ENSC450: VLSI Design

# LAB Tutorial: Napkin to Chip

This tutorial will guide through all the EDA material that is necessary to build a small CMOS digital macro performing a simple DSP task. It will be intended as a step-by-step guide through the use of the lab environment and the tools.

As reference task, we will use a very simple image processing task: RGB2grayscale conversion. Bewteen the various available methods for RGB2Grayscale we will use the lightness method as it accounts for a very straightforward hardware implementation: The **lightness** method averages the most prominent and least prominent colors: we will be working on 8-bit pixel and for each pixel

**(max(R, G, B) + min(R, G, B)) / 2.**

This application could be used by a processor receiving data from a digital camera and performing some security applications such as Motion Detection, Object recognition, etc that are typically performed on grayscale 8-bit images. In case many cameras are working concurrently, it could make sense to have a single fast and cheap IC performing the security processing, rather than a big FPGA, and this logic could be reused in this context. Depending on how fast and efficient we need conversion, we could choose an old but cheap technology node such as CMOS180nm or a recent fast but very costly technology node as CMOS045nm.

## 1. Front End Design

### a. HDL Coding and Simulation

The Front End design implies HDL coding, Simulation, constraints definition and Synthesis in two different technologies, and results estimation.

*In order to build our working environment we need to log on a Unix machine. We need to open a working terminal that we reserve for FE: Right click on the background, choose “Open Terminal”.*

*We will name the terminal “Front-End” with the menu “Terminal-> Set Title”*

It is advisable to perform the following command that will help you to build a more comfortable Unix environment:

```
source  ~fcampi/.mylogin
```

*We will then build and enter to a specific working area:*

```
 mkdir rgb2grayscale

cd !$
```

*Inside this area, we will build the following subfolders*

```
mkdir vhdl

mkdir sim

mkdir syn_045
```

*In terms of EDA tools for Front end we will need Mentor ModelSim for HDL simulation and Synopsys dc_shell for synthesis. To use them, we must set the environment and the appropriate licensing with the commands*

```
source /ensc/fac1/fcampi/setup/FE_setup.csh
```

*IMPORTANT NOTE: This source command is used to properly configure your terminal. If you start a new session, or even if you simply open a new terminal (Terminal==Command Window) you will have to run this command again. Try not to use the same terminal for different purpose. If you run the source command above, and you need to use a different tool, do not run a new source command on the same terminal, just open a new terminal for the new tool right-clicking on the background. It may also be a good idea to name the terminal window from the gui (Terminal-> set title) to make sure you remember what is the terminal purpose.*

We will start with typing our HDL solution and then trying to simulate its functioning and eventually synthesize it over the available technology supports.

Our hardware will not be a standalone IC, but a digital macro that can be included in some more complex design. We don't know the delay at which we will receive our inputs. So, rather than using set_input_delay statements that will impact our speed, it is wise to sample  input data as soon as they arrive in our macro, and sample them again before they exit so that we don't impose any delay on whatever logic is following our design. The complete code is described in file */ensc/fac1/fcampi/Tutorials/rgb2grayscale/*vhdl/rgb2gray.vhd .

*You can use the text editor of your choice for typing vhdl code. The most used editors are vi (that will use your terminal) or emacs (that is a graphic tool so will leave your terminal free). If you use emacs make sure to invoke it as "background" emacs vhdl/rgb2gray.vhd &". The "&" will make so that your terminal does not get stuck waiting for emacs to terminate.*

*In order to simulate our code we need to build a testbench, that is a Non-synthesizable vhdl that will feed test inputs to our logic in order to check its functioning*

The testbench is described in file Tutorials/rgb2grayscale/vhdl/tb_rgb2gray.vhd . In order to simulate the correctness of our code we will use the Mentor ModelSim HDL simulator, which is the state-of-art  tool for simulation. Since we don't know the timings yet before technology mapping, we will use a conventional 20ns period. We could use any other time, in RTL simulation the time is not significant as we don't know yet the specific cell delays.

Mentor ModelSim need a configuration file "modelsim.ini" containing simulator settings,  you can utilize the one in the reference folder as described below:

*cd sim*

*cp /ensc/fac1/fcampi/Tutorials/rgb2grayscale/sim/modelsim.ini   .*

*mkdir scripts*

*emacs scripts/compile.csh &*

*It makes sense, especially for HDL projects involving many files, to build a custom script for compiling all files in the correct order. See an example in* file */ensc/fac1/fcampi/Tutorials/rgb2grayscale /sim/scripts/compile.csh*

*You can run it with the command source scripts/compile.csh or make it executable chmod +x scripts/compile.csh and run it like an executable ./scripts/compile.csh*

*Once the compilation does not signal any error we shall proceed to simulation, running the command*

*vsim E &*

*From the vsim command window run : source scripts/simulate.tcl*

*Again, see an example in file /ensc/fac1/fcampi/Tutorials/rgb2grayscale/sim/scripts/simulate.tcl*

In simulation, please note the time shift of 2 clocks between the inputs and the corresponding output, caused by the two input and output registers. Inserting these registers we do not add to the critical path the delay of the inputs from the preceding blocks and the processing of the outputs from the following block in the overall IC, so can make our logic work faster. Hence, we process more data per time unit (this is called throughput) but we increase the time between the arrival of inputs and the availability of outputs (called result latency)

## b. Synthesis

For logic synthesis we will utlize synopsys design compiler, by means of its command shell dc_shell-xg-t. We could also utilize a GUI version, design_vision, but as we discussed utilizing scripts allow for much better documentation and control of the design process. Design_vision can be useful in case we need to depict our synthesized design and its critical path, although even in that case it may be preferable to browse the Verilog netlist rather than depicting the design graphically.

Synopsys design compiler  needs two configuration files, .synopsys_vss.setup and .synospys_dc.setup. In our context we will only utilize the first one, that we will compy form the reference folder as described below.

*In order to run technology synthesis of the block we have built, we need to create a new workspace:*

```
mkdir ../syn_045

cd  ../syn_045

mkdir scripts

mkdir results

mkdir work

mkdir logs
```

*in the folder scripts we will build the synthesis script: use as reference /ensc/fac1/fcampi/Tutorials/rgb2grayscale/*syn_*/scripts/synth.tcl

Please check and understand carefully the script, in particular adapt all constraints to your specific design environment. These constraints are the key to implementation, and will drive all future design steps, starting from synthesis all the way to the Place&Route and final checks of our design.

*Before we run Synthesis make sure to copy the following configuration file that will ensure that the Synopsys dc_shell tool will use the folder work for its temporary files.*

```
cp /ensc/fac1/fcampi/Tutorials/rgb2grayscale/syn_045/.synopsys_vss.setup   .
```

*To run the synthesis process we use the following command:*

```
dc_shell-xg-t –f scripts/synth.tcl
```

In order to understand the design tradeoff that we have we need to understand the best possible achievable performance and the power/area penalty that we would have to pay.

So we want to analyze (1) Power and area of an unconstrained design (unconstrained, with no or very high period such as 500ns) (2) Power and area of the fastest possible solution (3) A few points in between.

*Synthesis results are available in folder results. The folder contains*

1. *Verilog file that represents the"netlist", that is the list of cells that represent the technology equivalent of the behavioral code we synthesized*
2. *An internal ddc file, which is the synopsys database for our design*
3. *A sdc file, that represent the set of constraints we have used for synthesis*
4. *The .rpt file*

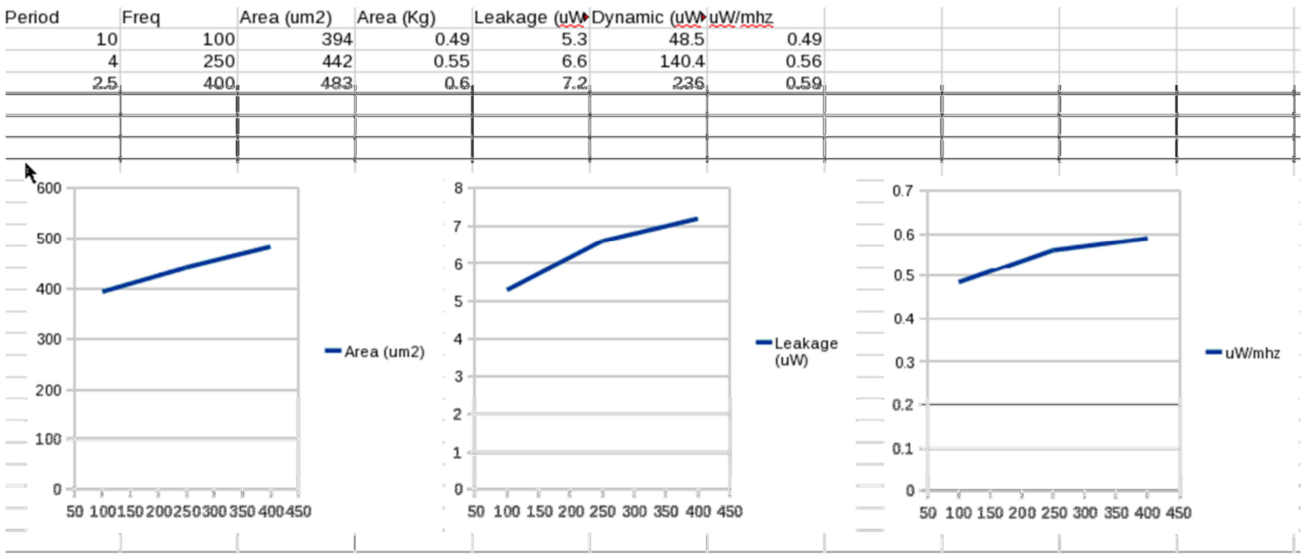
*In particular, the report (.rpt) file contains all information to evaluate the quality of our design. It is composed of 3 different reports:*

- *AREA report, where all Standard cells composing our design are listed, with the area of each cell and the total overall area. Note that area strictly depens on technology, if we were to synthesize the same* *design on a different technology we would have significantly different results. Sometimes, in order to express only the complexity of the design, independently from the chosen technology, it is customary to express the area of a given block in "Kilogates", where a gate is the smallest 2-input nand in the library. So Area(Kg)=Area(Block)/Area(Nand2).*

- *TIMING Report. We define path a "timing arc" between a start pin (can be an input pin of our block or a Q pin of an internal FF) and an end pin (can be an output pin of our block or a D pin of an internal FF). The tool will calculate the path with the longest delay, and will compare that with the Clock period we required, to verify if the timing constraints for the circuit are being met*
- *POWER Report. Power consumption of a given IC is composed by Leakage power (Power consumed by the circuit during stand-by) and Dynamic Power (Power consumed when Clock and inputs are toggling)*

*We normally report Leakage and dynamic power separately as the second depends on the clock frequency and the first does not. In order to report dynamic power in a way that is independent from technology, very often Dynamic power is expressed in mW/MHz (Dividing the dynamic power for the frequency value)*

*The fastest solution is the fastest design that does not create violations, we can obtain it (roughly!) with a few attempts. We can use excel or the Linux open office (command ooffice &) to build graph, use the file /ensc/fac1/fcampi/Tutorials/rgb2grayscale/*syn_*/rgb2gray.odb as reference. Here is an example of results obtained with cmos045nm technology.



| Period | Freq | Area (um2) | Area (Kg) | Leakage (uW | Dynamic (uW | uW/mhz |
|---|---|---|---|---|---|---|
| 10 | 100 | 394 | 0.49 | 5.3 | 48.5 | 0.49 |
| 4 | 250 | 442 | 0.55 | 6.6 | 140.4 | 0.56 |
| 2.5 | 400 | 483 | 0.6 | 7.2 | 236 | 0.59 |

During synthesis, it is possible to fix hold to explore the impact on timing of such activity. But since hold times depend very much on the clock tree clock skew, it makes really not much sense to perform that at this stage, so it is advisable to let the tool work only on setups and leave hold fix to stages following the implementation of the clock .

## c. Design Optimization

If we want to achieve higher speeds, and higher throughputs, we can try to break the critical path of the design: analyzing reports we can see that the critical path is divided as such:

1.75 ns Max/Min calculation

0.66 ns Final Sum and shift

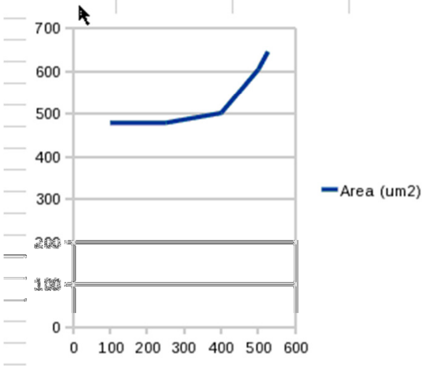
If we break the critical path in HDL we may reach better results, but we may pay a penalty in area and power. *Let's explore the related tradeoffs: in file /ensc/fac1/fcampi/Tutorials/rgb2grayscale/vhdl/rbs2gray_pip.vhd we find a pipelined version of the same architecture.*

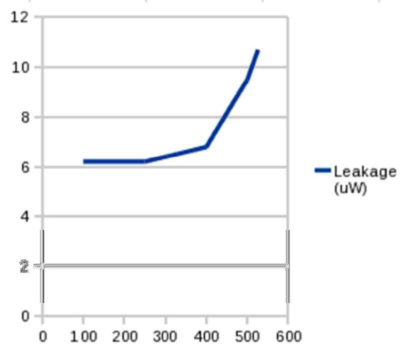
We can see that we are able now to reach 500MHz, but this comes at a price of a rough 20% additional area, 30% additional leakage, 15 % additional power per sample.

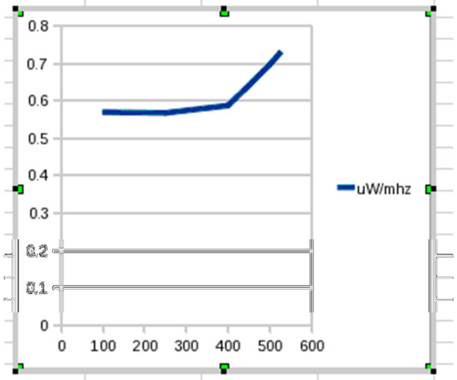
This leads us to think that this choice (more design costs, more mask costs, more battery costs) is significant only if that additional 100MHz allows us to gain a relevant market share, otherwise is not significant.

PIPELINED SOLUTION

| Period | Freq | Area (um2) | Area (Kg) | Leakage (uW | Dynamic (uW | uW/mhz |
|---|---|---|---|---|---|---|
| 10 | 100 | 479 | 0.6 | 6.2 | 57 | 0.57 |
| 4 | 250 | 479 | 0.6 | 6.2 | 142 | 0.57 |
| 2.5 | 400 | 502 | 0.63 | 6.8 | 235 | 0.59 |
| 2 | 500 | 604 | 0.76 | 9.5 | 349 | 0.7 |
| 1.9 | 526.32 | 646 | 0.81 | 10.7 | 385 | 0.73 |







Note: Many synthesis project show the elbow featured by the plots above. It shows where the synthesis tool is starting to adopt drastic solution in order to meet timing. It is likely that such drastic solutions may not remain valid during P&R, where conditions (especially wire delay and clock distribution) will be different so it is normally advisable to avoid the elbow and remain as near as possible on the plateau of the plot. It is advisable to avoid at least the 530MHz solution, and even the 500MHz may be a bit risky, and may cause some degradation during Back-End

## d. Post Synthesis Simulation

A very common error in writng HDL circuits for VLSI systems is to write simulable code that is not synthesizable. As we know, HDL was designed to simulate and document hardware systems, *NOT DO DESIGN THEM*. We can design on a very reduced subset of possible HDL constraints, and we must apply special care when writing HDL code for synthesis. What is worse, many HDL constraints have been added to FPGAs but are *NOT* valid for HDL synthesis.

Often the HDL synthesis tool will issue errors or warnings to detect such situations, but ore often our code is perfectly legal, but in fact it does not correspond to what we would like to do: it is simulable, and it yields correct HDL simulation results, but synthesized over a VLSI system it causes unwanted results that are not functionally correct. A very typical case is a FF statement with inputs in the sensitivity list, that may cause unwanted parasitic latches or produce incorrect output.

To make sure that this does not happen in our designs, a useful option is to run a modelsim simulation over our synthesized netlist. We can compile the Verilog produced by design_compiler, toghether with a specific Verilog file describing the appropriate behavior for each cell [this last file is available in most library installations including our FreePDK 045 libs]. In this way we can verify (for our inputs) that the netlist has the expected functional behavior. This procedure is described in the script sim/scripts/compile_netlist.csh .

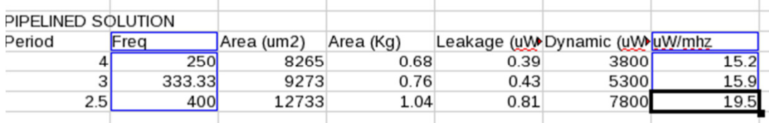
If the compilation of Library functional Verilog file [NangateOpenCellLibrary_PDKv1_3_v2010_12/Front_End/Verilog/NangateOpenCellLibrary.v] and design netlist file [syn_045/results/rgb2gray.ref.v] are correct, we can then use ModelSim in the same way we did with behavioral HDL. Now we will be able to simulate the inputs and outputs OF EACH CELL in our design, and we should check that the simulation outputs are the same as in the behavioral simulation.

## e. Trial on cheap technology: cmos180nm

Once we have understood limits and potentiality of this 045 technology, we may wonder if we could obtain comparable results with a cheaper technology, and evaluate whether it really makes sense to spend so much on a 045 mask-set or we could have comparable results with a cheaper node. 180nm was leading technology around 1999/2000, but it is still offered by various fabs, and its mask/diffusion costs may be ten times smaller than 045.

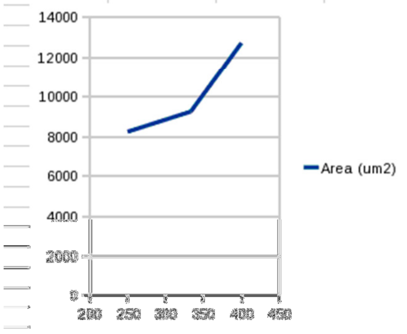
Let’s analyze this tradeoff with synthesis before we do the same after implementation.

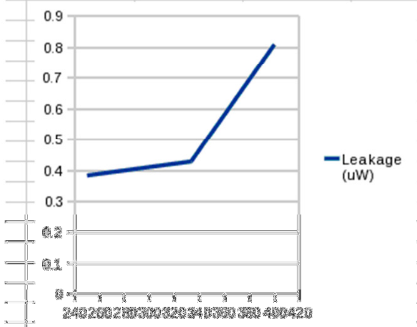
*From a scripting point of view it really does not change much: as described in /ensc/fac1/fcampi/Tutorials/rgb2grayscale/syn_180 we only need to change the specification of the target and link technology. We will only work for simplicity on the pipelined solution!*

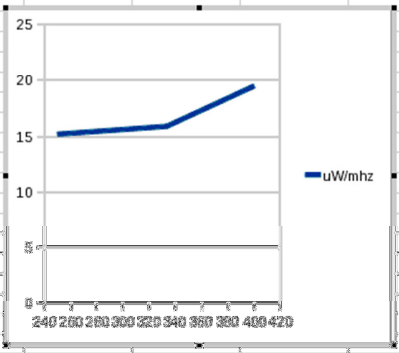


PIPELINED SOLUTION

| Period | Freq | Area (um2) | Area (Kg) | Leakage (uW | Dynamic (uW | uW/mhz |
|---|---|---|---|---|---|---|
| 4 | 250 | 8265 | 0.68 | 0.39 | 3800 | 15.2 |
| 3 | 333.33 | 9273 | 0.76 | 0.43 | 5300 | 15.9 |
| 2.5 | 400 | 12733 | 1.04 | 0.81 | 7800 | 19.5 |







*In terms of performance, we don't lose much. Peak speed is 2.5ns after synthesis, and used to be 2 ns in CMOS045, so it is definitely not worth the huge additional cost for 0.5 ns per pixel.*

*On the other hand, area and dynamic power are 20 times larger, while leakage power is 10 times smaller.*

*The area issue may not be dramatic, since area impact mask costs, but mask costs in 180 are way smaller than those of 045. The power consumption problem, especially if we have to target systems that are not plugged to the power grid, may be more critical. We should wait until after the place and route for a final comparison*

## 2. Back End Design

In order to start the implementation we need to choose one of the netlists produced by synthesis.

We must be aware that the conditions will dramatically change from the FE phase:

1. Wire delays will not be estimated by models but extracted from real wires resistance/capacitance
2. The clock tree will not be ideal, but described by a tree of buffers with a given skew between each leaf (FF)
3. Following Clock tree definition, hold violations will need to be fixed

The BE tools will

(i) Support the user in defining a Floorplan, that is a silicon area, with I/O pins and macros, where the cells shall be placed
(ii) remove all existing buffers from the Synthesis netlist,
(iii) place the remaining cells over the floorplan
(iv) perform clock tree synthesis
(v) Route connections between cells

In order to fix Setup, Max Cap, Max Tran violations induced by placing and routing and to fix hold violations, the tools will insert buffers. As a consequence:

a. the timing results obtained during synthesis may suffer degradation, often very significant. This is beacause during synthesis the wire parasitic were only very roughly estimated (In Verilog there is no specification of the length of a net or the metals used to build it, while this information is available during P&R).
b. The tool will need space to place these additional buffers, so we need to reserve floorplan area for this
c. Sometime the routing will need a little additional place to avoid congestion, depending on the number of metal layers available

In conclusion, we can expect a degradation of timing, and of the estimated area. Our aim is to control this degradation in order to meet the design specification!

The input for our BE activity are:

DESIGN INFORMATION:

1. Verilog Netlist derived from synthesis (.v)
2. Constraints file derived from synthesis (.sdc)
3. Specific set of constraints for Clock Tree Synthesis (.cts)

TECHNOLOGY INFORMATION

4. Timing libraries (Liberty) for STA and Physical libraries (LEF) for describing space occupation of each cell
5. Capacitance and resistance information of wires (Captables)

*The tool will be using for BE is encounter, by Cadence. We will start by building a specific environment. It is advisable to open a separate terminal specific for our BE activity.*

*In order to use encounter we need to prepare the environment:*

*Source /ensc/fac1/fcampi/setup/BE_setup.csh*

*mkdir BE_045*

*Cd !$*

*mkdir scripts*

*mkdir results*

*mkdir scripts*

*mkdir runs*

*Please refer to /ensc/fac1/fcampi/Tutorials/rgb2grayscale/BE_045 for template examples.The file *.conf are used to specify all inputs to encounter.*

*The backend procedure is divided in the following steps:*

1. *Import Design*
2. *Floorplanning & Power Distribution*
3. *Placing*
4. *Post-Placing Optimization*
5. *Clock Tree Synthesis*
6. *Post-CTS Optimization*
7. *Routing*
8. *Finishing*

*We will use a separate TCL script for each step, excluding the floorplan that is normally interactive by GUI*

*Most design flows include a post-routing optimization phase to fix violation induced by routing but in our exercise we will skip that step for simplicity.*

Note that in many libraries (not all!) there are specific buffers for Clock Tree. In fact, one could use any buf or IV to build a clock tree, but specific buffers are designed in order to (i) have a better Output transition on the raising edge, since it is the only one that counts (ii) be more resistant than average buffers to on-chip variation in order to limit variation on the clock skew that would impact all paths in the design.

## Import Design

*Encounter has both a GUI and a shell for launching command. Do not run encounter in background or it will freeze missing its console:*

*encounter*

*In the encounter shell we can launch command scripts from the scripts folder:*

*Encounter> source scripts/01-importDesign.tcl*

*From the encounter console we can get the description of a command (well, most commands)*

*Encounter>man <command>*

*Remember you can use most unix commands such as ls, pwd, less in the encounter console. See also the manuals in /ensc/fac1/fcampi/manuals/ [encounter.pdf | soceUG.pdf]*

The inputs for our P&R activity are Configuration file, Verilog Netlist and sdc file (that must be copied in folder inputs).

For each run, encounter will produce very useful log files called encounter.cmd# and encounter.log# , where # is the number of times we have been running encounter from the same folder. Old logs can be useful as a reference, but don’t accumulate too many of them in the folder or you will be very confused.

.cmd files contain the list of all the commands launched in the session history, whil e.log will keep track of all commands and relative outputs.

It is highly recommended to check the log file for each command, especially when launching long script, as it is difficult to follow all steps of the computation. In particular, use the Unix grep command from a free terminal (not the encounter console) to navigate logs and spot the keywords error and warnings.

grep –i "error" encounter.log

grep –I "warning" encounter.log

Errors most invariably means that something was wrong or the constraints (Period, density) were too tight and encounter cannot complete. Warnings often can be accepted but they should be checked and understood

Note: Encounter has a command window (on your terminal) as well as a graphic window. You will mostly work on the terminal window to run your script and commands but check often on the GUI the effect of your commands and how the design will evolve from the Verilog netlist to its completed form. The GUI can be navigate with most of the commands used to navigates layouts with cadence Virtuoso: ctrl-F will outzoom to the whole layout, shft-Z will zoom out, while right-clicking on an area of the design will zoom in.

You can make an item or a MOS layer visible and/or selectable with the menues on the GUI, and if you select an item, with q you will get a report on its logic function and the layer it is made of.

## Floorplanning

At the beginning we have to define in the conf file a given cell density in the area.

*The import design step will build a basic square floorplan based on the density information specified in the inputs/*.conf file. We can override this information to our pleasure using the GUI menu floorplan->specify floorplan.*

*In our example we have no IO pads to place, since we are building an embedded block, and no analog macro such as Memory or PLL/DLL. The only significant object in our Floorplan will be IO pins.*

*Since we have a very small design, it is not necessary to add a full grid, to avoid routing congestion we will utlize only vertical stripes. This technology has no well taps. Also,  end caps do not exist so we neet to route M1 stripes up to the ring.*

*It is possible to do everything by scripting, as described in the reference folder, but to have better visual control on the floorplan it is also possible  to use GUI:*

1. *Create floorplan usingFloorplan-> Specify Floorplan*
2. *place pins using the GUI Pin Editor*
3. *Route Power ring and stripes using Power Planning -> Add Ring / Add Stripe*

*As a rule, it is more convenient to use scripting also as a form of documentation. You can find the equivalent of your GUI commands in the encounter.log* files in the work directory. All commands can be part of a script or launched one by one in the console window on the terminal were you launched encounter.*

## Place and Route

Place and route is performed launching the appropriate TCL scripts in folder script. At each step timing analysis is provided, as well as a short analysis of the netlist size.

The procedure is automated, following the scripts as described by the "root" script top.tcl .

*Please at the first iteration don't run the scripts but try to launch the commands one by one and understand what they perform, with the help of the encounter man <command> utility.*

*Do not change the setting of the scripts unless you are at a very advanced stage. If you are not happy with your result, you can change the design by means of two parameters:*

1) *Change the floorplan, in particular changing the placing density*
2) *Change the constraints, enlarging the period or changing Input/Output delay or the clock tree specs*

*In particular, it would be interesting to replicate the P&R activity for different values of placing density, starting from 2%0 up to 90%. It is expected that larger density would cause higher delays at first (longer wires are more capacitive) but at a given point the routing congestion would be such that will create longer wires even if the space is smaller, with an increase in max cap and max tran violations, to the point where the tool will not have enough space to place clkTree and hold/setup fix buffer and will not be able to conclude.*

*Before you move on to result analysis check the session logs for errors and warnings, and inspect visually the database to see*

1) *Are Power stripes in place?*
2) *Are cells full placed over the row area?*
3) *Is routing present?*
4) *Is there any obvious violation highlighted by the tool?*

## Results Analysis

*Timing: For a quick assessment of the timing behavior of the design at the various stages, check the file results/timing/<step>.hold|setup/<design>.summary*

*This file will briefly resume the paths of the design and the relative violations. The setup analysis contains also Max Transition and Max Cap violations. More details are available in the same folder, in particular the files results/timing/08-<step>.hold|setup.rpt contain a detailed analysis of the critical path, but this should only be checked in case of issues.*

*We can check how the number of cells in the block have evolved in our design with the unix command grep to quickly browse our reports (Check the grep syntax online).*

*In a Unix shell (NOT the encounter console) we can run*

```
grep" Instances:"  results/summary/*.rpt
```

```
grep "Standard cells(Subtracting Physical Cells)"  results/summary/*.rpt
```

```
grep "Chip Density #2" results/summary/*.rpt
```

*For our case we obtain the following results:*

| *Stage* | *# of cells* | *Cell Area (um2)* | *Density* | *Violations* |
|---|---|---|---|---|
| *Post-Synthesis* | *322* | *574* | | |
| *Placing* | *316* | *571* | *0.31* | |
| *PostPlace OPT* | *311* | *520* | *0.28* | |
| *Cts* | *318* | *529* | *0.29* | |
| *Post-CTS OPT* | *309* | *523* | *0.28* | |
| *Routing/Signoff* | *309* | *523* | *0.28* | |

*We can see that there is no cell increase in the P&R stages, contrary to our expectations (except for clkTree insertion).*

*This is due to the fact that our design is very, very small, and the 045 technology is so dense that for such a small design the parasitic wire of the loads are so small, that they are almost negligible. So the design did not need any specific optimization, and is capable to run at 2ns without increasing its cell area, and with a very dense placing.*

*At the end of the P&R, depending on thelevel of density chosen for the floorplan the tool may have caused a few DRC (Design Rules Check) violations, such as shorts, antenna errors, spacing errors. Depending on the time we want to invest versus required performance, we may decide to loosen up the specs (especially density) and run another P&R hoping the violations will disappear, or fix them manually on the layout.*

*For the purpose of this exercise it is not necessary to resolve them all, especially if there are only a few of them, it can be acceptable just to understand their reason without fixing them one by one on the layout.*

*Violations can be analyzed from the encounter GUI with Tools-> Violation Browser*

*In conclusion, the performance of our device is*

*Area: 1392.959 um^2  (Core area 638.694 um^2)*

*Core Density:*

## Comparative trial in 180nm Technology

The design can be run with essentially the same scripts. The only parameters that we need to change are

1) Floorplan values such as power grid width, density, boundary dimensions (script floorplan.tcl)
2) Clock tree buffer names (file .cts)
3) Filler Names (script finishing.tcl)

*An exemplar run is available in /ensc/fac1/fcampi/Tutorials/rgb2grayscale/BE_180*