Lab 3 - Now we're back to assertions in SystemVerilog

EE4449 - HCMUT

Objective and Overview

The purpose of this project is to learn to use the special features of System Verilog that is assertion.

This is an **individual** project, to be done on your Altera DE10 board.

Schedule and Scoring

If you have not uploaded anything by the dropdead date, we will assume you are no longer in the course. Why? Because the syllabus says you must attempt every project. Not uploading anything and not showing up to explain what you've done is not attempting the project — see the syllabus for details.

A Note about Collaboration

Project 3 is to be accomplished individually. All work must be your own.

Hints from others can be of great help, both to the hinter and the hintee. Thus, discussions and hints about the assignment are encouraged. However, the project must be coded and written up individually (you may not show, nor view, any source code from other students). We may use automated tools to detect copying.

Use Emails/BKEL to ask questions or come visit us (203B3) during office hours.
We want to see you succeed, but you have to ask for help.

Project Overview

The assert macro is a very powerful feature of System Verilog HVL (Hardware Verification Language). It allows a programmer to specify assertions and ensure that their code will be correctly implemented. Nowadays it is widely adopted and used in most design verification projects.

May be one day you will be a design verification engineer and you will have to write a lot of assertions. So, it is a good idea to learn how to use it now.

The project is to design an asynchronous FIFO module and verify it using assertions. The FIFO controller module should have the following interface:

```
fifoctrl
    (
     clkw, //clock write
     clkr, //clock read
     rst,

     fiford,    // FIFO control
     fifowr,

     fifofull,  // high when fifo full
```

```
      notempty,  // high when fifo not empty
      fifolen,   // fifo length

                 // Connect to memories
      write,     // enable to write memories
      wraddr,    // write address of memories
      read,      // enable to read memories
      rdaddr     // read address of memories
      );
```

Take a look at **fifoctrl.v** to see how the module is coded. I have to say, it is not a good design. May be there's something wrong with it?

When I say controller, I mean it. The FIFO controller should be able to control the memories. It should be able to write data to the memories and read data from the memories. The FIFO controller should be able to read and write data at the same time. You already know how to use the Altera IP Catalog Memory base on **Lab 2**. You can use it to implement the memories and remember to add the megafunction files for simulation.

The memory module should be able to store 24-bit data. The memory should be able to store 32 data at most. When the memory is full, the memory should not accept any more data. When the memory is empty, the memory should not be able to read any data.

After you design the FIFO Controller and it's memories, you should write a testbench to verify it. The testbench should be able to generate all possible scenarios and check if the FIFO controller works correctly in all of them. If it does, then the testbench should pass. Otherwise, it should fail.

Please write in your report you test plan with three column "Test number | Case name | Case description | Pass Condition | Fail Condition". You should have at least 10 test cases.

Consider the following example of using assertion to check the property of design, do remember I use the name from another module so you have to change it to match your module name.

1. Asynchronous reset assertions

```
  // Reset startup check //
  // need this at the very begining of the simulation //
  property async_rst_startup;
      @(posedge i_clk) !i_rst_n |-> ##1 (wr_ptr==0 && rd_ptr == 0 &&
o_empty);
  endproperty

  // rst check in general
  property async_rst_chk;
      @(negedge i_rst_n) 1'b1 |-> ##1 @(posedge i_clk) (wr_ptr==0 &&
rd_ptr == 0 && o_empty);
  endproperty
```

2. Check data written at a location is the same data read when read_ptr reaches that location

```
  sequence rd_detect(ptr);
    ##[0:$] (rd_en && !o_empty && (rd_ptr == ptr));
  endsequence

  property data_wr_rd_chk(wrPtr);
    // local variable
    integer ptr, data;
    @(posedge i_clk) disable iff(!i_rst_n)
    (wr_en && !o_full, ptr = wrPtr, data = i_data, $display($time, "
wr_ptr=%h, i_fifo=%h",wr_ptr, i_data))
    |-> ##1 first_match(rd_detect(ptr), $display($time, " rd_ptr=%h,
o_fifo=%h",rd_ptr, o_data)) ##0  o_data == data;
  endproperty
```

3. Rule-1: Never write to FIFO if it's Full!

```
  property dont_write_if_full;
    // @(posedge i_clk) disable iff(!i_rst_n) o_full |-> ##1
$stable(wr_ptr);
    // alternative way of writing the same assertion
    @(posedge i_clk) disable iff(!i_rst_n) wr_en && o_full |-> ##1 wr_ptr
== $past(wr_ptr);
  endproperty
```

4. Rule-2: Never read from an Empty FIFO!

```
  property dont_read_if_empty;
    @(posedge i_clk) disable iff(!i_rst_n) rd_en && o_empty |-> ##1
$stable(rd_ptr);
  endproperty
```

5. On successful write, write_ptr should only increment by 1

```
   property inc_wr_one;
      @(posedge i_clk) disable iff(!i_rst_n) wr_en && !o_full |-> ##1
(wr_ptr-1'b1 == $past(wr_ptr));
   endproperty
```

Remember that you don't have to follow the interface of **fifoctrl.v**. You can change it if you want. But you have to write a testbench to verify it. I gave you 5 cases already.

Do you wonder where would the data come from? We're going to implement you FIFO on the DE10 Kit. On testbench, you can use Questa Sim **$random** functions to get random number. However on the DE10 Kit,

you can use a component called "Linear Feed Back Shift Register" to generate random number. You can learn how to create one here: https://en.wikipedia.org/wiki/Linear-feedback_shift_register

The interface from **lfshr.v** should help.

Lastly, create a top module name **top_lab3** to connect the LFSR, FIFO controller, Memory, BCD7SEG, and a clock divider. Use the CLOCK_2SEC 0.5Hz clock that you create your self (write a clock divider module) to drive the LFSR and FIFO controller **clkw**. USE the CLOCK_1SEC 1 Hz that you create your self (write another clock divider module) to drive the FIFO controller **clkr**. Connect the output data of the memory (24-bit) and the fifo_len signal to HEX5 to HEX0 (You should use **SW0** to toggle between the two). Use **KEY0** to turn on write enable for the fifo controller and **KEY1** for read enable. **KEY2** for reset. Use three LEDs LEDR0, LEDR1, LEDR2 to turn on when the FIFO is full, not empty, and empty respectively.

For Credit

You will be graded mostly on the completeness of your testbenches, do not just give a simple testbench with no assertion check. You will also be graded on your design working on the DE10 Standard board.

Some Other Things you Should Learn

Assertion is a beautiful things back in the days that we don't have Verilator (Cpp testbenches for the win). You can read more from: SystemVerilog for Verification A Guide to Learning the Testbench Language Features, Third Edition.

Don't misunderstand, we still use SV for some verification tasks that only it can figure out the errors.

https://link.springer.com/book/10.1007/978-1-4614-0715-7

How To Turn In Your Solution

This semester we will be using BKeL, simply submit the zip file with your reports and codes.

Demos and Late Penalty

We will have demo times outside of class times on or near the due date. Since we will demo from the files in your zip, it is possible that you'll demo on a following day.

Define Late: Lateness is determined by the file dates of your submission on BKeL.