Newcomers

VLSI, or Very Large-Scale Integration, is a cornerstone technology driving the modern world, silently powering the devices we rely on daily. Its importance stems from its profound impact on various aspects of our lives, making it ubiquitous. VLSI facilitated the fabrication of chips. A chip is a small piece of semiconductor material (usually silicon) containing integrated circuits. These circuits consist of millions or even billions of tiny switches called transistors and other electronic components that process information and perform various functions. These chips are the brains behind countless electronic devices, from smartphones and computers to cars and medical equipment.

One class of chips is called ASIC. ASIC is a type of chip custom-designed for a specific purpose, unlike general-purpose chips like microprocessors that can be used for various tasks. One of the most prominent examples of ASIC is smartphone processors. These chips are custom-designed for mobile devices, focusing on low power consumption and high performance for running apps, gaming, and streaming media. They often incorporate specialized cores for graphics processing, artificial intelligence, and image signal processing.

Designing an ASIC is a complex and fascinating process that entails various steps, from idea to the fabrication data. This process is filled with engineering challenges that require expertise and attention to detail. The entire process requires significant expertise and experience in chip design and can take several months to complete. The ASIC design flow is crucial to ensure successful ASIC design. It is based on a comprehensive understanding of ASIC specifications, requirements, low-power design, and performance. Engineers can streamline the process and meet crucial time-to-market goals by following a proven ASIC design flow. Each stage of the ASIC design cycle is supported by powerful EDA

(Electronic Design Automation) tools that facilitate the implementation of the design. The following are examples of steps needed to realize an ASIC.

- Design Entry: In this step, the logic design is described using a Hardware Description Language (HDL) like System Verilog. Typically, the description is done at the data flow (Register Transfer) or behavioral levels.
- Functional Verification: It is essential to catch design errors early on. The description must be checked against the requirements, which can be done through simulation or formal methods. Functional verification is performed on the RTL description as well as the netlists generated by the following steps.
- Synthesis: In this step, the HDL description is converted into a circuit of the logic cells called the Netlist.
- Layout/Physical Synthesis: Also called Physical Implementation. In this step, the logic circuit is converted into a layout of the photo masks used for fabrication. This complex step involves several sub-steps typically automated using its flow. These steps include Floorplanning, Placement, Clock-tree synthesis and Routing. Because Placement and Routing are the most time-consuming operations, sometimes we refer to this step as “Placement and Routing”, or PnR.
- Signoff: marks the final stage in the rigorous journey of an ASIC’s design; it ensures your creation functions flawlessly, operates efficiently, and ultimately delivers on its promise before sending your chip blueprint off to be carved in silicon.

Please note that the five mentioned steps are the major ones. There are several other design steps that are not mentioned here such as scan chain insertion and test pattern generations that are essential to testing the fabricated chip against fabrication defects.

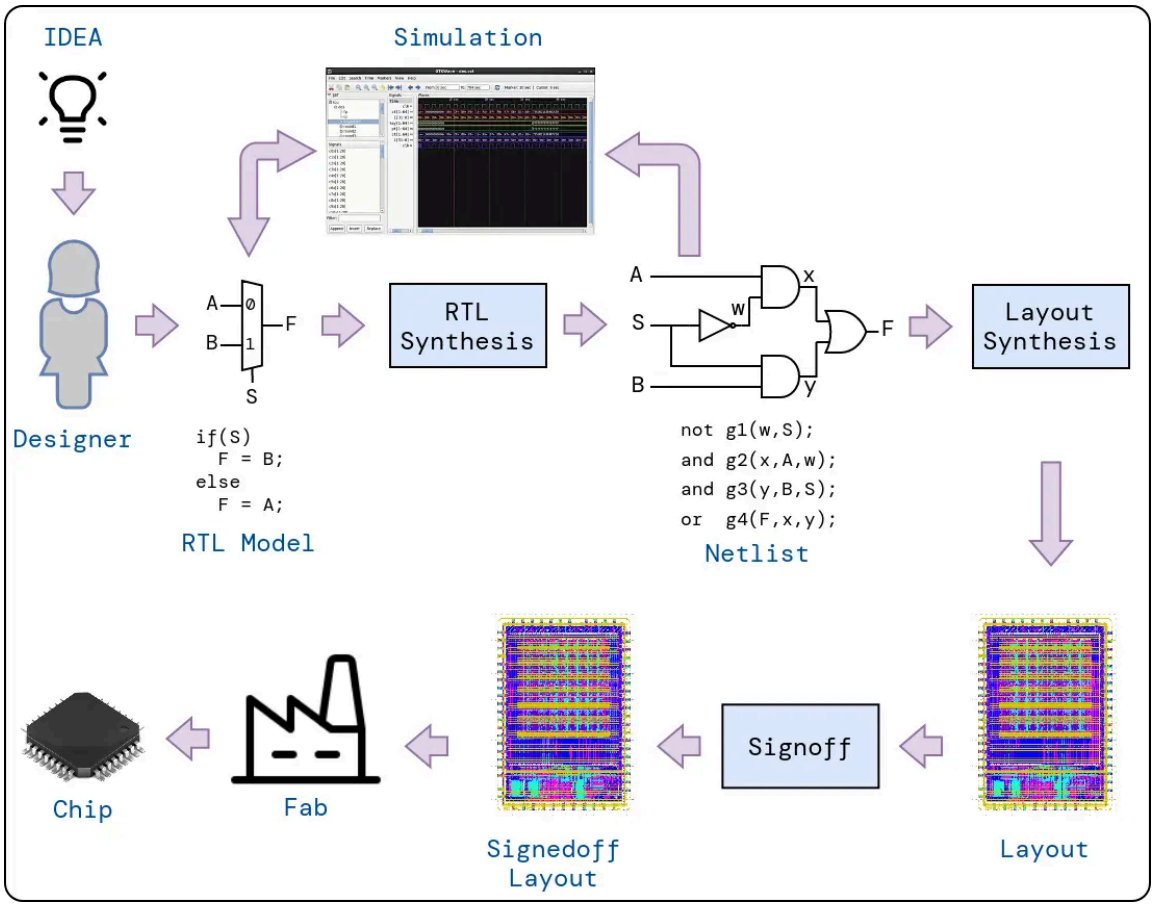

Fig. 2 ASIC Flow

What is OpenLane?

OpenLane is a powerful and versatile infrastructure library that enables the construction of digital ASIC physical implementation flows based on open-source and commercial EDA tools. It includes a reference flow (`Classic`) that is constructed entirely using open-source EDA tools –abstracting their behavior and allowing the user to configure them using a single file (See Figure 1).

OpenLane also supports extending or modifying flows using Python scripts and utilities. Here are some of the key benefits of using OpenLane:

- Flexibility and extensibility: OpenLane is designed to be flexible and extensible, allowing designers to customize the flow to meet their specific needs by developing Python scripts (plugins) and utilities or by modifying the existing configuration file.
- Open source: OpenLane is an open-source project that is freely available to use and modify, which makes it a good choice for designers looking for a transparent, cost-effective solution.
- Community support: OpenLane capitalizes on OpenLane's existing community of users and contributors, which means that a wealth of resources is available to help designers get started and troubleshoot any problems they encounter.

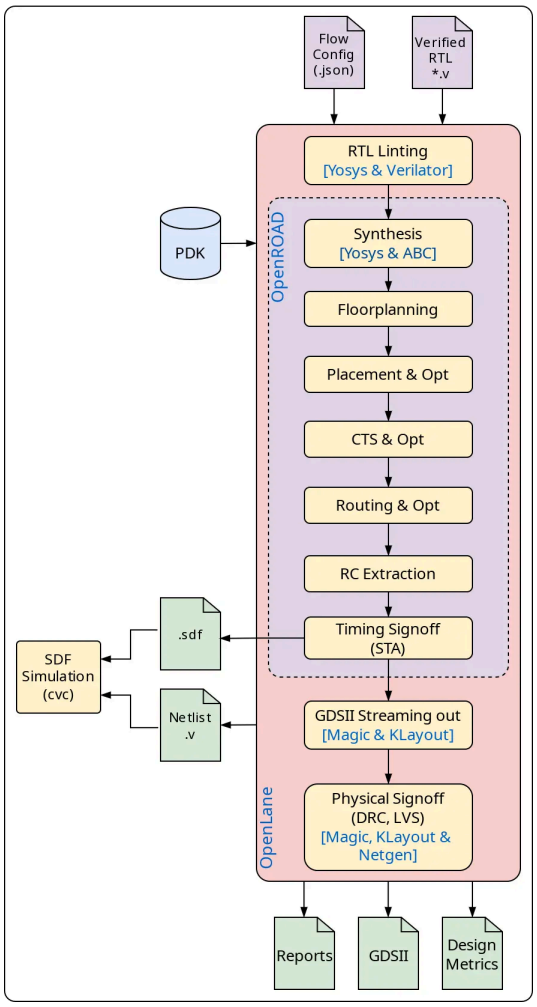

Fig. 3 OpenLane Flow

See also

You may want to check out OpenLane using Google Colab directly in your browser.

It's free, and there's no need to install anything on your machine!

Note

This guide assumes that the reader has some basic knowledge of Digital Design, ASIC, the JSON format and RTL.

Installation

1. Follow the instructions in Nix-based Installation to install Nix and set up Cachix.
2. Open a terminal and clone OpenLane as follows:

```
$ git clone https://github.com/efabless/openlane2/ ~/openlane2
```

3. Invoke `nix-shell`, which will make all the packages bundled with OpenLane available to your shell.

```
$ nix-shell --pure ~/openlane2/shell.nix
```

Some packages will be downloaded (about 3GiB) and afterwards, the terminal prompt should change to:

```
[nix-shell:~/openlane2]$
```

> **Important**
>
> From now on; all commands assume that you are inside the `nix-shell`.

4. Run the smoke test to ensure everything is fine. This also downloads sky130 PDK.

```
[nix-shell:~/openlane2]$ openlane --log-level ERROR --condensed --show-progr
```

That's it. Everything is ready. Now, let's try OpenLane.

Running the default flow

PM32 example

We are going to use a simple design: a 32-bit parallel multiplier which performs a simple multiplication between MP and MC and outputs the product on a bus P. MP32 uses a serial-by-parallel multiplier, a serializer, a deserializer, and a control unit. The block diagram of the PM32 is shown in the next figure.

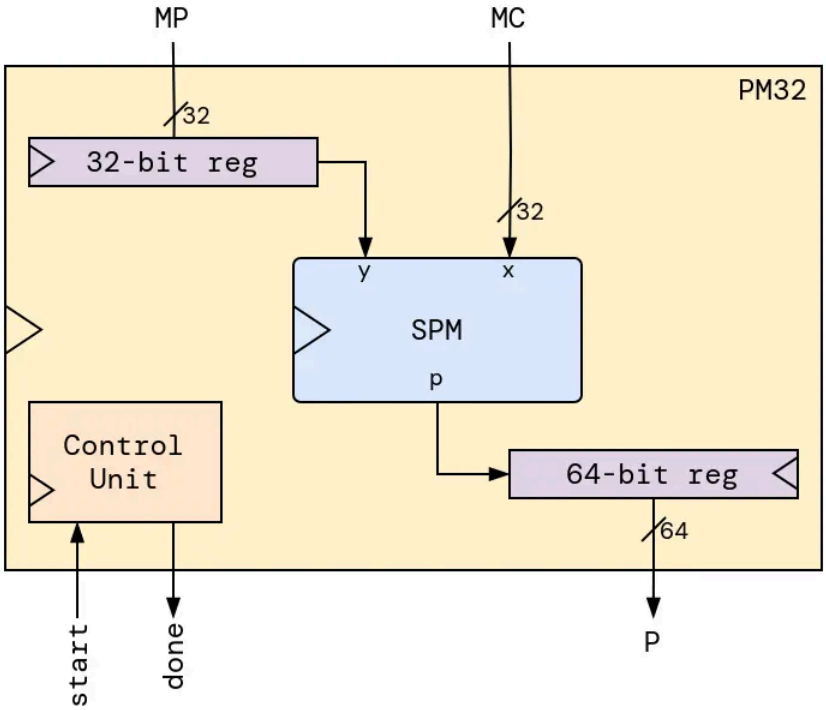

Fig. 4 PM32 (32-bit parallel multiplier)

The serial-by-parallel signed 32-bit multiplier `SPM` block inside the `PM32` performs a simple shift-add algorithm, where the parallel input `x` is multiplied by each bit of the serial input `y` as it is shifted in. The product is generated and serially output on the wire `p`. Check this paper for more information about the `SPM`.

RTL

The source RTL of the design will consist of 2 files, which are `spm.v` and `pm32.v`.

pm32.v

spm.v

Configuration

Designs in OpenLane have configuration files. A configuration file contains values set by the user for various `openlane.config.Variable` (s). With them, you control the flows. This is the configuration file for the `pm32` design:

config.json

Warning

For any design, at a minimum you need to specify the following variables:

- `DESIGN_NAME`
- `VERILOG_FILES`
- `CLOCK_PERIOD`
- `CLOCK_PORT`

See also

Check out the Classic flow's documentation for information about all available variables.

How to run?

1. Create a directory to add our source files to:

```
[nix-shell:~/openlane2]$ mkdir -p ~/my_designs/pm32
```

2. Create the file `~/my_designs/pm32/config.json` and add configuration content to it.
3. Create the files `~/my_designs/pm32/pm32.v`, `~/my_designs/pm32/spm.v`, and add RTL content to them.
4. Run the following command:

```
[nix-shell:~/openlane2]$ openlane ~/my_designs/pm32/config.json
```

Tip

Double-checking: are you inside a `nix-shell`? Your terminal prompt should look like this:

```
[nix-shell:~/openlane2]$
```

If not, enter the following command in your terminal:

```
$ nix-shell --pure ~/openlane2/shell.nix
```

Checking the results

Viewing the Layout

To open the final GDSII layout run this command:

```
[nix-shell:~/openlane2]$ openlane --last-run --flow openinklayout ~/my_designs
```

This opens KLayout and you should be able to see the following:

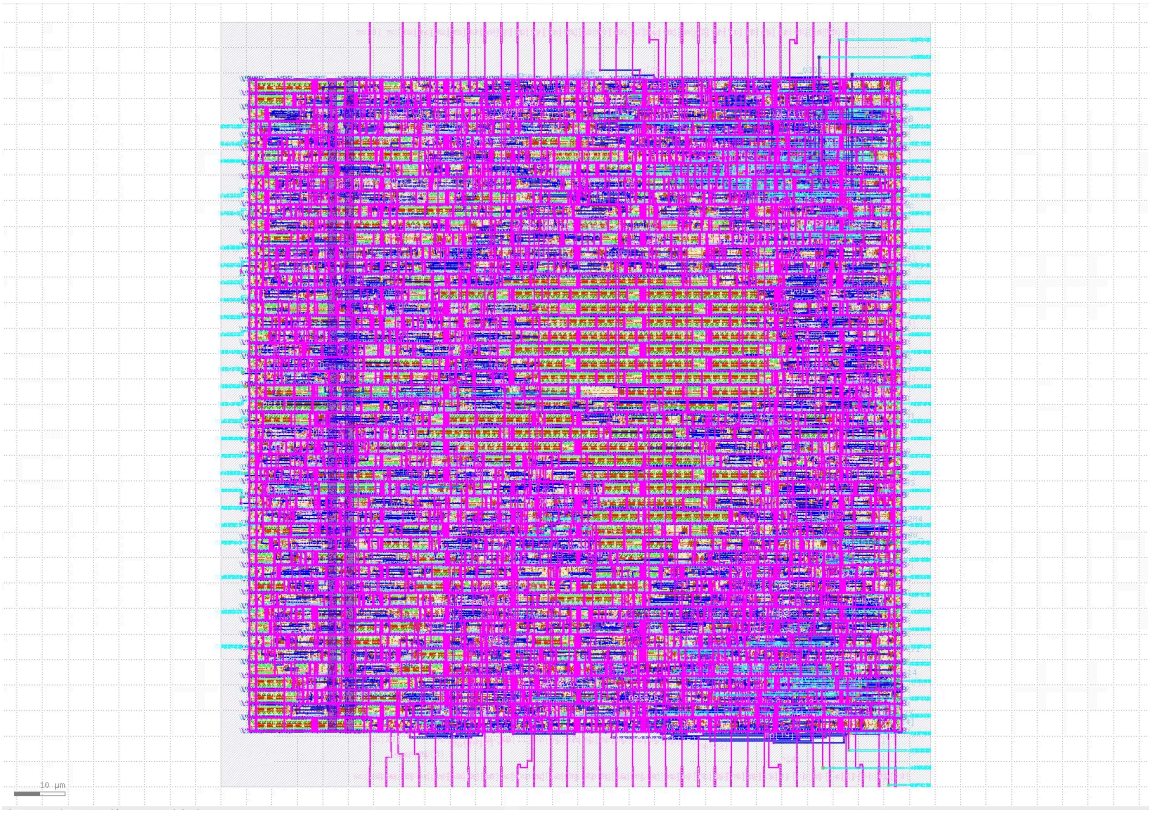

Fig. 5 Final layout of pm32

If you wish to view the layout in the OpenROAD GUI, try this command instead:

```
[nix-shell:~/openlane2]$ openlane --last-run --flow openinopenroad ~/my_design
```

Run directory

You'll find that a **run directory** (named something like `runs/RUN_2023-12-27_16-59-15`) was created when you ran OpenLane.

By default, OpenLane runs a `Flow` composed of a sequence of `Step`(s). Each step has its separate directory within the run directory.

For example, the `OpenROAD.TapEndCapInsertion` Step creates the following directory `14-openroad-tapendcapinsertion`.

A step directory has log files, report files, metrics and output artifacts created by the step.

For example, these are the contents of `14-openroad-tapendcapinsertion`:

```
14-openroad-tapendcapinsertion/
├── COMMANDS
├── config.json
├── openroad-tapendcapinsertion.log
├── openroad-tapendcapinsertion.process_stats.json
├── or_metrics_out.json
├── pm32.def
├── pm32.nl.v
├── pm32.odb
├── pm32.pnl.v
├── pm32.sdc
├── state_in.json
└── state_out.json
```

Here is a small description of each of those files:

Directory of contents

The run directory is composed of many of these step directories:

```
RUN_2023-12-27_16-59-15
├── 01-verilator-lint/
├── 02-checker-linttimingconstructs/
├── 03-checker-linterrors/
├── 04-yosys-jsonheader/
├── 05-yosys-synthesis/
├── 06-checker-yosysunmappedcells/
├── 07-checker-yosyssynthchecks/
├── 08-openroad-checksdcfiles/
├── 09-openroad-staprepnr/
├── 10-openroad-floorplan/
├── 11-odb-setpowerconnections/
├── 12-odb-manualmacroplacement/
├── 13-openroad-cutrows/
├── 14-openroad-tapendcapinsertion/
├── 15-openroad-globalplacementskipio/
⋮
├── final/
├── tmp
├── error.log
├── info.log
├── resolved.json
└── warning.log
```

Final Results

Inside the run directory, you may have noticed there is another, non-specific step folder named `final`.

`final` contains several directories that contain all the different layout views produced by the flow. It looks like this:

```
final
├── def/
├── gds/
├── json_h/
├── klayout_gds/
├── lef/
├── lib/
├── mag/
├── mag_gds/
├── nl/
├── odb/
├── pnl/
├── sdc/
├── sdf/
├── spef/
├── spice/
├── metrics.csv
└── metrics.json
```

Moreover, it contains `metrics.csv` and `metric.json` which represent the final metrics in `JSON` and CSV formats.

Signoff Steps

An ASIC design's signoff is the last phase of its implementation. It involves physical and timing verifications before committing to the silicon manufacturing process, which is commonly known as "design tape-out".

OpenLane runs a couple of Step(s) for the final signoff.

1. DRC
2. LVS
3. STA
4. Antenna Check

DRC

DRC stands for Design Rule Checking which checks, against rules set by chip foundries, that the layout has to satisfy in order to be manufacturable, such as checking for minimum allowed spacing between two `met1` shapes.

OpenLane runs two DRC steps using `Magic` and `KLayout`: `Magic.DRC` and `KLayout.DRC`. Both tools have blind spots that are covered by the other tools.

Both the layout and what is known as a PDK's DRC deck are processed by the tools running DRC, as shown in the diagram below:

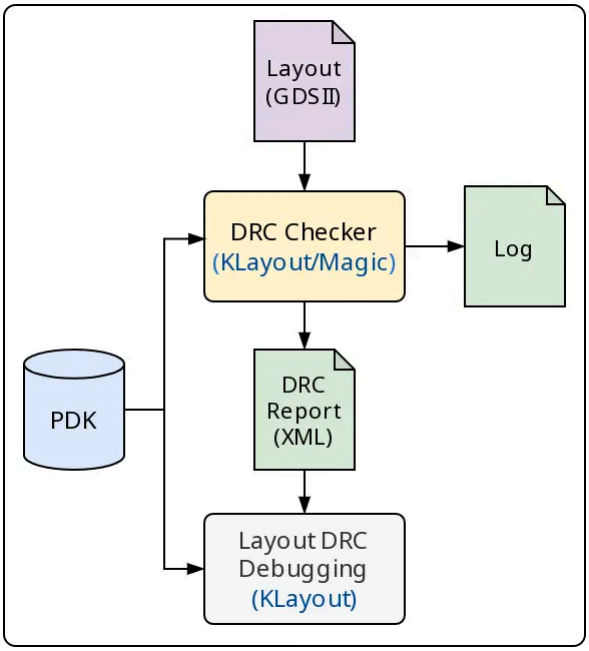

Fig. 6 DRC (Design Rule Checking) Flow

If DRC violations are found; OpenLane will generate an error reporting the total count of violations found by each Step.

To view DRC errors graphically, you may open the layout as follows:

```
[nix-shell:~/openlane2]$ openlane --last-run --flow openinklayout ~/my_designs
```

Then in the menu bar select Tools ► Marker Browser. A new window should open.

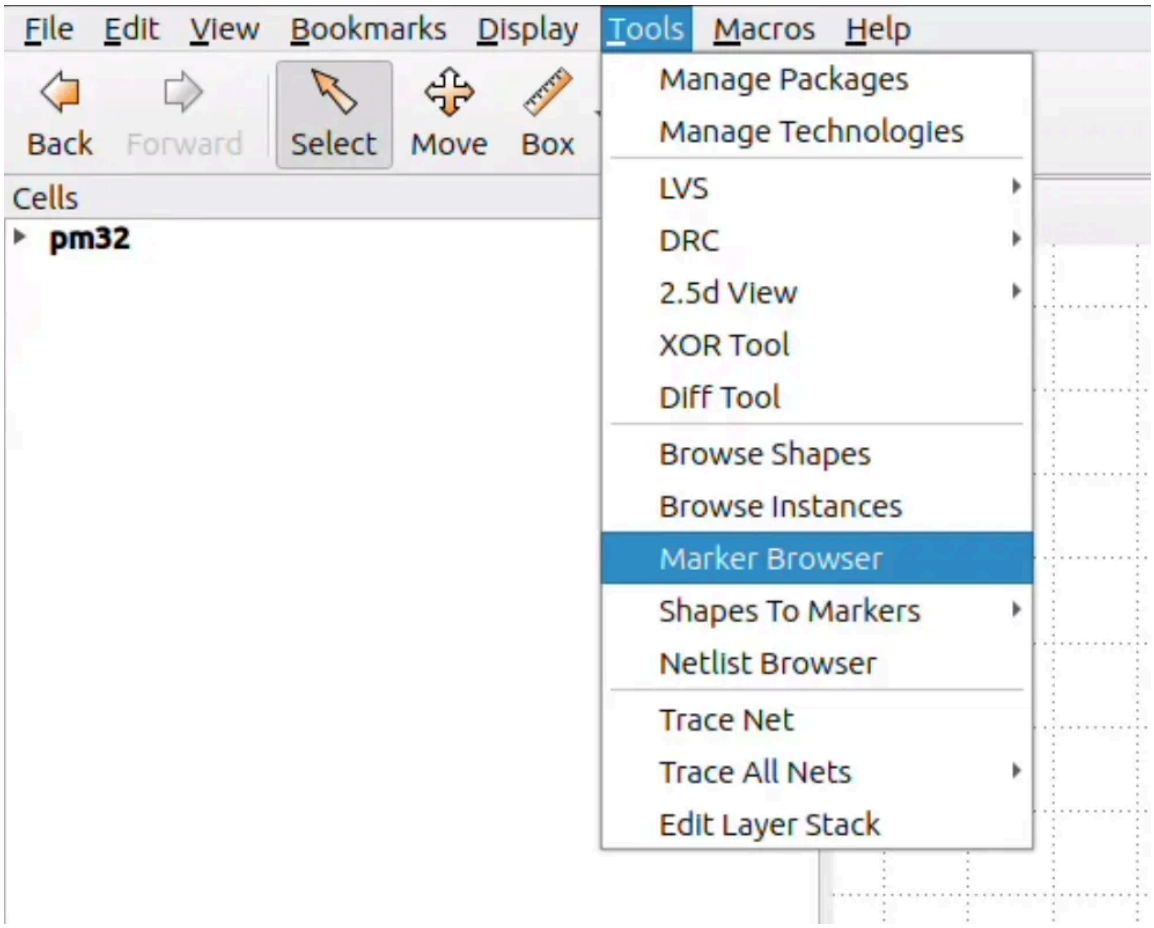

Fig. 7 Tools ► Marker Browser

Click File ► Open and then select the DRC report file, of which you'll find two: One under `52-magic-drc/reports/drc.klayout.xml` and the other under `53-klayout-drc/report/drc.klayout.xml`.

> **Tip**
>
> The initial number in `53-klayout-drc` (`53`) may vary according to the flow's configuration.

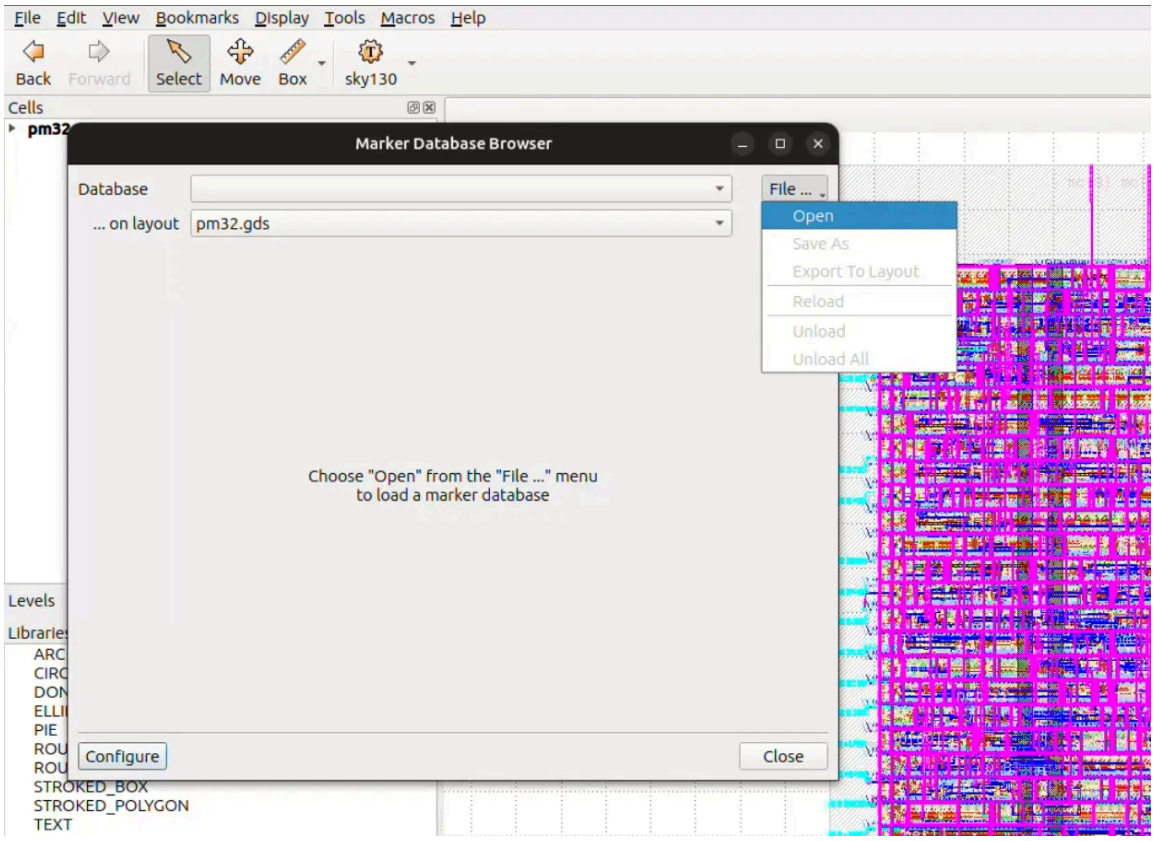

Fig. 8 Opening DRC xml file

LVS

LVS stands for Layout Versus Schematic. It compares the layout GDSII or DEF/LEF, with the schematic which is usually in Verilog, ensuring that connectivity in both views matches. Sometimes, user configuration or even the tools have errors and such a check is important to catch them.

Common LVS errors include but are not limited to:

- Shorts: Two or more wires that should not be connected have been and must be separated. The most problematic is power and ground shorts.
- Opens: Wires or components that should be connected are left dangling or only partially connected. These must be connected properly.
- Missing Components: An expected component has been left out of the layout.

`Netgen.LVS` is the Step run for LVS using a tool called Netgen. First, the layout is converted to SPICE netlist. Next, the layout and the schematic are inputted to Netgen, as shown in the diagram below:

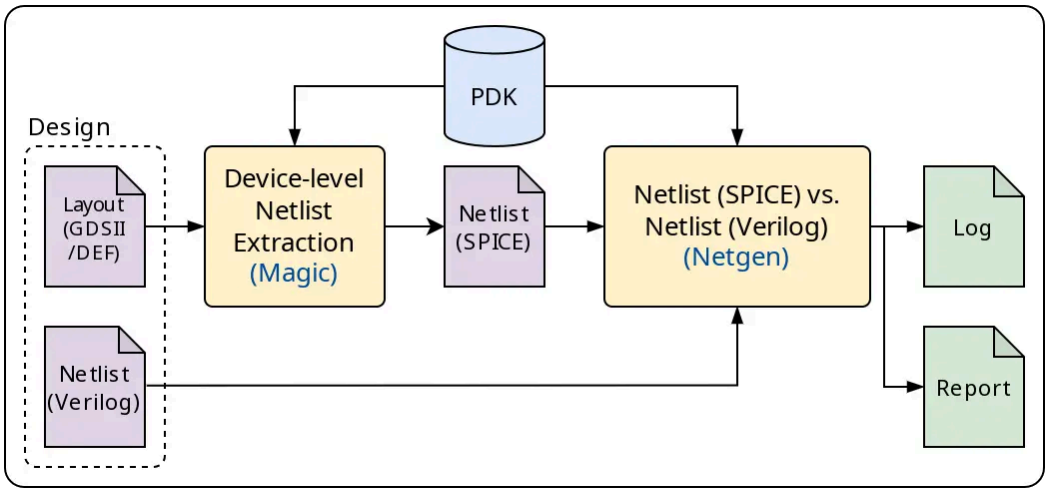

Fig. 9 LVS (Layout-versus-Schematic) Flow

`Netgen` will generate multiple files that can be browsed in case of LVS errors. As with all Step(s), these will be inside the Step's directory.

You would want to look at `netgen-lvs.log` which has a summary of the results of `LVS`. Ideally, you would find the following at the end of this log file:

```
Final result:
Circuits match uniquely.
```

In case of errors, there is also `lvs.rpt` which is more detailed. Inside it, you will find tables comparing nodes between the layout and the schematic. On the left is the layout (`GDS`) and the schematic (Verilog) is on the other side. Here is a sample of these tables:

```
Subcircuit summary:
Circuit 1: pm32                             |Circuit 2: pm32
--------------------------------------------|---------------------------------
sky130_fd_sc_hd__tapvpwrvgnd_1 (102->1)     |sky130_fd_sc_hd__tapvpwrvgnd_1 (10
sky130_fd_sc_hd__decap_3 (144->1)           |sky130_fd_sc_hd__decap_3 (144->1)
sky130_fd_sc_hd__inv_2 (64)                 |sky130_fd_sc_hd__inv_2 (64)
sky130_fd_sc_hd__nand2_1 (31)               |sky130_fd_sc_hd__nand2_1 (31)
sky130_fd_sc_hd__dfrtp_1 (64)               |sky130_fd_sc_hd__dfrtp_1 (64)
sky130_ef_sc_hd__decap_12 (132->1)          |sky130_ef_sc_hd__decap_12 (132->1)
```

STA

STA stands for Static Timing Analysis. The STA tool identifies the design timing paths and then calculates the data's earliest and latest actual and required arrival times at every timing path endpoint. If the data arrives after (in case of setup

checking) or before (hold checking) it is required, then we have a timing violation (negative slack). STA makes sure that a circuit will correctly perform its function (but tells nothing about the correctness of that function.)

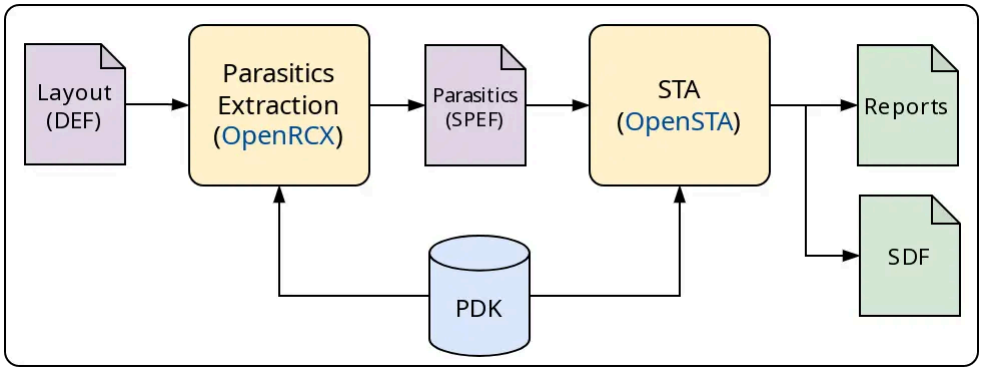

Fig. 10 STA (Static Timing Analysis) Flow

The default flow runs a step called `OpenROAD.STAPostPNR` for STA signoff.

Note

During the `Classic` flow the step `OpenROAD.STAMidPNR` is ran multiple times.

The results are not as accurate as `OpenROAD.STAPostPNR` since the design is not fully implemented yet. However, they provide a better sense of the impact multiple stages of the flow (such as optimizations steps) on STA results for the design.

Inside the Step directory, there is a file called `summary.rpt` which summarizes important metrics for each IPVT timing corner:

Corner/Group	Hold Worst Slack	Reg to Reg Paths	Hold TNS	Hold Vio
Overall	0.1097	0.1097	0.0000	0
nom_tt_025C_1v80	0.3314	0.3314	0.0000	0
nom_ss_100C_1v60	0.9203	0.9203	0.0000	0
nom_ff_n40C_1v95	0.1125	0.1125	0.0000	0
min_tt_025C_1v80	0.3275	0.3275	0.0000	0
min_ss_100C_1v60	0.9154	0.9154	0.0000	0
min_ff_n40C_1v95	0.1097	0.1097	0.0000	0
max_tt_025C_1v80	0.3356	0.3356	0.0000	0
max_ss_100C_1v60	0.9274	0.9274	0.0000	0
max_ff_n40C_1v95	0.1153	0.1153	0.0000	0

More about IPVT corners

IPVT corners stands for Interconnect, Process, Voltage, and Temperature corners. Interconnect corners take into account variations in interconnect width, thickness, and height, as well as the dielectric properties of the materials surrounding them. Interconnect variations affect the signal delay, power dissipation, and cross-talks. PVT corners represent the extreme conditions of manufacturing process variations, operating voltage fluctuations, and temperature ranges an IC might experience. These corners are used to simulate the worst-case scenarios to ensure that the IC functions correctly under all possible conditions. For instance, a slow process corner combined with low voltage, high temperature, and maximum interconnect delay could represent a worst-case scenario for timing, leading to slower transistor speeds hence more likely to have setup violations. On the other hand, a fast process corner combined with high voltage, low temperature, and min interconnects could represent the fastest scenario for timing, which is more likely to have hold violations. If we go back to the STA summary table, the worst setup slack is in the max_ss_100C_1v60 which has the maximum interconnect delay, slow device process, high temperature (100 Celsius), and low voltage (1.60V). Alternatively, the worst hold slack is in the min_ff_n40C_1v95 which has the minimum interconnect delay, fast device process, low temperature (negative 40 Celsius), and high voltage (1.95V).

There is also a directory per corner inside the Step directory which contains all the log files and reports generated for each `IPVT corner`.

```
45-openroad-stapostpnr/
└── nom_tt_025C_1v80/
    ├── checks.rpt
    ├── filter_unannotated.log
    ├── filter_unannotated.process_stats.json
    ├── filter_unannotated_metrics.json
    ├── max.rpt
    ├── min.rpt
    ├── power.rpt
    ├── skew.max.rpt
    ├── skew.min.rpt
    ├── pm32__nom_tt_025C_1v80.lib
    ├── pm32__nom_tt_025C_1v80.sdf
    ├── sta.log
    ├── sta.process_stats.json
    ├── tns.max.rpt
    ├── tns.min.rpt
    ├── violator_list.rpt
    ├── wns.max.rpt
    ├── wns.min.rpt
    ├── ws.max.rpt
    └── ws.min.rpt
```

Here is a small description of each file:

- `sta.log`: Full log file generated by STA which is divided into the following report files
- `min.rpt`: Constrained paths for hold checks.
- `max.rpt`: Constrained paths for setup checks.
- `skew.min.rpt`: Maximum clock skew for hold checks.
- `skew.max.rpt`: Maximum clock skew for setup checks.
- `tns.min.rpt`: Total negative hold slack.
- `tns.max.rpt`: Total negative setup slack.
- `wns.min.rpt`: Worst negative hold slack.
- `wns.max.rpt`: Worst negative setup slack.
- `ws.min.rpt`: Worst hold slack.
- `ws.max.rpt`: Worst setup slack.
- `violator_list.rpt` Setup and hold violator endpoints.
- `checks.rpt`: It contains a summary of the following checks:
  1. Max capacitance violations
  2. Max slew violations
  3. Max fanout violations
  4. Unconstrained paths
  5. Unannotated and partially annotated nets
  6. Checks the SDC for combinational loops, register/latch with multiple clocks or no clocks, ports missing input delay, and generated clocks
  7. Worst setup or hold violating path

> **See also**
>
> Check out our STA and timing closure guide for a deeper dive into what you can do to achieve timing closure when violations actually occur.

Antenna Check

Long metal wire segments that are connected to a transistor gate may damage the transistor's thin gate oxide during the fabrication process due to its collection of charges from the processing environment. This is called the antenna effect (Also, called Plasma Induced Gate Oxide Damage). Chip foundries normally supply antenna rules, which are rules that must be obeyed to avoid this problem. The rules limit the ratio of collection area and drainage (thin oxide) area. Antenna

effect can be avoided by instructing the router to use short wire segments and to create bridges to disconnect long from transistor gates during fabrication. This approach is not used by OpenLane as OpenROAD routers don't support this methodology. Instead, OpenLane uses another approach that involves the insertion of an antenna diode (provided as a standard cell) next to the cell input pin that suffers from the antenna effect. Antenna diode cell has a reversed biased diode which can drain out the charge without affecting the transistor circuitry.

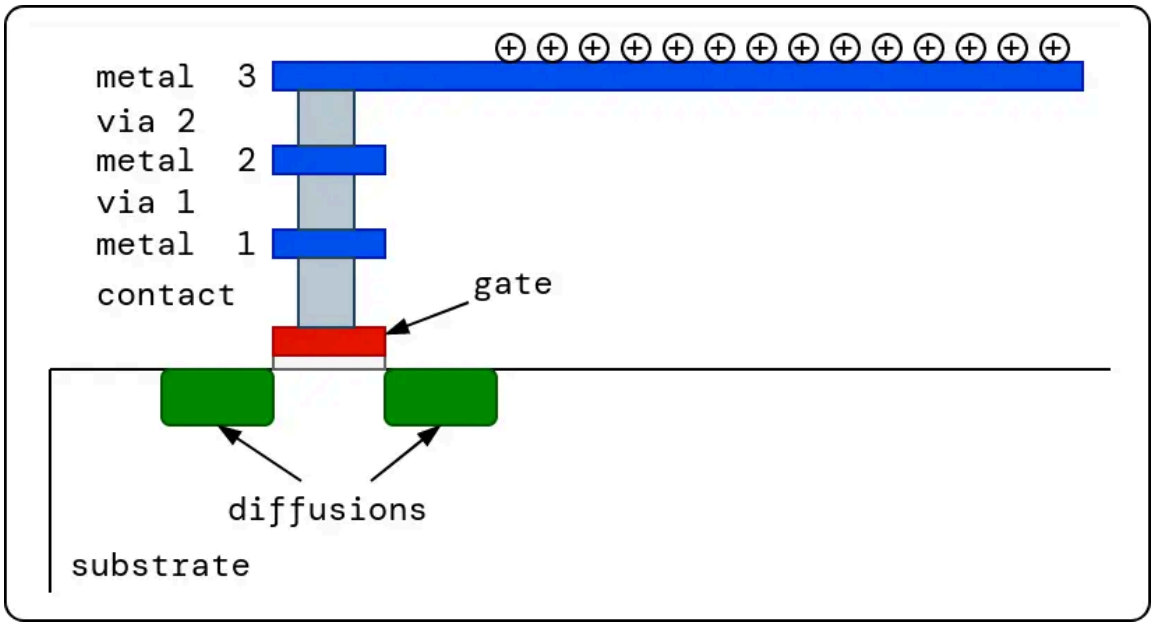

Fig. 11 Antenna effect

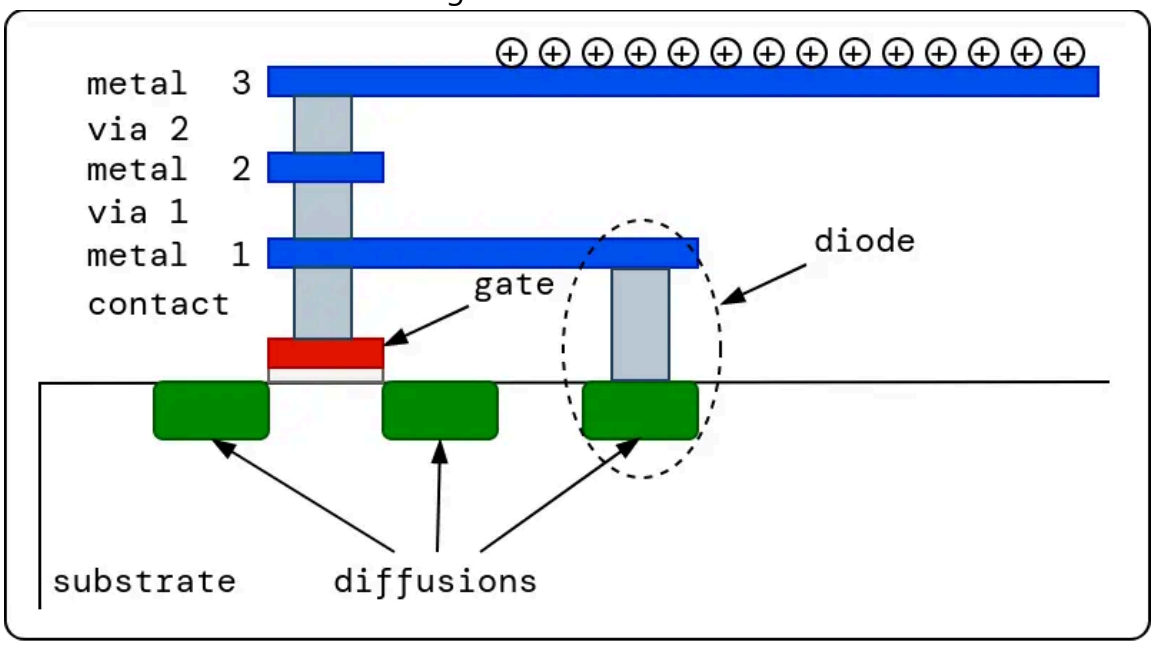

Fig. 12 Antenna diode insertion

The default flow runs a step called `OpenROAD.CheckAntennas` to check for antenna

rules violations.

> **Note**
>
> The checker runs thrice during the flow:
>
> 1. Once immediately after global routing
> 2. Once as part of the `OpenROAD.RepairAntennas` composite step
> 3. During signoff
>
> First two are ran to see the impact of antenna repair step.

Inside the step directory of `OpenROAD.CheckAntennas`, there is a `reports` directory that contains two files; the full antenna check report from `OpenROAD` and a summary table of antenna violations:

```
┏━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━┳━━━━━━━┳━━━━━━━━━━━━━┳━━━━━━━┓
┃ Partial/Required ┃ Required ┃ Partial ┃ Net   ┃ Pin         ┃ Layer ┃
┡━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━╇━━━━━━━╇━━━━━━━━━━━━━╇━━━━━━━┩
│ 1.43             │ 400.00   │ 573.48  │ net162 │ output162/A │ met3  │
│ 1.28             │ 400.00   │ 513.90  │ net56  │ _1758_/A1   │ met3  │
└──────────────────┴──────────┴─────────┴────────┴─────────────┴───────┘
```

Adding custom pin placement configuration

As seen in the previous layout, the pins of the `pm32` macro are placed randomly. If this macro will be integrated into a larger macro or chip, the pin placement should be according to the top-level connectivity. For example, if the clock pin will be derived from a clocking macro that will be placed on the east side of this macro, then the clock pin in the `pm32` should be placed on the east side. In order to change pin placement, we can add a pin placement configuration file to our flow as follows:

1. Create the file `~/my_designs/pm32/pin_order.cfg`
2. Add the following content to it

pin_order.cfg

3. Add `"FP_PIN_ORDER_CFG": "dir::pin_order.cfg"` to the config.json so the new config file will be like this

config.json

4. Run the flow again using the following command:

```
[nix-shell:~/openlane2]$ openlane ~/my_designs/pm32/config.json
```

5. Check the final layout again using this command:

```
[nix-shell:~/openlane2]$ openlane --last-run --flow openinklayout ~/my_designs
```

Since the configuration file `pin_order.cfg` has `#E` then `clk`, `rst`, `start`, then `done`, we will find those 4 pins on the east side of the macro. Similarly, we will find `mc` and `mp` buses on the south and `p` on the north of the macro.

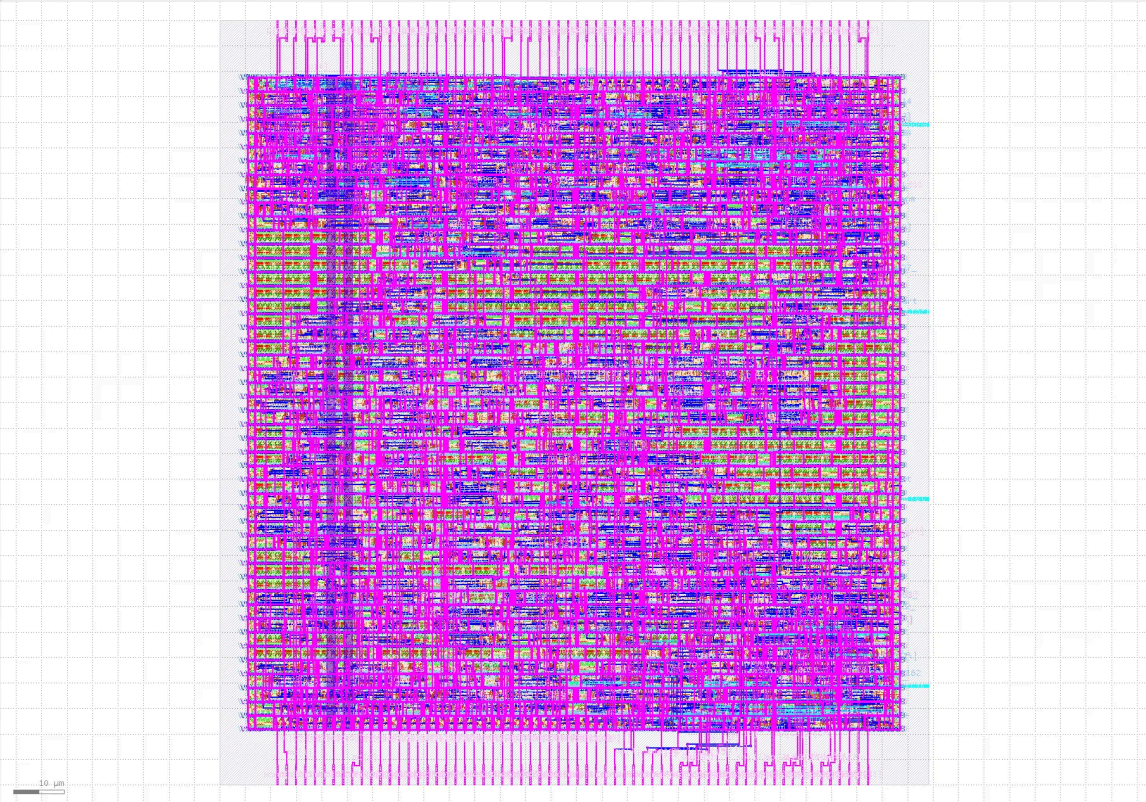

Fig. 13 Custom IO placed layout