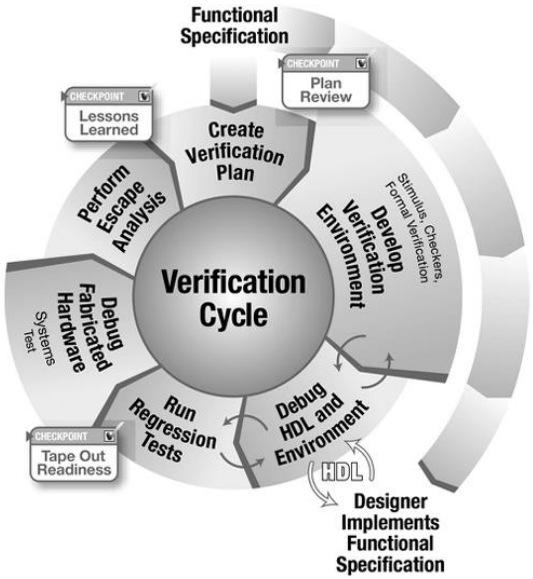
ECE 593
Fundamentals of Pre-Silicon Validation



Venkatesh Patil
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Portland State UNIVERSITY

IMPLEMENTATION AND VERIFICATION OF ASYNCHRONOUS FIFO USING BOTH CLASS-BASED AND UVM METHODOLOGIES

Session 1 – Group 1

KARNA KUMAR DURGA MANOHAR	kkarna@pdx.edu
MOHAMMEED ABBAS SHAIK	mohammee@pdx.edu
NIKHITHA VADNALA	vadnala@pdx.edu
NIVEDITA BOYINA	nivedita@pdx.edu

Portland State UNIVERSITY

Agenda

- ❑ Introduction of Team Members
- ❑ Project Introduction
- ❑ Contribution of Team Members
- ❑ Design Implementation
- ❑ Class based Verification
- ❑ UVM based Verification
- ❑ Overall Challenges and Learnings
- ❑ Demo
- ❑ Conclusion

Introduction of Team Members

- KUMAR DURGA MANOHAR KARNA
  PSU ID : 912527531
  Term : 2nd term
  Graduation term : Fall 2025
  Internship/ Offer : N/A

- MOHAMMEED ABBAS SHAIK
  PSU ID : 950376592
  Term : 3rd term
  Graduation term : Spring 2025
  Internship/ Offer : N/A

- NIKHITHA VADNALA
  PSU ID : 941449738
  Term : 2nd term
  Graduation term : Fall 2025
  Internship/ Offer : N/A

- NIVEDITA BOYINA
  PSU ID : 958615179
  Term : 3rd term
  Graduation term : Spring 2025
  Internship/ Offer : N/A

Project Introduction

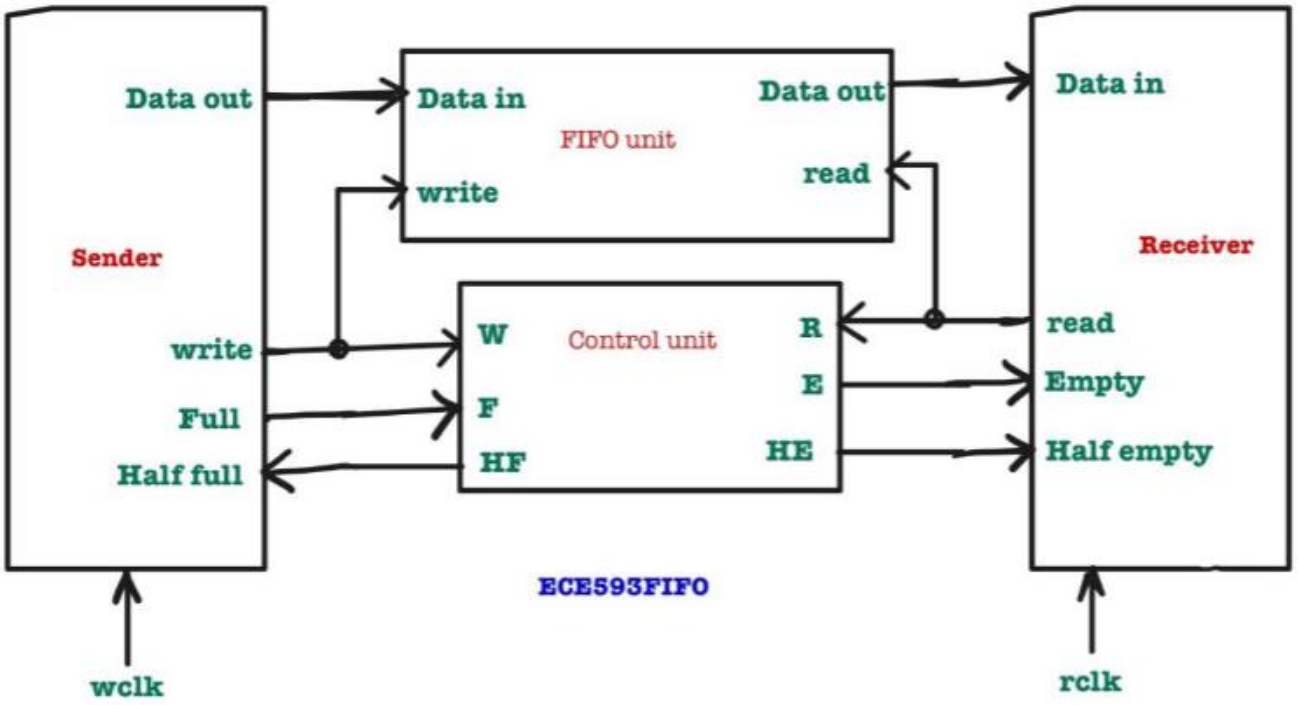
An Asynchronous FIFO (First-In-First-Out) is a specialized type of FIFO buffer that enables data transfer between systems operating on different clock domains. Unlike synchronous FIFOs, which use a single clock signal for both read and write operations, asynchronous FIFOs employ separate read and write clocks, allowing data to be read and written independently.

- ❑ Facilitates data transfer between subsystems with different clock frequencies.
- ❑ Prevents data loss or corruption during asynchronous transfers.
- ❑ Ensures effective coordination between different clock domains.
- ❑ Employs gray code pointers to track read and write positions, minimizing metastability.
- ❑ Optimizes memory usage to store and retrieve data efficiently.

Contribution of Team Members

Nivedita Boyina	Durga Manohar	Nikhitha Vadnala	Abbas
UVM environment Design Sequences -base sequence -full sequence -random sequence Sequence Item HLDS Document	UVM environment Write agent Write driver Write sequencer Write monitor Coverage Verification Plan Document	UVM environment Read agent Read driver Read sequencer Read monitor Scoreboard Environment Presentation Slides	UVM environment Test cases -(base, full, random) Testbench Interface Package Academic Paper
Class environment Design Driver Transaction	Class environment Monitor_in Monitor_out Generator	Class environment Scoreboard Environment run.do	Class environment Test Testbench Interface

Design Implementation



Level Block Diagram of the Design System

Design Specification

- **Asynchronous FIFO** : An Asynchronous FIFO (First-In-First-Out) buffer is used to transfer data between two clock domains with different clock frequencies.
- **Gray Code** : Gray code is used for the read and write pointers because it ensures that only one bit changes at a time, reducing the chance of sampling errors and metastability.
- **Memory Array** : The FIFO buffer uses a memory array to store data. Data is written to the memory in the write clock domain and read from the memory array in the read clock domain.
- **Write Pointer** : A binary write pointer keeps track of the location where the next data should be written. The binary write pointer is converted to a Gray code write pointer for synchronization across the clock domains.
- **Read Pointer** : A binary read pointer keeps track of the location from where the next data should be read. The binary read pointer is converted to a Gray code read pointer for synchronization across the clock domains.

Design Specification

- **Pointer Synchronization** : The Gray code write pointer is synchronized into the read clock domain using two-stage synchronization to mitigate metastability. The Gray code read pointer is synchronized into the write clock domain similarly.
- **Full Condition** : The FIFO is full when the next write pointer value (in Gray code) equals the read pointer value with the most significant bit inverted.
- **Empty Condition** : The FIFO is empty when the synchronized write pointer equals the read pointer.
- **Half Full** : The FIFO is Half – Full when read point is at depth/2
- **Half Empty** : The FIFO is Half – Empty when write pointer is at depth/2

Design Implementation

Design Specifications

Writing frequency = f_A = 120MHz

Reading Frequency = f_B = 50MHz

Burst Length = No. of data items to be transferred = 1024

No. of idle cycles between two successive writes is = 4

No. of Idle cycles between two successive reads is = 2

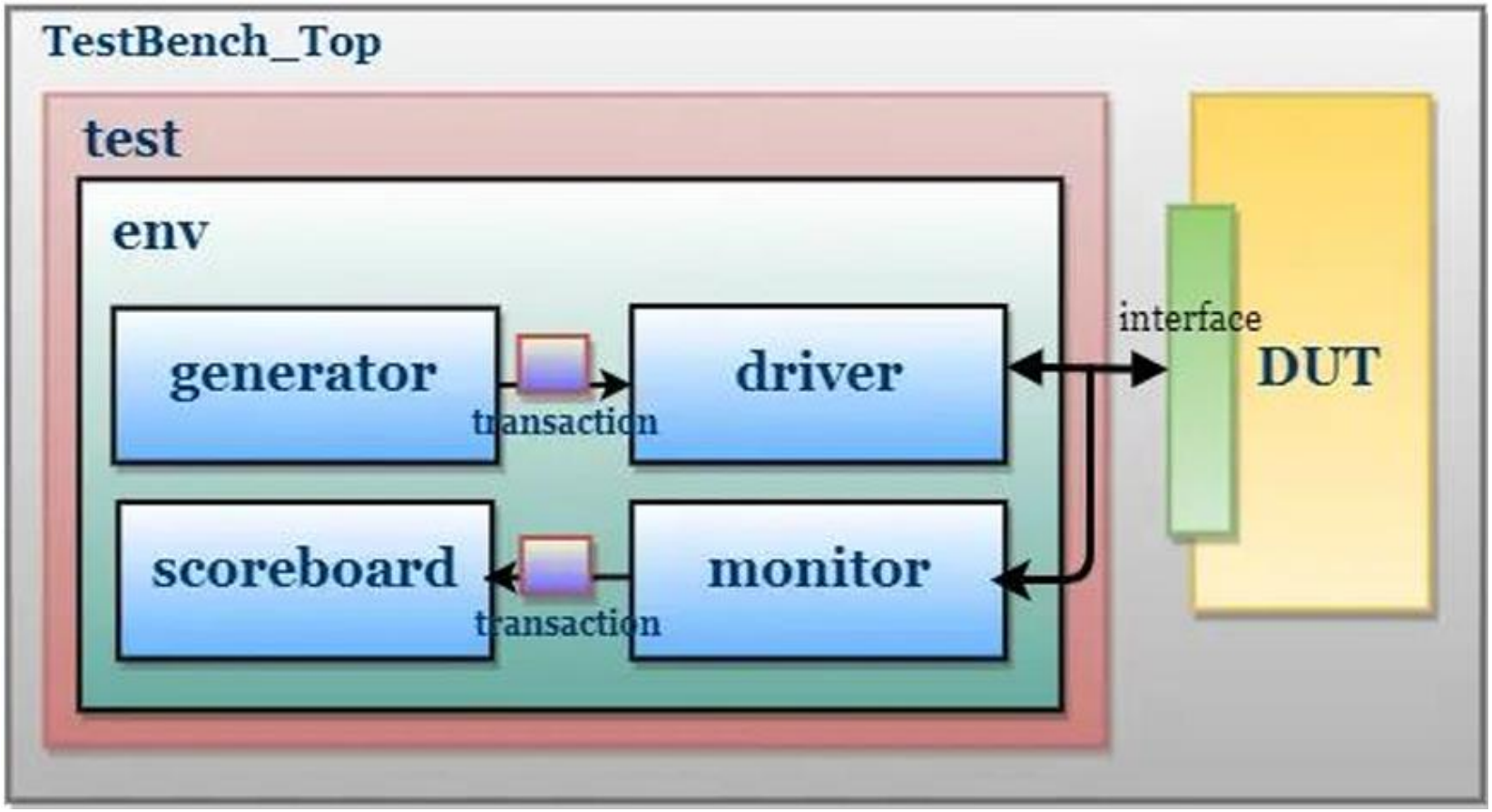
Minimum depth of FIFO = 307

FIFO depth considered here = 512

Time required to write one data item = 42 ns

Time required to read one data item = 60 ns.

Class based Verification (4 mins)



Class based Verification (4 mins)

Transaction :

```
class transaction;

    bit wrst_n, rrst_n, wclk, rclk;
    rand bit [7:0] data_write;
    rand bit write_enable;
    rand bit read_enable;
    logic [7:0] data_read;
    bit rempty;
    bit wfull;


endclass
```

Generator :

```
class generator;

    rand transaction tx;
    longint tx_count;
    event driv2gen;

    mailbox gen2driv;

    function new(mailbox gen2driv, event driv2gen);
        this.gen2driv = gen2driv;
        this.driv2gen = driv2gen;
    endfunction

    task main();

        repeat (tx_count)
        begin
            tx = new();
            if(!tx.randomize())
            $fatal ("Randomization for the transaction is failed:");
            gen2driv.put(tx);
        end
            ->driv2gen;
    endtask

endclass
```

Class based Verification (4 mins)

Driver :

```
//reset when write or read reset
task reset;
    $display("Reset Started");
    wait(driver_if.wrst_n || driver_if.rrst_n);
    driver_if.data_write <= 0;
    driver_if.write_enable <= 0;
    driver_if.read_enable <= 0;
    wait(!driver_if.wrst_n || driver_if.rrst_n);
    $display("Reset Ended:");
endtask
```

```
virtual task drive();
begin
    transaction tx1;
    driver_if.write_enable <= 0;
    driver_if.read_enable <= 0;
    gen2driv.get(tx1);

    @(posedge driver_if.wclk);
    if(tx1.write_enable)
    begin
        driver_if.write_enable <= tx1.write_enable;
        driver_if.data_write <= tx1.data_write;;
        tx1.wfull = driver_if.wfull;
        tx1.rempty = driver_if.rempty;
        $display ("\t write_enable = %0h \t data_write = %0h", tx1.write_enable, tx1.data_write);
    end
    else
    begin
        $display ("\t write_enable = %0h \t data_write = %0h", tx1.write_enable, tx1.data_write);
    end


    if(tx1.read_enable)
    begin
        driver_if.read_enable <= tx1.read_enable;
        @(posedge driver_if.rclk);
        driver_if.read_enable <=tx1.read_enable;
        @(posedge driver_if.rclk);
        tx1.data_read = driver_if.data_read;
        tx1.wfull = driver_if.wfull;
        tx1.rempty = driver_if.rempty;
        $display ("\t read_enable = %0h", tx1.read_enable);
    end
    else
    begin
        $display ("\t read_enable = %0h", tx1.read_enable);
    end

end
endtask
```

Class based Verification (4 mins)

Monitor :

```
class monitor;

    virtual intf moniter_if;
    mailbox mon2scb;

    function new(virtual intf moniter_if,mailbox mon2scb);
        this.moniter_if = moniter_if;
        this.mon2scb = mon2scb;
    endfunction

    virtual task drive();
    begin
        transaction tx2;
        tx2 = new();
        @(posedge moniter_if.rclk);
        tx2.read_enable = moniter_if.read_enable;
        tx2.write_enable = moniter_if.write_enable;
        tx2.data_write = moniter_if.data_write;
        tx2.wfull = moniter_if.wfull;
        tx2.rempty = moniter_if.rempty;
        tx2.data_read = moniter_if.data_read;
        mon2scb.put(tx2);
    end
    endtask


    task  main();
    begin
        for (int i = 0; i < 1; i++)
            drive();
    end
    endtask

endclass
```

Scoreboard :

```
virtual task main();
begin
    transaction tx3;
    mon2scb.get(tx3);

        if(tx3.write_enable)
            begin
                fifo_mem[wr_ptr] = tx3.data_write;
                wr_ptr = wr_ptr + 1;
            end


        if(tx3.read_enable)
            begin
                if(tx3.data_read == fifo_mem[rd_ptr])
                    begin
                        $display("design works correctly at address %0h - tx3.Data = %0h -
                            Saved Data = %0h",rd_ptr, tx3.data_read,fifo_mem[rd_ptr]);
                        rd_ptr = rd_ptr + 1;
                    end
                else
                    begin
                        $display("design has error at address %0h - tx3.Data = %0h - Saved Data = %0h",
                            rd_ptr,tx3.data_read,fifo_mem[rd_ptr]);
                    end
            end


       if(tx3.wfull)
            $display("FIFO is full");

      if(tx3.rempty)
            $display("FIFO is empty");


      tran_num++;

end
endtask
```

Class based Verification (4 mins)

Test :

```
`include "Environment.sv"

program test(intf in);
  environment env;

    initial
        begin

            $display("test environment started");
            env = new(in);
            env.gen.tx_count =10;
            env.tran_num=100;
            env.run();
            $display("TEST FINISHED SUCCESSFULLY");
            $finish;
        end

endprogram
```

Interface :

```
interface intf(input logic wclk,rclk,wrst_n,rrst_n);

    logic [7:0] data_write;
    logic write_enable;
    logic read_enable;
    logic [7:0] data_read;
    logic rempty;
    logic wfull;
    logic half_rempty;
    logic half_full;

endinterface: intf
```

Class based Verification (4 mins)

Top :

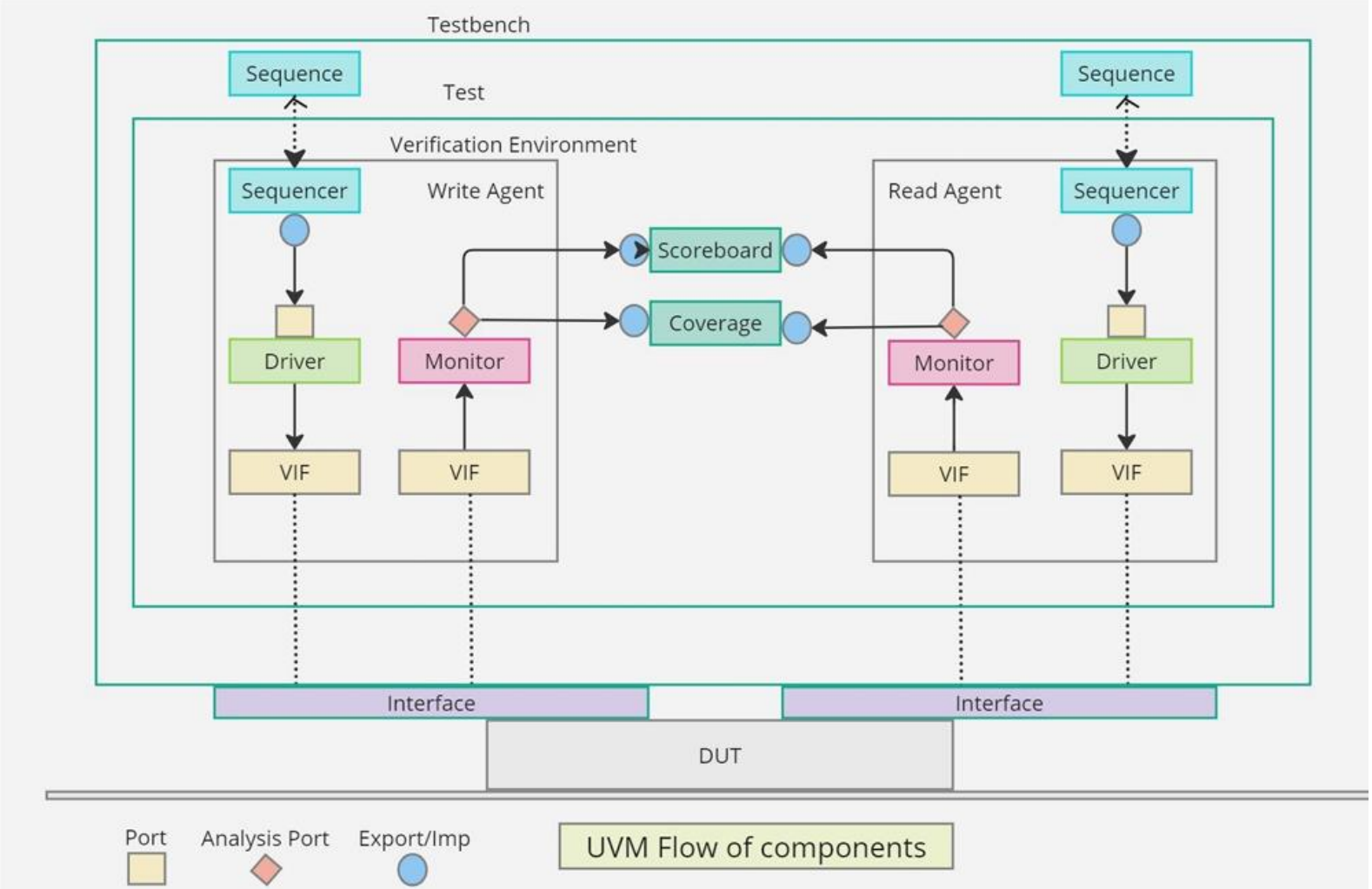
```
`include "async_fifo_test.sv"
`include "Interface.sv"

module async_fifo_top;

    bit rclk,wclk,rrst_n,wrst_n;
    logic [7:0] data_write;
    logic [7:0] data_read;

    always #21ns wclk = ~wclk;
    always #30ns rclk = ~rclk;

    initial
        begin
            wclk =0;
            rclk=0;
            wrst_n =0;
            rrst_n=0;
            #50;
            rrst_n =1;
            wrst_n=1;
        end


intf in (wclk,rclk,wrst_n,rrst_n);
test t1 (in);

asynchronous_fifo DUT (.wclk(in.wclk),
            .wrst_n(in.wrst_n),
            .rclk(in.rclk),
            .rrst_n(in.rrst_n),
            .write_enable(in.write_enable),
            .read_enable(in.read_enable),
            .data_read(in.data_read),
            .data_write(in.data_write),
            .wfull(in.wfull),
            .rempty(in.rempty),
            .half_full(in.half_full),
            .half_rempty(in.half_rempty));
```

Coverage :

```
covergroup async_fifo_cover;
  option.per_instance = 1;

  DATA_WRITE : coverpoint in.data_write{
      option.comment = "write data ";
          bins low_range  = {[1:511]};
          bins mid_range  = {[512:1023]};
          bins high_range = {[1023:2048]};
  }

  WRITE_FULL : coverpoint in.wfull {
      option.comment = "FULL_FLAG";
      bins full_c  = ( 0 => 1 );
      bins full_c1  = ( 1 => 0 );
  }

  READ_EMPTY: coverpoint in.rempty {
      option.comment = "WHEN RESET IS OFF check EMPTY";
      bins empty_c  = (0 => 1);
      bins empty_c1 = (1 => 0);
  }

  DATA_READ : coverpoint in.data_read{
      option.comment = "read data ";
          bins low_range  = {[1:511]};
          bins mid_range  = {[512:1023]};
          bins high_range = {[1023:2048]};
  }
```

```
  W_INC : coverpoint in.write_enable{
  bins incr_s = (0 => 1);
  bins incr_s1 = (1 => 0);
  }

  R_INC : coverpoint in.read_enable{
  bins incr_sr = (0 => 1);
  bins incr_s1r = (1 => 0);
  }


  W_RESET: coverpoint in.wrst_n {
    option.comment = "write reset signal";
    bins reset_low_to_high = (0 =>1);
    bins reset_high_to_low = (1 =>0);

  }

  R_RESET: coverpoint in.rrst_n {
    option.comment = "read reset signal";
    bins reset_low_to_high = (0 =>1);
    bins reset_high_to_low = (1 =>0);
  }


W_CLK: coverpoint in.wclk {
  option.comment = "write clock signal";
  bins clk_low_to_high = (0 => 1);
  bins clk_high_to_low = (1 => 0);
}

R_CLK: coverpoint in.rclk {
  option.comment = "read clock signal";
  bins clk_low_to_high = (0 => 1);
  bins clk_high_to_low = (1 => 0);
}

WRITExADDxDATA : cross W_CLK,W_INC,DATA_WRITE;
READxADDxDATA  : cross R_CLK, R_INC, DATA_READ;
READxWRITE     : cross DATA_WRITE,DATA_READ;
RESETxWRITE    : cross W_RESET, DATA_WRITE;
RESETxREAD     : cross R_RESET , DATA_READ;

endgroup
```

UVM based Verification (8 mins)



UVM based Verification (8 mins)

Sequence Item :

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

class transaction_write extends uvm_sequence_item;
    `uvm_object_utils(transaction_write)

    rand bit [8:0] data_write;
    rand bit write_enable;
    bit wfull,wHalf_full;

    function new(string name = "transaction_write");
        super.new(name);
    endfunction
endclass

class transaction_read extends uvm_sequence_item;
    `uvm_object_utils(transaction_read)

    rand bit read_enable;
    logic [8:0] data_read;
    bit rempty,wHalf_empty;

    function new(string name = "transaction_read");
        super.new(name);
    endfunction
endclass
```

Sequencer (write) :

```
class write_sequence extends uvm_sequence#(transaction_write);
    `uvm_object_utils(write_sequence)

    int tx_count_write = 400;
    transaction_write txw;

    function new(string name = "write_sequence");
        super.new(name);
        `uvm_info("WRITE_SEQUENCE_CLASS", "Inside constructor", UVM_LOW)
    endfunction

    task body();
        `uvm_info("WRITE_SEQUENCE_CLASS", "Inside body task", UVM_LOW)
        for (int i = 0; i < tx_count_write; i++) begin
            txw = transaction_write::type_id::create("txw");
            start_item(txw);
            if (!(txw.randomize() with {txw.write_enable == 1;}));
            //`uvm_error("TX_GENERATION_FAILED", "Failed to randomize transaction_write")
            finish_item(txw);
        end
    endtask
endclass
```

UVM based Verification (8 mins)

Driver (write) :

```
task drive_write(transaction_write txw);
    @(posedge intf_vi.wclk);
    this.intf_vi.write_enable = txw.write_enable;
    this.intf_vi.data_write = txw.data_write;
endtask

task run_phase (uvm_phase phase);
    super.run_phase(phase);
    `uvm_info("DRIVER_CLASS", "Inside Run Phase",UVM_LOW)
    this.intf_vi.data_write <=0;
    this.intf_vi.write_enable <=0;

    repeat(10) @(posedge intf_vi.wclk);

    for (integer i = 0; i < trans_count_write ; i++)
    begin
        txw=transaction_write::type_id::create("txw");
        seq_item_port.get_next_item(txw);
        $display("SK_DEBUG4 entered before wait statement wr_driver");
        wait(intf_vi.wfull ==0);
        $display("SK_DEBUG5 entered after wait statement wr_driver");
        drive_write(txw);
        seq_item_port.item_done();
    end
    @(posedge intf_vi.wclk);
    this.intf_vi.write_enable =0;
endtask
```

Monitor (write) :

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    port_write = new("port_write", this);

    if (!uvm_config_db#(virtual intf)::get(this, "", "vif", vif)) begin
        `uvm_error("build_phase", "No virtual interface specified for this write_monitor instance")
    end
endfunction

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
endfunction

task run_phase(uvm_phase phase);

    super.run_phase(phase);

    fork
       begin : write_monitor
           forever @(negedge vif.wclk) begin

               mon_write();
           end
       end

       begin : write_completion

           wait (w_count == trans_count_write);


       end
    join

endtask

task mon_write;

    transaction_write txw;

    if (vif.write_enable == 1)
    begin
        txw = transaction_write::type_id::create("txw");
        txw.write_enable = vif.write_enable;
        txw.data_write = vif.data_write;
        $display("\t Monitor write_enable = %0h \t data_write = %0h \t
            w_count = %0d", txw.write_enable, txw.data_write, w_count + 1);
        port_write.write(txw);
        w_count = w_count + 1;
    end
endtask
```

UVM based Verification (8 mins)

Agent (write) :

```
class write_agent extends uvm_agent;
    `uvm_component_utils(write_agent)

    write_sequencer ws;
    write_driver wd;
    write_monitor wm;

    function new (string name = "write_agent", uvm_component parent);
        super.new(name, parent);
        `uvm_info("WRITE_AGENT_CLASS", "Inside constructor",UVM_LOW);
    endfunction

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        ws = write_sequencer::type_id::create("ws", this);
        wd = write_driver::type_id::create("wd", this);
        wm = write_monitor::type_id::create("wm", this);
    endfunction

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        wd.seq_item_port.connect(ws.seq_item_export);
    endfunction

    task run_phase(uvm_phase phase);
        super.run_phase(phase);
    endtask
endclass
```

Scoreboard :

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    write_port = new("write_port", this);
    read_port = new("read_port", this);
endfunction

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
endfunction

function void write_port_a(transaction_write txw);
    tw.push_back(txw);
    $display ("\t Value of the Scoreboard data_write = %0h", txw.data_write);
endfunction

function void write_port_b(transaction_read txr);

logic [8:0] popped_data_write;
empty_count = tw.size;

    if (tw.size() > 0) begin
        popped_data_write = tw.pop_front().data_write;

        if (popped_data_write == txr.data_read)
            `uvm_info("ASYNC_FIFO_SCOREBOARD",
                $sformatf("TestBench PASSED ScoreBoard  Data: %0h --- DUT FIFO
                    Read Data: %0h", popped_data_write, txr.data_read), UVM_MEDIUM)
        else
            `uvm_error("ASYNC_FIFO_SCOREBOARD",
                $sformatf("TestBench Failed ScoreBoard Data: %0h Does not match DUT FIFO
                    Read Data: %0h", txr.data_read, popped_data_write))
    end
endfunction
```

UVM based Verification (8 mins)

Environment :

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

    virtual intf vif;

    write_agent wa;
    read_agent ra;
    fifo_scoreboard scb;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        wa = write_agent::type_id::create("wa", this);
        ra = read_agent::type_id::create("ra", this);
        scb = fifo_scoreboard::type_id::create("scb", this);

        if (!uvm_config_db#(virtual intf)::get(this, "", "vif", vif))
        begin
          `uvm_fatal("build phase", "No virtual interface specified for this env instance")
        end

    endfunction

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        wa.wm.port_write.connect(scb.write_port);
        ra.rm.port_read.connect(scb.read_port);
    endfunction

    task run_phase (uvm_phase phase);
        super.run_phase(phase);
    endtask
endclass
```

Test :

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = fifo_env::type_id::create("env", this);

    if (!uvm_config_db#(virtual intf)::get(this, "", "vif", vif)) begin
        `uvm_fatal("FIFO/DRV/NOVIF", "No virtual interface specified for this test instance")
    end
endfunction

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
endfunction

function void end_of_elaboration();
    super.end_of_elaboration();
    uvm_root::get().print_topology();
endfunction


task run_phase(uvm_phase phase );
    env.wa.wd.trans_count_write=400;
    env.ra.rd.trans_count_read=401;

    env.wa.wm.trans_count_write=400;
    env.ra.rm.trans_count_read=401;

    phase.raise_objection(this, "Starting fifo_write_seq in main phase");

    fork
        begin
            $display("/t Starting sequence w_seq run_phase");
            w_seq = write_sequence::type_id::create("w_seq", this);

            w_seq.start(env.wa.ws);
        end
        begin
            $display("/t Starting sequence r_seq run_phase");
            r_seq = read_sequence::type_id::create("r_seq", this);
            r_seq.start(env.ra.rs);
        end
    join

    #100ns;

    env.scb.compare_flags();
    phase.drop_objection(this , "Finished fifo_seq in main phase");


    #2000;
    $finish;
endtask
```

UVM based Verification (8 mins)

Coverage :

```
covergroup FIFO_coverage;

  coverpoint intf.data_write {
    bins data_bin[] = {[0:511]};
  }


  coverpoint intf.data_read {
    bins data_bin[] = {[0:511]};
  }


  coverpoint intf.wfull {
    bins full_bin[] = {0, 1};
  }


  coverpoint intf.rempty {
    bins empty_bin[] = {0, 1};
  }


  cross intf.data_write, intf.data_read;


  cross intf.data_write, intf.wfull;


  cross intf.data_read, intf.rempty;


endgroup


  FIFO_coverage fifo_coverage_inst;
  initial begin
    fifo_coverage_inst = new();
    forever begin @(posedge wclk or posedge rclk)
      fifo_coverage_inst.sample();
    end
  end
```

Top :

```
module tb_top;

    bit rclk, wclk, rrst_n, wrst_n;
    always #21ns wclk = ~wclk;
    always #30ns rclk = ~rclk;

    intf intf (wclk, rclk, wrst_n, rrst_n);

    ASYNC_FIFO DUT (
        .data_write(intf.data_write),
        .wfull(intf.wfull),
        .rempty(intf.rempty),
        .write_enable(intf.write_enable),
        .read_enable(intf.read_enable),
        .wclk(intf.wclk),
        .rclk(intf.rclk),
        .rrst_n(intf.rrst_n),
        .wrst_n(intf.wrst_n),
        .data_read(intf.data_read),
        .wHalf_empty(intf.wHalf_empty),
        .wHalf_full(intf.wHalf_full)
    );

    initial begin
        uvm_config_db#(virtual intf)::set(null, "*", "vif", intf);
        `uvm_info("tb_top", "uvm_config_db set for uvm_tb_top", UVM_LOW);
    end

    initial begin
        run_test("fifo_base_test");
       // run_test("fifo_full_test");
       // run_test("fifo_random_test");
    end

    initial begin
        wclk = 0;
        rclk = 0;
        wrst_n = 0;
        rrst_n = 0;
        intf.read_enable = 0;
        intf.write_enable = 0;
        #1;
        rrst_n = 1;
        wrst_n = 1;
    end
```

Overall Challenges and Learnings

- **Challenges Faced :**
  - **Coverage Closure :** Achieving 100% code coverage is some what easy but getting Functional Coverage to some extent is challenging, we need to create so many bins to get to know that the values are covered during verification and analyzing the transactions between one value to the another value implemented in the coverage is some what difficulty we faced.
  - **Clock Domain :** Asynchronous FIFOs inherently involve different clock domains for the write and read operations. So while creating the Design and monitor and driver class we faced some challenges on where and which clock domain that needs to specify the read and write operations.
  - **Connecting Modules :** We have faced some difficulty while doing the UVM verification in connecting the modules and remembering the inbuilt class names and method names. This challenges is overcomes by your lectures and some web resources.
- **Learnings :**
  - Clear understanding in the clock domain of write and read operations.
  - Clear understanding on how to improve the coverage i.e., code and functional coverage.
  - Can be able to get how the modules are connected in class based verification.

Test Scenarios and Coverage

Quality of test scenarios

- How many test scenarios covered
  - Data Write -> tested data write is happening for all range of the input values.
  - Write Full -> tested that full condition is happening or not by using transition bins
  - Data Read -> tested data read for all range of the values.
  - Read Empty -> tested that empty condition is happening or not.
  - Write pointer -> tested that increment of the write pointer is happening or not.
  - Read Pointer -> tested that increment in the read pointer is happening or not.
  - Reset -> tested that reset is happening through transition bins
- Code Coverage Metrics
  - 90.40%
- Functional Coverage Metrics
  - 84.16%

Demo

- Presenting the project in the action
- Executed within UVM verification Environment
- Running in Questa sim for demonstration
- Bug-injected scenario ready to show

Conclusion

- Successful design and verification of asynchronous FIFO using System Verilog (SV) and Universal Verification Methodology (UVM) ensure correctness and reliability.
- Employed techniques such as code functional coverage, assertions, and careful design implementation contributed to the robustness of the verification process.
- UVM methodologies facilitate reusability, and scalability, enhancing efficiency in verification efforts.
- Directed testing focused on critical scenarios, while random testing uncovered edge cases, ensuring comprehensive coverage.
- Integration of these techniques guarantees that the design meets requirements and operates reliably in diverse scenarios.