

Discussion 6: RTL Synthesis with Synopsys Design Compiler

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

-
- You will learn how to synthesize an RTL Verilog model using Synopsys Design Compiler
 - You will generate timing, area, and power estimates for the synthesized design
-

In this tutorial, you will learn how to use Synopsys Design Compiler (DC) to synthesize a digital circuit that has been described at the register-transfer-level (RTL) using a hardware description language (HDL). As learned in the previous tutorial, an RTL description of your hardware greatly simplifies digital VLSI design. Take, for example, the adder module, `Add.v` that we used in the previous tutorial. In order to describe an addition operation, you simply had to use the addition `+` operator, without (i) having to think the adder type you would like to use (ripple carry adder, carry skip adder, tree adder, etc.) or (ii) writing your adder in terms of logic gates (e.g., using INVs, NANDs, NORs, XORs, etc.).

However, as you have learned in this class, implementing a real VLSI design involves multiple steps. First, you need to come up with a design and describe it using a gate-level schematic. Second, you have to create standard-cells for the used logic gates. Third, you have to size the individual standard cells to optimize the circuit for timing, area, or power specifications. Fortunately, with commercial standard cell libraries, the cells have been properly designed and characterized (in terms of timing, area, and power), which enables the *synthesis tool* to automatically create a circuit that meets the desired specifications. In addition, the synthesis tool also generates area, delay, and power estimates for your final circuit. It is important to note that, after synthesis, only the standard cells to be used (and their sizing) are known. However, they have not been placed and routed. Hence, the characteristics of your final layout may vary from the ones reported by synthesis. Other computer-aided-design (CAD) tools exist to automatically resize, place, and route (connect) the standard cells. Such tools also generate more accurate estimates of timing, area, and power. This tutorial, however, will only focus on synthesis.

1 The Synthesis Process

In Synopsys DC, the synthesis procedure involves three main steps, which are described next:

- **Analysis:** In this step, your RTL HDL code is converted into an intermediate representation that is stored in a *design library*. Once you have added a module into the design library, other designs can refer to it, instantiate such module, and connect to it.
- **Elaboration:** In this step, a design from the design library is loaded into the Synopsys DC program memory. In case your design instantiates other designs, these will be brought into the memory as well. In this step, it is important to note that you can overwrite the default value of the parameters in your RTL code. In such case, the design name will change to specify the new parameter value.

If we take our Verilog adder source code as an example, you could elaborate the `RegisteredAdd.v` module with `DATA_WIDTH=16`. In this case, the design in Synopsys DC will be assigned the internal name `RegisteredAdd_DATA_WIDTH16`. If you do not specify any value for the `DATA_WIDTH` parameter, then your design in Synopsys DC will simply be named `RegisteredAdd`, and it will use the default parameter value of 12 for the `DATA_WIDTH` parameter.

- **Compilation:** In this step, the analyzed and elaborated design will be mapped to the available standard cells (with appropriate sizes to meet your constraints). During compilation, Synopsys DC starts by mapping high-level descriptions (e.g., the addition `+` operator and the behavioral description of the flip-flop `FF.v` module) into generic logic. Then, the design tool proceeds to apply Boolean logic optimization. Finally, the circuit gets mapped into the available standard cells. The synthesis tool tries to optimize your design by using the best possible available logic gates (e.g., a full-adder cell). At the end, design rules (such as fanout, capacitive load, etc.) are checked to see whether there are violations (which are then fixed). During the entire process, Synopsys DC will be trying to meet the **timing** constraints that you define. After that, the tool will perform an *area recovery* step, where it tries to reduce the circuit's area without creating additional timing violations.

2 Starting Synopsys DC

Student Activity 1: Downloading the Files

For this tutorial, you need to download the `ece5746-syn.zip` file available on Canvas. Unzip it in your local computer. Copy the `syn` directory into the `vip-brg` server and place it within the `tutorial` directory from the previous discussion. Make sure that you have placed the `syn` directory in the same level where the `src`, `tb`, `matlab`, and `modelsim` directories are. Also, make sure that the `.synopsys_dc.setup` file is within the `syn` directory. If it is not there, use the `mv` command to rename the `synopsys_dc.setup` file (note that this last file does not have a period at the start). You can do so with the following command in the `vip-brg` server:

```
mv synopsys_dc.setup .synopsys_dc.setup
```

Also, load the Synopsys DC program by executing the following command:

```
module load synopsys-dc-2016.12
```

You should add this last command to your `.bashrc` file, just like we did in the previous tutorial.

Let us now start Synopsys DC. In the `vip-brg` server, change directory to the `syn` folder that you just set up, and enter the following command:

```
dc_shell-xg-t -output_log_file dc_syn.log
```

This will start Synopsys DC, which will continuously write to a log file `dc_syn.log` that contains everything that happens while using Synopsys DC. Note that inside the `syn` folder, there is a `.synopsys_dc.setup` file. This file will always be executed by Synopsys DC at start-up. In our case, this file simply sources the `syn/scripts/syn_setup.tcl` script, which is loading the standard cell library files (so that you do not have to do this manually). For this tutorial, we use the Synopsys Armenia Educational Department (SAED) 90 nm library. Note that this is a standard-cell library designed only for educational purposes — You cannot really fabricate an integrated circuit with it. If our project is successful, we will fabricate our circuit in a 65 nm technology which you cannot access as it requires signing a non-disclosure agreement.

3 Synthesis with Synopsys DC

3.1 Analysis

Let us *analyze* the flip-flop `FF.v` module using Synopsys DC. This is accomplished with the command:

```
analyze -library work -format verilog ../src/FF.v
```

With this `analyze` command, the `-library` argument specifies the design library to which the design will be added. In this case, we are using a design library called `work`. The `-format` argument indicates the HDL being used. We use `verilog` to indicate that the source code is written in Verilog. If you are using System Verilog or VHDL, then you have to use `sverilog` or `vhdl` for the `-format` argument, respectively.

Student Activity 2: Analyzing the Rest of Your Adder Design

Use the `analyze` command to analyze the other **two** modules required for your design. **Do not analyze the testbench `tb/RegisteredAdd.TB.v` file!** The testbench is only used in simulation to check that your circuit is working properly (and that is why it gets compiled when working with ModelSim), but it is **not** intended to be part of our circuit (which is why we will not synthesize it in Synopsys DC). Also, the testbench cannot be synthesized anyway...

3.2 Elaboration

We have now finished the analysis step. It is time to move on to the *elaboration* step. Elaborate your top-module design, by entering the following command in the Synopsys DC shell:

```
elaborate RegisteredAdd -library work
```

With this command, you have elaborated the `RegisteredAdd` module from the `work` design library using the default value of 12 for the `DATA_WIDTH` parameter. You can verify this by reading carefully through the messages that Synopsys DC printed on the shell. While reading these messages, also note that there is a message that reads “Inferred memory devices in process,” after which a table appears. This table summarizes that the `FF.v` file was synthesized into a flip-flop with asynchronous reset (AR). You should *always* read these messages to verify that your memory elements got correctly inferred as the device you intended them to be; e.g., make sure that Synopsys DC infers a flip-flop (and not a latch) whenever you wanted to have a flip-flop; the same applies to asynchronous reset. We recommend always using flip-flops with asynchronous reset.

Now, you can list the available designs by typing the following command in the Synopsys DC shell:

```
list_designs
```

This command will print all the designs that are currently in the program’s memory (i.e., all the designs that have been elaborated). Note that, as expected, the `RegisteredAdd` design is there, but there are some others as well. Remember that the elaboration step also elaborates all the other designs that are used by your top-level module, and this is why we can see the `Add` and `FF` designs here as well. Note that each design (except the top-level one) is followed by the value used for their `DATA_WIDTH` parameter.

Student Activity 3: Elaborating a 16-bit Adder

With the previous `elaborate` command, you elaborated a 12-bit adder. However, we would like to synthesize a 16-bit adder. Instead of going back to the source Verilog file and changing the default value for the `DATA_WIDTH` parameter, you could also use the following command to overwrite this parameter:

```
elaborate RegisteredAdd -library work -param DATA_WIDTH=>16
```

Execute the `list_designs` command again. How is the list different now? How can you differentiate between the 12-bit and 16-bit versions of the `RegisteredAdd` design?

You may have noticed that the output of the `list_designs` command printed out an asterisk (*) next to one of the designs. This asterisk marks the currently selected design that will be synthesized once you go into the compilation step. So, before going on, let us make sure that the 16-bit `RegisteredAdd` design is the current design by typing the following command into the shell:

```
current_design RegisteredAdd_DATA.WIDTH16
```

3.3 Constraints Definition

We are almost ready to *compile* your design. We just need to define some constraints first. We will start by creating a clock object in Synopsys DC. This is done with the following command:

```
create_clock Clk_CI -period 6.0
```

With this command, we have created a clock on the port `Clk_CI` of our `RegisteredAdd` design, with a period of 6.0 ns. Note that the command does not specify the time units; these are defined by the technology library being used. For the SAED 90 nm library, the time unit is indeed ns. To make our clock more realistic, we will now define the clock signal's transition (rise and fall) times. We will set both transitions times to be 200ps with the following command:

```
set_clock_transition 0.2 [get_clocks Clk_CI]
```

The `get_clocks Clk_CI` part of the command is just recovering the clock objects that are associated with the `Clk_CI` pin.

Similarly to the input delay that we specified in ModelSim, we have to tell Synopsys DC that the changes at the input signals will be delayed with respect to the clock signal's rising edge, as to not violate the hold condition of the flip-flops that register the circuit's inputs. Note that, in practice, part of this input delay will be introduced by (contamination and propagation) delays at the circuitry generating the inputs to your circuit. Let us set an input delay of 200ps on all the input pins (except on the `Clk_CI` pin) with respect to the `Clk_CI` clock, by issuing the following command:

```
set_input_delay 0.2 -clock Clk_CI [remove_from_collection [all_inputs] [get_ports Clk_CI]]
```

Here, `[remove_from_collection [all_inputs] [get_ports Clk_CI]]` is getting a list with all the circuit inputs and removing from it the `Clk_CI` input, so we apply the constraint on all remaining inputs.

Finally, we will define a driver for our circuit's inputs and a capacitive load for its outputs (why are we doing this?) using the following commands:

```
set_driving_cell -library saed90nm.typ -lib_cell INVX4 -pin ZN [all_inputs]
set_load 0.01 [all_outputs]
```

The first command will set attributes from the output port of a cell in the standard-cell library to the input ports of your circuit, effectively considering that all of your circuit's inputs are being driven by such cell. Here, we will take the output port `ZN` of the `INVX4` component in the `saed90nm.typ` library. The second command will set a capacitive load of 0.01 pF on every output, where the capacitance unit is given by the technology library used.

3.4 Compilation

After defining all the constraints for your design, we are finally ready to compile it! In Synopsys DC, compilation is started by executing the `compile_ultra` command. However, this command has many different flags that you can use according to your design's needs. Let us take a look at some of these flags before compiling our design.

Student Activity 4: Reading the Manual of `compile_ultra`

To know and understand all the different flags available for the `compile_ultra` command, you should read its *manual*! To do so, just run the following command in the Synopsys DC shell:

```
man compile_ultra
```

In the output of this command, you will see several flags that might sound familiar (e.g., `-scan`, `-retime`, and `-gate_clock`). In this tutorial, we will only use the `-no_autoungroup` flag. Make sure you understand what this flag is doing.

Remember: You can use the `man` command to read the manual for any Synopsys DC command; this is super helpful to understand what each command and its different options do!

Now that we understand what the `-no_autoungroup` flag is doing, start your design's compilation by executing the following command in the Synopsys DC shell:

```
compile_ultra -no_autoungroup
```

As we are using a very relaxed timing constraint of 6 ns, this compilation step will not be that exciting. However, later in the tutorial you will synthesize your design for tighter timing constraints, in which case you will see how Synopsys DC will be optimizing your design to reduce the worst negative slack until it reaches a value of 0 ns. You will also note that, besides the delay optimization stage, there are stages for optimizing the area and leakage power of your design as well.

Once the compilation is finished, we can check our design's consistency with the command:

```
check_design
```

In this case, our design is so simple that there are no warnings. You should **always** run this command for every design you synthesize: You must ensure that there are no errors and that you can explain all the warnings so that they are safe to ignore.

4 Estimating a Circuit's Timing, Area, and Power with Synopsys DC

By now, your circuit has been correctly synthesized. It is now time to inspect the results you got by checking your design's timing, area, and power. Keep in mind that, up to here, your design has not been placed and routed. Hence, these numbers are just approximations as you have not connected the standard cells; Remember that wiring has a significant impact on your design's delay, area, and power.

4.1 Timing Reports

We will first start by reporting the setup checks for our design, with the following command:

```
report_timing -max_paths 5 > ./reports/timing_max.rpt
```

This will report the 5 most critical paths in your design (in terms of the setup condition) and write the report to a file named `timing_max.rpt` in the `syn/reports` directory. Go ahead and take a look at this report: You can see how Synopsys DC details each one of the most critical paths, indicating the specific cells and ports through which the path goes, the delay added by each one of these cells, and even if the transition happening at each node in the critical path is rising (r) or falling (f). Also note that each critical path is actually considering two paths: There is one path for the signal that is reaching the input port of the flip-flop, and one path for the clock signal reaching that same flip-flop. As you can see, the flip-flop's setup time is subtracted from the latter, instead of being added to the former.

The difference between the clock signal's arrival time (called `data_required_time` in the report) and the data signal's arrival time (`data_arrival_time`) is called the *slack*; i.e., $slack = data_required_time - data_arrival_time$. For your circuit to work at the specified cycle time, the clock signal's rising edge should always arrive later (or at the same time) than your data signal, which means that your slack should

be non-negative. In case you have a positive slack for the most critical path, this means that your data signal could get delayed (i.e., it could *slack*) for that amount of time, and it would still be sampled correctly at the flip-flop. Then, in principle, you could reduce the clock period by exactly the *slack* of the most critical path, and your circuit would still work. Finally, it is worth mentioning that, in these timing reports, the first path reported is always the *critical path*!

As you know from class, while we are all excited about the setup checks because they define how *fast* a circuit can work, there is another set of (often much more important) conditions that define if our circuit *will work*: The hold conditions. In Synopsys DC, you can report these hold conditions with:

```
report_timing -delay min -max_paths 5 > ./reports/timing_min.rpt
```

Note that the `-delay min` argument is what makes the report contain information about hold times. Take a look at the generated report and **be absolutely sure that all the hold conditions are met**. You will also notice that this hold-check timing report is very similar to the one for the setup checks. The only difference is that now you want your data to change (`data_arrival_time`) a hold-time after the clock signal's rising edge (`data_required_time`). Then, for hold checks we want `slack = data_arrival_time - data_required_time` (which is the negative of what is used for the setup checks) to be non-negative.

4.2 Area Report

Execute the following command in Synopsys DC:

```
report_area -hierarchy > ./reports/area.rpt
```

This will report the total area of your design, as well as an area break-down of each of the contained modules. Take a look at the generated report in `syn/report/area.rpt`. As you can see, the report lists the total combinational, non-combinational, buffer/inverter, and macro (e.g., an SRAM module) areas separately—it even includes an estimate of the interconnect area! The sum of the first four areas is called the *cell area*, while the sum of all listed area values is the *total area*. Then, the report proceeds to break down the area for each of the modules in your design (in this case, instances of `FF.v` and `Add.v`). Note that the adder is not even half of the design's area!

4.3 Power Report

Enter the following command in Synopsys DC to create a power estimate report for your circuit:

```
report_power -hierarchy > ./reports/power.rpt
```

Similar to the area report, the `-hierarchy` flag will break down the power consumption for each module in the design. Open the report in `syn/report/power.rpt`. You will see that the report starts by reporting the operating voltage, as well as the voltage, capacitance, time, and power units. Then, for each module, the dynamic and leakage power are listed. Note that these two quantities are expressed with different orders of magnitude (e.g., while the dynamic power is given in μW , the leakage power is in pW). Also note that the dynamic power is decomposed into two parts: The *internal power*, which corresponds to the cells' internal nodes, and the *switching power*, which happens at the cells' input and output nodes.

As you will learn in class, the power of a circuit depends not only on the node capacitances, supply voltage, and frequency, but also on the switching activity α . In this case, Synopsys DC is assuming a switching activity and a *static probability* (i.e., what is the probability of the signal having a logic 1 value) at your circuit's inputs. Using this information, the switching activities for the rest of the circuit are calculated to generate a power estimate. Alternatively, you could simulate your design to extract the switching activities at each node, and then use that information in Synopsys DC to get a better power estimate.

5 Generating a Verilog Gate-Level Netlist with Synopsys DC

Finally, we will get the most valuable output of the synthesis process, which is a gate-level netlist of the synthesized design. To get this netlist, run the following two commands in the Synopsys DC shell:

```
change_names -rules verilog -hierarchy
write -hierarchy -format verilog -output ./outputs/RegisteredAdd_Gate.v
```

This will create a Verilog file in `syn/outputs/RegisteredAdd_Gate.v`. Open this file. As you can see, there is a Verilog module for each different instance of your design modules, plus a top-level module that corresponds to the `Registered_Add` module (this one is located at the very end of the Verilog file). Note how all the circuits are described only in terms of standard cells, such as `DFFARX1` (D-flip-flop with asynchronous reset and a sizing factor of 1) and `FADDX1` (full-adder with sizing factor of 1). Since all the circuit's description happens at the standard cell (or gate) level, this Verilog file is called a *gate-level netlist*.

Student Activity 5: Examining the Synthesized Flip-Flop Module

In the gate-level netlist, go to the module that corresponds to a 1-bit flip-flop. Does the synthesized flip-flop implement the functionality described in the `FF.v` module? To answer this question, it might be useful to know what each standard-cell name (e.g., `FADDX1`) corresponds to, information that you can find in the standard-cell library databook. You can read the databook by running the following command in the standard terminal (i.e., not in the Synopsys DC shell; careful when copying this statement as there are spaces between the lines that should not be there):

```
evince /work/global/brg/install/adk-pkgs/saed-90nm/pkgs/SAED_EDK90nm_01252011/
SAED_EDK90nm/Digital_Standard_cell_Library/doc/databook/
SAED_Digital_Standard_Cell_Library_Rev1_11_25.01.2011.pdf &
```

You can also visualize the gate-level netlist in a graphical way. To do so, start Synopsys DC's graphical user interface (GUI) by issuing the following command in the Synopsys DC shell:

```
start_gui
```

In the Design Vision window, you will find a "Logical Hierarchy" panel on the left side of the screen. Click the top-level module to select it. Then, go to "Schematic" → "New Schematic View". This will open a new window that contains a block diagram of the top-level module. Double-click on the block to show its contents. You can double-click again to keep descending in the design hierarchy. Once you reach a standard-cell and cannot descend anymore, you can right click on the cell and go to "Properties". In the new window, search for the "ref.name" field to determine which cell this is.

Student Activity 6: Examining the Synthesized Adder with the GUI

Using Synopsys DC's GUI, show the schematic for the adder module (without the flip-flops). Can you tell what type of adder (e.g., carry-select adder, carry-skip adder, carry-save adder, etc.) this is?

We have finished RTL synthesis with Synopsys DC. Exit Synopsys DC by entering `exit` in its shell. The next step of the process will be to take your synthesized Verilog gate-level netlist and use another CAD tool that will automatically place and route the standard cells according to your gate-level netlist.

Student Activity 7: “Playing” with the Timing Constraint

Now, apply your synthesis knowledge! Repeat the synthesis process, but this time for two new clock constraints: One needs to have a clock period of 1.0 ns. For the other one, you decide! For the homework, generate an AT-plot that includes the area and delay **for at least three different clock constraints** that you synthesized during this tutorial. In the plot, for the area of each design, use only the combinational area (exclude the area of the flip-flops). For the timing, do not use the clock constraints, but the clock constraint minus the slack of the critical path in the setup check. **Do not forget to upload your results to Canvas to complete this homework!**