Insights from Basilisk: Are Open-Source EDA Tools Ready for a Multi-Million-Gate, Linux-Booting RV64 SoC Design?

Philippe Sauter*[x], Thomas Benz*[x], Paul Scheffler*, Frank K. Gürkaynak*, Luca Benini*†
* *Integrated Systems Laboratory, ETH Zurich*, Switzerland
† *Department of Electrical, Electronic, and Information Engineering, University of Bologna*, Italy
{phsauter,tbenz,paulsc,kgf,lbenini}@iis.ee.ethz.ch

***Abstract*—Designing complex, multi-million-gate application-specific integrated circuits requires robust and mature electronic design automation (EDA) tools. We describe our efforts in enhancing the open-source Yosys+Openroad EDA flow to implement Basilisk, a fully open-source, Linux-booting RV64GC system-on-chip (SoC) design. We analyze the quality-of-results impact of our enhancements to synthesis tools, interfaces between EDA tools, logic optimization scripts, and a newly open-sourced library of optimized arithmetic macro-operators. We also introduce a streamlined physical design flow with an improved power grid and cell placement integration. Our Basilisk SoC design was taped out in IHP's open 130 nm technology. It achieves an operating frequency of 77 MHz (51 logic levels) under typical conditions, a 2.3× improvement compared to the baseline open-source EDA flow, while also reducing logic area by 1.6×. Furthermore, tool runtime was reduced by 2.5×, and peak RAM usage decreased by 2.9×. Through collaboration with EDA tool developers and domain experts, Basilisk establishes solid "proof of existence" for a fully open-source EDA flow used in designing a competitive multi-million-gate digital SoC.**

***Index Terms*—Open-source EDA, SoCs, Synthesis, RISC-V**

I. Introduction

In recent years, interest in open-source electronic design automation (EDA) has significantly increased in academia and industry. Academia benefits from collaboration free of non-disclosure agreements (NDAs), wide tool accessibility for students and researchers, transparent research on EDA tools, and free exchange of generated artifacts (e.g., netlists or layouts). The EDA industry would benefit from an increased influx of skilled fresh talents trained by universities on EDA algorithms and their implementation in realistic open-source tool frameworks. Meanwhile, the silicon design industry could benefit from reduced cost and, perhaps more importantly, sovereignty and a transparent chain of trust from register transfer level (RTL) descriptions to finished layouts. As a consequence, open-source EDA (OS EDA) tools have experienced a strong influx of users and developers, most prominently around the synthesis tool *Yosys* [1] and the place and route (P&R) tool *OpenROAD* [2].

One key challenge in developing a strong open-source ecosystem is to raise the maturity and robustness of OS EDA tools in handling large digital designs. In this direction, Benz et al. [3] recently presented and released Iguana, a Linux-capable RISC-V system-on-chip (SoC) design built on the configurable Cheshire SoC platform [4]. Iguana combines an RV64GC core called CVA6, a HyperRAM DRAM controller, and a rich set of peripherals, including VGA and USB 1.1, to complete a representative, real-world Linux-capable system; its RTL description is freely available [5].

Iguana's 2 MGE[1] implementation was first taped out in IHP's open process design kit (PDK) 130nm technology [6] with a commercial closed-source tool flow [7]. Benz et al. open-sourced their work-in-progress OS EDA flow (henceforth *Iguana flow*) [5], enabling others to build on their work.

In this work, we present Basilisk [5], the first end-to-end open-source Linux-capable SoC implemented in IHP's open 130 nm technology from RTL to tapeout. Starting from the Iguana flow, we make systematic improvements to the involved EDA tools, flow scripts, and constraints to achieve a quality of results (QoR) that is not only acceptable for tapeout, but significantly exceeds the open-source state of the art. Basilisk achieves an operating frequency of 77 MHz, a 2.3× improvement over Iguana, while reducing the logic area by 1.6× from 1.8 MGE to 1.1 MGE. We improve the runtime of synthesis by 2.5× from 5.4 h to 2.2 h and the peak RAM usage by 2.9× from 217 GB to 75 GB. During our tapeout, Basilisk's P&R completed with zero remaining design rule check (DRC) violations.

For the Basilisk SoC design, we update the Cheshire SoC platform to the newest version, adding new features, including a USB OHCI controller, to increase the capabilities and use cases. We do not simplify the original RTL description of the Cheshire SoC platform, which uses industry-grade SystemVerilog constructs, to avoid tool weaknesses, instead we focus on improving tools and the OS EDA flow. In the collaborative spirit of open source, we are collecting knowledge and leveraging existing efforts on cutting-edge algorithms and OS EDA tools.

We focus our efforts toward a QoR-optimized yet human-understandable tool flow; this human-in-the-loop (HITL) philosophy allows the designers to understand and better evaluate

[x] Both authors contributed equally to this research.

[1] Gate equivalent (GE) is a technology-independent figure of merit measuring circuit complexity; it represents the area of a two-input, minimum-strength NAND gate.

each implementation step. Finally, we provide the complete Basilisk SoC design and its critical, hard-to-implement parts, such as the floating-point unit (FPU) or CVA6's scoreboard, as challenging benchmarks for all steps of synthesis and P&R. These benchmarks allow the community to push, optimize, and further improve OS EDA tools and the corresponding flow scripts beyond the scope of this work.

In this work, we present the following contributions:

- An extensive study on the state-of-the-art open-source EDA flow using *Yosys* for logic synthesis and *OpenROAD* for P&R. Starting from the Iguana flow, we identify QoR improvements in the flow steps with a particular focus on the synthesis engine and scripts.
- The integration of a library of hand-optimized implementations of arithmetic macro-operators in the Yosys-based synthesis flow and its open-source release [8].
- A QoR-optimized HITL open silicon implementation flow from RTL to GDSII on an open-source foundry-supported PDK qualified for manufacturing in regularly scheduled runs for both full reticle and multi-project wafer (MPW) runs [9].
- Basilisk, the first end-to-end open-source Linux-capable application-specific integrated circuit (ASIC) implemented in IHP's open 130 nm node achieving 77 MHz with a logic area of 1.1 MGE.

II. Related Work

The OpenROAD [2] developers maintain an example flow called OpenROAD flow scripts (ORFS) [10]. ORFS integrates a variety of open PDKs (platforms) and a handful of small example designs. ORFS serves as a reference flow to get designers started, and as a benchmark to end-to-end verify OpenROAD on example designs. ORFS further provides automated design space exploration, facilitating the collection of key metrics using the given input parameters for each run. With ORFS, all design-technology options are implemented using the same set of flow files configured through environment variables set by project-specific Makefile fragments. While providing a single solution that fits all design-technology options, such an approach makes the flow internals harder for non-expert tool developers to modify and tune.

The OpenLANE [11] flow provides a turnkey RTL-to-GDSII flow using Yosys and OpenROAD, similar to ORFS. It is maintained by Efabless and used in their popular Caravel [12] MPW shuttles. To complete the flow, OpenLane uses *OpenRCX* to extract parasitics, *Magic* to stream out the final fabrication files and perform DRC checks, and *netgen* to complete LVS checks. A turnkey flow massively reduces the barrier of entry to designing ASICs by novice designers, but also makes it harder for expert designers to exercise the low-level tool control required to implement and optimize large and complex designs that push the capabilities of the OS EDA tools to the limit.

Qflow [13] is one of the earliest complete OS EDA RTL-to-GDSII toolchains using either *VTR* (*ODIN-II* and *ABC [14]*) or Yosys to synthesize a Verilog design and implement the backend using their own placement and routing engines. Magic is again used to complete and check the final layout files. The *Raven SoC* [13] is a working ASIC designed with Qflow. While able to implement a working SoC, this flow is currently limited to small designs under 100 kGE and larger technology nodes in the range of 0.18 to 0.5µm.

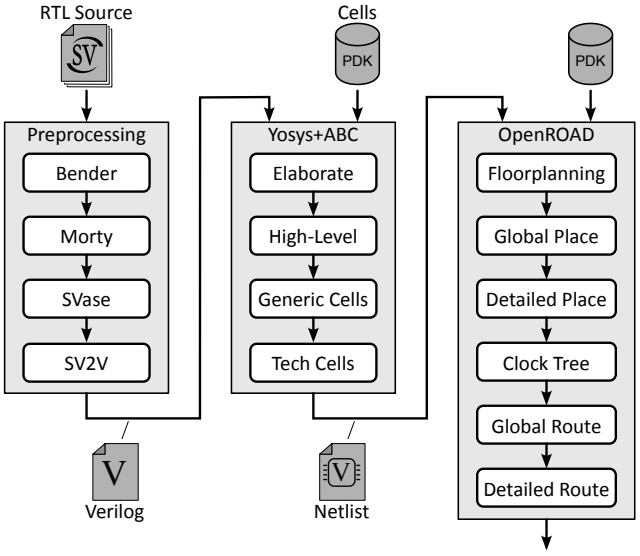


Fig. 1. Preprocessing, synthesis and P&R flow used for Iguana and Basilisk

ALLIANCE [15] is another full toolchain, synthesizing small designs written in VHDL and implementing them in a portable complementary metal-oxide semiconductor (CMOS) technology. The P&R flow of ALLIANCE has been replaced by *CORIOLIS* [15] to support larger designs on the order of 150 kGE. The developers recently started integrating Yosys into the ALLIANCE/CORIOLIS flow to synthesize designs. ALLIANCE and CORIOLIS use *symbolic layouts* to implement the ASIC backend, limiting their scaling to nodes above 130 nm [15]. Compared to regular standard-cell-based P&R, symbolic layout implements the design without any technology-specific design data.

iEDA [16] is a recent addition to the set of open P&R tools proven to implement layouts in 110 nm and 28 nm technology nodes. While being a complete P&R framework, iEDA does not yet have an active community around its ecosystem and its documentation is written in Chinese with only a partial translation into English available.

In addition to the above open RTL-to-GDSII flows, various standalone synthesis and P&R tools and frameworks exist. The *EPFL logic synthesis* [17] libraries allow optimizations of structural netlists; they provide a standalone tool called *mockturtle* built around these libraries, Yosys can be used to parse behavioral Verilog and convert the netlist to a supported format. *LSOracle* [18] uses EPFL logic synthesis libraries to implement optimization passes on and-inverter graphs (AIGs) and is available as a Yosys plugin. *GHDL* [19] is a capable open-source simulator for the VHDL language with experimental support for converting VHDL designs to structural

VHDL netlists; it is thus usable as a frontend to dedicated synthesis tools like Yosys. *LibrEDA* [20] is a recently developed open framework facilitating the development of P&R tools targeting research and education; it currently cannot produce implementable ASIC layouts.

Most users of these tools either target field-programmable gate arrays (FPGAs) or implement designs aimed at Efabless' Caravel MPW shuttles in SkyWater's 130 nm node. Caravel designs are limited to a maximum user-defined core area of 10 mm^2 and implemented designs are usually under 150 kGE with an average core density of 10 % to 30 % [21]–[23]. Caravel designs remain well within the established capabilities of the OS EDA tools and are unlikely to stress them. We aim to push OS EDA beyond its current limits towards supporting end-to-end open implementation of large multi-million-gate designs, focusing on developing the tools and flows to remove their current QoR and runtime bottlenecks.

III. Synthesis

Yosys [1] is a leading open-source synthesis engine widely used in OS EDA flows. At the time of writing, it offers limited support for SystemVerilog language constructs. With Cheshire's openly available RTL description written in industry-grade SystemVerilog, significant preprocessing work is needed to convert the design's RTL source to the simpler Verilog format that Yosys can parse. We use a chain of tools [3] to achieve this: *Bender* [24] collects and manages the dependencies from git repositories, *Morty* [25] combines all source files into a single compile context, *SVase* [26] propagates parameters and simplifies the most complex SystemVerilog constructs, and *SV2V* [27] converts the resulting simpler SystemVerilog code to behavioral Verilog.

At the first step of synthesis, Yosys parses this behavioral Verilog code into an abstract syntax tree (AST). Next, the syntax is elaborated into Yosys' main internal representation called RTL intermediate language (RTLIL) converting the behavioral code into a structural representation. The structural RTLIL first uses high-level constructs such as arithmetic operations or finite-state machines (FSMs) cells to represent the design. Yosys executes commands, called passes, on the RTLIL description to progressively optimize and transform this high-level structural description into a low-level representation using only generic standard cells (MUX, AND, NOR, ...). Finally, Yosys maps sequential elements to provided technology cells and calls ABC [14] on the combinational networks to optimize the logic representation further and map the generic gates to specified technology cells.

To improve QoR, we optimize three aspects of the synthesis chain. The QoR of the Iguana baseline flow and our contributions are summarized in Table I. The reported results show cumulative improvements from left to right.

TABLE I
Cumulative synthesis improvements from left to right; large impacts of each step are highlighted.

	Iguana [3]	**MUX**	**ABC**	**LAU**
Logic area	1.8 MGE	**1.4 MGE**	**1.1 MGE**	1.1 MGE
Timing	33 MHz	37 MHz	**71 MHz**	**77 MHz**
Logic levels [a]	182 LL	149 LL	**54 LL**	**51 LL**
Runtime [b]	5.4 h	**2.8 h**	2.2 h	2.2 h
Peak RAM [c]	217 GB	**105 GB**	76 GB	75 GB

[a] Number of logic gates in longest path [b] 2.5 GHz Xeon E5-2670

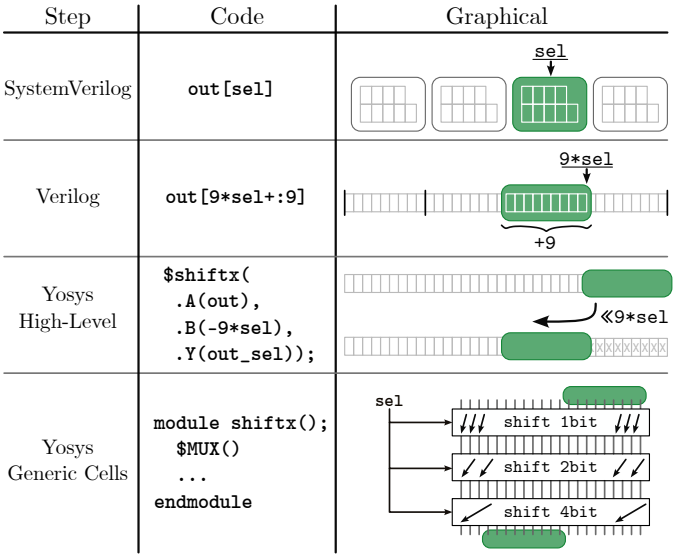


Fig. 2. Step-by-step preprocessing and synthesis of a part select operation

A. Part-Select Synthesis (MUX)

Yosys versions before our improvements[2](*<0.34*) elaborate indexed part-select operations to shift operations instead of more efficient *block-multiplexer trees*. A step-by-step overview of the previous *shift operation* inference is given in Figure 2.

First, *SV2V* preprocesses the SystemVerilog representation describing multi-dimensional arrays to a one-dimensional packed array with the size of the respective inner dimension as its stride. Then, Yosys elaborates the RTL description to the high-level RTLIL cell *$shiftx*, representing a shift where vacated positions are filled with don't-care bits. Shifting the selection/readout window is logically equivalent to directly selecting a group of bits. Yosys chooses *shift-by-N barrel shifters* to implement block selection prior to our fix. By not limiting the shift magnitude to a multiple of the stride, this approach produces redundant hardware capable of handling *invalid and unused* block selection scenarios. To support these unnecessary selections incurs significantly more area at a longer critical path than a simple *block multiplexer*.

This causes Yosys to implement a *MUX* tree many times larger and more complex than necessary. Constraining the magnitude of the shift in the low-level representation results in

[2]The authors highly acknowledge the support by the Yosys team and especially Martin Povišer with the Yosys implementation.

a significant amount of unused logic and many don't-care bits. The optimizations implemented in Yosys after this mapping process are not powerful enough to remove all unnecessary and redundant parts of the shifter, further propagating this complex logic through the flow. By the time the netlist is given to ABC, all remaining don't-care bits have been converted to logic zeros, blocking potential optimizations. This implementation of part selects using general barrel shifters thus significantly inflates design area and the number of logic levels in part-select paths. Due to the inflated logic description, peak RAM usage and synthesis time also increase by 2× (see Table I).

Instead of elaborating part selects to block multiplexers, we developed an additional optimization pass reducing all eligible shift operations in the design to block multiplexers. To achieve this, we detect constant strides in the control logic of a shift operation describing behavior equivalent to a part select. The pass then increases the stride to the next power-of-two value with the input and output padded accordingly. Increasing the stride to a power of two causes all lower-weighted barrel-shift stages to receive a well-defined constant input, allowing for trivial optimization using constant propagation. The additional padded bits are optimized away using existing Yosys passes.

An alternative implementation would be to directly infer block multiplexers from part select operations at the elaboration stage. We evaluate our optimization-pass-based solution against this more direct approach. In isolated benchmarks on part select operations, this direct solution produces the same hardware implementation as our high-level optimization pass. On an entire design, our optimization pass can optimize additional shift operations and thus produce a better overall design implementation than the simpler direct method.

As can be seen in Table I, our optimization pass reduces the logic area by 22 % and increases the operating frequency by 12 %. The computing resource utilization during synthesis is significantly reduced, the peak RAM usage is 52 % lower, and the synthesis runtime is 48 % lower.

B. ABC Scripts Overhaul (ABC)

In cooperation with the community of researchers and practitioners developing and using ABC[3], we overhaul the Yosys-internal ABC script for logic optimization and technology mapping. We leverage lazy man's synthesis (LMS), as proposed by Yang et al. [28], to improve the QoR while keeping the impact on runtime minimal. LMS uses a pre-generated library of optimal logic structures to replace so-called *cuts*, logic blocks of equal input-output behavior in the netlist. An overview of LMS is shown in Figure 3 for a simple example network (left). A library of records ① is generated in advance. The netlist is divided into cuts ② with the same number of inputs as the records. Then the library of records is probed ③ for structures implementing the same logic function. Finally, an optimal structure is selected and inserted back ④ into the netlist.

[3]The help and advice of Alan Mishchenko, Masahiro Fujita, Giovanni De Micheli, Andrea Costamagna, Alessandro Tempia Calvino is gratefully acknowledged.

Generating this library of optimal logic structures is a time-consuming process, but is highly justifiable as its generation is only required once. The cuts (records) are derived by running other optimization techniques on a large variety of different benchmarks and saving superior implementations of any logic function. Specifically, the record is created from six input cuts; e.g., each record can be mapped to one six-input lookup table (LUT). Records also contain the subsets of cuts with fewer inputs. To improve the library quality, we extend the 6-input record using cuts obtained from Basilisk through the **rec_add3** command available in ABC.

Once the record of cuts is loaded into ABC (**rec_start3**), the LMS optimizer can be called via **&if** −y −K NUM, where NUM is the number of inputs of the cuts stored in the record. The LMS flow presented in this work is closely based on the flow as presented by Yang et al.:

```
&st; &if −y −K 6; &syn2; &b;
&st; &dch −x; &if −K 4;
```

This iteration is applied multiple times; ten to twenty iterations are required to converge on a near-optimal solution. The script first performs structural hashing (**&st**) followed by LMS integrated into the mapper (**&if**) command. **&syn2** and **&b** rewrite and balance the depth of the graph, respectively. Structural choices are computed (**&dch**) and then considered when mapping to four-input LUTs. In the last step, it is possible to map to any type of LUT. We find that selecting four-input LUTs produces the best results for Basilisk. Finally, the ABCs technology mapper **&nf** maps the netlist to standard cells.

```
&st; &nf −D 6000;
```

The mapping to standard cells is executed strictly after completing all optimization iterations. If the mapping iteration is used before the optimization converges on a solution, the overall QoR is reduced. We experiment with adding structural choices before the technology mapper but see only very marginal QoR gains at a high runtime cost.

In addition to using LMS, we extend the **abc** command in Yosys to allow for more control over how the library and netlist are loaded into ABC. This exposes the **read_lib** command used to load the standard cell library, making it possible to define the additional parameters *-S slew -G gain*. These parameters are required for ABC to generate delays for each cell derived from the liberty timings. Previously, Yosys would not set these values, causing ABC to completely ignore the liberty timings and use unit delays instead. Using the properly loaded timing model increases the QoR obtained from **&nf**. Using a parametric sweep on a subset of Basilisk modules, we find *-S 20 -G 3* to be suitable parameters for IHPs 130 nm technology node.

The improved ABC script based on LMS, together with the correct delay models, improves the QoR substantially. The area is 21 % smaller and the critical path improves by 1.9× with significantly shorter runtime and reduced peak RAM usage, see Table I.

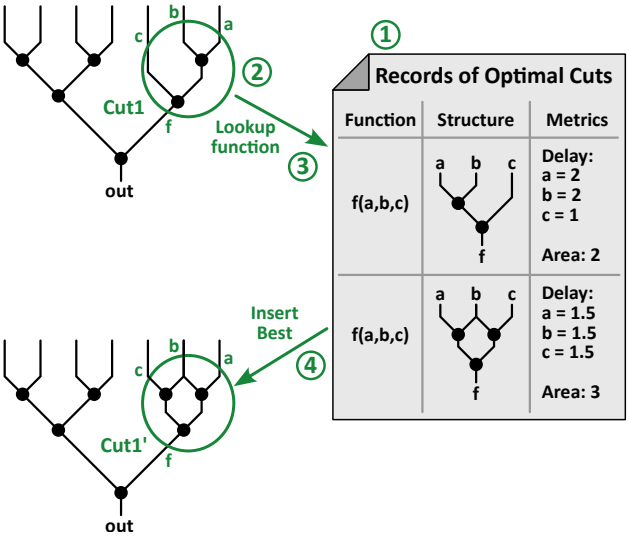


Fig. 3. A LMS example for a simple example network (left). Three input cuts are generated from the network and the three input records (right) are probed. The best implementation is taken and inserted into the graph.

C. Library of Arithmetic Units (LAU)

The critical path of the Basilisk design is in the datapath of its FPU; it traverses a 53×53 bit multiplier followed by two additions ($y = a \times b + c + d$), the latter with a 163-bit integer. We manually implement this datapath using modules from our library of arithmetic units, detail the necessary changes to Yosys, and present our approach's improved QoR.

Yosys maps arithmetic units to generic standard cells using built-in implementations of high-level blocks loaded from Verilog descriptions. The same fixed implementation is used for all instances of the same type without consideration of timing and area constraints. For example, all adders are implemented using a Brent-Kung parallel-prefix adder (PPA-BK) architecture without any alternatives available.

Some complex operations have specialized passes to infer and implement them. Most importantly, a generalized sum-of-products ($y = a\times b+c\times d+1\times e\ldots$) can be inferred from the current RTLIL using the Yosys pass **alumacc** which is later on implemented in the **maccmap** pass. Additionally, Yosys has a **booth** pass to implement multipliers. Since **maccmap** combines multipliers and adders into multiply-accumulate cells and **booth** transforms multipliers into generic standard cells, a designer needs to choose which approach they prefer; it is not possible to combine **booth** and **maccmap** without code changes to architectures implemented in C++ using Yosys' internal functions. Yosys' current way of implementing arithmetic operations makes it thus difficult to add more arithmetic operations and support additional architectural choices, targeting different scenarios.

Building on the work by R. Zimmermann [29], we build a library of arithmetic units to supersede Yosys' existing mappings. Our open-source library [8] has three speed grades optimized for different design points for each arithmetic operation. Yosys has an existing pass to match sub-circuits and wrap them into a custom cell called **extract**. We extend the **extract** pass to support variable width operators to match arbitrary arithmetic operations. We can improve area, timing, and power by extracting functionally equivalent sub-circuits and then implementing them with the custom mapping from our library of arithmetic units. To demonstrate the effectiveness of this approach, we create an optimized mapping for the critical data path, which is usable as part of the Yosys synthesis script, and benchmark it against the existing approaches in Yosys.

Figure 4 compares the architectures implemented by **booth**, **maccmap**, and our implementation using the library of arithmetic units. The theoretical unit-gate delays for each block are given, as they do not consider driving strength, wires, and other effects; they should be used to get an overview of the rough cost of each operation, not for accurate comparisons. **booth** (left) implements the multiplication using a radix-4 Booth encoder followed by a *Wallace tree* compressor. The compressor has a lower depth since the Booth encoding reduced the number of partial products from 54 to 28. The final carry-propagate adder (CPA) for the multiplication is implemented using a PPA-BK. The summation is performed using a carry-save adder (CSA) followed by the final PPA-BK. This architecture is constrained due to the separation of the partial product summation and the final summation into two discrete steps. The **maccmap** implementation (middle) improves on this by fusing the additions into the compressor tree implementing partial product summation. It does not employ Booth encoding to reduce the number of partial products, creating the deepest compressor tree. The final CPA is also a PPA-BK architecture.

Our implementation (right), built from modules in the library of arithmetic units, uses Booth-encoding and the faster Sklansky parallel-prefix adder (PPA-SK) CPA architecture to reduce the critical path.

The literature on the efficacy of radix-4 Booth encoding is split. Wolfgang et al. [30] formalize an analytical model and show a delay advantage in favor of Booth encoding. Shahzad et al. [31] implement a Booth-encoded and non-encoded multiplier using a reduced complexity Wallace tree and show an increase in delay using Booth encoding for most multiplier widths. We implement both variants and find the radix-4 Booth-Wallace architecture to produce superior QoR. Currently, all approaches map to generic standard cells and are passed to ABC, which may change the architecture. Still, passing an improved architecture to ABC is likely to produce a better final netlist.

Compared to the **booth** implementation used before, our approach reduces the critical path by 9 %, while it is reduced by 11 % compared to the alternative Yosys flow using **maccmap**, see Table I.

IV. Place & Route

We use *OpenROAD* [2] to implement Basilisk's synthesized netlist. As a reference, we re-run the Iguana flow; the final result has hundreds of DRC violations left after detailed routing. Using a commercial logic synthesizer, we can fix

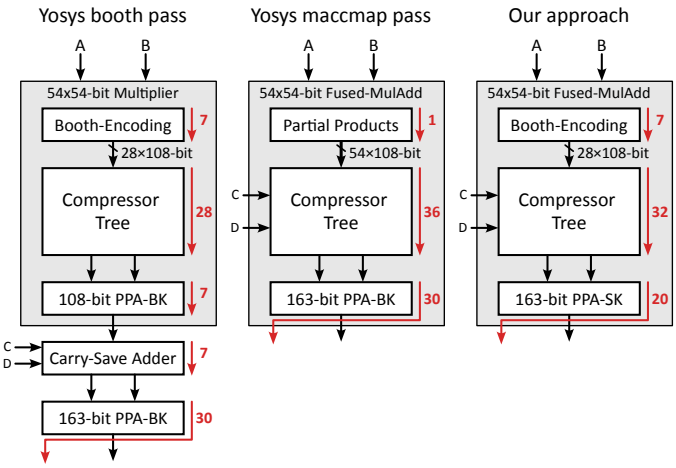


Fig. 4. Yosys elaborated and synthesized datapath (left, middle) and our optimized datapath (right) with unit delays.

problematic modules such as the boot ROM, CVA6's scoreboard in the issue stage, and the logic surrounding caches to enable proper comparisons. Analyzing Iguana's P&R, we identify improvement steps mainly in the *EDA tool flow* (how the individual components of OpenROAD are invoked) and the physical constraints of the ASIC.

Top metal power routing needs to pass through all metal layers to connect to the power rails of the standard cells. The created via stacks locally block routing resources. We improve the routability of the design by reducing the width of each power stripe. We increase their count on the top metal layer to compensate for the loss in current carrying capacity. Overall, this creates a more homogeneous pattern for P&R and directly eases local routing congestion underneath the stripes by reducing the number of blocked resources per stripe.

We change the SRAM configuration of the caches to reduce the number of SRAM macros from 36 to 24 by using larger SRAMs and adjusting the cache organization without changing the total size. We arrange the SRAM macros to maximize the uninterrupted core area and minimize routing channels. This reduces routing congestion as the macros block most metal layers, making routing resources above them sparse.

Currently, OpenROAD cannot restructure the netlist to improve routability or the timing of the physical implementation. It only inserts or removes buffers and resizes cells. This places additional pressure on the synthesizer as the netlist needs to be already routable. Yosys does not consider timing, wire fanout, or placement and will thus reduce any redundant logic, creating very high fanout nets. ABC, on the other hand, limits the maximum fanout and considers the standard cell library timings but otherwise does not consider placement and routing effects during optimization or mapping. These effects combined make it often difficult or impossible to implement netlists of complex designs without carefully configuring the design and the synthesis parameters.

Very dense modules with random routing patterns, such as boot ROMs, are a particular sources of issues. OpenROADs global placement engine does support a routability-driven mode, where dense routing regions are identified and the apparent size of connected cells is inflated. The routability-driven mode artificially increases the calculated cell density in global placement and lowers the actual cell density. This, in turn, lowers congestion by increasing the routing resources available per cell.

The default *hyper-parameters* of the global placement stage do not trigger the built-in *routability-driven* mode. The global placer will instead produce homogeneous placement density and the lowest possible wire length, not considering any local routing pattern, which may cause congestion. As OpenROAD currently only accepts global (as opposed to region- or instance-based) settings, we tune several *hyper-parameters* of the global placement engine to improve the placement of dense blocks. With proper tuning, we can get a routable design without any DRC violations. Specifically, we start routability-driven placement earlier by increasing *routability_check_overflow* and use a larger cell-area inflation ratio to more heavily penalize cells causing congestion by increasing both *routability_inflation_ratio_coef* and *routability_max_inflation_ratio*.

We observe OpenROAD's *global routing* to over-prioritize the lowest metal layers when planning the routing process. This increases the congestion close to the standard cells, making it difficult for detailed routing to fix remaining violations as it cannot find a clear path from the higher levels down to the lower metals in already congested regions. We reduce the *target metal utilization* of the lowest two metals by 30 % using *set_global_routing_layer_adjustment* to push global routing more onto the higher metal layers, increasing the flexibility for routing changes in the detailed router.

Figure 5 shows the die shot of our Iguana baseline flow (a) and the newest version of the Basilisk design (b). Basilisk has a more spread-out placement with less-distinct amoebae-shaped individual modules. The global nature of the routability-driven hyper-parameters makes it difficult to decrease local placement densities without affecting the overall placement solution. This increases the total wire length but not to the degree that it noticeably affects the critical path and reduces the operating frequency. The benefit of our improvements is also visible; Iguana has to use more routing on the top metal layers in dense regions, creating red-tinted vertical lines in the image. Requiring top metal routing indicates high local congestion in affected regions, as OpenROAD would otherwise prefer lower metal layers. A prominent example of this effect is to the right of (1) in Figure 5a.

V. Results

The cumulative QoR improvements of our Basilisk design over the Iguana flow are shown in Table I and Figure 6. We time the netlists obtained from Yosys in a commercial synthesis tool to ensure accurate timing reports using typical operating conditions.

In the AT-plot, Iguana denotes the baseline re-run of the Iguana flow presented by Benz et al. [3] with a logic area of

TABLE II
KEY METRICS OF BASILISK

Logic area (NAND2)	1.1 MGE
Logic levels [a]	51 LL
Technology	130 nm IHP
Operating frequency	77 MHz
SRAM memory	172 KiB (24 macros)
Chip / core area	39 mm / 21 mm
IO count	68

[a] Number of logic gates in longest path

1.8 MGE and a critical path of 30 ns. Our first contribution (*MUX*) to the synthesis of part-selects improves the logic area by 22 % and the critical path by 11 %. Building an optimized ABC script used in Yosys, utilizing LMS, and providing parameters necessary to create accurate delay models show the largest QoR improvements (*ABC*). The area is further reduced by 21 % (39 % compared to the Iguana flow) and the critical path is lowered by 2.1× to 14.1 ns. Finally, the new approach to mapping high-level Yosys cells using our library of arithmetic units (*LAU*) can further improve the critical path by 9 % to 13 ns.

The tight control over the optimization flow, gained through our optimized ABC scripts, and the multitude of choices when selecting architectural implementations of individual arithmetic operations allows our implementations to span a large area of the AT-plot. This enables developers more flexibility when considering area-timing tradeoffs during the design and implementation phases.

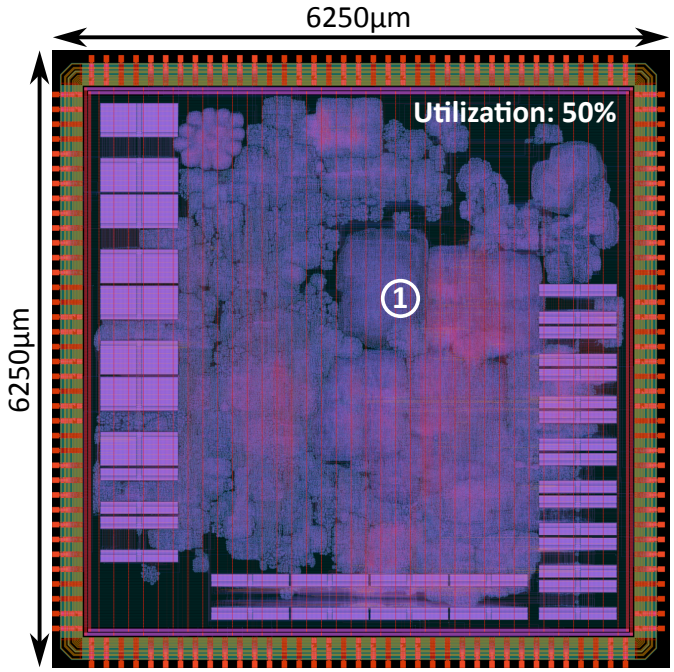
The Pareto-optimal point improves the timing by 2.3× and reduces the area by 1.6× compared to the Iguana flow. Using a timing-optimized variation of our ABC script, we can further decrease the critical path to 10.4 ns at the cost of an increased logic area.

Our contributions take a significant step towards closing the QoR gap of open-source flows compared to their commercial counterparts. Still, commercial EDA tools have a clear edge on multi-million-gate designs like Basilisk. They achieve this with timing-aware synthesis, tighter integration of elaboration and optimizations, deeper libraries of pre-optimized blocks, and a stronger focus on backend-aware synthesis. The best-achieved logic area and critical path length are within 50 % of a commercial synthesis flow of Basilisk.

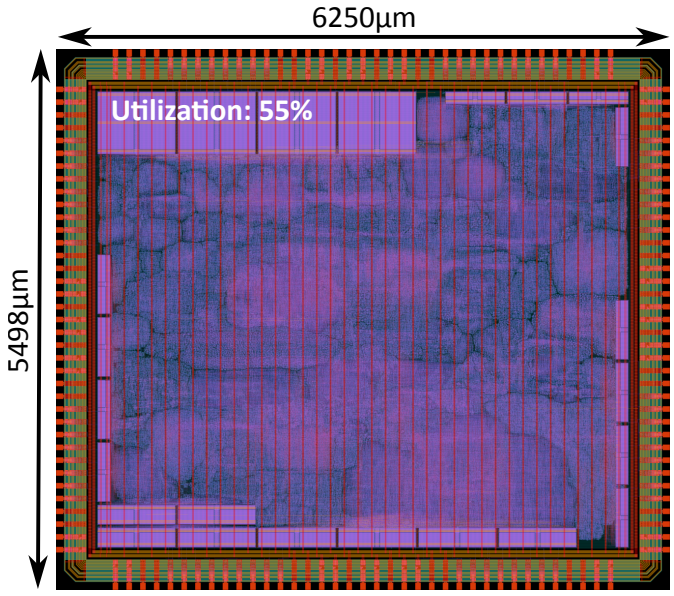
VI. CONCLUSION AND OUTLOOK

In this work, we evaluate the state-of-the-art open-source EDA flows and contribute significant improvements to the tools and flow, resulting in a flow viable for multi-million-gate SoC design tapeouts. In the process, we open-sourced a library of hand-optimized architectures of common arithmetic operations, which is compatible with Yosys.

Synthesizing and implementing Basilisk in IHP's open 130 nm technology, we optimize the design's clock frequency by 2.3× from 33 MHz to 77 MHz compared to the Iguana flow, while reducing the logic area from 1.8 MGE to 1.1 MGE and decreasing the synthesis runtime from 5.4 h to 2.2 h.



(a)



(b)

Fig. 5. Layout files produced by running the original Iguana flow (a) and of Basilisk (b).

Additionally, a timing-optimized synthesis script achieves a maximum frequency of 97 MHz.

Our improved P&R scripts and constraints successfully implement the Basilisk SoC design with zero DRC violations with an increased core utilization of 55 % compared to 50 % using the Iguana flow. All our improvements allowed us to tape out Basilisk successfully.

We contribute to improving open-source EDA tools by reporting and fixing tool issues, releasing our optimized *flow*

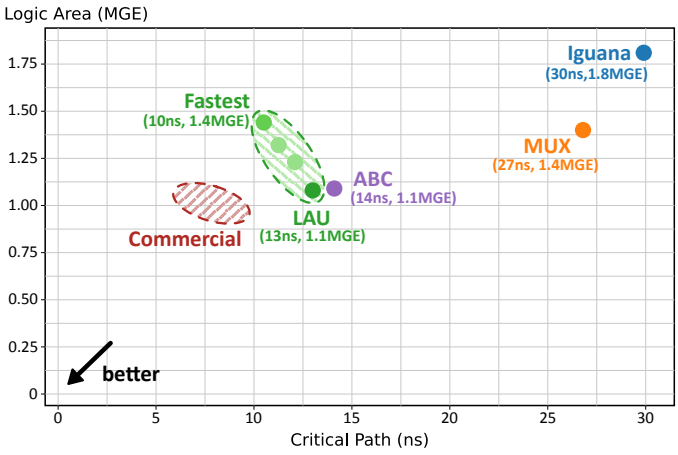


Fig. 6. Synthesis and P&R results of Iguana, our contributions and a commercial reference flow.

scripts, and implementing Basilisk, which can be used as a benchmark of a real-world multi-million-gate SoC design across the entire OS EDA toolchain.

To fully utilize the library of arithmetic units, more effort is required to adjust the existing Yosys ABC9 [32] system for standard cell designs. Leveraging the white-box approach, the implementation of arithmetic units could be controlled from Yosys without losing proper delay calculation and logic optimization of any connected circuits in ABC. Further, the timing reported from ABC can be utilized to change the speed grade and implementation during synthesis, allowing for the automatic selection of the optimal architectures for each individual arithmetic operator.

Work towards an integrated timing- and constraints-driven synthesis and P&R flow is required in a long-term effort. This would enable more aggressive timing, area, and routability optimizations where needed. This requires a high degree of coordination and long-term planning between OS EDA developers. Suitable standardized formats could facilitate this while maintaining the independence of tools and maintaining the ability to swap or replace individual tools in the flow, easing maintainability and integration of new research.

Acknowledgement

We thank Alan Mishchenko, Masahiro Fujita, Giovanni De Micheli, Andrea Costamagna, Alessandro Tempia Calvino, Osama Hammad Abdel Reheem, Matt Liberty, Martin Povišer, the Yosys team, Beat Muheim, and Zerun Jiang, for their valuable contributions to the research project. We further thank all contributors to the OS EDA tools.

We are deeply grateful to IHP for their generous support and providing us with the opportunity for an open-source tapeout of this scale.

This work was supported in part through the TRISTAN (101095947) project that received funding from the HORIZON CHIPS-JU programme.

References

[1] C. W. et al., "Yosys - a free verilog synthesis suite," 2013.
[2] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. Chhabria, D. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça *et al.*, "OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain," *Proc. GOMACTECH*, 2019.
[3] T. Benz, P. Scheffler, J. Schönleber, and L. Benini, "Iguana: An end-to-end open-source Linux-capable RISC-V SoC in 130nm CMOS," 2023.
[4] A. Ottaviano, T. Benz, P. Scheffler, and L. Benini, "Cheshire: A lightweight, Linux-Capable RISC-V host platform for domain-specific accelerator plug-in," *IEEE TCAS II*, 2023.
[5] PULP Platform contributors, "Iguana," https://github.com/pulp-platform/iguana, 2024.
[6] IHP-GmbH, "IHP Open Source PDK," https://github.com/IHP-GmbH/IHP-Open-PDK, 2022.
[7] T. Benz, P. Scheffler, J. Schoenleber, and P. Sauter, "Industry-grade SystemVerilog IPs and the open flow: How we synthesized Iguana," available: https://wiki.f-si.org/index.php?title=Industry-Grade_SystemVerilog_IPs_And_The_Open_Flow:_How_We_Synthesized_Iguana.
[8] PULP Platform contributors, "ELAU," https://github.com/pulp-platform/elau, 2024.
[9] IHP - Leibniz Institute for High Performance Microelectronics, "Mpw schedule 2024 & 2025 and price information 2024," https://www.ihp-microelectronics.com/services/research-and-prototyping-service/mpw-prototyping-service/schedule-price-list.
[10] The-OpenROAD-Project, "OpenROAD Flow," https://github.com/pulp-platform/OpenROAD-flow-scripts, 2024.
[11] A. Ghazy and M. Shalan, "Openlane: The open-source digital ASIC implementation flow," in *WOSET*, 2020.
[12] Efabless, "Caravel harness," https://github.com/efabless/caravel, 2022.
[13] R. T. Edwards, M. Shalan, and M. Kassem, "Real silicon using open-source EDA," *IEEE Design & Test*, 2021.
[14] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *CAV*, 2010.
[15] J.-P. Chaput, M.-M. Louërat, R. Chotin-Avot, and A. Satin, "RISC-V design using free open source software," in *the RISC-V Week*, 2019.
[16] Xingquan Li et al., "iEDA: An open-source infrastructure of EDA," in *ASP-DAC)*, 2024.
[17] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage *et al.*, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.
[18] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "LSOracle: A logic synthesis framework driven by artificial intelligence," in *IEEE ICCAD*, 2019.
[19] G. Biagetti, L. Falaschetti, P. Crippa, M. Alessandrini, and C. Turchetti, "Open-source HW/SW co-simulation using QEMU and GHDL for VHDL-based SoC design," *Electronics*, vol. 12, no. 18, 2023.
[20] Thomas Kramer, "LibrEDA," https://libreda.org, 2024.
[21] Z. R. Khan, W. ul Hasan, Z. Rafique, A. A. Ansari, and S. R. Naqvi, "GHAZI: An open-source ASIC implementation of RISC-V based SoC," *Authorea Preprints*, 2023.
[22] Y. Zhu, G. Yin, X. Wang, Q. Yang, Z. Luan, Y. Zhang, M. Wang, P. Guo, X. Wan, S. Hu *et al.*, "GreenRio: A modern RISCV microprocessor completely designed with a open-source EDA flow," in *WOSET*, 2022.
[23] M. H. Khan, A. A. Jalal, S. Ahmed, A. A. Ansari, and S. R. Naqvi, "IBTIDA: Fully open-source ASIC implementation of Chisel-generated system on a chip," *Authorea Preprints*, 2023.
[24] PULP Platform contributors, "bender," https://github.com/pulp-platform/bender, 2022.
[25] ——, "morty," https://github.com/pulp-platform/morty, 2022.
[26] ——, "SVase," https://github.com/pulp-platform/svase, 2023.
[27] Zachary Snow, "sv2v," https://github.com/zachjs/sv2v, 2020.
[28] W. Y. et al., "Lazy man's logic synthesis," in *IEEE/ACM ICCAD*, 2012.
[29] R. Zimmermann, "VHDL library of arithmetic units," in *Proc. First Int. Forum on Design Languages (FDL'98)*. Citeseer, 1998.
[30] W. Paul and P.-M. Seidel, "To Booth or not to Booth," *Integration*, vol. 32, no. 1, 2002.
[31] S. Asif and Y. Kong, "Performance analysis of wallace and radix-4 booth-wallace multipliers," in *2015 Electronic System Level Synthesis Conference (ESLsyn)*, 2015, pp. 17–22.
[32] B. L. Barzen, A. Reais-Parsi, E. Hung, M. Kang, A. Mishchenko, J. W. Greene, and J. Wawrzynek, "Narrowing the synthesis gap: Academic FPGA synthesis is catching up with the industry," in *DATE*, 2023.