Politecnico di Torino

Collegio ETF - ICM

Low-Power Electronic Systems
Lab 6

Mariagrazia Graziano, Fabrizio Riente, Marco Vacca

March 5, 2021

System Verilog for Verification

In this chapter, you will experiment with some SystemVerilog concepts to debug your design more effectively. Nowadays, hardware verification takes more time than the design implementation. Verification is a process of comparing the behavior of the actual design with respect to the design expected behavior. Every project starts with a design specification, which contains all the required information on the design construction and its intent. The verification process parallels the design creation process. Thus, two different engineers go through the same design specifications. One is a design engineer who creates its own implementation plan to implement the design using HDL language. The other, the verification engineer who implements the verification plan that contains a description of what features need to be exercised and the techniques to be used as well (Fig. 6.1). Consequently, the purpose of

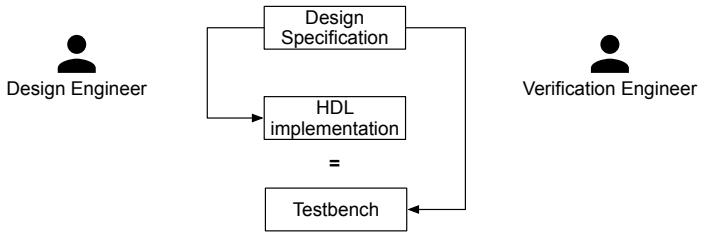

Figure 6.1:

the testbench is to determine the correctness of the design under test, usually named DUT. This can be achieved by generating stimulus and applying it to the DUT, then looking for the response of the DUT to find if the design matches the specifications.
From directory /home/repository/lowpower/ese6/ copy all files:

cp /home/repository/lowpower/ese6/* .

6.1 Simulate and Debug an Full Adder

6.1.1 Simulate a System Verilog Model

In this section, you will become familiar with simple System Verilog models, basic System Verilog-style testbenches and how to run System Verilog simulations. There is one RTL description and two gate-level descriptions of the adder contained respectively within the files *rtl_adder.sv*, *gate_adder.sv* and *gate2_adder.sv*.
There is a *run_*.f* file for each simulation:

run_rtl.f, run_gate.f, run_gate2.f

The *.f file lists the System Verilog files to be compiled for that simulation. If you open one of these files, (e.g. *run_rtl.f*), you can observe that the *+define+* is used to set the appropriate macro name. The macro allows us to select between different implementation and will be described later.

The testbench for the 1bit adder is represented by the *adder_test.sv* file. Here, two procedural blocks are present, enclosed between the *initial* keyword. The first block generates the input stimuli, while the second is used for reporting the output of the simulation. The *$monitor* statement makes it possible to displays the values of its parameters EVERY time ANY of its parameters changes value.

Usually, there is always a top level module that instantiates your DUT, the testbench, and connects them together. In this case, the top level module is represented by the *top.sv* file. The top level file uses conditional compilation to control which version of the adder is instantiated. You can create macros using the *'define* keyword, similarly to the #define in C. However, instead of commenting/uncommenting the *'define* lines, you can use the +define+ option on the compiler command line. This last approach is the one we are going to use in this lab. You can find the appropriate macro defined within the three *run_*.f* files.
Let's now try to run the simulation. We use Questa from Mentor. As the first step, initialize the environment and start the simulator running:

source /software/scripts/init_questa10.7c
vsim

Compile the first model (RTL) by typing the command:

```
vlog -f run_rtl.f
```

The *vlog* command compiles Verilog source code and SystemVerilog extensions. Optimize and simulate the compiled design using these commands:

```
vopt +acc top -o top_opt
vsim -c -do "run -all" top_opt
```

The simulation should produce and write the following results on your screen:

```
At    0.00 ns:   a=0  b=0  ci=0  sum=0  co=0
At   10.00 ns:   a=1  b=0  ci=0  sum=1  co=0
At   20.00 ns:   a=1  b=1  ci=0  sum=0  co=1
At   30.00 ns:   a=1  b=1  ci=1  sum=1  co=1
** Note: $stop    : adder_test.sv(19)
   Time: 40 ns  Iteration: 0  Instance: /top/test
```

The *+acc* switch that you use with the *vopt* command will allow you to look at internal signals if you wish to. The *-all* switch causes the simulator to run the current simulation forever, or until it hits a breakpoint or specified break event.

If you would like to view the different waveforms generated in the three cases, there are three *run_*.do* files, which compile, simulate and open the wave window for each simulation. As an example, the *.do* file for simulating the RTL description includes the following commands:

```
if ![file exists work] {
    vlib work
}
vlog -f run_rtl.f
vsim -voptargs=+acc top
do wave_rtl.do
run 100
run -continue
```

The wave window will then contain the simulation waveforms. The *-continue* continues the last simulation run after a run step. There are three *.do* files. You can run each of them without exiting the GUI, just type the following command after each run.

quit -sim

Now, simulate the gate-level version of the adder using the **.f* file. Do you see something different on the outputs? Why does the output change at different times from the RTL model? Try to look at the code of the gate-level implementation to understand why.

Now, simulate the second gate-level version of the adder. Compile the files: *top.sv*, *adder_test.sv*, *gate2_adder.sv* and the optimize and simulate by the typing the commands:

```
vlog -f run_gate2.f
vopt +acc top -o top_opt
vsim -c -do "run -all" top_opt
```

Is the output correct? This adder is similar to the previously simulated gate-level version, but has two subtle typographical errors. What are the errors? Correct them and verify your fix.

Remark point

Fixed System Verilog gate-level netlist, output of the three simulations and comment why the gate-level output differs from the RTL model.

6.1.2 Simulate a VHDL Model with System Verilog

The purpose of this section is to familiarize you with creating mixed VHDL and System Verilog simulations. We will focus on instantiating VHDL within System Verilog. Therefore, we keep the previously *top.sv adder_test.sv* files. Modify adding the following *elsif* statement:

```
‘elsif GATE_VH_ADDER
    gate_vh_adder  dut (.a(a), .b(b), .c_in(ci), .sum(sum), .c_out(co));  // design instance
```

Save the modified file and compile the VHDL adder and the System Verilog files running the following commands.

```
vlib work
vcom gate_vh_adder.vhd
vlog +define+GATE_VH_ADDER top.sv adder_test.sv
vopt +acc top -o top_opt
vsim -c -do "run -all" top_opt
```

Look at the output. Is it correct? What are the errors? Correct them and verify your fix. Once fixed, the output that you get should be similar to the one you obtained in sec. 6.1.1.

6.2 Automatically Verify your N-bit Adder

In this section, you will become familiar with System Verilog tasks to automatically generate random stimuli and verify the generated output. The aim is that you verify a generic N bit adder described by your self. Take the *gate_vh_adder.vhd* you have previously fixed, the *top_n.sv* and the *adder_n_test.sv* files.
Let's consider first the testbench (*adder_n_test.sv*), here you can find a more articulate *initial* block. Here, inputs are initialized, then we wait for two clock cycles before applying the inputs to our DUT. At every rising edge of the clock signal, the inputs are applied to the DUT, while on the falling edge the output of the adder is checked. This is repeated 100 times.

```
repeat (100) begin
   @(posedge test_clk);
     begin
       @(negedge test_clk) check_results();  // call output verification task
   randomize_inputs();
     end
end
```

One routine is in charge of checking the results and generate random inputs. The main difference between tasks and functions is that the former can have time-controlled statements, while the latter cannot. In addition, a task is like a procedure that creates reusable functionality. We have error counter variable to report the amount of wrong output detected during the simulation.

Open *top_n.sv*. Compared to the top level module we used for the 1bit adder, here there a procedural block for generating the clock signal. The clock period is set to 10 ns. Why if we are simulating a combinational circuit? It is required only to set the time to the testbench and generate new input stimuli. At the beginning of the module, the *parameter* keyword is used to define the parallelism of the adder (similarly to *generic* in VHDL). Now, describe in VHDL a generic ripple carry adder with entity name *rca_vh_adder* that uses as a component the *gate_vh_adder.vhd* you have previously fixed. Before running the simulation, instantiate your ripple carry adder after the *'ifdef RCA_VH_ADDER* macro. Call your instance *dut* as previously done in section 6.1.2. Look at the testbench instantiation how the port map and the parameter is set. Once you have completed your generic adder and you have added its instantiation to the *top_n.sv* you are ready for the simulation. Run the following commands to verify your design.

```
vlib work
vcom gate_vh_adder.vhd
vcom rca_vh_adder.vhd
vlog +define+RCA_VH_ADDER top_n.sv adder_n_test.sv
vopt +acc top_n -o top_n_opt
vsim -c -do "run -all" top_n_opt
```

Remark point

How to automatically verify the output of the DUT using the testbench.