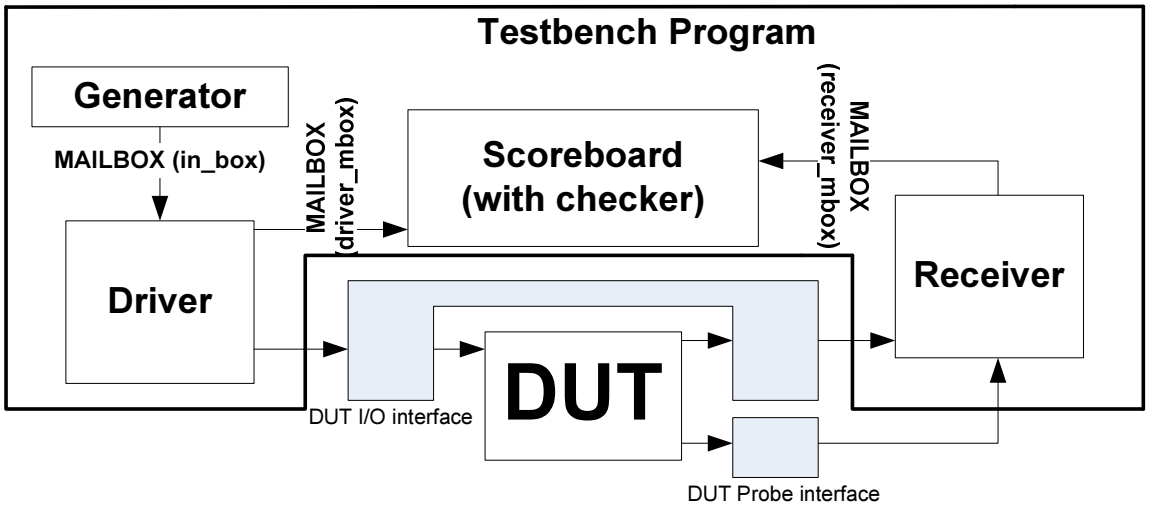
ECE 745: ASIC VERIFICATION
LAB 4: LAYERED & OBJECT ORIENTED TESTBENCH

Introduction:
This laboratory exercise extends Lab3 to a fully layered testing methodology with the aim of having object-oriented constructs on both the transmit and receive side. In conjunction with the materials in Lab3, this laboratory exercise provides a comprehensive and re-usable methodology for test and debug. The additional classes for this Lab exercise are:

- `OutputPacket` : Packet generated at the Receiver by sampling the outputs of the DUT
- `ReceiverBase`: Provides base definitions and function/task declarations for the 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. 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. In this case we also implement the checker with the scoreboard to determine the correctness of Arithmetic ALU operations.

The final testing structure that we are aiming for is shown below:



This model extends Lab3 by adding a receiver structure and the relevant mailbox from the receiver to the Scoreboard. This new mailbox will provide a snapshot of the output and the internal details of the DUT to the Scoreboard to enable correctness checks.

Lab4 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. Keep an eye out for modifications to the spec.

http://www.ece.ncsu.edu/asic/asic_verification/shared/System_Spec/System_Spec.pdf

Please download all the files below into a single directory as you did in Lab3:
The new testbench files are at:
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Execute.tb.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Execute.if.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Execute.test_top.sv

http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/OutputPacket.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/ReceiverBase.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Receiver.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Scoreboard.sv

and, the new DUT files are at:
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Ex_Preproc.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/ALU.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Arith_ALU.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/Shift_ALU.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab4/data_defs.v

The top level stitching remains unchanged. Also, you will continue to use the same `modelsim.ini,Generator.sv, Packet.sv, DriverBase.sv,` and `Driver.sv` 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. As stated earlier, in this case we are going to ensure object-oriented coding on the receive side as well. To enable this, we encapsulate all the necessary signals for observation in a packet class called the `OutputPacket`. This object would, at a given sample point, contain a snapshot of the internal state of the logic. The details of this class are shown below:

```
////////////////////////////////////////////////////////////////////////
class OutputPacket;
   string                              name;

   reg   [`REGISTER_WIDTH-1:0]         aluout;
   reg                                 mem_write_en;
   reg   [`REGISTER_WIDTH-1:0]         mem_data_write_out;

   reg   [`REGISTER_WIDTH-1:0]         aluin1;
   reg   [`REGISTER_WIDTH-1:0]         aluin2;
   reg   [2:0]                         opselect;
   reg   [2:0]                         operation;
   reg   [4:0]                         shift_number;
   reg                                 enable_shift;
   reg                                 enable_arith;
   reg                                 enable;

      extern function new(string name = "OutputPacket");
endclass
```

Snapshot of ALU output

Snapshot of Execute Pre-processor output

```
function OutputPacket::new(string name = "OutputPacket");
      this.name = name;
endfunction
//////////////////////////////////////////////////////////////////////
```

The receiver base class is very similar to the driver base class in that it encapsulates all the necessary functionality in it to perform signal acquisition. It, of course, does the exact opposite of the driver and attempts to determine the values at the outputs of the blocks of interest. To this end, the receiver is provided with a virtual interface (the usage of which has already been shown in Lab3) to connect to both the Execute interface and the Probe interface. This is done using the constructor for the ReceiverBase which is declared as

```
extern function new(string name = "ReceiverBase", virtual
   Execute_io.TB Execute, virtual DUT_probe_if Prober);
```

and extended later in the Receiver class to the following to include a mailbox

```
extern function new(string name = "Receiver", rx_box_type
   rx_out_box, virtual Execute_io.TB Execute, virtual DUT_probe_if
   Prober);
```

The incoming interfaces are mapped to local virtual interfaces declared in the ReceiverBase class

```
virtual Execute_io.TB Execute;
virtual DUT_probe_if Prober;
```

There is also the need for a new mailbox for sending packets from the Receiver to the Scoreboard. The means of sharing the two virtual interfaces and the declaration of new mailbox is shown below where the ReceiverBase class is extended.

```
`include "ReceiverBase.sv"
class Receiver extends ReceiverBase;
   typedef mailbox #(OutputPacket) rx_box_type;
   rx_box_type    rx_out_box;
```

New mailbox for receiver-scoreboard communication

```
   extern function new(string name = "Receiver", rx_box_type
         rx_out_box, virtual Execute_io.TB Execute, virtual
         DUT_probe_if Prober);
```

Extended constructor for Receiver.

```
   extern virtual task start();
endclass


function Receiver::new(string name = "Receiver", rx_box_type
   rx_out_box, virtual Execute_io.TB Execute, virtual DUT_probe_if
   Prober);
   super.new(name, Execute, Prober);
   this.rx_out_box = rx_out_box;
endfunction
```

Constructor calling the constructor of base class with interfaces and a new mailbox connection

```
task Receiver::start();
   @ (Execute.cb);
   fork
      forever
      begin
         @ (Execute.cb);
         recv();
         rx_out_box.put(pkt_cmp);
      end
   join_none
endtask
```

Begin receiving 1 clock cycle after the transmission of data into the DUT

Fork a process where a new snapshot of the output is created every clock cycle using the recv() task

The `recv()` task calls the `get_payload()` task and creates the `pkt_cmp` packet which is sent to the Scoreboard using the `rx_out_box` mailbox.

There are some improvements in the Scoreboard as well where we perform checking as a part of its functionality. To this end, we use the mailboxes from both the Driver and Receiver for a) creation of the expected and b) checking the contents of snapshot of the DUT. This checking is done using the combination of `check_arith()` and `check_preproc()` tasks which do not differ from the previous labs but for the fact that they are now within the Scoreboard. The core operation of the Scoreboard is captured in the code below:

```
fork
   forever
   begin
      while(receiver_mbox.num() == 0)
      begin
         $display ($time, "ns:  [SCOREBOARD] Waiting for Data in
              Receiver Outbox to be populated");
         #`CLK_PERIOD;
      end

      while (receiver_mbox.num()) begin
         $display ($time, "ns:  [SCOREBOARD] Grabbing Data From both
             Driver and Receiver");
         receiver_mbox.get(pkt_cmp);
         driver_mbox.get(pkt_sent);
         check();
      end
   end
join_none
```

Stay in loop until data is seen in receiver mailbox

Read data from both the receiver and driver mailboxes and perform checking using `check()` *task which in turn calls* `check_arith()` *and* `check_preproc()`

In the above, the `.num()` method used with mailboxes allows the user to query the number of objects presently queued up in it. It provides an easy way to determine if the program should stall or not for data arrival.

The only major addition that needs to be noted in `Execute.tb.sv` is in the additional constructs for the receiver class which will include the object declaration and memory allocation as:

```
Receiver   rcvr;

rcvr = new("rcvr[0]", sb.receiver_mbox, Execute, Prober);
```

Again, the order of memory allocation is of importance. In the above, we are going to call the receiver allocation after the scoreboard and hence use `sb.receiver_mbox` which would already have been allocated. Note also the explicit connection of the incoming Probe interface to the receiver instance. This is used to send data to the Scoreboard.

Thus, with this addition, we now have a fully constructed test bench which is layered and object-oriented on both transmit and receive side. Along with the requirements for constraining the data stimulus in the Packet class, this provides a working example of a re-usable layered and constrained random self checking environment for testing.

Important points of note:
To compile the files do the following after setting all the environment variables (`setenv, vlib etc`)

```
> vlog *.vp
> vlog *.v
// ALL OF THE BELOW SHOULD BE ON ONE LINE AND IN THE SAME ORDER
> vlog -mfcu -sv data_defs.v Packet.sv Driver.sv OutputPacket.sv
     Receiver.sv  Scoreboard.sv Generator.sv Execute.tb.sv
     Execute.test_top.sv Execute.if.sv
```

To Simulate, please do:

```
> vsim -novopt Execute_test_top
```

Lab4 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`"

Please note that the variable names might differ from `aluout_cmp` for your cases based on the way you have coded your checker function.

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 Lab4`(creates the directory `Lab4`)
- copy all the SystemVerilog files into the `Lab4` directory
- Zip the file using the command `> zip Lab4.zip Lab4/*`
- Submit the zip using the submit utility on the course webpage.