Cross Coverage & SVA Assignment

Q1. Add the following cross coverage in the class of the ALSU of the last assignment:

1. When the ALU is addition or multiplication, A and B should have taken all permutations of maxpos, maxneg and zero.
2. When the ALU is addition, c_in should have taken 0 or 1
3. When the ALSU is shifting or rotating, then direction must take 0 or 1
4. When the ALSU is shifting, then shift_in must take 0 or 1
5. When the ALSU is OR or XOR and red_op_A is asserted, then A took all walking one patterns (001, 010, and 100) while B is taking the value 0
6. When the ALSU is OR or XOR and red_op_B is asserted, then B took all walking one patterns (001, 010, and 100) while A is taking the value 0
7. Covering the invalid case: reduction operation is activated while the opcode is not OR or XOR

Note: You are free to add more cross coverage but make sure to document them in your verification plan. Also, you can breakdown any coverpoints in the class into multiple coverpoints if needed.

Q2. Write assertions for the following:

1) Write assert property statement, if signal a is high in a positive edge of a clock then signal b should be high after 2 clock cycles
2) Write assert property statement, If signal a is high and signal b is high then signal c should be high 1 to 3 clock cycle later
3) Write a sequence s11b, after 2 positive clock edges, signal b should be low
4) Write a property for the following specs:
    - 3-to-8 decoder output **Y**
        i. At each positive edge of clock, Y output must be only one bit high.
    - 4-to-2 priority encoder output **valid** (refer to assignment 1)
        i. At each positive edge of clock, if the input D bits are low then after one clock cycle, output valid must be low.

Q3. Verify the functionality of counter provided in the assignment 2 by adding assertions to the design. Change the reset of the design to asynchronous.

1- Adjust your verification requirements document in the functionality check column, add if the requirement is checked against golden model, assertion or both. In case of assertion then mention its label used in your SVA module

Label	Design Requirement Description	Stimulus Generation	Functional Coverage	Functionality Check
COUNTER_1	When the reset is asserted, the output counter value should be low	Directed at the start of the sim, then randomized with constraint that drive the reset to be off most of the simulation time.	-	Immediate assertion to check for the async reset functionality
COUNTER_2	When the load is asserted, the output count_out should take the value of the load_data input	Randomization under constraints on the load signal to be off 70% of the time	Cover all values of load data	Concurrent assertion to check the load_data

- To check the asynchronous reset, you can use the following example

```
always_comb begin
  if(reset)
   a_reset: assert final(out == 0);
end
```

2- Create an interface with DUT and TEST modports
3- Create the testbench as instructed in assignment 2 pdf but modify it to take an interface with modport TEST and do the necessary modifications in the testbench as we have done in the class. Also, remove the golden model since we will depend on the assertions only to fully verify the counter's functionality.
4- Create a top module where clock gen will take place and connections of interface.
5- Create SVA module and bind it in the top module with the following assertions (feel free to add more to enrich your verification process)
   i. When the load control signal is active, then the dout has the value of the din
   ii. When the load control signal is not active, and the enable is off then the dout does not change
   iii. When the load control signal is not active and the enable is active, and the up_down is high then the dout is incremented.
      - **Note:** to check for the increment, please use 1'b1 and do not use 1 since the simulator does not behave as expected for assertions using a 32-bit decimal
   iv. When the load control signal is not active and the enable is active, and the up_down is low then the dout is decremented.
      - **Note:** to check for the decrement, please use 1'b1 and do not use 1 since the simulator does not behave as expected for assertions using a 32-bit decimal
   v. When the asynchronous reset is asserted then the counter output is tied to low at the same instant. Note that you can use assert final to sample the new value of a signal.
   vi. max_count output is high when the counter output is maximum.
   vii. zero output is high when the counter output is zero.
6. Generate the coverage report and make sure to reach 100% code coverage, functional coverage and assertion coverage.

Q4. A configuration register is a bank of registers that is used to store setup information for the system (i.e. the configuration). All registers can be read and written and are 16-bits wide. The design contains 8 registers as defined in Table 1 along with their reset values. All bits are writable. The reset is active high.

The design has the following inputs:

- clk
- reset
- address
- data_in
- write: Write enable signal to enable sequential writing of the appropriate register below with data_in according to the address
- dataout: combinational dataout that holds the value of the appropriate register below according to the address

Register	Width	Address	Reset Value
adc0_reg	[15:0]	0	16'hFFFF
adc1_reg	[15:0]	1	16'h0
temp_sensor0_reg	[15:0]	2	16'h0
temp_sensor1_reg	[15:0]	3	16'h0
analog_test	[15:0]	4	16'hABCD
digital_test	[15:0]	5	16'h0
amp_gain	[15:0]	6	16'h0
digital_config	[15:0]	7	16'h1

Table 1: Config Register

Write a verification plan describing how you are going to verify the design. Then write a self-checking testbench to perform the verification. You do not have to add constraints/functional coverage or assertions to focus on debugging the bugs in the design using write and read operations. Just add in your verification plan what stimulus will be written and read from the design.

Your testbench will:

1. Use a user defined enumerated type that hold the values of the registers
2. Declare a variable from the enum. This variable will be assigned to the address connected to the DUT.
3. Use associative array, as a golden model for the reset values of the registers.
    a. Name the array “reset_assoc”
    b. Key of the array is string
4. Use a task for reset and a task to check the result
5. Use the enum methods .num/.last and .next inside of your loops to iterate on all of the registers

There is 1 bug per register. An encrypted version of the buggy configuration register code is provided. Compile this module just like an unencrypted module is compiled.

Identify the 8 bugs in config_reg_buggy.

For each bug, report the bugs in the following format:

a) Design Input for Bug to Appear:

E.g. Write FFFF to register x

b) Expected Behavior:

E.g. Bit 15 of register x is writable to a 1

c) Observed Behavior

E.g. Bit 15 of register x cannot be written to a 1

Remember, your bug report should contain enough detail for the designer of the configuration register to debug the problem.

Deliverables:

1. Verification plan
2. Testbench code testing the whole conf. reg.
3. Complete bug report for each register
    a. Optional: Add a separate small directed testbench for each bug detected to help the designer run this small testbench and debug the issue.
4. Separate snippets taken from QuestaSim for each bugs detected

Deliverables:

One PDF file having the following.

1. Testbench code
2. Top module code
3. SVA module code
4. Package code
5. Design code
6. Do File & Snippet to your verification requirement document
7. Code Coverage, Functional Coverage and assertion coverage report snippets
8. QuestaSim snippets