

# Substrate: A Statically Typed Framework for Designing Highly Configurable Analog and Mixed-Signal Circuit Generators

Rahul Kumar, Rohan Kumar, Borivoje Nikolić

Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA

{rahulkumar, rohankumar, bora}@berkeley.edu

**Abstract**—Analog and mixed-signal (AMS) integrated circuit design is often a time-consuming and costly process, due in part to manual design flows and long layout iterations. A number of tools have been developed aiming to automate the process of creating AMS designs. However, existing tools are often difficult to use due to unclear application programming interfaces (APIs), limited levels of abstraction, or insufficient control over generated collateral. We introduce Substrate, an open-source, statically typed framework for creating highly configurable schematic and layout generators using the Rust programming language. Substrate provides multiple levels of abstraction, allowing designers to navigate the tradeoff between fine-grained control over a design and increased automation. We also describe algorithms for programmatically creating and modifying circuit layouts, including two methods for automatically adjusting the aspect ratio of a layout. We use Substrate to design generators for a StrongARM comparator and a programmable resistor bank in Skywater 130nm and Intel 16nm, demonstrating 90 degree rotation, array folding, and the ability to change the aspect ratio by a factor of over 10 in both processes. These generators highlight Substrate’s ability to facilitate design reuse, process portability, and performance and area optimization.

**Index Terms**—analog circuits, layout synthesis, design automation, routing, algorithms, optimization, open-source

## I. INTRODUCTION

The rapid development of advanced process nodes has highlighted the highly manual nature of traditional AMS circuit design methodologies. Stringent design rules and increased layout parasitics in fin field-effect transistor (FinFET) and gate-all-around (GAA) nodes have significantly increased the design effort required to iterate on post-layout designs. New process nodes with numerous device options and metal stackups continue to come out every few years, forcing analog designers to spend considerable amounts of time redesigning circuits in the most competitive node rather than designing new features. Automation for AMS circuit design akin to that available for digital circuits would not only enable faster time to market, but also allow for architecture exploration, post-layout optimization, design reuse, and process portability.

Layout automation is particularly interesting because creating a fabrication-ready layout is often a bottleneck in AMS design. Frameworks such as MAGICAL [1] and ALIGN [2] accept an input netlist and a set of constraints, and use automatic analog placement and routing (P&R) algorithms to

quickly generate layouts. However, these frameworks do not provide fine-grained control over the generated layouts, which often do not match what a human might do by hand. Other generator frameworks require users to encode their own layout generation logic (which may leverage framework-provided utilities), allowing users more control over the final layout [3]–[5]. The main drawback of these generators is that they often lack configurability, which we define as the ease with which new layouts can be created by varying generator parameters. The Berkeley Analog Generator (BAG) framework [3], for instance, provides layout abstractions to allow users to produce process-portable, parametric generators, but lacks autorouting utilities that would facilitate configurability. Additionally, BAG generators often use ambiguously typed Python dictionaries whose purpose can only be deciphered by reading the source code.

In this paper, we present Substrate, an open-source<sup>1</sup> framework for designing complex AMS schematic and layout generators. In addition to APIs for netlist templating and polygon manipulation, Substrate provides higher-level utilities for autorouting, tiling, and aspect ratio resizing. Substrate’s plugin system automates tool invocation and output parsing for design rule checking (DRC), layout versus schematic (LVS), parasitic extraction (PEX), and simulation, enabling programmatic analysis of simulation waveforms. Substrate’s generator-based design flow allows for rapid post-layout iterations, as illustrated in Figure 1. In a traditional design flow, design iterations are limited by layout and system integration efforts. In a flow using Substrate, the initial generator development is somewhat longer than an initial manual layout, but subsequent layout and system iterations are dramatically accelerated. Substrate improves upon existing generator frameworks by introducing statically typed APIs that 1) eliminate common errors at compile time and 2) facilitate generator composability without needing to understand internal implementation details. Substrate also provides a flexible level of layout control to the designer. A comparison between Substrate and other generator frameworks is shown in Table I.

Our main contributions are:

- We propose a set of intermediate representations (IRs) that facilitate schematic and layout portability across different

This work was supported in part by the Microelectronics Commons Program, a DoW initiative, NSF AI4OPT, and BWRC member companies.

<sup>1</sup>Source code is available on GitHub.

TABLE I: Comparison of AMS Generator Frameworks.

	MAGICAL [1]	ALIGN [2]	BAG [3]	This work
Open-source	Yes	Yes	Yes	Yes
Input language	SPICE, custom format	SPICE, JSON	Python	Rust
Framework language	Python, C++	Python, C++	Python, C++	Rust
Constraints	Symmetry	Symmetry, Placement	None	None
Supports programmatic layout entry	No	No	Yes	Yes
Supports manual routing	No	No	Yes	Yes
Supports automatic routing	Yes	Yes	No	Yes
Algorithms for re-parameterizing layouts	No	Yes, using constraints	No	Yes

formats, simulators, process design kits (PDKs), and physical verification tools.

- We propose a set of statically typed APIs that enable compile-time checks, richer editor autocomplete suggestions, and clarity while reading and writing generators.
- We propose a layout methodology called Automatic Translation of Logical Layout (ATOLL) that allows users to manually place and route certain portions of a layout and autoroute the rest, enabling the design of process-portable, highly configurable layout generators.
- We propose two algorithms for aspect ratio configurability, enabling rapid floorplanning iterations and helping users explore the impact of layout sizing on performance. *Tile resizing* selects device parameters for minimum total width given a maximum total height and fixed device strengths.

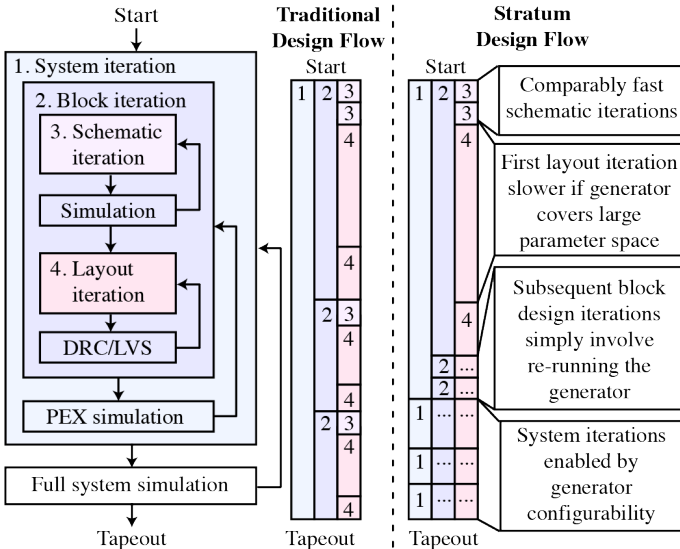


Fig. 1: Comparison of traditional and Substrate design flows. Both flows have the same steps (left), but configurable generators improve iteration speed.

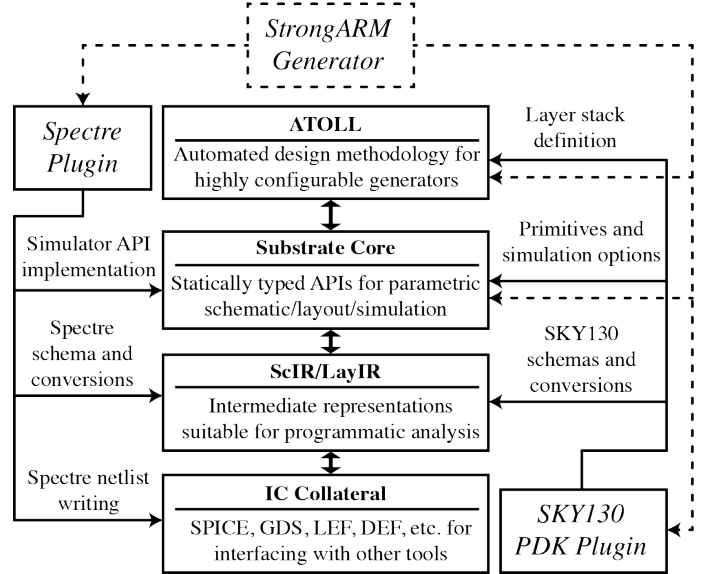


Fig. 2: Architecture of the Substrate framework.

*Segment folding* provides a systematic method for arranging and routing layouts involving a single repeated tile.

The remainder of this paper is organized as follows: Section II discusses the Substrate framework architecture, including details on its IRs and statically typed APIs; Section III describes the data structures and algorithms used to enable the design of highly configurable generators; Section IV describes generators for a StrongARM comparator [6], [7] and a programmable resistor bank that produce DRC- and LVS-clean instances across a wide parameter space in both Skywater 130nm and Intel 16nm; Section V concludes the paper.

## II. FRAMEWORK ARCHITECTURE

Substrate is a framework consisting of various libraries for designing AMS circuits written in the Rust programming language [8]. Rust has good performance (capable of approaching C/C++ [9], [10]), while also providing high-level language features such as memory safety, type safety, and robust tooling. Substrate libraries provide utilities for defining parametric schematic and layout generators, processing simulation waveforms, and interfacing with a variety of tools/PDKs. Substrate takes advantage of Rust’s type safety to provide clear APIs for defining AMS generators and builds upon low-level layout utilities provided by the Layout21 Rust library [5].

Substrate features a modular architecture with multiple levels of abstraction, enabling designers to exercise whatever level of control is appropriate for their application. Designers can import layouts from Virtuoso, programmatically manipulate polygons, or use higher-level abstractions such as autorouting. Tool-, process-, and circuit-specific libraries interact with the layer(s) of abstraction appropriate for their purposes. An overview of Substrate’s architecture is shown in Figure 2.

### A. IRs for Schematic and Layout

Substrate uses IRs to abstract away details of particular netlist or layout formats. Substrate’s IRs enable users to write

programmatic transformations on a relatively simple data format, without having to deal with the specifics of netlist syntax or layout records.

Substrate’s schematic IR (ScIR) represents data similar to SPICE, but introduces the concept of *schemas*<sup>2</sup>. All ScIR libraries have an associated schema, which defines the set of primitives that may be instantiated by cells of the library. For example, the `Sky130` schema enumerates the primitives allowed by the Skywater 130nm process (transistors, physical capacitors, physical resistors, etc.). A ScIR library in the `Sky130` schema is, at least in principle, physically realizable. On the other hand, a ScIR library in the `Spice` schema corresponds most directly to a SPICE netlist; it may be suitable for simulation or running LVS, but it may or may not be physically realizable, depending on the instantiated primitives. Schema conversions can be defined to describe appropriate netlist transformations. For example, a ScIR library in the `Sky130` schema can be converted to the `Spice` schema, allowing it to be easily exported to a SPICE netlist, or to the `Spectre` schema if a netlist in Spectre’s format is desired.

Schemas allow for precision in defining function signatures. For example, a plugin for a tool that performs PEX might take a ScIR library in the `Sky130` schema as an input, as it is only practical to run PEX on a physically realizable netlist. The plugin can then return a ScIR library in the `Spice` schema, since it will likely contain ideal resistors and capacitors that model extracted parasitics. Any attempt to pass a ScIR library with an incompatible schema to the plugin will fail at compile time, allowing the designer to fix the issue before invoking any other tools.

Substrate’s layout IR (LayIR) is very similar to ScIR, except that cells contain geometric shapes instead of connectivity information. Layout schemas define the set of allowable layers. For example, the `Sky130` layout schema enumerates named layers such as `Poly`, `Metal1`, `Metal2`, etc. The `Gds` schema defines layers as a pair of integers. Exporting a GDS file from a `Sky130` layout library involves providing a mapping from Skywater 130nm layers to GDS layer numbers. Similar mappings can be used to export to other file formats without changing the circuit generator itself.

### B. Statically Typed APIs

Substrate provides high-level, statically typed APIs for most of its features, including schematic entry, layout entry, and simulation. In order to prevent errors at compile time, Substrate encodes allowed behavior in the type system where other frameworks might use strings or dictionaries. For example, BAG addresses waveforms by string when retrieving them for programmatic analysis. If there is a typo in the string, an entire simulation may run before a `KeyError` is thrown in the analysis stage. This can have a significant impact if the analysis is part of a larger design script. In Substrate, making this error is not possible, assuming that blocks correctly define their interface. Available nodes are declared in structs referred to as *nested data*, an example of which is shown

```
#[derive(NestedData)]
struct StrongArmTbNodes {
    vop: Node,
    von: Node,
    vinn: Node,
    vinp: Node,
    clk: Node,
    let vin = sim_output.vin;
    no field 'vin' on type
    'StrongArmTranTbNodesView'.
}
(b)
```

(a)

Listing 1: Example of a compile-time check when retrieving waveforms. The user-declared nested data is shown in (a) and an attempt to access an undeclared node is shown in (b).

in Listing 1a. When retrieving these nodes from simulation results, the output waveforms are stored in struct fields (e.g. `sim_output.vinn`). An attempt to access an invalid node will result in a compiler error, as shown in Listing 1b.

Additionally, Substrate generators can clearly define their parameter space, ports, and nested data using Rust’s expressive type system. The concrete interface types of Substrate generators imply the intended usage without requiring an understanding of the generator implementation. Code editors can also provide richer autocomplete suggestions for statically typed interfaces, allowing designers to peruse available parameters, ports, nodes, and associated documentation while writing their generators. Substrate’s core APIs benefit similarly from the type system.

## III. HIGHLY CONFIGURABLE GENERATORS

Substrate enables the design of highly configurable generators by providing a library called Automatic Translation of Logical Layout (ATOLL). ATOLL allows designers to manually place and route portions of the layout, while leaving other portions to algorithmic design. ATOLL also improves generator configurability by providing automated mechanisms to adjust layout aspect ratios and to rotate a layout by 90 degrees while respecting preferred routing directions.

### A. ATOLL

In a typical ATOLL design flow, users define the schematic and layout of *tiles* simultaneously by instantiating primitives or other tiles, describing their schematic connectivity, and placing the instances using basic alignment APIs. Once the user has placed all layout objects and routed any critical signals by hand, ATOLL then performs autorouting on the remaining signals and power strap placement to produce a final DRC- and LVS-clean layout. An overview of the ATOLL generator flow is shown in Figure 3.

ATOLL requires that metal routing lies on a global uniform routing grid. This routing grid is constructed from a *layer stack*, which specifies the allowed routing directions, preferred routing direction, line, space, and via spacing of each metal routing layer. Adjacent layers must have alternating preferred routing directions. The routing grid on layer  $L$  is formed by the intersections between tracks on layer  $L$  and tracks on  $L$ ’s

<sup>2</sup>More details on ScIR can be found in Substrate’s online documentation.

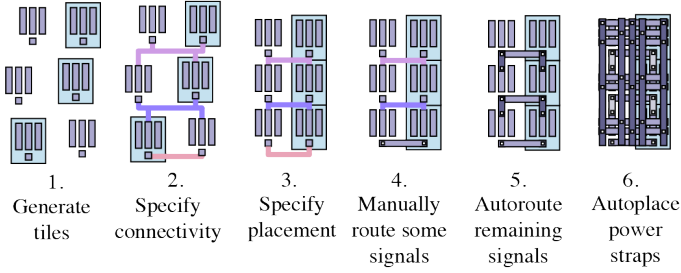


Fig. 3: Overview of the ATOLL generator flow.

*grid-defining layer*, which is the layer below  $L$  for all layers except the bottommost. For the bottommost routing layer, the grid-defining layer is the layer immediately above. The layer stack must be constructed in a manner that allows the router to produce DRC- and LVS-clean routes.

Tracks are placed at constant locations in physical space with respect to *routing grid units*, which are tiled starting from the origin. The width and height of a routing grid unit on layer  $L$  is determined by the least common multiple (LCM) of the pitches of vertical and horizontal metal layers equal to or below  $L$ , respectively. Since routing grid units describe the smallest unit at which all contained tracks repeat, they also play a role in defining the allowed placement of primitives and tiles in a way that keeps pins on the global routing grid. Constraining placements makes it simpler to hierarchically compose tiles. An example of a routing grid and allowed tile placements is described in Figure 4.

Once constructed and populated with tiles, the routing grid is used to store state for autorouting and strap placement.

### B. Configurable Aspect Ratios

ATOLL provides two mechanisms for achieving configurable layout aspect ratios: *tile resizing* for smaller circuits where folding would incur excessive overhead (e.g. a StrongARM) and *segment folding* for larger circuits that are composed of several repetitions of a tile (e.g. resistor banks, segmented drivers, etc.).

1) *Tile Resizing*: Tile resizing controls aspect ratio by choosing the shape of constituent devices. For example, in a process

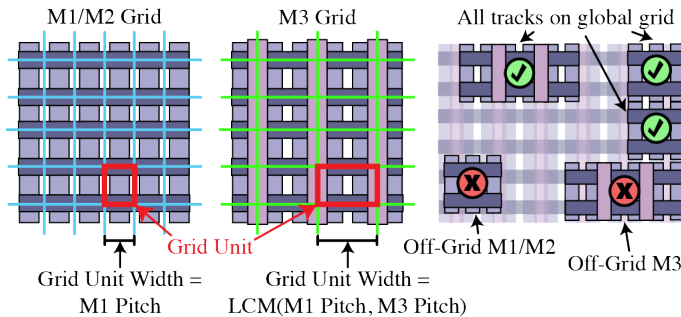


Fig. 4: Example of an ATOLL routing grid and allowed tile placements. The highlighted grid units have a height equal to the M2 pitch.

with horizontal gates, one might choose to use wider transistors with fewer fingers to decrease the height of their block while preserving the transistors' strength. However, resizing multiple transistors to achieve a certain aspect ratio requires some care, since decreasing the height of one transistor might increase the block width without decreasing the overall block height.

To formalize the problem, we assume that we are given an arrangement of tiles into rows. Each tile has a *strength* that we must keep constant. For example, a transistor tile's strength might be defined as the product of the number of fingers and the width of each finger. The optimization problem is to determine the size of each tile so that the overall block width  $W$  is minimized, subject to a maximum height constraint and given fixed strengths for each tile.

We assume that each constituent tile  $t$  can compute its minimum width  $W_{t,min}(H_{t,max})$  given a certain maximum height  $H_{t,max}$  and a constant strength  $S_t$ . We also assume that each tile has a minimum height increment  $\Delta h_t$ , which must be a multiple of its top layer's routing grid unit height. This allows us to discretize heights as multiples of  $\Delta h = \text{gcd}(\Delta h_{t1}, \dots, \Delta h_{tN_T})$ , where  $N_T$  is the number of tiles. We can then compute the minimum total width  $W_i(H_i)$  of row  $i$ , given its height  $H_i$ , by summing  $W_{t,min}(H_i)$  over tiles in the row. Assuming that the rows' centers are aligned horizontally and that each row can be sized independently, the maximum  $W_i$  sets the overall width of the block. These assumptions are acceptable since basic postprocessing, such as adding fillers, can be used to produce a DRC-clean design.

Let  $N_R$  be the number of rows. The optimization problem we would like to solve is:

$$\begin{aligned} \min_{H_0, \dots, H_{N_R-1}} \quad & \max_{0 \leq i < N_R} W_i(H_i) \\ \text{s.t.} \quad & \sum_{i=0}^{N_R-1} H_i \leq H_{max} \end{aligned}$$

An example of the problem formulation for a 5-transistor (5-T) operational transconductance amplifier (OTA) is provided in Figure 5.

We can solve this problem in polynomial time using dynamic programming. Define  $w[h, i]$  as the minimum width of rows 0 through  $i$  such that  $\sum_{j=0}^{i-1} H_j = h$ . Let  $T_i$  be the set of tiles in row  $i$ . We can define  $w[h, i]$  recursively as follows:

$$\begin{aligned} W_0(H_0) &= W_{t_0,min}(H_0) + W_{t_1,min}(H_0) \\ W_1(H_1) &= W_{t_2,min}(H_1) + W_{t_3,min}(H_1) \\ W_2(H_2) &= W_{t_4,min}(H_2) \\ \min_{H_0, H_1, H_2} \quad & \max(W_0(H_0), W_1(H_1), W_2(H_2)) \\ \text{s.t.} \quad & H_0 + H_1 + H_2 \leq H_{max} \end{aligned}$$

Fig. 5: Tile resizing problem formulation for a 5-T OTA.

$$w[h, 0] = \sum_{t \in T_0} W_{t, \min}(h)$$

$$w[h, i] = \min_{0 \leq H_i \leq h} \left[ \max \left( w[h - H_i, i - 1], \sum_{t \in T_i} W_{t, \min}(H_i) \right) \right]$$

The time complexity to solve for  $w[H_{max}, N_R - 1]$  and the metadata needed to reconstruct the required tile sizings is  $O(N_R H_{max}^2)$ , where  $H_{max}$  is specified in multiples of  $\Delta h$ . For analog blocks on the order of  $1000\Delta h$  wide and containing up to 100 rows, solving this problem is computationally feasible. The StrongARM instances described in Section IV, which span up to  $100\Delta h$  and contain 6 rows, each took around 170ms to generate (including tile resizing, autorouting, and strap placement) on an Intel Xeon-class CPU.

If the provided height constraint is not achievable, the algorithm will throw an error. While the algorithm described only covers a top-level height constraint, binary search could be used to implement a width constraint for tiles like MOS devices and resistors that satisfy the property that, for a fixed strength, minimum width decreases monotonically with increasing maximum height.

2) *Segment Folding*: The ATOLL segment folding procedure takes as input a unit element ATOLL tile, metadata about each pin of the tile, a number of rows  $R$ , and a number of columns  $C$ . The segment folder will construct an  $R$  by  $C$  grid of the input tile and attempt to connect the pins as described below.

Each pin to be routed by segment folding is associated with a routing layer and pin type, which can be one of *series*, *parallel*, or *escape*. Series pins are connected between adjacent tile instances, while parallel pins are connected between all tile instances. Escape pins remain unconnected and are simply routed to a specified edge of the tile grid. The tile must also specify a direction (horizontal or vertical) indicating how series and parallel pins are most naturally connected.

ATOLL requires that vertical tiles satisfy the following requirements (similar requirements apply to horizontal tiles):

- 1) Series pins must abut vertically.
- 2) Parallel pins must span the height of the tile and have one horizontal track available that spans the width of the tile and is on the layer immediately above the pin layer.
- 3) North/south (N/S) escape pins must be placed on a layer with horizontal tracks; east/west (E/W) pins on layers with vertical tracks.

Based on these inputs, the segment folding procedure analyzes the tile's layout, identifies free tracks, and allocates tracks to pins. To allocate tracks to pins, the ATOLL segment folder constructs a bipartite graph in which one set of vertices represents pins to be routed and the other set of vertices represents routing tracks. Each routing track that spans the width (for horizontal tracks) or height (for vertical tracks) of the tile corresponds to one vertex. Each parallel pin corresponds to one vertex, and each N/S escape pin maps to  $R$  vertices that represent the need for free vertical tracks. Similar logic applies to E/W escape pins. For each pin vertex, we draw an edge to a track vertex if the corresponding track is suitable for routing the

pin. The problem of allocating tracks then reduces to finding a maximal matching on this unweighted bipartite graph. We use the Hopcroft-Karp algorithm [11] to find such a maximal matching. If track allocation fails, we report the unmatched pins and instruct the user to modify the tile layout.

Finally, ATOLL forms the array and routes the pins appropriately. Figure 6 shows an example of the segment folding algorithm. Compared to naively placing each element and running a routing algorithm to connect all instances, segment folding produces more uniform and symmetric layouts, requires fewer computational resources, and is easier to debug if a routing solution is not found.

### C. 90° Rotation

In modern processes, changing a north/south-facing IP block to face east/west often requires a complete redesign, especially when trying to respect the preferred routing directions of all layers. ATOLL's partial layout automation significantly reduces the work required for rotating a layout by 90° while obeying preferred routing directions. Using ATOLL, rotating by 90° simply involves rearranging tiles programmatically and moving pin locations and layers. ATOLL will then complete the required routing to produce a DRC- and LVS-clean layout.

## IV. DESIGN EXAMPLES

To demonstrate ATOLL on real design examples, we showcase generators for two circuits: a StrongARM comparator [6], [7] and a programmable resistor bank.

The StrongARM comparator generator ensures symmetry of the comparator by first implementing an ATOLL generator for the comparator's half circuit. It then tiles the half circuit to form the full comparator, as shown in Figure 7. The generator also automatically adds taps to prevent latchup violations and places power straps densely over the comparator. The StrongARM generator supports aspect ratio configuration using ATOLL's tile resizing procedure. The generator also supports rotation by 90° by simply changing the tiling direction of the MOS devices and taps.

The code for generating the StrongARM is PDK-agnostic. The only PDK-specific information that is required to generate a layout is an ATOLL layer stack, a transistor tile, a tap tile,

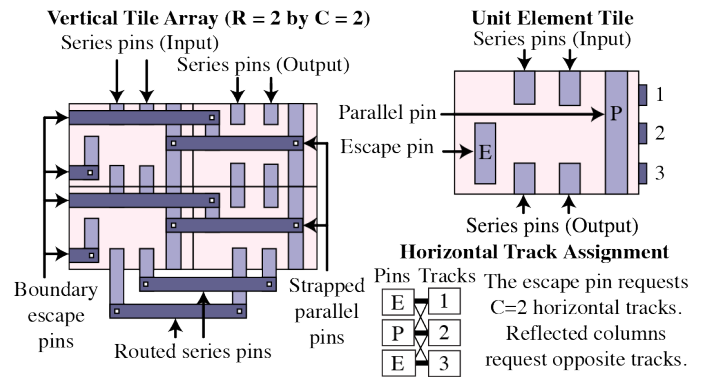


Fig. 6: Example of the segment folding algorithm. The selected matching of pins to horizontal tracks is shown with bold lines.

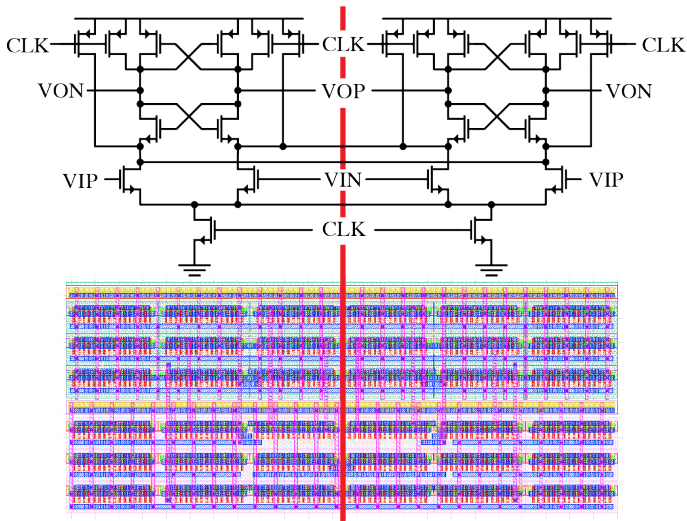


Fig. 7: The StrongARM generator creates two identical half circuits, reflects one horizontally, and connects them in parallel.

the required tap frequency, and a set of tasks to run on the finished layout (for example, drawing certain PDK-specific base layers). The remainder of the code is shared, allowing for some degree of process portability. The StrongARM generator was used to generate layouts in the Skywater 130nm and Intel 16nm FinFET technologies. DRC- and LVS-clean instances of various generator configurations are shown in Figure 8. All instances produced the correct comparison result in response to a 40mV differential input in post-extraction simulations across corners. In Skywater 130nm, StrongARM aspect ratios for a single sizing range from  $8\mu\text{m} \times 43\mu\text{m}$  to  $32\mu\text{m} \times 16\mu\text{m}$ . In Intel 16nm, StrongARM aspect ratios range from  $4\mu\text{m} \times 8\mu\text{m}$  to  $14\mu\text{m} \times 2.5\mu\text{m}$ . With the help of ATOLL’s abstractions, similar levels of configurability were achieved in two very different processes with relatively little process-specific code.

The programmable resistor bank generator, illustrated in Figure 9, demonstrates ATOLL’s segment folding algorithm. Each unit slice is a resistor in series with an NMOS transistor for enabling or disabling the slice. The generator produced DRC- and LVS-clean  $2 \times 20$  and  $4 \times 10$  instances in both Skywater 130nm and Intel 16nm. The tiled element was generated with standard ATOLL APIs, and the array was arranged and routed by the segment folder.

## V. CONCLUSION

Substrate is an open-source, statically typed framework for writing AMS generators. It provides flexible layers of abstraction, enabling users to leverage Substrate at varying stages of the design process. Substrate’s ATOLL library provides algorithms for rapid design space exploration and automatic routing, enabling faster design iterations. We demonstrate Substrate’s functionality through two circuit design examples and show that it can be used to quickly iterate on custom circuit layouts. We envision future work that will relax some of ATOLL’s restrictions (such as generalizing ATOLL’s routing grid to support non-uniform tracks), design new abstractions

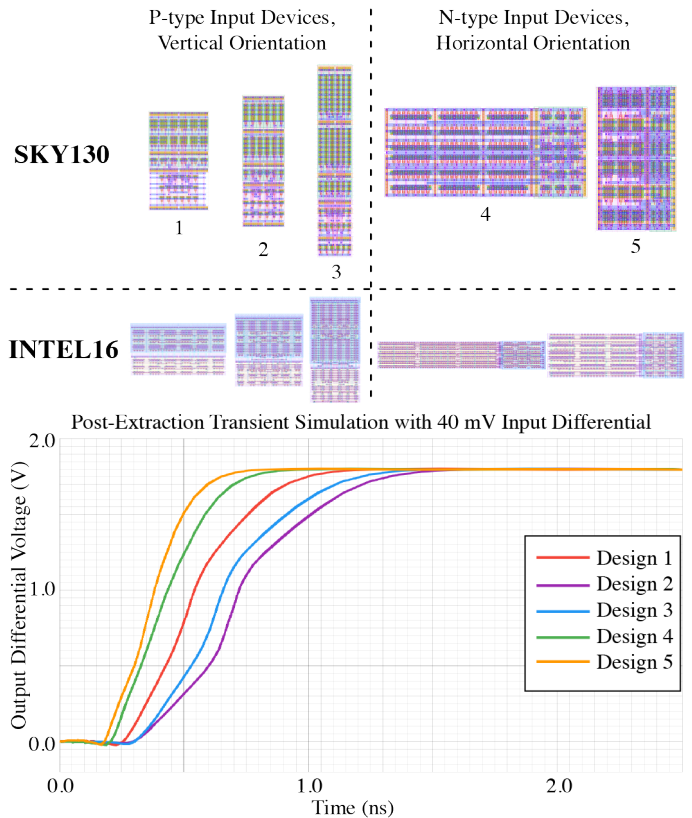


Fig. 8: Instances produced by the StrongARM generator with varying aspect ratio and input device type. The waveforms show the output transient behavior of the five SKY130 designs shown when supplied with a 40mV input differential voltage and a rising clock edge at  $t = 0$  (typical corner,  $V_{DD} = 1.8\text{V}$ ,  $T = 25^\circ\text{C}$ ). Intel 16nm simulation results and high resolution layouts have been intentionally omitted as per Intel’s request.

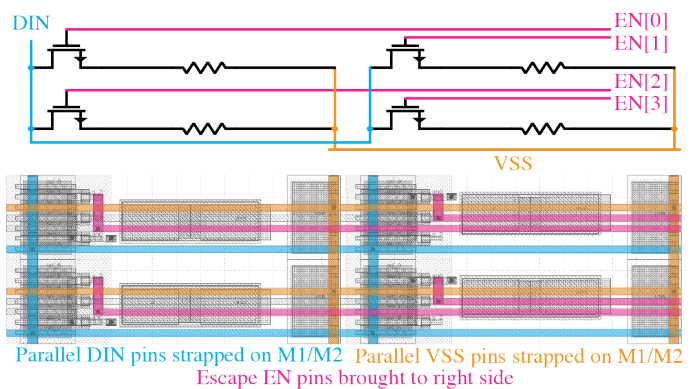


Fig. 9: A  $2 \times 2$  programmable resistor bank in Skywater 130nm generated using segment folding.

for representing circuits, and make programmatic layout more accessible to users who prefer the visual feedback provided by a graphical user interface. We believe that Substrate’s modular design provides a good foundation upon which further improvements can be layered.

## REFERENCES

- [1] H. Chen, M. Liu, B. Xu, K. Zhu, X. Tang, S. Li, Y. Lin, N. Sun, and D. Z. Pan, "MAGICAL: An open-source fully automated analog IC layout system from netlist to GDSII," *IEEE Design & Test*, vol. 38, no. 2, pp. 19–26, April 2021.
- [2] K. Kunal, M. Madhusudan, A. K. Sharma, W. Xu, S. M. Burns, R. Harjani, J. Hu, D. A. Kirkpatrick, and S. S. Sapatnekar, "Invited: ALIGN – open-source analog layout automation from the ground up," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–4.
- [3] F. Guo, B. Zhou, A. Biswas, P. Kwon, Z. Liu, K. Ho, V. Stojanović, and B. Nikolić, "BAG3++: An extensible generator framework for automated layout-aware AMS design," *IEEE Open Journal of Circuits and Systems*, vol. 6, pp. 181–191, 2025.
- [4] T. Shin, D. Lee, D. Kim, G. Sung, W. Shin, Y. Jo, H. Park, and J. Han, "LAYGO2: A custom layout generation engine based on dynamic templates and grids for advanced CMOS technologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 4402–4412, Dec 2023.
- [5] D. Fritchman, "An integrated circuit design framework for human, computer, and ML designers," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2023. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-275.html>
- [6] B. Razavi, "The StrongARM latch [a circuit for all seasons]," *IEEE Solid-State Circuits Magazine*, vol. 7, no. 2, pp. 12–17, Spring 2015.
- [7] W. C. Madden and W. J. Bowhill, "High input impedance, strobed CMOS differential sense amplifier," U.S. Patent US4 910 713A, 1990.
- [8] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–104. [Online]. Available: <https://doi.org/10.1145/2663171.2663188>
- [9] M. Costanzo, E. Rucci, M. Naiouf, and A. D. Giusti, "Performance vs programming effort between Rust and C on multicore architectures: Case study in N-body," in *2021 XLVII Latin American Computing Conference (CLEI)*, Oct 2021, pp. 1–10.
- [10] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu, "Towards understanding the runtime performance of Rust," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3551349.3559494>
- [11] J. E. Hopcroft and R. M. Karp, "An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973. [Online]. Available: <https://doi.org/10.1137/0202019>