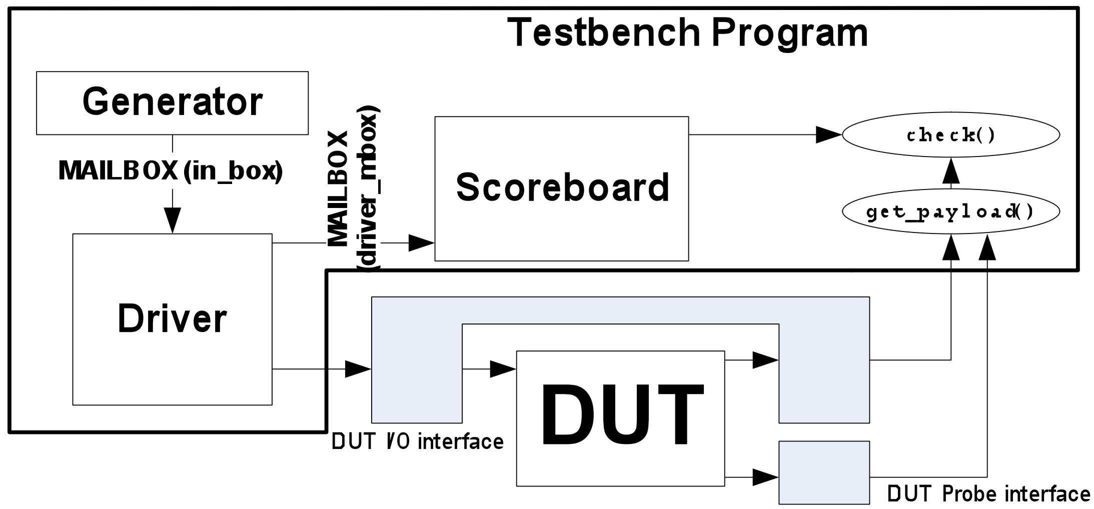
ECE 745: ASIC VERIFICATION
LAB 3:OBJECT ORIENTED TRANSMIT SIDE

Introduction:

The aim of this laboratory exercise is to give you a example a typical testbench with a multi-level object-oriented transmit structure and basic checking features. The difference from Lab2 is that the testbench is explicitly layered and object oriented and the checking is performed using probes into the internals of the DUT and performs checking using a pipelined aproach. This example represents that typical structure for a comprehensive randomized self-testing environment on the stimulus creation and driving side. The testbench is broken up into classes for

- `Packet` : Input packet to the DUT from the transmit side. This continues to be the one used in Lab2 where we have constraints imposed on randomization of the different inputs to the DUT.
- `Generator` : Used to create relevant inputs to the DUT using constraints imposed within the packet class.
- `DriverBase` Provides base definitions and function/task declarations for the driver block
- `Driver`: Extends the driver base class to include specific features of interest to the verifier and includes newer functions/tasks specific for this extension of the base classes. Note the use of the “extends” keyword. It must also be kept in mind that the extension of the base classes allows the functions/tasks here to use the classes declared as “extern” within the base class.
- `ReceiverBase`: Provides base definitions and function/task declarations for receiver block.
- `Receiver`: Extends the receiver base class to include specific features of interest to the verifier and includes newer functions/tasks specific for this extension of the base classes. Again, please note the use of the “extends” keyword. It must also be kept in mind that the extension of the base classes allows the functions/tasks here to use the classes declared as “extern” within the base class.
- `Scoreboard`: The Scoreboard is used to keep tabs on the inputs to the DUT.

The aim is to achieve a checking structure of the form shown below:



Lab3 requirements:
Please note that a basic description of the Execute block that will be used as the DUT and the valid commands for it can be found at the class website. Also, you will continue to use the same `modelsim.ini` as before. In this lab we will continue with the trend of verifying the arithmetic operations using a golden model in the example provided while leaving the rest to the student to work on. A significant change from the previous lab is use of classes here to ensure encapsulation of testbench behavior into reusable units. This, of course, leads to the need for the instantiation of these classes as objects. These details will be covered in the coming sections.

Let us begin the process of understanding the coding structure by noting that the `Packet.sv` has been modified to include an enable is created at the driver side to sensitize the DUT. Other than this, there are no new constructs in the Packet class.

Similarly, the `OutputPacket.sv` is created at the Receiver side to receive the signals from the DUT. You will notice the OutputPacket.sv contains the signals defined as output from the DUT as well as the internal signals we have been probing.

The Generator is going to be used to introduce an extremely important construct called the **mailbox.** This is much like a queue but allow for much easier communication between threads which use asynchronous stalls to wait till data is ready for analysis/processing. We will also look at the typical structure of a class and its usage. It is very important for each class to have a constructor i.e. the definition for `new()` wherein things like memory allocation is performed if needed, unique identifiers are provided and such. The code below declares a bare-bones generator with just the declaration of the methods (`gen()`, `start()`, and new()) and data structures (here, `name`, `in_box`, `pkt2send`, `num_packets`, `packet_number`) within it.

```
##########################################################################
```

```
class Generator;
   string  name;
   Packet  pkt2send;

   typedef mailbox #(Packet) in_box_type;
   in_box_type in_box;

   int      packet_number;
   int      number_packets;
   extern function new(string name = "Generator", int number_packets);
   extern virtual task gen();
   extern virtual task start();
endclass
```

*Declaration of Mailbox that handles the Packet datatype. The typedef is a **MUST**. The two lines are needed.*

```
######################################################################
```

The methods are expanded outside the declaration of the class as shown below for the Generator. The `new()` function is used to map the parameters provided on making an instance of the Generator to the data fields inside the class using the "`this.`" construct. An example of the same is shown below. Note that the Generator will be instantiated as `generator = new("Generator", number_packets);` which will cause "`Generator`" and `number_packets` be mapped to the `name` and `number_packets` variables within the class (the variable names need not be the same)

```
######################################################################
function Generator::new(string name = "Generator", int number_packets);
   this.name = name;
   this.pkt2send = new();
   this.in_box = new;
   this.packet_number = 0;
   this.number_packets = number_packets;
endfunction
```

make instance of mailbox

```
task Generator::gen();
   pkt2send.name = $psprintf("Packet[%0d]", packet_number++);
   if (!pkt2send.randomize())
   begin
      $display("\n%m\n[ERROR]%0d gen(): Randomization Failed!", $time);
      $finish;
   end
      pkt2send.enable = $urandom_range(0,1);
endtask

task Generator::start();
$display ($time, "ns:  [GENERATOR] Generator Started");
   fork
   for (int i=0; i<number_packets || number_packets <= 0; i++)
   begin
      gen();
      begin
         Packet pkt = new pkt2send;
         in_box.put(pkt);
      end
   end
   join_none
endtask
######################################################################
```

Call `gen()` *to create a certain a packet of stimulus and queue it up in the the mailbox using the* **`put()`** *method*

The `start()` method is used to kick start the functioning of the class under user control. The aim is to be able to spawn off the function associated with a class using forking and let it run on its own in parallel. This can be achieved by a call to the above start task through its instance as `generator.start()`. The above will be shown in greater detail later in the document. Thus, at this point, we have a mailbox which acts as a provider of packets for the DUT. This needs to be sent asserted at the input of the DUT for which we have the driver. The driver class is broken down into a base class called `DriverBase` which contains all the basic constructs that would be useful for any extension of this class. The base-class is takes the `send_payload()` and `send()` tasks from Lab2 and makes them methods within the class. Also, to enable the an instance of the class to assert inputs to the DUT, the DUT interface is going to have to be connected to a local interface, called `Execute` here, which is instantiated as a virtual interface as shown in the example below:

```
##########################################################################
class DriverBase;
   virtual    Execute_io.TB Execute;
   string     name;
   Packet     pkt2send;
```

Virtual interface to enable driving of DUT inputs from class instance

```
   reg       [6:0]                    payload_control_in;
   reg       [`REGISTER_WIDTH-1:0]    payload_src1, payload_src2;
   reg       [`REGISTER_WIDTH-1:0]    payload_imm, payload_mem_data;
   reg                                payload_enable;

   extern function new(string name = "DriverBase", virtual
            Execute_io.TB Execute);
   extern virtual task send();
   extern virtual task send_payload();

endclass
```

Connectivity to the interface at Test program will be done during instantiation using `new()`

```
function DriverBase::new(string name = "DriverBase", virtual
            Execute_io.TB Execute);
   this.name    = name;
   this.Execute = Execute;
endfunction
```

Connection of the incoming interface to the local virtual interface

```
task DriverBase::send();
   send_payload();
endtask

task DriverBase::send_payload();
   $display($time, "ns:  [DRIVER] Sending Payload Begin");

   Execute.cb.src1              <=    payload_src1;
   Execute.cb.src2              <=    payload_src2;
   Execute.cb.imm               <=    payload_imm;
   Execute.cb.mem_data_read_in  <=    payload_mem_data;
   Execute.cb.control_in         <=  payload_control_in;
   Execute.cb.enable_ex         <=  payload_enable;

endtask
##########################################################################
```

Sending stimulus packet into the DUT I/O's

```
#######################################################################
`include "DriverBase.sv"
class Driver extends DriverBase;
```

Extension of Driver Base class

```
    typedef mailbox #(Packet) in_box_type;
    in_box_type in_box = new;
```

Declaration for mailbox from Generator to Driver

```
    typedef mailbox #(Packet) out_box_type;
    out_box_type out_box = new;
```

Declaration for Mailbox from Driver to Scoreboard

```
    extern function new(string name = "Driver", in_box_type in_box,
           out_box_type out_box, virtual Execute_io.TB Execute);

    extern virtual task start();
endclass
```

This new() overrides the new() from the base-class

```
function Driver::new(string name= "Driver", in_box_type in_box,
         out_box_type out_box, virtual Execute_io.TB Execute);
   super.new(name, Execute);
   this.in_box = in_box;
   this.out_box = out_box;
endfunction
```

Function new() with incoming mailbox and outgoing mailbox to be connected during instantiation. Also, we see the assignment of incoming mailboxes to local instances

```
task Driver::start();
   reg      [6:0] control_in_temp;
   int get_flag = 10;
   int packets_sent = 0;
   $display ($time, "ns:  [DRIVER] Driver Started");
   fork
      forever
      begin
         in_box.get(pkt2send);
```

Get packet from mailbox coming in from Generator

```
         packets_sent++;
         control_in_temp = {pkt2send.operation_gen,
                    pkt2send.immp_regn_op_gen, pkt2send.opselect_gen};
         $display ($time, "[DRIVER] Sending in new packet BEGIN");
         this.payload_control_in = control_in_temp;
         this.payload_src1 = pkt2send.src1;
         this.payload_src2 = pkt2send.src2;
         this.payload_imm =   pkt2send.imm;
         this.payload_mem_data = pkt2send.mem_data;
         this.payload_enable = pkt2send.enable;

         send();
```

Construct packet that will be sent into the DUT and call send() which uses the virtual interface

```
         out_box.put(pkt2send);
         if(in_box.num() == 0)
         begin
            break;
         end

         @(Execute.cb);
      end
   join_none
endtask
#######################################################################
```

Copy packet sent to DUT to the mailbox from driver to Scoreboard

An interesting language usage is shown in `super.new(name, Execute);` where the `Driver` class instance, when instantiated, will call the new() of the `DriverBase` class that it extends.

At this point we see the necessary constructs to create stimulus and send it to the DUT using an object oriented coding scheme. We will now look at the Receiver. The main function of the Receiver class is to receive the signals from the DUT. This needs to be sent asserted at the input of the DUT for which we have the driver. The receiver class is broken down into a base class called `ReceiverBase` which contains all the basic constructs that would be useful for any extension of this class. The base-class takes the `get_payload()` and `recv()` tasks from Lab2 and makes them methods within the class. Also, to enable the an instance of the class to obtain outputs from the DUT, the DUT interface is going to have to be connected to a local interface, called `Execute` here, which is instantiated as a virtual interface as shown in the example below: We have also connected the probe interface to the virtual interface, called Prober in this class. You will have noticed a lot of similarities between the structures of the Receiver and Driver.

We will now look at a rudimentary Scoreboard that will keep tabs on the data transmitted to the DUT and determine correctness of the result from the DUT.

```
########################################################################
class Scoreboard;
   string    name;
   typedef mailbox #(Packet) out_box_type;
   out_box_type driver_mbox;

   Packet pkt_sent = new();    // Packet object from Driver
   OutputPacket   pkt2cmp = new();  // Packet object from Receiver

   typedef mailbox #(Packet) out_box_type;
   out_box_type driver_mbox;  // mailbox for Packet objects from
Drivers

   typedef mailbox #(OutputPacket) rx_box_type;
   rx_box_type    receiver_mbox; // mailbox for Packet objects from
Receiver

      // Declare the signals to be compared over here.
       reg   [`REGISTER_WIDTH-1:0]   aluout_chk = 0;
       reg                    mem_en_chk;
       reg   [`REGISTER_WIDTH-1:0]   memout_chk;

       reg   [`REGISTER_WIDTH-1:0]   aluin1_chk =0 , aluin2_chk=0;
       reg   [2:0]            opselect_chk=0;
       reg   [2:0]            operation_chk=0;
       reg   [4:0]                  shift_number_chk=0;
       reg                    enable_shift_chk=0, enable_arith_chk=0;
       reg   [16:0]                 aluout_half_chk;

       extern function new(string name = "Scoreboard", out_box_type
       driver_mbox, rx_box_type receiver_mbox);
       extern virtual task start();
       extern virtual task check();
```

Declaration of mailbox that will come in from Driver

We have defined the check() and its sub-functions check_artih() and check_preproc() as virtual tasks over here.

```
        extern virtual task check_arith();
        extern virtual task check_preproc();

endclass

function Scoreboard::new(string name, out_box_type driver_mbox,
rx_box_type receiver_mbox);
   this.name = name;
   if (driver_mbox == null)
      driver_mbox = new();
   if (receiver.mbox == null)
      receiver_mbox = new();
   this.driver_mbox = driver_mbox;
   this.receiver_mbox = receiver_mbox;
endfunction
```

The driver mailbox must be instantiated ONLY if it has not already been done before in the driver class. Remember that the mailbox data structure should be allocated only once

```
task Scoreboard::start();
        $display ($time, "[SCOREBOARD] Scoreboard Started");
        $display ($time, "[SCOREBOARD] Receiver Mailbox contents = %d",
receiver_mbox.num());
        fork
              forever
              begin
                    if(receiver_mbox.try_get(pkt2cmp)) begin
                          driver_mbox.get(pkt_sent);
                          check();
                    end
                    else
                    begin
                          #1;
                    end
              end
        join_none
endtask
```

The start() task runs forever and is used to obtain the packets from both the Driver and the Receiver using the respective mailboxes. Notice that only if the receiver packet is obtained, we obtain the corresponding packet from the driver.

```
task check();
   $display($time, "ns: [CHECKER] Checker Start\n\n");
         // Grab packet sent from scoreboard
   sb.driver_mbox.get(pkt_sent);
   $display($time, "ns:   [CHECKER] Pkt Contents: src1 = %h, src2 = %h,
         imm = %h, ", pkt_sent.src1, pkt_sent.src2, pkt_sent.imm);
   $display($time, "ns:   [CHECKER] Pkt Contents: opselect = %b,
               immp_regn= %b, operation = %b, ", pkt_sent.opselect_gen,
               pkt_sent.immp_regn_op_gen, pkt_sent.operation_gen);
   check_arith();
   check_preproc();
endtask
##########################################################################
```

After we have obtained the packets from the Driver and Receiver, we will then call the check() task. This check task will then generate the golden model using the Driver packet contents and compare it with the contents of the Receiver Output packet.

In the above, we see that the checking is performed in reverse order i.e. ALU and then pre-processor given that we are dealing with a pipeline and any snap-shot of the internals and externals of the DUT can be best used when analyzed from the last to first pipeline stage.

You will also observe we have used some immediate assertions while comparing the results of the model and DUT. Assertions are a powerful tool in System Verilog and will be covered later in the course. The interested reader is encouraged to look up the use of immediate assertions, what happens if an assertion fails, and the severity of the failure. You can play around in the code with the different types of severity system tasks that can be included in the fail statement to specify a severity level: $fatal, $error (the default severity) and $warning.

Again, please pay close attention to the way mailboxes are instantiated within the Scoreboard. An example is shown below where we are creating a mailbox that is parameterized to be of the type Packet and is then instantiated.

```
typedef mailbox #(Packet) out_box_type;
out_box_type driver_mbox;
```

##

To enable checking to be performed correctly, we need to be able to view the internal details of the DUT after the first pipeline stage. This is done using the declaration of the following interface in `Execute.if.sv`

##

```
interface DUT_probe_if(
      input bit clock,
      input logic [`REGISTER_WIDTH-1:0]   aluin1,
      input logic [`REGISTER_WIDTH-1:0]   aluin2,
      input logic [2:0]                   opselect,
      input logic [2:0]                   operation,
      input logic [4:0]                   shift_number,
      input logic                         enable_shift,
      input logic                         enable_arith
      );
```

Note that there is no clocking block used to probe the internal signals. All the internal signals are just obtained by tapping into the DUT using the Probe interface.

###

The above interface is connected to the DUT instance as shown below in the `Execute.test_top.sv`. By doing this we gain access to the signals which run between the Preprocessor and the ALU in the DUT.

##

```
DUT_probe_if DUT_probe(
      .aluin1(dut.aluin1),
      .aluin2(dut.aluin2),
      .opselect(dut.opselect),
      .operation(dut.operation),
      .shift_number(dut.shift_number),
      .enable_shift(dut.enable_shift),
      .enable_arith(dut.enable_arith)
  );
```

##

To enable the testbench to read the contents of the `DUT_probe` interface and hence the internals of the DUT we add it to the list of interfaces that go to the testbench program as shown below for both top level instance of the program (in `Execute.test_top.sv`) and the declaration of the test program (`Execute.tb.sv`):

```
Execute_test test(top_io, DUT_probe);

program Execute_test(Execute_io.TB Execute, DUT_probe_if Prober);
```

Thus, by doing this we have all the requisite blocks to perform stimulus generation, driving, receiving and checking using the DUT signals. The code below provides the means of making instances of each of these classes and the connections using mailboxes that exist between the Generator-Driver and Driver-Scoreboard. In `Execute.tb.sv` we see the creation of the necessary class objects as:

```
    Generator   generator;  // generator object
    Driver      drvr;       // driver objects
    Scoreboard  sb;         // scoreboard object
    Receiver    rcvr();     // Receiver object
```

In addition to the above declaration, it is necessary to allocate memory for each object (instance) and hence we would need to follow the procedure detailed below. **An extremely important point to remember is that the order of allocation is very important.** In the example below we instantiate the scoreboard and generator first and hence we are going to have to use `generator.in_box, sb.driver_mbox` when we instantiate the driver given that the mailboxes would already have been allocated at that point. Note also the passage of the `Execute` interface to the driver through the test program which will be connected to the virtual interface within it. Lastly, we instantiate the receiver. In this case, we have passed both the `Execute` interface and the `Prober` interface to the receiver which will be connected to the virtual interfaces connected within it. Another interesting piece of code minimization is the lack of any input parameters to the constructor of the scoreboard. In this case the defaults that exist in the class declaration will be used.

```
######################################################################
initial begin
   number_packets = 21;

   generator = new("Generator", number_packets);
   sb = new(); // NOTE THAT THERE ARE DEFAULT VALUES FOR new()
               // FUNCTION CALL WITHIN THE SCOREBOARD
   drvr = new("drvr[0]", generator.in_box, sb.driver_mbox, Execute);
   rcvr = new("rcvr[0]", sb.receiver_mbox, Execute, Prober);
```

Instantiate the different classes in the correct order

```
  reset();

   generator.start();
   drvr.start();
   sb.start();
   rcvr.start();
```

When we call each start() task in the above i.e. `generator.start(), drvr.start(), etc.` we are forking different processes which will in-turn create data and send this data to the DUT.

What you need to take away from this lab is the modularity of the testbench and the means of kicking off the tasks for each type of class.

To compile the files do the following after setting all the environment variables (`setenv, vlib etc`)

```
> vlog *.vp *.v
> // ALL OF THE BELOW SHOULD BE ON ONE LINE AND IN THE SAME ORDER
> vlog –sv vlog -mfcu -sv data_defs.v Packet.sv OutputPacket.sv
Driver.sv Receiver.sv Scoreboard.sv Generator.sv Execute.tb.sv
Execute.if.sv Execute.test_top.sv
To simulate, do the following
> vsim -novopt Execute_test_top

You can suppress the warnings that come while compiling the
Testbench files (.sv files) you can use the –suppress option, i.e.
```

```
> vlog -mfcu -sv -suppress 2217 data_defs.v Packet.sv
OutputPacket.sv Driver.sv Receiver.sv Scoreboard.sv
Generator.sv Execute.tb.sv Execute.if.sv
Execute.test_top.sv
```

Lab3 Submission Requirements:

As stated above, the checker has been used, at present, to perform only arithmetic operation checking. You will have to

1. Modify the `check()` task to perform correctness checks for the rest of the operations (`Memory Read, Memory Write, Shift`)
2. Run multiple inputs into the DUT (mostly by using the correct constraints within the Packet class) and determine correctness of the various DUT result for inputs in 1. Note that you might need to vary the run time as well to make sure that the requisite input types are met.

If there is an error in the result from the DUT and the expected value use a `$display` statements of the form shown below to display your check:

```
$display($time, "[ERROR] Expected ALU Value = %h,  Observed ALU Value
= %h”, aluout_cmp, aluout_q_val);
```

A bug that is observed in the design should be documented in the following format:

a. Design Input for Bug to Appear.
b. Expected Behavior referring to the erroneous signal. For example, `aluout` should be ________ for this instruction because .....
c. Observed Behavior. For Example. `aluout` was found to be__________

d. Summary of your thoughts on the error. For example, "we conclude that there is an error in the logical shift left. We find that it shifts only by `shift_number -1` instead of `shift_number`"

To get credit for your work, make sure that these results are displayed by running your program. The same file names as provided need to be used and submitted.

Follow the following steps for submissions (Solaris/Linux only please)

- `mkdir Lab3` (creates the directory `Lab3`)
- copy all the SystemVerilog files into the `Lab3` directory
- Zip the file using the command `> zip Lab3.zip Lab3/*`
- Submit the zip using the submit utility on the course webpage.