SV Randomization & Functional Coverage Assignment

Deliverables are mentioned at the end of the document.

Q1. Modify on the ALU deign done in the class to reach 100% code and functional coverage.

1. Any exclusions must be justified.
2. Make the testbench self-checking by adding tasks to automate checking.
3. Add three tasks in the testbench:
   - One for reset.
   - One to calculate the expected output.
   - One to compare the expected output with the design output.
4. Implement correct_count and error_count counters to track verification results. Display them at the end of the simulation.
5. Use a DO file to automate running the simulation and generating the coverage report.

Q2. Add functional coverage in the class for the counter of assignment 2.

Note: Add the functional coverage in the functional coverage column of your verification plan.

Below is a snippet of the verification plan for reference:

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.	-	A checker in the testbench to make sure the output is correct
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	A checker in the testbench to make sure the output is correct
COUNTER_3	When the load/reset is deasserted, the output count_out increment as long as the enable is active and up_down is high	Randomization for up_down, and randomization for enable to be 70% of the time	Cover all values of count_out, and transition bin from max to zero	A checker in the testbench to make sure the output is correct

Functional Coverage – to be sampled with the positive edge of the clock

1. Coverpoint for load data when load is asserted and reset is deasserted
2. Coverpoint for count out if the reset is deasserted, enable is active and up_down is high
   a. Autogenerate bins for all values
3. Coverpoint for count out if the reset is deasserted, enable is active and up_down is high
   a. Transition bin to check when overflow occurs (maximum value to zero)
4. Coverpoint for count out if the reset is deasserted, enable is active and up_down is low
   a. Autogenerate bins for all values
5. Coverpoint for count out if the reset is deasserted, enable is active and up_down is low
   a. Transition bin to check when underflow occurs (zero to maximum value)

You are free to add more coverpoints to enrich your verification to reach 100% functional coverage. Use a do file to compile the package, design and testbench then simulate and save the coverage. Finally generate the code and functional coverage report.

Q3. Expand the testbench done for the ALSU done in assignment 2.

Note: Add the functional coverage in the functional coverage column of your verification plan.

1. Functional coverage model in covergroup cvr_gp inside of the class. <u>You are free to breakdown the coverpoints mentioned below into multiple coverpoints if needed.</u>
    - 2 Coverpoints for ports A and B (A_cp and B_cp)
    - Coverpoint A_cp will cover the following bins

Bins	Values	Comment
A_data_0	0	-
A_data_max	MAXPOS	-
A_data_min	MAXNEG	-
A_data_default	Remaining values	-
A_data_walkingones[]	001, 010, 100	If only the red_op_A is high

    - Coverpoint B_cp will cover the following bins

Bins	Values	Comment
B_data_0	0	-
B_data_max	MAXPOS	-
B_data_min	MAXNEG	-
B_data_default	Remaining values	-
B_data_walkingones[]	001, 010, 100	If only the red_op_B is high and red_op_A is low

    - Create a cover point ALU_cp with the following bins

Bins	Values
Bins_shift[]	Generate bins for shift and rotate opcodes
Bins_arith[]	Generate bins for add and mult opcodes
Bins_bitwise[]	Generate bins for or and xor opcodes
Bins_invalid	Illegals bins for opcodes 6 or 7
Bins_trans	Transition from opcode 0 > 1 > 2 > 3 > 4 > 5

There are 7 constraints added in assignment 3. You are required to add an eighth constraint as follows:

- Create a fixed array of 6 elements of type opcode_e. constraint the elements of the array using foreach to have a unique valid value each time randomization occurs.

Testbench Implementation Steps

1. Sampling Logic:

- Use the sample task to sample the values for the covergroup.

- Stop sampling when reset, bypass_A, or bypass_B is asserted.

2. Stimulus Generation (Two Loops):

- First Loop:

  - Randomize inputs with all constraints 1 to 7 from Assignment 2 enabled.

  - Disable constraint number 8.

- Before Starting the Second Loop:

  - Disable all constraints.

  - Force the following inputs to zero: rst, bypass_A, bypass_B, red_op_A, red_op_B.

  - Enable constraint number 8.

3. Second Loop (Opcode Verification):

- Apply valid opcodes to heavily verify opcode functionality while randomizing other inputs.

- Constraint 8 will generate a unique sequence of valid opcodes.

- Other inputs except for the mentioned in step 2 will be randomized without constraints since constraint 1 to 7 are disabled

- Nested Loop (6 Iterations):

  - Iterate over the unique sequence of opcodes while the other randomized inputs have constant values across the 6 iterations.

4. Coverage Analysis:

- Check code and functional coverage reports.

- Modify the testbench as needed to achieve 100% coverage.

- Avoid directed test patterns unless randomization alone is insufficient.

5. Golden Model for Expected Output:

- Either write your own Verilog design and instantiate it as a golden model.

- Or create a task (golden_model) to return expected output values.

6. Bug Reporting and Debugging:

- Report any detected bugs in the design.

- Fix them before finalizing the testbench.

Note: You are free to add more constraints or functional coverage points to enrich your verification. If you will add any, you must document this in your verification document.

Use a do file to compile the package, design and testbench then simulate and save the coverage. Finally generate the code and functional coverage report.

Q4. The design to be tested is a synchronous single-port 8-bit x64K (512kBit) RAM. The RAM will read on the positive edge of the clock when input read =1 and write on the positive edge of the clock when input write = 1. Write enable signal has a higher priority than the read enable signal and both write and read data from the RAM is not allowed at the same time. Even parity will be calculated on data written to the RAM and placed in the 9th bit of the memory. The partially completed memory model is below (add the memory declaration)

```
module my_mem(
                    input clk,
                    input write,
                    input read,
                    input [7:0] data_in,
                    input [15:0] address,
                    output reg [7:0] data_out
);

  // Declare a 9-bit associative array using the logic data type & the key of int datatype
  <Put your declaration here>

  always @(posedge clk) begin
    if (write)
      mem_array[address] = {~^data_in, data_in};
    else if (read)
      data_out =  mem_array[address];
  end

endmodule
```

Testbench Implementation Steps:

1. Setup and Data Structures

- Define a local parameter TESTS = 100 to specify the number of write/read operations.
- Create dynamic arrays:
  - address_array: Stores random addresses.
  - data_to_write_array: Stores corresponding random data values.
- Create an associative array:
  - data_read_expect_assoc: Stores expected data, indexed by address (key datatype: int).
- Create a queue:
  - data_read_queue: Stores the actual data read from the RAM.
- Implement:
  - Clock generation
  - DUT instantiation
  - Counters (error_count and correct_count)

2. Task Implementations

- Task: stimulus_gen
  - Loops TESTS times to generate:
    - Random addresses stored in address_array.
    - Random data values stored in data_to_write_array.
- Task: golden_model
  - Loops TESTS times to populate data_read_expect_assoc using:
    - address_array as the index.
    - data_to_write_array as expected values.
- Task: check9Bits
  - Checks if the expected data out of the RAM is correct or not

3. Initial block Implementation

3.1 Data preparation

- Call the stimulus_gen and then the golde_model tasks to prepare the data to be sent and expected data to be read from the RAM

3.2 Write Operations

- Loop TESTS times to write data into the DUT:

  - On each negative clock edge, drive the address and data_in ports using address_array and data_to_write_array.

3.4 Read and Self-Checking

- Loop TESTS times to read data from the DUT:

  - Drive the address port using address_array.

  - Call check9Bits task to compare:

    - Data read vs. data_read_expect_assoc[address].

  - Store the read data into data_read_queue.

3.5 Test Completion and Reporting

- Print out the read data stored in data_read_queue using a while loop and pop_front method (search online how to do it).

- Display error_count and correct_count.

Requirements:

1. Create verification plan based on the previous requirements
2. Use a do file to compile the package, design and testbench then simulate and save the coverage.
3. You do not have to analyze the coverage or add constraints or functional coverage for this example. However, feel free to add any and mention them in your verification plan.

Deliverables:

One PDF file having the following

1. Testbench code
2. Package code
3. Design code
4. Report any bugs detected in the design and fix them
5. Snippet to your verification plan document
6. Do file
7. Code Coverage report snippets (Questions 1, 2, and 3 only)

8. Functional Coverage report snippets (Questions 1, 2, and 3 only)
9. **Clear and neat** QuestaSim waveform snippets showing the functionality of the design
10. Your PDF file must have this format <your_name>_Assignment3 for example Kareem_Waseem_Assignment3