ECE 745: ASIC VERIFICATION
LAB 2

Introduction:
The aim of this laboratory exercise is to give you a flavor of a typical testbench with a object oriented transmit and receive structure with some basic checking features. To this end, we create a class of type `"Packet"` which would represent all the necessary inputs into the DUT for testing to be achieved. We would then iterate a pre-defined number of times while creating a payload of packets where the size of the payload is arbitrarily decided using the `$urandom()` directive. Each packet is then sent into the DUT while kicking off receive and check tasks to do the necessary logging of the outputs of the DUT and checking for correctness respectively.

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. (This remains unchanged from the previous laboratory exercise)
http://www.ece.ncsu.edu/asic/asic_verification/shared/System_Spec/System_Spec.pdf

The system consists of two sub-modules: the preprocessor which creates all the control signals and the ALU. In this laboratory exercise we are not going to concern ourselves with the preprocessor block. But we must cater for the two cycle latency in the creation of the final output for an input sent in.

Lab2 Requirements:
Please download all the files below into a single directory as you did in Lab1:
new testbench files at:
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/Execute.tb.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/Execute.if.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/Execute.test_top.sv
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/Packet.sv

and, the new DUT files at:
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/Ex_Preproc.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/ALU.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/Arith_ALU.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/Shift_ALU.vp
http://www.ece.ncsu.edu/asic/asic_verification/shared/Lab2/data_defs.v

You can use the same modelsim.ini that you did in Lab1. Make sure you copy it to the same directory that you are using for the files above.

The important files that need to be considered for this Laboratory exercise are `Execute.tb.sv` and `Packet.sv`. The `Execute.test_top.sv` and `Execute.if.sv` remain unchanged from Lab1. To begin with, we look at the contents of `Packet.sv` which consists of a class that contains the fields that would be sent into the DUT as an input. These would consist of appropriate constraints on these fields to keep them within

valid ranges and constructs that allow for randomization to be applied to these fields and hence the entire class instance. The `Packet.sv` file is shown below:

```
class Packet;

   rand   reg [`REGISTER_WIDTH-1:0]       src1;
   rand   reg [`REGISTER_WIDTH-1:0]       src2;
   rand   reg [`REGISTER_WIDTH-1:0]       imm;
   rand   reg [`REGISTER_WIDTH-1:0]       mem_data;
   rand   reg                             immp_regn_op_gen;
   rand   reg [2:0]                       operation_gen;
   rand   reg [2:0]                       opselect_gen;
```

ensuring relevant fields are randomizable (could also be randc)

```
   string   name;

   constraint Limit {
      src1 inside {[0:65534]};
      src2 inside {[0:65534]};
      imm inside  {[0:65534]};
      mem_data inside   {[0:65534]};

      opselect_gen inside {[1:1]}; //only arithmetic inputs

      if ((opselect_gen == `ARITH_LOGIC)){
         operation_gen inside {[0:7]};
      }
      else if ((opselect_gen == `SHIFT_REG)) {
         immp_regn_op_gen inside {0};
         operation_gen inside {[0:3]};
      }
      else if ((opselect_gen == `MEM_READ)) {
         immp_regn_op_gen inside {1};
         operation_gen inside {[0:4]};
      }
      else if ((opselect_gen == `MEM_WRITE)) {
         immp_regn_op_gen inside {1};
         operation_gen inside {[0:7]};// make sure it does not matter
      }
   }

   extern function new(string name = "Packet");
endclass

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

Constraint (called Limit) on class Packet

opselect takes values for just arithmetic operations in this case. For all valid cases do: inside {[0:1], [4:5]};

conditional selection of other fields based on opselect

Also note the use of "`rand`" which allows for the randomization of the necessary fields (`mem_data, src1, src2` etc) within the packet. Also, note the constraints section which limits each of the randomizable fields to an upper and lower limit. This allows for the creation of only valid inputs into the DUT. You can vary the requirements for your test by specifically focusing on the

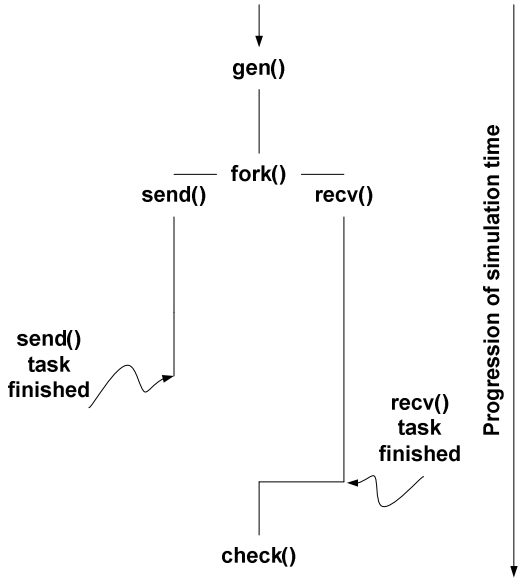
```
opselect_gen inside {[0:1], [4:5]};
```

statement which will automatically restrict the rest of the fields to valid inputs. Ergo, if you wanted to particularly focus on arithmetic inputs, you would change this line to

```
opselect_gen inside {[1:1]}; // weird yes, but you need
                             // upper and lower limits
```

The above is because the `opselect` value for Arithmetic operations is `3'b001`. You will also note that this exercise has a `check()`(within `Execute.tb.sv`) task called at the end that checks for the correctness of, at present, only the arithmetic operations that the Execute unit performs.

The method of testing followed in this laboratory exercise is: a) Create a payload of randomized packets (shown above) with appropriate constraints `gen()` b) Send all the packets created into the DUT one cycle at a time `send()` while also storing it in a queue to be used later for checking c) in parallel with b) kick off a task that grabs the result from the DUT to enable checking `recv()` d) after all the data has been sent into the DUT, run a checker that performs a check for correctness on the result from the DUT taking into consideration the data sent in `check()`. This is pictorially represented below:



The above is accomplished using the following code segment:

```
initial begin
   run_n_times = 21;
   reset();
   repeat(run_n_times) begin
      $display($time, "ns: Sending Another Packet");
      gen();
      fork
         send();
         recv();
      join
      check();
   end
   repeat(5) @(Execute.cb);
end
```

The above code segment kicks off the `gen()` task which creates a payload of a certain number of packets decided by `number_packets = $urandom_range(3,12);` command. This is shown below:

```
task gen();

   number_packets = $urandom_range(3,12);

   GenPackets = new[number_packets];
   for (i=0; i<number_packets; i++) begin
      GenPackets[i] = new();
   end
   for (i=0; i<number_packets; i++) begin
      pkt2send.name = $psprintf("Packet[%0d]", i);
      if (!pkt2send.randomize()) begin
         $display("\n%m\n[ERROR]gen():Randomization Failed!", $time);
         $finish;
      end

      GenPackets[i].src1 = pkt2send.src1;
      GenPackets[i].src2 = pkt2send.src2;
      GenPackets[i].imm = pkt2send.imm;
      GenPackets[i].mem_data = pkt2send.mem_data;
      GenPackets[i].immp_regn_op_gen = pkt2send.immp_regn_op_gen;
      GenPackets[i].operation_gen = pkt2send.operation_gen;
      GenPackets[i].opselect_gen = pkt2send.opselect_gen;
   end
endtask
```

allocate memory for dynamic array of Packet pointers and call constructor for `number_packets` instances of Packet

Create randomized packets that would be sent into DUT as inputs. Randomization done using randomize() method call

move the randomized packet into the dynamic array

The `pkt2send.randomize()` command randomizes the fields of the `pkt2send` packet which have the "`rand`" qualifier. If the command can not be executed it returns a 0 which is used to perform an `if(pkt2send.randomize()==0)` check for error. This works in conjunction with the declaration of the Packet class.

Then, the `send()` and `recv()` tasks are kicked off in parallel by virtue of the `fork join` commands. The two tasks run in parallel and only when both are finished will `check()` be run. The `send()` task calls the `send_payload()` task (it could call others if there was a need for it based on the protocol in use) which has the following structure

```
task send_payload();

   Execute.cb.enable_ex       <= 1'b1;
   packets_sent = 0;
   i = 0;

   while(i < GenPackets.size()) begin

      pkt2send = GenPackets[i];

      Execute.cb.src1   <=    pkt2send.src1;
      Execute.cb.src2   <=    pkt2send.src2;
      Execute.cb.imm    <=    pkt2send.imm;
      Execute.cb.mem_data_read_in   <=    pkt2send.mem_data;
```

grab next packet from dynamic array

Create input signals to the DUT (through the clock block interface) using the packet popped out of array

```
        Execute.cb.control_in   <=  {pkt2send.operation_gen,
                    pkt2send.immp_regn_op_gen, pkt2send.opselect_gen};


        packets_sent++;
        Inputs.push_back(pkt2send);
        Enables.push_back(1'b1);
        i++;
        @(Execute.cb);
    end

        GenPackets.delete();
endtask
```

move packet sent in into queue to keep tabs on all the inputs sent into the DUT

Clear allocated memory for dynamic array to enable creation of next set of inputs in a recursive manner

The `Inputs.push_back(pkt2send);` command in the `send_payload()` task is used to store all the commands sent into the DUT. This is a very basic scoreboard (this will be talked about in greater detail as we progress towards more complicated labs). What we achieve by doing this is checking for things like ensuring completion of all inputs and in the proper order, checking each of the commands send in for correctness of execution, to ensure the correct number of clocks taken for execution and such. This would be of particular importance in your projects.

The `recv()` task has the function of grabbing the outputs from the DUT. Again, note that we are not going to be dealing with the preprocessor block in this laboratory exercise. We will be dealing with the acquisition of the internal signals of the DUT in the next laboratory exercise. This task has the following structure:

```
task recv();
    int i;
    @ (Execute.cb);
    //delay for synchronization with the outputs from DUT
    repeat(number_packets+1) begin
        @ (Execute.cb);
        get_payload();
    end
endtask
```

the first `@(Execute.cb)` is used to ensure that the fact that this is a DUT with 2 pipelined stages. At the core of this task is the call to the `get_payload()` task which performs the acquisition of the DUT outputs and pushes it to the relevant queues for checking

```
task get_payload();

    aluout2cmp = Execute.cb.aluout;
    mem_en2cmp = Execute.cb.mem_write_en;
    memout2cmp = Execute.cb.mem_data_write_out;

    aluout_q.push_back(aluout2cmp);
    memout_q.push_back(memout2cmp);
    mem_en_q.push_back(mem_en2cmp);

endtask
```

Acquisition of output values from the DUT

Sent values acquired to the relevant queues (one for each output signal in this case)

Once all the values from the DUT have been acquired (remember to cater for the 2 clock cycle latency between the input and the output), we can run a the `check()` task that attempts to compare all the outputs acquired with the inputs sent in with the. This is accomplished using the synchronization of reads (using `pop_front()`) from the `Inputs` queue and the reads from the `aluout` queue. The reads from the `Inputs` queue is followed by the creation of the relevant control signals and inputs (`aluin1, aluin2, arith_op`) to mimic the functionality of the preprocessor unit. The control signals are then used in the `check_arith()` task to create the expected values and compare them with the values from the DUT.

Important points of note:

- Please note that the timing of the Memory write operations differ from the arithmetic, shift and memory read operations. Cater to these in your receive and checking functions.
- Also, please observe only the relevant outputs in these labs. We are not going to observe outputs that are not supposed to be affected by a given operation.
- To compile the files do the following after setting all the environment variables (`setenv, vlib etc`)

```
> vlog *.vp
> vlog –sv Execute.if.sv  Execute.test_top.sv  Execute.tb.sv
> vsim -novopt Execute_test_top
```

Lab2 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 `opselect` (`Memory Read, Memory Write, Shift`) values and their variants in terms of the operations.
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 (`Execute.tb.sv`, `Execute.test_top.sv`, `Execute.if.sv`, `Packet.sv`) need to be used.

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

- `mkdir Lab2` (creates the directory `Lab2`)
- copy all the SystemVerilog files into the `Lab2` directory
- Zip the file using the command `> zip Lab2.zip Lab2/*`
- Submit the zip using the submit utility on the course webpage.