Out: Thursday, October 1, 2020
Due: Monday, October 5, 2020

The goal of this practicum is to become familiar with SystemVerilog Object Oriented Programming (OOP) and development phases of testbench architecture. The testbench architecture that was explained in classis copied below. Note that testbench is a program, while other verification components are classes.

```
top module: tbench_top
        |- testbench program: test
                |- environment class: env
                        |- driver class: driv
                        |- generator class: gen
                        |- monitor class: mon
                        |- scoreboard class: sb
        |- interface: intf
        |- <design> module: dut
```

This assignment will prepare you for the upcoming lab. Step-by-step instructions contained in this document will guide you through the process of creating an OOP-based testbench to verify a 4-bit adder. In the lab, you will follow the same development process to develop an OOP-based testbench for a more complex and large design.

Procedures

1. Log into a lab machine.
2. Create a cme435_ex3 folder.
3. Download and unzip the exercise files

Phase I: Top

4. Group signals using **interface**.

   4.1. Open the file named interface.sv.

```
interface intf();

endinterface
```

   4.2. Complete Interface including clk, reset, valid, a, b, and c signals, where clk and reset should be external signals.

```
interface intf(input logic clk, reset);
  logic       valid;
  logic [3:0] a;
  logic [3:0] b;
  logic [6:0] c;
endinterface
```

5. **Testbench** is a program block which is responsible for creating the environment, configuring the testbench (e.g. setting the type and number of transactions to be generated), and initiating the stimulus driving.

Copyright © 2020. All rights reserved.

1.1. Open the file named test.sv.

```
program testbench(intf i_intf);

endprogram
```

1.2. Add an initial block for test cases.

```
program testbench(intf i_intf);

  initial begin
    $display("****************** Start of testcase ***************");
  end

  final
    $display("****************** End of testcase ***************");

endprogram
```

6. Create a wrapper module in a file named adder_top.sv so that the DUT can later be connected to the testbench by using the interface, i_intf, without the need for port mapping.

```
module dut_top(interface i_intf);

  adder adder_core (
    .clk(i_intf.clk),
    .reset(i_intf.reset),
    .a(i_intf.a),
    .b(i_intf.b),
    .valid(i_intf.valid),
    .c(i_intf.c)
   );

endmodule
```

7. The top module connects testbench and DUT. Hence, the top module instantiates the test program, the DUT, and the interface. Clock and reset can also be declared in the top module.

7.1. Open the file named tbench_top.sv.

```
module tbench_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
endmodule
```

7.2. Instantiate the interface.

```
`include "interface.sv"
module tbench_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
  intf i_intf(clk, reset);

endmodule
```

7.3. Instantiate the test program and connect it to the interface.

```
`include "interface.sv"
`include "testbench.sv"
module tbench_top;
  // ...

  // Creating instance of interface, in order to connect DUT and testcase
  intf i_intf(clk,reset);

  //Testcase instance, interface handle is passed to test as an argument
  testbench t1(i_intf);

endmodule
```

7.4. Instantiate DUT and connect it to the interface.

```
`include "interface.sv"
`include "testbench.sv"
module tbench_top;
  // ...

  // Creating instance of interface, in order to connect DUT and testcase
  intf i_intf(clk, reset);

  //Testcase instance, interface handle is passed to test as an argument
  test t1(i_intf);

  //DUT instance, interface signals are connected to the DUT ports
  dut_top dut(i_intf);

endmodule
```

7.5. Optionally, enable wave dump.

```
module tbench_top;
  // ...

  //enabling the wave dump
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

8. Compile and launch the simulation. Debug the testbench if there are errors. Note: Use *+incdir+* switch with vlog; e.g. vlog testbench/testbench.sv *+incdir+"./testbench"*.

Phase II: Environment

9. **Environment class** is a container class that contains verification components.

9.1. Open the file named environment.sv. Note that an interface must be declared as virtual interface in a class.

```
class environment;

  //virtual interface
  virtual intf vif;

  //constructor
  function new(virtual intf vif);
    //get the interface from test
    this.vif = vif;

  endfunction

endclass
```

9.2. Add three placeholders for pre-test, test and post-test tasks:

```
class environment;

  // ...

  task pre_test();
    $display("%0d : Environment : start of pre_test()", $time);
    $display("%0d : Environment : end of pre_test()", $time);
  endtask

  task test();
    $display("%0d : Environment : start of test()", $time);
    $display("%0d : Environment : end of test()", $time);
  endtask

  task post_test();
    $display("%0d : Environment : start of post_test()", $time);
    $display("%0d : Environment : end of post_test()", $time);
  endtask

endclass
```

9.3. Add a run task that calls pre_test(), test(), and post_test().

```
class environment;

  // ...

  //run task
  task run;
    $display("%0d : Environment : start of run()", $time);
    pre_test();
    test();
    post_test();
    $display("%0d : Environment : end of run()", $time);
    $finish;
  endtask

endclass
```

9.4. In the testbench program, declare and construct environment.

```
`include "environment.sv"
program testbench(intf i_intf);

  //declaring environment instance
```

```
  environment env;

  initial begin
    //creating environment
    env = new(i_intf);
    $display("****************** Start of testcase ***************");
  end

  final
    $display("****************** End of testcase ***************");

endprogram
```

1.3. Initiate testing by calling the run() task in the environment class.

```
`include "environment.sv"
program testbench(intf i_intf);

  //declaring environment instance
  environment env;

  initial begin
    //creating environment
    env = new(i_intf);

    $display("****************** Start of testcase ***************");

    //calling run of env, it in turns calls other main tasks.
    env.run();
  end

  final
    $display("****************** End of testcase ***************");

endprogram
```

10. Compile and launch the simulation. Debug the testbench if there are errors.

11. Add a reset task. Note the reset task waits for reset being asserted and deasserted.

```
class environment;

  // ...

  task pre_test();
    $display("%0d : Environment : start of pre_test()", $time);
    reset();
    $display("%0d : Environment : end of pre_test()", $time);
  endtask

  task reset();
    wait(vif.reset);
    $display("[ ENVIRONMENT ] ----- Reset Started -----");
    vif.a <= 0;
    vif.b <= 0;
    vif.valid <= 0;
    wait(!vif.reset);
    $display("[ ENVIRONMENT ] ----- Reset Ended   -----");
  endtask

   // ...
```

```
endclass
```

12. Compile and launch the simulation. Debug the testbench if there are errors.

Phase III: Base Classes

13. **Transaction class** contains fields required to generate the stimulus. Transaction class can also be used as a placeholder for the activity monitored by monitor on DUT signals.

    13.1. Create a file named transaction.sv.

```
class transaction;

endclass
```

    13.2. Declare the fields for transaction class.

```
class transaction;
  bit [3:0] a;
  bit [3:0] b;
  bit [6:0] c;
endclass
```

    13.3. Add display() method to display transaction properties.

```
class transaction;
  bit [3:0] a;
  bit [3:0] b;
       bit [6:0] c;
  function void display(string name);
    $display("- %s ",name);
    $display("-------------------------");
    $display("- a = %0d, b = %0d",a,b);
    $display("- c = %0d",c);
  endfunction
endclass
```

    13.4. Base classes are normally tested in a separate program. Create a file named test_transaction_class.sv

```
`include "transaction.sv"
program test_transaction_class;

transaction trans;

initial begin
  trans = new();
  repeat(10) begin
    trans.a = random();
    trans.b = random();
    trans.display("[test_transaction_class]");
  end
end

endprogram : test_transaction_class
```

    13.5. Compile and launch the simulation. Debug the code if there are errors.

Phase IV: Generator

14. **Generator class** is responsible for generating the stimulus by randomizing the transaction class and send the randomized class to the driver.

14.1. Create a file named generator.sv.

```
class generator;

endclass
```

14.2. Declare the transaction class handle.

```
class generator;
  //declaring transaction class
  rand transaction trans;
endclass
```

14.3. Generate random inputs.

```
class generator;
  //declaring transaction class
  rand transaction trans;

  //main task, generates (creates and randomizes) transaction packets
  task main();
    trans = new();
    trans.a = $random();
    trans.b = $random();
  endtask

endclass
```

14.4. Display the randomized transaction packet.

```
class generator;
  // ...

  //main task, generates (creates and randomizes) transaction packets
  task main();
    trans = new();
    trans.a = $random();
    trans.b = $random();
    trans.display("[ Generator ]");
  endtask

endclass
```

14.5. Add a variable, repeat_count, to control the number of packets to be created.

```
class generator;
  // ...

  //repeat count, to specify number of items to generate
  int  repeat_count;
  //main task, generates (creates and randomizes) the repeat_count number
of transaction packets
  task main();
    repeat(repeat_count) begin
      trans = new();
");
      trans.a = $random();
      trans.b = $random();
      trans.display("[ Generator ]");
    end
  endtask
```

```
endclass
```

14.6. In environment, declare and construct the generator.

```
`include "transaction.sv"
`include "generator.sv"
class environment;

  //generator instance
  generator gen;

  //virtual interface
  virtual intf vif;

  //constructor
  function new(virtual intf vif);
    //get the interface from test
    this.vif = vif;
    //creating generator
    gen = new();
  endfunction

  // ...

endclass
```

14.7. In testbench, configure the number of transactions to be generated.

```
`include "environment.sv"
program testbench(intf i_intf);

  // ...

  initial begin
    //creating environment
    env = new(i_intf);

    //setting the repeat count of generator such as 5, means to generate 5
packets
    env.gen.repeat_count = 5;

    $display("****************** Start of testcase ***************");

    //calling run of env, it in turns calls other main tasks.
    env.run();
  end

  // ...

endprogram
```

15. Compile and launch the simulation. Debug the testbench if there are errors.

16. In SystemVerilog, the built-in **mailbox** class can be used for exchanging messages between tasks.

    16.1. To use a mailbox for sending randomized transaction packets to the driver, declare the mailbox first. Since the mailbox is shared by the generator and driver, a mailbox named **gen2drive** should be declared in the **environment class**.

```
`include "transaction.sv"
```

```
`include "generator.sv"
class environment;

  //generator instance
  generator gen;

  //mailbox handle's
  mailbox gen2driv;

  //virtual interface
  virtual intf vif;

  //constructor
  function new(virtual intf vif);
    //get the interface from test
    this.vif = vif;
    //creating the mailbox (Same handle will be shared across generator and
driver)
    gen2driv = new();

    //creating generator
    //gen = new();
    gen  = new(gen2driv);
  endfunction

  // ...

endclass
```

16.2. For the generator class to use the mailbox in, simply get the mailbox handle from the **environment class**.

```
class generator;
  // ...

  //mailbox, to generate and send the packet to driver
  mailbox gen2driv;

  //constructor
  function new(mailbox gen2driv);
    //getting the mailbox handle from env.
    this.gen2driv = gen2driv;
  endfunction

  // ...
endclass
```

16.3. To place a message in a mailbox, use the **put()** method that is provided by the mailbox class.

```
class generator;
  // ...

  //mailbox, to generate and send the packet to driver
  mailbox gen2driv;
  //constructor
  function new(mailbox gen2driv);
    //getting the mailbox handle from env.
    this.gen2driv = gen2driv;
  endfunction
```

```
  //main task, generates (creates and randomizes) the repeat_count number
of transaction packets
  task main();
    repeat(repeat_count) begin
      trans = new();
      if( !trans.randomize() ) $fatal("Gen:: trans randomization failed");
      trans.display("[ Generator ]");
      gen2driv.put(trans);
    end
  endtask
endclass
```

17. Compile and launch the simulation. Debug the testbench if there are errors.

Phase V: Driver

18. **Driver class** receives stimulus generated by the generator and drives DUT by assigning transaction class values to interface signals.

    18.1. Create a file named driver.sv.

```
class driver;

endclass
```

    18.2. Declare the interface.

```
class driver;
  //creating virtual interface handle
  virtual intf vif;
  //constructor
  function new(virtual intf vif);
    //getting the interface
    this.vif = vif;
  endfunction
endclass
```

    18.3. Add the drive task to drive transaction packets to interface signals.

```
class driver;
  //creating virtual interface handle
  virtual intf vif;

  //constructor
  function new(virtual intf vif);
    //getting the interface
    this.vif = vif;
  endfunction

  //drive the transaction items to interface signals
  task main;
    forever begin

    end
  endtask
endclass
```

    18.4. Recall that transaction packets generated by the generator class are placed in the mailbox named gen2drive. In order to drive the transaction packets, first of all, use the **get()** method provided by the mailbox class to get the transaction from **environment class**.

```
class driver;
```

```
  //creating virtual interface handle
  virtual intf vif;
  //creating mailbox handle
  mailbox gen2driv;

  //constructor
  function new(virtual intf vif, mailbox gen2driv);
    //getting the interface
    this.vif = vif;
    //getting the mailbox handles from environment
    this.gen2driv = gen2driv;
  endfunction

  //drive the transaction items to interface signals
  task main;
    forever begin
      transaction trans;
      gen2driv.get(trans);
    end
  endtask
endclass
```

18.5. Add a main method to drive the packets to interface signals.

```
class driver;
  // ...

  //drive the transaction items to interface signals
  task main;
    forever begin
      transaction trans;
      gen2driv.get(trans);
      @(posedge vif.clk);
      vif.valid <= 1;
      vif.a     <= trans.a;
      vif.b     <= trans.b;
      @(posedge vif.clk);
      vif.valid <= 0;
      trans.c   = vif.c;
      @(posedge vif.clk);
      trans.display("[ Driver ]");
    end
  endtask
endclass
```

18.6. Add a local variable, no_transactions, to track the number of packets driven, and increment the variable in drive task. Note: This local variable is useful for ending a test case; for example, end simulation if the generated packets and driven packets are equal.

```
class driver;
  //used to count the number of transactions
  int no_transactions;

  // ...

  task main;
    forever begin
      transaction trans;
      gen2driv.get(trans);
      @(posedge vif.clk);
      vif.valid <= 1;
```

```
      vif.a     <= trans.a;
      vif.b     <= trans.b;
      @(posedge vif.clk);
      vif.valid <= 0;
      trans.c   = vif.c;
      @(posedge vif.clk);
      trans.display("[ Driver ]");
      no_transactions++;
    end
  endtask
endclass
```

18.7. In environment, declare and construct the driver.

```
`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
class environment;

  //generator and driver instance
  generator gen;
  driver    driv;

  //mailbox handle's
  mailbox gen2driv;

  //virtual interface
  virtual intf vif;

  //constructor
  function new(virtual intf vif);
    //get the interface from test
    this.vif = vif;

    //creating the mailbox (Same handle will be shared across generator and
driver)
    gen2driv = new();

    //creating generator and driver
    // gen = new();
    gen  = new(gen2driv);
    driv = new(vif,gen2driv);
  endfunction

  // ...

  task test();
    $display("%0d : Environment : start of test()", $time);
    fork
      gen.main();
      driv.main();
    join
    $display("%0d : Environment : end of test()", $time);
  endtask

  // ...

endclass
```

19. Compile and run the simulation. Debug the testbench if there are errors.

20. Note forever block is used in the driver. Stop simulation when generator stops generates new transactions.

    20.1. Optionally add an event in the generator.

```
class generator;
  // ...

  //event, to indicate the end of transaction generation
  event ended;

  // ...

  //main task, generates (creates and randomizes) the repeat_count number
of transaction packets
  task main();
    repeat(repeat_count) begin
      trans = new();
      if( !trans.randomize() ) $fatal("Gen:: trans randomization failed");
      trans.display("[ Generator ]");
      gen2driv.put(trans);
    end
    -> ended; //trigger end of generation
  endtask
endclass
```

    20.2. Modify the environment accordingly.

```
`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
class environment;

  // ...

  task test();
    $display("%0d : Environment : start of test()", $time);
    fork
      gen.main();
      driv.main();
    join_any
    wait(gen.ended.triggered); // optional
    wait(gen.repeat_count == driv.no_transactions);
    $display("%0d : Environment : end of test()", $time);
  endtask

  // ...

endclass
```

21. Compile and run the simulation. Debug the testbench if there are errors.

Phase VI: Monitor

22. **Monitor class** samples DUT output signals through the interface and convert the signals to a transaction. The transaction is then sent the scoreboard via a mailbox. The process is quite similar to that used for the generator and driver.

    22.1. Create a file named monitor.sv.

```
class monitor;

```

```
endclass
```

22.2. Declare interface.

```
class monitor;
  //creating virtual interface handle
  virtual intf vif;
endclass
```

22.3. Declare a mailbox named mon2scb.

```
class monitor;
  //creating virtual interface handle
  virtual intf vif;
  //creating mailbox handle
  mailbox mon2scb;
endclass
```

22.4. Get the interface and mailbox handles from **environment** through constructor.

```
class monitor;
  //creating virtual interface handle
  virtual intf vif;
  //creating mailbox handle
  mailbox mon2scb;
  //constructor
  function new(virtual intf vif, mailbox mon2scb);
    //getting the interface
    this.vif = vif;
    //getting the mailbox handles from  environment
    this.mon2scb = mon2scb;
  endfunction
endclass
```

22.5. Add a main method to sample interface signals and convert the signals into a transaction.

```
class monitor;
  // ...

  task main;
    forever begin
      transaction trans;
      trans = new();
      @(posedge vif.clk);
      wait(vif.valid);
      trans.a   = vif.a;
      trans.b   = vif.b;
      @(posedge vif.clk);
      trans.c   = vif.c;
      @(posedge vif.clk);
    end
  endtask
endclass
```

22.6. Send the transaction to the scoreboard via the mailbox mon2scb, and also display the transaction.

```
class monitor;
  // ...

  task main;
    forever begin
      transaction trans;
```

```
      trans = new();
      @(posedge vif.clk);
      wait(vif.valid);
      trans.a   = vif.a;
      trans.b   = vif.b;
      @(posedge vif.clk);
      trans.c   = vif.c;
      @(posedge vif.clk);
      mon2scb.put(trans);
      trans.display("[ Monitor ]");
    end
  endtask
endclass
```

22.7. Modify the environment

```
`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
class environment;

  //generator and driver instance
  generator gen;
  driver    driv;
  monitor   mon;

  //mailbox handle's
  mailbox gen2driv;
  mailbox mon2scb;

  //virtual interface
  virtual intf vif;

  //constructor
  function new(virtual intf vif);
    //get the interface from test
    this.vif = vif;

    //creating the mailbox (Same handle will be shared across generator and
driver)
    gen2driv = new();
    mon2scb  = new();

    //creating generator, driver and monitor
    // gen = new();
    gen  = new(gen2driv);
    driv = new(vif,gen2driv);
    mon  = new(vif,mon2scb);
  endfunction

  // ...

  task test();
    $display("%0d : Environment : start of test()", $time);
    fork
      gen.main();
      driv.main();
      mon.main();
    join_any
    wait(gen.ended.triggered); // optional
    wait(gen.repeat_count == driv.no_transactions);
```

```
    $display("%0d : Environment : end of test()", $time);
  endtask

  // ...

endclass
```

23. Compile and run the simulation. Debug the testbench if there are errors.

Phase VII: Scoreboard

24. **Scoreboard class** compare the transaction packets received from monitor with the expected results. The scoreboard class also reports errors if there are mismatches.

    24.1. Create a file named scoreboard.sv.

```
class scoreboard;

endclass
```

    24.2. Recall that packets are to be received via the mailbox mon2scb. Declare the mailbox and connect the handle through constructor.

```
class scoreboard;
  //creating mailbox handle
  mailbox mon2scb;

  //constructor
  function new(mailbox mon2scb);
    //getting the mailbox handles from environment
    this.mon2scb = mon2scb;
  endfunction
endclass
```

    24.3. Use the built-in get() method to get transaction from the mailbox.

```
class scoreboard;
  //creating mailbox handle
  mailbox mon2scb;

  //constructor
  function new(mailbox mon2scb);
    //getting the mailbox handles from environment
    this.mon2scb = mon2scb;
  endfunction
  //Compares the actual result with the expected result
  task main;
    transaction trans;
    forever begin
      mon2scb.get(trans);
    end
  endtask
endclass
```

    24.4. Compare the received packets with the expected packets, and display the received transaction.

```
class scoreboard;
  // ...

  task main;
    transaction trans;
```

```
    forever begin
      mon2scb.get(trans);
      if((trans.a+trans.b) == trans.c)
        $display("Result is as Expected");
      else
        $error("Wrong     Result.\n\tExpeced:     %0d     Actual:     %0d",
(trans.a+trans.b), trans.c);
      trans.display("[ Scoreboard ]");
    end
  endtask
endclass
```

24.5. Declare a variable no_transaction to keept count of the received transactions. Display the count after each comparison.

```
class scoreboard;
  // ...

  //count the number of transactions
  int no_transactions;
  //Compares the actual result with the expected result
  task main;
    transaction trans;
    forever begin
      mon2scb.get(trans);
      if((trans.a+trans.b) == trans.c)
        $display("Result is as Expected");
      else
        $error("Wrong     Result.\n\tExpeced:     %0d     Actual:     %0d",
(trans.a+trans.b), trans.c);
      no_transactions++;
      trans.display("[ Scoreboard ]");
    end
  endtask
endclass
```

25. To add monitor and scoreboard to the testbench, edit environment.sv to update **environment class**.

```
`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
`include "scoreboard.sv"
class environment;

  //generator and driver instance
  generator    gen;
  driver       driv;
  monitor      mon;
  scoreboard   scb;

  // ...

  //constructor
  function new(virtual intf vif);
    //get the interface from test
    this.vif = vif;

    //creating the mailbox (Same handle will be shared across generator and
driver)
    gen2driv = new();
```

```
    mon2scb  = new();

    //creating generator and driver
    gen  = new(gen2driv);
    driv = new(vif,gen2driv);
    mon  = new(vif,mon2scb);
    scb  = new(mon2scb);
  endfunction

  // ...

  task test();
    $display("%0d : Environment : start of test()", $time);
    fork
      gen.main();
      driv.main();
      mon.main();
      scb.main();
    join_any
    wait(gen.ended.triggered); // optional
    wait(gen.repeat_count == driv.no_transactions);
    wait(gen.repeat_count == scb.no_transactions);
    $display("%0d : Environment : end of test()", $time);
  endtask

  // ...

endclass
```

26. Compile and run the simulation. Debug the testbench if there are errors.
27. Add an error_count to the testbench. In the post_test() task, check the error_count and display a message indicating whether a test is passed or failed.