# Lab1: Counter Simulation and Synthesis

Duration: roughly two sessions

## 1 Overview

The goal of this first lab work is to get along with the tools and scripting environments that are encountered in digital designs.

We just want here to write a very simple circuit, like a counter, along with a testbench, and synthesize it to the UMC65nm technology.

What you should learn during this lab work:

- ✓ Get along with the Linux command line
- ✓ Get along with simple TCL scripts
- ✓ Write a simple Verilog/VHDL circuit
- ✓ Simulate the counter using Cadence INVISIV Simulator
  - Write a testbench
  - Configure it to run and stop in case of errors
- ✓ Synthesize the circuit
  - Use Cadence RTL Compiler
  - Understand the TCL script driving the process
  - Understand the standard cells selection
  - Write Timing Constraints
  - Perform 0-wire delay synthesis and physical synthesis
  - Analyse timing
  - Try synchronous and asynchronous resets

### 1.1 Where should I start?

- If you have never used the Linux environment, please work through the Bash Command Line from 2.1
- The Tools chapter presents the documentation paths for the design kit and the tools
- The assignments start with 4 Counter Simulation.

# 2 Prerequisite

## 2.1 Bash Command Line

**If you are not comfortable with the Linux Command Line**, you can start by getting comfortable with it. The website http://linuxcommand.org/ offers some tutorials, you can also just use google.

You should be able to navigate through the filesystem, create folders, and understand the "source" command.

## 2.2 TLC Scripting

The synthesis and place and route tools are driven by scripts written in the TCL Language. This language is easy to learn, and it can be very useful as the scripts driving the tools can become complex.

A Tutorial on the official Tcl community website is available:

http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html

The reference documentation of the basic commands can be found on the same website:

http://www.tcl.tk/man/tcl8.5/

# 3 Tools

## 3.1 Loading the tools (same as DaS Übung)

To use the tools needed during the lab work, you just need to source a bash script in a terminal. This script will change the terminal's environment so that the various needed applications can be started.

**!! Note that you have to execute this command every time you open a new terminal !!**

**!! This command does not start any application, it just enables you to access the software commands presented later !!**

```
source /var/autofs/cadence/umc_65.sh
```

## 3.2 Documentation

### 3.2.1 Verilog/System Verilog

Keep the System Verilog reference next to you to find out about the allowed language elements, both for synthesisable logic and simulation code.

http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

### 3.2.2 Target Technology

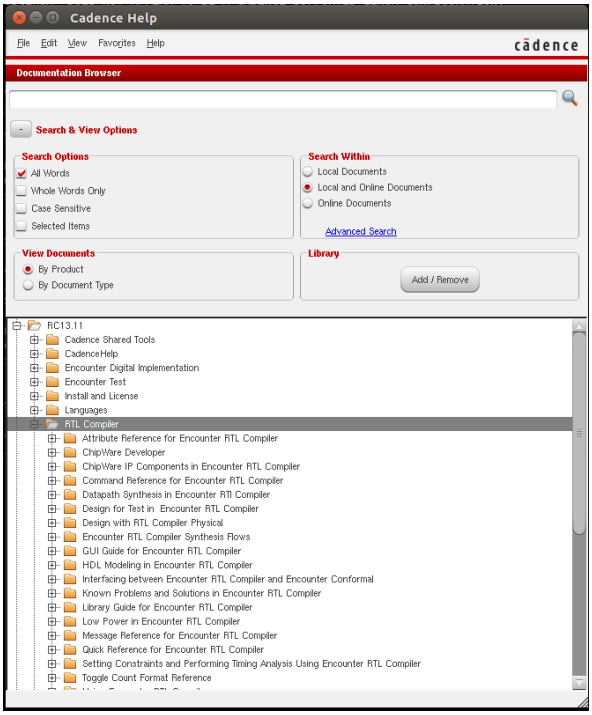
All the designs will be synthesised and mapped to the UMC65 process, meaning it is a 65nm technology by the vendor UMC.

The documentation is available under /var/autofs/cadence/umc-65/

| Documentation | File |
|---|---|
| Standard Cells | 65nm-stdcells/doc/databook.pdf |
| Input Output Cells/Pads | 65nm-pads/doc/databook.pdf |
| | |

### 3.2.3 Cadence Tools Documentation

After loading the tools, you can run the “cdnshelp” command. This will open a window with the software documentation. This is very useful for the RTL compiler and EDI tools, and contains a lot of application notes, as well as a reference documentation of the programming interfaces.




# 4 Counter Simulation

## 4.1 Organising the work folders

The various tools we are going to use usually produce a lot of output when running. It is thus a good idea to work in different folders to avoid conflicts and lose sight of the data.

To start, create a folder structure like this one:

- ✓ common : Put the scripts common to all lab work here
  - synthesis: Some synthesis scripts will be common to all work
- ✓ lab1
  - sources : Put the Verilog sources in this folder
  - simulation : Put the test bench sources in this folder, and run the simulation from there
  - synthesis: Put the synthesis scripts in this folder
  - run : Create a subfolder here for each implementation run
    - synthesis : Run synthesis from this folder
    - encounter: Run place and route from this folder

Just reuse this folder structure for each new design. You can also use your own folder scheme, the basic idea is to avoid mixing design data and tools outputs.

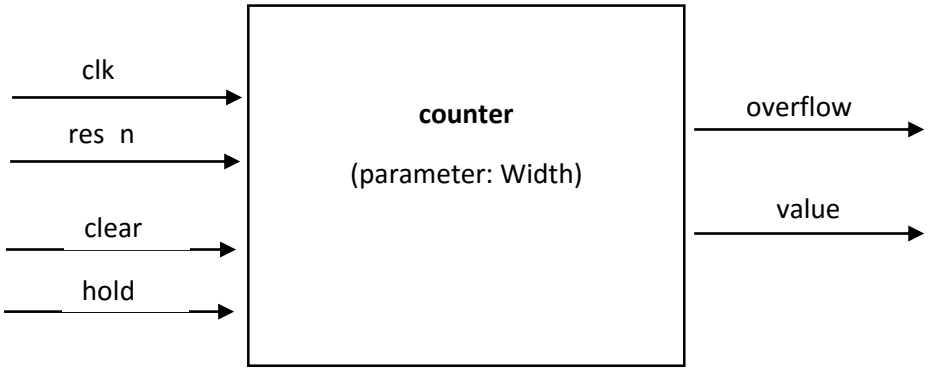
## 4.2 Open A Text Editor

To edit your scripts and source files, you have to use a simple text editor. You look for a tool called "gedit":

- Look in the start menu under "editors" or "development" or "programming"
- If you don't have a start menu, press the Windows key, then type "gedit" on the keyboard

## 4.3 Specification (see lecture for a reminder)

Write a Verilog description of a simple counter matching the following specification. Write the module in a file called "counter.v", and save it under the sources folder.




| Name | Width | Description |
|---|---|---|
| Parameters | | |
| WIDTH | - | Number of counter bits (size of value) |
| Inputs | | |
| clk | 1 | Main clock |
| res_n | 1 | Negative Reset, active when 0 |
| hold | 1 | Stops counting |
| clear | 1 | Resets counter to 0 |
| Outputs | | |
| value | WIDTH | counter output |
| overflow | 1 | 1 for one cycle when the counter overflows (during the next 0 value cycle) |
| | | |

## 4.4 Testbench for 8 bit counter

Write a Verilog module called “tb_top”, which instantiates the counter with a WIDTH of 8. Write the testbench in a file called “counter_tb.v”, and save it under the simulation folder.

Setup the testbench with a reset sequence and a clock generator. See the Verilog introduction lecture for details.

### 4.4.1 Design verification

To verify your counter is working, instead of manually checking validity in the waveforms, you can write your testbench so that it will produce an error (using the $error simulator function).

This is a good idea to do so, so that when modifying the Verilog source to improve the synthesis and place and route, you can check the design is still valid easily without reopening the whole graphical environment.

## 4.5 Simulation

### 4.5.1 Opening the simulation environment

To run the simulation, we are going to use a set of tools from the Cadence INCISIV suite. They are invoked through a single command called *irun*.

To simulate the design, we must pass the design files to the simulator (counter.v, counter_tb.v), along with some configuration arguments.

To do so, create a file named *counter.f* inside the simulation folder.

This F-File contains one argument per line, which can be:

- ✓ The path to a source file
- ✓ An argument for the simulator

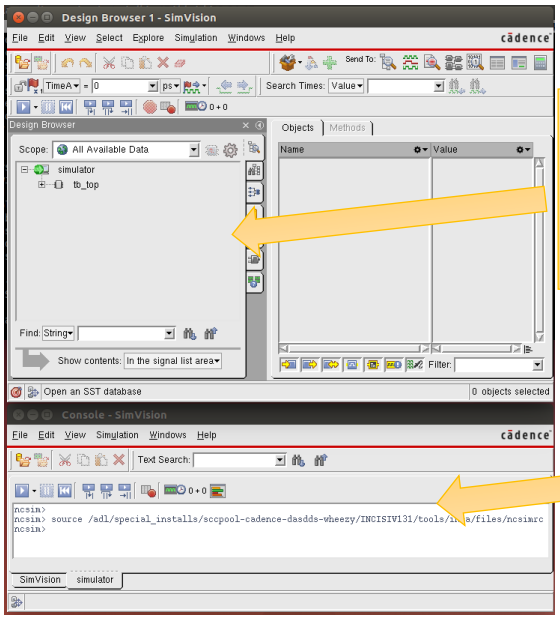
The F-File for this simulation contains the following lines (remember we run the simulation from the simulation folder):

```
../sources/counter.v

counter_tb.v

-access r+w
-sv
-timescale 1ns/10ps
```

Now, to run the simulator, just enter at the command line:

***irun –f counter.f –gui***

The "-gui" argument launches the SimVision environment, which allows browsing the design, running the simulation and viewing the signal waveforms.




Design Browser

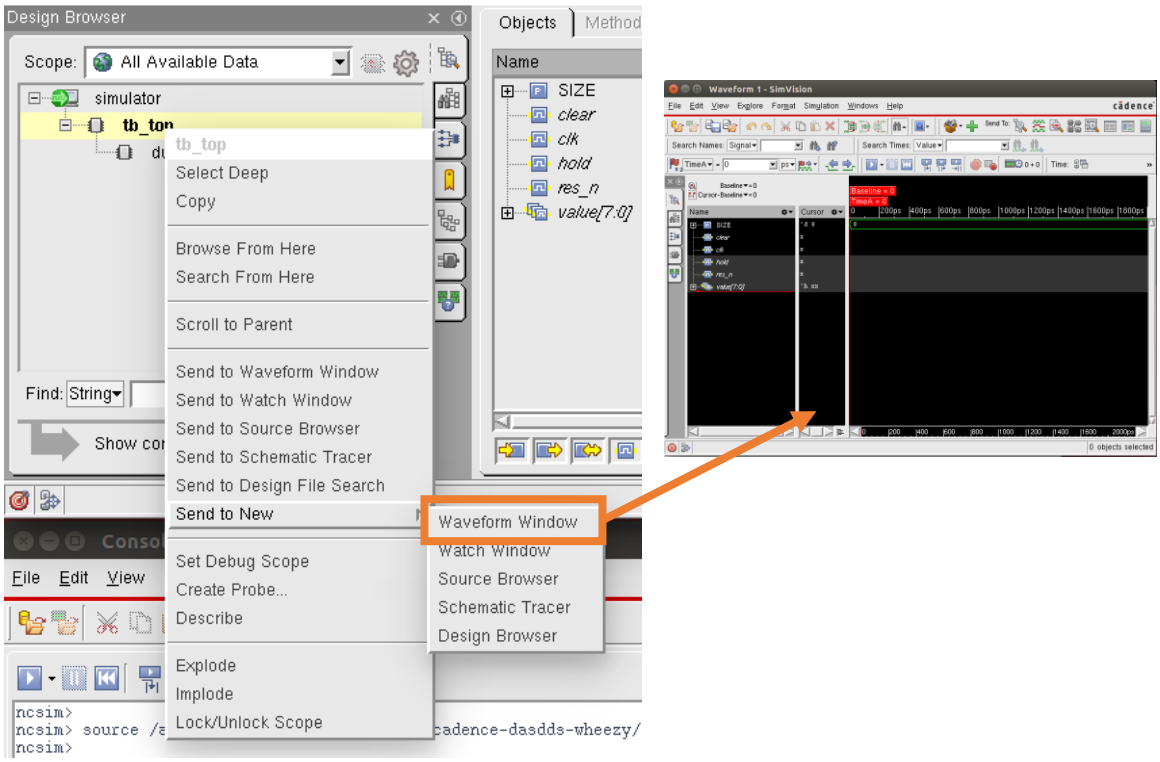
See design hierarchy, variables and open the exploration tools (waveform etc…)

Control Console

Enter some commands, see simulator text output, but globally not used much

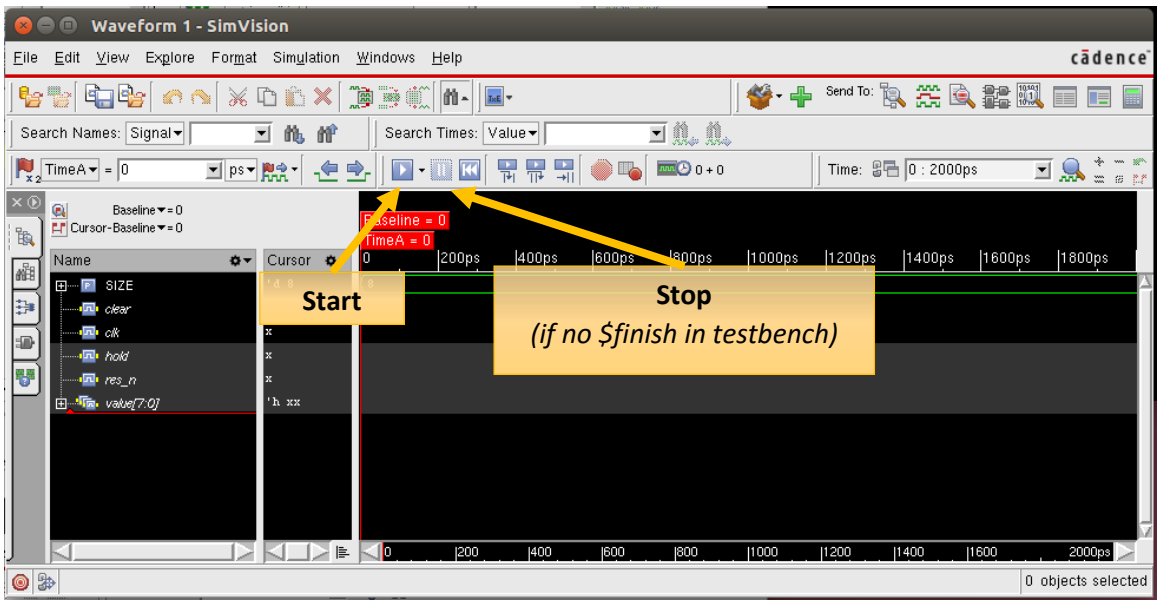
### 4.5.2 Opening the waveform

To open a waveform viewer, right-click on a design hierarchy level on the design browser, and select “Send to New -> Waveform Window”, an empty waveform window will appear



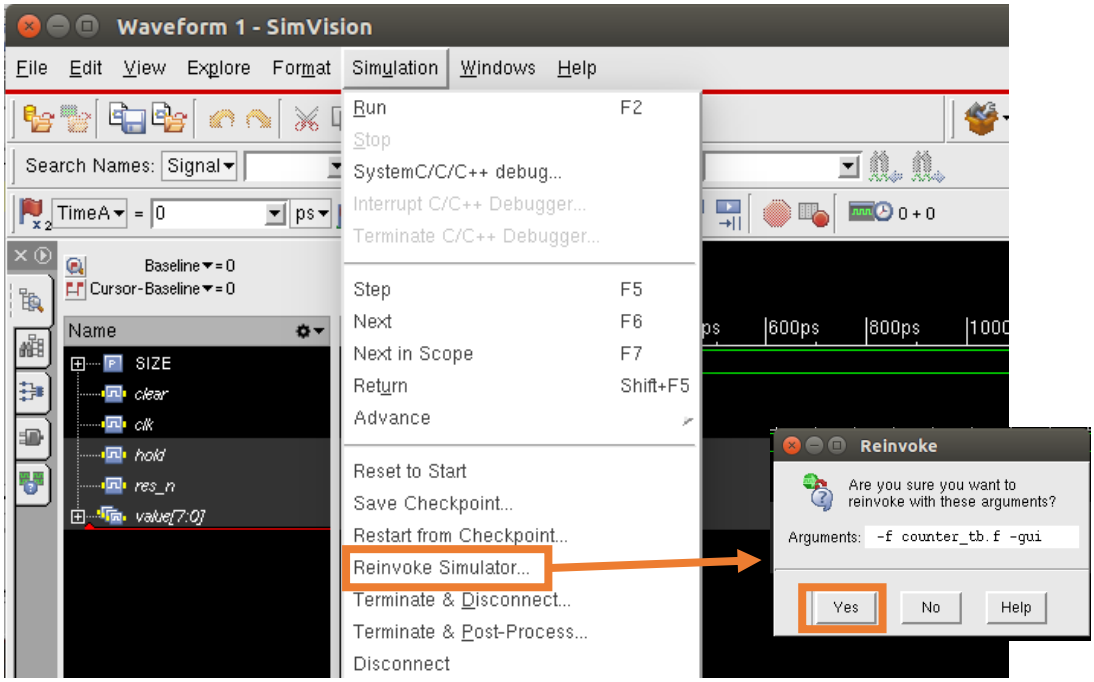

### 4.5.3 Starting the simulation

To start the simulation, click on the “Play” button. It will stop when the testbench $finish has been reached, otherwise you must press the “Pause” button to stop the simulation.



### 4.5.4 Restart the simulation after a design change

If the Verilog sources (testbench or circuit) have been modified, the simulator must be completely restarted. To do so, you can use the “Simulation -> Reinvoke Simulator” Menu.




Afterwards, you can rerun the simulation.

### 4.5.5 Using the waveform window

The waveform window can be tricky to use at first, you have to try around by yourself, or consult the cadence documentation.

Here are a few interesting shortcuts to get started:

| Function | Shortcut |
|---|---|
| **Zoom in/out** | CTRL + Mouse Wheel |
| **Zoom on a specific area** | Left Mouse Drag |
| **Move the time cursor** | Left Click anywhere on the waveform or CTRL + Left Mouse Drag |
| **Move the baseline cursor** | Middle Mouse click anywhere on the waveform or CTRL + Middle Mouse Drag |
| **Move a signal in the signal list** | Left Click and drag the signal up/down in the signal list on the left |
| | |

# 5 Counter Synthesis

Now that we have a functional counter, we can synthesize it to map the circuit in the UMC65 technology. We are going to run the synthesis multiple times with different options to see how the circuit maps. After synthesis, we will simulate the counter again, but with the gate-level netlist, not the original Verilog.

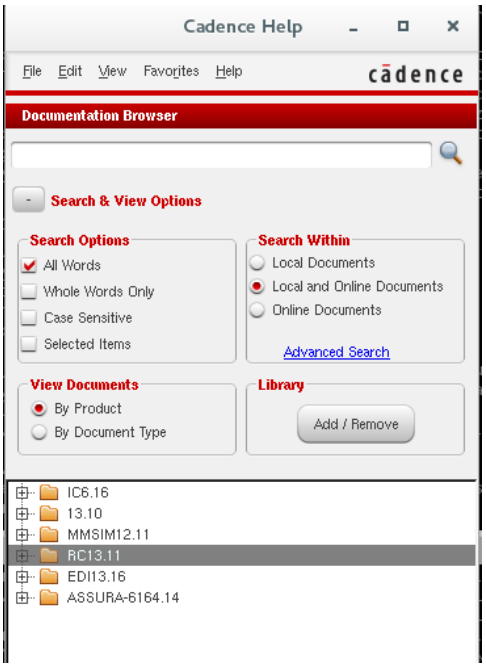
To summarize, the design flow is the following:

- ✓ Prepare a Verilog module with a fixed-width counter instantiation, that would be the final design to implement
- ✓ Prepare the Synthesis run:
  - Prepare the synthesis flow script
  - Write the timing constraints, or use existing ones
  - Synthesize
  - View the results, analyse timing
  - Simulate the Gate-Level netlist
- ✓ Repeat the synthesis run for different parameters

## 5.1 RTL Compiler documentation

The synthesis tool is called RTL compiler. When you call the *cdnshelp* command (see 3.2.3), you can find the tool documentation under the "RC 13.1" folder.

Please note that the commands presented here are not always the only way to go. The tool has a lot of functions, and sometimes various approaches can lead to the same result.




## 5.2 TCL programming training

As mentioned before, the synthesis and place and route tools are driven by TCL scripts. That is, all the functionalities are bound to a TCL programming interface, and must be called through scripts.

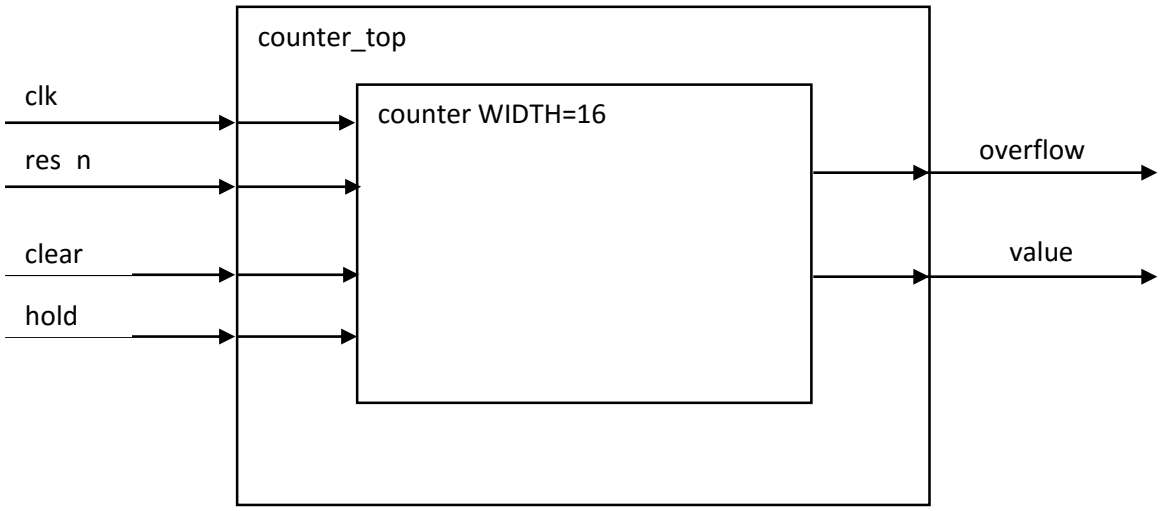
If you are not comfortable with programming and/or TCL programming, start by walking through a tutorial to learn the very basics. You can also learn on the fly.

## 5.3 Prepare the top level

The counter base module we wrote has a parametrized width. The module to synthesise must configured as the final wished configuration.

To create the top level, write a Verilog module called “counter_top”, which only contains an instance of the counter module, with a size of 16bits. The top level input and outputs are the same as the counter, you can just forward them to the counter instance.

Save the file under the “sources” folder, or the “synthesis” folder.




## 5.4 First Synthesis

To run the synthesis, we are going to write a TCL script.

Create a folder called **run_first** under the **run/synthesis** base folder you created at the beginning.

```
$ cd lab1/run/synthesis
$ mkdir run_first
```

Create a TCL file script called **synthesize.tcl** and save it under the **run_first** folder.

The content of the synthesize script will just mirror the synthesis flow presented in the lecture slides. All the synthesis commands presented afterwards must be written inside the **synthesize.tcl** file.

### 5.4.1 Configure the tool

The RTL compiler must be configured at the beginning. Configuration attributes are used to setup the global behaviour of the software, like the number of Threads used during synthesis.

In our case, we are only going to configure the tool to produce an error if a black block is detected. A blackbox is a Verilog Module instance which has not source file associated. Its content is thus unknown. Add the following lines at the beginning of the script:

```
## Configuration
####################

# Fail on blackbox
set_attribute hdl_error_on_blackbox true /
```

### 5.4.2 Read the libraries

Next, we need to load the LIB file containing the Standard Cell and timing information.

The LIB file is located in the software installation. The Environment variable UMC_HOME points to the installation location:

```
## Read Timing Libraries
######################
set_attribute library $::env(UMC_HOME)/65nm-stdcells/synopsys/uk65lscllmvbbr_090c125_wc.lib
```

There are lib files available for various temperature, voltage and speed corners. The one we chose is for 0.9V core voltage, 125° and worst case timing.

Remember that we try to implement the circuit so that it will always work, thus the worst case library.

Finally you can add a check library command which will output a report containing information about the cells that were found in the library.

```
check_library
```

### 5.4.3 Read the Design

Now the design can be read and elaborated. The principle is the same as for simulation, you must pass the files:

- ✓ counter_top.v which is the physical top level
- ✓ counter.v which is the counter implementation

```
## Read design
########################
read_hdl -sv [list counter_top.v  counter.v]
```

The **-sv** argument enables SystemVerilog version of Verilog

### 5.4.4 Elaborate

The next step is to elaborate the design, just call the elaborate command:

```
## Elaborate
################
elaborate
```

### 5.4.5 Start the tool a first time

At this point, we can start the tool, and have a look at the elaboration results.

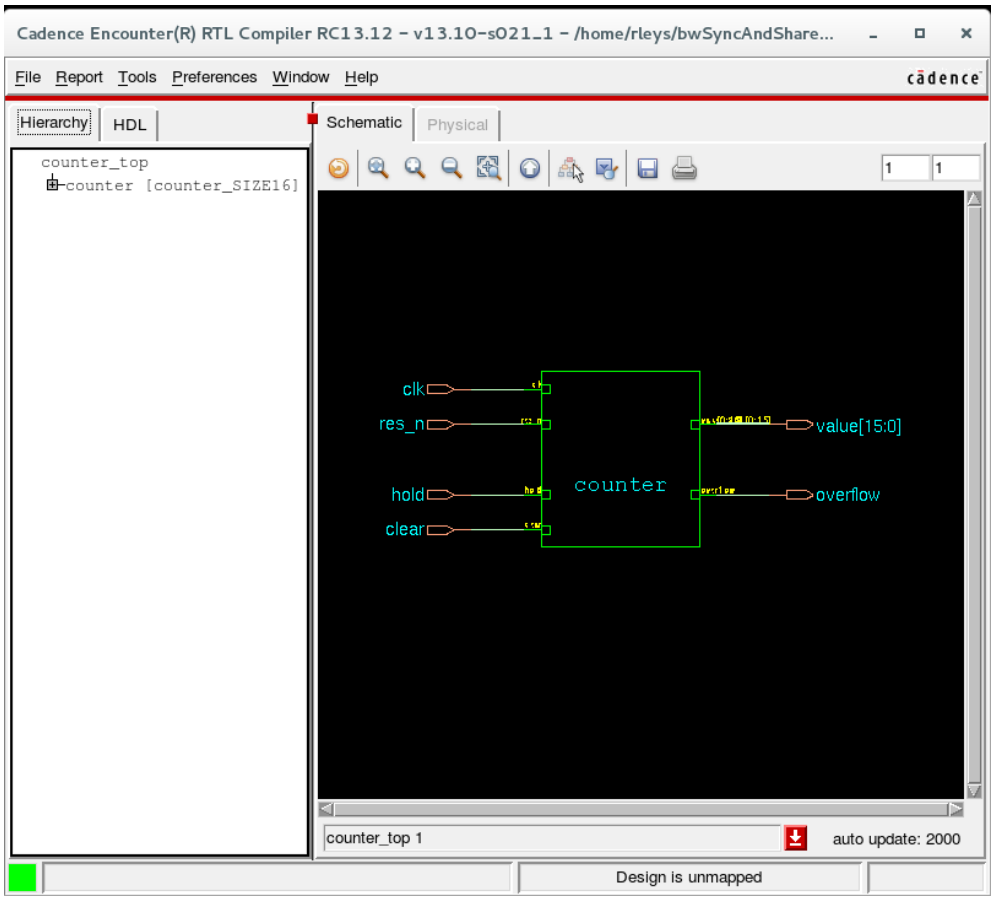
Go inside the **run_first** folder, if not already there, don't forget to load to tools, and type:

```
$ rc -64 -files synthesize.tcl
```

The tool should start, if some errors appear, check all the paths to the library and Verilog files are correct.

Per default, no graphical interface is started. To have a look at the circuit, you can open the GUI from the command line:

```
rc:/> gui_raise
```



### 5.4.6 Write the constraints

Once the design has been elaborated, the tools has a representation of the circuit and the constraints can be loaded.

To write the constraints, create a new file called **constraints.cstr.**

This file is also a TCL script and can contain variables and such.

#### *5.4.6.1 Create the clock*

First we are going to create the clock with the *create_clock* command:

```
## Create Clock from top level clk port
######################
create_clock -name clk -period 1.3 [get_port clk]
```

- ✓ The name can be set freely
- ✓ The period is expressed in nano seconds per default
- ✓ The *get_port* function retrieves the tool's representation of the input/output definition for "clk" (input wire clk in Verilog)

We also need to add some uncertainty to the clock definition, because the design is in an early stage. The uncertainty will just be removed from the clock period during timing analysis:

```
## Set Clock uncertainty: just some overclocking
######################
set_clock_uncertainty 0.150 -setup clk
```

We only add uncertainty for the setup timing checks, because no hold check is performed during synthesis.

#### *5.4.6.2 Input Output constraints*

The input and outputs must be constrained as well to spare some clock period time for the logic on the outside world. This can be adjusted once one precisely know how the final design looks like.

For now, we can use such constraints:

```
##I/O Delays
set_input_delay -clock clk  .600 [remove_from_collection [all_inputs] [concat clk] ]
set_output_delay -clock clk .600 [all_outputs]

## Output Cap: 15pf is typical and safe
set_load -pin_load 0.15 [all_outputs]

## An 8 driving strength buffer at the input
set_driving_cell -lib_cell BUFM8R [remove_from_collection [all_inputs] [concat clk] ]
```

- ✓ 600ps from the clock is reserved for the outside world on both input and outputs
- ✓ The input and output constraints are only set on signals relevant the clock. That is why we don't constraint the clock itself.
- ✓ The set_load constraint indicates the tool how much capacitance must be driven outside the pin. This is very important so that the output cell won't be too slow.
- ✓ The driving cell specifies what kind of buffer is located outside the input. This is also important to make sure the tools can add enough buffers to the circuit so that the input buffer can still drive the inputs.

### 5.4.7 Load the constraints

Loading the constraints simply consists in reading in the constraints file:

```
1 ## Read constraints
2 ########################
3 read_sdc constraints.cstr
```

Now run the synthesis tool again, and see that constraints applying was successful.

```
Statistics for commands executed by read_sdc:
 "all_inputs"              - successful      1 , failed      0 (runtime  0.00)
 "all_outputs"             - successful      1 , failed      0 (runtime  0.00)
 "create_clock"            - successful      1 , failed      0 (runtime  0.00)
 "get_port"                - successful      1 , failed      0 (runtime  0.00)
 "remove_from_collection"  - successful      1 , failed      0 (runtime  0.00)
 "set_clock_uncertainty"   - successful      1 , failed      0 (runtime  0.00)
 "set_input_delay"         - successful      1 , failed      0 (runtime  0.00)
 "set_output_delay"        - successful      1 , failed      0 (runtime  0.00)
Total runtime 0
rc:/>
```

### 5.4.8 Configure Synthesis

If needed, we can set some extra attributes to configure the synthesis. For now, we will just set the interconnect mode the physical synthesis. We won't perform physical synthesis right away, thus the interconnect time will always be 0.

```
1 # Physical Synthesis Mode -> no wire delay
2 set_attribute interconnect_mode ple /
```

### 5.4.9 Run Synthesis

To run the synthesis, we just use the synthesize command. You can have a look at the documentation for this command.

For example we can use following calls:

```
## To Mapped
synthesize -to_mapped -effort high

## Incr Optimisation
synthesize -to_mapped -effort high -incr
synthesize -to_mapped -effort high -incr
```

- ✓ The first call just maps the circuit
- ✓ The second and third calls try to improve the timing

You can play around with the number of incremental runs and the effort values to find out how good the synthesis performs.

### 5.4.10 Timing report

To check the timing, a good idea is to first check the constraints are correctly set by using the timing linter:

rc:/> report timing –lint

Does the setup seems ok?

To get a timing report for the worst path, just use the command:

rc:/> report timing

What does the timing report mean? Can you identify the various delays (cells, wires, constraints etc…) ?

Now try commenting out the output capacity and input driving cell constraint, and re-run the tool.

What does report timing –lint tell you?

What is the worst timing path reported?

**Have a look at the documentation for the « report » command, to see all the options available**

### 5.4.11 Global Reports

Additionally a nice global report is the Quality of result report:

```
rc:/> report qor
```

What can you see in this report?

### 5.4.12 Write out the netlist

Finally, we can write the gate level Verilog netlist:

```
## Write report and data
##############################
write_hdl > counter_top.v
```

### 5.4.13 Simulate the Gate Level Netlist

To simulate the netlist, just run the digital simulation as previously done, but with following changes:

- ✓ The file “counter.v” must be replaced by the new netlist “counter_top.v”
- ✓ The testbench must be adapted to simulate a counter of 16 bits
- ✓ Check that the “-timescale 1ns/10ps” argument is present in the “counter.f” file
- ✓ The Verilog model of the standard cells must be added to the F file:

**${UMC_HOME}/65nm-stdcells/verilog/uk65lscllmvbbr.v**

How different does the design in the simulator look like?

## 5.5 Synthesis with asynchronous reset

In a second time, we can try to modify the reset pin to be an asynchronous reset.

Asynchronous resets are not part of the logic path, thus not dependent on the clock, and are useful to ensure the flip-flops have reached their reset state even if no clock is available, for example during power-up.

When we synthesise the design with an async reset, we should see a difference in the wiring on the output circuit.

Start by creating a run_async folder next to the run_first folder, and copy-paste the synthesize script and the constraints.

### 5.5.1 Modify the counter Verilog

The “always” block in the counter must specify the reset in the sensitivity list. Thus, the block will reach a state on both clock edges and reset edges.

The asynchronous reset format is the following:

```
always @(posedge clk or negedge res_n) begin
```

### 5.5.2 Modify the timing constraints

The reset path is now independent from the clock, thus we must exclude it from timing.

Modify the input delay constraints to exclude the res_n input:

```
set_input_delay -clock clk  .600 [remove_from_collection [all_inputs] [concat clk res_n] ]
```

Additionally, the reset can be set as false path, because it has no relation to the clock and does not need to be timed. Add this line to the constaints.cstr file:

```
## Don't time reset
set_false_path -through [get_port  res_n]
```

### 5.5.3 Run elaborate

Run the synthesis script, at least until elaborate.

What happens?

How can you fix the Verilog sources if an error is reported?

FIX the design if needed.

### 5.5.4 Synthesize

Now run the synthesis again.

What difference can you see in the Circuit View in the GUI?

### 5.5.5 Simulate

Run the simulation with this new netlist.

Modify the test bench and set the res_n signal to 0 somewhere in the middle of a clock period.

What happens?

## 5.6 Synthesis with clock gating

Now we configure the synthesis to use clock gating.

Clock gating means that if a register's value should not be changed, for example when hold is asserted on the counter, the synthesis will disable the clock on the flip-flop.

Again, create a folder called “run_cgate”, and copy paste the scripts and constraints from the asynchronous reset run.

### 5.6.1 Enable clock gating

At the configure synthesis location in the synthesize script, add the following attribute:

```
1 # Enable clock-gating insertion
2 set_attribute lp_insert_clock_gating true /
```

### 5.6.2 Synthesize

Run the synthesis.

Have a look at the clock gating report:

```
rc:/> report clock_gating
```

Did it work?

Now have a look at the circuit view in the GUI.

What happened to the clock line?

Output the quality of results report again (report qor).

Can we see any difference to the previous runs?

### 5.6.3 Simulation

Simulate the design once again.

Can you watch the clock line of the value register?

During the hold time, does the clock line looks consistent with the expectations?

### 5.6.4 Next Steps

Now you can switch to lab2 where we are going to use the place and route tools to physically place the counter on an die area.