SV Randomization & Functional Coverage Assignment

Q1.

Create a SystemVerilog code for the following requirements.

- Create a queue of datatype byte and initialize it with 2, -1, and 127
- Print out the sum of the queue in decimal
- Print out the min and max values in the queue
- Sort all values in the queue and print out the resulting queue
- Print out the index of any negative values in the queue
- Print out the zero and positive values in the queue
- Reverse sort all values in the queue and print out the resulting queue

Q2.

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 testplan using bullet points describing how you are going to test the design. Then write a self-checking testbench to perform the testing you described in your testplan.

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 and .next (slide 78) 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:

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.

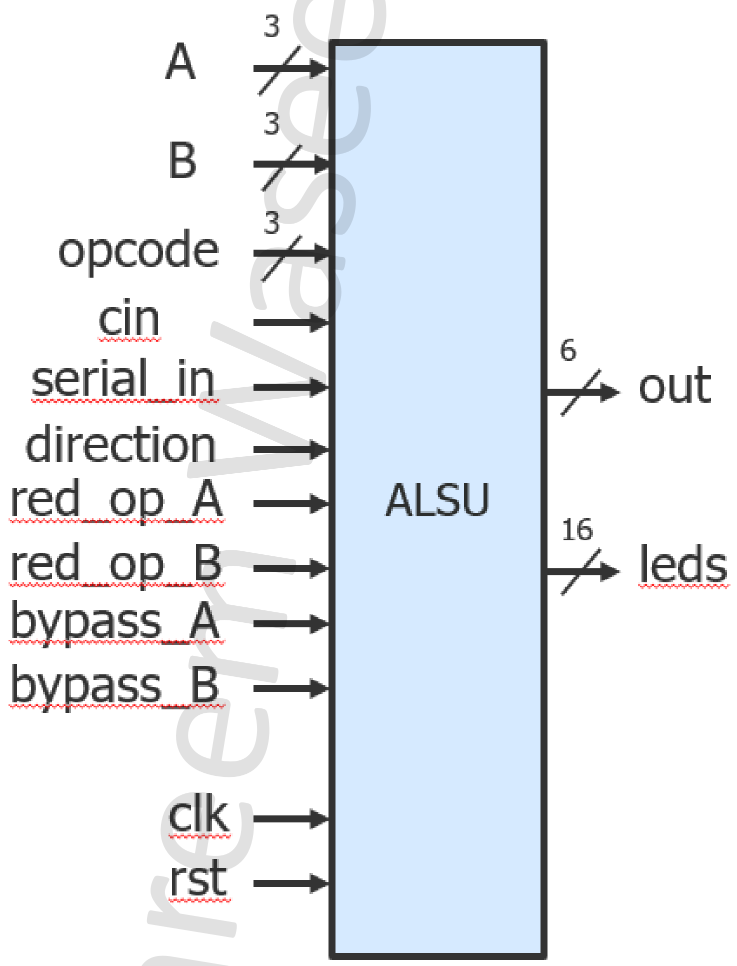
Deliverables:

1. Test plan
2. Testbench code testing the whole conf. reg.
3. Complete bug report for each register and a separate small 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

Q3.

ALSU is a logic unit that can perform logical, arithmetic, and shift operations on input ports

- Input ports A and B have various operations that can take place depending on the value of the opcode.
- Each input bit except for the clk and rst will be sampled at the rising edge before any processing so a D-FF is expected for each input bit at the design entry.
- The output of the ALSU is registered and is available at the rising edge of the clock.



Inputs

Each input bit except for the clk and rst will have a DFF in front of its port. Any processing will take place from the DFF output.

Input	Width	Description
clk	1	Input clock
rst	1	Active high asynchronous reset
A	3	Input port A
B	3	Input port B
cin	1	Carry in bit, only valid to be used if the parameter FULL_ADDER is "ON"
serial_in	1	Serial in bit, used in shift operations only
red_op_A	1	When set to high, this indicates that reduction operation would be executed on A rather than bitwise operations on A and B when the opcode indicates OR and XOR operations
red_op_B	1	When set to high, this indicates that reduction operation would be executed on B rather than bitwise operations on A and B when the opcode indicates OR and XOR operations
opcode	3	Opcode has a separate table to describe the different operations executed
bypass_A	1	When set to high, this indicates that port A will be registered to the output ignoring the opcode operation
bypass_B	1	When set to high, this indicates that port B will be registered to the output ignoring the opcode operation
direction	1	The direction of the shift or rotation operation is left when this input is set to high; otherwise, it is right.

Outputs and parameters

Output	Width	Description
leds	16	When an invalid operation occurs, all bits blink (bits turn on and then off with each clock cycle). Blinking serves as a warning; otherwise, if a valid operation occurs, it is set to low.
out	6	Output of the ALSU

Parameter	Default value	Description
INPUT_PRIORITY	A	Priority is given to the port set by this parameter whenever there is a conflict. Conflicts can occur in two scenarios, red_op_A and red_op_B are both set to high or bypass_A and bypass_B are both set to high. Legal values for this parameter are A and B
FULL_ADDER	ON	When this parameter has value "ON" then cin input must be considered in the addition operation between A and B. Legal values for this parameter are ON and OFF

Opcodes & Handling invalid cases

Invalid cases

1. Opcode bits are set to 110 or 111
2. red_op_A or red_op_B are set to high and the opcode is not OR or XOR operation

Output when invalid cases occurs

1. leds are blinking
2. out bits are set to low

Opcode	Operation
000	OR
001	XOR
010	ADD
011	MULT
100	SHIFT (Shift output by 1 bit)
101	ROTATE (Rotate output by 1 bit)
110	Invalid opcode
111	Invalid opcode

Testbench:

- You are required to verify the functionality of the ALSU under the default configuration only.
- Note: Some of the verification requirements mentioned below are extracted from assignment 1 solution which was done using directed test case patterns, check it out if you have not already.

Requirements:

1. Create a verification requirement document to support your verification planning, an example of the document can be found in the link here. Please copy this document to have your own version and fill the document with the design requirements.
2. Add comments in your testbench and class with the labels taken from your verification requirement document for easy tracking of the implementation of constraints, and functional coverage model.

Create a package that have a user defined enum opcode_e that takes the value of the opcode, you can name the invalid cases INVALID_6 and INVALID_7. Also, create another user defined enum opcode_valid_e that will take only valid opcodes. Use different string values for each enum to avoid compilation errors.

3. Create a class in the package to randomize the design inputs under the following constraints
    1. Reset to be asserted with a low probability that you decide
    2. Constraint for adder inputs (A, B) to take the values (MAXPOS, ZERO and MAXNEG) more often than the other values when the ALU is addition or multiplication.
    3. In case of ALU opcode OR or XOR and red_op_A is high, constraint the input A most of the time to have one bit high in its 3 bits while constraining the B to be low
    4. In case of ALU opcode OR or XOR and red_op_B is high, constraint the input B most of the time to have one bit high in its 3 bits while constraining the A to be low
    5. Invalid cases should occur less frequent than the valid cases
    6. bypass_A and bypass_B should be disabled most of the time
    7. Do not constraint the inputs A or B when the operation is shift or rotate
    8. Create a 2-D fixed array of type opcode_valid_e. constraint the elements of the array using foreach to have a unique value each time randomization occurs

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

Bins	Values
data_0	0
data_max	MAXPOS
data_min	MAXNEG
data_default	Remaining values
Data_walkingones	001, 010, 100

- Create a cover point ALU_cp with the following bins

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

- Cross coverage
  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 or multiplication, c_in should have taken 0 or 1
  3. When the ALSU is shifting, then shift_in must take 0 or 1
  4. When the ALSU is shifting or rotating, then direction 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 (0001, 0010, 0100, and 1000) 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 (0001, 0010, 0100, and 1000) while A is taking the value 0
  7. Invalid case to be covered: reduction operation is activated while the opcode is not OR or XOR

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

➢ In your testbench
  - Use sample task to sample A and B values. Stop sampling the input values when the reset or the bypass_A or the bypass_B are asserted.
  - Use 2 loops in your stimulus generation, the first loop will randomize the inputs and all constraints (numbered from 1 to 7) are enabled, disable constraint number 8 (check the previous page that has the constraints). After the first loop finishes, the second loop will start
  - Before starting the second loop, disable all constraints, force bypass_A, bypass_B, red_op_A and red_op_B to zero and enable constraint number 8. This loop will apply valid cases with randomized data inputs, constraint number 8 will return a unique sequence of valid opcodes, create a nested loop of 6 iterations inside the second loop to loop on that unique sequence of opcodes while keeping the other inputs having the same randomized value during the 6 iterations
  - Check the code coverage and functional coverage reports and modify the testbench as necessary to achieve 100% code coverage and functional coverage. Don't use directed test patterns until you could not reach 100% functional coverage using the randomization.

- Every time you check for the result if it is correct or not, call a task called golden_model where it will return the expected values of the outputs. Check the output after 2 clock cycles since there are 2 stages inside of the design.
- Report any bugs detected in the design and fix them

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.

Deliverables:

One PDF file having the following

1. Testbench code
2. Package code
3. Design code
4. Snippet to your verification requirement document
5. Do file
6. Code Coverage & Functional Coverage report snippets
7. QuestaSim snippets