Basilisk: An End-to-End Open-Source Linux-Capable RISC-V SoC in 130nm CMOS

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

***Abstract*—Open-source hardware (OSHW) is rapidly gaining traction in academia and industry. The availability of open RTL descriptions, EDA tools, and even PDKs enables a fully auditable supply chain for end-to-end (RTL to layout) open-source silicon, significantly strengthening security and transparency. Despite promising developments, existing OSHW efforts have so far fallen short of producing end-to-end open-source SoCs at the complexity and performance level needed to run a general-purpose OS. We present Basilisk, the first end-to-end open-source, Linux-capable RISC-V SoC taped out in IHP's open 130 nm technology. Basilisk features a 64-bit RISC-V core, a fully digital HyperRAM DRAM controller, and a rich set of IO peripherals including USB 1.1 and VGA. To tape out Basilisk, we create a reusable tool pipeline to convert its industry-grade SystemVerilog description to Verilog. We optimized logic synthesis in the open source Yosys synthesis tool, obtaining an increase in Basilisk's peak clock speed by 2.3× to 77 MHz and reducing its cell area by 1.6× to 1.1 MGE while also reducing synthesis runtime and RAM usage. We further optimized place and route in OpenROAD, enabling convergence to zero DRC violations while increasing core area utilization by 10 % and reducing die area by 12 %.**

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

I. Introduction

In recent years, open-source hardware (OSHW) has gained attention from industry and academia alike. The increased momentum on open source hardware has led to the creation of open register transfer level (RTL) descriptions for intellectual property (IP) blocks [1]–[3], electronic design automation (EDA) tools [4], [5], and even process design kits (PDKs) [6]. OSHW opens up a traditionally closed design process, curtailing or even eliminating IP and tool licensing costs and enabling open research and collaboration without non-disclosure agreements (NDAs).

Most notably, OSHW enables a *transparent* and *verifiable* hardware supply chain from RTL description to layout. By combining open IPs, EDA tools, and PDKs into an end-to-end open-source flow, open hardware designers can empower third parties to not only reproduce the final layout, but to fully audit its design process and verify its logic equivalence to the original RTL description. Instead of having to trust closed-source tools and IPs from commercial providers, the functionality and security of OSHW can be verified independently across levels of abstraction.

Strengthening security through auditable OSHW is not a new idea; the *OpenTitan* project [7] provides open-source root of trust (RoT) IPs and recently taped out a first chip with RoT functionality. However, while some OpenTitan test chips were designed using partially open tools, none of their existing designs are end-to-end open-source. Moreover, even if applications can trust an open RoT with isolating security-critical data and operations, the remaining system-on-chip (SoC) is usually closed-source and cannot be trusted by programmers or end users. Some partially open-source processors [3], [8] and SoCs [2], [9] are available, but to date, there is no existing end-to-end open-source Linux-capable SoC.

In this work, we present Basilisk[1], the first end-to-end open-source, Linux-capable RISC-V SoC taped out in IHP's open 130 nm technology. Basilisk is based on the configurable Cheshire [9] SoC platform and combines an RV64GC core with a fully digital HyperRAM DRAM controller and a rich set of peripherals, including VGA and USB 1.1, to complete a useful real-world Linux system. Like its hardware, Basilisk's boot code and firmware are completely open; from reset to Linux init, all executed code is open-source and auditable.

To tape out Basilisk with competitive quality of results (QoR), we vastly improve the state-of-the-art open EDA flow using *Yosys* [5] and *OpenROAD* [4]. First, we create a reusable tool pipeline simplifying Basilisk's industry-grade SystemVerilog (SV) RTL description to Yosys-supported Verilog by introducing our parameter-resolving SV pre-elaborator *SVase*. Then, we optimize Yosys' logic synthesis by improving multiplexer handling, integrating lazy man's synthesis (LMS) [10], and mapping arithmetic units to a library of preoptimized designs. Finally, we improve the OpenROAD place and route (P&R) tool flow by designing a routing-friendly power grid and tuning global hyperparameters. Overall, our flow optimizations improve Basilisk's peak clock frequency from 33 MHz to 77 MHz, reduce logic area from 1.8 MGE to 1.1 MGE, increase core utilization from 50 % to 55 %, and reduce synthesis runtime and peak RAM usage by 2.5× and 2.9×, respectively.

To summarize, our contributions are as follows:

- We present Basilisk's open-source, extensible architecture featuring a Linux-capable 64-bit RISC-V core, a HyperRAM DRAM controller, a hierarchical interconnect, and a rich set of IO peripherals including USB 1.1 and VGA.
- We create a reusable open-source tool pipeline simplifying Basilisk's industry-grade SV RTL description to a single Yosys-readable Verilog file by leveraging our parameter-resolving SV pre-elaborator *SVase*.
- We optimize Yosys' logic synthesis QoR by improving multiplexer handling, integrating LMS, and leveraging a library of optimized arithmetic units, increasing Basilisk's

§ Authors contributed equally to this research.

[1]https://github.com/pulp-platform/cheshire-ihp130-o

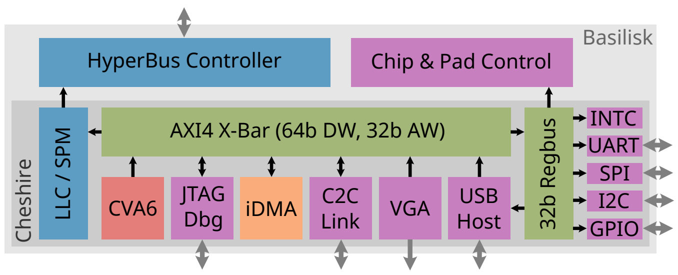

Fig. 1. Top-level block diagram of Basilisk.

clock speed by 2.3× and reducing its cell area by 1.6× while also reducing synthesis runtime and RAM usage.

- We improve the OpenROAD P&R tool flow by designing a routing-friendly power grid and tuning global hyperparameters, achieving zero design rule violations, improving core utilization by 10 %, and reducing die area by 12 %.

II. Architecture

Basilisk is built around *OpenHWGroup*'s energy-efficient RV64GC CVA6 [8] processor. It is based on our open-source, silicon-proven *Cheshire* SoC platform [9] designed to provide a minimal, highly configurable, autonomously booting 64-bit RISC-V host for Linux-capable systems.

Fig. 1 shows Basilisk's top-level architecture. It includes all hardware components necessary to boot and run Linux without support from an external host, such as RISC-V-compliant interrupt controllers, various standard IO interfaces, and a fully digital DRAM interface. To balance performance and design complexity, Basilisk uses a two-stage interconnect: request initiators and high-throughput endpoints attach to a fully connected 64-bit Advanced eXtensible Interface 4 (AXI4) [11] crossbar, while low-throughput peripherals and configuration interfaces without burst support are accessed through a lightweight Regbus [12] demultiplexer.

Basilisk features a fully digital HyperRAM DRAM controller supporting two chips and a transfer speeds of up to 154 MB/s. The controller is connected to the AXI4 crossbar through a 4-way, 64 KiB last-level cache (LLC); each way can dynamically be configured as scratchpad memory to provide on-chip SRAM when needed. CVA6 is configured with 2-way, 16 KiB L1 instruction and data caches.

Basilisk provides a rich set of peripherals. In addition to I2C, quad SPI, and UART for serial communication, it includes a four-port USB 1.1 (OHCI) host controller and a VGA controller for video output. Unlike later protocol revisions, USB 1.1 can be implemented using only digital logic and regular-speed IOs, keeping Basilisk's design highly accessible. Each USB port is multiplexed with GPIOs, providing a software-controlled IO bus up to 8 bit wide. A JTAG test access point connected to a RISC-V debug module enables live debugging of the CVA6 core and full memory bus access. A fully-digital, double-data-rate, duplex 77 Mbit/s chip-to-chip (C2C) link serializing the AXI4 protocol allows two Basilisk chips to communicate through direct interconnect accesses. A high-efficiency, asynchronous DMA engine [13] capable of 2D transfers relieves CVA6 of data movement tasks. All of Basilisk's peripherals are designed to be Linux-compatible, with many already having working drivers for our version of CVA6 Linux (kernel version 5.10.7, updates ongoing).

Basilisk's boot ROM enables autonomous boot from a GPT-formatted SD card, SPI NOR flash, or I2C EEPROM. It loads a small binary of up to 48 KiB into its internal scratchpad, which in turn may load a firmware (e.g. OpenSBI) and a full-fledged bootloader (e.g. U-Boot) into DRAM. Alternatively, code may also be preloaded through JTAG, UART, or the C2C link. In our upstream setup, all software run from SoC reset to Linux userspace is completely open-source and auditable, including the boot ROM, firmware, and our Linux modifications.

Basilisk is highly extensible and reconfigurable by design. Adding interconnect ports and interrupts for new IPs, removing existing blocks, or even using multiple CVA6 cores is simply a matter of reparameterization. We hope this will allow other designers to build on and extend Basilisk with minimal effort.

III. Implementation Flow

We implement Basilisk in IHP's open 130 nm technology using *Yosys* and *OpenROAD*. To this end, we implement a reusable tool pipeline simplifying Basilisk's industry-grade SV RTL description to Verilog supported by Yosys (Section III-A), enhance synthesis to significantly improve QoR while reducing runtime and memory footprint (Section III-B), and optimize the reference backend flow to improve QoR and minimize design rule violations (Section III-C).

A. RTL Description Preprocessing

Yosys currently cannot read in synthesizable SV, supporting only Verilog-2005 and a few selected SV constructs. Existing open-source solutions to synthesize SV designs in Yosys include the SV-to-Verilog conversion tool *SV2V* [14] and the third-party *Synlig* [15] frontend. Unfortunately, these solutions cannot handle Basilisk's industry-grade RTL description; both fail to correctly resolve hierarchically propagated design parameters, which are essential to keeping complex, parametric designs manageable without resorting to code generation.

Prior to our work, the only open-source tool correctly resolving Basilisk's parameterization was *Slang* [16], a library providing full SV elaboration and the best SV language support of all open-source tools evaluated by ChipsAlliance's SV test suite [17]. However, as a library, Slang only provides elaborated designs as in-memory data structures with no existing solution to pass them on to Yosys for synthesis.

To close this flow gap, we created *SVase* [18], a source-to-source SV pre-elaborator leveraging Slang. SVase rewrites all parameter expressions as literals, unrolls all generate constructs, and uniquifies every used module parameterization. The resulting SV RTL description has *no* unresolved parameters or dependencies between instances and is simple enough to correctly be translated to Verilog by SV2V.

For simplicity, SVase assumes that all input SV sources are collected in a single file, which we automate using our existing source management tool *Bender* [19] and our SV source pickler *Morty* [20]. Fig. 2 shows the resulting end-to-end tool flow converting Basilisk's multi-file SV RTL description to a single Verilog file readable by Yosys. We emphasize that this flow is not specific to Basilisk or Yosys; it can simplify *any* industry-grade SV design to a single Verilog file for use with *any* tool,

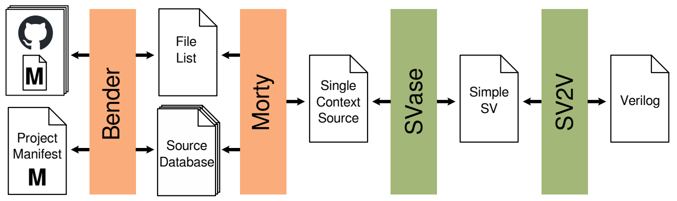

Fig. 2. Basilisk's SV-to-Verilog RTL description preprocessing flow.

including simulators with limited SV support. Furthermore, the full flow takes less than *two minutes* to run on Basilisk, which is two orders of magnitude less than Yosys synthesis after our optimizations (2.2 h) and comparable to elaboration steps in commercial, SV-capable simulators and synthesis tools.

B. Synthesis Enhancements

Before our work, synthesizing Basilisk with Yosys resulted in inacceptable QoR and inflated runtime and memory usage (see Section III-C). We present three key, design-independent improvements to Yosys' logic synthesis that drastically improve design QoR and Yosys' resource footprint.

Part-select synthesis: Yosys versions prior to our improvements (<0.34) use generic shift operations (`$shiftx`) to represent all indexed part-select operations instead of more efficient *block-multiplexer trees* where possible. Thus, for any part select, a generic barrel shifter supporting an arbitrary shift amount is inferred at elaboration. This broad generalization significantly inflates area and increases runtime and peak memory usage; it is especially problematic in conjunction with SV2V, which translates all selections into arrays of packed data as part selects. Later logic optimization stages are unable to simplify these shifters to the desired multiplexer trees, strongly impacting QoR. Instead of changing the representation of part-selects, we develop a new optimization pass that identifies shift operations with constant strides, pads the implied blocks to a power of two, and thus enables existing optimizations to remove unnecessary logic. Compared to a solution inferring block multiplexers, our approach achieves the same results on part selects, but overall superior results as other shift operations may also benefit from this optimization.

Lazy man's synthesis: In cooperation with logic synthesis researchers and ABC developers, we overhaul the ABC script, leveraging Yang et al.'s work on LMS [10] to improve QoR at the cost of minimal additional runtime. Near-optimal implementations of six-input, one-output logic functions are precomputed using other optimizers in ABC. The functions and implementations, together with their characteristics, are stored in a look-up table. This process is time- and resource-intensive, but only needs to be performed once, and the resulting table can be re-used across different designs. During synthesis, the netlist is divided into blocks with six inputs and one output and the table is probed for the best fitting implementation, replacing each block with the corresponding structure.

Library of Arithmetic Units: Yosys currently uses a suboptimal approach to implement fundamental arithmetic units. Addition operations infer one globally selected adder architecture, impeding a balanced area-speed tradeoff. Multiplication operations are implemented using Booth's algorithm. More complex operations are implemented using the basic operation mappings; in the case of the multiply-accumulate (MAC) operation, a Booth multiplier followed by an adder is inferred. A more efficient solution is to integrate adders into the *CSA* tree of preceding multipliers, creating fused multiply-add (FMA) units. We use our library of optimized arithmetic unit implementations to map a timing-critical MAC operation as an FMA unit, shortening our critical path. We further replace the default adder implementation in Yosys with a selection of optimized adder architectures. A solution to automatically infer FMAs and other fused operations and implement them from our library is in the works.

C. Place and Route Optimizations

We use *OpenRoad* [4] to place and route Basilisk's synthesized netlist. We mainly identify possible improvements in the *EDA tool flow* (how the individual components of OpenROAD are invoked) and the *physical constraints* of the application-specific integrated circuit (ASIC). We improve the routability of the design by redesigning the power grid; we reduce the width and increase the count of the power stripes on the top metal layer to ease routing congestion underneath the stripes. Very dense modules with random routing patterns, such as the boot ROM, were a particular source of issues. As OpenRoad currently only accepts global (as opposed to region- or instance-based) settings, we tune several *hyper-parameters* of the routability-driven global placement engine to improve the placement of dense blocks and get a routable design without design rule check (DRC) violations.

IV. Results

We present the final QoR of Basilisk and quantify our synthesis and P&R optimizations in relation to a baseline Yosys-and-OpenROAD flow without improvements.

A. Synthesis

Figure 3 summarizes the cumulative QoR effects of our synthesis enhancements in an area-time (AT) plot. We time our netlists in a commercial tool to ensure accurate results under typical conditions (1.2 V, 25 °C). We compare our work to a *baseline* Yosys flow without our optimizations using the default, speed-optimized ABC script from the OpenROAD flow scripts; it yields a logic area of 1.8 MGE and a critical path length of 30 ns (33 MHz).

Our first optimization improving the synthesis of part-selects (*MUX*) reduces logic area by 22 % and the critical path length by 11 %. Building an optimized ABC script that leverages LMS (*ABC*) yields the largest QoR improvement, further reducing area by 21 % and shortening the critical path by another 2.1×. Finally, using our optimized library of arithmetic units (*LAU*) further shortens the critical path by 9 % to 13 ns (77 MHz). Together, our Yosys optimizations reduce synthesis time from 5.4 h to 2.2 h (2.5×) and peak synthesis RAM usage from 217 GB to 75 GB (2.9×).

Our parametric ABC script and the span of choices in our library of arithmetic units can provide designers with flexible control over area-timing tradeoffs, allowing us to generate a pareto-frontier of multiple designs as shown. While the minimal-area design we ultimately chose (1.1 MGE, 13 ns) improves timing by 2.3× and reduces the area by 1.6×, tuning our ABC

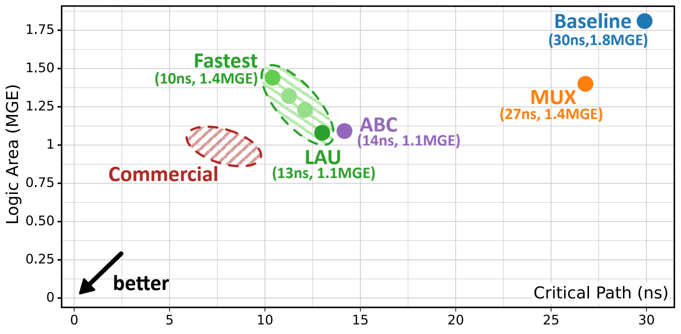

Fig. 3. Area-time plot summarizing the incremental QoR benefits of our Yosys synthesis optimizations and comparing them to commercial QoR.

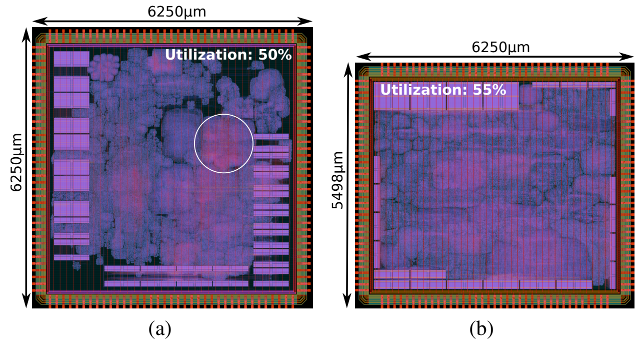

Fig. 4. Basilisk layouts produced by the baseline (a) and optimized flow (b). The white circle highlights excessive top metal routing (red) in the baseline.

script for timing further reduces the critical path to 10.4 ns (97 MHz) at the cost of notably increased area.

Despite our significant QoR improvements, commercial synthesis tools still have a clear edge on multi-million-gate designs like Basilisk. They achieve their superior QoR through timing-aware synthesis, tighter integration of elaboration and optimizations, larger libraries of pre-optimized blocks, and a stronger emphasis on physically aware synthesis. Nevertheless, our optimizations take a significant step toward closing the QoR gap between open-source synthesis and commercial flows; our best logic area and critical path length are both within 50 % of what a commercial synthesis tool achieves.

B. Place and Route

Figure 4 shows the final Basilisk layout from OpenROAD without and with our synthesis and P&R QoR optimizations. In the former (baseline) case, we re-synthesize some problematic modules including the boot ROM, the CVA6 issue stage, and the L1 data cache with a commercial tool to avoid an unmanageable number of DRC violations primarily caused by inefficient part-select handling in synthesis (See Section III-B).

Our improvements to the physical implementation flow increase core area utilization from 50 % to 55 % (+10 %) while reducing local peak routing resource utilization, enabling OpenROAD to converge to *zero* DRC violations. When using only Yosys for baseline synthesis, we reduce the peak routing resource utilization, found in the boot ROM, from an unfeasible 210 % to 100 %. Tuning OpenROAD's hyperparameters reduces routing congestion both on the global and local scale, obviating most signal routing on the unsuited top metal layers and resulting in a less clustered floorplan. Overall, we reduce Basilisk's die area from 39 mm^2 to 34 mm^2 (-12 %).

V. Conclusion and Outlook

We present Basilisk, the first end-to-end open-source, Linux-capable RISC-V SoC taped out in IHP's open 130 nm technology. Basilisk features a 64-bit RISC-V core, a fully digital HyperRAM DRAM controller, and a rich set of IO peripherals including USB 1.1 and VGA. To tape out Basilisk, we create a reusable tool pipeline converting its industry-grade SystemVerilog description to Yosys-readable Verilog. We also significantly optimize Yosys' synthesis QoR, improving Basilisk's clock speed by 2.3× to 77 MHz and reducing cell area by 1.6× to 1.1 MGE while also reducing synthesis runtime and peak RAM usage by 2.5× and 2.9×, respectively. Finally, our OpenROAD P&R optimizations enable convergence to zero DRC violations, improve core area utilization by 10 %, and reduce die area by 12 %. In future work, we hope to enhance Basilisk with open-source RoT IPs to also provide robust cryptographic security and a verified boot chain.

Acknowledgement

We thank A. Mishchenko, M. Fujita, G. Micheli, A. Costamagna, A. T. Calvino, O. Hammad, M. Liberty, M. Povišer, the Yosys team, B. Muheim, and Z. Jiang, for their valuable contributions to the research project. We further thank all contributors to the open-source EDA tools. We are deeply grateful to IHP for their generous support and providing us with the opportunity for an open-source tapeout at this scale.

References

[1] PULP Platform, "PULP Platform," https://pulp-platform.org/, 2024.
[2] D. Petrisko et al., "Blackparrot: An agile open-source risc-v multicore for accelerator socs," *IEEE Micro*, 2020.
[3] C. Chen et al., "Xuantie-910: Innovating cloud and edge computing by risc-v," in *2020 IEEE HCS*, 2020.
[4] T. Ajayi et al., "OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain," *Proc. GOMACTECH*, 2019.
[5] C. Wolf et al., "Yosys - a free verilog synthesis suite," 2013.
[6] IHP-GmbH, "IHP Open Source PDK," https://github.com/IHP-GmbH/IHP-Open-PDK, 2022.
[7] lowRISC contributors, "OpenTitan," https://github.com/lowRISC/opentitan, 2019.
[8] F. Zaruba et al., "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE VLSI*, 2019.
[9] A. Ottaviano et al., "Cheshire: A lightweight, Linux-Capable RISC-V host platform for domain-specific accelerator plug-in," *IEEE TCAS II*, 2023.
[10] W. Yang et al., "Lazy man's logic synthesis," in *IEEE/ACM ICCAD*, 2012.
[11] A. Kurth et al., "An open-source platform for high-performance non-coherent on-chip communication," *IEEE TC*, 2022.
[12] PULP Platform contributors, "Generic register interface," https://github.com/pulp-platform/register_interface, 2022.
[13] T. Benz et al., "A high-performance, energy-efficient modular dma engine architecture," *IEEE TC*, 2024.
[14] Zachary Snow, "sv2v," https://github.com/zachjs/sv2v, 2020.
[15] CHIPS Alliance, "Synlig," https://github.com/chipsalliance/synlig, 2021.
[16] Mike Popoloski, "slang - SystemVerilog Language Services," https://github.com/MikePopoloski/slang, 2019.
[17] CHIPS Alliance, "SystemVerilog Tester," https://github.com/chipsalliance/sv-tests, 2021.
[18] PULP Platform contributors, "SVase," https://github.com/pulp-platform/svase, 2023.
[19] ——, "bender," https://github.com/pulp-platform/bender, 2022.
[20] ——, "morty," https://github.com/pulp-platform/morty, 2022.