Memo lab 1–4 in

DAT110 Methods for Electronic System Design and Verification

Version 2022-10-24

Per Larsson-Edefors
Chalmers University of Technology

In the following four labs, you will get hands-on experience of a number of design and verification steps associated with digital ASIC design using standard-cell libraries. However, several of the highlighted design steps are generic and consequently of relevance also to, e.g., FPGA synthesis.

While the lectures of this course will address key electronic design automation (EDA) techniques and methods for design and verification, these four labs are essential in so far as they will give highly practical insights in EDA in a context of design and verification in an industrial setting.

Course website on Canvas

`https://chalmers.instructure.com/courses/20977`

Getting started in the Linux environment

We are going to use the Linux server `knuffodrag.ita.chalmers.se`. We have prepared some files for the labs and you can find them in the course directory on `knuffodrag`:

`/usr/local/cad/course/DAT110`

To run the tools, we need to define license servers and file search paths. The shell script `setup.sh` below, which assumes the bash environment, can be copied from the course directory. (In bash, the backslash (\) indicates the command continues on the new line.)

```
#!/bin/bash

export CDS_LIC_FILE=5280@cadence1.lic.chalmers.se:5280@cadence2.lic.chalmers.se:\
5280@cadence3.lic.chalmers.se

PATH=$PATH:/usr/local/cad/incisive-15.20.061/tools.lnx86/bin:/usr/local/cad/genus\
-18.14.000/bin:/usr/local/cad/innovus-19.13.000/bin
```

Run the script by entering `source setup.sh` at the command line. If you are looking for manuals for the tools, consider using `cdnshelp` which is a huge help resource on Incisive, Genus and associated tools:

```
/usr/local/cad/incisive-15.20.061/bin/cdnshelp
/usr/local/cad/genus-18.14.000/bin/cdnshelp
```

Suggestions on other tools include gedit, emacs, vi (Linux text editors), LaTeX, Open office (word processors) and Inkscape, xfig, Visio (for drawings).

It is advisable to use different directories for different assignments. You need to be careful with the file structure when using the logic simulator Ncsim of the Incisive software suite: If you compile several different VHDL files that use identical entity names, the tools may mix up the different entities.

Lab 1: Verification Using Testbench and Test Vectors

Starting from the RTL code of two different 16-bit adder implementations, you are in this lab going to develop a testbench that you will use to evaluate the logic functionality of adders.

Preparation

1. Retrieve the VHDL code for the 16-bit ripple-carry adder `RCA.vhdl` (and its supplementing file `FA.vhdl`) under the course directory in `knuffodrag`: `/usr/local/cad/course/DAT110/RCA`

2. Retrieve the VHDL code for the wrapper file `Wrapper.vhdl` which will encapsulate `RCA.vhdl` by adding registers on input and output.

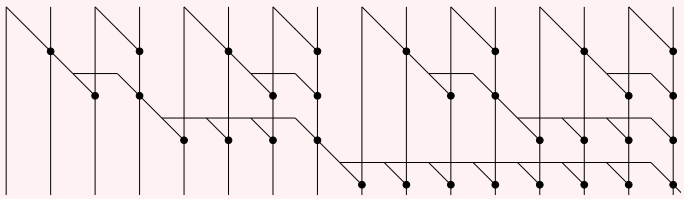
3. Draw a block diagram and a timing diagram on the hardware which is represented inside the VHDL code. **We are going to discuss these diagrams at the Zoom meeting on Friday Nov. 4, 13:30; see Canvas.**

4. Also, retrieve the VHDL code skeleton for the 16-bit Sklansky prefix tree adder (`Sklansky.vhdl`) (and its supplementing files `InitCarry.vhdl`, `Init.vhdl`, `DOT.vhdl`, `gDOT.vhdl`, and `E.vhdl`) from `knuffodrag`: `/usr/local/cad/course/DAT110/Sklansky`

5. In `Sklansky.vhdl`, we have intentionally commented some definitions out, namely

   `(--!find the correct description!)`.

   On these lines, apply your understanding of parallel prefix adders — the Sklansky adder type has been previously reviewed in MCC092 — to define the interconnection of dot-operators.



6. Based on information in Lecture 1 and 2, develop a testbench structure. **We are going to discuss your testbenches at the Zoom meeting on Friday Nov. 4, 13:30; see Canvas.**

   To enable loading of test-vector data from file, we provide in the course directory a skeleton of a testbench (`TB_skeleton.vhdl`) in which file I/O functions are predefined.

Background: The adders that we are going to verify through simulation will be placed inside two sets of positively edge-triggered registers provided in the wrapper file. The purpose of this wrapper is twofold: 1) Since the registers represent timing boundaries, the synthesis tool which we will use later can calculate the timing of the combinational logic. 2) To reach high throughput, digital blocks always operate within logic pipelines. Take for example the MIPS-I-like processor of the CREEP environment[1] which is shown in Fig. 1. Here the compute units are part of the execute stage of the pipeline. The ALU can fit inside one clock cycle, but since the integer multiplier has a long timing path, it uses two cycles to complete one multiplication.

[1] Read more in "CREEP: Chalmers RTL-based Energy Evaluation of Pipelines," Tech report, Chalmers, 2017, https://research.chalmers.se/publication/246746.

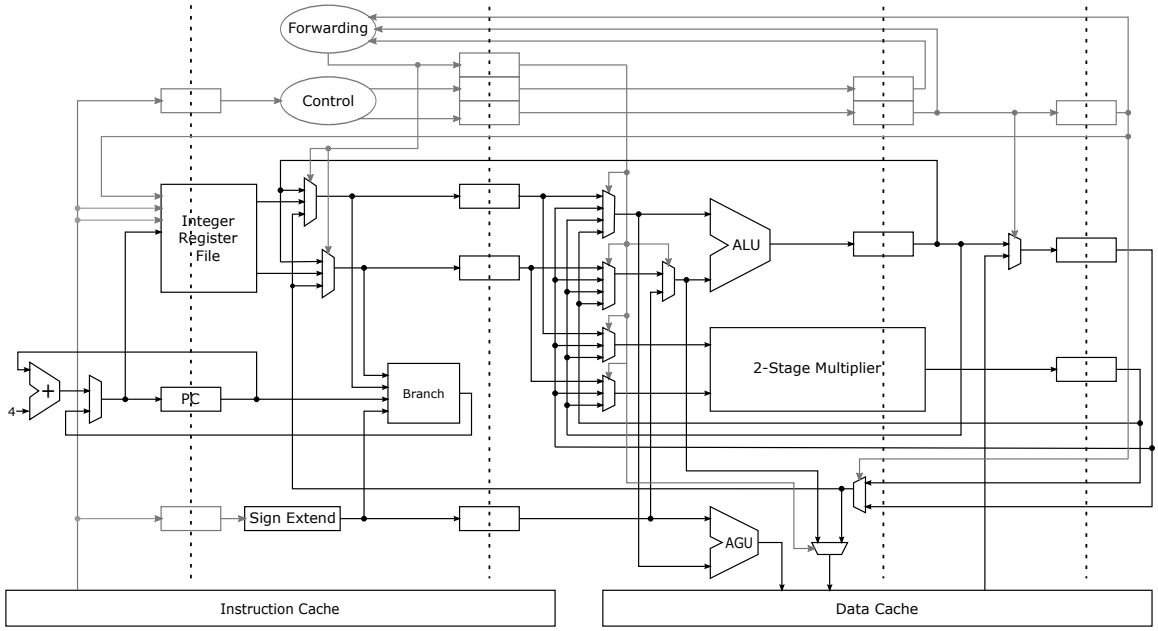


Figure 1: CREEP pipeline [from https://research.chalmers.se/publication/246746].

Logic simulation: Now, using a testbench, we have to make sure that the adder VHDL is correct and that it represents the functional behavior that we expect. We need to verify our hardware descriptions against a reference case which is constructed out of proven test vectors.

This part of the labs deals with testing and debugging using the Ncsim logic simulation tool of the Incisive Enterprise Simulator (IES); Cadence's correspondence to Siemens/Mentor's ModelSim/QuestaSim. During logic simulation, the testbench will read test vector data from different stimuli files and send these to the adder wrapper. On the output of the wrapper, the testbench will collect the produced data and compare these to the expected — the reference case — data.

Assignment

- Write a testbench for the adders, for which the wrapper file is the top file. In contrast to the structural VHDL used in the hardware implementations, a testbench is written using behavioral VHDL. This is possible since the testbench will never be synthesized. One important behavioral feature we can use in a testbench is the `wait` statement. This will be further discussed in Lectures 1–3.

Once we have the testbench, we can compile the VHDL files. In the following we assume a testbench, called `TB.vhdl`[2], which uses the `wrapper` entity inside `Wrapper.vhdl`. The latter in turn uses the `adder` entity inside `RCA.vhdl`, in which the `FA` entity of `FA.vhdl` is used. To simulate this VHDL hierarchy we first compile the VHDL blocks, including the testbench:

```
ncvhdl -64bit -v93 -messages FA.vhdl
ncvhdl -64bit -v93 -messages RCA.vhdl
ncvhdl -64bit -v93 -messages Wrapper.vhdl
ncvhdl -64bit -v93 -messages TB.vhdl
```

after which we elaborate (connect) them via the `TB` entity in the testbench

```
ncelab -64bit -v93 -messages TB
```

In the course directory, under `1000_random_TVs`, you can find three different files: `A.tv`, `B.tv` and `ExpectedOutput.tv`. These are 1,000 randomized reference test vectors that we provide for your

[2]In Linux, avoid using file names that contain spaces. I often replace a space with an underscore.

logic simulation. Note that only the 16-bit inputs `a` and `b` are defined in the vectors; the carry-in signal to the adder must be set to 0 in the testbench.

Once you have copied the above files to your current folder, you can simulate the testbench using Ncsim.

```
ncsim -64bit TB
```

Assignment

- Simulate both the ripple-carry and Sklansky adder using your testbench to ensure your designs work when simulated against the 1,000 randomized reference test vectors.

  If you identify bugs, modify either the RTL code or the testbench code to rectify the problems.

Practical hints

1. Use the `assert` function with `warning` as severity level to compare the obtained data with the expected. Never trust your eyes to spot errors in a long and complex waveform! Only use waveform viewing, which can be invoked for Ncsim with the `-gui` flag[a], when you need to dig deeper into a block to localize a bug.
2. You are using RTL code largely prepared for you, so the bugs may be few. However, if your `assert` functions discover surprisingly few (or no) errors when testing the Sklansky adder, try to use test vectors that you know are erroneous. By deliberately flipping a few test vector bits, you may expose bugs in the verification functions.
3. Make it a habit to re-run all above commands, including `ncvhdl` and `ncelab`, every time you have modified a file, even if this change is limited to one of the VHDL files.
4. In your testbench, time progresses cycle by cycle. Avoid hardcoding a parameter like the clock period, but make it a constant which you later can change.
5. For the lab report, you may want to consider the following book chapters and research articles: [1, 2, 3, 4, 5].

[a] This assumes you have added flags `-access +r` to `ncelab`

Learning outcomes of Lab 1

When you have completed this part of the labs, also assuming that you have attended the supporting lectures, you should be able to ...

- verify and debug digital logic units from the command line, using file-driven testbenches.
- describe what is functional verification, what is logic simulation and what is the purpose of testbenches.
- describe previously encountered adder types (ripple-carry and Sklansky adders) at the RT level.
- perform basic functional verification using an industrial EDA tool for logic simulation.
- elaborate on how many test vectors does it really take, to know that we have verified a design.

Lab 2: Synthesis of Adder RTL Code

Here we are going to implement the 16-bit adders using the Genus synthesis tool. We will study how this synthesis tool goes about to map the hardware description to a netlist of standard cells in the 7-nm ASAP7 process technology.

Starting Cadence Genus: To start Genus, type `genus` at the command line. Once Genus has initialized, its graphical user interface (GUI) window will be hidden. While we will mainly work through the Linux-like command line interface, we can later use the GUI to study schematics of our synthesized netlists.

Retrieving the technology files: The VHDL descriptions will need to be mapped to a certain process technology, for which a vendor/chip foundry has supplied a library of standard cells. We attach the ASAP7 process technology to our work by giving the path and file name to libraries (`.lib`) of our choice. These library files contain cells which are implemented using regular-threshold-voltage (RVT) 7-nm FinFET transistors which are characterized (see Lecture 5) at nominal/typical process corners, at a 0.7-V supply voltage and a temperature of 25 °C.

```
set_db / .library \
{/usr/local/cad/asap7/asap7sc7p5t_28/LIB/CCS/asap7sc7p5t_AO_RVT_TT_ccs_211120.lib.gz \
/usr/local/cad/asap7/asap7sc7p5t_28/LIB/CCS/asap7sc7p5t_INVBUF_RVT_TT_ccs_220122.lib.gz \
/usr/local/cad/asap7/asap7sc7p5t_28/LIB/CCS/asap7sc7p5t_OA_RVT_TT_ccs_211120.lib.gz \
/usr/local/cad/asap7/asap7sc7p5t_28/LIB/CCS/asap7sc7p5t_SEQ_RVT_TT_ccs_220123.lib.gz \
/usr/local/cad/asap7/asap7sc7p5t_28/LIB/CCS/asap7sc7p5t_SIMPLE_RVT_TT_ccs_211120.lib.gz}
set_db / .lef_library \
{/usr/local/cad/asap7/asap7sc7p5t_28/techlef_misc/asap7_tech_4x_201209.lef \
/usr/local/cad/asap7/asap7sc7p5t_28/LEF/scaled/asap7sc7p5t_28_SL_4x_220121a.lef \
/usr/local/cad/asap7/asap7sc7p5t_28/LEF/scaled/asap7sc7p5t_28_L_4x_220121a.lef \
/usr/local/cad/asap7/asap7sc7p5t_28/LEF/scaled/asap7sc7p5t_28_R_4x_220121a.lef}
set_db / .qrc_tech_file \
/usr/local/cad/asap7/asap7sc7p5t_28/qrc/qrcTechFile_typ03_scaled4xV06
```

Like before, we can use backslash (\) to signal that the line continues below.

Reading the VHDL code: The provided VHDL code for the 16-bit ripple-carry adder is written in a hierarchy, where the top design file `Wrapper.vhdl` refers to the combinational part of the ripple-carry adder, which in turn refers to a full-adder block. Tell Genus to read in the hardware descriptions, by using the following command:

```
read_hdl -vhdl FA.vhdl RCA.vhdl Wrapper.vhdl
```

You should here list all your VHDL files that belong to the current design. If the files are located in a different directory, you need to specify this explicitly. For example, if you store them in a subdirectory called VHDL, you call on `VHDL/file1.vhdl`; if you store them in the parent directory, you call on `../file1.vhdl`. Note that the testbench should not be included in the files that you read into the tool at this stage, since the testbench is not synthesizable but only a behavioral definition.

Practical hints

1. It gets tedious to write long file paths for all commands. Then it is handy to use Genus's built-in variables. Take for example the file path used for the VHDL code, which can be defined like this:

   `set_db / .init_hdl_search_path VHDL`

   to reflect the location of the files which you have stored in the subdirectory called `VHDL`.

2. You can also define the process technology path used in `knuffodrag`:
   `set_db / .init_lib_search_path /usr/local/cad/asap7/asap7sc7p5t_28`
   to avoid using `/usr/local/cad/asap7/asap7sc7p5t_28` during technology file retrieval.

Elaboration of VHDL code: We instruct Genus to assemble all our VHDL files into an internal data representation (where we impose restrictions on what characters the tool can use) by typing

```
elaborate
change_names -allowed \
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_
```

If we receive no error messages, the VHDL code has turned out to be synthesizable: An HDL code that is synthesizable is often referred to as RTL code.

Practical hints

If we wish to study the logic structure implementation resulting from the elaboration (and, later, synthesis) phase, there are basically three different ways available to us.

1. We can have a look at the gate netlist produced during elaboration. First generate a netlist via `write_hdl > elaborated_adder.v` and then read this from a Linux terminal via `less elaborated_adder.v`. Note that the produced netlist is made up of Verilog code, thus the `.v` extension.
2. We can also study the implementation by trying to read the design directories, e.g., by using the `vls` command. Try for instance `vls designs/wrapper/add_inst`.
3. Finally, and this is probably the most useful way, we can study how the logic gates are assembled and interconnected through the GUI. Type `gui_show` to invoke the window system. You activate the schematic view by adding this view next to the default layout view.

Unconstrained synthesis: We will now convert our hardware descriptions to real hardware, to physical standard cells. First we will take a look at unconstrained synthesis which we perform to get a first estimate of the delay of the circuit.

During the initial logic synthesis phase where technology-independent RTL optimizations are performed, Genus makes use of a virtual gate library, to which no specific process technology is associated.

```
set_db / .auto_ungroup none
set_db / .syn_generic_effort low
syn_gen
```

Here we have told the tool to avoid doing automatic ungrouping. Additionally, we have chosen a low computational effort to ensure our RTL code's structure is preserved[3]. Although no physical cells have been used yet, we can find information on what generic gates the synthesis tool assumed.

The next step is to perform technology mapping, which is discussed in Lecture 4.

```
set_db / .syn_map_effort low
syn_map
```

Above we did not set any timing constraint, since we at this stage need to get an indication of the intrinsic implementation timing in order to set proper timing constraints in the next phase.

[3]The other effort modes are medium and high.

Now when the VHDL design has been taken through technology mapping, we can start to trust the estimated data from the general synthesis procedure. In fact, we can get a crude number on the worst-case path and its delay by using the static timing analysis (STA) engine internal to Genus:

```
report timing -unconstrained
```

Assignment

- Follow the flow outlined above and synthesize your VHDL code for the ripple-carry and Sklansky adders without any timing constraint. Study the worst-case delay value for the two adders.
- Next, track the worst-case signal propagation in the adders by examining the timing report. Reflect on whether this propagation agrees with your understanding of the adder architectures.

Once done with this assignment, exit Genus.

Timing-driven synthesis: After Genus has been restarted, and after the technology and VHDL files have been read and elaborated, we can introduce a signal with a clock waveform to our design. This will have the effect that a timing constraint is enforced. Here we assume all our clock signals on input and output registers (check the wrapper file) are called `clk`:

```
create_clock [get_ports clk] -name master_clock -period user_value
```

Here we say that our clock net has the name of `master_clock` and that the worst-case cycle time is the value you use to replace `user_value`[4]. It can be a good idea to start with the delay obtained for the unconstrained synthesis in the previous assignment.

By again using the synthesis commands, we can make Genus perform timing-driven synthesis:

```
set_db / .auto_ungroup none
set_db / .syn_generic_effort medium
set_db / .syn_map_effort medium
set_db / .syn_opt_effort high
syn_gen
syn_map
syn_opt
```

Here we also added an optimization phase. In general, the optimization goal during synthesis is to create the smallest possible implementation of the design that satisfies a certain timing constraint (if defined). However, if your timing constraint is too strict, the timing report will show a negative slack, meaning the circuit cannot operate at the clock rate implied by the period time of the timing constraint.

It may be useful to consider what is the area of the implementation. You can use

```
report gates
```

to find useful information.

[4]Picoseconds (ps) and femtofarads (fF) are Genus' default units for time and capacitance, respectively.

Assignment

- Synthesize your VHDL code for the ripple-carry and Sklansky adder, respectively, using as timing constraint the unconstrained delay you obtained in the previous assignment.
- Identify the worst-case delay value achieved after synthesis and ensure your timing constraint is met. Read the timing report to find out if the current worst-case timing path is different from the one shown in the unconstrained case.
- Contrast the timing and area reports obtained after `syn_map` with those that you obtain after `syn_opt`.
- Generally, it is important to find out how fast circuits can run and how strict timing constraints can be accepted before one obtains negative slacks.
  - Explore where is the limit to how fast you can run your adders, by gradually reducing the timing constraint value, until the synthesis results in a negative slack.
  - Make a note of the estimated area of each of the implementations you explored. Since you are writing a report on these labs, it is a good idea to send the tool's report outputs not only to the screen but also to files, for future use, e.g., by `report gates > gates.txt`.

Practical hints

1. There are many commands to write at the command prompts and the risk of making mistakes increases with command complexity. To make sure we use the commands in a consistent manner, we can use tool control language (TCL) scripts that help us batch a number of commands.
   (a) Define a number of commands in a text file, say `test.tcl`.
   (b) When inside Genus, run the commands that are lined up inside the script by typing `source test.tcl` or ...
   (c) you can start Genus from scratch and run the script from the command line: `genus -files test.tcl`. If you add the flag `-overwrite`, the number of log files will not grow.
2. You can generate timing reports where more than the longest path is present. For example, `report timing -max_paths 5` gives you the five most critical paths.
3. For the lab report, you may want to consider the following book chapters and research articles: [6, 7, 8, 9, 10, 11].

Learning outcomes of Lab 2

When you have completed this part of the labs, also assuming that you have attended the supporting lectures, you should be able to ...

- describe what is the difference between synthesizable and non-synthesizable/behavioral hardware descriptions.
- describe what is ASIC cell-based synthesis, i.e., logic synthesis and technology mapping.
- describe constraints and optimization goals during synthesis.
- describe what is static timing analysis (STA).
- perform basic logic synthesis; from hardware description language to generic gate level.

- perform basic technology mapping; from generic gate level to standard cells of a library.
- perform basic timing-constrained synthesis and carry out subsequent STA analysis.
- make use of basic TCL scripts to control a design flow.
- elaborate on how the timing constraint impacts design area.
- elaborate on what is the relation between VHDL code and synthesized Verilog code.

Lab 3: Verification of Netlist

In industrial design flows the final outcome of synthesis and technology mapping needs to be verified against the original specification, since synthesis tools are known to introduce errors once in a while; not often, but it happens. We now need to make sure that the implementation created by Genus — the gate netlist — is logically equivalent with the original VHDL code and that it meets performance requirements.

Netlist functional verification: To verify a netlist for functionality, we first need to generate one. Using what you learned in the previous lab, synthesize your VHDL code with a timing constraint for which function is guaranteed (check the timing report after synthesis!). Then instruct Genus to write a netlist: `write_hdl > netlist.v`.

Simulation of gate netlists bears many similarities to simulation of VHDL codes. We will in the following set up a logic simulation run using the same testbench and reference test vectors that we used in lab 1. But instead of using the VHDL RTL code of lab 1, we will use ASAP7's Verilog library, which describes all standard cells, and then add the netlist we just generated.

To avoid exiting the tool, we can use `shell` to issue Linux commands from within Genus. For example, read in the five required Verilog library files[5] using `ncvlog`:

```
set PATH_LIB /usr/local/cad/asap7/asap7sc7p5t_28
shell ncvlog $PATH_LIB/Verilog/asap7sc7p5t_AO_RVT_TT_201020.v
shell ncvlog $PATH_LIB/Verilog/asap7sc7p5t_INVBUF_RVT_TT_201020.v
shell ncvlog $PATH_LIB/Verilog/asap7sc7p5t_OA_RVT_TT_201020.v
shell ncvlog $PATH_LIB/Verilog/asap7sc7p5t_SEQ_RVT_TT_220101.v
shell ncvlog $PATH_LIB/Verilog/asap7sc7p5t_SIMPLE_RVT_TT_201020.v
```

Now, we can continue to read in the netlist and the testbench, and elaborate all descriptions:

```
shell ncvlog netlist.v
shell ncvhdl -64bit -v93 -messages TB.vhdl
shell ncelab -64bit -v93 -messages -access +rwc -timescale 1ns/1ps -delay_mode zero TB
```

You can now perform the simulation just as we did in the first lab.

```
shell ncsim -64bit TB
```

Assignment

- Synthesize your VHDL code and perform the netlist simulation as outlined above. Ensure the output produced by the netlist agrees with the expected output defined by the test vectors.

Timing analysis using netlist simulation: One aspect that should be verified is that the delay of the netlist satisfies the timing constraint that was used to generate the netlist. It is true that static timing analysis (STA) is performed as an integral process inside synthesis and although we can trust the STA to a large extent, we must ensure aspects that pertain to dynamic circuit behavior, e.g., initialization, are verified.

The timing simulations that you will perform are accurate in the sense that high-fidelity gate timing models are used. However, a weakness of simulation-based timing analysis is that we only sample some input test vectors out of a huge number of possible vectors.

[5] While these files have a TT designation, these Verilog files have no connection to timing corners.

In the previous section, we verified that the resulting netlist agrees with the intended logic functionality that was described in the VHDL files. This was possible because we used the `ncelab` switch `-delay_mode zero` which selects a zero delay model. In this part of this lab, delay models will be used in the logic simulation of netlists.

The analysis flow is quite similar to that of the netlist functional verification. First we synthesize the VHDL code with a timing constraint. Like before we instruct Genus to write a netlist, but in addition to this, we now need an SDF file:

```
write_hdl > netlist.v
write_sdf > netlist.sdf
```

Standard delay format (SDF) files[6] are used to describe timing information that can be shared between EDA tools. Using an SDF file for our adders enables us to run a timing simulation in which gate timing information is explicitly used.

We must now compile the SDF file and create an associated control file which can be used during elaboration. The first thing you can do from within Genus,

```
shell ncsdfc netlist.sdf
```

but the second thing requires a text editor. Either exit Genus or open one more terminal window. Create a text file by the name of `sdf.control` which has the following content:

```
COMPILED_SDF_FILE = "netlist.sdf.X",
SCOPE = :instance_name,
MTM_CONTROL = "MAXIMUM",
SCALE_FACTORS = "1.0:1.0:1.0",
SCALE_TYPE = "FROM_MTM";
```

where `netlist.sdf.X` is the name of the compiled SDF file and `instance_name` refers to the label used to instantiate the adder inside the testbench. Now we have completed the detour required to run the timing simulation and we can continue to read in the Verilog files as in the last assignment, elaborate the design and run the simulation.

```
ncvlog netlist.v
ncvhdl -64bit -v93 -messages TB.vhdl
ncelab -64bit -v93 -messages -access +rwc -timescale 1ns/1ps \
-sdf_cmd_file sdf.control -sdf_verbose TB
ncsim -64bit -messages TB
```

Above we assumed we worked from Linux and, furthermore, we modified the command line for `ncelab` to support SDF.

Assignment

- Following the flow given above, run a timing simulation on your adders. Ensure the simulations complete without any timing errors.

  While the flow above should work, you may have neglected one aspect which is that two different timing parameters are critically important: The timing constraint of the synthesis and the clock period of the testbench.

- Given that the simulation above yielded no errors, run a series of timing simulations where you gradually increase the testbench clock rate until you discover errors in the output result. Carefully document the behavior of the circuit for your lab report.

[6] Read more in the SDF document in Canvas: `sdf_3.0.pdf`.

Practical hints

1. When you have completed one elaboration/simulation run, remove the temporary folders `INCA_libs` and `fv` by using recursive remove `rm -r INCA_libs fv`.
2. When you edit the testbench for the first time, make sure the arrival time of all primary input signals is delayed from the clock edge to emulate a real circuit (use a `wait` statement); otherwise you will have hold-time violations.
3. Since ASAP7 is not a production library, there are some known issues with the cells. You may encounter a warning "`ncelab: *W,SDFNEP`" associated with a cell type. This problem can be remedied by excluding the cell from synthesis. Find the complete name of the cell, using `get_db lib_cells *keyword*`, exclude it using `set_db cell_name .avoid true` and re-run the synthesis.
4. For the lab report, you may want to consider the following book chapters and research articles: [12, 13, 14, 15].

Learning outcomes of Lab 3

When you have completed this part of the labs, also assuming that you have attended the supporting lectures, you should be able to ...

- describe what is simulation-based timing analysis, and what are the fundamental differences between this and static timing analysis.
- describe how functional verification of a synthesized netlist differs from functional verification of VHDL code.
- perform functional verification of synthesized implementations to verify implementation quality.
- perform simulation-based timing analysis of synthesized implementations to verify implementation quality.
- elaborate on cell timing models, in particular the composite current source (CCS) model used in the labs, e.g. in `asap7sc7p5t_AO_RVT_TT_ccs_211120.lib.gz`.
- elaborate on how will you be able know if you have exercised the worst case (longest delay) in the simulations.

Lab 4: Power Analysis

Switching power is the major power dissipation mechanism in digital circuits:

$$P_{sw} = f\,{V_{DD}}^2 \sum_{i=1}^{N} (C_i\,\alpha_i) \tag{1}$$

where i denotes one node in a circuit with N nodes. The switching activity α is obtained via an estimate based on likelihood of signal switching (from 0 to 1) in a node. The most accurate estimation is based on actual netlist logic simulations that use representative test vectors for the application domain. Clearly, in contrast to an accurate timing analysis based on STA, an accurate power analysis requires that the netlist implementation is simulated, which makes power (and energy) analysis complex and power values application dependent.

Simulation-based power analysis: We can begin our simulation-based power analysis by following the flow used for netlist verification in lab 3, until the testbench has been compiled with `ncvhdl`. The next two lines involve `ncelab` and `ncsim`, and here, by adding a configuration file, we can make Ncsim produce switching data that can be used by Genus to estimate the power dissipated.

In a different terminal window from where you use Genus, create a file called `ncsim-VCD.tcl` which contains the following five lines:

```
run 2 ns
database -open vcddb -vcd -default -into db.vcd -timescale ps
probe -create -vcd :instance_name -all -depth all
run
exit
```

using the proper replacement of the placeholder `instance_name`. The script tells `ncsim` to run the simulation for 2 ns and then open a database where value change dump (VCD[7]) switching data (`db.vcd`) are saved. This means we don't save any VCD data the first two cycles; yes, we have assumed that the clock period is 1 ns above. The reason we do this is to avoid switchings that are caused by the wrapper's pipeline being filled. Since we use no reset, we don't know the initial state of the pipeline registers.

Let use now perform the elaboration and simulation, from within the Genus window. Notice the `-input` flag added below for `ncsim`.

```
shell ncelab -64bit -v93 -messages -access +rwc -timescale 1ns/1ps -delay_mode zero TB
shell ncsim -64bit -input ./ncsim-VCD.tcl TB
```

After the simulation finishes, a VCD file called `db.vcd` should have been generated. Now we can use this VCD file to obtain the actual power dissipation, as follows:

```
read_vcd -static -vcd_module instance_name db.vcd
report power
```

As before, remember that `instance_name` is a placeholder for your instance name.

After we issued the command to backannotate the VCD file, we should doublecheck the messages from Genus. The asserted nets must be close to 100%. If this number is far away from 100%, then it is likely something is wrong.

[7] A VCD file contains information on all signals that toggle, for every clock cycle. As a result, VCD files can grow huge.

Assignment

- According to the flow above, synthesize and simulate each adder and generate power and area reports for each of them. Analyze the reports and reflect on the differences between ripple-carry and Sklansky adders, also considering different timing constraints/operating frequencies.
- In lab 1, you retrieved randomized test vectors from the course directory. Now it is time to retrieve another set of vectors from `1000_regular_TVs`. Use the previous flow, with the exception that you should use the new vectors during simulation. Finish by generating a power report for each of the two adders. Compare the reports generated using the random vectors with those generated with regular vectors.

Signal statistics: Eq. 1 shows that the switching frequency of a signal determines the average power dissipated. Genus can in fact present useful signal statistics in a format called toggle count format (TCF). The TCF data contain, on a per-instance basis, information on 1) to what extent a signal is in its high state and 2) how many times it switches during the run of all test vectors.

We can save the TCF signal statistics by issuing

```
write_tcf > interesting_data.tcf
```

The TCF file contains information such as: `"g52/Y" : "0.49770 919517000";` which reveals that pin Y was high 49.77% of the time VCD data were dumped and that this pin had a switching activity $\alpha = \{\frac{919{,}517{,}000}{2 \cdot 10^9} \cdot \text{clock period in ns}\}$.

Assignment

- Perform each of the four runs in the above assignment (RCA and Sklansky adders for random and regular vectors) and write out a TCF file for each run. For each file, consider the statistics that relate to the clock signal.
- Compare the TCF files obtained from the random vectors with those from the regular vectors. Study several different signals, among them the primary inputs `a` and `b` whose test vector files you can examine for comparison.

Practical hints

There are several ways to retrieve power data after synthesis.

1. Beside `report power`, one can also study individual gate instances by using
   `report instance -power instance_path_and_name`
   where path and name refers to arbitrary instances, such as
   `report instance -power designs/wrapper/add_inst/FAG_8__FA_inst/g78`
2. Another option is to go for a power breakdown into different types of cells:
   `report gates -power`
   This option also has the informative feature (like `report gates` that was used for area estimation) to present the portion of power for each of sequential, inverter, buffer and logic type of cells.
3. The following option is also very useful, because it lets you monitor power dissipation based on the code lines in the VHDL code. In your script, insert
   `set_db hdl_track_filename_row_col true`
   before the VHDL files are read. Invoke Genus, carry out synthesis, logic simulations and switching data backannotation, and generate an RTL power report via
   `report power -rtl_cross_reference -detail -flat`

Switching vs leakage power: When you study the power reports, you will see that the static, leakage portion of power is insignificant. Since we are using an RVT library, with its relatively high VT, this is however to be expected.

When we read in the cell libraries, replace RVT with SLVT to retrieve the cells using super-low-threshold-voltage FinFETs. The SLVT variant below is for the TT corner, but we may also try the other corner options, FF and SS, for our analysis.

```
set db / .init_lib_search_path /usr/local/cad/asap7/asap7sc7p5t_28
set_db / .library \
{LIB/CCS/asap7sc7p5t_AO_SLVT_TT_ccs_211120.lib.gz \
 LIB/CCS/asap7sc7p5t_INVBUF_SLVT_TT_ccs_220122.lib.gz \
 LIB/CCS/asap7sc7p5t_OA_SLVT_TT_ccs_211120.lib.gz \
 LIB/CCS/asap7sc7p5t_SEQ_SLVT_TT_ccs_220123.lib.gz \
 LIB/CCS/asap7sc7p5t_SIMPLE_SLVT_TT_ccs_211120.lib.gz}
```

Remember also to change the Verilog files for the cell library; while these contain no timing information, the RVT and SLVT libraries differ in terms of which cells are supported.

Assignment

- Synthesize one of your adders with the SLVT libraries above, and simulate it using the random vectors. Save power and area reports and compare these with the reports obtained for the same adder with random vectors synthesized with the RVT option.
- Replace the SLVT TT option with SLVT SS and SLVT FF and synthesize and simulate the adder again under the same conditions as above. Make a note of differences in the power and area reports.
- Copy one of the five technology files, use `gunzip` to uncompress it and read the information at the top of the file. Do this for each of the corners, SS, TT and FF, and compile what supply voltage and temperature assumptions are made to set up these libraries.
- We can expect leakage to increase when a lower VT is used. But in exchange higher performance is possible. Perform SLVT synthesis runs to establish how strict timing constraints the three different corners can handle.

Practical hints

1. You may want to run your synthesis through a range of parameter values, such as timing constraints. Then you can develop a TCL script which loops through a number of values and calls on another TCL script, e.g.

   ```
   for {set TC 10} {$TC <= 20} {set TC [expr $TC + 1]} {
      source synthesize.tcl
      delete_obj [get_db design:wrapper]}
   ```

   In `synthesize.tcl`, we can then make use of the variable called `$TC`.
2. For the lab report, you may want to consider the following book chapters and datasheet: [16, 17, 18, 19].

Learning outcomes of Lab 4

When you have completed this part of the labs, also assuming that you have attended the supporting lectures, you should be able to ...

- describe what parameters decide switching power dissipation in digital circuits.

- describe how stricter timing constraints impact area and power dissipation.
- describe how clock rate and signal toggling probability impact power dissipation.
- describe what are sequential, inverter, buffer and logic cells.
- describe what is the impact of using higher-speed standard cells on timing and static power dissipation.
- perform power analysis using information on signal switching activity.
- perform power analysis using backannotation of logic simulations of real use cases.

References

[1] L. Vega, J. Roesch, J. McMahan, and L. Ceze, "LastLayer: Toward hardware and software continuous integration," *IEEE Micro*, vol. 40, no. 4, pp. 103–111, 2020.

[2] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001.

[3] J.-L. Huang, C.-K. Koh, and S. F. Cauley, "Chapter 8 - Logic and circuit simulation," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 449–512.

[4] H.-P. C. Wen, L.-C. Wang, and K.-T. T. Cheng, "Chapter 9 - Functional verification," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 513–573.

[5] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-level simulation with GPU computing," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, June 2011.

[6] S. P. Khatri, N. V. Shenoy, J.-C. Giomi, and A. Khouja, "Chapter 2 - Logic synthesis," in *Electronic Design Automation for Integrated Circuits Handbook, Vol 2*, 2nd ed., L. Lavagno, I. L. Markov, G. E. Martin, and L. K. Scheffer, Eds. Boca Raton: CRC Press, 2016.

[7] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 240–253, 2007.

[8] W. Gosti, S. Khatri, and A. Sangiovanni-Vincentelli, "Addressing the timing closure problem by integrating logic optimization and placement," in *IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*, 2001, pp. 224–231.

[9] P. Larsson-Edefors and K. Jeppson, "Timing-and power-driven ALU design training using spreadsheet-based arithmetic exploration," in *10th European Workshop on Microelectronics Education (EWME)*, 2014, pp. 151–154.

[10] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer, "Statistical timing analysis: From basic principles to state of the art," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 4, pp. 589–607, 2008.

[11] L. T. Clark, V. Vashishtha, D. M. Harris, S. Dietrich, and Z. Wang, "Design flows and collateral for the ASAP7 7nm FinFET predictive process design kit," in *IEEE Int. Conf. on Microelectronic Systems Education (MSE)*, 2017.

[12] T. B. Ahmad and M. J. Ciesielski, "Fast STA prediction-based gate-level timing simulation," in *Design, Automation & Test in Europe Conference*, 2014.

[13] D. Garyfallou, S. Simoglou, N. Sketopoulos, C. Antoniadis, C. P. Sotiriou, N. Evmorfopoulos, and G. Stamoulis, "Gate delay estimation with library compatible current source models and effective capacitance," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 5, pp. 962–972, 2021.

[14] V. A. Chhabria, B. Keller, Y. Zhang, S. Vollala, S. Pratty, H. Ren, and B. Khailany, "XT-PRAGGMA: Crosstalk pessimism reduction achieved with GPU gate-level simulations and machine learning," in *ACM/IEEE Workshop on Machine Learning for CAD*, 2022, pp. 63–69.

[15] R. Trihy, "Addressing library creation challenges from recent liberty extensions," in *Design Automation Conference*, 2008, pp. 474–479.

[16] J. Monteiro, R. Patel, and V. Tiwari, "Chapter 3 - Power analysis and optimization from circuit to register-transfer levels," in *Electronic Design Automation for Integrated Circuits Handbook, Vol 2*, 2nd ed., L. Lavagno, I. L. Markov, G. E. Martin, and L. K. Scheffer, Eds. Boca Raton: CRC Press, 2016.

[17] J. Rabaey, "Chapter 3 - Power and energy basics," in *Low Power Design Essentials*. Boston, MA: Springer US, 2009.

[18] *PrimePower RTL to Signoff Power Analysis*, Synopsys, Inc., 2020, Accessed 2022-10-04. [Online]. Available: www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/primepower-ds.pdf

[19] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: a 7-nm FinFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.