Out: Tuesday, September 27, 2022
Due: Monday, October 3, 2022

The goal of this practicum is to become familiar with SystemVerilog Object Oriented Programming (OOP) and development phases of testbench architecture. The testbench architecture explained in class is 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
        |- wrapper module: dut_top
                |-> <dut_core> module: dut_core
```

This practicum 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/subtractor. In the lab, you will follow a similar development process to develop an OOP-based testbench for a design. For each step, text highlighted in blue means new code for that step.

Important: System function $random(), although can be used for randomization during the testbench development using SystemVerilog OOP, this random number generator is not allowed to use for this exercise as well as for the remainder of the class.

Procedures

1. Log into a lab computer or a computer that allows running Queata SIM in batch mode.
2. Download and unzip the cme_ex3.zip file.

Phase I: Top

3. Group signals using **interface**.

   3.1. Open the file named interface.sv.

```
interface intf();

endinterface
```

   3.2. Complete Interface including clk, reset, en, op, a, b, and y signals, where clk and reset should be external signals. Also, add an optional include guard.

```
`ifndef _INTERFACE_
`define _INTERFACE_

interface intf(input logic clk, reset);
  logic       en;
```

```
  logic       op;
  logic [3:0] a;
  logic [3:0] b;
  logic [7:0] y;
endinterface

`endif
```

4. **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.

   4.1. Open the file named test.sv.

```
program testbench(intf i_intf);

endprogram
```

   4.2. Add an initial block for test cases.

```
program testbench(intf i_intf);

  initial begin
    $display("[Testbench]: Start of testcase(s) at %0d", $time);
  end

  final
    $display("[Testbench]: End of testcase(s) at %0d", $time);

endprogram
```

5. Create a wrapper module in a file named dut_top.sv in the dut folder 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);

  alu alu_core (
    .clk(i_intf.clk),
    .reset(i_intf.reset),
    .en(i_intf.en),
    .op(i_intf.op),
    .a(i_intf.a),
    .b(i_intf.b),
    .y(i_intf.y)
   );

endmodule
```

6. 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.

   6.1. 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;
    #6 reset = 0;
  end
endmodule
```

6.2. Instantiate the interface. Alternatively, interface.sv can also be compiled separately.

```
`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;
    #6 reset = 0;
  end
  // create interface instance to connect DUT and testcase
  intf i_intf(clk, reset);

endmodule
```

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

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

  // create interface instance to connect DUT and testcase
  intf i_intf(clk, reset);

  // create testcase instance where interface handle is passed to test as
an argument
  testbench test(i_intf);

endmodule
```

6.4. Instantiate DUT and connect it to the interface.

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

  // create interface instance to connect DUT and testcase
  intf i_intf(clk, reset);

  // create testcase instance where interface handle is passed to test as
an argument
  testbench test(i_intf);

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

endmodule
```

6.5. Optionally, enable wave dump.

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

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

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

8. Create shell and Tcl "DO" scripts to compile files by executing script/run.csh. Note: The *+incdir+* switch can also be added to the top of –f file.

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("[Environment]: Start of pre_test() at %0d", $time);
    $display("[Environment]: End of pre_test() at %0d", $time);
  endtask

  task test();
    $display("[Environment]: Start of test() at %0d", $time);
    $display("[Environment]: End of test() at %0d", $time);
  endtask

  task post_test();
    $display("[Environment]: Start of post_test() at %0d", $time);
    $display("[Environment]: End of post_test() at %0d", $time);
  endtask

endclass
```

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

```
class environment;

  // ...

  // run task
  task run;
    $display("[Environment]: Start of run() at %0d", $time);
    pre_test();
    test();
    post_test();
    $display("[Environment]: End of run() at %0d", $time);
    $finish;
  endtask

endclass
```

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

```
program testbench(intf i_intf);

  // declare environment instance
  environment env;

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

    $display("[Testbench]: Start of testcase(s) at %0d", $time);
  end

  final
    $display("[Testbench]: End of testcase(s) at %0d", $time);

endprogram
```

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

```
program testbench(intf i_intf);

  // declare environment instance
  environment env;

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

    $display("[Testbench]: Start of testcase(s) at %0d", $time);

    // call the run task in env, which in turns calls other test tasks
    env.run();
  end

  final
    $display("[Testbench]: End of testcase(s) at %0d", $time);

endprogram
```

10. Include environment.sv in tbench_top.sv.

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

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

```
class environment;

  // ...

  task pre_test();
    $display("[Environment]: Start of pre_test() at %0d", $time);
    reset();
    $display("[Environment]: End of pre_test() at %0d", $time);
  endtask

  task reset();
    wait(vif.reset);
    $display("[Environment]: Reset started at %0d", $time);
    vif.en <= 0;
    vif.op <= 0;
    vif.a  <= 0;
    vif.b  <= 0;
    wait(!vif.reset);
    $display("[Environment]: Reset ended at %0d", $time);
  endtask

   // ...

endclass
```

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

Phase III: Base Classes

14. **Transaction class** contains fields required to generate the stimulus. Transaction class can also be used as a placeholder for activities to be monitored by the monitor.

    14.1. Create a file named transaction.sv.

```
class transaction;

endclass
```

    14.2. Declare the fields for transaction class.

```
class transaction;
       bit       op;
  rand bit [3:0] a;
  rand bit [3:0] b;
       bit [7:0] y;
endclass
```

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

```
class transaction;
       bit       op;
  rand bit [3:0] a;
  rand bit [3:0] b;
       bit [7:0] y;
  function void display(string name);
    $display("-------------------------");
    $display(" %s: a = %0d, b = %0d", name, a, b);
    $display(" %s: op = %0d: y = %0d", name, op, y);
  endfunction
endclass
```

14.4. 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
    if( !trans.randomize() ) $fatal("Gen:: trans randomization failed");
    trans.display("[test_transaction_class]");
  end
end

endprogram : test_transaction_class
```

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

Phase IV: Generator

15. **Generator class** is responsible for generating transaction packages and sending them to the driver.

15.1. Create a file named generator.sv in the verification folder and add an optional include guard.

```
class generator;

endclass
```

15.2. Declare the transaction class handle.

```
class generator;
  // declare transaction class
  transaction trans;
endclass
```

15.3. Generate random inputs.

```
class generator;
  // declare transaction class
  transaction trans;

  bit op_code;

  // main task to generate (create and randomize) transaction packets
  task main();
    trans = new();
    if( !trans.randomize() ) $fatal("Gen:: trans randomization failed");
    trans.op = op_code;
  endtask

endclass
```

15.4. Display the randomized transaction packet.

```
class generator;
  // ...

  // main task to generate (create and randomize) transaction packets
  task main();
```

```
    trans = new();
    if( !trans.randomize() ) $fatal("Gen:: trans randomization failed");
    trans.op = op_code;
    trans.display("[Generator]");
  endtask

endclass
```

15.5. Add a variable, repeat_count, to control the number of packets to be created. Note: The value of this variable will be set in the testbench program.

```
class generator;
  // ...

  // repeat count, to specify number of items to generate
  int  repeat_count;
  // main task to generate (create and randomize) the repeat_count number
of transaction packets
  task main();
    repeat(repeat_count) begin
      trans = new();
      if( !trans.randomize() ) $fatal("Gen:: trans randomization failed");
      trans.op = op_code;
      trans.display("[Generator]");
    end
  endtask
endclass
```

15.6. In environment, declare and construct the generator, and run the main task in the generator.

```
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;
    // create generator
    gen = new();
  endfunction

  // ...
  task test();
    $display("[Environment]: Start of test() at %0d", $time);
    gen.main();
    $display("[Environment]: End of test() at %0d", $time);
  endtask
  // ...

endclass
```

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

```
program testbench(intf i_intf);

  // ...
```

```
  initial begin
    // construct environment instance
    env = new(i_intf);

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

    $display("[Testbench]: Start of testcase(s) at %0d", $time);

    // call the run task in env, which in turns calls other test tasks.
    env.run();
  end

  // ...

endprogram
```

16. Include transaction.sv and generator.sv in tbench_top.sv.

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

18. In SystemVerilog, the built-in **mailbox** class can be used for exchanging data.

    18.1. 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**.

```
class environment;

  // generator instance
  generator gen;

  // mailbox handles
  mailbox gen2driv;

  // virtual interface
  virtual intf vif;

  // constructor
  function new(virtual intf vif);
    // get the interface from test
    this.vif = vif;
    // create mailbox(es) for data exchange
    gen2driv = new();

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

  // ...

endclass
```

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

```
class generator;
  // ...
```

```
  // mailbox for sending packets to the driver
  mailbox gen2driv;

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

  // ...
endclass
```

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

```
class generator;
  // ...

  // mailbox to send packets to the driver
  mailbox gen2driv;
  // constructor
  function new(mailbox gen2driv);
    // get the mailbox handle from env.
    this.gen2driv = gen2driv;
  endfunction

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

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

Phase V: Driver

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

20.1. Create a file named driver.sv in the verification folder and add an optional include guard.

```
class driver;

endclass
```

20.2. Declare the interface.

```
class driver;
  // create virtual interface handle
  virtual intf vif;
  // constructor
  function new(virtual intf vif);
    // get the interface
```

```
    this.vif = vif;
  endfunction
endclass
```

20.3. Add the drive task to drive transaction packets to interface signals. Note that "forever" is used in main() instead of "always" (why?)

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

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

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

    end
  endtask
endclass
```

20.4. 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;
  // create virtual interface handle
  virtual intf vif;
  // create mailbox handle
  mailbox gen2driv;

  // constructor
  function new(virtual intf vif, mailbox gen2driv);
    // get the interface
    this.vif = vif;
    // get 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
```

20.5. 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.en    <= 1;
      vif.op    <= trans.op;
      vif.a     <= trans.a;
      vif.b     <= trans.b;
      @(posedge vif.clk);
      vif.en    <= 0;
      @(posedge vif.clk);
      trans.display("[Driver]");
    end
  endtask
endclass
```

20.6. 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;
  // count the number of transactions
  int no_transactions;

  // ...

  task main;
    forever begin
      transaction trans;
      gen2driv.get(trans);
      @(posedge vif.clk);
      vif.en    <= 1;
      vif.op    <= trans.op;
      vif.a     <= trans.a;
      vif.b     <= trans.b;
      @(posedge vif.clk);
      vif.en    <= 0;
      @(posedge vif.clk);
      trans.display("[Driver]");
      no_transactions++;
    end
  endtask
endclass
```

20.7. In environment, declare and construct the driver.

```
class environment;

  // generator and driver instances
  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;

    // create mailbox(es) for data exchange
    gen2driv = new();

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

  // ...

  task test();
    $display("[Environment]: Start of test() at %0d", $time);
    fork
      gen.main();
      driv.main();
    join
    $display("[Environment]: End of test() at %0d", $time);
  endtask

  // ...

endclass
```

21. Include driver.sv in tbench_top.sv.
22. Compile and run the simulation. Debug the testbench if there are errors. Note that simulation may not stop.
23. Note forever block is used in the driver. Stop simulation when generator stops generates new transactions.

    23.1. Optionally add an event in the generator.

```
class generator;
  // ...

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

  // ...

  // main task to generate (create and randomize) the repeat_count number
of transaction packets
  task main();
    repeat(repeat_count) begin
      trans = new();
      // ...
      trans.display("[Generator]");
      gen2driv.put(trans);
    end
    -> ended; // trigger the end of generation
  endtask
endclass
```

    23.2. Modify the environment accordingly.

```
class environment;
```

```
  // ...

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

  // ...

endclass
```

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

Phase VI: Monitor

25. **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.

    25.1. Create a file named monitor.sv in the verification folder and add an optional include guard.

```
class monitor;

endclass
```

    25.2. Declare interface.

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

    25.3. Declare a mailbox named mon2scb.

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

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

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

```
endclass
```

25.5. 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.en);
      trans.op  = vif.op;
      trans.a   = vif.a;
      trans.b   = vif.b;
      @(posedge vif.clk);
      trans.y   = vif.y;
      @(posedge vif.clk);
    end
  endtask
endclass
```

25.6. Send the transaction to the scoreboard (yet to be created) via the mailbox mon2scb, and also display the transaction.

```
class monitor;
  // ...

  task main;
    forever begin
      transaction trans;
      trans = new();
      @(posedge vif.clk);
      wait(vif.en);
      trans.op  = vif.op;
      trans.a   = vif.a;
      trans.b   = vif.b;
      @(posedge vif.clk);
      trans.y   = vif.y;
      @(posedge vif.clk);
      mon2scb.put(trans);
      trans.display("[Monitor]");
    end
  endtask
endclass
```

25.7. Modify the environment

```
class environment;

  // generator, driver and monitor 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;

    // create the mailbox(es) for data exchange
    gen2driv = new();
    mon2scb  = new();

    // create 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
```

26. Include monitor.sv in tbench_top.sv

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

Phase VII: Scoreboard

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

    28.1. Create a file named scoreboard.sv in the verification folder and add an optional include guard.

```
class scoreboard;

endclass
```

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

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

  // constructor
  function new(mailbox mon2scb);
    // get the mailbox handle from environment
```

```
    this.mon2scb = mon2scb;
  endfunction
endclass
```

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

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

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

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

```
class scoreboard;
  // ...

  task main;
    transaction trans;
    forever begin
      mon2scb.get(trans);
      trans.display("[Scoreboard]");
      if((trans.a+trans.b) == trans.y)
        $display("[Scoreboard]: Pass");
      else
        $error("[Scoreboard]  Error:  \n\tExpected  result:  %0d  Received:
%0d", (trans.a+trans.b), trans.y);
    end
  endtask
endclass
```

28.5. Declare a variable no_transaction to keep count of the received transactions. Display the count after each comparison.

```
class scoreboard;
  // ...

  // count the number of transactions
  int no_transactions;

  task main;
    transaction trans;
    forever begin
      mon2scb.get(trans);
      trans.display("[Scoreboard]");
      // ...
      no_transactions++;
    end
  endtask
endclass
```

29. To add scoreboard to the testbench, edit environment.sv to update **environment class**.

```
class environment;

  // generator, driver, monitor and scoreboard instances
  generator    gen;
  driver       driv;
  monitor      mon;
  scoreboard   scb;

  // ...

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

    // create mailbox(es) for data exchange
    gen2driv = new();
    mon2scb  = new();

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

  // ...

  task test();
    $display("[Environment]: Start of test() at %0d", $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("[Environment]: End of test() at %0d", $time);
  endtask

  // ...

endclass
```

30. Include scoreboard.sv in tbench_top.sv.

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

Go Further

32. 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.

33. Create another test case where op_code is set to 1 from the testbench. And modify the scoreboard to check for subtraction results accordingly.

Deliverable

You are not required to submit this tutorial exercise. You will be asked to construct a verification environment using SystemVerilog OOP in the lab exercises by following a similar process.