Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

EFCL Winter School 2026

Sample Solution Exercise 04

Floorplanning and Power Grid

F. K. Gürkaynak
Prof. L. Benini

Last Changed: 2026.02.09

1 Overview

In this exercise, we will start the back-end design process that will bring the chip design from the synthesized netlist all the way to the fabrication data ready to be sent to the factory. The software tool that we will mainly use for the back-end design is OpenROAD. We will learn:

- What type of input data and files are needed to run OpenROAD.
- How to add I/O drivers and power connections to our design.
- How to do floorplanning, including how to determine the chip area, place macro blocks, and make power connections.

This exercise covers the first part of the back-end design process, and subsequent exercises will cover further steps as we go along. The starting point of this exercise is a synthesized Verilog netlist which is the result of the synthesis exercise (for consistency, I recommend you start with the newly provided netlist). We will define the size of our chip and place I/O drivers and we will see how we can connect power and ground of the chip.

The rest of the back-end design flow will be covered in the exercise on Placement and Routing, where we will continue from where we stopped at the end of this exercise.

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[1].

```
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 ~/ex04
```

- 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] 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.

2 Technology Files

The technology files (excluding the standard cells) can be found in the IHPs Github repository[2] for the exercises we use the in-house managed files found in `technology/` directory. You will find the following files here:

- **Technology files**

  lef/*tech.lef Base technology description, defines metal layers, vias, spacing rules, routing.

- **Library files** (standard cells, pads, macro-cells)

  lef/*.lef Physical description, shape and allowed orientation of cells, layer and shape of pins, blockages, antenna information, etc.

  lib/*.lib Functional description, timing and power information, maximum load/fanout or transition-time allowed, etc.

  gds/*.gds The design converted to the internal language of laser writing equipments, in a format commonly used to transfer IC layout data.

  verilog/*.v Simulation models for all cells.

3 Setting up a Design

First you will see a few basic steps necessary to load a design into OpenROAD to then work with it.

Student Task 2:

- Open `openroad/scripts/01_floorplan.tcl` and `openroad/scripts/init_tech.tcl`
- Try to understand the necessary steps to get a synthesized design into OpenROAD, what files/data do we load, why?
  (from beginning of file until `check_setup`)

  Sample solution:

  We need to load the Liberty files for the functional description and timings, the base technology LEF for the metal layers, the standard cell, IO and macro LEFs for their physical description.
  For the liberty files we load two sets, one for each corner. We have a typical corner (tt) with the cells characterized around a typical process-voltage-temperature point and a fast corner (ff) with the cells characterized at their fastest point. The second one is necessary for hold-analysis, hold being the timing how long a flip-flop input must remain stable after the clock-edge.
  Then we also need the netlist, the name of the top-level design and constraints which define the clock and so on.

- In `01_floorplan.tcl` it later on executes a command called `initialize_floorplan` with one set of coordinates for the die-area and one for the core-area, what is the difference between these two terms? Sketch the chip with these two areas, the syntax of the command is as follows:

```
OpenROAD >  initialize_floorplan -die_area "x0 y0 x1 y1" -core_area "x0
               y0 x1 y1"
```

Sample solution:

Die is the entire chip, core is where the standard cells go, in-between is the core power ring.

[2] https://github.com/IHP-GmbH/IHP-Open-PDK

Besides technology files and the main flow we also have these files:

- **`synth/out/croc_yosys.v`**: The netlist
- **`openroad/src/instances.tcl`**: A file containing the paths (names) of the SRAMs
- **`openroad/src/constraints.sdc`**: Simple timing constraints
- **`openroad/src/padring.tcl`**: Tells OpenROAD where each IO pad should go
- **`openroad/scripts/01_floorplan.tcl`**: First step of the main flow, loads the design and creates the die
- **`openroad/scripts/ex_floorplan.tcl`**: Places the padring and macros (you work in here)
- **`openroad/scripts/power_grid.tcl`**: Creates the power distribution network ((you work in here)

At this point lets make sure everything works.

Student Task 3:

- Run the following command to check that the current setup is ready, it should open OpenROAD with a completely empty chip with a size of $1.916\,\mathrm{mm} \times 1.916\,\mathrm{mm}$

```
sh >  ./run_backend.sh --open scripts/01_floorplan.tcl
```

4 IOs/Pads

Microchips require specialized circuitry for electrostatic discharge (ESD) protection and strong buffers to drive large off-chip loads. I/O drivers do include all these circuits and also include bonding pads that enable physical connection between the microchip and the package. I/O drivers are basically specialized buffers, much larger than regular standard cells, and are designed to be placed around the core[3]. Open the file **`rtl/croc_chip.sv`**, the hierarchy level `croc_soc` is instantiated at the bottom of this file. `croc_soc` contains the core circuitry but not the I/O cells. The most practical way to add I/Os is to add another level of hierarchy which instantiates both the original design and the I/O cells, hence we have the `croc_chip` module. We call this hierarchy the *pad frame*. In addition to the core design and the I/O drivers, the pad frame will also contain pads for power and ground supply for both the core and the I/O drivers, as well as filler cells and corner cells to connect the edges of the pad frame. There are many different ways in which we can obtain a netlist describing the pad frame. It is not necessary to write this pad frame before synthesis, you can just as well go through synthesis with `croc_soc` and then write a structural Verilog module by hand that instantiated all power, ground and IO cells (and also tie-cells for constant low and high).

The necessary information about the IO cells (physical size, pin names, driving strength and so on) are found in the PDK. In this exercise, we will explain one particular way that we think is practical.

Student Task 4: Explore the **`technology/lib`** and **`technology/lef`** directories.

Pick one IO cell instantiated in `croc_chip` and try to find the following information:

- The physical size of the cell
- The name of its ports (pins)
- What the function if this cell is

Sample solution:

The physical information is in the lef, the names of the pins and the function can be found either in the Verilog models or in the Liberty files.

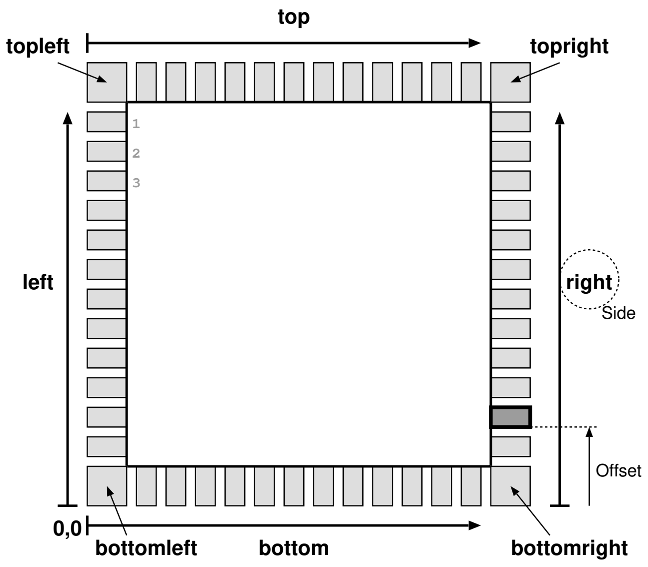
Now also try to find the corner cells, what are they called and how big are they?

[3] An internal buffer may drive loads in the order of a few fF, whereas external loads can easily be in the pF range and above.

Sample solution: The cell is called `corner` and its size is $180\,\mu m \times 180\,\mu m$

4.1 Placing the IOs

So we now have a netlist with all necessary pads (power, ground and IO). However, we also need to provide OpenROAD with the information about where on the die each pad should be placed. The pad placement is very important as it directly determines the PCB layout[4]. The pad frame we will use here consists of 48 I/O cells, four core supply cells, four core ground cells, four IO supply cells, four IO ground cells, as well as four corner cells. OpenROAD allows you to precisely specify the location of the IOs with a few commands, we will first place a few IOs by hand to get a feeling for the commands and then use the prepared script instead.



Student Task 5:

- Enter the following command to prepare for placing IO cells. Its the first command in **`openroad/src/padring.tcl`** if you just want to copy it.

```
OpenROAD >  make_io_sites -horizontal_site sg13g2_ioSite -vertical_site
                   sg13g2_ioSite -corner_site sg13g2_ioSite -offset 70
                   -rotation_horizontal R0 -rotation_vertical R0
                   -rotation_corner R0
```

- This command takes a 'site' (a placement unit from the LEF files Imagine a LEGO base plate, each stud/knob is one 'site' and then you place cells (bricks) on it) and uses it to build rows/columns of them. The default names are `IO_WEST`, `IO_SOUTH`, `IO_EAST` and `IO_NORTH`
- Select any IO cell from **`rtl/croc_chip.sv`** and place it anywhere in a row

```
OpenROAD >  place_pad -row <row> -location <offset> "pad_<name>"
```

- Once you are happy with the IOs you have placed, we need to fill in the gaps, for now only fill one row

```
OpenROAD >  place_io_fill -row <row> {*}$iofill
```

The `iofill` list is set in **`init-tech.tcl`** and contains a list of all IO-filler cells available

[4] A good pinout can simplify the routing on the PCB, allows you to use fewer layers and results in less parasitics.

- The last cells to place are the corners

```
OpenROAD >  place_corners $iocorner
```

- Now one side of your chip should be continuously filled with IO cells and fillers.
- In the OpenROAD gui activate 'Misc/Instances/Pins' and 'Misc/Instances/Pin Names' and investigate, what are you looking at?

Sample solution:

The IO cells and fillers have large pins going over them on most metal layers. These are internal power rings that get connected by abutment (one instance touching the other).

- Placing every pin is tedious so once you are happy with your progress we can call the next script to place the padring for us

```
source scripts/ex_floorplan.tcl
```

At the very start of **`scripts/ex_floorplan.tcl`** it un-places all placed instances. This makes it so you can source the **`scripts/ex_floorplan.tcl`** script over and over to quickly experiment with new floorplans.

- In **`scripts/01_floorplan.tcl`** Add back the line '`source scripts/ex_floorplan.tcl`' towards the end of the script. From here on out when you start OpenROAD it should run the padring placement and your next steps directly.

Note 2: If you encounter a problem with this repeated reloading of **`scripts/ex_floorplan.tcl`** it is a good idea to quit openroad and then start it again using

5 Floorplanning

Now we will have to decide how cells and macro-cells will be placed on our chip. This process is called floorplanning. For a standard design, our main concern would be to find a floorplan that will result in the smallest possible area, while fulfilling all performance and reliability requirements. This is purely driven by economical reasons, since chip costs are mainly determined by the area. In some cases there are additional geometrical constraints. The manufacturing company may impose certain limits to the aspect ratio of the final layout,[5] or even dictate the maximum height or width of the layout.

Back-end design is not only used for complete chips. Macro-cells that will be part of a larger system-on-chip design can also be designed in this way. In such cases there might be even more restrictions. For example, certain metal layers might be reserved for the system level. Take the SRAM macros, they only go from Metal1 to Metal4 so we are able to route over them on Metal5, TM1 and TM2.

So the question is, "*How small can my layout be so that I am still able to fulfill all specifications?*". As a lower bound, you will need enough area to place all your I/O pads and standard cells. Ideally, in terms of area you will want to place standard cells without leaving extra space in between, completely filling out the core area. This is hardly ever possible because:

- The number of interconnections that can pass through a certain area is limited by the number of metal layers available,[6] wire width and minimum spacing requirements. Depending on the interconnection overhead, the area above the cells may not be sufficient for routing.[7]

[5] Especially in MPW runs, a lot of silicon area is wasted if all designs have wildly different dimensions.
[6] For our technology there are 5+2 metal layers.
[7] Cells in our technology use primarily the lowest metal layer M1 for internal connections but the flip-flops use M2 as well, all other layers are completely free for routing.

- Timing is greatly affected by the placement of your cells. Placing them next to each other with no space in between does not leave the tool any flexibility in placing cells. This in turn reduces the optimization options of the tool, like the ability to cluster cells that are closely interconnected.
- All designs require power routing for operation. Some wires of the power connection limit where the cells can be placed, or restrict signal routing which in turn increases the area requirement.
- The majority of designs require a clock tree to function. This clock tree is added during the back-end design. This requires additional area for the buffers used in the clock tree. Furthermore, the clock tree synthesis algorithm can produce better results if it has more freedom to place its buffers.
- Macro-cells, like the RAM in our example, usually require some extra space along the edges so that they can be properly integrated next to the standard cells.
- Designs that have a high switching activity require a lot of current for a short time which is called a surge. The power distribution network may need additional decoupling capacitors to store some charge that can provide some of the current of the standard cells during such a surge. Additional space for these *decoupling* cells may be required during placement.

As a consequence, the standard cell rows can not be filled completely with standard cells. There needs to remain some free space between the cells. Utilization indicates to what percentage the standard cell rows are filled.[8] Usually, it is not possible to predict whether all requirements can be fulfilled with a certain utilization. Both placement and routing are NP complete problems. Without going through a complete routing and placement, it's not possible to know. You will have to try and find out, which is the main reason why back-end design is an iterative process.[9]

5.1 Sketching a Floorplan

Before we go on with OpenROAD we need to make some planning and understand some key concepts. The figure on the following page is an example floorplan (not an ideal one) that shows the important concepts.

In OpenROAD **die area** corresponds to the total silicon area available to place pads and core cells. All pads (I/O, power and corner) are placed in the **padframe**. Inside the padframe you can have extra space for a core power ring that is part of delivering the power to the cells in the core. The remaining area can be used for the **core** of the chip.

In the default configuration, the die area is $1916\,\mu\text{m} \times 1916\,\mu\text{m}$, whereas the maximum core area is $1416\,\mu\text{m} \times 1416\,\mu\text{m} = 2\,\text{mm}^2$ (assuming no extra space for the core power ring).

In the figure, we notice that the core area is surrounded by a **core power ring**. In its simplest form this consists of two wide metal lines (one for `VDD`, one for `VSS`) that evenly distribute the power all around the chip.[10] In order to leave room for the power ring, we need to leave a certain **I/O to core spacing**.

The **standard cells** are designed in such a way that, when placed next to each other their `VDD` and `VSS` pins can be connected with one horizontal metal line each. These power connections are relatively narrow and run on Metal1 over the entire width of the core area to connect to the core power rings. This may create problems for designs that consume much current, since the cells towards the middle would not have a good power connection.[11] To improve this, the two uppermost metal layers (TM1 and TM2) are used to place a **power grid** over the core area. The vertical and horizontal **power stripes** of that grid connect down through all metal layers to the narrow standard cell power connections.

The core area is filled with **standard cell rows** on which all standard cells will be placed. In the same area, we will also need to reserve some room for our macro-cells. Most macro-cells need some *free space* around

[8] At 100% utilization all cells are abutted and there is no extra space, whereas an utilization of 50% means that half of the core area is empty.

[9] Obviously, technology plays an important role, and it is possible to give certain guidelines for a technology. However, back-end design is always highly dependent on the design itself. You will usually see in a few iterations what is possible and what is not.

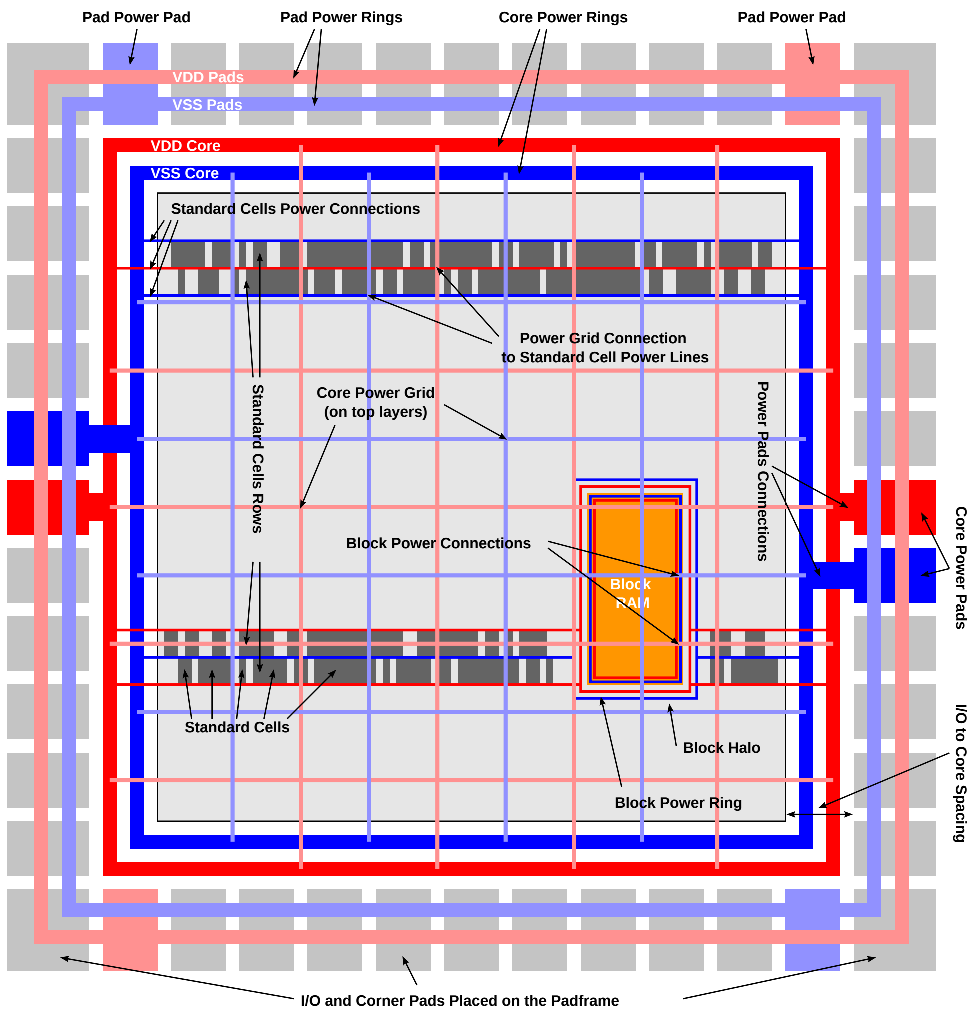
[10] The width of the metal line depends on the amount of current drawn from the line as well as the sheet resistance of the metal layer.

[11] The problem is that if much current is drawn, there will be a significant voltage drop along the power lines. The cells in the middle will be supplied with a lower VDD than the ones on the sides. This could dramatically affect the performance of the system.

to make signal connections, to add a **block power ring**, or simply to prevent standard cells from being placed too close to the macro.

When placing a macro-cell, you should also take into account where the power and signal pins of the block are located and what metal layer they are on. Often signal connections are only on two edges and you want them to face the core and not the I/O pads (or even worse, directly into another macro).

Now, when we consider all the above, the core area that remains for core cell placement is much smaller than the $2.50\,\mathrm{mm}^2$ that we started with. However, our example design has a total cell area (including SRAMs) of $0.5\,\mathrm{a}$nd should therefore comfortably fit into the available area. This is an exercise, so if you want to try something more ambitious or easier by make the chip larger or smaller, you are welcome to do so.



5.2 Placing Macros

Next, we place the SRAM macro-cell. Remember that the RAM macro will completely block the lower four metal layers (`M1` - `M4`). The remaining metal layers will be available for routing over the RAM macro-cell but TM1 and TM2 are not really meant to have a lot of signal routing on them, so when you place macros close together you can easily create congested areas.

Student Task 6:
Work in the **`ex_floorplan.tcl`** file in the designated area. Once placed you can also edit some placement information in OpenROADs gui instead.
To place a macro you have two options, either use the OpenROAD built in `place_macro` or a helper function from **`floorplan_util.tcl`** called `placeInstance` (it supports all orientations instead of a subset). Below is the syntax for both:

```
place_macro
-macro_name macro_name
-location "x y"
[-orientation orientation]

placeInstance NAME X Y orientation
```

Where the orientation for `place_macro` is one of `R0, MY, MX, R180` and `placeInstance` additionally allows `R90, R270, MYR90, MXR90` as defined in the source code[a]

Place one macro and look at where the pins are facing, then plan the overall picture you want and try to place them accordingly.

To make sure the standard cells don't get too close to the macros, you can use the following command after placing the macros

```
cut_rows -halo_width_x <X> -halo_width_y <Y>
```

It cuts out the standard cell placement sites around each macro.

[a] https://github.com/The-OpenROAD-Project/OpenROAD/blob/master/src/odb/include/odb/dbTypes.h#L54

If you want to see if your floorplan is any good you can quit openroad and run `./run_backend.sh --open scripts/01_floorplan.tcl` to re-generate the floorplan immediately.

OpenROAD also has two different commands to automatically place macros. I recommend you continue with the exercise and then return to this at a later point.

Student Task 7:
Try to create a floorplan using either OpenROADs automatic macro placer[a] or the hierarchical macro placer[b].

[a] https://openroad.readthedocs.io/en/latest/main/src/mpl/README.html
[b] https://openroad.readthedocs.io/en/latest/main/src/mpl2/README.html

6 Power Planning

The next step is to create the power distribution network.

The Verilog netlist that we started with does not contain any power connections, therefore we need to create this connectivity now. We have to connect the power/ground pins of all instances to the respective global power/ground net that was specified on the DESIGN IMPORT form (category POWER)

The example design has only one power domain where all the core instances are connected to the same power and ground nets. However, it is possible to have different power domains in a chip (e.g., if certain parts of an integrated system need to be completely shut off dynamically to save power). OpenROAD can handle different power domains and the signal crossing from one domain to another, if they are properly specified in what is known as the *Power Intent*. The power intent is written in the *Unified Power Format* (UPF).

Student Task 8:
Open **`openroad/scripts/power_connect.tcl`** and try to get an idea of what the individual commands do.
`add_global_connection`

Sample solution: create a rule for a connection between pins (from a selection of instances) to a net

`global_connect`

Sample solution: actually connect according to above rules

`set_voltage_domain`

Sample solution: define a voltage/power domain and give it a name

`define_pdn_grid`

Sample solution: Assign the domain to a grid we are about to create

After you call the **`openroad/scripts/power_connect.tcl`** script once, the script **`power_grid.tcl`** is similar to **`ex_floorplan.tcl`** in that you can execute it multiple times to iterate quickly. This is a built in functionality of OpenROAD. The command `pdngen` is like a manager for the entire power routing system. You tell it what you would like, then you call it and it tries to build it.

Don't touch the commands in the 'Reset' and 'Generate' sections at the start and end of the script.

- First make sure the power connections are proper. Execute the following commands and for future iterations, add them back in the **`scripts/01_floorplan.tcl`** script towards the end.

```
source scripts/power_connect.tcl
global_connect
```

- Also add back the `source scripts/power_grid.tcl` and all following commands in **`scripts/01_floorplan.tcl`** until the end of the file. Now whenever you re-open OpenROAD using the **`scripts/01_floorplan.tcl`** script it will go through the entire floorplanning flow.
- We start with the physical power network, you can directly enter the following commands in OpenROAD and then later migrate them to **`scripts/power_grid.tcl`** once you want to iterate. Or you can directly start adding them in the script, either works.
- The first thing we want to create is a core power ring, everything else will connect to it so its central. To do so we use the `add_pdn_ring` command, a good set of options are the following:

  ```
  add_pdn_ring -grid {core_grid} \
  -layer        {TopMetal1 TopMetal2} \
  -widths       "7 7" \
  -spacings     "5 5" \
  -core_offsets "3 3" \
  -add_connect        \
  -connect_to_pads
  ```

  The first options adds it to the core grid (this is also default but its a good habit to be specific). Then we tell it on which layers the ring should be, one for horizontal and one for vertical. The next options define different dimension, you can play with them until it feels right to you. Then we tell it to automatically add a connection (vias) between TM1 and TM2 in the core_grid and finally we tell it to connect to the power pads. Note that you will see the effects of this only after the actual PDN generation (`pdngen`).
- Play with the dimensions of the core power ring until you are happy with it.
- Next we add the standard cell power rails on M1, this is where the power should go in the end.
  To do so we use the `add_pdn_stripe` command, which just adds stripes, but we tell it to follow the standard cell power pins using `-followpins`

  ```
  add_pdn_stripe -grid {core_grid} -layer {Metal1} -followpins
      -extend_to_core_ring
  ```

  The `extend_to_core_ring` tells pdngen to not stop at the boundary of the core area but rather extend the stripes until it intersects with the core power ring.
- Zoom in and lock at the stripes, how do they repeat and why does this make sense? How do they connect to standard cells?

  Sample solution:

  It alternates VDD and VSS. This is possible because every second row of standard cells is mirrored vertically.

- The next thing we want is for the stripes to terminate somewhere so they aren't just dangling. Specifically they should connect to the power ring (currently the via stacks are missing). We tell pdngen to connect different layers via:

  ```
  OpenROAD >  add_pdn_connect -grid {core_grid} -layers {TopMetal2 Metal1}
  ```

  Using these three commands you should now have a power grid that delivers electrons from the pads through the core power ring, then a via stack and finally into the Metal1 power rails.
- As you might imagine, currently the resistance (and with that the voltage drop) is dominated by the relatively thin but long Metal1 stripes.
  So lets support the power rails by regularly injecting power from a path with less resistance. In particular we want a grid of vertical stripes over the entire chip.

- Using the add_pdn_stripe command without the -followpins option, add a grid of vertical stripes that connects to the core power ring and the M1 power rails. Here is the syntax you need to use:

```
add_pdn_stripe -grid {core_grid} -layer <layer> -width <width> \
-pitch <pitch> -spacing <spacing> -offset <offset> \
-extend_to_core_ring
```

  Pitch is the distance from one pair of VDD and VSS to the next, spacing is the distance between VDD and VSS of the same pair. Offset defines where it starts and width is the width of each stripe. If you chose another layer than TM2, remember to also add the via connections

6.1 Macro Power

The one thing missing now is the power to the macros and a power ring around each macro so the Metal1 power rails terminate and aren't left dangling.

You can look at macros as their own small 'core' that needs a power ring and some stripes over it. In fact this is exactly what we are going to do.

Student Task 9:

- We want a separate grid for all macros of the same type and compatible orientation, look at the SRAMs power pins and think about the stripes we will put over them, which orientations are not compatible?
- Define the power grid, see the openroad docs[a]

```
define_pdn_grid -macro -cells RM_IHPSG13_1P_256x64_c2_bm_bist -name
    sram_256x64_grid -orient "R0 R180..." \
-grid_over_boundary -voltage_domains {CORE} \
-halo {1 1}
```

- Now again lets add a power ring to this grid using:

```
add_pdn_ring -grid <grid> \
-layer         <layers> \
-widths        "<widthX widthY>" \
-spacings      "<spaceX spaceY>" \
-core_offsets "<offstX offstY>" \
-add_connect
```

- Then add stripes over the macro
- Finally make sure you tell pdngen to connect the correct layers in this grid as well.

[a] https://openroad.readthedocs.io/en/latest/main/src/pdn/README.html#define-power-grids

7 Last Words

At this point you should know how to place IOs and macros, what a good floorplan roughly is and you can add a simple power grid to a chip. If you have time left you can go back to exploring macro placement or you can try and see what happens to the routing congestion if you make TM2 stripes over the chip really really thick.