Migrating from OpenLane 1

Version 2 of OpenLane is a complete re-imagining of OpenLane not just as a simple, somewhat customizable flow, but rather as an infrastructure that can support innumerable flows.

Being rebuilt from the ground up, there is a small learning curve to adopting OpenLane 2. This document aims to help those making the jump.

Why migrate?

At a minimum, the default flow for OpenLane 2, named `Classic`, is essentially a more robust re-implementation of the OpenLane 1 flow that is still entirely backwards compatible, with some conveniences:

> **Note**
>
> While the OpenLane 2 infrastructue is stable, the default OpenLane 2 flow in itself is in beta, pending silicon validation, and we still recommend the OpenLane 1 flow for use with OpenMPW and chipIgnite. See the FAQ for more information.

- Full configuration validation: if you have a typo, it will be caught, and if you accidentally provide a string to a number, it will be caught.
- More graceful failures: if the design fails mid-flow, because of a more strict separation of concerns, you still have access to metrics and reports from all previous steps.
  - In OpenLane 1, they are all extracted at the end.
- The ability to use command-line flow control options such as `--from`, `--to`, `--skip` and `--only`, with the ability to resume from a snapshot of your design at certain parts of flows, without worrying about surprises related to state variables missing.

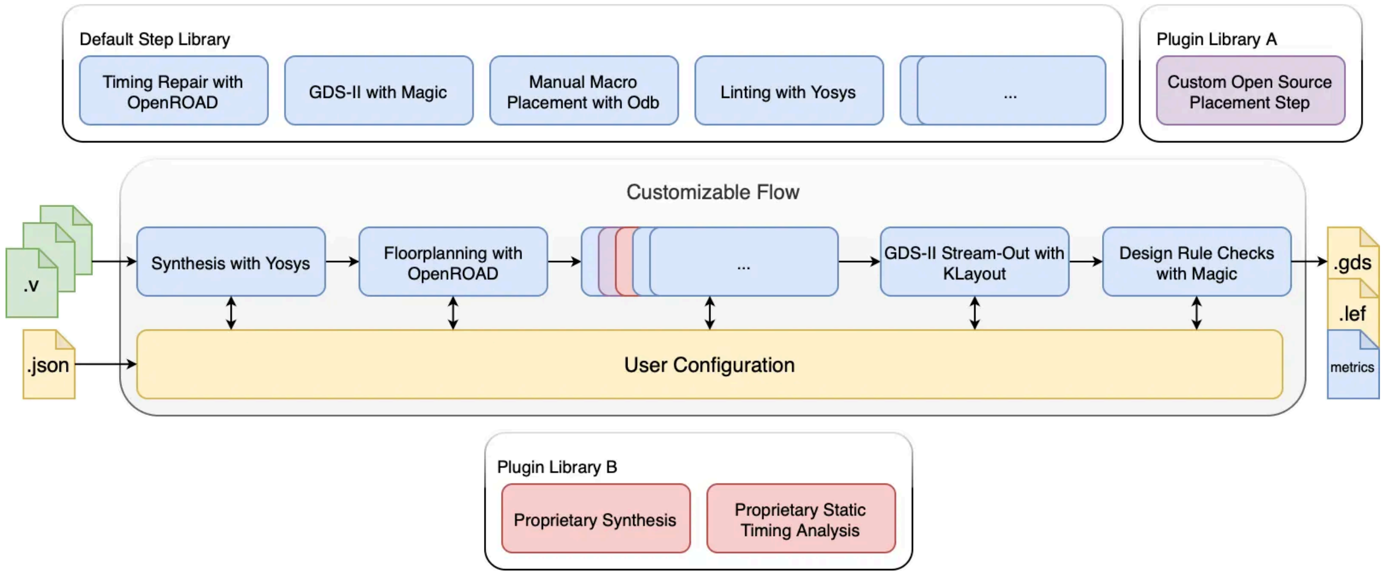
Default Step Library
Timing Repair with OpenROAD
GDS-II with Magic
Manual Macro Placement with Odb
Linting with Yosys
...
Plugin Library A
Custom Open Source Placement Step
Customizable Flow
.v
.json
Synthesis with Yosys
Floorplanning with OpenROAD
...
GDS-II Stream-Out with KLayout
Design Rule Checks with Magic
User Configuration
.gds
.lef
metrics
Plugin Library B
Proprietary Synthesis
Proprietary Static Timing Analysis

Fig. 1 Writing custom flows and steps using OpenLane 2

Additionally, if you're a more savvy user, a *whole new world* of possibilities await with OpenLane 2. Built around "flows" composed of "steps," OpenLane 2 can implement hardware flows for ASIC implementation by relying on basic Python object-oriented programming principles, and this naturally allows you to:

- Write your own step, in Python. For example, create a custom placement step for your design using OpenROAD Tcl or Python scripts, or even completely custom code.
- Write complex flows with decision-making capabilities, including the ability to repeat some steps or even try multiple strategies simultaneously and proceed with the best result.
  - You can even do this in a Python Notebook!
- Access a standardized and more formal form of design metrics based on METRICS2.1.

For example, using a custom OpenLane 2-based flow, the team over at TinyTapeout were able to integrate dozens of tiny designs together into a complex chip; leveraging custom flows and custom steps to tape-out a complex chip for ChipIgnite.

Installation

Like OpenLane 1, installations of OpenLane 2 include all underlying utilities for the default flow, including but not limited to; Yosys, OpenROAD, Magic, and KLayout.

OpenLane 2 uses a deterministic and reproducible environment builder called Nix to both build its underlying utilities and distribute them.

While using a Dockerized environment is still supported, Nix yields a number of advantages and is overall recommended.

Nix-based Installation (Recommended)

The Nix method involves installing the Nix build utility/package management software and cloning the OpenLane repository.

You can install Nix and set up the OpenLane binary cache by following the instructions at Nix-based Installation.

Afterwards, you can run an example as follows:

1. In your terminal, clone OpenLane as follows

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

2. Start a shell with OpenLane and its underlying utilities installed

```
nix-shell openlane2/shell.nix
```

3. Copy and run the `spm` example under a folder named `my_designs`

```
mkdir my_designs
cd my_designs/
openlane --run-example spm
```

Docker-based Installation (Not Preferred)

Docker is still supported if you have it installed from OpenLane 1, although the Docker image is built with a Nix environment instead of CentOS. The way it is invoked is also much simpler, with the Python script handling mounts and calling the image for you, as you can see below:

OpenLane <2.0

```
git clone https://github.com/The-OpenROAD-Pr
make pdk
make mount
./flow.tcl -design spm
```

OpenLane ≥2.0

```
pip3 install --upgrade openlane
openlane --dockerized --run-example spm
```

This allows you to start the OpenLane environment from anywhere, without having to rely on a Makefile.

Warning

`--dockerized` will make your home folder, your PDK root and your current working directory available to the OpenLane Docker container.

If you need any other directories mounted, you can pass them in the following manner: `--docker-mount <dir1> [--docker-mount <dir2> [--docker-mount <dir3>]]`.

Designs and Configuration

The first question you may ask is if your existing designs are supported, and we're happy to say that, for the most part, the answer is yes! OpenLane 2 supports about 99% of the config files from OpenLane 1, whether they're written in JSON or Tcl, although Tcl is finicky at this point and we recommend rewriting them in JSON.

> **Note**
>
> A small caveat is interactive scripts written in Tcl, however, are not supported in OpenLane 2- we've replaced them with Python-based flows, which are much more flexible and stable: you can check out how to write one under Writing Custom Flows.

A very limited number of variables, are a bit more of a pain to migrate. We've documented them in the following sections for your convenience:

- Variable Migration Guides
  - Migrating `FP_PDN_VOFFSET` / `FP_PDN_HOFFSET` / `FP_PDN_VPITCH` / `FP_PDN_HPITCH` for smaller designs
  - Migrating `DIODE_INSERTION_STRATEGY`
  - Migrating `MACROS` (Optional, but highly recommended)

Running flows

The command line interface for OpenLane 2 is more streamlined, and entirely handled by the OpenLane 2 Python module instead of relying on Makefiles.

PDK Installation

OpenLane <2.0

```
make pdk
```

OpenLane ≥2.0

```

```

OpenLane 2 will automatically download and install the default PDK(s) version using Volare.

> **Tip**
>
> Don't forget to add `--dockerized` to `openlane` invocations if you're using the Docker-based installation.

Installation Smoke-Testing

OpenLane <2.0

```
make test
```

OpenLane ≥2.0

```
openlane --smoke-test
```

Being built into the command-line interface of OpenLane makes it runnable from anywhere.

Running an example design

OpenLane <2.0

```
./flow.tcl -design spm
```

OpenLane ≥2.0

```
openlane --run-example spm
```

The example designs are now copied to the current working directory before being run, instead of relying on a global “OpenLane installation” directory.

Tip

Whenever you don’t specify a flow, the `Classic` flow will be used by default.

Running your own design

OpenLane <2.0

```
./flow.tcl ~/my_designs/picorv32
```

OpenLane ≥2.0

```
openlane [--flow Classic] ~/my_designs/picor
```

We’ve done away with providing the design folder and automatically trying to detect which configuration file should be run, instead opting to use the configuration file automatically.

Outputs

Examining outputs for OpenLane 2.0+ is very different compared to previous versions.

Run Folders

OpenLane <2.0

```
runs/<run_tag>
├── OPENLANE_VERSION
├── PDK_SOURCES
├── cmds.log
├── config.tcl
├── config_in.tcl
├── errors.log
├── logs
├── openlane.log
├── reports
├── report.csv
├── results
├── runtime.yaml
├── tmp
└── warnings.log
```

OpenLane ≥2.0

```
runs/<run_tag>
├── final
├── tmp
├── error.log
├── info.log
├── resolved.json
├── warning.log
├── 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
⋮
```

For one thing, run folders have been redone entirely: instead of a haphazardly and and ill-defined set of meta-step folders, each step now has its own folder.

By inspecting a step's folder, you'll find its outputs, including design views, logs and reports. Each step is only allowed to write within its own step folder, so to find the LVS report, for example, you'll have to look for the folder `*-netgen-lvs`.

Final Views

OpenLane <2.0

```
./results/final
```

OpenLane ≥2.0

```
./final
```

Fairly straightforward translation. There may be a number of new views for OpenLane ≥2.0 runs, but all the views from previous versions should continue to exist.

Final Resolved Configuration

OpenLane <2.0

```
./config.tcl
OPENLANE_VERSION
```

OpenLane ≥2.0

```
./resolved.json
```

The final resolved configuration after loading defaults, values from the PDK, the configuration file and any command-line overrides, and also some metadata such as the flow used and the version of OpenLane.

The generated `resolved.json` is a valid OpenLane configuration file for the same flow; so you may re-run a flow with the same exact configuration as follows:

```
openlane <path to run folder>/resolved.json
```

Warning/Error Logs

OpenLane <2.0

```
./errors.log
./warnings.log
```

OpenLane ≥2.0

```
./error.log
./warnings.log
```

In OpenLane 2, the error log lacks the level prefix (`[ERROR]`/`[WARNING]`) in the files themselves.

Metrics

OpenLane <2.0

```
./report.csv
```

OpenLane ≥2.0

```
./final/metrics.csv
./final/metrics.json
```

The CSV table is transposed in comparison to OpenLane 1, i.e., where the fields were columns in legacy versions, they have been made into rows into the new versions, as metrics may vary greatly depending on the flow.

A more computer-friendly JSON representation of the metrics is also available.

Timing and Clock Reports

OpenLane <2.0

```
./reports/signoff/*-sta-rcx_<interconnect co
./reports/signoff/*-sta-rcx_<interconnect co
```

OpenLane ≥2.0

```
./*-openroad-stapostpnr/summary.rpt
./*-openroad-stapostpnr/<IPVT corner name>/{
```

One summary table is created for all corners, instead of being grouped by interconnect corner.

Other reports however are per “IPVT”-(Interconnect, Process, Voltage, Temperature,) corner.

Viewing Layouts

Instead of relying on an external script similar to OpenLane 1, OpenLane 2 implements flows to allow you to load your designs into a number of the GUI tools included with OpenLane.

Opening final GDS in KLayout

OpenLane <2.0	OpenLane ≥2.0
`python3 ./gui.py --viewer klayout --format g`	`openlane [--run-tag <run tag>|--last-run] --`

Opening in KLayout is implemented as a one-step flow named, well, `OpenInKLayout`.

OpenLane 2 allows you to run multiple flows in the same run directory, and thus opening the run in KLayout is just another step. When you do so, the last state of the design is used as an input, meaning that KLayout will preview the latest GDS stream-out in the design.

Opening earlier DEF view in KLayout

OpenLane <2.0	OpenLane ≥2.0
`python3 ./gui.py --viewer klayout --stage ro`	`openlane --with-initial-state <run folder>/*`

For steps of the flow where there is no GDS view yet, `OpenInKLayout` will preview the DEF view instead. You can tell OpenLane 2 which state to use explicitly, where here we’ve opted for the output state of the detailed routing step.

Opening in OpenROAD

OpenLane <2.0	OpenLane ≥2.0
`python3 ./gui.py --viewer openroad --stage r`	`openlane --with-initial-state <run folder>/*`

Similar to KLayout, opening designs in OpenROAD is implemented as a one-step flow named `OpenInOpenROAD`. Like with KLayout, you can give it a run folder.