Post-Synthesis Validation

ECE 551: Digital System Design and Synthesis
Spring 2012, Hoffman

Overview

You have already finished your behavior or RTL level Verilog design, and synthesized it using Synopsys mapping it to the TSMC **tcbn40lpbwptc** technology library. It is time to simulate your synthesized design and compare the result with your original design. This tutorial demonstrates how to do post-synthesis simulation in ModelSim with your synthesized design.

Extract Synthesized Netlist

Assuming your synthesis was done with a script based method in dc_shell. Then the final line of your synthesis script should be to write out the gate level verilog netlist.

```
write –format verilog <top_module> -output <top_module>.vg.
```

For instance if your top level module was **uart** this would be:

```
write –format verilog uart –output uart.vg
```

Post-Synthesis Functional Simulation

You should put the post synthesis verilog file into a new directory and create a new project for your simulation. You can copy the original testbench to use.

Timescale

TSMC cells use *`timescale 1ns/1ps* compiler directive. Therefore, it is a good design practice to add the *`timescale 1ns/1ps* compiler directive in **both** your synthesized netlist (uart.vg in our example) and testbench (uart_tb.v in our example). Typically this is done as the first line in the Verilog file.

```
---------------------
`timescale 1ns/1ps
uart_tb();
   .
   .
   .
---------------------
```

Testbench

Besides modifying *`timescale* compiler directive, your original testbench might need more fixes if you use #(delay) statements or have reset signals.

- **#(delay)**
  You should set the clock period to match your target frequency. Before synthesis, the clock period might be set to 10ns for simplicity. However we are targeting 500MHz for the project so this would be a 2ns clock. The testbench of your original design might contain statements like

```
initial begin
        a=0;
    #11 a=1;
    ....
end
```

where signal *a* should change in the second clock cycle, but for the synthesized design, the signal will change in the sixth clock cycle. So check your #(delay) statements. For changing this, it is useful to define a **parameter clk_period**, and have all delays calculated based on that parameter, **or** have all delays specified using **@(posedge clk)** instead of an explicit timed delay.

- **RESET**

If you use only positive clock edge in your design, you might think that you can reset all the flip-flops as long as the reset pulse covers a positive clock edge. This is not always true, because in LSI Logic cells, both positive and negative clock edge timing constrains are given. Your RESET pulse should cover both a positive and a negative clock edge to properly reset your circuit. Since we are using many asynch reset flops, it is good practice to deassert reset on the negative edge of clock. Below is an example of an initial block that works well.

```
-------------------------
initial begin
  clk = 0;              // clock must be initialized
  rst_n = 0;            // assert reset
    …                   // initialize other inputs to your DUT
  @(posedge clk);       // wait for a positive edge of clock
  @(negedge clk);       // wait for negative edge of clock
  rst_n = 1;            // deassert reset
    .
    .
    .
-------------------------
```

- **Initial Assignment to all inputs of DUT from testbench**

Sometimes if an input from the testbench is X it can propagate in an unexpected manner through the gates. Ensure all input to your DUT from the testbench have a logical 0 or 1 assignment prior to reset deasserting.

Cell Library (very important)

The synthesized netlist you wish to simulate is made up of a bunch of library cells from the TSMC library. How does ModelSim know the function of a NR3D0BWP gate? We need to reference a pre-compiled library of all the cells in the TSMC library (tcbn40lpbwptc) that our synthesis run targeted.

It is best if you make a local copy of the library so you have full linux file permissions on the compiled libaray you will reference. You can copy the library from the linux user ece551.

Copy the TSMC_lib somewhere in your home directory structure. Perhaps at your <root>/ece551 level

```
unix_prompt>cd  ~/ece551
unix_prompt>cp  –r  ~ece551/TSMC_lib* .
```

Note: this copy of TSMC_lib can take a while because there is a lot of stuff in it, but you only have to do this one time.

Search Library (very important)
When you launch simulation in ModelSim (Simulate ➔ Start Simulation). You have to add the precompiled TSMC_lib library you just copied to the seach path. The Simulate ➔ Start Simulation form has many tabs across the top. One of them is the "Libraries Tab".

select **Simulate**-> **Simulate** and select **Library** tab
click **Add** under the Seach Libraries section
Now click **Browse** and a "Select Library Browser" form comes up.
In this form type in:
~/ece551 in the Library path search string at the bottom of the form, and hit enter.
Now select the TSMC_lib library you copied.

Disable Timing (under the Verilog Tab of Start Simulation) (also very important)
Synopsys is the final arbiter of setup and hold time violations, not the timing checks in the library files. We want to disable the timing checks in the library files so they don't cause problems.

Under the "Verilog" tab of Start Simulation, **check the box that says:**
"Disable timing checks in specify blocks(+notimingchecks)"

Select **Design** tab and select your module (testbench) to simulate.
Change the **Resolution** dropdown box to ns.
Remember to **uncheck** the **Enable optimization** check box
Click **OK.**

The design should now load using cell modules out of the TSMC library. You may receive many timescale directive warnings, but these can be ignored. You may also receive warnings like: " missing connections for port Q". The TSMC library contains flops with both Q and QN outputs, and sometimes Synopsys only uses one of the connections, so these are expected warnings.

Summary

1. **Extract synthesized netlist**
2. **Make necessary changes**
   a. **Timescale**
   b. **Testbench**
   c. **Copy the TSMC_lib from ~ece551 account**
   d. **Search library ➔ Reference the TSMC_lib you copied**
   e. **Disable Timing Checks**
3. **Simulate Design**

If your design does not work with post-synthesis simulation:

1. Are there latches in the design?
   a. Run "**report_register –level_sensitive**" inside design_vision. If this report has no results, you have no latches (good).

2. Is every flip flop being reset?
    a. Start looking at main important registers such as a state register or counters used. All flops that are involved in control functions will need a asynch reset from **rst_n**. If not X's will propagate.
    b. If something is not being reset, try using **force…release** statements inside your testbench to force the register to a known value so you can see if other parts work
3. Does your testbench change input signals concurrent with the clock edge?
    a. You might be creating a race. Using @(posedge clk); statements to wait to change stimulus is fine because the flops have already been evaluated by the time the change occurs. However if you are using delays `(#clock_period  strt_tx = 1;)` you might be changing input stimulus right at the clock edge and creating race conditions. Remember flops need hold time, so input stimulus should transition slightly after clock edges.
4. Did you assign a value to all input stimulus in your testbench at time zero? Remember your stimulus signals in your testbench are of type reg, and therefore assigned X at time zero. Behavioral simulation is more forgiving about how X's propagate. In post synthesis simulation X's propagate in nasty ways. Assign all input stimulus to a known value.
5. Are there other warnings from synthesis about unused connections, wire connected to GROUND/VCC?
    a. These are an indication that your design may not be connected as you thought
6. Is the clock speed of your testbench faster than the clock speed your design could achieve?