ETH zürich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

EFCL Winter School 2026

Sample Solution Exercise 05

Place & Route

F. K. Gürkaynak
Prof. L. Benini

Last Changed: 2026.02.09

1 Overview

In *Ex. Floorplanning and Power Grid* we learned how to setup the back-end design flow in OpenROAD, we added the pad frame, placed macro blocks, and generated the power grid. In this exercise, we will continue the back-end design flow where we stopped at the end of *Ex. Floorplanning and Power Grid*, and will perform the main tasks of the back-end design: placement and routing. In the process you will learn:

- How to place the standard cells
- How to synthesize the clock tree
- How to perform signal routing
- How to make sure that your chip after back-end design still meets all the timing constraints you have specified

The starting point for this exercise is a saved checkpoint with the same progress as you had at the end of *Ex. Floorplanning and Power Grid* in which the floorplaning and power routing was completed.[1]

1.1 About the Style

We will use a number of different styles to identify different types of actions as shown below:

Student Task: Parts of the text that have a gray background, like the current paragraph, indicate steps required to complete the Exercise.

Actions that require you to select a specific menu will be shown as follows:
`menu→ sub-menu→ sub-sub-menu`

Whenever there is an option or a tab that can be found in the current view/menu we will use a BUTTON to indicate such an option.

Throughout the exercise, you will be asked to enter certain commands using the command-line[2].

```
sh>  command to be entered on the Linux command line
```

1.2 Getting Started

Student Task 1:

- Start by navigating to your home and then the exercise directory

```
sh>  cd ~/ex05
```

- Enter the `oseda` container's shell

```
sh>  oseda bash
```

Note 1: If you ever open a new terminal window or tab, don't forget to re-enter the oseda containers shell, otherwise you won't be able to start the open-source tools.

[1] You can also reuse your floorplan by deleting the `gui::show` in last exercise's **`scripts/01_floorplan.tcl`** and then running it via `./run_backend.sh --floorplan`. This will generate the same checkpoint we are about to load.

[2] There are many reasons for using a command-line, some functionality can not not be accessed through GUI commands, and in some cases, using the command-line will be much faster. Most importantly, things you enter on the command line can be converted into a script and executed repeatedly.

1.3 Reloading saved designs

At any point during this exercise you might do something unintended or you might want to explore one specific step further. Re-running the whole flow takes time, in small designs like this one its only a few minutes which might be acceptable, in big designs this can take hours. So a better way is to create save points and reload from them. For this we have the two functions:

```
save_checkpoint <name>

load_checkpoint <name>
```

You can find all checkpoints in the **save/** directory. The name you need to give the function is the same as the basename of the ***.zip**.

If you want to use `load_checkpoint`, the easiest way is to first start OpenROAD with the startup script to get a clean sheet and then load the checkpoint:

```
./run_backend.sh --open scripts/startup.tcl
```

For the next tasks, you can directly edit and run the prepared **scripts/02_placement.tcl** script instead, which will load the floorplanning checkpoint by itself.

```
./run_backend.sh --open scripts/02_placement.tcl
```

Note 2: As with previous exercises, I strongly recommend you write a script (you can expand **02_placement.tcl**) instead of only entering commands by hand.

1.4 Preparing the netlist

Before we go into the first proper place & route step, we need to 'clean up' the netlist. We will put this into the 'Initial repair netlist' section, there are already a few commands in there, they will be explained as we go along, do not remove them.

First, we want to duplicate constant 0/1 drivers so we don't waste precious routing to connect one constant driver to too many sinks[3].

Second, the buffering and cell sizing (how strong the driver in the cell is) from the synthesis tool might not be what we want in the backend, instead we would like the backend tool to be able to control this fully. Currently, OpenROAD commands dealing with buffering and resizing only add buffers but do not remove them. So to give OpenROAD as much freedom as possible, we strip the buffering from the netlist. Then, to make sure various design rules limiting the maximum fanout, capacitive load and slew rates are fulfilled, we need to repair the netlist by re-adding buffers.

Student Task 2:

- The syntax of the command is:

  ```
  repair_tie_fanout <cell>/<pin>
  ```

- In the technology files (**technology/lib/ez130_8t_tt_1p20v_25c.lib**, find the names of the constant 0/1 drivers.
  Hint: The cellnames contain "tie" because they tie the signal to constant high or constant low.

  Sample solution: `TIELO/Y` and `TIEHI/Y`

- Use the command to replicate these tie-cells

[3] A sink is the receiving end of a signal, an input of a cell.

- Use `remove_buffers` to strip all buffering from the netlist
- Use `repair_design` to re-add the minimum necessary buffers

2 Placement

At this point the netlist is in a state where we can start with placement.

Placement is often split into two parts:

Global Placement : Takes a chip-wide view of the problem and tries to make sure every cell is roughly in a good spot. It is allowed to violate placement rules to some extent.

Detailed Placement : Looks at individual cells and makes sure they are in a legal spot. Sometimes this is further subdivided into one step that still actively tries to optimize the cell placement and one step only concerned with finding a legal (empty) spot to put the cell.

What makes a placement 'good' can sometimes be a difficult question to answer. When looking at the entire IC design process a 'good' placement is one that minimizes the used area, while keeping the design routable and delivering the highest possible operating frequency. However, evaluating these metrics is only fully possible at the very end of the design process, not in the middle where we are. So instead we try to estimate said metrics.

2.1 Estimations

Estimating the operating frequency (among other timings) requires estimates about the parasitic effects of the wiring (mostly resistance and capacitance), as it will delay any signal going through it. There are four 'levels' to estimate these effects, each requires incrementally more information and can thus only be used at later stages in the backend:

Wire load model : Includes in the Liberty files (`.lib`) is a wire load model. The exact formatting of it can differ depending on how much detail the foundry wants to provide. Wire load models are especially useful when we do not yet have any wire lengths and we need to guess their lengths. For this the foundry can evaluate different previous designs and figure out the distribution of wire lengths in it and then construct a 'best guess' from that. Typically there are multiple characterized 'points' representing designs of different sizes. **Important!** Open-STA (the static timing analysis tool used by OpenROAD) is currently not capable of interpreting the way the wire load model is written in our PDK so we cannot use it.

Placement : The next step is to use estimated lengths derived from the current placement of cells, along with characteristic parasitics per unit-length to estimate the total parasitics effects of a wire. This is done using `set_wire_rc` (tells OpenROAD the characteristic parasitics) and `estimate_parasitics -placement` which then constructs the estimates.

Global Routing : We will explore global routing later on but similar to global placement, it represents a rough routing. Using `estimate_parasitics -routing` we can use it to a significantly better estimate than we get from just using placement data.

Parasitic Extraction : At the very end once you have the actual geometry of each wire you can use it model different electrical behaviors and again get more accurate information about the wire parasitics.

2.2 Global Placement

Since we don't have a working wire load model we can't fix any setup timing violations at the moment anyway, so the next step is to perform global placement (GPL).

The OpenROAD online documentation[4] includes a very nice animation showing GPL in action. I highly recommend you take a look.

[4] https://openroad.readthedocs.io/en/latest/main/src/gpl/README.html

As you might notice, there are clusters of cells sticking together as well as a sort of 'connective tissue' of cells holding them together. This is similar to the amoeba we observed in *Ex. Intro to OpenROAD*. Some parts of the design are strongly interconnected because they implement some functionality together (clusters), this is then more loosely connected to other parts implementing something else ('connective tissue').

Student Task 3: Lets recreate this animation for our design (in a reasonable time, so we won't have as many steps).

- Make sure 'NETS/POWER' and 'NETS/GROUND' are turned off, giving us a better view of whats going on
- Use the following command to set up debug mode for the next execution of global placement:

```
OpenROAD > global_placement_debug -update 20 -pause 20
```

- Then run the global placement command

```
OpenROAD > global_placement -density 0.5
```

  We use a lowered target density to speed up the routing phase, it is possible to finish this design with target densities above 70% (0.7).
  It will first run an initial placement phase before pausing, you can continue with the yellow "Continue" button on the bottom left. While it is paused, you can freely use the GUI.

- What do you observe on a macroscopic level, what happens on a per-cell basis, are their 'phases' to global placement?

  Sample solution:

  At first the cells are closely grouped together. Then 'tendrils' start shooting out, the two most prominent going to a central point between the pins of the two SRAM banks. Then it starts to build tightly grouped balls of cells which then rapidly repel from other balls and move very quickly to other places. Then the balls start expanding and filling the space in between, slowly settling on a placement.

- After GPL zoom in, what can you see, how are the cells placed?

  Sample solution: The cells still have a lot of overlap, especially smaller cells do.

By default `global_placement` will target a homogeneous target density of 70% (0.7) and it won't consider any timing constraints. These are fine defaults for smaller designs (such as Croc) but do not work well for large designs. In the following I will introduce what global placement is trying to do under the hood and how you could fix certain problems, especially on larger designs. It is not directly relevant for this exercise so you may skip it or only look at the included picture.

2.2.1 Metrics

Global placement (GPL) uses two main metrics internally. The so called Half-Perimeter Wire Length (HPWL)[5] and the target density or rather how much we are over the target density, the so called overflow.

GPL wants to meet the target density, so minimize the overflow, and it wants less wiring meaning it tries to minimize HPWL. These two metrics are then modeled as forces. One is the overflow force, it only applies to cells in some area where there are too many cells and it pushes them apart. The other force is the wire force, it pulls connected cells together.

[5] imagine taking all pins connected to a common wire, constructing some center point and measuring the length of all pins to this point

2.2.2 Target density

GPL always works with some target density. Without it, we would only have forces pulling cells together and they wouldn't want to spread out. Generally speaking, a lower target density will make it easier to route the design (more routing resources per cell), it makes later cell additions easy to handle (ie from timing repairs later on) and it makes it easier to find a good legalized placement during detailed placement. On the other hand, it makes your chip larger, increasing cost and it may degrade your timing since on average the wires are longer.

2.2.3 Timing driven

Quite often a small handful of wires are critical in the sense that they might not meet our target frequency (or other timings) if they are too long. This makes it attractive to tell the placement tool to prioritize these wires.

This is exactly what the flag `timing_driven` does. Periodically during the GPL process OpenROAD will estimate which wires matter most for the timing, then it will increase the wire force pulling connected cells stronger together. This will on average reduce the wire length between critical cells. There are additional options:

`timing_driven_nets_percentage`: How many wires should be reweighted (get a stronger wire force).

`timing_driven_net_weight_max`: How strong the maximum reweighting should be (if you reweight a lot of cells you can't do it too much).

`timing_driven_net_reweight_overflow`: How early on it should first reweight the wires. If you look at the console output from GPL, the 'overflow' (how much the target density is violated) goes from 1.0 towards 0.1. Setting this option closer to one starts the process earlier, setting it closer to zero starts it later on.
If you start it too early, you might upset the initial clustering or influence too heavily where they should be pulled to. Starting it too late means the effect will likely be smaller.

2.2.4 Routability driven

Certain parts of large designs like a ROM (synthesized to a bunch of seemingly random logic gates) or the scoreboard keeping track of issued instructions in a processor can have a lot more connections between cells than your normal module. This means that per cell, they need more routing resources or put differently, each cell needs more routing tracks on the metal layers above to transport signals. An easy way to give cells more tracks is to decrease the local density so it has to share the existing routing resources with less other cells.

In OpenROAD this is achieved using the `routability_driven` flag. Similar to timing driven, GPL will periodically enter a reweighting step except here we do not reweight the wire force, we change the apparent area of a cell. So the routing is estimated and areas with high congestion are identified. A ratio between the used routing resources and available resources is calculated. Using this ratio, a suitable area increase for each cell in this position is calculated and applied. Finally, using the increased cell area a suitable new target density for this location is also calculated. The overflow increases, pushing the cells apart.

There are a lot of options for this mode, we only highlight the most important ones:

`routability_inflation_ratio_coef`: An exponent applied to the used vs available ratio mentioned above, resulting in the calculated area-increase ratio.

`routability_max_inflation_iter`: A maximum value for the area-increase ratio (applies per iteration, so if you increase the number of iterations you may want to decrease this value).

`routability_check_overflow`: Similar to `timing_driven_net_reweight_overflow` except with an added danger of making the entire GPL unstable. Starting this too early can cause placement to diverge and not find a solution at all. In my experience everything up to 0.6 is fine but its possible (or even likely) this depends on your design. Additionally, we may need to reduce `max_phi_coef` (think step size) to converge on a solution.

`routability_max_density`: Sets an upper limit for the target density. If you are having trouble with `detailed_placement` or timing repairs later on because some areas are too dense, reduce this.

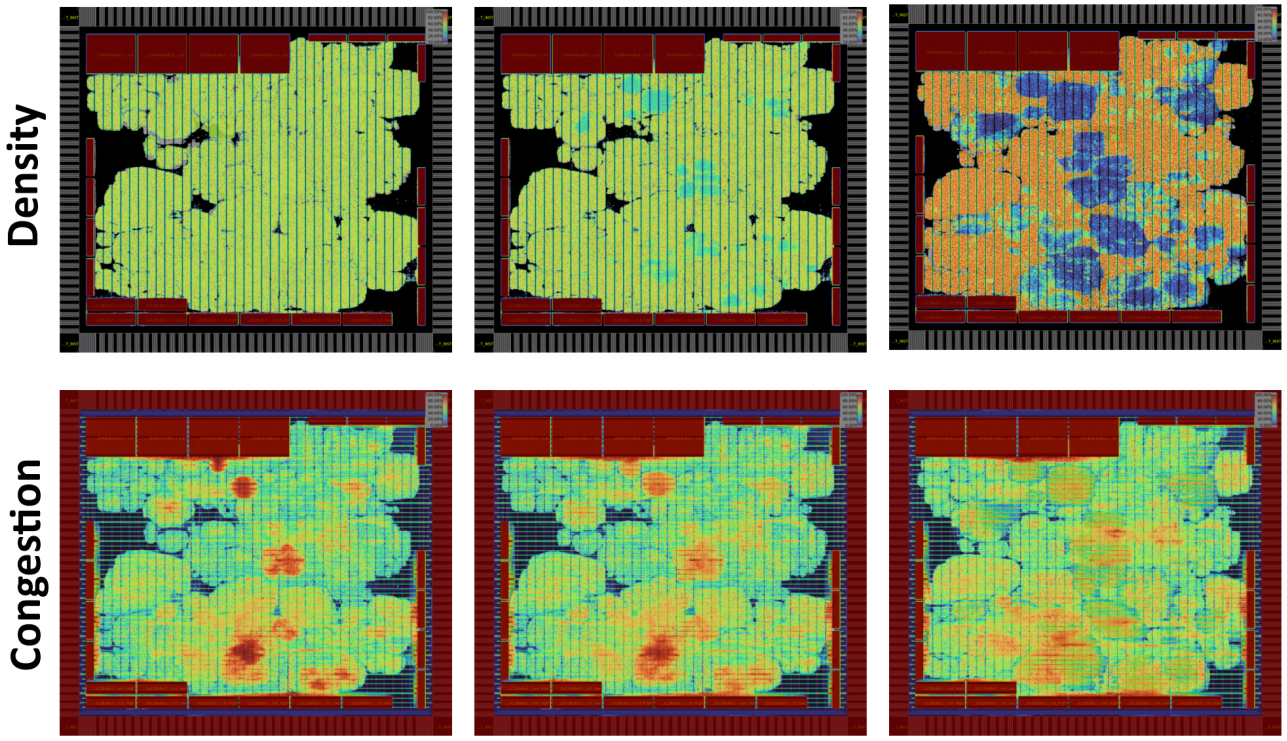


Figure 1: Increasing `routability_mode` strength on Basilisk from left to right. Effect on density (top) and on routing congestion (bottom).

Figure 1 shows a few different parametrization of `routability_driven` on our large design. As you can see in the pictures on the left, having a homogeneous density (no `routability_driven`) doesn't help us as it only increases the congestion in some areas[6]. The middle column show a light use of `routability_driven`, the density in some areas is starting to become smaller and the worst congestion peaks disappear but it still has high congestion and is not easy (or even possible) to route. The last column shows an aggressive use of `routability_driven`. Congestion is basically eliminated but we may now run into other problems as it becomes increasingly difficult to place additionally cells in favorable locations (for timing fixing etc).

2.3 Detailed Placement

Student Task 4:

- While zoomed in, execute `detailed_placement` (DPL), what happens?

  Sample solution: All the cells are shifted into a legal position.

- Run `optimize_mirroring`, what did it do, why does this help us?

  Sample solution: It reduces the total wire length by mirroring some cells.

2.4 Timing Repairs

Now that we have a valid placement, we can use the estimated parasitics from it to repair any timing violations we may have.

[6] Only the ability of GPL to reach some target density locally is important, there is no point in having a globally homogeneous target density, we gain nothing from this.

Student Task 5:

- Report the currently used area and write down the number

```
OpenROAD > report_design_area
```

- First we need to get the parasitics

```
OpenROAD > estimate_parasitics -placement
```

- Then we can fix all setup violations

```
OpenROAD > repair_timing -setup -repair_tns 100 -verbose
```

- Was there a significant area increase?

  Sample solution: No, it only inserted a few buffers and resized a small handful of cells.

- If it did, then it might be enough to influence the placement and it would be a good idea to re-run global placement[a].

[a] This is especially useful if you already have a high density since then it is difficult to place new cells.

Student Task 6:
At this point we are done with the `scripts/02_placement.tcl` script and can move on to the clock tree synthesis. As a last step run the entire placement flow a final time to make sure we have an up-to-date checkpoint:

```
./run_backend.sh --placement
```

Alternatively if something does not work, you can go into the `save/` directory and rename `02_croc.placed.zip.sol` to `02_croc.placed.zip` to use the provided solution for the next steps.

3 Clock Tree Synthesis

The fan-out of a net refers to the number of inputs driven by a particular output. High fan-out nets (that drive hundreds or even thousands of inputs) may need to be handled differently from standard interconnections. Every synchronous circuit has at least one high fan-out net, namely the clock net. The main problem with high fan-out nets is the large load capacitance that needs to be driven. Each driven input adds its own input capacitance to the total load capacitance and in addition, the interconnection required to distribute the signal to all these inputs increases the load capacitance further.

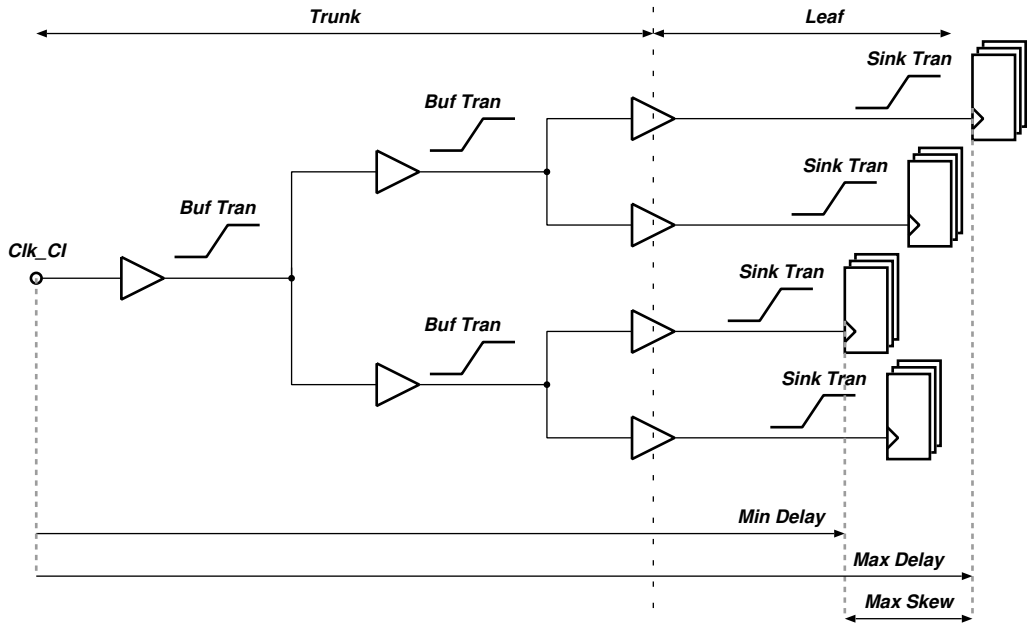
There are three important parameters for such nets:

Transition time This is the time it takes to change the logic level of a node (e.g. $0 \rightarrow 1$). Basically, the more load an output has to drive, the more time is required to charge this load. CMOS drivers consume additional short circuit current during the transition, therefore long transition times are not very welcome. Furthermore, noise on signals with long transition times can result in glitching. Most libraries set an upper limit for the transition time (for the technology we are using this is 0.56 ns for typical libraries). To lower the transition time, a tree of buffers can be inserted so that the total load is shared between the buffers. The lower the desired transition time, the more buffers are required.

Insertion delay The time required for the signal to travel from the driver to the end-points. This delay is usually different for each end-point. Each level of buffers in the buffer tree will add a delay to the signal.

Skew The difference between insertion delays of different end-points. To minimize skew, a balanced buffer tree has to be built. Generally, the lower the desired skew the more buffers are required.

Our main focus today is on the clock, our main concern is to control the skew, since it will affect the timing. The maximum acceptable skew depends on the defined clock period. As an example, for a 10 MHz clock a clock skew of 1 ns is acceptable. But for a 100 MHz clock, the same skew equals to 10% of the clock period and would be too high.



Student Task 7:

- For the clock tree synthesis tasks, work in **`scripts/03_cts.tcl`** and start it with:

```
./run_backend.sh --open scripts/03_cts.tcl
```

- First we need to do some preparation

```
OpenROAD > estimate_parasitics -placement
OpenROAD > repair_clock_inverters
```

  In the initial repair we set the clock nets to don't touch to prevent any other commands from adding anything to the clock nets. Now we need to change the nets we unset this option.
  The `repair_clock_inverters` command makes sure that inverters in the clock nets do not stop CTS from seeing it as one big clock net.

- Now we can construct our clock tree

```
OpenROAD > clock_tree_synthesis -buf_list $ctsBuf -root_buf $ctsBufRoot
```

  This is again a command with many options. Refer to the OpenROAD documentation if you want to know what exactly each one does.

- Explore the generated clock tree using the widget, does it look like the image above?
- Our design will run at a few tens to $100\,\mathrm{MHz}$, does the clock skew look reasonable in comparison?

Sample solution: The clock skew should be roughly $1\,\mathrm{ns}$ or smaller. Assuming we encounter the worst case, the launching flip-flop would start the data at $1\,\mathrm{ns}$ and the receiving flip-flop would sample it exactly at the one period mark (eg $10\,\mathrm{ns}$) so we still have most of the time available for the data to propagate, the clock skew is within expectation. (note: clock skew can also be advantageous if the capturing flip flop is delayed compared to the launching one, it is not always bad.

- Use the reporting command to get the exact number (we care about clk_sys)

```
OpenROAD > report_clock_skew
```

- Especially if you have a bigger chip, the root buffer of the trunk might be far away from the clock input, this can be fixed using:

```
OpenROAD > repair_clock_nets
```

- Finally, its always a good idea to make sure the placement is legal

```
OpenROAD > detailed_placement
```

At this point we are done with the **scripts/03_cts.tcl** script and can move on to the routing. As a last step run the entire placement flow a final time to make sure we have an up-to-date checkpoint:

```
./run_backend.sh --cts
```

Alternatively if something does not work, you can go into the **save/** directory and rename **03_croc.cts.zip.sol** to **03_croc.cts.zip** to use the provided solution for the next steps.

4 Routing

Like placement, routing is split into a global and a detailed routing step.

4.1 Global Routing

During global routing, the entire area is divided into global routing cells (GCells). Each cell keeps track of how many wires go through it and how many tracks are available on the metal layers. Since we have vertical and horizontal layers, we have a number of vertical and horizontal tracks. Wires are now planned in this strict horizontal/vertical fashion and are added to all cells they go through. If a GCell has more wires than tracks in any direction, then we need to re-plan some of the wires and try to go through different GCells instead.

From observation, OpenROAD has a strong preference for lower routing layers (M2, M3) which makes sense since less vias should also mean less wasted space and less parasitics. However, especially during the end stages of detailed routing, it will mostly have to fix problems on the lower metals since all signals need to go through there to connect to cells. This makes it a good idea to 'save' some space on the lowest metals for later fixes.

Student Task 8:

- For therouting tasks, work in **scripts/04_routing.tcl** and start it with:

```
./run_backend.sh --open scripts/04_routing.tcl
```

- First we tell OpenROAD which layers it should route on

```
OpenROAD > set_routing_layers -signal Metal2-TopMetal1 -clock
             Metal2-TopMetal1
```

Remember that all wires on TM1 and TM2 are a lot larger so they are poor routing layers. Still, we need TM1 as a horizontal layer above the macros.

- Reduce the available routing resources on the lowest layers using

```
OpenROAD > set_global_routing_layer_adjustment <StartLayer>-<EndLayer>
             <reduction>
```

  Where `<reduction>` is a number below 1.0 determining the fraction of blocked-off tracks and Start/End-Layer are the names of the metals, the reduction counts on them and on all layers between.

- Reduce it also on TM1, since we don't really want routing there if possible

```
OpenROAD > set_global_routing_layer_adjustment TopMetal1 0.30
```

- Run global routing

```
OpenROAD > global_route
```

4.2 Timing Repairs

We are now able to use the global routing to estimate wire parasitics, we should use it to make sure we still meet our timings.

Student Task 9:

- First run

```
OpenROAD > estimate_parasitics -global_routing
```

- Since the timing repairs will add cells, we need to re-adjust our global routing to match, to do this we enter the incremental mode

```
OpenROAD > global_route -start_incremental
```

- Now we can actually repair the design and our timings, in this case setup and also hold

```
OpenROAD > repair_design -verbose
OpenROAD > repair_timing -skip_pin_swap -setup -verbose -repair_tns 100
OpenROAD > repair_timing -skip_pin_swap -hold -hold_margin 0.05
             -verbose -repair_tns 100
OpenROAD > detailed_placement
```

- Finally, we need to end the incremental mode and in doing so, make the necessary adjustments to our routing plan

```
OpenROAD > global_route -end_incremental -verbose
```

- We want to repair antennas (very long wires, which may cause problems during manufacturing)

```
OpenROAD > repair_antennas
```

- And as a last step, we add filler cells (they make sure all standard cell rows are continuous and do not have any gaps), you can also do this after detailed routing

```
filler_placement $stdfill
```

4.3 Detailed Routing

With global routing complete, we are in the last phase of the backend. The 'only' thing left is to implement the routing plan with geometry and fix and shorts and so on. This is no light task and usually requires the most amount of time out of all steps.

Because this takes a long time, start it only before the break and then let the computer work for you while you enjoy some coffee. At the end you should get something very similar to the design you saw in the very first exercise.

Student Task 10:

- It is vital that we use multiple threads to finish in a reasonable amount of time

```
OpenROAD > set_thread_count 16
```

- Now we can start detailed routing, this takes $30\,\mathrm{min}$ or more. If you have enough time you can run it using

```
OpenROAD > detailed_route -output_drc route_drc.rpt \
OpenROAD >    -droute_end_iter 15 \
OpenROAD >    -drc_report_iter_step 5 \
OpenROAD >    -save_guide_updates \
OpenROAD >    -clean_patches \
OpenROAD >    -verbose 1
```

  The DRC reports (which you can load using the DRC viewer widget) are really handy to debug any potential problem spots.

5 Next steps

Congratulations, we have finished the placement and routing stage. The last part of the design will be DRC and LVS checks which will be done using Calibre, a commercial tool. Once our design passes those checks, it can be manufactured.

We strongly encourage you to go through the placement and routing flow once as early as possible for your design. This will allow you to

- Prepare the basic setup that you can reuse for the final chip
- Get some experience and a feel of the design. Some designs offer more challenges than others, this will allow you to judge how much time you will need to factor in for the final back-end flow.
- See if the constraints for area and timing are realistic for your design.
- Obtain a usable power estimation for your circuit. Power estimations are wildly inaccurate if parasitics are not properly accounted for.