OpenLane 2: Making the Most Popular Open Source ASIC Flow Modular

Mohamed Gaber*, Kareem Farid*, Bassant Hassan*, Mohamed Shalan†
{donn,kareem.farid,bassant.hassan}@efabless.com, mshalan@aucegypt.edu
* Efabless Corporation
† The American University in Cairo

***Abstract*—We present OpenLane 2, a re-imagining of the world's most popular open-source RTL-to-GDSII flow as a modular infrastructure for flow creation. We show how the infrastructure has been architected to make flows more flexible, and crucially, repeatable. We demonstrate how we enhanced the flow's distribution and delivery using the Nix build and deployment system. Finally, we show some of the active current uses of OpenLane 2 in industry and academia.**

***Index Terms*—eda, electronic design automation, asic, asic flows, digital design**

I. Introduction

The releases of OpenROAD in 2019 [1] and the Open Source Google/Skywater 130nm Process Design Kit (PDK) in 2020 [2] heralded what could only be described as a revolution in breaking down the barriers to application-specific integrated chip (ASIC) development. Much like Yosys [3] and Project IceStorm [4] enabled a vibrant open-source field-programmable gate array (FPGA) scene, the once-secretive ASIC industry was now made accessible free of charge and without restrictions to academia, corporations, and hobbyists alike. This was further accelerated by the Google OpenMPW program, where Google sponsored a number of multi-project wafer (MPW) shuttles for open source designs, letting designers for the first time design, manufacture and get an ASIC chip without signing a single non-disclosure agreement [5].

Preparing for the mass enablement of a new, wide audience to try chip design was, however, no small task. It was quickly made apparent that the otherwise disjoint open-source utilities: Yosys, OpenROAD, Magic [6], KLayout [7] needed to be harmonized into a turn-key flow going all the way from register-transfer-level (RTL) Verilog code to the final graphical layout used by the foundries, GDSII. This culminated in the release of OpenLane [8][9] in late 2020, which was positioned as the primary flow for the Google OpenMPW project. OpenLane has since endured as the world's most popular open-source RTL-to-GDSII flow with over 1200 stars on GitHub [10], in no small part due to its ease of installation and use; until 2024, OpenLane was the only flow to bundle its dependencies in the form of a container so users may install and use it within a matter of minutes.

Yet, OpenLane was not without its limitations. For most digital MPW projects, few ran into issues, but under the hood, the flow amounted to little more than a set of Tcl procedures haphazardly manipulating the global state and calling tools. Additionally, Tcl's lack of data-types and lack of a separation between data and code makes it very difficult to maintain a large and robust codebase which impeded our and community contributors' ability to extend OpenLane– despite these very features making Tcl an excellent embedded language for adding a command-line interface to a large utility which it owes its popularity in the EDA space to. Finally, the lack of modularity made it borderline impossible to mix-and-match open source and proprietary tools, which combines the flexibility of the former with the robustness of the latter to ensure a great quality of results.

We thus endeavored to re-imagine OpenLane - maintain its trademark ease-of-installation and ease-of-use, yet, do so re-envisioning OpenLane as not just a flow, but as an infrastructure with which flows can be created, extended, manipulated and maintained, with the primary goal still maintaining full-compatibility with its less modular predecessor. We referred to this project as **OpenLane 2**, which we released in late 2023.

II. Related Work

Unlike in late 2020 through early 2021, there is no shortage of options of open-source ASIC flows to choose from. A number have since endeavored to create open-source RTL-to-GDSII flows; both from industry (SiliconCompiler [11]), Academia (mflowgen [12], Hammer [13]) and joint ventures between industry and academia (OpenROAD Flow Scripts). The current second-most popular flow OpenROAD Flow Scripts (ORFS), a GNU Make and Tcl-based flow which acts as a reference implementation of flows using OpenROAD. While more modular than OpenLane, the issues from the languages used largely persists: Makefiles make it very difficult to encapsulate state or implement complex flow logic, though that conversely enables interesting experiments such as Bazel-ORFS [14], which uses the hermetic and reproducible Bazel [15] build-system to bring better traceability and incremental building to ASIC.

Another popular flow is SiliconCompiler, an option developed by a team headed by Andreas Oloffson, one of the founders of the OpenROAD Project. Unlike either OpenLane or OpenROAD, SiliconCompiler is presented as a build-system: "Make for chips", as it were, where all chips would be implemented as a Python script using the SiliconCompiler

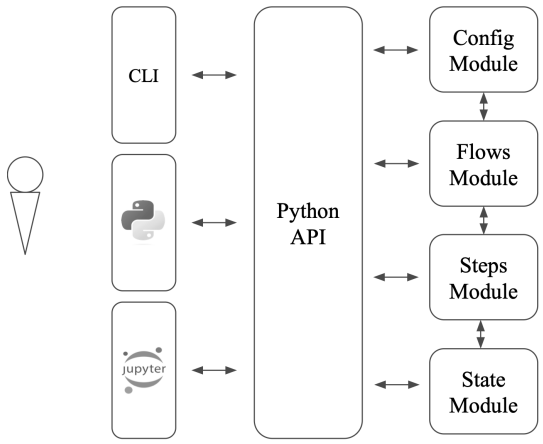

Fig. 1. A high-level view of OpenLane 2's Architecture

library, and ASIC implementation flows can be represented as graphs. The project is very ambitious and covers a lot more ground than either OpenLane or OpenROAD Flow Scripts, however, with that came complexity to the end user as configuring and running a flow with a single Tcl or JSON file in the style of OpenLane is not possible. Additionally, unlike OpenLane or OpenROAD Flow Scripts, SiliconCompiler never provides a batteries-included option for installation or running, electing to provide remote machines to alleviate installation issues. Shifting this burden to the users, however, is a prime area for misconfiguration issues to happen or for mismatched versions to cause crashes, severely degrading the flow quality.

All of these projects are valiant efforts in their own right, however OpenLane does (and continues to) occupy a certain niche that none of them adequately serve: (relative) ease-of-installation and absolute ease-of-use. In OpenLane, packaging and distribution is a first class concern co-equal with the flow, and OpenLane 2 includes a default/reference flow named "Classic" that still, like OpenLane 1, can be configured with exactly one Tcl or JSON file (indeed, in overwhelming majority of cases the *same* as OpenLane 1).

III. Architecture and Implementation

A key part of OpenLane 2's design is the high-level architecture. The software architecture for OpenLane 2 is fully codified and rather strict to ensure code quality and modularity. The architecture is composed of a unified API, which can then be used either via commandline using the `openlane` command or programmatically. This is done to ensure users can easily create custom flows using the OpenLane 2 infrastructure, having access to the very same tools OpenLane developers do.

OpenLane 2 is implemented in Python 3, the world's most popular programming language (in no small part due to current AI boom) [16]. Regardless, as a dynamically typed language, Python makes it very difficult to maintain large, robust codebases, which is why we include static type checking using *mypy* [17]; which greatly enhances the API's consistency and helps avoid otherwise frequent type errors. The Python API can be used directly from Python using `import openlane`, which also enables OpenLane to be easily integrated into the popular Google Colaboratory [18] environments. The Python API is composed of four modules, `state`, `step`, `flow`, and `config`, as shown in Figure 1.

A. `state`

At the core of OpenLane 2 lies the **State** module. The State module is centered around its namesake `State` object, which is an encapsulation of the current status of a design under implementation. Specifically, captured in the form of a an immutable mapping from **Design Formats** - various text and binary representations of a design (such as Verilog Netlist, Design Exchange Format (DEF), etc.) to file paths.

As the structure is immutable, creating new States requires creating a copy of the object. This is by design, so the flow may be programmatically "rewound" to any point (as no states are discarded.)

B. `step`

The **Step** module centers around its namesake, the Step class. The Step class is an encapsulation of a transformation on the State object. Specifically, each Step expects a **Configuration object** and a State object with one or more Design Formats as its input, and emits a new, altered State as its output.

Steps operate within what is known as a **step directory**. The step directory acts as a form of "scratch space" for a Step: it is where steps are allowed to create intermediate and output files (including reports, logs, and whatnot.) Steps will save a copy of their inputs and output state in a JSON format within the Step directory for traceability.

Steps in OpenLane abide by a set of strictures that serve an ultimate goal: **The same step with the same input configuration and same input state must emit the same output**. The strictures are as follows:

1) Steps are not allowed to create any files outside of the step directory.
2) Steps are not allowed to mutate any files. If the tool does not support out-of-place modification, Steps must copy the files into the step directory then modify the copies.
3) Steps do not modify the input configuration or state object.
4) Steps do not rely on filesystem paths outside of those listed in the configuration object thereof with the exception of temporary directories and `$PATH` to find executables.

All of these strictures work towards to goal of making OpenLane 2 steps **hermetic**. A hermetic build process is a process for which all inputs are explicitly and unambiguously specified externally to the build process itself [19]. Hermeticity is all but a necessary precursor for what is the actual goal, reproducibility: i.e., the same set of inputs should produce the same set of outputs, which logically is not possible without an explicit and unambiguous set of inputs.

While pure hermeticity would be too much of a restriction upon engineers, requiring hashes of inputs from the filesystem or similar, OpenLane Flows and Steps are highly repeatable unless either the Flow or Step in question violates a stricture or attempts to access the network or a user chooses to modify files on their filesystem haphazardly. Nevertheless, this in turn enables a limited form of reproducibility across different setups, which helps speeds up the diagnosis of bugs and faults within the flows, steps or inputs to ensure quality.

Outside of those strictures, Steps are free to use one or more executables or libraries to achieve their stated functionality, which is usually an EDA process of some kind: an example would be Synthesis of a list of Verilog RTL files, or Floorplanning an OpenROAD database.

With these strictures in place, results from OpenLane on the same operating system tend to be reproducible, even across hardware architectures, e.g., the step `Yosys.Synthesis` returns an identical result on both x86 and ARM-based devices.

Steps have a string-based ID that is registered to a global factory method so they may be retrieved by name. This is useful for constructing custom flows which will be detailed later. This global factory method has no effect on the execution of a successfully compiled flow and does not affect reproducibility. OpenLane 2 comes with a number of built-in steps chiefly based on those from its predecessor.

C. `flow`

Much like Step, the **Flow** module centers around the Flow class. While Steps can technically be run individually, Flow classes are aggregations of steps, which offers a number of advantages:

- **Flow control**: Flows may include logic as to if and how the steps would be run, including skipping steps, running only some steps, or even running multiple steps in parallel.
- **One-shot configuration**: Flows compile the configurations for all steps simultaneously. This verifies ahead of time that no two incompatible steps are used within the same flow.

An invocation of a Flow creates a **run directory**, which would contain all the step directories from earlier. Flows are, in fact, responsible for giving names to the step directories.

An abstract child of the Flow class is `SequentialFlow`. Sequential Flow is used for the common Flow pattern where Flows are simply a number of Steps run serially, where $State_i = Step_i(State_{i-1})$. OpenLane 2, by default, uses a Sequential Flow named **Classic** that emulates the behavior of the OpenLane 1 flow.

Custom flows may be written in Python using the API, but for convenience, they may also be constructed from within configuration files by simply listing Step IDs. This allows PnR engineers to quickly construct custom flows without writing any Python code.

Flows require a configuration object as an input, alongside an optional initial state object. If an explicit initial state object is not provided, the latest State saved to the Flow directory will be used. If the Flow directory is newly created or empty, an empty initial state will be created. This behavior allows flows to be stopped and resumed, or multiple runs to branch off from an initial run.

Flows have a string-based ID that is registered to a global factory method, so they may be retrieved by name. This allows flows to be picked from the commandline or for configuration files to specify their own flow.

D. `config`

Both Steps and Flows may be configured to alter their behavior. These configuration options are exposed to users via **configuration variables**. These variables essentially consist of a name, a type, and an optional default value. There exists three kinds of those variables:

- **Universal**: Variables that exists for all flows and all steps, containing critical information about the current design or process design kit.
- **Flow-specific**: Variables that exist as part of a specific Flow (but not its constituent steps). Typically, those are variables that control the flow execution in some manner.
- **Step-specific**: Variables that exist as part of a specific Step. These either provide the Step with essential information for execution or controls the step's execution in some manner.

Each variable may also be marked as a PDK or non-PDK variable. This only affects where the step gets its default value; non-PDK variables have their default stored in code, whereas PDK variables have their defaults stored in a PDK configuration file (which comes included with an installed PDK.) The user may use use configuration files (user configs) to explicitly specify the value of either type of variable.

Each Variable may be assigned a default value, but are otherwise required to be specified by either the PDK or the user. Variables can be of option types, a sum type of `None` and either a scalar or product type [20], where they hold an implicit default of `None`. All variables will ultimately share the same namespace for an entire flow, so if two steps have two variables with the same name but different types or default values, the steps are considered mutually exclusive and incompatible.

The configuration variables of a flow are considered to be the sum of the universal flow configuration variables, steps' configuration variables plus any flow-specific configuration variables. These variables are then used to compile a user-input Tcl, JSON, or YAML file into an immutable mapping representing a particular flow run's configuration. This immutable mapping is then filtered per-step so each step only has access to values for configuration variables it explicitly declares (a sore spot with the global state/configuration from OpenLane and other flows.)

IV. Packaging and Distribution

For the longest time, OpenLane's primary appeal was its ease of installation using Linux userspace containers, more specifically, using Docker images. Unlike traditional installation methods (`apt`, `yum`, etc.), containers mostly guarantee

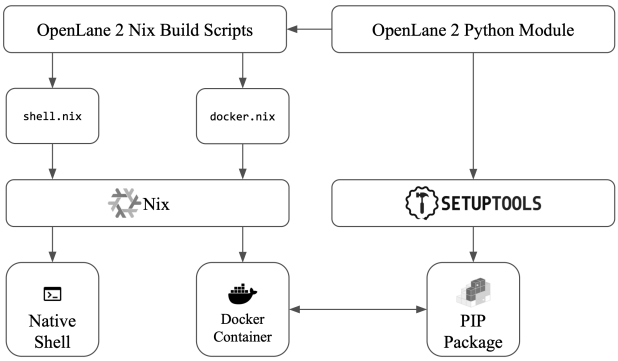

Fig. 2. A diagram showing how Nix is used to bundle OpenLane utilities

a consistent experience across different Linux distributions (and indeed across operating systems by virutalizing Linux, which adds Windows and macOS compatibility.) Container technologies, however, come with a number of unpleasant limitations, specifically:

1) **Complexity around using GUIs**: GUIs for Docker applications require X11 forwarding to be set up, which requires special software on macOS and the Windows Subsystem for Linux, and is frequently interfered with by kernel-based security solutions such as SELinux or AppArmor.
2) **Lack of full filesystem access**: Interactions between filesystems require an explicit bind mount between the primary userspace and the container userspace. This is actually desirable behavior in some cases for security reasons, but ends up being a hindrance when users need to, for example, use a design located on an NFS store or similar.
3) **Giant update deltas**: Docker images are composed of layers, whose caching is naïvely invalidated if the command is changed to create a layer or if any previous layers are changed, even if there is no causal relationship between the two layers (i.e. they affect different parts of the filesystem.)

To address all of these issues, we adopted the Nix build utility and deployment system [21]. Nix is a tool for reproducible builds powered by a functional programming language of the same name, supporting both (systemd) Linux and macOS for both x86-64 and ARM64 machines. With complete hermeticity (including but not limited to performing all builds in a sandbox), Nix build tasks have reasonably reproducible results (unlike Docker, where most image builds are non-hermetic and are directly exposed to changes in internet files) that can be cached. By making a public binary cache available, users may get a full, native OpenLane shell environment on either macOS or Linux by simply typing `nix-shell` in the OpenLane folder.

Nix is also sufficiently flexible that it is possible to generate a Docker image from a Nix-based environment, which is useful for containerized deployments and/or holdouts from OpenLane 1 that do not wish to install Nix. Nevertheless, the new environments provide native GUI support on both Linux and macOS, full native filesystem access, and smaller deltas (as only updated tools and their dependencies need to be re-downloaded), all while still guaranteeing functionality across Linux distributions and versions of macOS 11+. The only improvement possible upon this delivery method is bespoke builds for every Linux distributions.

Alongside the Nix builds, OpenLane 2 is also distributed using the Python Package Index (PyPI), which can be accessed using the PIP utility. This provides an avenue to automatically pull and use the Docker image (if the `--dockerized` command-line flag is used) or for those who want to use their custom-built tools at their own risk. Figure 2 shows all the ways Nix and PIP package could be used.D

Plugins

Using Nix, OpenLane 2 bundles all Steps and utilities required for the default Classic flow, which are: Verilator, Yosys, OpenROAD, Magic, Netgen, and KLayout. However, it is unreasonable to expect the infrastructure to include all possible EDA tools for all possible Steps and Flows that users may come up with.

To allow users to add Flows and Steps without altering OpenLane itself, OpenLane supports the concept of plug-ins. Plugins are Python packages that start with the prefix `openlane_plugin_` that add more Steps and Flows to their respective factory methods. Plugins also are given a Nix-based API to include any free-and-open-source utilities. However, plugins may require their dependent utilities to be installed separately, which is the only option for plugins that support proprietary software. The plugins and built-in steps can then be used in harmony to create custom flows as shown in Figure 3.

V. Adoption

While the Classic flow is still in nominal beta pending silicon validation, OpenLane has received moderate adoption in some areas in both academia and industry, both as a pure open-source flow and as a mixed proprietary and open-source flow. A subset of these adopters follow:

A. Industry

1) Efabless: OpenLane 2 has been used internally at Efabless for a number of projects and entirely superseded its predecssor for all non-trivial chip design: using both the Classic flow and bespoke flows for complex designs. It has been deployed for a number of projects, including but not limited to: a re-implementation of Caravel, the next generation of Caravel, "Caravel Panamax," and the Cheetah TinyML chip.

Internally, we have also tested a number of plugins, most importantly a plugin for Synopsys® EDA tools which we are actively using. We are also currently developing a plugin for an alternatives to OpenROAD, including iEDA [22], to expand the options for tools available.

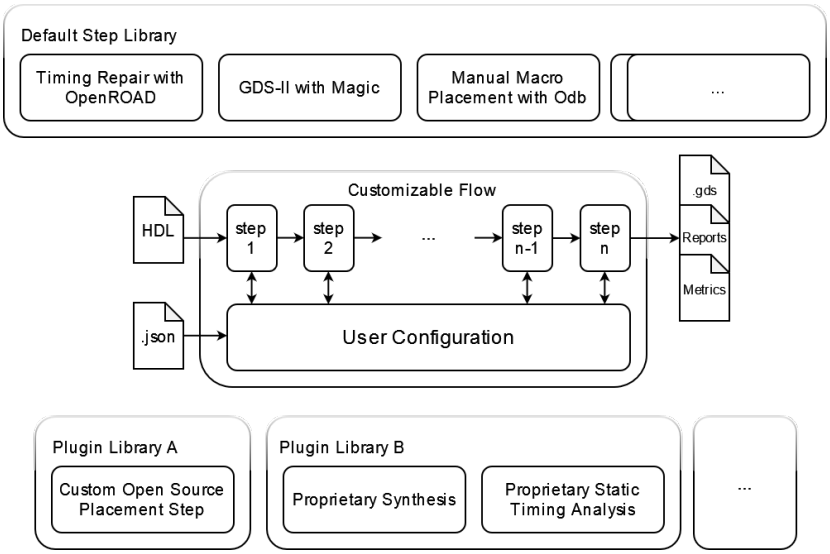

Fig. 3. A diagram showing how the built-in step library and plugins can be used to construct a flow in OpenLane 2

In all instances, OpenLane 2 was positively reviewed by place-and-route (PnR) engineers for its superior state-management (allowing for branching explorations of timing repair parameters, for example). Additionally, we had also and its ability to seamlessly seamlessly integrate Synopsys® DesignCompiler™ for increased area savings and Synopsys® PrimeTime™ for a more reliable signoff process for highly complex SoCs.

2) Tiny Tapeout: OpenLane 2 has also been used for every Tiny Tapeout [23] since Tiny Tapeout 4, where a custom OpenLane 2-based flow has been used to harden the final top-level multiplexer and the Classic flow was used to harden every single digital user project.

B. Academia

1) Purdue STARS: OpenLane 2 with the plugin for Synopsys® EDA tools was successfully used for the Purdue University STARS program, where students learned the basics of chip design using an OpenLane 2-based flow incorporating Design Compiler™ and PrimeTime™, which enabled the use of industry-standard proprietary utilities with minimal configuration.

2) Piel: OpenLane 2 is currently in use by Photonic Integrated ELectronics (Piel) [24]. A PhD project at the University of Bristol by Dario Quintero, Piel aims to provide an integrated workflow to co-design photonics and electronics for classic and quantum computing: OpenLane 2 is leveraged for hardening electronics.

VI. Conclusion

We have presented OpenLane 2, a reimagining of the world's premier open-source RTL-to-GDSII flow as an extensible architecture for creating countless flows while maintaining ease of installation, use and responsibility. We have outlined the architecture for OpenLane 2 and have shown why each design decision was taken and how it enhances the user experience and the reproducibility and traceability of flows built using OpenLane 2. We have shown how OpenLane 2 bundles its open source utilities using Nix, and yet using plugins, supports proprietary utilities with relative ease. As stated, OpenLane 2 internally at Efabless for a number of projects, and also readily adopted by the community across both industry and academia.

There are many possible avenues for future research and development of the OpenLane 2 infrastructure. One such avenue would be programmatic enforcement of architectural strictures to ensure users cannot write steps and programs which may cause non-repeatable behavior. Another possible avenue of research is in the realm of step portability, namely, trusted remote execution of steps, which could be used to provide secure access to proprietary utilities or simply off-load workflows to more capable machines.

Acknowledgments

OpenLane 2 is developed and maintained by Efabless Corporation. We would like to thank members of our community, especially the Tiny Tapeout team, for their eager adoption of and consistent feedback on the new infrastructure and Classic flow. We would also like to thank Mohamed Hosni [25] for contributing valuable input to the development of OpenLane 2 during his work on the Cheetah TinyML SoC.

OpenLane 2 is available at https://github.com/efabless/openlane2. Documentation, including full installation instructions, are available at https://openlane2.readthedocs.org.

An example dummy plugin is available at https://github.com/efabless/openlane_plugin_example, where we show how to create a plugin in Python and bundle utilities using Nix.

References

[1] Tutu Ajayi et al. "Toward an open-source digital flow: First learnings from the openroad project". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–4.

[2] Edwards, Tim. "Google/SkyWater and the Promise of the Open PDK". In: *Proceedings of the 3nd Workshop on Open-Source EDA Technology (WOSET)*. 2020. URL: https://woset-workshop.github.io/PDFs/2020/a03.pdf.

[3] Claire Wolf and Johann Glaser. "Yosys-A Free Verilog Synthesis Suite". In: *Proceedings of The 21st Austrian Workshop on Microelectronics*. 2013. URL: https://api.semanticscholar.org/CorpusID:202611483.

[4] Claire Wolf. *Project IceStorm*. 2015. (Visited on 09/22/2024).

[5] Skywater Inc. *First Google-Sponsored MPW Shuttle Launched at SkyWater with 40 Open Source Community Submitted Designs*. 2021. URL: https://www.skywatertechnology.com/first-google-sponsored-mpw-shuttle-launched-at-skywater-with-40-open-source-community-submitted-designs/ (visited on 09/22/2024).

[6] J.K. Ousterhout et al. "Magic: A VLSI Layout System". In: *21st Design Automation Conference Proceedings*. Albuquerque, NM, USA: IEEE, 1984, pp. 152–159. ISBN: 978-0-8186-0542-0. DOI: 10.1109/DAC.1984.1585789. URL: http://ieeexplore.ieee.org/document/1585789/.

[7] Matthias Köfferlein. *KLayout Layout Viewer and Editor*. URL: https://www.klayout.de (visited on 09/22/2024).

[8] Mohamed Shalan and Tim Edwards. "Building OpenLANE: A 130nm OpenROAD-based Tapeout- Proven Flow : Invited Paper". In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–6. URL: https://ieeexplore.ieee.org/document/9256623.

[9] Ahmed Ghazy and Mohamed Shalan. "OpenLANE: The Open-Source Digital ASIC Implementation Flow". In: *Proceedings of the 3nd Workshop on Open-Source EDA Technology (WOSET)*. 2020. URL: https://woset-workshop.github.io/PDFs/2020/a21.pdf.

[10] *Star history of The-OpenROAD-Project/OpenLane*. URL: https://api.star-history.com/svg?repos=The-OpenROAD-Project/OpenLane&type=Date (visited on 09/30/2024).

[11] Andreas Olofsson, William Ransohoff, and Noah Moroze. "A Distributed Approach to Silicon Compilation: Invited". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. San Francisco, California, 2022, pp. 1343–1346.

[12] Alex Carsello et al. "mflowgen: a modular flow generator and ecosystem for community-driven physical design: invited". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 1339–1342. ISBN: 9781450391429. DOI: 10.1145/3489517.3530633. URL: https://doi.org/10.1145/3489517.3530633.

[13] Harrison Liew et al. "Hammer: a modular and reusable physical design flow tool: invited". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 1335–1338. ISBN: 9781450391429. DOI: 10.1145/3489517.3530672. URL: https://doi.org/10.1145/3489517.3530672.

[14] Antmicro. *Bazel-orfs: Combining Bazel with OpenROAD Flow Scripts for rapid ASIC design iteration with caching*. 2024. URL: https://antmicro.com/blog/2024/08/bazel-orfs-for-rapid-asic-design-iteration-with-caching/ (visited on 09/22/2024).

[15] *Bazel build tool*. URL: https://bazel.build (visited on 09/22/2024).

[16] *TIOBE Programming Language Index*. URL: https://www.tiobe.com/tiobe-index/ (visited on 09/25/2024).

[17] *mypy documentation*. URL: https://mypy.readthedocs.io/en/stable/ (visited on 09/25/2024).

[18] *Google Colaboratory*. URL: https://colab.research.google.com/ (visited on 09/23/2024).

[19] Jeremiah Spradlin and Mark Lodato. "Building Secure and Reliable Systems". In: Google Inc., 2020. Chap. Deploying Code. URL: https://google.github.io/building-secure-and-reliable-systems/raw/ch14.html#hermeticcomma_reproduciblecomma_or_veri.

[20] Robert Harper. "Finite Data Types". In: *Practical Foundations for Programming Languages*. Cambridge University Press, 2012, pp. 79–80.

[21] Dolstra, Eelco. "The Purely Functional Software Deployment Model". PhD thesis. Utrecht, NL: University of Utrecht, 2006. URL: https://edolstra.github.io/pubs/phd-thesis.pdf.

[22] Xingquan Li et al. *iEDA: An Open-Source Intelligent Physical Implementation Toolkit and Library*. 2023. arXiv: 2308.01857 [cs.AR]. URL: https://arxiv.org/abs/2308.01857.

[23] Matt Venn. "Tiny Tapeout: A shared silicon tape out platform accessible to everyone". In: *IEEE Solid-State Circuits Magazine* 16.2 (2024). Conference Name: IEEE Solid-State Circuits Magazine, pp. 20–29. ISSN: 1943-0590. DOI: 10.1109/MSSC.2024.3381097. URL: https://ieeexplore.ieee.org/document/10584359 (visited on 09/22/2024).

[24] Quintero, Dario. "Photonic Integrated ELectronics". In: *The 4th Free Silicon Conference*. Sorbonne University, Paris, France, 2024. URL: https://wiki.f-si.org/index.php?title=Integrating_Mixed-Signal_Microelectronics_and_Photonics:_A_Co-Design_Approach_with_Piel (visited on 09/22/2024).

[25] Mohamed Hosni. *Mohamed Hosni on GitHub*. URL: https://github.com/mo-hosni.