Lab2: Synthesis and Place and Route

Duration: Until the end of semester

1 Overview

We have seen during lab 1 how to simulate and synthesise a counter. In this lab we will try to go through the place and route.

First, we will synthesize the design with proper I/O cells instantiation, then perform floorplaning and synthesize the design using physical synthesis.

For the constraining and synthesis part, you can just reuse and adapt the scripts written during lab1.

IMPORTANT INFORMATION:

- **Download the lab2.zip file from ILIAS which contains all the scripts for physical synthesis and place and route.**
- **You can use those scripts to run the tools, their content is explained in the various assignments.**
- **The paths to various files in the scripts (like source files etc…) will probably differ from your own. If some "file not found" errors happen, don't forget to check the file paths in the scripts.**

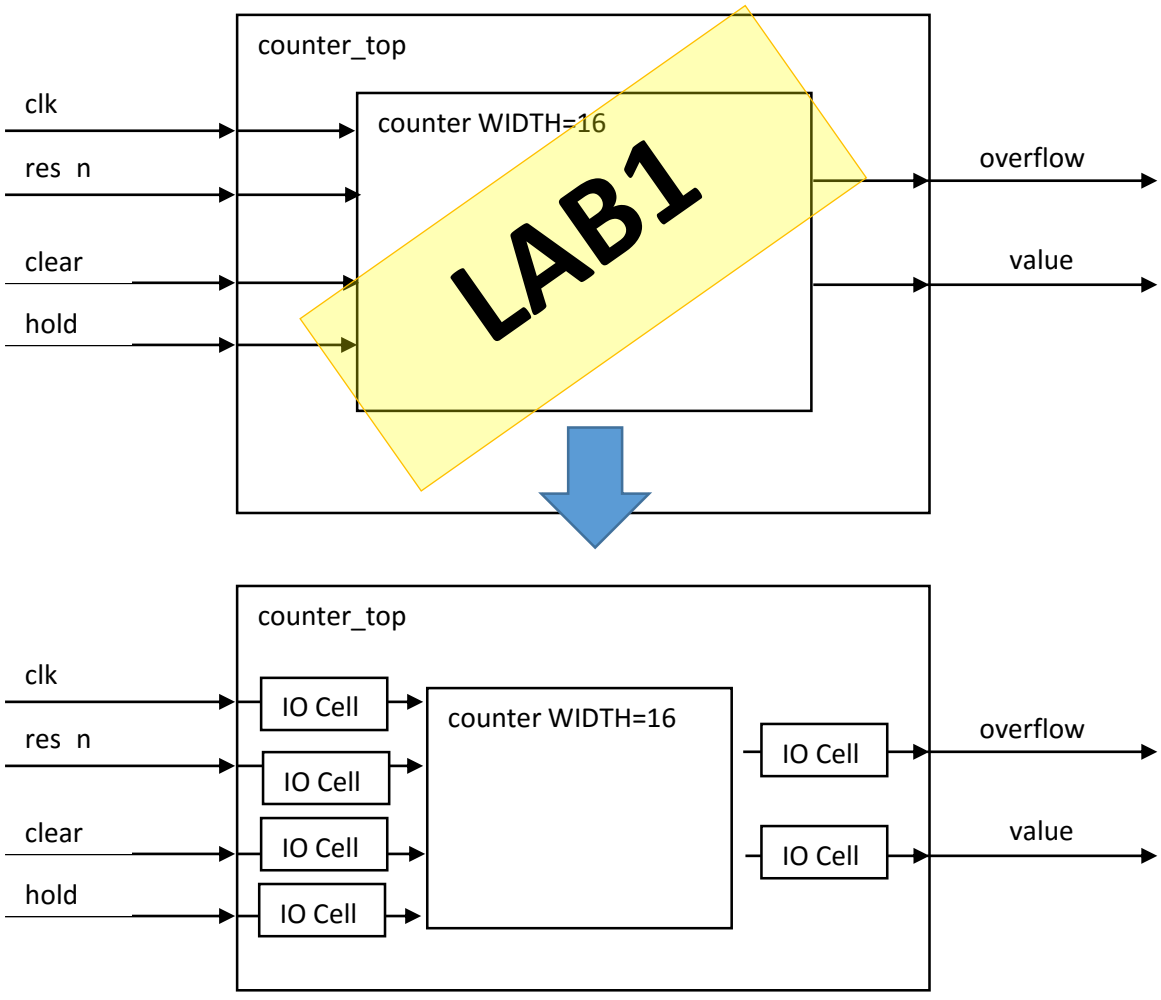
2 Synthesis with IOs

2.1 Understand the Top Level I/O

This time we want to synthesise the design for a real physical implementation.

To do so, we will add the Input/Output Cells to the top level Verilog file:

- In the first lab, you had prepared a counter_top.v module with a 16bit counter.
- The counter_top module was only a wrapper to the counter, we will now add the IO cells



2.2 IOCell Documentation

To understand how the I/O cells are used, you must refer to the technology documentation. In this case, you can have a look at the file:

/var/autofs/cadence/umc-65/65nm-pads/doc/databook.pdf

2.3 Prepare the Top level

Now we need to add the IO cells instances to the counter_top.v

Here is an example for the hold input cell instantiation:

```
// Signal for internal connection
    wire hold_int;

    // IO Cell
    IUMA hold_in (
        .PAD(hold), // IO signal from counter_top
        .DI(hold_int), // Internal wire for counter
        .OE(1'b0), // Next: Enable signals for IO Features
        .PIN1(), .PIN2(),
        .SMT(1'b0),.SR(),
        .PD(1'b0), .PU(1'b0),.DO());
```

The counter signal connection must be made with the internal signal then:

```
counter   #(.SIZE(16))  counter (
        .clk(clk),
        .res_n(res_n),

        .hold(hold_int), // Changed!

        .clear(clear),
        .value(value)

    );
```

Use the documentation to find out what the various IO cell signals mean

Reproduce the instantiation of the hold signal for all the other signals

Beware, the value signal have multiple bits, so you need a “generate” Verilog construct to create one IO cell per bit.

You can have a look at sources/counter_top.v for a reference

2.4 Prepare the Constraints

You can reuse the same constraints file as for lab 1

2.5 Run Synthesis

You can reuse the same scripts and constraints as in the Lab1, with one difference, you now need to add the timing library file for the IO cells.

Look for the timing library setup line in your synthesis script, and add the following file:

$::env(UMC_HOME)/65nm-pads/synopsys/u065gioll18gpir_wc.lib

It should look like:

```
#### Read Timing Libraries
set_attribute library [list  \
    $::env(UMC_HOME)/65nm-stdcells/synopsys/uk65lscllmvbbr_090c125_wc.lib \
    $::env(UMC_HOME)/65nm-pads/synopsys/u065gioll18gpir_wc.lib \
]
```

2.6 Results

Have a look at the timing reports and have a look at the IO cell impact.

3 Floorplaning

To perform floorplaning, we will need to learn:

- How to load the synthesized netlist in encounter
- Setup all the correct timing corners
- Modify the Die area and place the IO Cells

To start, create a new folder along the “synthesis” folder called “encounter” and work from it.

3.1 Opening the Cadence Help Documentation

Sometimes you will need to have a look at the tool documentation to understand the commands, or find the ones that suits your needs. To open a cadence help window, proceed as following:

Open a terminal, source the tools environment and call the “cdnshelp” tool, to access the documentation. The Chapter for Encounter is called “EDI13.16”.

Use the Cadence Help to lookup the cited command definitions and arguments.

3.2 The Load Design Scripts

Loading the design in Encounter is merely the same process as in RTL compiler, but the setup must be more complete, especially for the timing libraries, as we need both slow and fast timing corners for setup and hold time violation.

To load the design, we are going to:

- Load the timing libraries using the Multi-Mode Multi Corner API
- Set the variables for the physical LEF files and the netlist
- Call the init_design passing the timing corners used for setup and hold time analysis.

To do so, let’s create a script called “load_design.tcl”, and as first line, call the “freeDesign” command. This way we can recall this script over and over.

Additionally, we can add some basic configuration for 65nm technology node:

```
1 ## First Free the design
2 catch {freeDesign}
3
4 ## 65nm configuration
5 setDesignMode -process 65
6 setDelayCalMode -engine Aae -signoff false -SIAware false
```

3.2.1 Load the timing library

As mentioned previously, loading the Timing libraries is more difficult in encounter because of the Multi Mode Multi Corner setup. Just as a remainder, we have:

- Mode: A mode is a functional setup for the ASIC, like the standard running mode, or the test mode during which a different clock is used to test the circuit. Each mode requires its own set of constraints, and in our case we only defined one mode, which we will call “functional”.
- The corners are the combination of library characterisation (slow, fast, typical) and operating conditions available (Temperature, voltage etc…). They are more difficult to define because they can grow in number quite fast. We will try to not really use them to avoid long runtimes.

When MMMC mode is set, some analysis views are created, which are a combination of a mode and a corner.

When initialising the design, we can define which views are to be used for setup and hold analysis. We will then always used the slow corners for setup and fast corners for hold.

Let’s start by creating the library sets, which gather the timing libraries, just like in RTL compiler:

```
1 create_library_set -name worstHT \
 2     -timing [list $::env(UMC_HOME)/65nm-
stdcells/synopsys/uk65lscllmvbbr_108c125_wc.lib \
 3      $::env(UMC_HOME)/65nm-pads/synopsys/u065giolll18gpir_wc.lib \
 4      ] \
 5     -si      [list $::env(UMC_HOME)/65nm-
stdcells/synopsys/uk65lscllmvbbr_108c125_wc.db \
 6      $::env(UMC_HOME)/65nm-pads/synopsys/u065giolll18gpir_wc.db]
 7
 8 create_library_set -name bestLT \
 9     -timing [list $::env(UMC_HOME)/65nm-
stdcells/synopsys/uk65lscllmvbbr_132c0_bc.lib \
10       $::env(UMC_HOME)/65nm-pads/synopsys/u065giolll18gpir_bc.lib \
11       ] \
12     -si      [list $::env(UMC_HOME)/65nm-
stdcells/synopsys/uk65lscllmvbbr_132c0_bc.db \
13       $::env(UMC_HOME)/65nm-pads/synopsys/u065giolll18gpir_bc.db]
```

(this is not easy to read, the scripts are available online for reference)

Here we have chosen the Worst case libraries characterized for 1.08V@125° and the best case for 1.32V@0°.

Then we can load the constraints file. We can reuse the same one as for the synthesis:

```
1 ## Load constraints
2 create_constraint_mode -name functional -sdc_files /path/to/constraints
```

Now that we have constraints and libraries, we can create the delay corners. A delay corner is the combination of:

- A Library set, i.e a timing specification (slow, fast, typical)

- A configuration for the RC extraction, called a RC corner

Thus, we need to create the RC corners, and then the delay corners which associate a library set and an RC corner.

This step can become hard to understand, and we are lacking some configuration files for the moment, so we will keep it very simple:

- RC Corner: one 125° and one 0° corner
- Delay Corner: Slow library + 125° RC and Fast library + 0° RC

```
create_rc_corner -name rcworstHT   -T 125
create_rc_corner -name rcbestLT   -T 0

create_delay_corner -name worstHTrcw -library_set worstHT -rc_corner rcworstHT
create_delay_corner -name bestLTrcb -library_set bestLT -rc_corner rcbestLT
```

Finally, we must combine the Modes with the delay corners in analysis views. This is easy for us, because we only have one mode and two corners, so that will be:

```
## Final: Create views
create_analysis_view -name functional_worstHT -constraint_mode functional -delay_corner worstHTrcw
create_analysis_view -name functional_bestLT -constraint_mode functional -delay_corner bestLTrcb
```

3.2.2 Load the Physical libraries

This step is very easy, we only need to set a variable with the LEF files. This is the same as for the RTL compiler tool setup:

```
set init_lef_file [list \
    $::env(UMC_HOME)/65nm-stdcells/lef/tf/uk65lscllmvbbr_9m2t1f.lef \
    $::env(UMC_HOME)/65nm-stdcells/lef/uk65lscllmvbbr.lef \
    $::env(UMC_HOME)/65nm-pads/lef/u065gioll18gpir_9m2t2h.lef \
    $::env(UMC_HOME)/65nm-pads/lef/tf/u065gioll18gpir_9m2t2h.lef]
```

3.2.3 Initialise the design

Finally, we must provide the path to the netlist which was produced during the synthesis run, and call the init_design command, passing the desired list of timing analysis views:

```
set init_verilog "../../synthesis/_nosync_run/netlist/netlist.v"
set init_top_cell counter_top
```

And

```
1 ## Init design by giving corner pairs for hold/setup
2 init_design -setup [list functional_worstHT ] -hold [list functional_bestLT]
```

3.3 Start encounter and load the design

We could start encounter like RTL compiler, with a script performing all the steps, but this time it is more complicated, so we will just start the command line, experiment and enrich our scripts.

To start encounter, don't forget to source the tools, then in your encounter folder, type:

```
$ encounter
```

The application starts with a command line interface, and a GUI from which you can visualise the design.

From the command line, you now source the load_design.tcl script:

```
$encounter:> source load_design.tcl
```

The GUI should freeze until all the libraries are loaded, then you should be able to watch the design in the main windows.

Use the keyboard "R" shortcut to redraw the view, or "F" to fit the view.

How does the floorplan look like?

Can you see the IO cells, how are they placed?

3.4 Encounter GUI

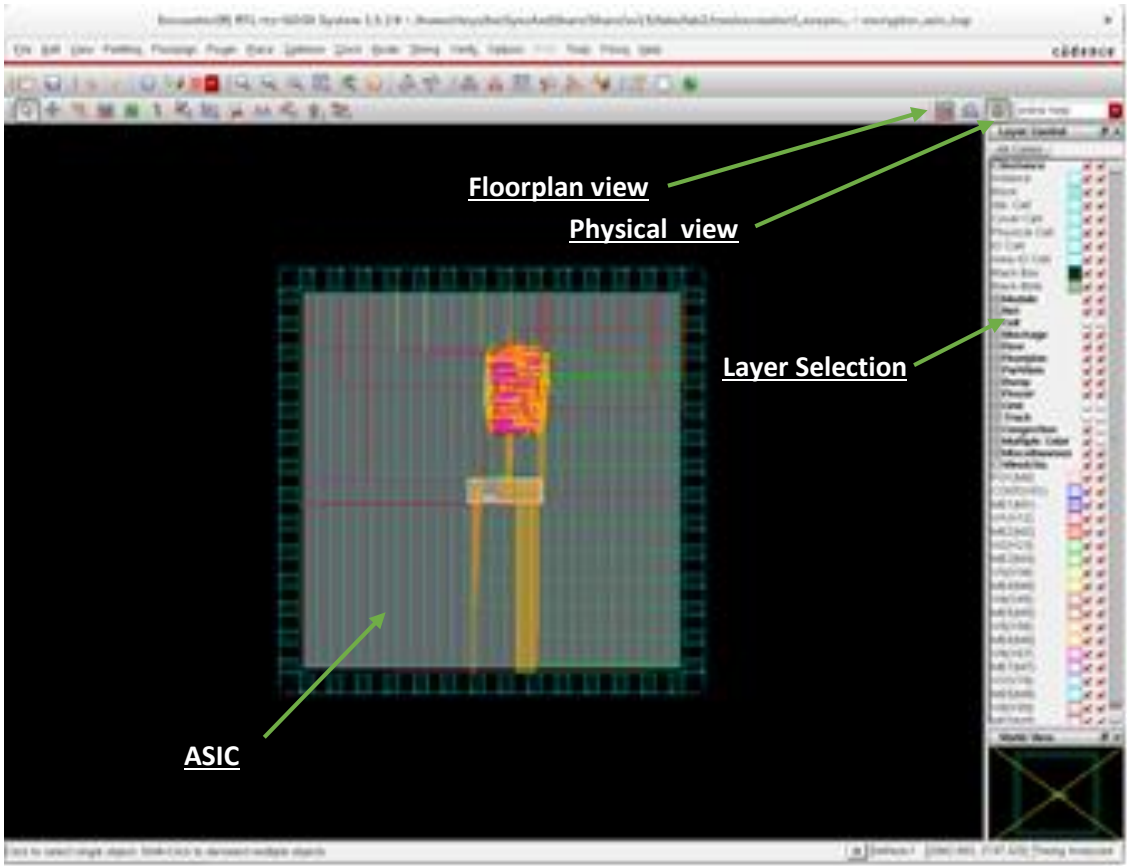
The GUI of Encounter provides a window in which you can inspect the physical view of the ASIC.

Two views are important:

- The Floorplan view, on which you see all components, the unplaced one being located on the bottom-right side of the die
- The Physical View, in which you can see all the structures: Cells, Blocks, wires, vias etc..

If you use the GUI to play around the design and call some functions, the TCL commands called are written to a file called “encounter.cmd”, which is located in the folder where you started the tool.

Sometimes the tool behaves strangely, and using the GUI is the only way to achieve a routing for example. In such cases, you can just watch the encounter.cmd file and save its content to a script to be called back later.



3.5 Floorplan: Prepare the Die Size

To prepare the floorplan, create a script called “floorplan.tcl”.

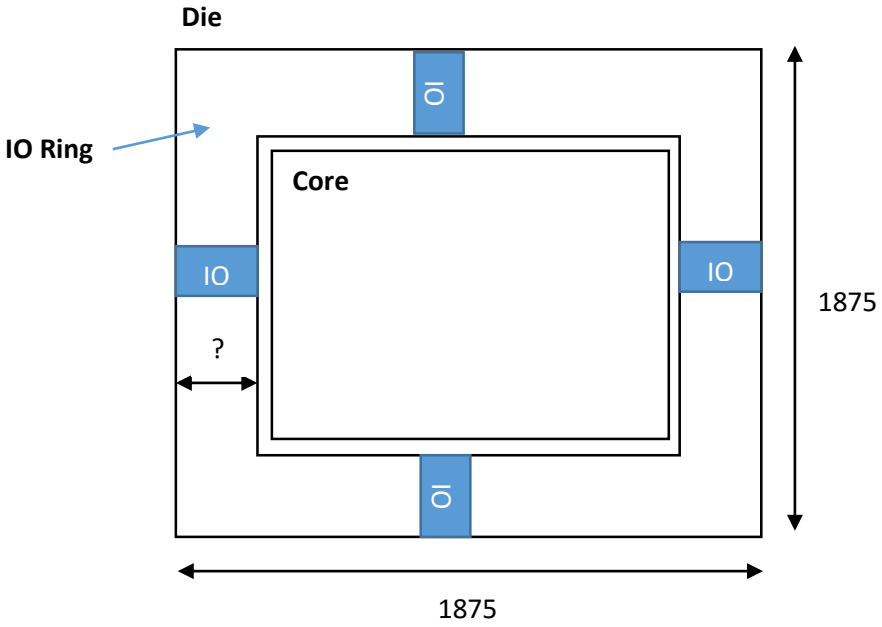
To initialise the floorplan, we need to make a few choices:

- How big?
- What kind of I/O Placement: Area IO (I/O cells on the core area), peripheral IO (I/O cells around the core Area) ?

To make scripting less complex easy, we will keep the IO cells around the core area, and choose as die size the minimal size that we would have to order if the ASIC were to be submitted to Europractice for a multi-project run.

The minimal size for UMC65 is 1875x1875µm.

Look for the “floorPlan” command in the Encounter documentation (Under Encounter Text Command Reference -> Floorplaning commands”, and use it to create a die with following specification:



Use the IO Cell databook.pdf document to find out how wide the I/O ring must be.

Add a spacing of 20µm between the I/O ring and the core area.

Add the created command to the floorplan.tcl script.

3.6 Preparing the I/O

The initial setup in encounter shows the I/O cells we have instantiated placed around the Core area.

This is however for a final design not sufficient, because the I/O cells must include power source cells, ESD protection cells, fillers etc…

Open the I/O cell application note under:

/var/autofs/cadence/umc-65/65nm-pads/doc/application_note.pdf

In this document, you can find the design rules for the I/O cells.

Try to determine which I/O Cells you need to add to the I/O Ring.

Make a connection plan to determine where you want to place the regular I/O cells around the core.

Try to write a script that will instantiate the extra cells, and place them around the core with the regular cells:

➔ Have a look at the Floorplan commands “addIoInstance” , “selectInst” and “moveSelObj”.

Add the commands to the floorplan.tcl script.

3.7 Finishing the Floorplan

To finish floorplaning, we can add extra structures and constraints for the rest of the implementation.

Typically, those extra steps will be:

- Cutting the Standard Cell rows. Indeed, the standard cells are placed on the core area in Rows. However, if we have placed the SRAM cell on the core area, and we want to keep some spacing between the block and the standard cells, because they are not compatible if abutted to each other. Cutting rows will do so.
- Add some routing blockage over special blocks. The RAM cells will typically use metals from layers 1 to 4. That means that the tool cannot use the metal layers ME1 to ME4 on top of the RAM cell.
- Add any other routing blockage or guides for which we already know they are going to be helpful.
- Add some extra cells on the design like Well Taps. Well Taps are bias connection to the substrate. They ensure the substrate potential is evenly distributed and not weak in any point to prevent latch-up effects. The rules for placing the well-taps are typicall found in technology documentation. You could have a look at the following document in our case:

  /var/autofs/cadence/umc-65/65nm-stdcells/doc/databook.pdf and application_note.pdf

Have a look at the floorplan.tcl script from the online sources, and add the row cutting, routing blockage and well-tap cells to your own script.

3.8 Save the Floorplan

Finally, use the defOut command to save the floorplan.

See the last command in the floorplan.tcl script:

```
1 defOut -floorplan floorplan.def
```

4 Physical Synthesis

To run physical synthesis, the idea is to reuse the same scripts as in 2 Synthesis with IOs, but providing a floorplan information.

To so, you need to change the synthesis setup in a few places:

- Setup a small script to configure the place and route tool when it is called by synthesis
- Add the Physical Information for the Standard Cell by loading the LEF file
- Read the floorplan information saved previously in the DEF file.

The first two items can be covered during library loading before elaborate, you can have a look at the synthesize.tcl script from the source package:

```
1 set_attribute enc_user_constraint_file ../encounter_configuration.tcl
```

```
1 #### Read Physical Definitions
2 set_attribute lef_library [list \
3     $::env(UMC_HOME)/65nm-stdcells/lef/tf/uk65lscllmvbbr_9m2t1f.lef \
4     $::env(UMC_HOME)/65nm-stdcells/lef/uk65lscllmvbbr.lef \
5     $::env(UMC_HOME)/65nm-pads/lef/u065gioll18gpir_9m2t2h.lef]
```

Then read the DEF floorplan right before performing synthesis, you can just copy the DEF file to your synthesis folder:

```
1 read_def floorplan.def
```

Then call synthesis in placed mode:

```
1 synthesize -to_placed -effort medium
2 synthesize -to_placed -effort medium -incr
```

4.1 Results

After running in placed mode, you should be able to select the physical tab and watch the design. As you can see, it is pretty empty.

Now use the “report timing” command to watch a few negative slack paths:

$ report timing -num_paths 4

Can you find some wire delay information in the timing report?

Run the synthesis with high effort if not already, can you see a difference in the number of negative slack paths?

5 First Place and Route

Go back to the encounter folder, and load the design again using the load_design script

5.1 Floorplaning

We have done floorplaning before, so you can just load the floorplan.tcl script you have prepared before.

5.2 Prepare Power Structures

To deliver power across the design, we can setup a power grid on Metal layers 8 and 9, and make sure that VIA connections are placed down to the power lines of the standard cells.

Typically, the power nets are global nets, because all the cells from a power domain will connect to the same power net.

Look in the reference script *power.tcl* for the global net commands. First the global nets for power and ground are created, then the power pins of all the structures connected to it. The connection is virtual, but it allows the tools to ensure all the cells have those power nets connected to a physical wire of the same name at some point.

The command used to place the power stripes is called *addStripe* (see reference script for usage).

Basically, you can add a tight power grid, as long as you think it won't prevent routing from being successful. On Technology nodes like 65nm with 9/10 Metal layers, you will use 4 to 5 routing layers (from 1 to 5/6), and layers from 7 to 9/10 are used for power grids and Input Output or special wire connections.

Use the reference script and the addStripe command to add a power grid.

Try to change the spacing and or layers of the power grid. Make sure the horizontal and vertical stripes

How do you choose on which layer the horizontal and vertical stripes are placed? Have a look at the LEF file which contains the LAYER definitions, what can you learn from that file?

Now to add connections from the power grid down to the power lines of the cells, we can use the *editPowerVia* command.

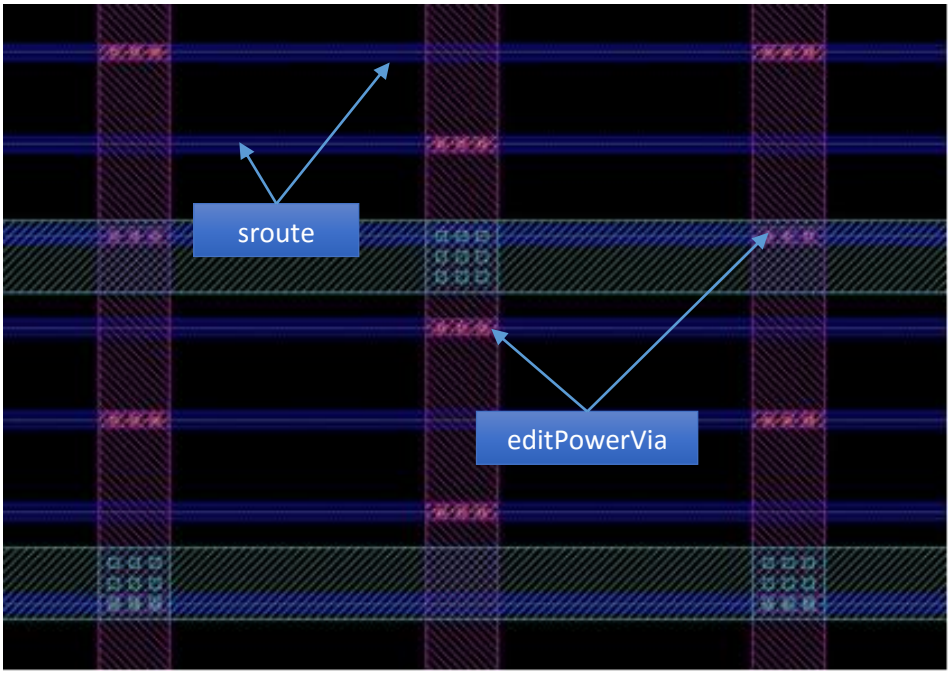
This command adds VIA connection from the specified bottom to top layers, wherever some wires or cell PIN connected to the same power global net are crossing each other.

Some power nets connection might still be missing at this point however. Some blocks might not be located under a power grid. This is the case if the design is big and complex, then the top level usually only contains blocks and partitions.

Another example would be the standard rows power connections. Indeed, we have seen that standard cells have VDD and VSS power lines on top and bottom of their design on Metal 1. At the moment, the standard cells are not placed on the design so no power connections can be made.

The tools feature a tool called Special Route (*sroute* command), which takes care of routing those special nets between blocks. In our case, we can call *sroute* to add Metal 1 wires along the standard cell rows, and enable nice power connections from the power grid.

The following picture illustrates the two steps. Special route adds some Metal 1 wiring between the rows, and editPowerVia make connections to the power grid. This way the power connection is regular, but no too tight massively blocking routing.



5.3 Placing the design

Now that most structures have been prepared, we can place the design.

This can be done easily by using the *placeDesign* command.

You can check the *setPlaceMode* command which defines the placement options. The goal is to try to minimize the number of specific options required. Ideally, calling *placeDesign* would be enough.

Look at the place script, you can see there is a bit more than just placing the design:

- First we add some spare cells on the die area. This is only relevant if you are producing an ASIC whose Production Masks can be modified later. In most cases, like for Multi Project Runs, you don't need any spare cells.
- Then the design is just placed
- Finally, we can add the Tie Hi and Tie Low cells. These are used to connect data path constants 0 and 1 to the power lines. They are added at this point in our design, but they could also be added during synthesis.
- The timeDesign command is called to output a timing report

5.4 Save the Design (optional)

At this stage, you could save the design state, and restore it later. It is very useful if you want to try various options, or need to close encounter.

Look up the “saveDesign” command in the documentation, save the design, close encounter and restore the design using the “restoreDesign” command afterwards.

5.5 First Timing

Find the *timeDesign* command in the documentation and use it to write timing reports for hold and setup times. Have a look at the place script to see how it is used.

How does timing look like? What is the slack of the Worst Path?

Try to use the “Timing -> Debug Timing” Menu from the GUI to get a graphical analysis of the timing report.

Why is the worst path so bad?

5.6 First Design optimisation

You can try to run this step, but it may be very slow. Just ignore it if that is the case.

After design placement, and before creating the clock tree, we can already optimise the design to get a better timing overview.

This step can be found in the “cts.tcl” script from the sources.

Find the *optDesign* and *setOptMode* commands in the documentation to see the options.

To use optDesign in our case, let’s keep it simple:

- Use setOptMode to set the optimisation effort to low
- Call “optDesign -preCTS” because we haven’t performed clock tree synthesis yet.

Use the timeDesign command again, or the GUI debugger to have a look at timing now.

How has timing improved? Or did it improve at all?

5.7 Clock Tree Synthesis

During clock tree synthesis, the tool will try to balance the clock tree to minimize the clock skew.

First, have a look at the *cts.tcl* script. Besides some configuration commands, a Clock Tree Specification File is generated by the *createClockTreeSpec* command.

Find the file and have a look at this content.

What clock skew seems to be set as acceptable by the tool?

The *clockDesign* command will run the clock tree synthesis step.

After the clock tree has been created, consult the timing reports for the reg2reg paths.

How are they different from the earlier ones, or the synthesis timing reports?

Can you identify which timing slack belongs to which component on the pipeline stage?

Can you identify the clock skew on the worst path?

5.8 Post CTS optimisation and Hold Timing Optimisation

At the end of the Clock tree synthesis, some timing reports for the hold violations were created.

Are there any hold violations?

The *optDesign* command can be used to fix setup and hold violations.

So far we have just fixed setup violations to try to get timing as close as possible to the target. However, at some point, we must make sure no hold violation are present, and then get setup violations as good as possible.

Remember that setup violations happen between clock periods, so that if some violations are present, reducing the clock frequency will help solve the problems. Hold violations however, happen on the data path, meaning that if a path is too fast, reducing the clock frequency won't make it slower.

Try to call the *optDesign -postCTS -hold* command, or have a look at the cts_opt.tcl script.

Were hold violations fixed? Can you see on the terminal what happened to the design during fixing?

5.9 Design Rules Verification

At this point, the tool has already added a lot of structures in our design, and might have made some mistakes.

It is then wise to often verify the design for design rules violations, like overlapping cells, and call some special commands to fix them. A typical example would be that some cells are overlapping, in that case, calling the *refinePlace* command would move the cells around to solve those issues.

Call the *verifyGeometry* command to perform a simple physical check on the design.

What kind of errors did the tool find?

Are there any overlapping cells? If yes, call *refinePlace* and rerun *verifyGeometry*.

5.10 Route the design

Now we can jump on to routing the design.

As usual, the router must be configured depending on how the routing results should look like, especially for VIA creation.

The route2.tcl script gives an insight on these configuration steps.

What happens before routing starts?

How does timing look like after routing for both setup and hold views?

Call *verifyGeometry* again, is it better or worse?

5.11 Save the design

Save your design using the saveDesign command as presented earlier

5.12 Finishing the design

As was briefly presented in the lecture, finishing the design consists mostly in:

- Verifying Physical Rules, with “verify*” commands or external tools
- Fix the issues, retime the design and make sure everything is ok
- At the end, perform metal filling if necessary and stream out a GDS file

These steps can require a long time, and sometimes even custom fixes done per hand.

You can have a look at the metalfill.tcl and streamout.tcl script for a glimpse at metal filling and outputting a GDS file (don’ run this script as the output would be too large for your account).

If you have arrived so far, you have seen most of the required steps to create a digital ASIC.

6 Post Place and Route Simulation with Timing Information

Finally, we will try to simulate our design using the post-route Verilog netlist. We will also extract the timing information and have a look at the delays in the simulation waveform.

To do such, you can have a look at the streamout.tcl file, you will find two commands which create the necessary outputs:

- ***saveNetlist*** is used to write the Circuit to a Verilog file, this is pretty much the same as the ***writeHdl*** command in RTL compiler.
- **write_sdf** is used to save the wire and gate delay extracted from the routed design to a file, called an SDF file, which can be used by the simulator to display the signal delays.

If you take a closer look at the saveNetlist command, you will see we used the argument -inclusePhysicalCell.

The standard behaviour of the saveNetlist command is to save back the logic circuit only, without the Physical Cells, which are all the Cells added so that the circuit will work once produced, meaning without the cells used to dispatch the clock tree, the power cells, the well tap cells, the Tie High/Low cells etc.

In our case, we requested to keep the cells used during clock tree synthesis. This way, we should be able to see the delay on the clock line between the moment it is generated by the simulator, and the moment it actually arrives at a Flip Flop.

6.1 Prepare the timing delay File information for the simulator

The Simulator needs to read the SDF file to display delay information. Sadly, we have to add a configuration file and compile the SDF file first. This step has been prepared for you:

- In the lab2.zip file from ILias, you will find a simulation folder, which you can extract in your lab2 folder
- Move into the simulation folder and edit the “compile_sdf.bash” file:
    - Modify the *sdfFile* variable to point to the SDF file counter_top.worst.sdf located in the stream_out folder from your encounter run folder.
    - Don’t forget to load the tools in your current terminal
    - Run the compile_sdf.bash file by executing the following command:
        - bash compile_sdf.bash
    - Execute the “ls” command, you should see a file called counter_top.worst.sdf_compiled in the output list.
- Now we are nearly ready to run the simulator as in the previous exercises:
    - Edit the “counter_top.routed.f” file
        - Modify the second line to point to the counter_top.routed.v file located in the stream_out folder
    - In the counter_top.routed.f file, you can see the usage of the argument:
        - -sdf_cmd_file counter_top.routed.sdf_cmd
    - If you have a look at the counter_top.routed.sdf_cmd file, you will see the compiled SDF file referenced as delay source, and the MTM_CONTROL = "MAXIMUM"

parameter means we want to display the MAXIMUM timing information, in other words the slowest times.

- Now you can run the simulation by running the command:
  - irun -f counter_top.routed.f -gui
- Look at the command Line output:
  - If some errors appear, like a design unit not found, make sure the path in counter_top.routed.f is correct
  - You should see two lines similar to these:

```
Annotation completed with 0 Errors and 210 Warnings
SDF statistics: No. of Pathdelays = 15087  Annotated = 18.63% -- No. of Tchecks = 7798
Annotated = 16.95%
```

- If all is ok, we can try to have a look at the timing output

6.2 Understand an Interconnect delay path

If you open the stream_out/ counter_top.worst.sdf, you should find at the beginning, a line very similar to this line:

```
(INTERCONNECT clk_int__L3_I5/Z counter/value_reg\[7\]/CK  (0.015::0.015) (0.015::0.015))
```

If not, try to look for such a line.

This Line means:

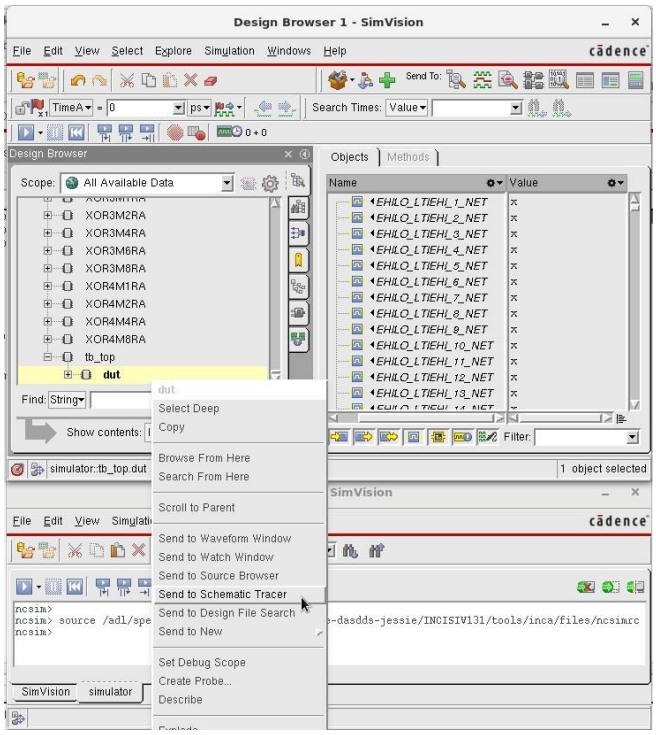
- **INTERCONNECT** delay, in other words, the time required by a signal to propagate on a wire between two pins of some cells
- **clk_int__L3_I5/Z** is the wire starting point, here it means the pin "Z" of the Cell named "clk_int__L3_I5" which will be a clock buffer, as the name starting with "clk" can indicate. The Z pin is typically the output, the input would be named "A".
- **counter/value_reg\[7\]/CK** is the wire arriving point, here the Clock input of the Flip-Flop holding the value of 7th bit of the counter
- **(0.015::0.015) (0.015::0.015)** is the delay for SETUP and HOLD, each timing is a triplet (SLOW:TYPICAL:FAST). The TYPICAL value is empty for us, because we only have SLOW and FAST setup.

Your SDF file might be slightly different than the one used while writing these lines. Just try to look for a similar INTERCONNECT line, it doesn't have to the 7th bit for example, it could be another one.

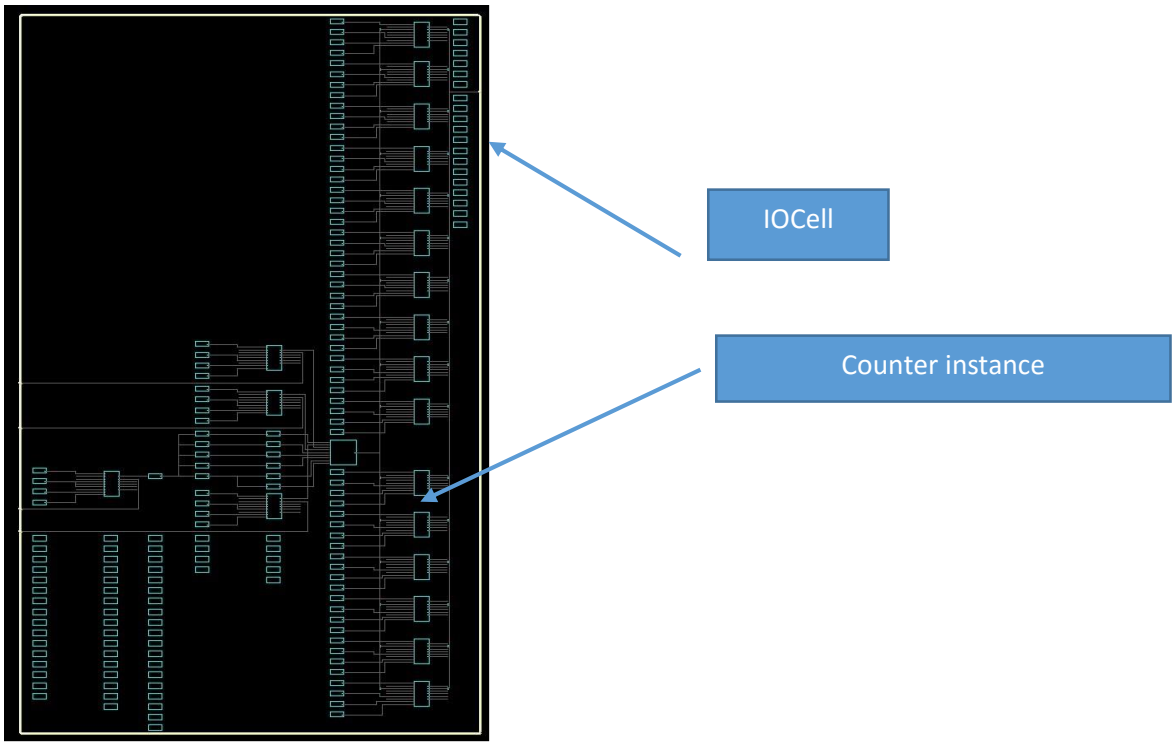
6.3 Display the interconnect delay

In the main Graphical Interface of the simulator, we can use the Schematic Viewer to find the value_reg[7] flip flop, and the clk_int__L3_I5 clock buffer.

To do so, open the schematic tracer on the “dut”:

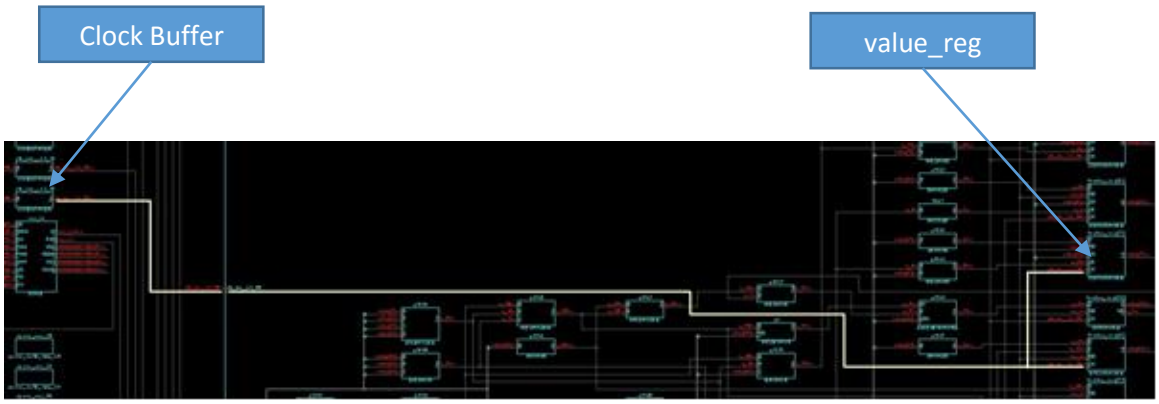


You should get a schematic view of counter_top, where you can see the IOCells, some top level clock buffer cells etc.:

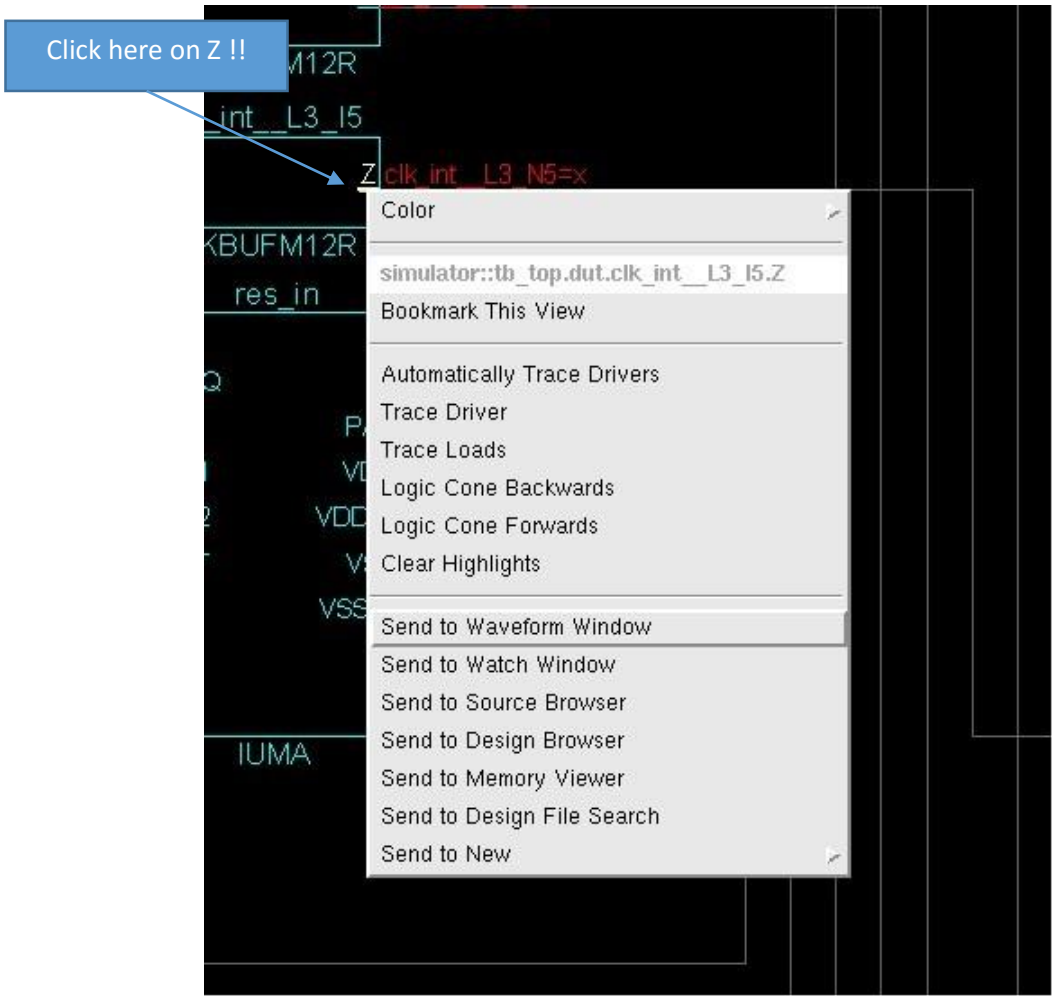


Now you can zoom around the counter instance, and double click it to open its internal schematic.

In the internal schematic, you can look for the value_reg Flip-Flop matching the bit number of the INTERCONNECT path we are looking at, in our case the 7th. If you click on the wire coming to the CK pin, you can Highlight it and trace it back to the Clock buffer:



To watch the delay in the simulation output, we can add the Z and CK pins to the waveform. Be careful, we want to see the signal at the Z and CK pins, not the wire itself. To do so, you have to click on cell's PIN not the wire, like this:



Repeat the operation to the CK pin on the Flip Flop.

Finally, add the clock from the test bench to the waveform as we did in the Lab1 to see the starting time of the clock.

Your Waveform should look like following: