Midterm Lab Exam

Instructions: Please Read Carefully Before Proceeding.

1. This is a group work exam (same group you work with in the lab).
2. Please reference appropriately any documents, URLs, or books that you use.
3. Collaboration is allowed only within the group.
4. This exam contains 5 pages, including this one**:** Pages 2-4 briefly explain the concept of randomization in System Verilog. Read carefully and then follow the instructions on page 5.
5. Only electronic submissions through Moodle are allowed.
6. Submissions will only be accepted in the form of one PDF file for the report and one zip/rar folder for the design files.
7. The report should have:
    - All group members' names and IDs on the front page.
    - Tasks carried out by each group member.
    - Comments on the design, if any.
8. The deadline for submitting this lab exam is **14th of April.**

References

[1] C. Spear, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer Science & Business Media, 2008.

Randomization:

As designs grow larger, it becomes more difficult to create a complete set of stimuli needed to check their functionality. You can write directed test cases to check a certain set of features, but you cannot write enough directed test cases when the number of features keeps doubling on each project.

The solution is to create test cases automatically using Constrained Random Tests (CRT). A directed test finds the bugs you think are there, but a CRT finds bugs you never thought about by using random stimuli. You restrict the test scenarios to those that are both valid and of interest by using constraints.

A CRT requires an environment to predict the result using a reference model, transfer function, or other techniques, plus functional coverage to measure the effectiveness of the stimulus.

A CRT is made of two parts: the test code that uses a stream of random values to create input to the DUT and a seed to the pseudo-random number generator (PRNG). You can make a CRT behave differently just by using a new seed.

What to Randomize:

- Device configuration
- Environment configuration
- Primary input data
- Encapsulated input data
- Protocol exceptions
- Delays
- Transaction status
- Errors and violations

Randomization in SystemVerilog

Randomization is the process of making something random; SystemVerilog randomization is the process of generating random values to a variable. Verilog has a $random method for generating the random integer values. This is good for randomizing the variables alone, but it is hard to use in case of class object randomization. For easy randomization of class properties, SystemVerilog provides **rand** keyword and **randomize()** method. You can create constraints to limit the random values to legal values, or to test specific features.

Random variables

The class variables that get random values on randomization are called random variables. To make variables as random variables, Class variables need to be declared using the **rand** and **randc** type-modifier keywords.

The following types can be declared as rand and randc:

- singular variables of any integral type
- arrays
- arrays size
- object handle's

rand keyword:

Variables declared with the rand keyword are standard random variables. Their values are uniformly distributed over their range.

randc keyword:

randc is random-cyclic. For the variables declared with the randc keyword, on randomization variable values don't repeat a random value until every possible value has been assigned.

As we said before, class variables need to be declared using the rand and rand to be randomized later. However, you should not randomize an object in the class constructor. Your test may need to turn constraints on or off, change weights, or add new constraints before randomization.

The randomize function assigns random values to any variable in the class that has been labeled as rand or randc , and also makes sure that all active constraints are obeyed. Randomization can fail if your code has **conflicting constraints**, so always check the status. If you don't check, the variables may get unexpected values, causing your simulation to fail.

The SystemVerilog constraint solver handles the process of solving constraint expressions. The solver chooses values that satisfy the constraints. The values come from SystemVerilog's PRNG (pseudo-random number generator), which is started with an initial seed. If you give a SystemVerilog simulator the same seed and the same testbench, it should always produce the same results.

What if, for some reason, it is required not to generate a random value for a random variable, or you have many constraints in your class, and you choose which one you need for now? Yes, it is possible to disable the randomization of a variable by using the SystemVerilog randomization method **rand_mode**.

- rand_mode(1) means randomization enabled.
- rand_mode(0) means randomization disabled.
- The default value of rand_mode is 1, i.e enabled
- syntax: <object_hanlde>.<variable_name>.rand_mode(enable);

Constraint:

By writing constraints to a random variable, the user can get specific value on randomization. Constraints to a random variable shall be written in constraint blocks.

Constraint blocks:

- Constraint blocks are class members like tasks, functions, and variables
- Constraint blocks will have a unique name within a class.
- Constraint blocks consist of conditions or expressions to limit or control the values for a random variable
- Constraint blocks are enclosed within curly braces { }
- Constraint blocks can be defined inside the class or outside the class like extern methods, constraint block defined outside the class is called as extern constraint block.
- Syntax:

```
constraint <constraint_block_name> { <condition/expression>;
                                     .....
                                     <condition/expression>; }
```

Some of the Constraint Types:

1. Simple Expressions
   Such as <, <=, ==, >=, or >

2. Equivalence Expressions
   Assignment in a constraint block using ==

3. Weighted Distributions
   The dist operator allows you to create weighted distributions so that some values are chosen more often than others. Using **: =** or **: /** operator.
   The : = operator specifies that the weight is the same for every specified value in the range, whereas the : / operator specifies that the weight is to be equally divided between all the values.

4. Inside constraint
   During randomization, it might require to randomize the variable within a range of values or with inset of values or other than a range of values.

5. Implication if else Constraints
   Used to declaring conditional relations between two variables. Implication operator is denoted by the symbol ->
   The implication operator is placed between the expression and constraint.

Midterm: FIFO Testbench

Your task is to verify a FIFO design based on the provided RTL code "*FIFO.v*".

It is essential to keep the RTL design itself unchanged.

You are required to include the following in your final submission:

1) Interface Creation: Develop an interface that uses a clocking block to handle the clock signal for synchronous operations, ensuring proper sampling of input signals and driving of output signals.
2) Input Randomization: Randomize the input data for the FIFO according to the following constraints:
    a. The first 8 bits of the input data (data_in) should fall within the range [100:230].
    b. The second 8 bits of the input data should fall within the range [200:255].
    c. The third 8 bits of the input data should be generated with the following probabilities:
    d. Range [0:100] with a probability of 30%.
    e. Range [100:200] with a probability of 60%.
    f. Range [200:255] with a probability of 10%.
    g. The last 8 bits of the input data should fall within the range [0:50] if the first 8 bits are greater than 150; otherwise, they should fall within the range [0:255].
    h. Randomize the Read_enable and Write_enable signals with appropriate weights of your choice.
3) Coverage Points: Implement cover points to ensure 100% coverage of the Full Flag and Empty Flag signals.
4) Assertions1: Write an assertion to verify the following condition:
    When the Write_enable signal is asserted and the FIFO is not full, the write pointer (write_ptr) should increment.
5) Assertion 2 (Bonus):
    a. Write any additional assertion that you consider important for verifying the correctness or functionality of the FIFO design.
    b. Explain the purpose and significance of this assertion.
6) Code Coverage: Perform code coverage analysis to assess the effectiveness of your testbench in exercising the RTL code.