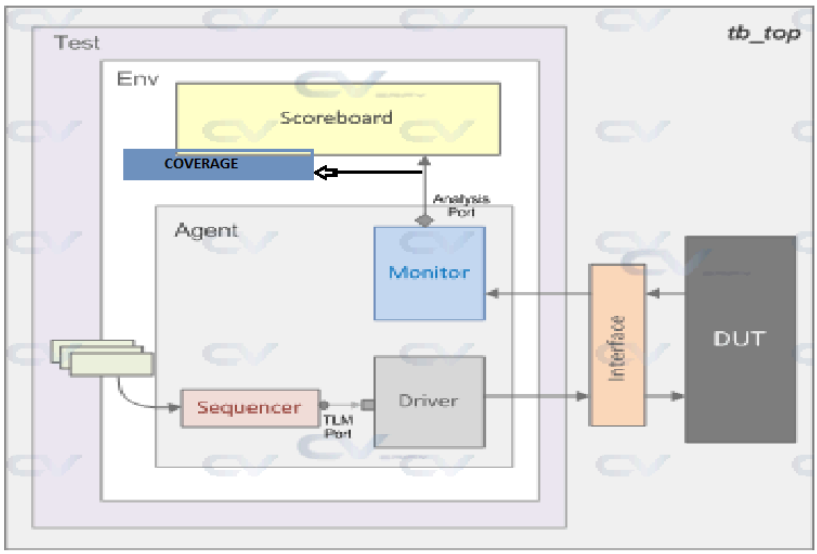
ECE-593 Fundamentals of Pre-Silicon Validation
Final UVM TestBench Hierarchy

-By: Group 3

UVM TestBench Hierarchy:



SEQUENCE ITEM:

Sequence item is a collection of all the inputs that are randomized and defined according to the design needs and requirements. All the inputs that need to be randomized are taken from the DUT. The actual randomization of all the inputs happens in the Sequence and sent through the sequencer to the driver.

```
constraint wr {w_en dist {1:/50, 0:/(50)};}
    function void post_randomize();
    r_en = ~w_en;
    endfunction
```

SEQUENCE:

A sequence is a collection of sequence items and control logic that defines a specific verification scenario. Sequences control the generation of stimulus to drive the DUT and monitor its response. Sequences can be hierarchical and modular, allowing for easy reuse and scalability. In our uvm test bench environment.

We have taken two different sequences for two different operations which depend upon read and write enable. The fifo_sequence_wr and sequence_fifo_rd are the two sequences for read and write operations respectively.

```
int num=4;

    task body();
        for (int i=0; i<num;i++) begin
            `uvm_info("FIFO_SEQUENCE","INSIDE THE TASK BODY!",UVM_LOW)
            fifo_pkt = fifo_seq_item#(8,3)::type_id ::create ("fifo_pkt");
            start_item(fifo_pkt);
            fifo_pkt.randomize() with {w_en==1 & r_en==0;};
                        `uvm_info("SEQ",$sformatf("Generate new item:
%s",fifo_pkt.convert2str()),UVM_LOW)
            fifo_pkt.print();
            finish_item(fifo_pkt);
      `uvm_info("SEQ",$sformatf(" Done Generate new item: %d",i),UVM_LOW)
        end
   `uvm_info("SEQ",$sformatf(" Done Generation of items: %d",num),UVM_LOW)
    endtask
```

SEQUENCER:

The sequencer controls the generation and sequencing of stimulus transactions. It coordinates with the testbench to select and execute sequences based on the test requirements. The sequencer communicates with the driver to transmit stimulus to the DUT and receives responses from the monitor.

Build phase and connect phase are the phases of UVM used in the Sequencer to connect to the driver and pass on the item from sequencer to the driver.

```
// Build phase
```

```
function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("SEQUENCER", "Build Phase!",UVM_LOW)
    endfunction
    //connect phase
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        `uvm_info("SEQUENCER", "Connect Phase!",UVM_LOW)
    endfunction
```

DRIVER:

The driver is responsible for converting transaction-level stimulus generated by the sequencer into signals or transactions suitable for transmission to the DUT. It drives the actual interface signals to stimulate the DUT.

There are Build Phase, Connect Phase and Run Phase responsible for connecting and responsible for performing different operations and for example: connecting with the DUT in build phase and driving the signals from DUT in run phase inside the driver.

```
task run_phase(uvm_phase phase);
        super.run_phase(phase);
        `uvm_info("DRIVER CLASS", "Inside run Phase!",UVM_LOW)
        forever
         begin
            drv_pkt = fifo_seq_item#(8,3)::type_id::create("drv_pkt");
            seq_item_port.get_next_item(drv_pkt);
            drive(drv_pkt);
            seq_item_port.item_done();
         end
    endtask
```

AGENT:

An agent represents a logical entity responsible for interfacing with a specific part of the DUT. It typically consists of a driver, monitor, sequencer, and possibly a scoreboard. Agents abstract away the details of the interface protocol and provide a clean interface for stimulus generation and response monitoring.

Build phase and connect phase and run phase along with creating a new constructor for the agent class which calls the create method for the sequencer , driver, and monitor using the factory registration methods from the uvm_object class in build phase and also used the connect phase to connect the port of driver to the sequencer export as shown in the figure below.

```
        // Build Phase
        s0 = sequencer::type_id::create("s0",this);
        d0 = driver::type_id::create("d0",this);
        mon = monitor:: type_id::create("mon", this);
        //connect phase
        `uvm_info("AGENT CLASS", "CONNECT PHASE!",UVM_LOW)
        d0.seq_item_port.connect(s0.seq_item_export);
```

MONITOR:

Monitoring the activity on the DUT's interfaces or signals of interest is the monitor's responsibility. It records events or transactions that take place on designated interfaces by serving as an interface between the DUT and the testbench environment. The monitor continually gathers input and output data from the DUT and transforms it into transaction-level data that may be used by other testbench components to process.

Monitor uses Build Phase to ensure proper connection with the interface using uvm_config_db and prompts an error if it is failed to connect with the interface and also uses connect phase and run phase to connect with the other components in uvm verification and to get the signals from the DUT through the interface.

```
//Build Phase
`uvm_info("MONITOR", "Inside Build Phase!",UVM_LOW)
    monitor_port = new("monitor_port", this);
    if(!(uvm_config_db # (virtual intfc):: get (this, "*", "vif", vif)))
    begin
    `uvm_error (" DRIVER CLASS", "Failed to get vif from config DB!")
    end
```

Important features of a monitor consist of:

- Recording transactions for input and output.
- Transaction data formatting and packing.
- Supplying a uniform interface via which additional components can obtain transaction data.
- In order to provide flexibility and simple interaction with other UVM components, monitors are frequently built as distinct modules or classes inside the testbench environment.

SCOREBOARD:

The scoreboard is in charge of confirming that the DUT is acting appropriately by contrasting its output with the predicted output that the testbench generates. It serves as a benchmark model by which the behavior of the DUT is verified. Transaction-level data is sent from the monitor to the scoreboard, which then compares it with the testbench's predicted outcomes.
A reference fifo queue is declared in the scoreboard as trans. The data_in from the seq_item is pushed into the queue and popped into another temp_data. The temp_data and the actual data_out are compared.

There is a write method to push the values into the queue and task read method to get the values from the reference queue and compare the output with the actual data_out.
Scoreboard contains an analysis port and that is connected to different components in the uvm testbench hierarchy. An object is created for this in the scoreboard build phase along with the connect phase and run phase , which majorly focus on checking the actual values with the expected values during read and write operation.

```
function void write(fifo_seq_item seq item);
     trans.push_front(seq item);
       if(seqitem.w_en &!seq item.full)
       begin
        trans_data.push_front(seqitem.data_in);
                  `uvm_info("Scoreboard  write  data_in",$sformatf("Burst
Details:w_en=%d, data_in=%d, full=%0d",seqitem.w_en, seqitem.data_in, seq
item.full),UVM_LOW)
        end
    endfunction
```

ENVIRONMENT:

The environment represents the primary container for organizing and managing the verification components. It includes agents, drivers, monitors, sequencers, scoreboard, and other necessary modules. The environment is responsible for creating and configuring these components and ensuring proper communication among them.
Environment follows the same pattern as the agent does but environment calls the create method for agent, scoreboard and coverage using the factory registration methods in the build phase and also connects the monitor analysis port to the scoreboard and coverage analysis port through the hierarchical instantiation in connect phase as shown in the figure below.

```
// Build Phase
        agnt = agent ::type_id::create("agnt",this);
        scb = scoreboard:: type_id::create("scb", this);
```

```
        cov=coverage::type_id::create("cov",this);
// Connect Phase
agnt.mon.monitor_port.connect(scb.scoreboard_port);
agnt.mon.monitor_port.connect(cov.coverage_port);
```

TEST:

The testbench encapsulates the entire verification environment and orchestrates the execution of tests. It coordinates the setup, execution, and teardown of tests, as well as manages communication between various components of the testbench. The testbench is typically implemented as a subclass of "uvm_environment" or "uvm_component".

The two sequences are used here to generate the tests for read and write separately.
The actual raise objection and drop objection will be defined in this test run phase where the actual run phase begins in the uvm testbench hierarchy. Uvm_test calls the create function using the factory override method in the build phase.

```
super.run_phase(phase);
        `uvm_info("TEST CLASS", "INSIDE RUN PHASE!",UVM_LOW)
        phase.raise_objection(this);
            write_seq=  fifo_sequence::type_id::create("write_seq");
            write_seq.start(e0.agnt.s0);
        #10;
            read_seq= sequence_fifo_rd::type_id::create("read_seq");
            read_seq.start(e0.agnt.s0);
        #10;
        phase.drop_objection(this);
```

TESTBENCH TOP:

At the highest level of the hierarchy is the test. A test represents a specific verification scenario or test case that exercises certain functionalities or features of the DUT. Tests are usually defined as individual classes extending the "uvm_test" base class provided by UVM.
This is the top level class in the entire uvm testbench hierarchy and no phases are defined in this class and only reset and clocks are generated and the DUT instantiation takes place in this class.

```
initial begin
        uvm_config_db #(virtual intfc):: set(null, "*", "vif", ifuvm);
    end
    initial begin
        run_test("uvmtest");
    end
    always #5 rclk=~rclk;
```

```
    always #2 wclk=~wclk;
```

COVERAGE:

UVM coverage refers to the metrics and measures used in the Universal Verification Methodology (UVM) to assess the completeness of verification efforts during digital design verification.

The extract phase and report phase are defined in the coverage class which are the last phases as described below in the uvm phases where we display the coverage score and display them by also specifying the verbosity as well. The following code snippets shows the ways of defining the coverage.

```
function void extract_phase(uvm_phase phase);
   super.extract_phase(phase);
  coverage_score1=cov_mem.get_coverage();
coverage_score2=test_write.get_coverage();
coverage_score3=test_read.get_coverage();
endfunction

function void report_phase(uvm_phase phase);
    super.report_phase(phase);

`uvm_info("COVERAGE",$sformatf("Coverage=%0f%%",coverage_score1),UVM_MEDIU
M);

`uvm_info("COVERAGE",$sformatf("Coverage=%0f%%",coverage_score2),UVM_MEDIU
M);

`uvm_info("COVERAGE",$sformatf("Coverage=%0f%%",coverage_score3),UVM_MEDIU
M);
endfunction
```

UVM Phases:

S.NO	Type of Phase
1.	Build Phase
2.	Connect Phase

<table>
<tr><td>3.</td><td colspan="3">End Of Elaboration Phase</td></tr>
<tr><td>4.</td><td colspan="3">Start Of Simulation Phase</td></tr>
<tr><td rowspan="5">5.</td><td colspan="3">Run phase</td></tr>
<tr><td>Pre-Reset</td><td>Reset</td><td>Post- Reset</td></tr>
<tr><td>Pre-Configure</td><td>Configure</td><td>Post-Configure</td></tr>
<tr><td>Pre- Main</td><td>Main</td><td>Post - Main</td></tr>
<tr><td>Pre-Shutdown</td><td>Shutdown</td><td>Post-Shutdown</td></tr>
<tr><td>6.</td><td colspan="3">Extract Phase</td></tr>
<tr><td>7.</td><td colspan="3">Check Phase</td></tr>
<tr><td>8.</td><td colspan="3">Report Phase</td></tr>
<tr><td>9.</td><td colspan="3">Final Phase</td></tr>
</table>

IMPLEMENTATION:

As per the project we considered one agent,two sequences,one sequencer,one sequence item,one scoreboard with coverage.

- The data_in is randomized and allowed to pass through the different components such as driver,monitor,scoreboard.
- Whenever write enable is asserted,the data will be written to the fifo memory and the burst details are displayed in each component.The write method which is called in the monitor is responsible for transferring the data to the scoreboard.
- Whenever read enable is asserted, the data will be read from the fifo memory and the burst details are displayed in each component.This will happen in the run phase of the scoreboard.

CONTRIBUTION OF TEAM MEMBERS:

- Bhavani: Implemented monitor, Write Sequence, Sequencer,Sequence Item successfully.
- Asritha: Implemented scoreboard,driver, read sequence , agent successfully.
- Vijay: Implemented Environment, Test, TestBench Top successfully.
- Asritha and Bhavani together tested the burst passing from sequence-> sequencer-> driver thoroughly.
- Vijay and Bhavani together tested the burst passing from the driver->monitor

->scoreboard thoroughly.

- Asritha and Vijay : Implemented the coverage and verified the entire design thoroughly and wrote the uvm verification plan document.

CHALLENGES FACED:

- Received data_out value as ‘x’ initially due to improper connection during the initial run.
- Incorrect values for read address increment.
- Not received correct full and empty values after read and write transfers.
- Not getting the updated data out after every read transfer in monitor and driver.
- Did not implement the error correction and detection mechanism to correctly verify the asynchronous fifo.
- Race Conditions between write and read pointers.
- Unable to reset back to the normal operation.

CHALLENGES RESOLVED:

- Checked thoroughly the entire testbench and identified the improper connection in between the classes and resolved the issue by placing a proper connection between the components.
- Identified the incorrect values for the read address increment due to extra delays in the few testbench components and resolved by reducing the delay and checking the proper flow.
- Unable to connect with the design properly to achieve the full and empty conditions and resolved by thoroughly verifying the connection between the design and Testbench through interface and saw the difference of receiving the correct values in the transcript by removing the clocking blocks.
- Identified the previous values getting updated in the monitor for two every two read operations performed and resolved the issue by providing the same delays in both driver and monitor.
- Identified the different values propagating from data_in to data_out resulted in race conditions between read and write pointers and resolved it by checking and modifying the code thoroughly in both design and testbench
- Identified the Incorrect reset value assigned in the top level testbench of uvm testbench verification hierarchy and resolved it by correctly specifying the value to the reset signal.