Discussion 5: RTL Simulation with ModelSim

© 2019 Oscar Castañeda (oc66@cornell.edu) and Christoph Studer (studer@cornell.edu)

- You will learn how to simulate an RTL Verilog model using ModelSim
- You will generate input-stimuli/expected-output files to verify the functionality of an RTL model
- You will find and fix annoying bugs!

In this tutorial, you will learn the use of ModelSim to simulate a digital VLSI design that has been described at the register-transfer-level (RTL) using a hardware description language (HDL). RTL means that the digital circuit is described as a series of data-transformations across registers: You take signals coming from registers (e.g., flip-flops), transform the signals using combinational logic, and then store the results in registers. HDLs, such as Verilog and VHDL, enable you to create such RTL descriptions in an "easy" manner, so that you can quickly prototype a digital VLSI design and simulate it. This same RTL code can later be used to synthesize an actual circuit (in terms of logic gates, such as INVs, NANDs, NORs, flip-flops, full-adders, etc.) that implements the described functionality. Finally, a circuit layout for the synthesized circuit can be created automatically using tools that place and route logic gates (e.g., standard cells) and other components (e.g., memories or input-output pads). An RTL description using HDLs comes in very handy to simplify digital VLSI design. This tutorial will guide you through a simple Verilog HDL example that implements a multi-bit adder. We then simulate the described circuit using ModelSim in order to test its correct operation.

Student Activity 1: Downloading the Files

Download the `ece5746-modelsim.zip` file from the ECE 5746 Canvas and unzip it in a directory of your preference in your computer.

1 Describing a Digital Circuit using Verilog

Go to the `src` directory. Here, you will find three Verilog (with suffix `.v`) files. Each Verilog file describes a different (sequential or combinational) block, which we detail next.

1.1 Adder (`Add.v`)

We describe a simple adder that has several inputs (`A_DI`, `B_DI`, and `Cin_DI`), and outputs (`Sum_DO` as well as `Cout_DO`). Some inputs and outputs are only one bit, while others are multi-bit, with a word-length (aka. bitwidth) given by the parameter `DATA_WIDTH`. Note how simple the addition is: The only command we used is "+." You do not even have to think about which kind of adder (ripple carry adder, tree adder, carry skip adder, etc.) you want to use! Clearly, this Verilog module implements a two-operand addition, where the inputs each have `DATA_WIDTH` bits, including carry-in and carry-out signals of the adder.

Although not required for correct operation, all the signal names have a suffix `_D`, which indicates that they are data signals (as opposed to control signals). Also note that the inputs end with `I`, while the outputs end with `O`. This naming convention is particularly useful when (i) connecting different circuit modules and (ii) designing larger VLSI designs.

1.2 Flip-Flop (`FF.v`)

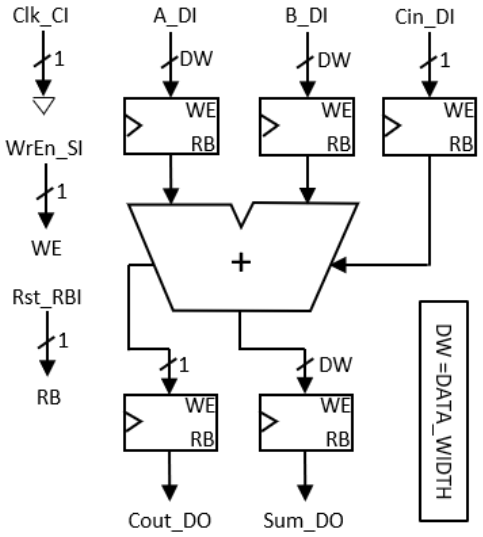
This module implements a flip-flop with write enable and asynchronous active-low reset. The flip-flop stores a number of bits determined by the `DATA_WIDTH` parameter. This module is a sequential element, which requires us to use the `reg` keyword for the signal `Q_DO` and is described *behaviorally* inside a procedural `always` block. Note that the sensitivity list of the `always` block contains not only the clock signal `Clk_CI`, but also the reset signal `Rst_RBI`; this, in combination with having the `Rst_RBI` as the first signal being evaluated inside the `always` block, is required to describe an asynchronous reset.

We recommend the use of behavioral HDL code and the `reg` keyword only for simple sequential elements (such as flip-flops). Behavioral descriptions in Verilog are dangerously resemblant of conventional software coding and may imply sequential thinking. However, it is important to always keep in mind that you are describing *hardware*, where most operations actually happen in parallel. While behavioral descriptions might work perfectly in an RTL simulation, they might get synthesized into circuits that do not have the intended functionality or cannot be synthesized at all! Both of these aspects cause major problems. As a result, limit the use of `always` blocks to only your flip-flops and other memory elements, as used in this tutorial (unless you are an experienced Verilog programmer!). Put simply, *always separate your combinational data path from sequential storage cells when writing HDL code*.

Note that in this module, there are signal names ending in `_C`, `_S`, and `_RB`. The suffix `_C` indicates that the signal is a *clock* signal and `_S` indicates that it is a control (*state*) signal. The suffix `R` in `_RB` is used to indicate that the signal is used as a *reset*, while `B` indicates "bar" which means the signal is active low (i.e., the flip-flops are reset if this signal is low).

1.3 Registered Adder (`RegisteredAdd.v`)

This is our top-level module for this tutorial. As you can see from the file, it is just using flip-flops (instances of the `FF.v` module) to store the inputs and outputs of the adder (an instance of `Add.v`). Note that some flip-flops store only 1 bit, while others have a word length given by the `DATA_WIDTH` parameter. This module describes your circuit in an *structural* manner: It is just interconnecting different modules using `wires`, giving you the same information that a block diagram or schematic would. As a matter of fact, this module is just a textual representation of the block diagram shown to the right. This is a good example to illustrate the importance of good block diagrams. Also, note that your top-level modules should always be described in such a structural manner.



2 Creating a File-Based Testbench with Verilog and MATLAB

Go into the `tb` folder. Here you will find only one file, `RegisteredAdd_TB.v`, which, as the name indicates, is the testbench (TB) for our `RegisteredAdd.v` module. Let us take a more detailed look at this file:

2.1 A File-Based Verilog Testbench

At the very top of the testbench code, you will see a set of *local parameters* (`localparam`):

- `CLK_PERIOD`: This defines the clock period that you will use to simulate your RTL code. **Make sure to always include a number after the decimal point to avoid running into issues later!**
- `IN_DELAY`: This defines the amount of time after the clock's rising-edge at which you will apply the input signals to your device-under-test (DUT). You need this time to be sufficiently large so that you do not violate the hold conditions of the current clock signal's rising-edge at the input flip-flops, but it also needs to happen early enough in the clock cycle to meet the setup condition of the next clock signal's rising-edge. A recommended *input delay* for **most** scenarios is 20% of the clock period.
- `OUT_DELAY`: This defines the amount of time after the clock signal's rising-edge at which you will sample (read out) the outputs of your circuit. This time should be long enough to exceed the propagation delay of the circuitry that is between the output flip-flops and pads of your circuit. In our example, this parameter only needs to be larger than the propagation delay of our flip-flops (as we do not have any pads nor any combinational circuits between the flip-flops and outputs of the circuit). A recommended *output delay* for **most** scenarios is 80% of the clock period.
- `DATA_WIDTH`: This is the value that will be assigned to the `DATA_WIDTH` parameter in the module described in `Registered_Add.v`. We will set this value to 16 to simulate a 16-bit adder.

Note that in an RTL simulation, we actually have no timing information for our circuit (and therefore, many of the previous parameters are not relevant). However, you could use the same testbench to simulate a post-synthesis or post-layout netlist, which are annotated with timing information (meaning they behave like real circuits with actual propagation and contamination delays).

After the parameters section, the testbench proceeds to instantiate the DUT with the correct parameters. Then, a section follows in which the clock signal is generated. Note that the clock signal will be always running. Finally, we have the two most important sections of our Verilog testbench:

- Stimuli application: This process occurs with a delay (with respect to the clock signal's rising-edge) given by the input delay parameter, `IN_DELAY`. In every clock cycle, the testbench will read one line from the `RegisteredAdd_in.txt` file and assign the values contained in this line to the inputs of our circuit.
- Expected-responses comparison: This process occurs with a delay (with respect to the clock signal's rising-edge) given by the output delay parameter, `OUT_DELAY`. In every clock cycle, the testbench will read one line from a `RegisteredAdd_out.txt` file and compare the values contained in this file with the signals coming from your circuit's outputs. In case the outputs and the expected responses do not match, the testbench will report the time and signal in which the error occurred, indicating the expected output (acquired from the file, and identified by an `E` at the end of the signal name) and the actual value (acquired from the RTL simulation). If no errors occur, then the testbench will print a corresponding message.

In short, the Verilog testbench is reading, in every clock cycle, one line out of an input-stimuli file, applying it to the DUT (your circuit) after an input delay, sampling the output after an output delay, and then comparing the sampled values with the ones read from a line of the expected-output file. Note that in these two files, each line represents a different clock cycle, and that the same line number in both files correspond to the same clock cycle. The question is now: where do we get these input-stimuli/expected-output files from?

2.2 Generating Input-Stimuli/Expected-Output Files with MATLAB

Go to the `matlab` folder. Here you will find a simple script, called `TestAdd.m`, which generates the files needed to run our testbench. In short, this MATLAB script will, for a given number of test trials, (i) generate inputs to your circuit, and (ii) use a so-called "Golden Model" to compute the expected outputs. With this information, the script can add a new line to each one of the two files, both corresponding to the same test. For this tutorial, the input generation consists of creating three numbers (two of them with 16 bits, and one with one bit): First, some directed (hand-coded) cases are generated, and the rest are randomly generated. For this case, the Golden Model is nothing else than the standard MATLAB addition `+` operation, as you expect your circuit to implement a simple addition. You should carefully go through the `TestAdd.m` file to see the details of this script.

Student Activity 2: Creating the Testbench Files

On your computer, execute the `TestAdd.m` script.

Now, go back to the `tb` folder. You will notice that there are two new files: `RegisteredAdd_in.txt` and `RegisteredAdd_out.txt`. Both of these were generated by the MATLAB script that you just executed. Take a look at both files. Note that both files have 10 lines, which means that the testbench will run for 10 clock cycles. Also note that, for the input-stimuli file (`RegisteredAdd_in.txt`), the values in each column correspond to the signals `Rst_RBI`, `WrEn_SI`, `A_DI`, `B_DI`, and `Cin_DI` of `RegisteredAdd.v`, respectively. This is the order in which the MATLAB script printed each line (see `matlab/TestAdd.m`), and also the way in which the testbench file will read them (see `tb/RegisteredAdd_TB.v`). Similarly, in the expected-output file (`RegisteredAdd_out.txt`), each column corresponds to the expected values for the signals `Sum_DO` and `Cout_DO`, respectively.

Now, have a look at the first three clock cycles in the input-stimuli file. You can see that only the `Rst_RBI` signal is changing from high to low to high: This is used to properly reset your circuit. Finally, look again at the expected-output file. You can see that there are some `x` values. These `x` entries are used to denote that you do not know what to expect from the circuit for a specific cycle and output. **Note that you can only use the `x` entries in the expected-output file.** If you do not care about the input of your circuit, then just assign any defined value (e.g., 0) to it in the input-stimuli file. We cannot do such thing with the outputs, as there might be nodes in your circuit that have not been initialized and for which we do not know their actual value—this is exactly why we properly reset our circuit: to ensure all notes are set to a known value!

3 Setting up to the vip-brg server

Now that we have a Verilog RTL design and a testbench with the corresponding files, we are ready to simulate our circuit using ModelSim. ModelSim (as well at the other tools that we will be using during this class) are installed in the vip-brg server, to which we will first connect now.

Student Activity 3: Connecting to the server

Install a terminal with `ssh` and `sftp` capabilities. For Windows, we recommend MobaXTerm, which you can download from: `https://mobaxterm.mobatek.net/download.html`.

Then, open a terminal. Log in to the vip-brg server using the following command:

```
ssh -X <netID>@vip-brg.ece.cornell.edu
```

Here, replace `<netID>` with your Cornell NetID. Use your Cornell password to log in. **If you**

are using Mac OS, then use `-Y` instead of `-X`. More details how to work with Apple computers are given below. After logging in, your terminal prompt should read `<netID>@vip-brg`. Once you are in the vip-brg server, use the following command to create a *directory* to store the files for this tutorial:

```
mkdir tutorial
```

Change into the newly created `tutorial` directory by using this command:

```
cd tutorial
```

Now, create a new directory called `modelsim`. Also copy the `src` and `tb` directories from your computer into the `tutorial` directory in vip-brg. If you are using MobaXTerm, you can do this by using the left panel to simply drag-and-drop the files into the right location.

If you are not using MobaXTerm, e.g., **if you have an Apple computer**, then you can use `sftp`. To copy the files to the `vip-brg` server, go to the `tutorial` folder on your own machine (*not* on the `vip-brg` server). In that folder, type

```
sftp <netID>@vip-brg.ece.cornell.edu
```

and use your Cornell password. You should see the prompt "`sftp>`". Type

```
mkdir tutorial
```

to create a tutorial folder on the `vip-brg` server. Enter this folder by typing

```
cd tutorial
```

By typing `ls`, you should see the contents of that folder (empty!). If you type `lls` you will see the folder contents of your *local machine* (not empty; containing the MATLAB code, Verilog code, and testbench). You can now copy the files of the `tutorial` from your own (local) computer to the `vip-brg` server by typing

```
put -r *
```

which takes all the files and folders of your local directory and recursively copies (puts) it onto the target folder on the `vip-brg` server. If you now type `ls` you should see the three folders, which have been copied onto the server. Type `exit` to quit `sftp`. As described above, you can now use `ssh` to log onto the server by typing

```
ssh -Y <netID>@vip-brg.ece.cornell.edu
```

Do not forget to add a `modelsim` folder to the `tutorial` folder as described above.

Note that if you are trying to connect via `ssh` from home, you need to use a Cisco AnyConnect client. The Cornell IT website describes how to do that.

4 Simulating your Verilog code with ModelSim

4.1 Compiling RTL for ModelSim Simulation

Before we can run a simulation using ModelSim, we have to compile our source files for efficient simulation. To do so, first we need to create a library in which the compiled sources will be stored.

Student Activity 4: Compiling your Verilog Code for Simulation

In the terminal, go to the `modelsim` folder. Load the ModelSim program with the command:

```
module load mentor-modelsim-de-10.7a
```

To avoid having to load ModelSim every time that you log into the vip-brg server, you can add the previous command to your `.bashrc` file that gets executed every time that you log in:

```
echo 'module load mentor-modelsim-de-10.7a' >> ~/.bashrc
```

Here, `~` represents your home directory. Ask us in case you want to know more details about the previous command.

Within the `modelsim` library, create a ModelSim library named `work` by entering in the terminal:

```
vlib work
```

Now, compile the `FF.v` module into the `work` library using the following command:

```
vlog -work work ../src/FF.v
```

Make sure that the module compiled and that no warnings nor errors were reported. Use the `vlog` command three times more to also compile the other circuit modules, including the testbench module. Do not forget these steps!

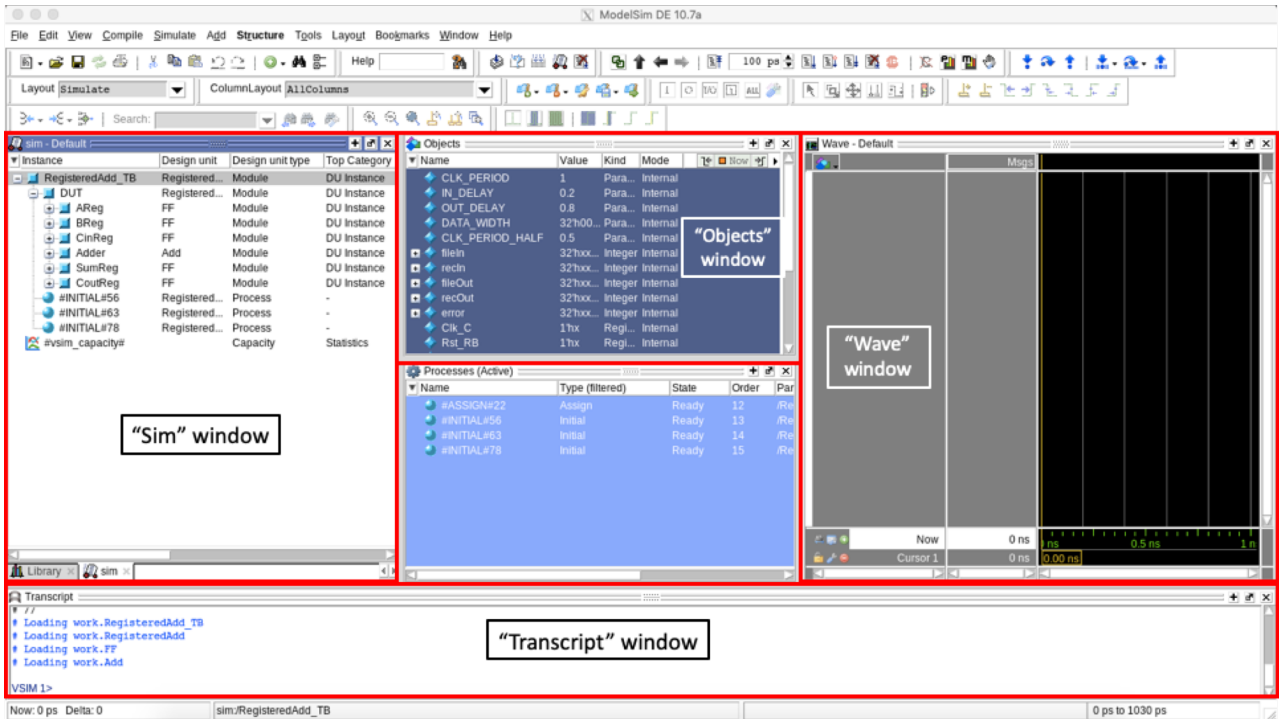
Note: To compile System Verilog for Modelsim, add the flag `-sv` *to the* `vlog` *command. If you ever want to compile VHDL for ModelSim, you should use the* `vcom` *command instead.*

4.2 Using ModelSim

Now that all your modules have been compiled, start ModelSim with the following command:

```
vsim -lib work -voptargs="+acc" RegisteredAdd_TB
```

This command is telling ModelSim to simulate the module `RegisteredAdd_TB` (which corresponds to the testbench) from the `work` library. The `-voptargs="+acc"` part of the command will enable us to see all the inner nodes of all modules. After ModelSim launches, you will see a window like the one shown below; Make sure that your design was loaded correctly by reading the contents of the "Transcript" window.



The most important windows in ModelSim are the following:

- "Transcript" window: This is basically ModelSim's terminal, where you can enter commands or read out messages.

- "Sim" window: In here, you can find the design that you are simulating, and navigate through its hierarchy.
- "Objects" window: Shows the quantities (signal values, parameters, etc.) associated with the module selected in the "Sim" window.
- "Wave" window: Here you can see the waveform from different objects (e.g., signals) of your design. To add an element to the "Wave" window, drag-and-drop it from the "Objects" window.

Before we continue, we will add all the inputs and outputs of our circuit to the wave window. Go ahead and add all the input/output signals from the top-level module to the "Wave" window. Then, close ModelSim and open it again with the same command as before. What happened to the waveforms that you have added before? As you can see, the "Wave" window got completely reset. If you have to set up the waveforms layout every single time that you open and run ModelSim, things quickly become extremely frustrating. You will now learn how to avoid this problem.

Student Activity 5: Saving the "Wave" Window State

1. Once again, add the desired signals to the "Wave" window. Include the expected output signals (which are available in the `RegisteredAdd_TB` module and end with `_DE`).
2. Click on the wave window: Make sure that the title bar of this window becomes blue.
3. Press **"Ctrl+S"** in your keyboard. A "Save Format" window will show up.
4. Save the waveform format in the default path with the default `wave.do` file name.
5. Close ModelSim and open it again.
6. In the "Transcript" window, enter the following command:

   ```
   do wave.do
   ```

   where `wave.do` corresponds to the name of the file we saved before. Note how this recovers the "Wave" window just as it was when you saved the `wave.do` file.

Now, we are finally ready to simulate our RTL design. For this, you should use the `run` command, which should be followed by a number to indicate for how long should the simulation be executed. For example, if you type `run 500ps` in the "Transcript" window, then the circuit will be simulated for 500ps. Alternatively, you could enter `run -all` (or even `run -a`) in the "Transcript" window to simulate until a finishing point. For our case, the finishing point is just after all the lines of the expected-output file have been read.

Student Activity 6: Debugging your Design, Part I

Run your simulation until completion, by entering in ModelSim:

```
run -a
```

You will notice that your circuit has some errors. Using the error messages and by analyzing the waveforms, identify the error and fix it. If fixing the error requires you to change Verilog code (either the circuit modules or the testbench file), you will need to compile your RTL code again. Make sure to restart your simulation in ModelSim (by entering `restart -f`) to load the new compiled source into the simulation. In case you change the input-stimuli/expected-output

files, there is no need to recompile your RTL code. You only need to upload the new test files from your computer to the vip-brg server, and to restart the simulation (with `restart -f`) to reset the simulation back to time zero.

After fixing the bug, everything seems to be going great! You simulated a 16-bit adder in ModelSim and it passed all of your tests. Hence, you may think about closing your computer already. However, there is a tiny problem: You only tested five cases. Do you really think this is enough to be sure that your circuit is behaving as you expect it to?

Student Activity 7: Debugging your Design, Part II

Go back to the MATLAB script, and this time modify the variable `trials` to a value of 100. Run your simulation with ModelSim again. Is your circuit still passing all tests?

Is the circuit working as expected? If not, find the new bug and fix it. Please do not tell your classmates where the bug was (that would defy the purpose of this tutorial). If you found the but, please let us know! Then, close the ModelSim window and close your connection to the vip-brg server by typing on the terminal:

```
exit
```