Report on Class-Based implementation

a. Package

i. Included files Transaction, Generator, Driver, Monitor, Scoreboard and Environment with `include inside a package, 'AFIFO_Pkg'

```
package AFIFO_Pkg;
  `include "AFIFO_Transaction.sv"
  `include "AFIFO_Generator.sv"
  `include "AFIFO_Driver.sv"
  `include "AFIFO_Monitor.sv"
  `include "AFIFO_Scoreboard.sv"
  `include "AFIFO_Environment.sv"
endpackage
```

b. Transaction Class

i. The data width sizes are kept parameterizable in the class.

ii. Signals like wr_data are declared using randc to enable constrained cyclic randomization.

```
class AFIFO_Transaction #(parameter DSIZE = 8);
 randc logic [DSIZE-1:0] wr_data;
```

iii. We declare memory to represent data transfer events to generate stimulus or represent expected results. Some are stored in memory for later processing (e.g., in a scoreboard).

```
logic [DSIZE-1: 0]rd_data;
bit rd_empty,wr_full;
```

iv. We create a function copy() to create a deep copy of the object when called.

```
function AFIFO_Transaction copy();// create deep copy
copy = new();

copy.wr_data = this.wr_data;
copy.rd_data = this.rd_data;
copy.rd_empty = this.rd_empty;
copy.wr_full = this.wr_full;
 return copy;
endfunction
```

c. Interface

i. This serves as a physical connection between the testbench and DUT. It groups all necessary signals to interact with the DUT.
ii. The interface is kept parameterizable to be scalable to different FIFO configurations.

```
interface AFIFO_Interface #(parameter DSIZE = 8,parameter ASIZE = 4);
```

iii. Data signals, Status flags, Control signals, Clock and Reset signals are included in the interface, simplifying code connectivity.

```
logic [DSIZE-1:0] rd_data;
logic [DSIZE-1:0] wr_data;
logic wr_full;
logic rd_empty;
logic wr_inc;
logic rd_inc;
logic wr_clk;
logic rd_clk;
logic wr_rst;
logic rd_rst;
```

d. Generator

i. We first declare the transaction class object, a mailbox to send transactions from the generator to the driver, events for synchronization, a count to track transactions.

```
class AFIFO_Generator;

AFIFO_Transaction tr;

mailbox mbx_gen2drv;

event gen_done;
event scb_done;
event driver_done;

int count = 0;
```

ii. A function new() allocates memory for the transaction object created, and connects the mailbox to the one passed.

```
function new(mailbox mbx_gen2drv);
        tr = new();
        this.mbx_gen2drv = mbx_gen2drv;
endfunction
```

iii. In task run(), we call the main stimulus generation loop.

iv. Inside its forever begin, it first waits for the event drv_next to be triggered.

v. We then randomize the transaction, using assertion we also check if randomization fails and display error.

```
wait( drv_nxt.triggered);

assert(tr.randomize) else $error("[GEN]: Randomization Failed");
```

vi. We then create a deep copy of the transaction class and send it from Generator to Driver, and display a message accordingly of [PUT SUCCESS], otherwise displayed as [PUT FAILED] if failed.

```
if(mbx_gen2drv.try_put(tr.copy)) $display("[GEN: DEBUG [PUT SUCCESS] placed in mailbox");
else $display("[GEN: DEBUG] [PUT FAILED]: Mailbox full, could not put ");
$display(" [GEN] Generated Write Data : %0d", tr.wr_data);
#2;
```

vii. So in short, the generator waits for the driver to request a transaction using wait(drv_nxt.triggered). Then randomises the transaction, sends the transaction to the driver and prints debug information. Also going into a forever loop with #2 delay.

e. Driver

i. First, a virtual interface is declared to communicate with the DUT, a transaction object is declared.

```
virtual AFIFO_Interface vif;
AFIFO_Transaction tr;
```

ii. Mailboxes for transaction transfers from generator to driver and from driver to scoreboard are declared.

```
mailbox #(AFIFO_Transaction) mbx_gen2drv; //generator to driver
mailbox #(bit [DSIZE-1:0]) mbx_drv2sco;    // driver to scoreboard
```

And an event driver_done.

```
event driver_done;
```

iii. The mailboxes are initialised using a constructor function.

```
function new(mailbox #(AFIFO_Transaction) mbx_gen2drv, mailbox #(bit [DSIZE-1:0]) mbx_drv2sco);

this.mbx_gen2drv = mbx_gen2drv;
this.mbx_drv2sco = mbx_drv2sco;

endfunction
```

iv. **task reset () :** It asserts the active low signals, vif.rd_rst and vif.wr_rst to be 0; wait for 5 vif.wr_clk clock cycles and asserts vif.rd_rst and vif.wr_rst to be 1 again. Once done, displays Reset Done.

```
task reset(); // Test Case :1
  $display("[DRV] : Entered Reset state");
  vif.rd_rst <= 1'b0; //Active Low Reset
  vif.wr_rst <= 1'b0;

  $display("[DRV] : RESETING FOR 5 WRITE CLK CYCLES");
  repeat(5) @(posedge vif.wr_clk);
  vif.rd_rst <= 1'b1;
  vif.wr_rst <= 1'b1;
  $display("[DRV] : Reset Done");
  $display("----------------------------------------");
endtask
```

v. **task write () :** Triggers drv_nxt event. And sets active low signal vif.wr_rst to 1.

```
->drv_nxt;
```

Then in a repeat(drv_repeat_count) loop, at posedge of vif.wr_clk,

```
repeat(drv_repeat_count) begin
       @(posedge vif.wr_clk);
```

It fetches a transaction from mbx_gen2drv, and writes it to vif.wr_data.

```
(mbx_gen2drv.try_get(tr))
```

Accordingly the vif.wr_inc is set to 1.

```
vif.wr_data = tr.wr_data;
vif.wr_inc = 1'b1;
```

The written data is also sent to the scoreboard.

```
mbx_drv2sco.put(tr.wr_data);
```

The vif.wr_inc is set to 0 and driver_done event is triggered after each write at the next clock cycle.

```
@(posedge vif.wr_clk); //Experimental
vif.wr_inc = 1'b0;
->driver_done;
```

vi. **task read () :**

```
task read(input int drv_repeat_count);
```

Reads data from FIFO for drv_repeat_count number of times, by toggling rd_inc to trigger a read operation.

```
repeat(drv_repeat_count) begin
        @(posedge vif.rd_clk);
```

```
vif.rd_inc <= 1'b1;
$display("[DRV] : Data Read to rd_data");
```

```
@(posedge vif.rd_clk);
vif.rd_inc <= 1'b0;
```

vii. Defined various such test cases to trigger write full, read empty, to have continuous reads and writes.

viii. A task run is called to run all test cases sequentially and $finish the simulation.

```
task run;
testcase1();
testcase2();
testcase3();
testcase4();
testcase5();
$finish;
endtask
```

f. Monitor

i. It is responsible for monitoring the FIFO read interface and sending data to the scoreboard via mailbox.

ii. We first declare a virtual interface handle to connect to the DUT.

```
virtual AFIFO_Interface vif;
```

iii. A mailbox mbx_mon2sco is declared to send data to the scoreboard.

```
mailbox #(bit[DSIZE -1 :0]) mbx_mon2sco;
```

iv. A function new() is called to initialise the mailbox mbx_mon2sco and assign memory to it.

```
function new(mailbox #(bit[DSIZE -1 :0]) mbx_mon2sco);
 this.mbx_mon2sco = mbx_mon2sco;
endfunction
```

v. Inside task run(), a forever loop is started to monitor the FIFO state.

vi. At the posedge of vif.rd_clk, it checks if vif.rd_inc && !vif.rd_empty (read enable is asserted and FIFO is not empty).

```
@(posedge vif.rd_clk);
if(vif.rd_inc && !vif.rd_empty) begin
```

vii. It then captures the vif.rd_data and puts it in the mailbox to scoreboard.

```
mbx_mon2sco.put(vif.rd_data);
```

viii. Also checks for FIFO full or empty with $display statements.

```
if(vif.rd_empty) $display("[MON] : READ EMPTY, Nothing to read in FIFO Mem");
if(vif.wr_full) $display("[MON] : WRITE FULL, Full FIFO Mem");
```

g. Scoreboard

i. It is a key verification component used to compare expected and actual FIFO read data.

ii. A virtual interface, rd_data to receive from the monitor, queue of the written data for comparison against the read data like handles and variables are declared.

```
virtual AFIFO_Interface vif;
```

```
bit [DSIZE-1:0] rd_data_dut; // DUT read data
reg [DSIZE -1 :0] wr_data_drv_q[$]; //Queue Driver to Scoreboard
bit[DSIZE-1 :0] get_wr_data_drv, ref_wr_data_drv;
```

iii. Mailboxes mbx_mon2sco and mbx_drv2sco are declared to hold read data and written data.

```
mailbox #(bit[DSIZE-1 :0]) mbx_mon2sco; // monitor to scoreboard
mailbox #(bit[DSIZE-1 :0]) mbx_drv2sco;  // driver to scoreboard
```

iv. A function new(), acts as a constructor to the mailboxes and initialises them.

```
function new(mailbox #(bit[DSIZE-1 :0]) mbx_mon2sco,
             mailbox #(bit[DSIZE-1 :0]) mbx_drv2sco);
this.mbx_mon2sco = mbx_mon2sco;
this.mbx_drv2sco = mbx_drv2sco;
endfunction
```

v. In a task run(), it stores read data from the mailbox from the monitor in rd_data_dut.

```
mbx_mon2sco.get(rd_data_dut);
```

vi. If a write is triggered (vif.wr_inc), get written data from the driver through the mailbox and store in the end of the queue wr_data_drv_q using push_back for later comparison.

```
if(vif.wr_inc) begin
mbx_drv2sco.get(get_wr_data_drv);
wr_data_drv_q.push_back(get_wr_data_drv);
```

vii. Reset FIFO if (wr_rst or rd_rst) is triggered; the scoreboard clears all stored transactions using delete method in queues.

```
if(vif.wr_rst || vif.rd_rst)begin
wr_data_drv_q.delete();
end
```

viii. If FIFO is reading and its not read empty (vif.rd_inc && !vif.rd_empty), start comparing the actual data with the expected data from the queue, and print success or error accordingly.

```
if(vif.rd_inc && !vif.rd_empty)begin
       ref_wr_data_drv =  wr_data_drv_q.pop_front();
```

```
if(rd_data_dut == ref_wr_data_drv) begin
$display("[SCO]: SUCESS, Data Matched : %d",rd_data_dut);
end
else begin
$error("[SCO]: FAILED, Data NOT Matched : [DUT] %d != [DRV] %d",rd_data_dut, ref_wr_data_drv);
end
```

h. Environment

i. It is a container for all testbench components.

```
AFIFO_Driver dr;
AFIFO_Generator gr;
AFIFO_Monitor mo;
AFIFO_Scoreboard sco;
```

1. Driver (AFIFO_Driver) - Drives transactions into the DUT.
2. Generator (AFIFO_Generator) - Generates input stimulus for the driver.
3. Monitor (AFIFO_Monitor) - Observes DUT outputs and sends data to the scoreboard.
4. Scoreboard (AFIFO_Scoreboard) - Compares expected vs. actual results.

ii. Events are declared for synchronisation.

```
event nextgd;
event next_gen;
```

iii. Mailboxes are declared for communication.

```
mailbox #(bit[DSIZE-1 :0]) mbx_mon2sco; // monitor to scoreboard
mailbox #(AFIFO_Transaction) mbx_gen2drv; //generator to driver
mailbox #(bit [DSIZE-1:0]) mbx_drv2sco;    // driver to scoreboard
```

iv. A virtual interface connects the testbench with the DUT signals, allowing multiple components (driver, monitor, scoreboard) to interact with the same interface.

```
virtual AFIFO_Interface vif;
```

v. A function new() is used to take a virtual interface (vif) and assign it to the environment.

```
function new(virtual AFIFO_Interface vif);
```

It also initialises mailboxes;

```
mbx_mon2sco = new();
mbx_gen2drv = new();
mbx_drv2sco = new();
```

creates driver, generator, monitor, and scoreboard instances and passes mailboxes for proper linking.

```
dr = new(mbx_gen2drv, mbx_drv2sco);
gr = new(mbx_gen2drv);
mo = new(mbx_mon2sco);
sco = new(mbx_mon2sco,mbx_drv2sco);
```

It binds the interface to Driver, Monitor, and Scoreboard, so they can access DUT signals.

```
this.vif = vif;
dr.vif = this.vif;
mo.vif = this.vif;
sco.vif = this.vif;
```

Events are assigned for synchronization between the generator (gr) and driver (dr). Ensuring both the generator and driver share the same event handles.

```
gr.drv_nxt = nextgd;
dr.drv_nxt = nextgd;
gr.gen_done = next_gen;
dr.gen_done = next_gen;
```

The task run() spawns all testbench components in parallel using fork..join_any.

```
task run();
fork
gr.run();
dr.run();
mo.run();
sco.run();
join_any
```

i. TB_TOP

i. It sets up the test environment for verifying the Asynchronous FIFO. It connects the interface (vif), instantiates the FIFO, and drives clock signals for simulation.

ii. First we declare a virtual interface for connecting the testbench components to the DUT.

```
AFIFO_Interface vif();
```

iii. Instantiate the FIFO DUT.

```
FIFO #(DSIZE, ASIZE) fifo (vif.wr_data, vif.wr_clk, vif.wr_rst, vif.wr_inc,
                    vif.rd_clk, vif.rd_rst, vif.rd_inc, vif.rd_data, vif.rd_empty, vif.wr_full);
```

iv. Initialise the read and write clock.

```
initial begin
vif.wr_clk <= 0;
vif.rd_clk <= 0;
end
```

v. Generate the read and write clock periods to operate at different frequencies.

```
always #10ns vif.wr_clk <= ~vif.wr_clk;
always #35ns vif.rd_clk <= ~vif.rd_clk;
```

vi. The environment is initiated and the virtual interface is passed through it.

```
AFIFO_Environment en;

initial begin
 en = new(vif);
 en.run();
end
```

vii. en.run is called which starts the generator, driver, monitor and scoreboard.

j. COVERAGE:

Functional Coverage: 75%
Code Coverage: 67.47%

```
// ====== Coverage variables ======
covergroup fifo_coverage @(posedge vif.wr_clk);
  cp1: coverpoint vif.wr_data {
    bins zero_z = {0};
    bins low_l = {[1:10]};
    bins medium_m = {[11:100]};
    bins high_h = {[101:255]};
  }

// ====== covergroup read_data_cov @(posedge vif.rd_clk); ======
  cp2: coverpoint vif.rd_data {
    bins zero_z = {0};
    bins low_l = {[1:10]};
    bins medium_m = {[11:100]};
    bins high_h = {[101:255]};
  }

// ====== covergroup wr_operations_cov; ======
  cp3: coverpoint vif.wr_inc {
    bins write_zero = {0};
    bins write_one = {1};
  }

//====== covergroup rd_operations_cov; ======
  cp4: coverpoint vif.rd_inc {
    bins read_zero = {0};
    bins read_one = {1};
  }
endgroup
```

k. Results achieved:

- Implementation of **Clock Domain Crossing**
- Implementation of **FIFO Depth**
- **Developed Class Based Verification Testbench** in SV
- Stimulus generated for **100 to 150 FIFO Writes and Reads**
- **All Reads MATCHED**
- Implemented Functional and Code Coverage (Results in Verification Plan)
- Warnings: Only one warning related to the multiple compilation of module
    - Explaination: Because of using package (SV Constructs)
    - **Solution:** Will be cleared in next Milestone using **Guard Rings**
- **Functional Coverage: 75% (Will be improved in next Milestone)**
- **Code Coverage: 67.47% (Will be improved in next Milestone)**