Exercises in

DAT110 Methods for Electronic System Design and Verification

Per Larsson-Edefors, November 1, 2018

In these exercises you will get hands-on experience of a number of critical 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, for example, FPGA synthesis.

While the lectures of this course will address key Electronic Design Automation (EDA) techniques and methods for design and verification, these exercises 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

```
https://pingpong.chalmers.se/launchCourse.do?id=10227
```

Getting started in the Linux environment

First, license servers and file search paths need to be defined. This example assumes bash:

```
#!/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/rc-14.13.000/bin:/usr/local/
cad/encounter-14.27.000/bin

export SNPSLMD_LICENSE_FILE=5290@synopsys1.lic.chalmers.se PATH=$PATH:/
usr/local/cad/coretools-I-2013.12-SP5/bin:/usr/local/cad/pts-2011.06/bin
```

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

Note that `cdnshelp` is a huge help resource on NCSIM, RTL Compiler and associated tools:

```
/usr/local/cad/incisive-15.20.061/tools.lnx86/bin/cdnshelp

/usr/local/cad/rc-14.13.000/tools.lnx86/bin/cdnshelp
```

My acknowledgments: Thanks to ...

- Cadence for granting us licenses on Encounter.
- Dr. Magnus Själander for support during initial development (2007). Dr. Tung Hoang for support during development of lab exercise 4 (2008). Dr. Alen Bardizbanyan for support on ALU updates (2010, 2014), P&R timing closure (2011), SDF (2012), RC extraction (2013), and ChipWare (2014). Dr. Kasyab Subramaniyan for support on P&R features (2010, 2012). Junaid Elahi, Erik der Hagopian, Christoffer Fougstedt, Erik Börjeson, Dr. Daniel Andersson and Dr. Minh Q Do for reviewing various lab memo versions.
- the late Lars Kollberg for software management during the initial four years.

Content

Preparation

In the following exercises, you are going to explore an industrial digital ASIC design flow using an ALU for a 32-bit processor. We begin with design preparations, since you need to have a structural idea about what you are going to design before you start coding.

Specification

The ALU input (`A`, `B`, `Op`) and output (`Outs`) should be registered in edge-triggered flip-flops, active on the rising clock edge (if `Reset_n` = 1). With the required reset signal, the following ports should be defined using VHDL:

```
entity ALU is
    port(
        Clk     : in  std_logic;
        Reset_n : in  std_logic;
        A       : in  std_logic_vector(31 downto 0);
        B       : in  std_logic_vector(31 downto 0);
        Op      : in  std_logic_vector(3 downto 0);
        Outs    : out std_logic_vector(31 downto 0));
end ALU;
```

We assume two's complement number representations (**note** that 1000 .... 0000 = -2^{31} does not represent a legal two's complement 32-bit number) and the following opcodes (`Op`):

```
* 0000: add A+B (signed)
* 0001: add A+B (unsigned)
* 0010: sub A-B (signed)
* 0011: sub A-B (unsigned)
* 0100: bitwise OR
* 0101: bitwise AND
* 0110: bitwise XOR
* 0111: bitwise NOR
* 1000: shift left
* 1010: shift right (logical)
* 1011: shift right (arithmetical, signed)
* 1110: SLT (Set on Less Than)
* 1111: SLTU (Set on Less Than Unsigned)
```

For the three shift operations, operand `A` is shifted by an amount decided by the value of `B`. Since there is a limit in shifting distance for `A`, only the least significant bits of `B` are used. For SLT and SLTU, compare `A` and `B` as signed and unsigned numbers, respectively. If `A` < `B`, then set the 32-bit ALU output to 0000 ... 0001, else set the output to 0000 ... 0000.

Preparatory assignment

a) Draw an ALU block diagram, showing all major building blocks (addsub, logic, and shift).

b) Draw a timing diagram that shows the input and output data signals of the ALU entity above, in relation to the reset and clock signals.

c) Briefly explain the difference of a ripple-carry adder and Sklansky adder in terms of gates, timing, logic depth etc. (*Consult the textbook of MCC092 and its chapter on datapaths.*)

d) What is the difference between arithmetic and logical right shift, and which elementary arithmetic operation does the shift right operation correspond to?

e) Assume the add/subtract unit generates 32-bit output results as well as two flags called, for example, *CO* (carry out, 1-bit) and *V* (overflow, 1-bit): Based on the available signals, define two Boolean functions that return logic '1' for SLT and SLTU respectively.

f) In `Sklansky.vhdl`, some definitions are left out (`--!find the correct description!`). Beside *CO* and *V* above, three more are left out. Define the three proper *G* and *P* signals.

Individual PDF solutions to this assignment
are to be uploaded to the PingPong system
on Monday November 12, 2018 (midnight) at the latest.

Exercise 1: ALU Design and Verification (SW 2-3)

In the preparatory assignment you were asked to conceive an ALU of a 32-bit processor. Now it is time to describe, by using a hardware description language, the ALU hardware that you have conceived and the testbench required to test the ALU function.

Hints on file and directory structure: It is advisable to use different directories for different exercises and 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.

1.1. ALU description and testbench design

ASSIGNMENT 1.1.1:

a) Based on the preparatory assignment, write the ALU's VHDL code. Prepare one code (`ALU_RCA.vhdl`) which uses an ALU adder based on the ripple-carry principle and another code (`ALU_SKL.vhdl`) for an adder based on a prefix tree of Sklansky type, which has been reviewed in the previous course of MCC092.

You may make use of an expressive VHDL-library such as `numeric_std`, as complement to the standard one of `std_logic_1164`. Furthermore, you have access to a Sklansky code skeleton (`Sklansky.vhdl`) in PingPong, under *Documents/Lab exercises*.

b) Write a testbench for the ALUs. This testbench should be written using behavioral VHDL, since it will never be synthesized. Furthermore, it should have an empty entity, and it must supply features such as clock and reset generation (see Lecture 2).

Some functions for parsing the test vector files may be useful for the testbench; see below. The functions supplied here assume that there are at most 1,000 vectors in the test vector stimuli files. You may download the VHDL example below, as `initial_ALU_code.vhdl` from PingPong, *Documents/Lab exercises*.

```
--------------------------------------------------------------------------------
use std.textio.all; --this library is needed for textio functions
--------------------------------------------------------------------------------
-- Declarations
--------------------------------------------------------------------------------

constant Size    : integer := 1000;
type Operand_array is array (Size downto 0) of std_logic_vector(31 downto 0);
type OpCode_array is array (Size downto 0) of std_logic_vector(3 downto 0);

--------------------------------------------------------------------------------
-- Functions
--------------------------------------------------------------------------------
function bin (
  myChar : character)
  return std_logic is
  variable bin : std_logic;
begin
  case myChar is
    when '0' => bin := '0';
    when '1' => bin := '1';
    when 'x' => bin := '0';
    when others => assert (false) report "no binary character read" severity failure;
  end case;
  return bin;
end bin;
```

```
  function loadOperand (
    fileName : string)
    return Operand_array is
    file objectFile : text open read_mode is fileName;
    variable memory : Operand_array;
    variable L      : line;
    variable index  : natural := 0;
    variable myChar : character;
  begin
    while not endfile(objectFile) loop
      readline(objectFile, L);
      for i in 31 downto 0 loop
        read(L, myChar);
        memory(index)(i) := bin(myChar);
      end loop;
      index := index + 1;
    end loop;
    return memory;
  end loadOperand;

  function loadOpCode (
    fileName : string)
    return OpCode_array is
    file objectFile : text open read_mode is fileName;
    variable memory : OpCode_array;
    variable L      : line;
    variable index  : natural := 0;
    variable myChar : character;
  begin
    while not endfile(objectFile) loop
      readline(objectFile, L);
      for i in 3 downto 0 loop
        read(L, myChar);
        memory(index)(i) := bin(myChar);
      end loop;
      index := index + 1;
    end loop;
    return memory;
  end loadOpCode;
```

In the main code we can call the parsing functions to open and read files, by for example using

```
AMem        <= loadOperand(string'("A.tv"));
BMem        <= loadOperand(string'("B.tv"));
OpMem       <= loadOpCode(string'("Op.tv"));
OutputMem   <= loadOperand(string'("Output.tv"));
```

Here the arrays, in which we store test vector data, have been defined as signals inside the architecture

```
signal AMem       : Operand_array := (others => (others => '0'));
signal BMem       : Operand_array := (others => (others => '0'));
signal OpMem      : OpCode_array := (others => (others => '0'));
signal OutputMem  : Operand_array := (others => (others => '0'));
```

1.2. Verification of ALUs

Using the ALU testbench, we now have to make sure that the hardware that we have described is correct in the sense 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 first exercise deals with testing and debugging of the 32-bit ALU designs using the NCSIM logic simulation tool of the Incisive Enterprise Simulator (IES); Cadence's correspondence to Mentor Graphics' ModelSim. We will take the ALU hardware descriptions and run them inside a testbench. During logic simulation, the testbench will read test vector data from different stimuli files and send these to

the ALU. On the output of the ALU the testbench will collect the produced data and compare these to the expected — the reference case — data.

We will now compile the VHDL files. In the following we assume a testbench, called `ALU_TB.vhdl`[1], whose architecture is called `BEHAVIORAL` and which uses the entity ALU inside `ALU_RCA.vhdl`. The latter in turn uses the adder entity RCA inside `RCA.vhdl`, in which full adder cell entities FA (of `FA.vhdl`) are used. To simulate this VHDL hierarchy we first compile the VHDL blocks, including the testbench (`ALU_TB.vhdl`)

```
ncvhdl -v93 FA.vhdl
ncvhdl -v93 RCA.vhdl
ncvhdl -v93 ALU_RCA.vhdl
ncvhdl -v93 ALU_TB.vhdl
```

after which we elaborate (connect) them

```
ncelab -messages -v93 WORKLIB.ALU_TB:BEHAVIORAL
```

We are now in the position to simulate the testbench, using NCSIM

```
ncsim WORKLIB.ALU_TB:BEHAVIORAL
```

ASSIGNMENT 1.2.1:

a) Verify `ALU_RCA.vhdl` by using logic simulation, inside the testbench from **Assignment 1.1.1(b)**.

b) Verify `ALU_Skl.vhdl` like above.

Both your designs should give no errors when simulated against the 1,000 randomized reference test vectors that are provided in PingPong, under *Documents/Lab exercises*: `tvs_random.zip` (containing `A.tv`, `B.tv`, `Op.tv`, and `Output.tv`). The test vector files are terminated on line 1001 with a dummy vector x...x, except `Op.tv` which is terminated by 1001, which is an illegal op code that can be used by a testbench to detect end of file.

Note that the test vector files we provide represent a combinatorial ALU with zero latency. The testbench, thus, has to internally account for the pipelining that is associated with the input and output registers.

Practical verification tips:

1. Use the assert function 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 when you need to dig deeper into a block to localize a bug.

2. If your assert functions discover surprisingly few (or no) errors, 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 rerun 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.

---

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

1.3. Using coverage metrics

To ensure all parts of our code are covered in simulation, we can use a tool called Incisive Metrics Center (IMC). However, to be able to use this tool, we need to first re-run the simulations in the previous section using slightly different settings to generate coverage data. In the general flow of `ncvhdl` -> `ncelab` -> `ncsim`, we can keep the `ncvhdl` commands intact, while we need to add a flag to the elaboration phase:

```
ncelab -covfile cov_options.ccf -messages -v93 WORKLIB.ALU_TB:BEHAVIORAL
```

The coverage options file that has been added, `cov_options.ccf`, should contain the following line:

```
select_coverage -block -expr -toggle -module *
```

Using this options file, we instruct the tools to generate coverage data with respect to block, expression and toggle coverage (see Lecture 2) for all modules in the ALU design.

Now run `ncelab` as described above, followed by `ncsim` (as before, in **Section 1.2**). Then start IMC by issuing the following command:

```
imc
```

Once the main tool window has initialized, click the *Load...* button (under the File tab) and select the directory containing the coverage data; this is called `test` and can be found in `cov_work/scope`: A summary page is now being presented. You can change the information that is shown in this page, by using the *Views* menu which currently has *All_Metrics* enabled, but which just as well can take on code, block or toggle views. The Details window down to the right will adapt to the view and the component selection that you choose. For the default *All_Metrics* view, the Details window contains all possible coverage parameters, i.e., Code (including Block, Expression and Toggle), FSM and Functional. In our ALU design, the only relevant coverage metrics produced during the simulations are Block and Toggle, so you can instead select *Metrics_Code* view as the standard view.

Now left click the logic unit instance in the Verification Hierarchy window. (If you have forgotten the name of this instance, hover the cursor over the instance names to get more information.) Once you've clicked on the instance, the Details window is updated to reflect this. If you select the Source tab, you'll get a view of the VHDL code of the logic unit component as it is being instantiated.

Select the default *Metrics* tab in the Details window and double click on Block. (If a particular instance does not contain any block coverage information, nothing will happen.) Now IMC opens a new page, which is called by the name of the logic unit instance and is listed in the leftmost Navigation pane, under the default summary page. On this page you'll have access to detailed information on the block coverage of the logic unit. (As was introduced in Lecture 2, a block is a number of code lines in which all lines are executed once the block has been entered. This implies a block of code lines that follows on statements like if-then-else and case.) The left window of this page is called Blocks and here the coverage of the different blocks in the logic unit file are listed, from index 1 and onwards. When you click on any of the blocks presented, the Source window to the right will highlight the VHDL code line where the block you clicked on starts. In the Source window, a covered block is highlighted green and an uncovered block is highlighted red.

ASSIGNMENT 1.3.1:

Ensure that block and toggle coverage are 100% for all instances in the RCA-based ALU when exposed to the 1,000 randomized reference test vectors and consider the following questions:

Are there any coverage data that stand out from the rest?

Can you think of any potential problems with the code even in the presence of 100% coverage?

If you experience lower coverage in, e.g., the testbench due to error-detecting assert statements never visited, you have a possibility to exclude code from the coverage analysis. You can include, in a VHDL file, pragmas that control the coverage engine of the tools. Turn off coverage using

```
-- pragma coverage off
```

and turn it on again using

```
-- pragma coverage on
```

Of course, once you have edited a VHDL file, you need to re-compile, re-elaborate and re-simulate all files.

The VHDL codes for ALU_RCA, its subblocks and the testbench are to be submitted on
Friday November 23, 2018 (midnight) at the latest.
Only submit the files specified above
(this means that the Sklansky design is not to be submitted).

Call the top-level VHDL file `ALU_RCA.vhdl` and the testbench `ALU_TB.vhdl`.
Since we will automate the testing of the submitted files, make sure you conform to our conventions for filenames, port definitions, etc.

Although you work in a team, this is an individual submission.

1.4. Learning outcome of Exercise 1

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

- design, verify and debug an ALU and other units of such complexity, with associated testbench, using your updated hardware description language skills.
- describe what is functional verification and what is logic simulation.
- describe the purpose of testbenches for functional verification.
- describe what is code coverage.
- perform basic functional verification using an industrial EDA tool for logic simulation.
- elaborate on the issue of code coverage in logic simulation;
  that is, how many test vectors does it really take to know that we have verified a design.

Exercise 2: Synthesis and Timing Analysis (SW 4)

This second exercise is focused on the implementation of our 32-bit ALU using synthesis. We will study how the synthesis tool goes about to map the hardware description to standard cells in our 65-nm process technology.

2.1. Synthesis of ripple-carry ALU

Starting Cadence RTL Compiler

Type

```
rc -gui
```

to start RTL Compiler.

Since we will mainly work through the Linux-like command line interface, we can exit or hide the graphical user interface (GUI) window when RTL Compiler has finalized its initialization phase. We will later use the GUI, but only to the extent that it shows block schematics of our synthesized netlists.

Retrieving an STM 1.2-V 65-nm technology file

The VHDL descriptions will need to be mapped to a certain process technology, for which our vendor ST Microelectronics has supplied a library of standard cells. We attach the 65-nm process technology to our work by giving the path and file name to a library (`.lib`) of our choice. This library file contains cells which are implemented using Low-Power Standard-VT transistors and which are characterized at nominal process corners, at 1.2-V supply voltage and a temperature of 25 C.

```
set_attribute library {{ /usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/
5.2.c/libs/CORE65LPSVT_nom_1.20V_25C.lib }}
```

Reading the ALU_RCA VHDL code

You have most likely written the VHDL code in a hierarchy, where the top design file `ALU_RCA.vhdl` refers to subblocks such as the ripple-carry adder, which in turn perhaps refers to full-adder subblocks. Tell RTL Compiler to read in the hardware descriptions, by using the following command:

```
read_hdl -vhdl file1.vhdl file2.vhdl ...
```

You should here list all your VHDL files that belong to the ripple-carry adder ALU. 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.

Elaboration of VHDL code

We instruct RTL Compiler to assemble all our VHDL files into an internal 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.

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, by using the following command:

   ```
   write_hdl > elaborated_ALU_RCA.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, for example, by typing[1]

   ```
   ls /designs/ALU/
   ```

   or whatever we have called our ALU entity in `ALU_RCA.vhdl`.

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, in case you exited this after startup. The current netlist status is likely to be shown automatically; if not, press Update GUI under the File menu.

A first try at synthesizing the VHDL code to hardware

Now it is time to convert our hardware descriptions to real hardware, to physical standard cells. During the elaboration phase — the initial logic synthesis — RTL Compiler makes use of a virtual gate library, to which no specific process technology is associated. The synthesis phase, as defined in RTL Compiler and in these lab exercises, is commonly referred to as technology mapping, see Lecture 4.

Carry out the technology mapping by typing

```
synthesize -to_mapped -effort low
```

In general, the optimization goal during synthesis and, in particular, technology mapping is to create the smallest possible implementation of the design that satisfies a certain timing constraint (if defined). Above we did not set any timing constraints, since we needed to get an indication of the intrinsic implementation timing in order to set proper timing constraints in later design steps. Furthermore, we chose a low computational effort[2] above, since we did not want the synthesis engines of RTL Compiler to optimize timing at all. **Note**, however, that the ALU now has been taken through technology mapping and, thus, that we now can start to trust the estimated data from the general synthesis procedure.

We find the worst-case timing through the following command:

```
report timing
```

ASSIGNMENT 2.1.1:

As defined previously, synthesize `ALU_RCA.vhdl` without any timing constraint.

a) What is the worst-case delay value?

---

1. Type this Linux-like command from within RTL Compiler that has its own design directory structure.
2. The other effort modes are medium (which is default) and high.

b) Track the worst-case signal propagation of the ALU by examining the timing report and mark (draw) the signal path in your block diagram. In the GUI window, it is possible to get a simplified schematic of this path: *Report -> Timing -> Worst Path ...*

c) What is the estimated area of the implementation? `report gates` gives useful information.

Since you are writing a report on these lab exercises, it is a good idea to send the report outputs also to files, for future use, for example, by `report gates > gates_report.txt`.

Defining timing constraints for synthesis

Assume all our clock signals on input and output registers are called Clk. We can introduce a signal with a clock waveform to our design, with the effect that a timing constraint is enforced

```
define_clock -name main_clk -period 2092 [find / -port Clk]
```

Here we say that our clock net has the name of `main_clk` and that the worst-case cycle time should be 2092 ps[1]. This value happens to be 50% of the value that the teachers obtained in **Assignment 2.1.1(a)**.

If we study the timing report (yes, type `report timing` again) we can now see that we have changed the timing requirement; check the last line in the report.

ASSIGNMENT 2.1.2:

Synthesize `ALU_RCA.vhdl` with the new timing goal of 50% of the delay you obtained in **Assignment 2.1.1(a)**. Let the synthesis optimization spend more computational power than previously, by employing medium synthesis effort.

a) Retrieve the worst-case delay value achieved after synthesis. Does the synthesis manage to fulfil the timing constraint?

b) Evaluate the timing report to find out what has happened to the worst-case timing path and the associated gates.

Hint: As a reference on what 866 standard cells are available, one databook is useful: `CORE65LPSVT_1.20V_ds.pdf`. This document is available in PingPong, under *Documents/Lab exercises*.

c) What is the estimated area of the implementation?

---

1. picoseconds and femtofarads are RTL Compiler's default units for time and capacitance, respectively.

Stricter timing constraints

The original intention of our ALU design actually is to put it inside a 1-GHz processor.

Assignment 2.1.3:

Synthesize `ALU_RCA.vhdl` with the new timing goal, using medium effort.

Note that you first need to delete the design database of RTL Compiler via `rm /designs/*` every time you initiate a new synthesis run. The reason for this is that heuristic algorithms are strongly dependent on the intitial state. Thus, to be able to reproduce synthesis results, which is important from a traceability point of view, you have to start from a known state, that is, a clean design database. (A more cumbersome option is to re-start RTL Compiler in order to clear the database.)

a) Document the worst-case delay value achieved after synthesis. Does the synthesis manage to fulfil the timing constraint?

b) Evaluate the timing report to find out what has happened to the worst-case timing path: Is the critical path located in the same circuits as before, in **Assignment 2.1.1(b)**?

c) What is the estimated area of the implementation?

Practical tip: There are a lot of 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 scripts that help us batch a number of commands.

1. Define a number of commands in a text file, say `test.s`.

2a. When inside, for example, RTL Compiler run the commands that are lined up inside `test.s` by writing

```
source test.s
```

or

2b. you can start RTL Compiler from scratch and run the script:

```
rc -files test.s
```

If you add the flag `-overwrite`, the number of log files will not grow.

2.2. Verification of synthesized netlist for ripple-carry ALU

In industrial design flows the final outcome of synthesis and technology mapping needs to be verified[1] 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 proposed by RTL Compiler — the gate netlist — is equivalent with the original VHDL code.

---

1. Either through logic simulation (in, for example, NCSIM) or by formal methods.

Assignment 2.2.1:

Synthesize `ALU_RCA.vhdl` with a timing goal for which function is guaranteed (check the timing report after synthesis!). Then instruct RTL Compiler to write a netlist

```
write_hdl > ALU_RCA_netlist.v
```

Now, exit RTL Compiler or open a new terminal window. Run a logic simulation on the netlist using the testbench of `ALU_TB.vhdl` and the reference test vectors to ensure that the implementation is ok. The procedure is outlined below[1].

In **Section 1.2** we learned the basic trade of logic simulation of VHDL code. The simulation of the gate netlist bears many similarities to simulation of VHDL code; we can for example use the same testbench for both types of simulations. But instead of reading in our VHDL code, that is, `FA.vhdl`, `RCA.vhdl`, `ALU_RCA.vhdl`, and `ALU_TB.vhdl`, we read in the Verilog library that describes the standard cells that we have used for synthesis as well as the netlist that resulted from synthesis:

```
ncvlog /usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/5.2.c/behaviour/
verilog/CORE65LPSVT.v
```

```
ncvlog ALU_RCA_netlist.v
```

We also read in the testbench and elaborate all descriptions:

```
ncvhdl -v93 ALU_TB.vhdl
```

```
ncelab -messages -v93 -timescale 1ns/1ps -delay_mode zero
WORKLIB.ALU_TB:BEHAVIORAL
```

You can now perform the simulation just as we did in the previous simulation in **Section 1.2**.

With the `ncelab` switch `-delay_mode zero` we select a zero delay model, that is, we only verify the logic functionality in this simulation. Next, delay models will be considered for logic simulation.

2.3. Timing analysis using netlist simulation of ripple-carry ALU

In the previous section, we verified that the resulting netlist agrees with the intended logic functionality that was described in the VHDL files. One more aspect that should be verified is that the timing of the netlist satisfies that timing constraint that was used to generate the netlist. So far, only static timing analysis (STA) has been performed as an integral process inside synthesis. Although we will be able to trust the STA to a large extent, we must ensure aspects that pertain to dynamic circuit behavior — for example initialization — are verified.

Standard delay format (SDF) files[2] are used to describe timing information that can be shared between EDA tools. Using an SDF file for our ALU enables us to run a timing simulation, that is, a simulation similar to the functional simulation of **Section 2.2** but one in which gate timing information is explicitly used.

To perform simulation-based timing analysis we must create an SDF file and an associated control file for the ALU. Begin by invoking the Synopsys PrimeTime tool:

```
pt_shell
```

---

1. When you are using NCSIM, if you feel that the simulator is behaving in a strange way, try to remove the temporary folder (INCA_libs) and elaborate all your files again.
2. Read more in the SDF document in PingPong, under *Documents/Lab exercises*: `sdf_3.0.pdf`.

From within PrimeTime issue the following commands to create `alu.sdf`:

```
set target_library {/usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/5.2.c/
libs/CORE65LPSVT_nom_1.20V_25C.db ALU_RCA_netlist.v}

set link_library "/usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/5.2.c/libs/
CORE65LPSVT_nom_1.20V_25C.db ALU_RCA_netlist.v"

read_verilog ALU_RCA_netlist.v

link_design ALU

write_sdf -version 3.0 -map /usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/
5.2.c/behaviour/verilog/CORE65LPSVT.verilog.map alu.sdf
```

This PrimeTime phase is necessary since only `.db` timing models are available for this version of the process technology.

To later use the SDF file, we must compile it to a different format:

```
ncsdfc alu.sdf
```

The resulting file has an `.X` extension.

To run the timing simulations, we also need an SDF control file. Create a text file by the name of `sdf.control` which has the following content:

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

where `instance_name` refers to the label used to instantiate the ALU inside the testbench.

Before the timing simulations commence, compile the Verilog files like we did in **Section 2.2**:

```
ncvlog /usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/5.2.c/behaviour/
verilog/CORE65LPSVT.v

ncvlog ALU_RCA_netlist.v
```

For each new timing simulation, you clearly have to update the clock period of the testbench. When you edit the testbench for the first time, make sure the arrival time of all primary input signals is delayed to emulate a real circuit (use a `wait` statement); otherwise you will have hold time violations. The following three steps should be performed for each new timing simulation:

```
ncvhdl -v93 ALU_TB.vhdl

ncelab -messages -v93 -access +rwc -timescale 1ns/1ps -sdf_cmd_file
sdf.control -sdf_verbose WORKLIB.ALU_TB:BEHAVIORAL

ncsim WORKLIB.ALU_TB:BEHAVIORAL
```

Assignment 2.3.1:

a) Following the flow given above, run a timing simulation on your ripple-carry ALU using a testbench clock period that is identical to the timing constraint of **Section 2.2**. Ensure the simulation completes without any errors.

b) Given the simulation in **(a)** 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.

The simulations that you perform above 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. Consider how will you be able know if you have exercised the worst case (longest delay) in the simulations.

2.4. Learning outcome of Exercise 2

When you have completed this exercise, 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, that is, logic synthesis and technology mapping.
- describe what is static timing analysis (STA) and what is simulation-based timing analysis, and what are the fundamental differences between the two types of analysis.
- describe how functional verification of a synthesized netlist differs from functional verification of VHDL code.
- perform basic logic synthesis; from hardware description language level 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.
- perform functional verification of synthesized implementations to verify implementation quality.
- perform simulation-based timing analysis of synthesized implementations to verify implementation quality.
- make use of basic tool script to control a design flow.
- elaborate on what is the relation between initial ALU code and synthesized Verilog code.

Exercise 3: Timing Closure and Power Analysis (SW 5)

3.1. Synthesis of Sklansky ALU

A parallel-prefix adder is known to have a better potential for high speed than a ripple-carry structure; the former depends logarithmically on the word length, whereas the latter has a linear dependence.

Assignment 3.1.1:

Synthesize `ALU_Skl.vhdl` with low effort and without any timing constraint.

a) Document the worst-case delay value achieved after synthesis.

b) What is the estimated area of the implementation?

When we consider the result from the unconstrained synthesis it becomes clear that a Sklansky ALU can deliver higher performance than a ripple-carry ALU. It would be very interesting to see how `ALU_Skl.vhdl` performs under a stricter timing constraint. Can this be used for our 1-GHz processor?

Assignment 3.1.2:

Synthesize `ALU_Skl.vhdl` with a timing constraint corresponding to the 1-GHz processor.

a) Document the worst-case delay value achieved after synthesis. Does the synthesis tool manage to fulfil the timing constraint?

b) Study the timing report for the worst-case timing path and notice what gates are used. Are there any differences if you compare this critical path with that of **Assignment 2.1.2(b)** and **Assignment 2.1.3(b)**? This question is very relevant since the Sklansky adder is intrinsically fast so there is a good chance that the delays of other ALU logic blocks become timing critical.

c) Generate a timing report where the ten longest signal propagation paths are given. For simplicity, a file dump is useful for this large amount of information

```
report timing -num_path 10 > some_file_name
```

Study in which ALU blocks the ten longest paths are located.

d) What is the estimated area of the implementation?

e) How does the area of the Sklansky ALU scale with stricter timing constraints if you compare to the ripple-carry ALU? Consider drawing a graph with implementation area on the y-axis and timing constraint on the x-axis.

3.2. Verification of synthesized netlist for Sklansky ALU

If we managed to synthesize the Sklansky ALU with a timing constraint corresponding to 1 GHz, we should now also make sure that the netlist generated by RTL Compiler is in agreement with our reference test vectors, by repeating the procedure in **Section 2.2**.

Assignment 3.2.1:

Synthesize `ALU_Skl.vhdl` with the timing goal of **Assignment 3.1.2**. Dump the netlist to `ALU_Skl_netlist.v`, and run a logic simulation on the netlist using `ALU_TB.vhdl` and the reference test vectors to ensure that the implementation is ok.

3.3. Probabilistic power analysis of ALUs

If we assume that we have a need for an ALU that is running at a lower clock rate than 1 GHz, it may be fruitful to reconsider the ALU that employs a ripple-carry adder. After all, the ripple-carry adder is very simple and contains much fewer gates than the Sklansky adder. An interesting question is: Can the ripple-carry ALU compete with the Sklansky ALU in terms of power, when we run them at the same, relatively high frequency? Let us for the sake of example target a clock rate[1] which is the same as that obtained when synthesizing the Sklansky ALU without timing constraint in **Assignment 3.1.1**.

The total switching power — the major power dissipation mechanism — is the sum of the switching power in all nodes $\sum_i f \cdot {V_{DD}}^2 \cdot A_i \cdot C_i$, see Lecture 6. Either the switching activity A_i in this equation is obtained via an estimate based on likelihood of signal switching in node *i*, or it is obtained, more accurately, from actual netlist logic simulations. 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.

In early development phases, test vectors for a block may not be available. To get an estimate on the power dissipation of that block, we can assume — based on experience — some default signal switching probabilities on the primary data inputs, and let the synthesis tool propagate these probabilities through all downstream logic levels, to find A_i. This is called probabilistic power estimation and you can read more on this in **3.2.4/3.1.4 Logic-Level Power Estimation** in Vol 2/Ch 3 of the EDA textbook. (This chapter is available under *Documents/Selected EDA textbook chapters* in PingPong.)

Assignment 3.3.1:

a) Synthesize both `ALU_RCA.vhdl` and `ALU_Skl.vhdl` using a timing constraint that corresponds to the timing resulting from **Assignment 3.1.1**, that is, the unoptimized delay of the Sklansky ALU.

Check the timing report for the ripple-carry ALU and make a note of the timing slack, so you know that this implementation works at the desired cycle time. If the RCA-based ALU does not work with this constraint, increase the cycle time by steps of 5% until both ALUs work.

---

1. We must define a clock when making a power analysis. Otherwise RTL Compiler will have no notion of synchronization and time.

b) Assign the following probabilities on the primary inputs:

1. probability for high logic state on `A`, `B` and `Op` should be 0.5, but 1.0 on `Reset_n`
2. the switching activity A_i on `A`, `B`, and `Op` should be 0.01, but 0.0 on `Reset_n`

The probability for a high logic state on, for example, ALU input ports `A` is set by

```
set_attribute lp_asserted_probability 0.5 /designs/ALU/ports_in/A*
```

while the toggling probability (you need to calculate `togg` from A_i, see Lecture 6) is set by

```
set_attribute lp_asserted_toggle_rate togg /designs/ALU/ports_in/A*
```

We instruct RTL Compiler to propagate the signal probabilities down through the logic by setting an analysis effort

```
set_attribute lp_power_analysis_effort medium /
```

When the probabilities are set, we can generate a power report (via `report power`) and make a note of the total power[1] for the two ALUs and for their main instances, in particular the adder units. Make sure that you understand how total leakage and dynamic power are compiled, by summing the individual power dissipation values.

c) Change the toggling probabilities on the primary inputs, so that the switching activity A_i on `A`, `B`, and `Op` becomes 0.25, but keep the high logic state probability at 0.5. The probabilities of `Reset_n` should remain unaltered. Generate a power report for each of the two ALUs and compare the power values to those of **Assignment 3.3.1(b)**.

d) If we relax the timing constraint of both designs, elaborate on which ALU — ripple-carry ALU or the Sklansky adder — is the more power-efficient one. Carry out a few synthesis runs with varying timing constraints to study power (and area) versus timing constraints, preferably using a graph similar to that of **Assignment 3.1.2**. When evaluating the ALUs for power and area, focus on the adder components; otherwise the adder properties may be overshadowed by other large blocks, such as the shifter block.

e) Retrieve the power dissipated in the clock net for the designs in **Assignment 3.3.1(c)**. The clock net power dissipation is actually not included in the default power report, but to obtain this you must type

```
report power Clk
```

Given the capacitance in the clock node, check that the power is in agreement with the common expression $f \cdot V_{DD}^{\ 2} \cdot C$.

1. *Total Power* is the sum of *Leakage Power* and *Dynamic Power*.

There are several ways to retrieve power data from synthesis. Beside the straightforward `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 below `/designs/ALU/`, such as

```
/designs/ALU/instances_hier/U0/instances_comb/g2752
```

Another option is to go for a power breakdown into the different types of standard 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

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_attribute hdl_track_filename_row_col true /
```

before the VHDL files are read. Invoke RTL Compiler and carry out the synthesis. Assign the appropriate signal switching probabilities, and generate an RTL power report via

```
report power -rtl -detail
```

Static power

When you study the power reports, you may feel that the leakage portion of power is insignificant. Since we are using a Low-Power (LP) cell library, this is however to be expected.

Assignment 3.3.2:

(a) Go back and repeat the steps of **Assignment 3.3.1(a-b).** This time however, only perform one evaluation (that for the ripple-carry RCA) using the following library: `/usr/local/cad/stm-cmos065-5.4/CORE65GPSVT/5.2/libs/CORE65GPSVT_nom_1.10V_125C.lib`. In this library, cells are of General-Purpose (GP) type, that is, of high-speed type. Like before we use Standard-VT devices for nominal process corners, however, the temperature has increased.

(b) Reflect on the lower supply voltage. Why is there no 1.2-V option for the GP libraries?

(c) Consider what has happened to the power values with respect to leakage and dynamic power dissipation.

3.4. Simulation-based power analysis of ripple-carry ALU

In the previous section we assumed random signal probabilities, which represent input test vectors that have no particular pattern but basically represent white noise. As we will see in the following section, the power estimates can be made much more accurate if actual signal activities obtained from real logic simulations are used. We call this simulation-based or use-case-based power analysis.

The flow for obtaining signal switching statistics from netlist logic simulation starts with the following:

```
‘read libraries’
‘read VHDL files’
elaborate
‘define timing constraint = constraint of Assignment 3.3.1 and 3.3.2’
synthesize -to_mapped
‘check so that timing is ok’
write_hdl > netlist.v
```

Nothing new so far. Now we take a break in using RTL Compiler; don’t exit the program, though[1].

Switch to another shell to work with NCSIM. We will do something that is very similar to the verification that was described in **Section 2.2**: We read in and compile the standard-cell libraries, our recently created netlist and the testbench. Then we elaborate the designs and carry out the simulation. However this time our intention is to transfer the actual signal switching statistics, which result from the use of test vectors in a logic simulation of a circuit, to RTL Compiler in order to calculate the true power dissipation.

There are two ways to do perform this test-vector based analysis. Either a VCD (Value Change Dump) or a SAIF (Switching Activity Interchange File) file is generated, and later backannotated into RTL Compiler to drive the power analysis. A VCD file contains information on all signals that toggle, for every clock cycle. As a result, VCD files can grow huge. The advantage of VCD files is that the power dissipation can be calculated for any cycle, allowing for identification of the cycle with the highest power dissipation and subsequent analysis of power dissipation for a smaller time window of the entire simulation[2]. In contrast, the SAIF file contains only the average switching activity of all nodes.

We will create two scripts to enable the generation of VCD and SAIF files. First we define the VCD script called `ncsim_VCD.tcl`

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

after which we define the SAIF script `ncsim_SAIF.tcl`:

```
dumpsaif -hierarchy -output ALU.saif -scope instance_name
-overwrite -verbose
run
dumpsaif -end
```

1. Another alternative is to issue Linux commands from within RTL Compiler: Simply add `shell` in front of the Linux command. This is a viable approach if you have already prepared the two `.tcl` scripts that are needed.
2. As an example of the latter, we could calculate the average power for a time window of 100 - 200 ns from a VCD file that has been generated for, say, 0 - 10,000 ns.

To generate the VCD file, we must perform a logic simulation that makes use of `ncsim_VCD.tcl`

```
ncvlog /usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/5.2.c/behaviour/
verilog/CORE65LPSVT.v

ncvlog netlist.v

ncvhdl -v93 ALU_TB.vhdl

ncelab -messages -v93 -access +rwc -timescale 1ns/1ps -delay_mode zero
WORKLIB.ALU_TB:BEHAVIORAL

ncsim -input ./ncsim_VCD.tcl WORKLIB.ALU_TB:BEHAVIORAL
```

After the simulation finishes, a VCD file should have been generated.

In order to generate the SAIF file we will run the same simulation, but `ncsim_SAIF.tcl` is used as input to NCSIM, like this

```
ncsim -input ./ncsim_SAIF.tcl WORKLIB.ALU_TB:BEHAVIORAL
```

Now, if we copy the recently produced VCD and SAIF files to the directory in which we saved the netlist — and where RTL Compiler is still running — we can take the final steps to the conclusion of this power analysis. First we may use the VCD file to obtain the actual power dissipation[1]

```
read_vcd -static -vcd_module instance_name ALU.vcd

report power
```

after which we switch our attention — inside the same RTL Compiler session — to the SAIF file

```
read_saif -instance instance_name ALU.saif

report_power
```

In case you did exit RTL Compiler, you need to set up it again, in order to read the switching information:

```
set_attribute library {{ /usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/
5.2.c/libs/CORE65LPSVT_nom_1.20V_25C.lib }}

read_netlist netlist.v

define_clock -name main_clk -period clock_period [find / -port Clk]
```

Assignment 3.4.1:

a) According to the flow above, create a VCD file for the ripple-carry ALU assuming the test vectors in `tvs_random.zip`. Read the VCD file into RTL Compiler and generate a power report.

b) Create a SAIF file for the ripple-carry ALU assuming the test vectors in `tvs_random.zip`. Read the SAIF file into RTL Compiler and make a power report.

Study the power reports generated in the VCD and SAIF flows, and confirm that the power values are similar. Also, check the sizes of the VCD and SAIF files. What can be said about the file sizes?

---

1. Doublecheck the messages from NCSIM. The asserted nets must be close to 100%. If this number is far away from 100%, then it is likely something is wrong.

c) RTL Compiler can present signal statistics in the Toggle Count Format (TCF). The TCF data contains, 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. Save the TCF signal statistics by

```
write_tcf > alu_random.tcf
```

Consider the statistics in `alu_random.tcf` that relate to the clock signal and explain what information a TCF file provides.

d) Create a SAIF file for the ripple-carry ALU assuming the test vector in `tvs_realtrace.zip` which you can find under *Documents/Lab exercises*. These (233,450!) test vectors — 233,451 since there is a dummy line at the end — are the real trace from an architectural simulation of a MIPS processor executing the Viterbi algorithm in which the ALU is used extensively. Let RTL Compiler read the SAIF file and make a power report.

Compare this SAIF power report with that you produced in **(c)** and reflect on the differences.

e) Save the TCF statistics for the real trace simulation by

```
write_tcf > alu_realtrace.tcf
```

Consider the TCF signal statistics for the signals `A[15]` and `A[16]`.

Compare these values with those obtained from the random test vectors; see **(c)** above. Specifically, what is the correspondence between TCF signal statistics and the actual test vector data of the `A.tv` files?

3.5. Learning outcome of Exercise 3

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

- describe what is timing closure in the context of a synthesis flow.
- perform a design respin in an ASIC design flow to fulfil timing requirements, by going back to the original hardware description code before synthesis and replace a slow block with one that is inherently faster.
- perform power analysis using information on signal switching.
- perform power analysis using backannotation of logic simulations of real use cases.
- describe how stricter timing constraints impact area and power dissipation.
- describe how clock rate and signal toggling probability (*togg*) impact power dissipation.
- describe how signal toggling probability (*togg*) relates to switching activity (A_i).

Exercise 4: ALU Place and Route (SW 6)

It takes considerable experience to make good place-and-route implementations of synthesized netlists. It is necessary to have in-depth understanding of many peculiarities of the EDA tools, such as knowing what fields should be filled out, what boxes should be checked and, not least, in which order to use the different commands. Since we have a limited time period for trying out the tools in the ASIC design flow, the following exercise significantly constrains your design freedom. However, we encourage you to try out different options and, above all, to take your own ALU design through the flow.

4.1. Starting Cadence Encounter

Before you start Encounter,

1. create a directory called `ALU_RCA_Encounter`[1].
2. download the zipped file archive `encounter_files.zip` from PingPong and unzip the file in the `ALU_RCA_Encounter` directory, in the process creating three new subdirectories containing a number of files.

The `netlist` directory holds the Verilog netlist (`ALU_RCA.v`) of a predesigned and synthesized ALU. We encourage you to instead use your own RCA-based ALU[2] and only use our predesigned ALU as reference.

The `inputs` directory contains a script (`ALU_RCA.conf`) to initialize our place-and-route (P&R) session, a file to define analysis mode (`mmmc.tcl` which refers to Multi-Mode Multi-Corner), and a script (`ALU_RCA.sdc`) to define the timing requirement of 3.3 ns on the P&R phase.

There also exist a few other scripts, in the directory `scripts`, and these will be referred to later in this lab exercise.

Now we start Encounter by typing

```
encounter
```

at the Linux command line.

Whenever we wish to take a break, we can save our current design by choosing *File -> Save Design* (using the Encounter Data Type)..., and later restore it by choosing *File -> Restore Design* ...

Importing netlist and process technology information

In the menu *File*, choose *Import Design* ... It is clearly possible to define an array of information about netlists, LEF-libraries, timing, power, etc., but it is more convenient to have a start file that contains all that information. As hinted above, a start file has been prepared for you: Choose *Load* ..., select `inputs/ALU_RCA.conf` and press Open to obtain the configuration data. Spend some time to study the configuration data and other options of the form, and then click OK to load the prepared design.

4.2. Floorplanning the design

Several settings pertaining to general floorplanning issues already exist in `ALU_RCA.conf`, such as aspect ratio and core to I/O boundaries, that is, the area used for power rings.

1. Before you start Encounter, make sure you are in the proper file directory. While it is possible to use Linux commands like `ls` and `cd` (change directory), use of the latter from inside Encounter may mess up your file hierarchy.
2. During ALU synthesis using RTL compiler, you can produce not only the netlist file, using the correct timing constraint of 3.3 ns, but also a constraint file (.sdc): `write_sdc > ALU_RCA.sdc`.

The interesting part of this step, and what can make a difference for the design quality, is the preplacement of modules/blocks — the floorplanning phase. The human designer may have an overall picture of how to best organize the gates of a design, especially if it is a regular design such as a datapath component. These floorplanning preferences can in fact be communicated to the EDA tool.

Encounter will automatically identify blocks that reflect the module organization of the RTL coding — as far as the reference, `ALU_RCA.v`, is concerned, LOGU, Shift and ADDSUB would be the modules. To show the principle of guiding the cell placement, we will first further partition `ALU_RCA.v` and then propose where these blocks should be placed to enable an efficient structure for the standard-cell P&R phases.

Partitioning

Choose *Tools -> Design Browser* ... Now we can see all the entities that have been instantiated at the top level of the actual ALU. We see terminals, nets, different gates (*StdCells*) and a few bigger blocks (*Modules*). When you expand *Modules (3)*, you discover three blocks called 'U0', 'U1', and 'U2'. Each block is an instance of the module in the RTL code; for example, 'U0' is an instance of the logic module.

Before we preplace any blocks, we will group together the pipeline registers so that the overall design can be made more efficient.

1. Expand *StdCells (148)*, and highlight all cells starting with *A_reg*, *B_reg* and *Op_reg*. You can either highlight the cells manually or you can use Find Instance and filter out the appropriate cells by using, for example, 'ALU*' as search string.

   When you have highlighted the cells, left-click Select (the icon in the Design Browser window depicting an arrow). Leave the Design Browser window open ... perhaps you want to move it to the side of the main window.

2. Choose *Floorplan* (in the main window) -> *Instance Group -> Create New Group* ... and give, for example, 'INPUTREG' as Group Name[1] in the Group Instances form. Click OK.

3. **After you have finished creating the new block, make sure you click the *Deselect All* icon inside the design browser!**

Now you should repeat steps 1-3 above for the output registers (those called *Outs_reg*): Use, for example, 'OUTPUTREG' as Group Name. You can now check that the two groups that you have created are presented in the Design Browser, under *Groups*.

Floorplanning

The blocks that you are about to preplace are located to the left on the screen, assuming you use Floorplan view[2], so either zoom out or scroll the screen using the left arrow key to get a view of them.

To move a block into the target placement area — the core that is represented by the rectangle that has a green boundary — you left-click on the Move icon depicting a four-arrow cross (located to the right of the main window's Select arrow). Move the cursor and left-click once on the block that you intend to move. Now you should be able to drag this block into the target placement area. You release the block by left-clicking once more[3].

1. In this Group Instances dialog box, you can also specify the target area utilization (cell density) of the design. This value will, however, change later anyway, so we refrain from giving any.
2. If a module contains less than 100 cells, the threshold for what Encounter considers a block worth showing must be changed: *Options -> Set Preference ... -> Display -> Min. Floorplan Module Size*.
3. If the *Fence* option would have been selected for a specific block, you'd had to resize the blocks to avoid area overlaps.

Organize the blocks according to data flow; for example, if you let data enter the ALU on the left side, you should place the 'INPUTREG' block to the left. The floorplanning step is really important so do not rush this phase. Hopefully you have time to try out a number of different floorplans, not only changing the block positions, but also changing the size and aspect ratio of the blocks.

Notice the area utilization of each block. This number is given after 'TU=' and should never be above 90% to allow the tools to operate efficiently.

Select floorplanning method

Before we move on to the next stage in the P&R flow, we need to make sure that our block preplacement will be interpreted correctly by Encounter.

When inside the main window, make sure your cursor is in Select mode (left-click on the arrow icon). Left-click on any of the blocks inside the core placement area. Now left-click the light-blue, round icon containing an "i" (this is the Attribute Editor). A new dialog box should appear. Make sure that the field Constraint Type represents the *Guide* option; this option tells the placement tool how to interpret the preplacement of our blocks. By using the *Guide* option, we tell Encounter to try to place the standard cells within the block. However, if the tool considers this to be a particularly poor idea, it has the freedom to place some cells outside the block boundaries. If we would have used the *Fence* option instead, all cells (no exceptions!) have to be placed within the specified block area.

Pin placement

Once the blocks have been preplaced it is time to perform placement of pins. Choose *Edit -> Pin Editor* ... You should be able to see `Clk`, `Reset_n`, `A[]`, `B[]`, `Op[]`, `Outs[]` in a list.

Now select `A[]`, `B[]` and `Op[]`. These inputs are supposed to enter the ALU design, for example, at the top or to the left. Consequently, after choosing *Along Entire Edge* as Spread option, select Side/Edge as *Top* or *Left*, depending on your design. Also, check the option *Assign Fixed Status* (meaning that the pin cannot be moved by the tool). Click Apply, and repeat the same procedure for `Outs[]`, with the difference that these are placed at a different side ... you choose which. `Clk` and `Reset_n` can be left unplaced.

If you zoom in on the periphery of the target placement area, you will now see small triangles where the pins have been placed. If you click on a pin, information on the pin is given at the bottom of the main window.

4.3. Power routing

The next step in the design flow is to route the power distribution network (PDN).

Power rings

We first add power rings around the core: Choose *Power -> Power Planning -> Add Ring* ...

1. In the emerging window, make sure that the field *Net(s)* contains two net names (gnd and vdd). You click ... to open the Net Selection window. Here you pick vdd and gnd from Possible Nets (defined in `ALU_RCA.conf`) and add these to Chosen Nets. Click OK.
2. Still in the Basic tab, make sure the width of the rings in Ring Configuration is 2.5 μm for all four sides. Assign 1 μm for Spacing, and specify the Offset to be 2.5 μm[1].

1. We have to make sure that ((Width * 2 + Spacing) * Number of Bits + Offset) is smaller than 10 μm, which is the distance between core and IO, as defined in `ALU_RCA.conf`.

3. Move to the Advanced tab, and select *Use wire group*. To obtain one pair of ground and supply wires, select *Interleaving* and enter '1' as *Number of bits*.

You are now done with the Add Rings form and can click OK.

To study the resulting features, you may want to try the Query button at the bottom of the design window (the button marked *Q*), as an alternative to the Attribute Editor. When the Query feature is enabled you only need to move the cursor on top of an object to get information on it; for example, what metal layers are used in the horizontal and vertical directions are easily found out this way.

Power stripes

For larger designs, the power rings must be complemented by vertical power stripes, because a weak PDN may otherwise result in IR drops and electromigration. We have not carried out any analysis regarding what DC currents the PDN needs to deliver, so all rings and stripes of this exercise are chosen in a conservative manner, most likely leading to an overdesign of the PDN.

Choose *Power -> Power Planning -> Add Stripe* ... Select vdd and gnd for net. Other options and values are as follows: *Direction* should be Vertical, *Width* should be 2.5 μm, *Spacing* should be 1 μm, *Set-to-set distance* should be 50 and *X from left* should be 40. After you click OK, one vertical stripe set will appear.

Cell power wires

To make the power routing for the standard-cells rows, we use the script below (it is available inside `encounter_files.zip` as `cell_power_routing.script`). Before you administer the script, by entering it on the Encounter command line using `source`, it is a good idea to switch the presentation style from Floorplan view to Physical view.

```
globalNetConnect vdd -type pgpin -pin {vdd } -inst * -module {}
globalNetConnect gnd -type pgpin -pin {gnd } -inst * -module {}

applyGlobalNets

deselectAll

cutRow

sroute -connect { blockPin padPin padRing corePin } \
-layerChangeRange { M1 AP } \
-blockPinTarget { nearestRingStripe nearestTarget } \
-padPinPortConnect { allPort oneGeom } \
-checkAlignedSecondaryPin 1 -blockPin useLef -allowJogging 1 \
-crossoverViaBottomLayer M1 -allowLayerChange 1 \
-targetViaTopLayer AP -crossoverViaTopLayer AP \
-targetViaBottomLayer M1 -nets { gnd vdd }

deselectAll
```

Now there should exist metal lines along all standard-cell rows.

4.4. Standard-cell placement

We are now ready to place the cells of the design. Choose *Place -> Place Standard Cell* ... Select the *Include Pre-Place Optimization* option, but deselect *Include In-Place Optimization* to avoid violations during the placement phase. Click on the *Mode* button and make sure *Run Timing Driven Placement* has been selected. Now run full placement by clicking OK in both windows.

We will now study how the cells were placed.

* The *Check Placement ...* option in the Place menu gives some information, such as number of cells placed and unplaced, and the present placement density (uncheck file output if you want to read this data at the command line). If the placement check reports any violations, try the following: *Place -> Refine Placement* … followed by a new placement check.

* Check if the placement has followed your preplacement constraints. How? Well, inside Design Browser, select a block and monitor what cells are highlighted in the Physical view mode. You may want to turn off Net in the Layer Control panel, to focus on the cells. By pressing Ctrl-R you refresh the screen.

* With the Query feature still enabled, you can move the cursor over the cells and gather information on them. Alternatively you can zoom in on a cell and left-click to get the information in the Attribute Editor window. The current status of the cells is PLACED, but you can actually change this to FIXED if you do not want the placement tool to change cell location as further optimizations are done.

* Use the command

  ```
  report_timing
  ```

  to obtain the critical path report. Are you fulfilling the timing constraint?

4.5. Clock Tree Synthesis (CTS)

Before you perform the steps inside the Clock Tree Synthesis (CTS) phase, make sure you use the Physical view to study what network the Clk pin drives (open *Nets* in Design Browser to find `Clk`). Turn off (uncheck) Net visibility in the Layer control panel on the main window in order to better see the route.

As the CTS steps are progressing, study how the clock network changes.

Pre-CTS optimization

For the ALU design you can run the following commands to perform pre-CTS optimization:

```
setOptMode -fixCap true -fixTran true -fixFanoutLoad false
optDesign -preCTS
```

Study the command line to find information (for example, ***Finished optDesign***) on the optimization process. Note the way the Physical view changes after the optimization is completed. To improve the synthesis quality, the CTS tool needs information about wires, so a so-called trial routing was performed.

The actual CTS step

To specify clock tree settings, first run

```
setCTSMode -engine ck
```

followed by

```
clockDesign -genSpecOnly Clock.ctstch
```

to create a CTS specification file. This command converts the constraints in `ALU_RCA.sdc` into the format used by the Encounter Timing System (ETS) tool. Compare `Clock.ctstch` to the SDC file.

Now choose *Clock -> Synthesize Clock Tree ...* Under the Basic tab, type `Clock.ctstch` in the Clock Specification Files field. Now, click the Mode button. On the Mode Setup form, go to the Route/EM tab and select both Route Clock Nets and Route with Nanoroute Guidance. Click OK. This step allows you

to route the clock nets, while the clock tree gate netlist is being set up. Click OK on the Synthesize Clock Tree form to start CTS and routing.

What happens to the Clk net after CTS? View the complete clock network after CTS, in the Physical view mode, by choosing *Clock -> Display -> Display Clock Tree ...* and selecting *Post-Route* for Route Selection and *Display Clock Tree* / *All Level* for Display Selection. Click OK.

Describe the clock tree generated by CTS, and consider the positions of clock buffers and clock nets. You can use the Design Browser window to identify net names etc.

Check the timing and compare this to the result of pre-CTS optimization.

When you are done, turn off the clock tree display via *Clock -> Display -> Clear Clock Tree Display.*

Reset net routing

Before proceeding to the post-CTS optimization let us also route the `Reset_n` net. (Why are clock and reset nets given priority over the other nets?). In the Design Browser, change the Find tab to Net from the drop down menu and type 'Reset*' and hit enter. Select the `Reset` net from the results which appear (as in the case of floorplanning, selection needs to be done explicitly using the icon in the Design Browser window depicting an arrow).

Choose *Route -> NanoRoute -> Route ...* In the form that opens, click Mode at the bottom. In the Mode Setup form, select Route Selected Nets Only. Under the Timing/SI tab, select Timing Driven. Click OK, first in the Mode form, then in the Routing form.

After routing, the status of the design should now change to Routed. The routed nets should be visible in the Physical view. First press Deselect All in the Design Browser, and then view the routed `Clk` and `Reset_n` nets by choosing the appropriate net in the Design Browser. Are all nets routed?

Post-CTS optimization

It is possible to make more placement optimizations after the clock network has been generated. Choose *Optimize -> Optimize Design ...* Use Post-CTS as Design Stage. Start the optimization by clicking OK

Check the timing and compare this to the result after CTS.

4.6. Routing

Now we will route the entire design. Choose *Route -> NanoRoute -> Route ...* and choose the Mode Setup. Make sure that the option Route Selected Nets Only is deselected and that the option Timing Driven is selected, then click OK. Click OK in the Nanoroute form.

Study your Physical view carefully and find out what routing has been performed. Click on `Reset_n` and `Clk` in the Design Browser and check these in particular.

Make sure you check the timing after routing to see if the specification of 3.3 ns has been fulfilled.

4.7. Final layout fixes

To make sure the layout is manufacturable, there are a few more things that we need to fix.

Filler cells

We can add filler cells to fill the gaps between already placed cells:

```
set FILLER_CELL_LIST { HS65_LS_FILLERCELL1 HS65_LS_FILLERCELL2
HS65_LS_FILLERCELL3 HS65_LS_FILLERCELL4 HS65_LS_FILLERNPW3
HS65_LS_FILLERNPW4 HS65_LS_FILLERNPWPFP16 HS65_LS_FILLERNPWPFP3
HS65_LS_FILLERNPWPFP32 HS65_LS_FILLERNPWPFP4 HS65_LS_FILLERNPWPFP64
HS65_LS_FILLERNPWPFP8 HS65_LS_FILLERPFOP12 HS65_LS_FILLERPFOP16
HS65_LS_FILLERPFOP32 HS65_LS_FILLERPFOP64 HS65_LS_FILLERPFOP8
HS65_LS_FILLERPFOP9 HS65_LS_FILLERPFP1 HS65_LS_FILLERPFP2
HS65_LS_FILLERPFP3 HS65_LS_FILLERPFP4 HS65_LS_FILLERSNPWPFP3
HS65_LS_FILLERSNPWPFP4 }

setFillerMode -core $FILLER_CELL_LIST -corePrefix FILLER -fitGap true -
minHole true -viaEnclosure true -doDRC false -ecoMode false

addFiller

setFillerMode -reset

setFillerMode -core $FILLER_CELL_LIST -corePrefix FILLER -fitGap true -
minHole true -viaEnclosure true -doDRC true -ecoMode true

addFiller

globalNetConnect vdd -type pgpin -pin vdd -inst * -module {}

globalNetConnect gnd -type pgpin -pin gnd -inst * -module {}

applyGlobalNets
```

The above commands are given in `add_filler.script`.

Post-route optimization

Optionally we can try to use post-route optimization. After issuing the following mode command

```
setAnalysisMode -analysisType onChipVariation
```

we choose *Optimize -> Optimize Design ...*, and select Post-Route as Design Stage.

Layout verification

In the *Verify* menu, choose *Verify Geometry*, then *Verify Process Antenna*[1], and finally *Verify Connectivity*. Study the outcomes after each step on the command line. We may encounter some geometry errors. Use *Tools -> Violation Browser ...* to study these errors in detail.

4.8. Power analysis

So far Encounter has not considered the actual wires that result from the routing phase; instead Encounter has been using a wire load model to estimate wires. In order to accurately capture the wire information, we need to perform a layout extraction of wire capacitances and wire resistances:

```
extractRC

rcOut -spef ./ALU_netlist.spef

saveNetlist ./ALU_netlist.v
```

1. The process antenna issue relates to long polysilicon wires. These wires may act as antennas and collect charges during plasma etching, building up a potential that may destroy the gate oxides.

Save your design, exit Encounter and start RTL Compiler. Set up the process technology used in `ALU_RCA.conf` and read in the netlist:

```
set_attribute library {{
/usr/local/cad/stm-cmos065-5.4/CORE65LPSVT/5.2.c/libs/
CORE65LPSVT_nom_1.20V_25C.lib
/usr/local/cad/stm-cmos065-5.4/CORL65LPSVT/5.2/libs/
CORL65LPSVT_nom_1.20V_25C.lib
/usr/local/cad/stm-cmos065-5.4/CLOCK65LPSVT/3.2/libs/
CLOCK65LPSVT_nom_1.20V_25C.lib
}}
read_netlist ALU_netlist.v
```

In the following we will compare the power dissipation of the P&R implementation *without* parasitics with that of the P&R implementation that *includes* extracted wire parasitics. In order to set up the power analysis, we use a conservative clock period of 5 ns:

```
define_clock -name main_clk -period 5000 [find / -port Clk]
```

Then we assign signal properties to the primary inputs:

```
set_attribute lp_asserted_probability 0.5 /designs/ALU/ports_in/A*
set_attribute lp_asserted_probability 0.5 /designs/ALU/ports_in/B*
set_attribute lp_asserted_probability 0.5 /designs/ALU/ports_in/Op*
set_attribute lp_asserted_probability 1 /designs/ALU/ports_in/Reset_n
set_attribute lp_asserted_toggle_rate 0.03 /designs/ALU/ports_in/A*
set_attribute lp_asserted_toggle_rate 0.03 /designs/ALU/ports_in/B*
set_attribute lp_asserted_toggle_rate 0.03 /designs/ALU/ports_in/Op*
set_attribute lp_asserted_toggle_rate 0 /designs/ALU/ports_in/Reset_n
```

As before, make sure you use the appropriate names for your design, for your clock and other signals.

We can now generate a power report for the netlist that was previously loaded:

```
report power > power_without_RC.txt
```

Next we read in the extracted RC values from the spef file and generate a power report for the netlist which now is associated with the extracted parasitics:

```
read_spef ./ALU_netlist.spef
report power > power_with_RC.txt
```

Consider the two power reports. Do you see any differences in overall dynamic power dissipation? Can you identify any block for which the RC-extracted power dissipation is increasing more than for the other blocks? If so, what could be the reason that the impact of parasitics on power dissipation varies with block type?

4.9. Timing closure

If the implementation did not satisfy the timing constraint in **Section 4.6**, restart Encounter and load the configuration file (`ALU_RCA.conf`) again. This time, before going through the whole P&R procedure, from the *Optimize* menu click *Optimize Netlist ...* followed by OK. Watch the Encounter command window and its optimization steps.

When the optimization is finished you might not be able to distinguish your modules anymore. This can happen if a unit is small. If you monitored the optimization sequence carefully, you may have noticed the hierarchical boundary optimization in the beginning. The tool may merge your small units with the ungrouped design (like output multiplexer) in order to increase floorplanning/ placement efficiency[1].

Now you are ready to follow the P&R steps again. Use the same placement strategy you used before and do the all the succeeding steps again. Finally check your timing. Has anything significant happened to the timing that you obtain? If so, what can be the causes of this difference?

4.10. Learning outcome of Exercise 4

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

- describe the main features of a place-and-route flow; floorplanning, placement and routing, including optimization at different stages.
- describe the connection between custom layout, taught in circuits courses, and placement and routing of standard cells.
- perform the basic steps of place and route.
- describe what steps the place-and-route software handles efficiently.
- describe what steps the place-and-route software does not handle efficiently, and for which the designer’s understanding helps to improve design significantly.
- elaborate on the importance of extracting wire properties from the final layout.

1. After the placement is finished, you can mark (in the design browser) the module that has disappeared and you will see that the cells for that part of the design are placed.