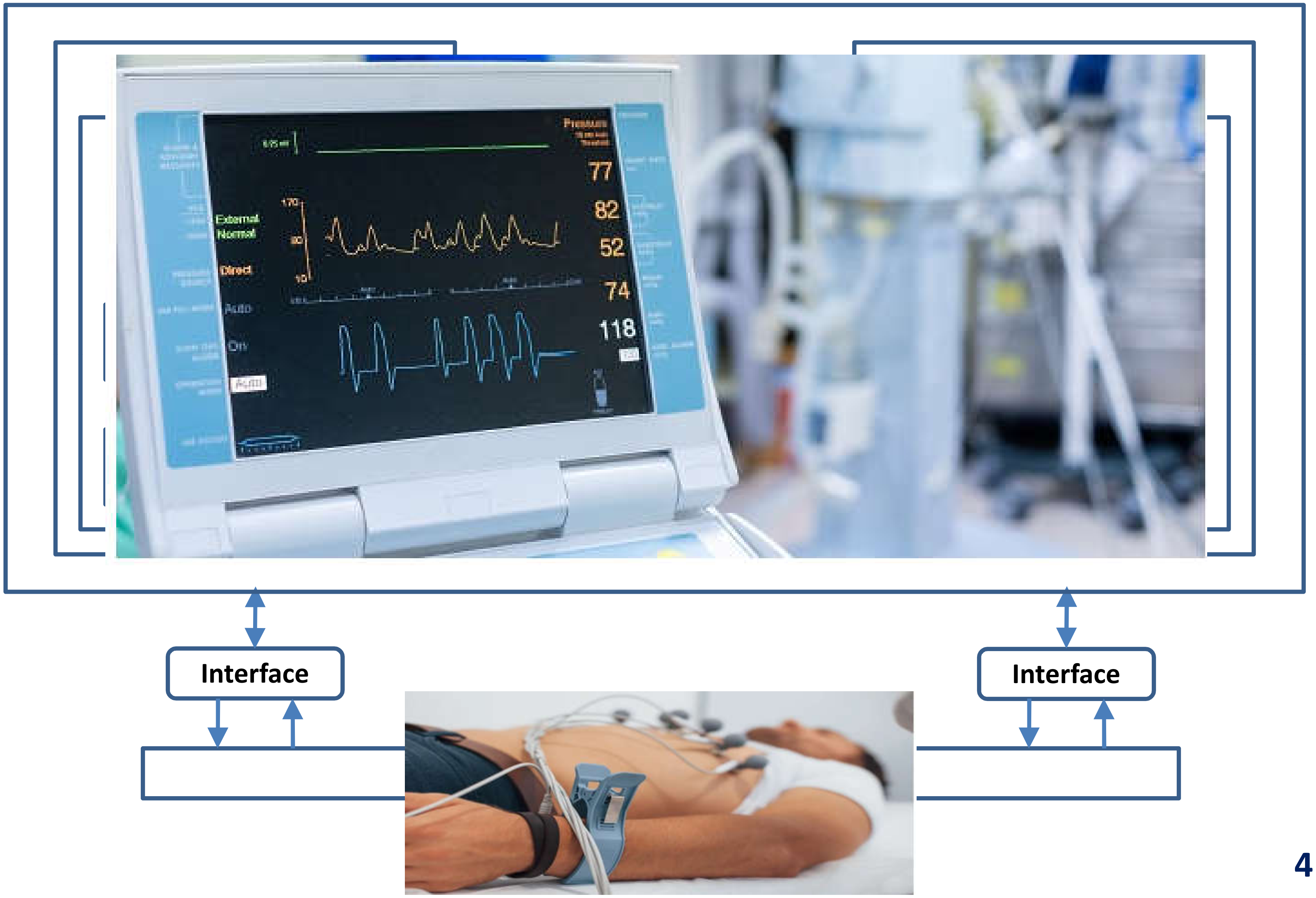
Challenges of verifying complex systems – An Example


ARM's CCN-512 Mixed Traffic Infrastructure SoC Framework
Up to 4 cores per cluster
Virtualized Interrupts
Heterogeneous processors – CPU, GPU, DSP and accelerators
Up to 24 I/O coherent interfaces for accelerators and I/O
GIC-500
Cortex-A57
Cortex-A53
Cortex CPU or CHI master
10-40 GbE
PCIe
DSP
SATA
DPI
Crypto
USB
NIC-400
I/O Virtualisation CoreLink MMU-500
Up to 12 coherent clusters
CoreLink™ CCN-512 Cache Coherent Network
1-32MB L3 cache
Snoop Filter
Memory Controller DMC-520
x72 DDR4-3200
Network Interconnect NIC-400
Flash
SRAM
GPIO
Integrated L3 cache
Up to Quad channel DDR3/4 x72
Peripheral address space

Challenges of verifying complex systems (2)

- Typical **processor development** from scratch could be **100s** of engineering years
  - Requires parallel developments across multiple sites,
    - and it takes a large team to verify a processor
- The typical method is to divide and conquer,
  - partitioning the whole CPU into smaller units
    - and verify those units,
    - then reuse the checkers and stimulus at a higher level
- The challenges are numerous
  - Reuse of code becomes an absolute key to avoid duplication of work
  - It is essential to have the ability to integrate an external IP
  - This requires rigorous planning, code structure, & lockstep development
  - Standardization becomes a key consideration
- ➔ UVM can help solve this

Key Components of a UVM Testbench



UVM Testbench Top

- All verification components, interfaces and **DUT**
  - are instantiated in a **top** level module called **testbench**.

- It is a static container to hold everything required to be simulated
  - and becomes the root node in the hierarchy.
    - This is usually named **tb** or **tb_top**
      - although it can assume any other name.

- **Simulators** (e.g. NCSIM, Questasim, etc.) typically need to know the **top** level module
  - so that each can
    - analyze components within the **top** module
    - and elaborate the design hierarchy.

UVM Testbench Top

- The **testbench top** is a static container
  - that has an instantiation of **DUT** and **interfaces**.

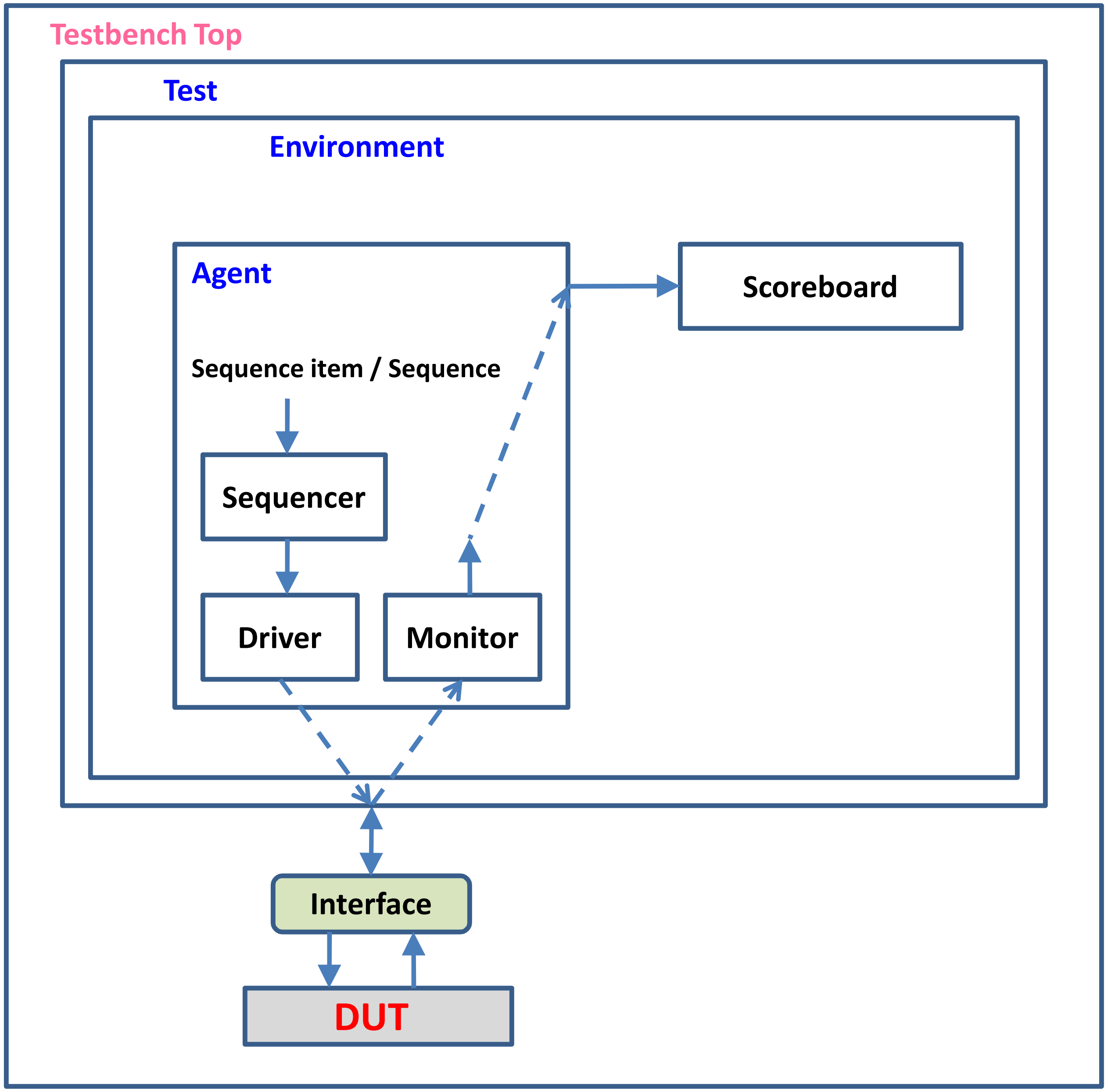
- The **interface** instance connects with **DUT** signals in the **testbench top**.

- The clock is generated and initially reset is applied to the DUT.
  - It is also passed to the **interface** handle.

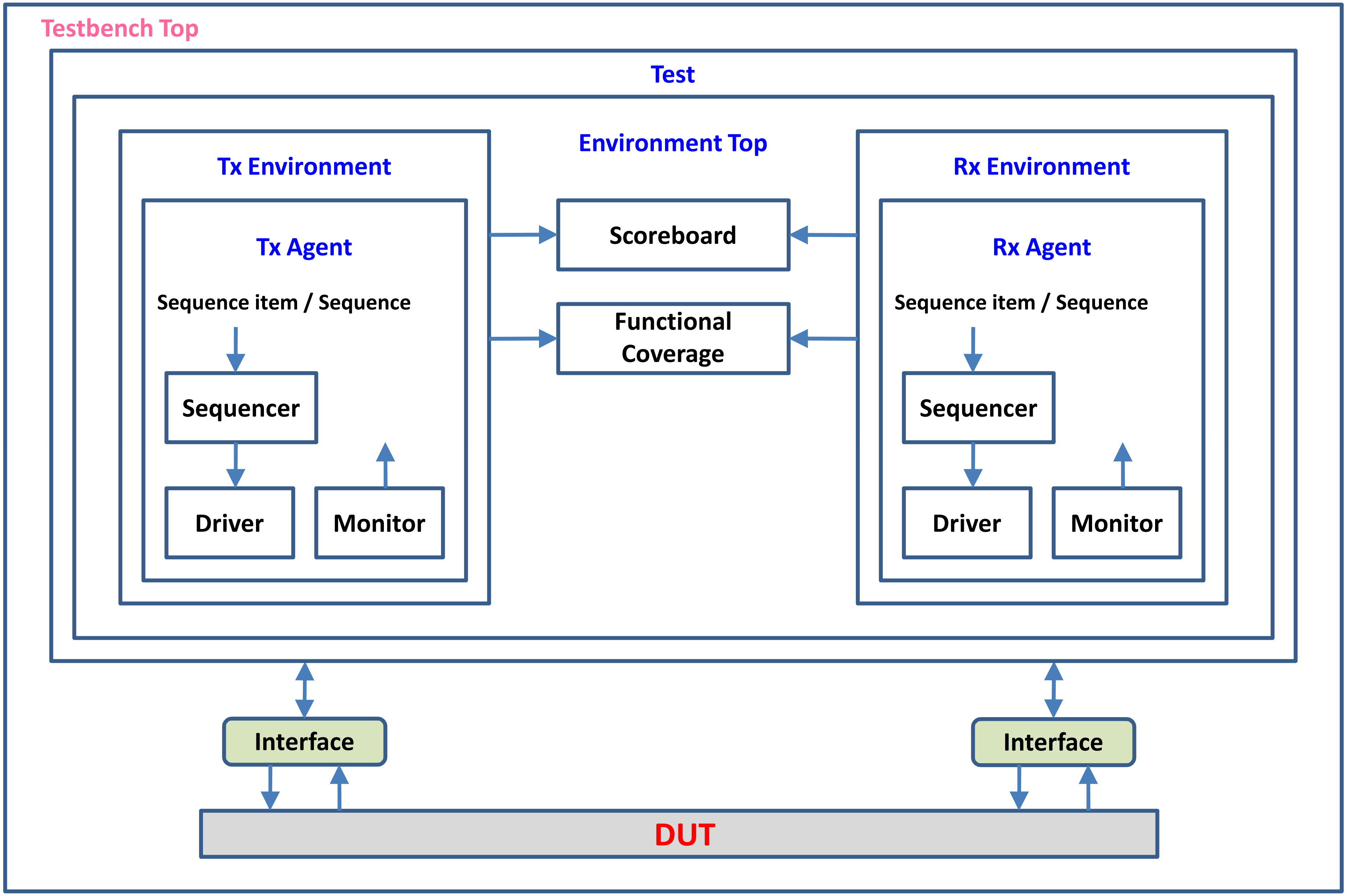
- An **interface** is stored in the **uvm_config_db**
  - using the **set** method
    - and it can be retrieved down the hierarchy
      - using the **get** method.
- UVM testbench top is also used to trigger a test
  - using **run_test()** call.

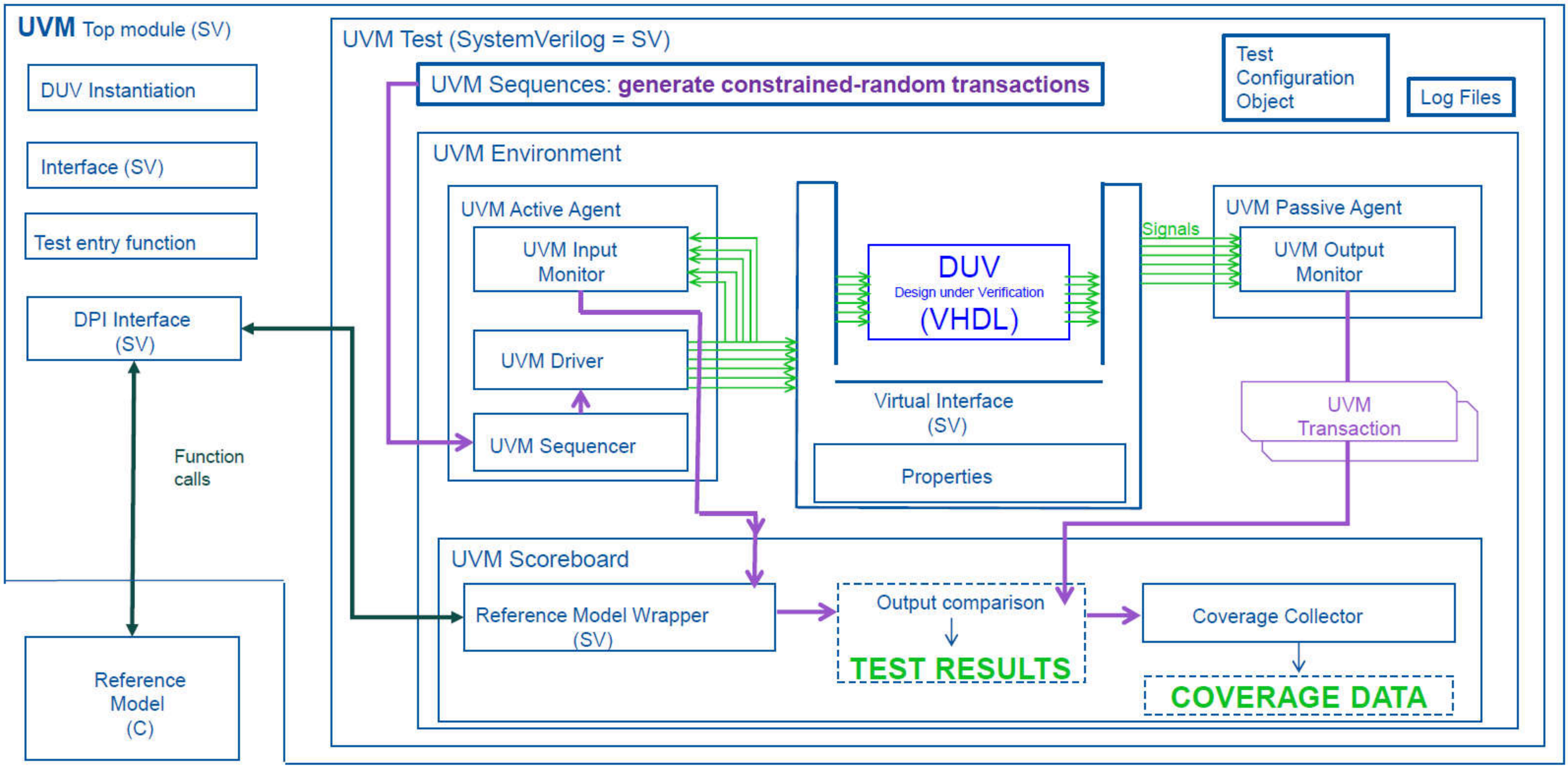
- REF: https://vlsiverify.com/uvm/uvm-testbench-top/

UVM - Simple Architecture w/ Single Agent



Key Components of a UVM Testbench




UVM Top module (SV)
DUV Instantiation
Interface (SV)
Test entry function
DPI Interface (SV)
Function calls
Reference Model (C)
UVM Test (SystemVerilog = SV)
UVM Sequences: generate constrained-random transactions
Test Configuration Object
Log Files
UVM Environment
UVM Active Agent
UVM Input Monitor
UVM Driver
UVM Sequencer
DUV
Design under Verification
(VHDL)
Virtual Interface (SV)
Properties
Signals
UVM Passive Agent
UVM Output Monitor
UVM Transaction
UVM Scoreboard
Reference Model Wrapper (SV)
Output comparison
TEST RESULTS
Coverage Collector
COVERAGE DATA

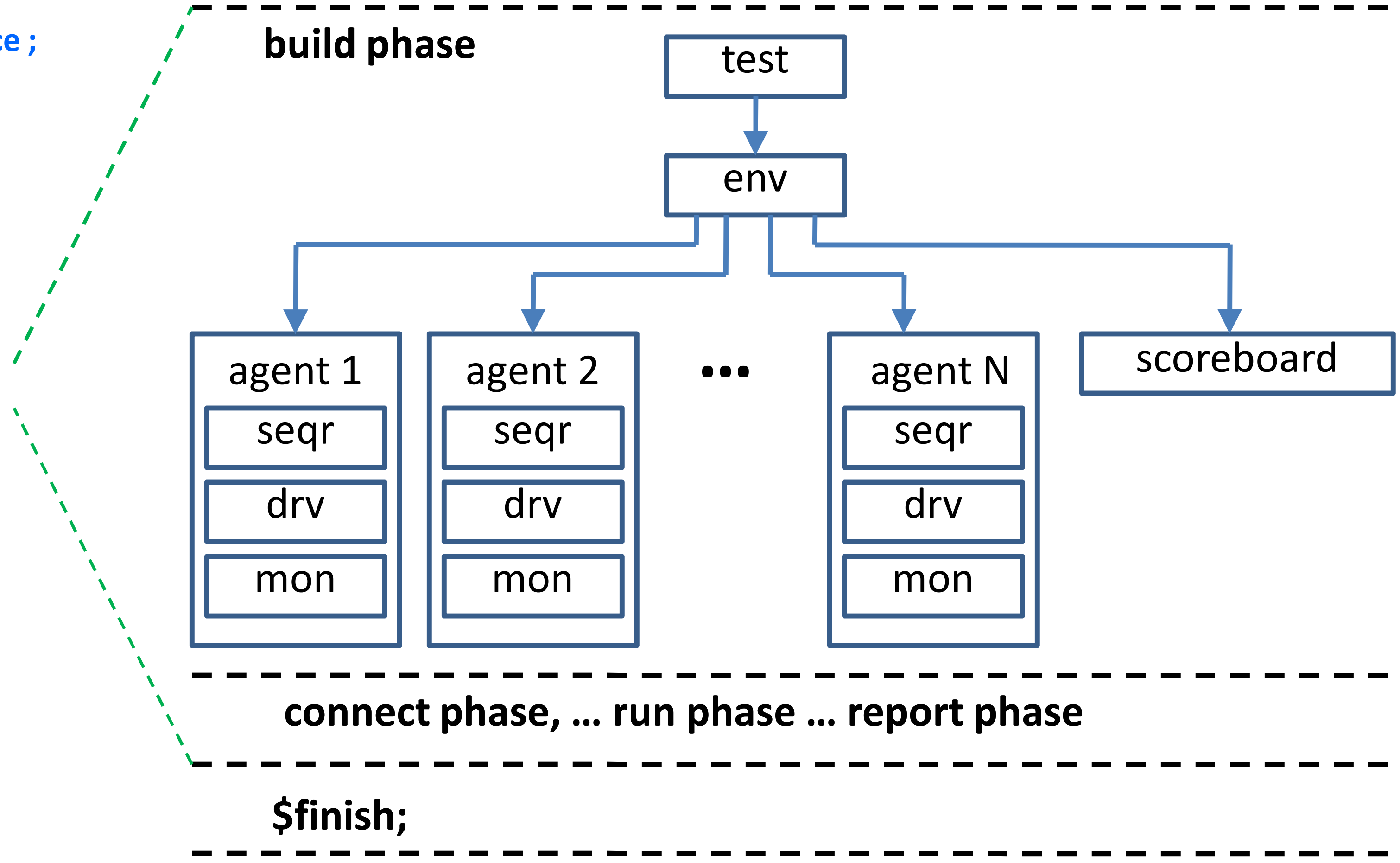
UVM tb top

- Typical **Testbench_top** contains,
  - **DUT** instance
  - **interface** instance
  - **run_test()** method
  - virtual interface set config_db
  - clock and reset generation logic
  - wave dump logic

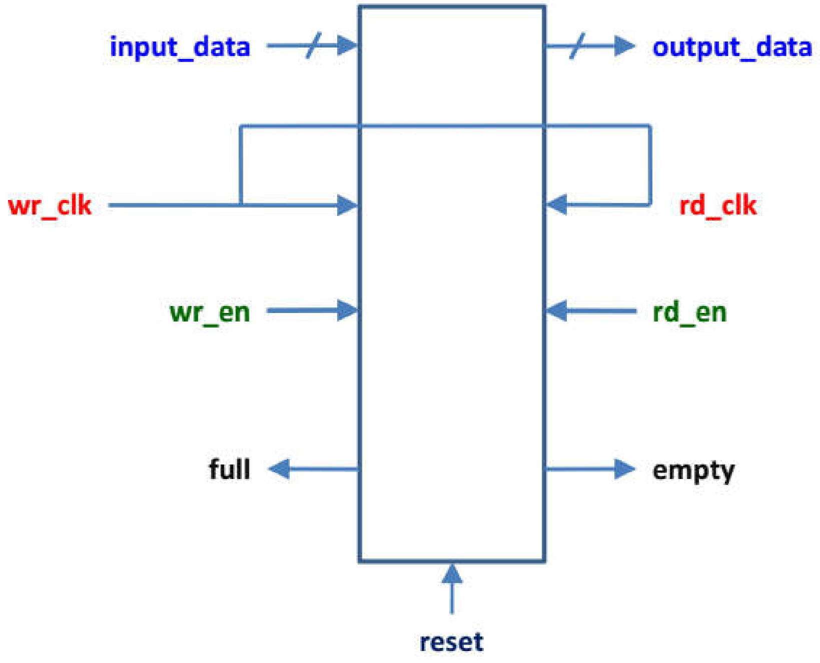
UVM Phases

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module testbench_top;
        //
        //
        //
        Interface instance ;
        DUT instance;
        //
        //
        //
        initial begin
            //
            run_test ();
        end
        //
        //
        //


endmodule
```



```
module testbench_top;
   //clock and reset signal declaration
 bit clk;
 bit reset;
   //clock generation
 always #5 clk = ~clk;
   //reset Generation
 initial begin
   reset = 1;
   #5 reset =0;
 end
   //creating instance of interface, in order to connect DUT and testcase
 sync_fifo_if intf(clk,reset);
   //DUT instance, interface signals are connected to the DUT ports
 sync_fifo DUT (
   .clk(intf.clk),
   .reset(intf.reset),
   .full(intf.full),
   .empty(intf.empty),
   .wr_en(intf.wr_en),
   .rd_en(intf.rd_en),
   .input_data(intf.wdata),
   .output_data(intf.rdata)
  );
  //enabling the wave dump
 initial begin
   uvm_config_db#(virtual sync_fifo_if)::set(uvm_root::get(),"*",”sync_fifo_intf",intf);
   $dumpfile("dump.vcd"); $dumpvars;
 end
 initial begin
   run_test();
 end
endmodule
```



UVM TestBench Architecture

- To maintain **uniformity** in naming the components/objects,
  - all the component/object name’s are starts with **sync_fifo_***.

Sequence Item/Transaction

UVM TB Architecture: Sequence Item/Transaction

- **Sequence Item** is the same as a **Transaction**
  - Examples: packet, AXI transaction, pixel
- Fields required to generate the **stimulus** are declared in the **sequence item**.

1. sequence item is written by extending uvm_sequence_item,

```
class sync_fifo_seq_item extends uvm_sequence_item;
  //Utility macro
  `uvm_object_utils(sync_fifo_seq_item)
  //Constructor
  function new(string name = “sync_fifo_seq_item");
    super.new(name);
  endfunction
endclass
```

UVM TB Architecture: Sequence Item/Transaction (2)

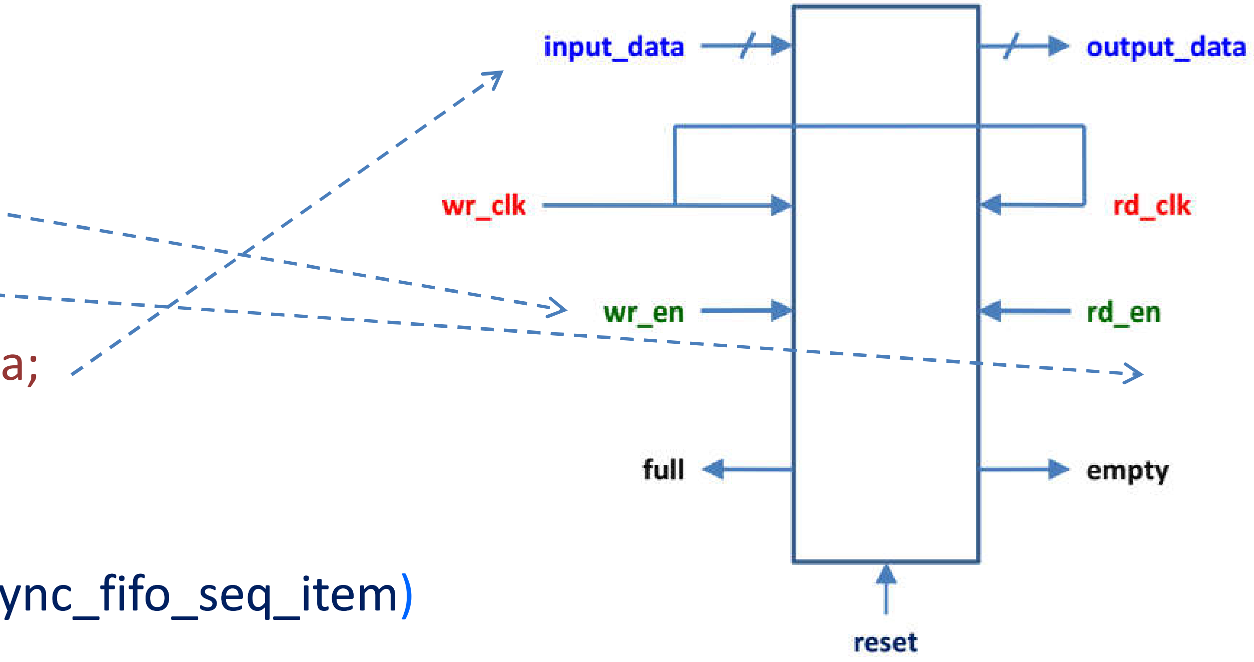
2. Declaring the fields in **sync_fifo_seq_item**,

```
class sync_fifo_seq_item extends uvm_sequence_item;
 //data and control fields
 bit      full;
 bit      empty;
 bit      wr_en;
 bit      rd_en;
 bit [15:0] wdata;
 bit [15:0] rdata;
 //Utility macro
 `uvm_object_utils(sync_fifo_seq_item)
 //Constructor
 function new(string name = " sync_fifo_seq_item ");
   super.new(name);
 endfunction
endclass
```

UVM TB Architecture: Sequence Item/Transaction (3)

3. To generate the random stimulus, declare the fields as rand.

```
class sync_fifo_seq_item extends uvm_sequence_item;
  //data and control fields
  bit      full;
  bit      empty
  rand bit      wr_en;
  rand bit      rd_en;
  rand bit [15:0] wdata;
  bit [15:0] rdata;
  //Utility macro
  `uvm_object_utils(sync_fifo_seq_item)
  //Constructor
  function new(string name = " sync_fifo_seq_item ");
    super.new(name);
  endfunction
endclass
```



UVM TB Architecture: Sequence Item/Transaction (4)

4. In order to use the uvm_object methods (copy, compare, pack, unpack, record, print, and etc),

all the fields are registered to **uvm_field_*** macros.,

```
class sync_fifo_seq_item extends uvm_sequence_item;
 //data and control fields
    bit full, empty;
 rand bit      wr_en;
 rand bit      rd_en;
 rand bit [7:0] wdata;
     bit [7:0] rdata;
 //Utility and Field macros,
 `uvm_object_utils_begin(sync_fifo_seq_item)
   `uvm_field_int(wr_en, UVM_ALL_ON)
   `uvm_field_int(rd_en, UVM_ALL_ON)
   `uvm_field_int(wdata, UVM_ALL_ON)
 `uvm_object_utils_end
 //Constructor
 function new(string name = “sync_fifo_seq_item");
   super.new(name);
 endfunction
endclass
```

UVM TB Architecture: Sequence Item/Transaction (5)

5. Either write or read operation will be performed at once,

   so the **constraint** is added to generate wr_en and rd_en.

```
class sync_fifo_seq_item extends uvm_sequence_item;
 //data and control fields
    bit full, empty;
 rand bit      wr_en;
 rand bit      rd_en;
 rand bit [7:0] wdata;
     bit [7:0] rdata;
 //Utility and Field macros,
 `uvm_object_utils_begin(sync_fifo_seq_item)
   `uvm_field_int(wr_en, UVM_ALL_ON)
   `uvm_field_int(rd_en, UVM_ALL_ON)
   `uvm_field_int(wdata, UVM_ALL_ON)
 `uvm_object_utils_end
 //Constructor
 function new(string name = “sync_fifo_seq_item");
   super.new(name);
 endfunction
//constaint, to generate any one among write and read
 constraint wr_rd_c { wr_en != rd_en; };
endclass
```

UVM TB Architecture: Sequence Item/Transaction (6)

Complete **sync_fifo_seq_item** code.

```
class sync_fifo_seq_item extends uvm_sequence_item;
 //data and control fields
    bit      full;
    bit      empty;
 rand bit      wr_en;
 rand bit      rd_en;
 rand bit [7:0] wdata;
     bit [7:0] rdata;
  //Utility and Field macros,
 `uvm_object_utils_begin(sync_fifo_seq_item)
  `uvm_field_int(wr_en,UVM_ALL_ON)
  `uvm_field_int(rd_en,UVM_ALL_ON)
  `uvm_field_int(wdata,UVM_ALL_ON)
 `uvm_object_utils_end
  //Constructor
 function new(string name = “sync_fifo_seq_item");
  super.new(name);
 endfunction
  //constaint, to generate any one among write and read
 constraint wr_rd_c { wr_en != rd_en; };
endclass
```

Sequence

UVM TB Architecture: Sequence

- UVM Sequence is a collection/list of UVM Sequence Items.
- UVM Sequence generates the stimulus
  - and sends to UVM Driver via UVM Sequencer.
- A UVM Agent can have any number of UVM Sequences.

UVM TB Architecture: Sequence (2)

1. A sequence is written by extending the uvm_sequence,

```
class sync_fifo_sequence extends uvm_sequence # (sync_fifo_seq_item);
 `uvm_sequence_utils(sync_fifo_sequence, sync_fifo_sequencer)
 //Constructor
 function new(string name = “sync_fifo_sequence");
  super.new(name);
 endfunction
endclass
```

UVM TB Architecture: Sequence (3)

2. Logic to generate and send the sequence_item is added inside the body() method,

```
class sync_fifo_sequence extends uvm_sequence # (sync_fifo_seq_item);
 `uvm_sequence_utils(sync_fifo_sequence, sync_fifo_sequencer)
 //Constructor
 function new(string name = “sync_fifo_sequence");
   super.new(name);
 endfunction

 virtual task body();
   req = sync_fifo_seq_item ::type_id::create("req");
   wait_for_grant();
   req.randomize();
   send_request(req);
   wait_for_item_done();
 endtask
endclass
```

Sequencer

UVM TB Architecture: Sequencer

- Sequencer is written by extending uvm_sequencer,
  - there is no extra logic required to be added in the sequencer.

1. sequence item is written by extending uvm_sequence_item,

```
class sync_fifo_sequencer extends uvm_sequencer #(sync_fifo_seq_item);
  //Utility macro
  `uvm_object_utils(sync_fifo_sequencer)
  //Constructor
  function new(string name, uvm_component parent);
    super.new(name);
  endfunction
endclass
```

UVM TB Architecture: Sequencer (2)

- A **UVM sequencer** connects a UVM Sequence to the UVM Driver
  - It sends a transaction from the Sequence to the Driver
  - It sends a response from the Driver to the Sequence
- **Sequencer** can also arbitrate between multiple sequences and send a chosen transaction to the Driver
- Provides the following methods:
  - send_request (),
  - get_response ()

Driver

UVM TB Architecture: Driver

- A **UVM driver** is responsible for decoding a transaction obtained from the **Sequencer**
- It is responsible for driving the **DUT** interface signals
- It understands the pin level protocol and the timing relationships
- Driver receives the stimulus from **Sequence** via **Sequencer** and drives on interface signals.

UVM TB Architecture: Driver (2)

1. Driver is written by extending the **uvm_driver**,

```
class sync_fifo_driver extends uvm_driver #(sync_fifo_seq_item);
  `uvm_component_utils(sync_fifo_driver)
  // Constructor
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new
endclass : mem_driver
```

UVM TB Architecture: Driver (3)

2. Declare the virtual interface,

```
// Virtual Interface
virtual sync_fifo_if vif;
```

UVM TB Architecture: Driver (4)

3. Get the interface handle using get config_db,

```
if(!uvm_config_db # (virtual sync_fifo_if)::get(this, "", "vif", vif))
    `uvm_fatal ("NO_VIF", {"virtual interface must be set for:", get_full_name(),".vif"});
```

UVM TB Architecture: Driver (5)

4. Adding the get config_db in the build_phase,

```
function void build_phase(uvm_phase phase);
   super.build_phase(phase);
    if(!uvm_config_db#(virtual sync_fifo_if)::get(this, "", "vif", vif))
      `uvm_fatal("NO_VIF",{"virtual interface must be set for:", get_full_name(),".vif"});
endfunction: build_phase
```

UVM TB Architecture: Driver (6)

5. Add driving logic, get the seq_item and drive to DUT signals,

```
// run phase
 virtual task run_phase(uvm_phase phase);
  forever begin
  seq_item_port.get_next_item(req);
   //...
   //.. driving logic ..here
   //...
  seq_item_port.item_done();
  end
 endtask : run_phase
```

Monitor

UVM TB Architecture: Monitor

- Monitor's responsibility is to observe communication on the **DUT** interface
- A Monitor can include a protocol checker that can immediately find any pin level violations of the communication protocol
- Monitor samples the **DUT** signals through the virtual interface and converts the signal level activity to the transaction level.
- **UVM Monitor** is responsible for creating a transaction based on the activity on the interface
  - This transaction is consumed by various testbench components for checking and functional coverage
  - **Monitor** communicates with other testbench components using UVM Analysis ports

UVM TB Architecture: Monitor (2)

1. The **Monitor** is written by extending the **uvm_monitor**,

```
class sync_fifo_monitor extends uvm_monitor;
  `uvm_component_utils(sync_fifo_monitor)
  // new - constructor
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new
endclass : sync_fifo_monitor
```

UVM TB Architecture: Monitor (3)

2. Declare virtual interface,

```
// Virtual Interface
virtual sync_fifo_if vif;
```

UVM TB Architecture: Monitor (4)

3. Connect interface to Virtual interface by using get method,

```
function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   if(!uvm_config_db#(virtual sync_fifo_if)::get(this, "", "vif", vif))
     `uvm_fatal("NOVIF",{"virtual interface must be set for: ", get_full_name(),".vif"});
  endfunction: build_phase
```

UVM TB Architecture: Monitor (5)

4. Declare Analysis port,

```
uvm_analysis_port #(sync_fifo_seq_item) item_collected_port;
```

UVM TB Architecture: Monitor (6)

5. Declare seq_item handle, Used as a place holder for sampled signal activity,

```
sync_fifo_seq_item  trans_collected;
```

UVM TB Architecture: Monitor (7)

6. Add Sampling logic in run_phase,
   - sample the interface signal and assign to trans_collected handle
   - sampling logic is placed in the forever loop

```
// run phase
 virtual task run_phase(uvm_phase phase);
   forever begin
    //sampling logic
     @(posedge vif.MONITOR.clk);
     wait(vif.monitor_cb.wr_en || vif.monitor_cb.rd_en);
     trans_collected.full = vif.monitor_cb.full;
     trans_collected.empty = vif.monitor_cb.empty;
    if(vif.monitor_cb.wr_en) begin
     trans_collected.wr_en = vif.monitor_cb.wr_en;
     trans_collected.wdata = vif.monitor_cb.wdata;
     trans_collected.rd_en = 0;
     @(posedge vif.MONITOR.clk);
    end
    if(vif.monitor_cb.rd_en) begin
     trans_collected.rd_en = vif.monitor_cb.rd_en;
     trans_collected.wr_en = 0;
     @(posedge vif.MONITOR.clk);
     @(posedge vif.MONITOR.clk);
     trans_collected.rdata = vif.monitor_cb.rdata;
end
   end
 endtask : run_phase
```

UVM TB Architecture: Monitor (8)

7. After sampling, by using the write method send the sampled transaction packet to the Scoreboard,

```
item_collected_port.write(trans_collected);
```

Agent

UVM TB Architecture: Agent

- An **Agent** is a container class contains a **Driver**, a **Sequencer**, and a **Monitor**.
- **UVM Agent** is responsible for connecting the sequencer, driver and the monitor
- It provides analysis ports for the monitor to send transactions to the scoreboard and coverage
- It provides the ability to disable the sequencer and driver; this will be useful when an actual DUT is connected

Scoreboard

UVM TB Architecture: Scoreboard

- **Scoreboard** receives the transaction from the **Monitor** and compares it with the reference values.
- **Scoreboard** is one of the trickiest and most important verification components
- **Scoreboard** is an independent implementation of specification
  - It takes in transactions from various monitors in the design, applies the inputs to the independent model and generates an expected output
  - It then compares the actual and the expected outputs

- A typical **Scoreboard** is a queue implementation of the modeled outputs resulting in a pop of the latest result when the actual **DUT** output is available
- A **Scoreboard** also has to ensure that the timing of the inputs and outputs is well managed to avoid false fails

Environment

UVM TB Architecture: Environment

- The **Environment** is the container class,
  - It contains one or more **Agents**, as well as other components such as the **Scoreboard**, top-level **Monitor**, and **checker**.
- It means that
  - It instantiates and connects:
    - all the Agents
    - all the Scoreboards
    - all the functional coverage models
- And thus
  - The **Environment** is responsible for managing various components in the testbench

Test

UVM TB Architecture: Test

- The **Test** defines the test scenario for the testbench.

- **uvm_test** is responsible for
  - creating the **Environment**
  - controlling the type of test we want to run
  - providing configuration information to all the components through the **Environment**

Thank You