Project

Synchronous FIFO

Parameters

- FIFO_WIDTH: DATA in/out and memory word width (default: 16)
- FIFO_DEPTH: Memory depth (default: 8)

Ports

Port	Direction	Function
data_in	Input	Write Data: The input data bus used when writing the FIFO.
wr_en		Write Enable: If the FIFO is not full, asserting this signal causes data (on data_in) to be written into the FIFO
rd_en		Read Enable: If the FIFO is not empty, asserting this signal causes data (on data_out) to be read from the FIFO
clk		Clock signal
rst_n		Active low asynchronous reset
data_out	Output	Read Data: The sequential output data bus used when reading from the FIFO.
full		Full Flag: When asserted, this combinational output signal indicates that the FIFO is full. Write requests are ignored when the FIFO is full, initiating a write when the FIFO is full is not destructive to the contents of the FIFO.
almostfull		Almost Full: When asserted, this combinational output signal indicates that only one more write can be performed before the FIFO is full.
empty		Empty Flag: When asserted, this combinational output signal indicates that the FIFO is empty. Read requests are ignored when the FIFO is empty, initiating a read while empty is not destructive to the FIFO.
almostempty		Almost Empty: When asserted, this output combinational signal indicates that only one more read can be performed before the FIFO goes to empty.
overflow		Overflow: This sequential output signal indicates that a write request (wr_en) was rejected because the FIFO is full. Overflowing the FIFO is not destructive to the contents of the FIFO.
underflow		Underflow: This sequential output signal Indicates that the read request (rd_en) was rejected because the FIFO is empty. Under flowing the FIFO is not destructive to the FIFO.
wr_ack		Write Acknowledge: This sequential output signal indicates that a write request (wr_en) has succeeded.

Overview of the testbench flow:

The top module will generate the clock, pass it to the interface, and the interface will be passed to the DUT, tb, and monitor modules. The tb will reset the DUT and then randomize the inputs. At the end of the test, the tb will assert a signal named test_finished. The signal will be defined as well as the error_count and correct_count in a shared package that you will create named shared_pkg.

The monitor module will do the following:

1. Create objects of 3 different classes (FIFO_transaction, FIFO_scoreboard, FIFO_coverage). These classes will be discussed later in the document.
2. It will have an initial block and inside it a forever loop that has a negedge clock sample. With each negedge clock, the monitor will sample the data of the interface and assign it to the data of the object of class FIFO_transaction. And then after that there will be fork join, where 2 processes will run, the first one is calling a method named sample_data of the object of class FIFO_coverage and the second process is calling a method named check_data of the object of class FIFO_scoreboard.
3. So, in summary the monitor will sample the interface ports, and then pass these values to be sampled for functional coverage and to be checked if the output ports are correct or not.
4. After the fork join ends, you will check for the signal test_finished if it is high or not. If it high, then stop the simulation and display a message with summary of correct and error counts.

Steps:

1. Adjust the design to take an interface and change the file extension to sv.
2. Create a package in a new file that will have a class named FIFO_transaction
    a. Inside of this class add the FIFO inputs and outputs as properties of the class as well as adding 2 integers (WR_EN_ON_DIST with default 70 and RD_EN_ON_DIST with default 30)
    b. Add the following 3 constraint blocks
        1. Assert reset less often
        2. Constraint the write enable to be high with distribution of the value WR_EN_ON_DIST and to be low with 100-WR_EN_ON_DIST
        3. Constraint the read enable the same as write enable but using RD_EN_ON_DIST
3. Create a package in a new file that will have a class for functional coverage collection named FIFO_coverage
    a. Import the previous package (Add the import statement after the package declaration)
    b. The class will have an object of the class FIFO_transaction named F_cvg_txn.
    c. Create a function inside it named sample_data that takes one input named F_txn. The input is an object of class FIFO_transaction. This function will do the following
        1. Assign F_txn to F_cvg_txn
        2. Trigger the sampling of the covergroup using the .sample method
    d. Create a covergroup. The coverage needed is cross coverage between 3 signals which are write enable, read enable and each output control signals (outputs except data_out) to make sure that all combinations of write and read enable took place in all state of the FIFO.
4. Create a package in a new file named FIFO_scoreboard
    a. Import the FIFO_transaction package.
    b. Add variables for the data_out_ref, full_ref, etc. to be used in the reference_model function

c. Create a function named check_data that takes one input which of type FIFO_transaction
    1. Inside this function, call another function named reference_model that you will create and pass to it the same object that you have received
    2. Reference_model function will check the input values from the input object and assign values to the class properties data_out_ref, full_ref, etc.
    3. After the reference_model function returns, you will compare the reference outputs calculated with the outputs of the object received. Increment the error_count or correct_count. Also, display a message if error occurs.

5. Open the design file and add assertions to the FIFO inside the design file.
    a. Add assertions to all the FIFO output flags (outputs except data_out) as well as the internal counters of the FIFO.
    b. Extra part to be done: Guard the assertions using conditional compilation1 with the `ifdef directive with macro named SIM, and then include the macro in the vlog command +define+SIM option in the vlog command of your do file. Refer to this link to learn more about conditional compilation.

Note:

Inside of the reference model, if you would like to have parallel processes for write and read then use fork join. Also, it is fine if you have previously designed a FIFO and would like to use as a reference model instead of using the function defined in the scoreboard but doing this will make you do some changes to the testbench to send the stimulus to the scoreboard.

Requirements

1. Verification Plan, where you will list your verification plan flow
2. 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 to support your verification. In the functionality check column, you can specify whether the requirements are being checked only by the golden/reference model, assertion or both. Add the assertion label if you will use an assertion for the functionality check for easy tracking.
3. You are free to create your own constraints, coverpoints and assertions. The quality and work done in them will be part of the grading process. Try not to use the directed test stimulus unless necessary.
4. Use Do file to run the top and make sure to generate the coverage report and check that the code coverage, functional coverage, and sequential domain (assertions) coverage are 100%
5. QuestaSim snippets, report any bugs detected and modify the RTL. Report the before and after to show the changes made to the RTL.

Submission file:

.rar file containing the following:

- PDF file having the requirements.
- SystemVerilog files (add snippets to the code in the PDF, it will be easier for the grading process)
- Do file to run top