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 "`Instruction`" 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 instructions where the size of the payload is arbitrarily decided using the `randomize()` directive. Each instruction 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 DUT and the valid commands for it remain the same as the last lab.

Lab2 Requirements:
Please download all the files below into a single directory as you did in Lab1:

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 `Basic_ALU.tb.sv`. The `Basic_ALU.test_top.sv` and `Basic_ALU.if.sv` remain unchanged from Lab1. To begin with, we look at the contents 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 `Instruction` class is shown below:

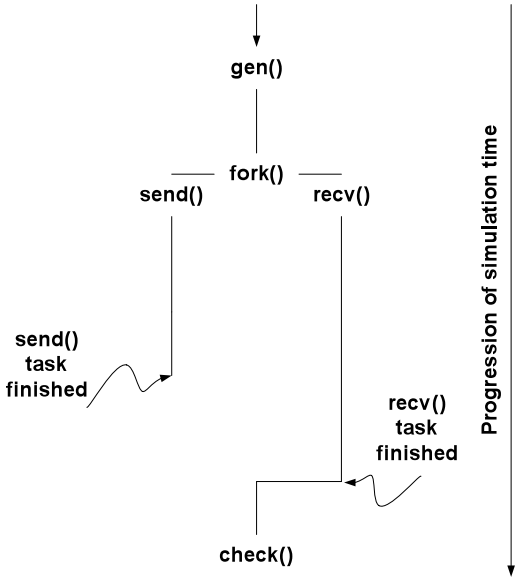
```
class Instruction;

  rand   reg [`REGISTER_WIDTH-1:0]       src1;
  rand   reg [`REGISTER_WIDTH-1:0]       src2;
  rand   reg [2:0]                       alu_operation;

                                  ensuring relevant fields are randomizable
                                   (could also be randc)
  string   name;

  constraint Limit {                    Constraint (called Limit) on class Instruction
      src1 inside {[0:65534]};
      src2 inside {[0:65534]};
      alu_operation inside    {[1:6]};
}
```

Also note the use of "`rand`" which allows for the randomization of the necessary fields (`src1, src2, alu_operation`) within the Instruction. 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 will also note that this exercise has a `check()`(within `Basic_ALU.tb.sv`) task called at the end that checks for the correctness of, at present, only the arithmetic operations that the Basic_ALU unit performs.

The method of testing followed in this laboratory exercise is: a) Create a payload of randomized Instructions (shown above) with appropriate constraints `gen()` b) Send all the instructions 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 Instruction");
      gen();
      fork
         send();
         recv();
      join
      check();
   end
   repeat(5) @(Basic_ALU.cb);
end
```

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

```
task gen();

   number_instructions = $urandom_range(3,12);

   GenInstructions = new[number_instructions];
   for (i=0; i<number_instructions; i++) begin
      GenInstructions[i] = new();
   end
   for (i=0; i<number_instructions; i++)  begin
      inst2send.name = $psprintf("Instruction[%0d]", i);
      if (!inst2send.randomize()) begin
         $display("\n%m\n[ERROR]gen():Randomization Failed!", $time);
         $finish;
      end

      GenInstructions [i].src1 = inst2send.src1;
      GenInstructions [i].src2 = inst2send.src2;
      GenInstructions [i].alu_operation = inst2send.imm;
         end
endtask
```

allocate memory for dynamic array of Instruction pointers and call constructor for `number_instructions` instances of Instruction

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

move the randomized instruction into the dynamic array

The `inst2send.randomize()` command randomizes the fields of the `inst2send` instruction which have the "`rand`" qualifier. If the command can not be executed it returns a 0 which is used to perform an `if(inst2send.randomize()==0)` check for error. This works in conjunction with the declaration of the Instruction 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();


   instructions_sent = 0;
   i = 0;

   while(i < GenInstructions.size()) begin
      inst2send = GenInstructions [i];
      ALU_Interface.cb.src1<= inst2send.src1;
      ALU_Interface.cb.src2<= inst2send.src2;
      ALU_Interface.cb.alu_operation<=inst2send.alu_operation;

      instructions_sent++;
      Inputs.push_back(inst2send);
```

grab next instruction from dynamic array

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

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

```
      @(ALU_Interface.cb);
   end

      GenInstructions.delete();
endtask
```

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

The `Inputs.push_back(inst2send);` 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. This task has the following structure:

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

the first `@(ALU_Interface.cb)` is used to ensure that the outputs are grabbed after two clock cycles. 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 = Basic_ALU.cb.aluout;

   aluout_q.push_back(aluout2cmp);

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, alu_operation`) 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:

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

```
   > vlog *.vp
   >  vlog  -sv  ALU_Interface.if.sv  ALU_Interface.test_top.sv
ALU_Interface.tb.sv
   > vsim -novopt Basic_ALU_test_top
```

As stated above, the checker has been used, at present, to perform only arithmetic operation checking. Answer the following questions for the regular (unparallel) check() task and parallel check() task.

1. Note the time at which the send() recv () and the check () tasks start and stop if you send in 50 instructions.
2. Run multiple inputs into the DUT (mostly by using the correct constraints within the Instruction class) and determine correctness of the various DUT result for inputs. Note the time at which the checker is done performing for 100 instructions sent into the DUT and repeated 5 times.
3. Make appropriate changes to the code to generate number of instructions sent into the DUT randomly between 10 to 100.
4. Draw a timing diagram for all the tasks in the testbench with regular check() function and parallel check() function.

NOTE: Please pay close attention at the instant where you shall be popping values from your queues for checking. i.e, both your DUT queues (containing values from the DUT) and the payload queues (containing instructions sent into the DUT) so that your checking task output is in sync with the output you observe from the DUT at that instant.

NOTE: Parallel check task means check() task included within fork and join.