RS5V-SoC: Architecture and ASIC Design of a Vector-Enabled RISC-V SoC

Fernando Gehm Moraes, Antônio dos Santos, Lucas Damo, Nathan Cidal, Sophia Mendes da Silveira, Vinícius Mibielli, Vitor Zanini, Angelo Dal Zotto, Willian Nunes, Rafael Follmann Faccenda

School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
fernando.moraes@pucrs.br, {antonio.s001, lucas.damo, nathan.cidal, sophia.mendes, vinicius.mibielli, vitor.balbinot, angelo.dalzotto, willian.nunes, rafael.faccenda}@edu.pucrs.br

***Abstract*— Systems-on-Chip (SoC) using the RISC-V Instruction Set Architecture (ISA) play a central role in the development of embedded systems by enabling flexibility, extensibility, and low power consumption. This SoC, submitted to tape-out in the context of the SBMicro-APCI 2025 call, presents the RS5V-SoC, which integrates custom peripherals, RISC-V ISA extensions for cryptographic (AES) and vector processing, and support for the Zephyr RTOS. The vector processing extension accelerates data-parallel kernels, such as those commonly used in machine learning applications. This paper has two main objectives. First, it presents the RS5V-SoC architecture and a design-space exploration of the vector unit. We demonstrate that for embedded workloads, a 4-lane, 128-bit vector unit with a 32-bit bus offers the best area-performance trade-off. Second, it details the ASIC implementation of the RS5V-SoC in a 65 nm CMOS technology, describing each step of the design flow, from floorplanning to metal fill, the final step prior to tape-out. The design is validated through an FPGA-to-ASIC flow and targets embedded applications, such as Internet of Things devices and machine learning. The processor is publicly available at `https://github.com/gaph-pucrs/RS5`.**

***Index Terms*— RISC-V, System-on-Chip (SoC), Vector Processing, ASIC Design, Design-Space Exploration, Embedded Systems**

I Introduction

Systems-on-Chip (SoC) featuring RISC-V processors and comprehensive software support are essential for developing embedded systems that target both Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA) technologies. Open-source RISC-V microcontrollers have been increasingly adopted in applications such as Internet of Things (IoT) devices, wearables, and real-time control systems. These platforms typically integrate one or a few 32-bit RISC-V cores, on-chip Static Random Access Memory (SRAM), and standard peripherals, with an emphasis on low power consumption and ease of development. They commonly support bare-metal or real-time operating systems.

For example, Chiu et al. [1] focus on balancing low latency and low power consumption for real-time edge computing by integrating a RISC-V CVA6 processor with custom hardware accelerators in the ESP SoC platform [2]. Similarly, Azad et al. [3] proposed a RISC-V–based SoC optimized for cryptographic processing in edge-computing scenarios involving homomorphic encryption. In virtualized environments, Sá et al. [4] introduced optimizations to the RISC-V CVA6 core by integrating RISC-V hypervisor extensions. Expanding configurability and peripheral integration, Machetti et al. [5] developed X-HEEP, a configurable and low-power platform based on CORE-V processors such as the CV32E20 and CV32E40X/P. This platform introduced an Extensible Accelerator Interface to enable the integration of specialized accelerators. Wang et al. [6] presented an open-source framework optimized for energy-efficient neural network inference on IoT microcontrollers.

In both industrial and academic contexts, platforms such as the SiFive FE310 [7], the OpenHW CORE-V MCU [8], Shakti RIMO [9], and OpenTitan Earl Grey [10] provide reference designs. The SiFive FE310 is a low-cost, open-source microcontroller that emphasizes compatibility with Real-Time Operating System (RTOS) software. The CORE-V MCU facilitates peripheral integration and targets IoT and edge machine-learning applications. In contrast, Shakti RIMO offers a 64-bit open-source processor designed for industrial and automotive applications. OpenTitan emphasizes security by integrating cryptographic accelerators and secure firmware execution capabilities into embedded systems.

In contrast to the scalar microcontroller-oriented platforms discussed above, hardware acceleration for kernels that can be parallelized through vector processing has traditionally targeted complex processors designed for high-performance computing [11, 12]. In this context, the RISC-V Vector (RVV) extension [13] enhances the RISC-V Instruction Set Architecture (ISA) [14] with scalable vector-processing capabilities, enabling efficient data-level parallelism for workloads such as scientific computing, machine learning, multimedia processing, and other performance-intensive applications.

For instance, SPEED [15] is an RVV processor optimized for multi-precision deep neural network inference, adding customized instructions, hardware optimizations, and specialized dataflow mappings to maximize efficiency. Johns et al. [16] move toward lower-complexity designs by introducing a RISC-V vector processor for microcontrollers and embedded systems, based on an in-order, single-stage RV32E core augmented with a tightly integrated Vector Processing Unit (VPU) that supports a subset of the vector extension. Nevertheless, the literature remains limited in proposals integrating the RVV into RISC-V processors for resource-constrained embedded systems, highlighting a gap that this work addresses.

To address this gap, this work has two main **objectives**. **First**, it presents the `RS5V-SoC` architecture (Section II) and its vector unit (Section III). Section III describes the modifications to the original vector unit, including a parameterizable lane count and bus width, as well as the corre-

Digital Object Identifier 10.29292/jics.v21i2.1196

sponding design-space exploration conducted to identify the hardware configuration that achieves the best area–speedup trade-off.

The **second** objective is to detail the ASIC implementation of the RS5V-SoC in a 65 nm CMOS technology, describing each step of the design flow, from floorplanning to metal fill, the final step prior to tape-out (Section IV). This section aims to help designers develop integrated circuits by highlighting the importance of starting the design process with floorplanning rather than with logical synthesis, as is commonly done.

Finally, Section V concludes the paper and outlines directions for future work.

II RS5V-SoC Architecture

Figure 1 presents the architecture of the RS5V-SoC. All SoC modules were developed in-house, without third-party components. The RS5V-SoC is derived from the RS5-SoC [17] and is extended with a vector processing unit [18] targeting Single Instruction, Multiple Data (SIMD) workloads.

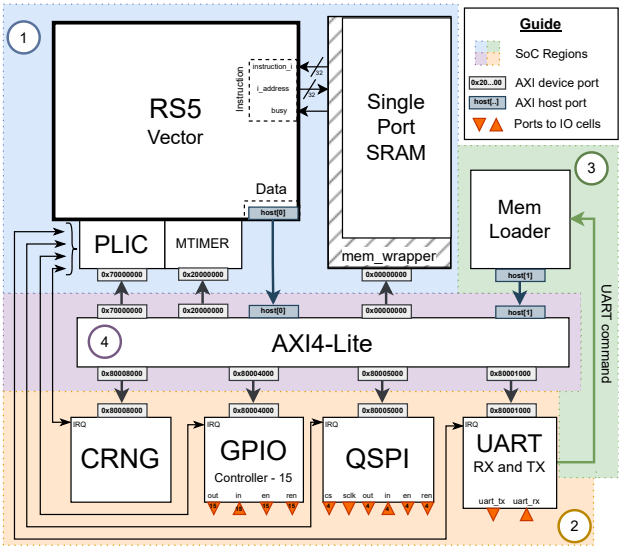


Fig. 1 RS5V-SoC architecture.

There are four groups of modules in Figure 1:

1. **Processing subsystem:** This block includes the RS5 processor [19] extended with a vector unit, a 32 kB single-port static RAM, a machine timer (MTIMER), and a Platform-Level Interrupt Controller (PLIC).
2. **Peripherals:** This group comprises Chaotic Random Number Generator (CRNG), General Purpose Input/Output (GPIO), Quad Serial Peripheral Interface (QSPI), and Universal Asynchronous Receiver/Transmitter (UART) modules. The QSPI interface enables connection to an external flash memory, extending the processor address space.
3. **MemLoader:** This module receives commands from an external host via the UART and initializes the on-chip memory with object code.
4. **Multi-master AXI4-Lite interconnect** [20]: Both the processor and the MemLoader act as bus masters, with the MemLoader assigned higher priority.

The **MemLoader** operates as a command interpreter for instructions received from the UART interface. It loads binary code from a host computer into memory, and upon completion, releases the processor to execute the program. Future versions of the SoC will integrate a bootloader to load code directly from flash memory via the QSPI interface. As an AXI4-Lite master, the MemLoader also provides debugging capabilities, allowing read and write access to both memory and peripheral address spaces, enabling memory dumps and manipulation of Memory-Mapped Registers (MMRs).

Our SoC integrates a **CRNG** module [21], which derives pseudorandom sequences from deterministic chaotic dynamics. The method adopts key properties of chaos, most notably strong sensitivity to initial conditions and rapid state divergence. Thus, successive iterations of a simple nonlinear map yield highly unpredictable outputs. The generated numbers are used within the SoC to support cryptographic operations.

The RS5V-SoC is described in SystemVerilog and uses synthesis directives to support both ASIC and FPGA implementations. The main differences between these targets concern the register file and memory implementation. FPGA versions use on-chip Block Random Access Memory (BRAM), whereas ASIC implementations rely on memory generators in the technology Process Design Kit (PDK).

II.A RS5 RISC-V Processing Subsystem

The RS5 [19] is a 32-bit integer RISC-V processor that supports Machine and User privilege levels. The RS5 core, in the RS5V-SoC, implements the following Instruction Set Extensions (ISE):

- **M**: integer multiplication and division;
- **A**: atomic instructions performing read-modify-write operations that enable locking primitives;
- **C**: compressed instructions;
- **Zicntr** and **Zihpm**: hardware performance timers and counters;
- **Zicond**: integer conditional operations;
- **Zicsr**: Control and Status Register (CSR) instructions;
- **Zcb**: simple code-size-saving instructions;
- **Zkne**: instructions for accelerating the encryption and key-schedule functions of the Advanced Encryption Standard (AES) block cipher;
- **Zve32x**: vector operations (Section III).

Following the naming conventions employed by the RISC-V community, the implemented set of extensions corresponds to the RV32IMACZicntr_Zicond_Zicsr_Zihpm_Zcb_Zkne_Zve32x ISA. These ISEs enable multitasking operating systems and provide hardware-accelerated cryptographic and vector support.

Figure 2 presents the RS5 organization, where green rectangles separate the four pipeline stages: instruction fetch (**IF**), instruction decode (**ID**), execution unit (**XU**), and load/store unit (**LS**).

RS5 uses PLIC, an interrupt management approach defined by SiFive [22], to handle global and external interrupts

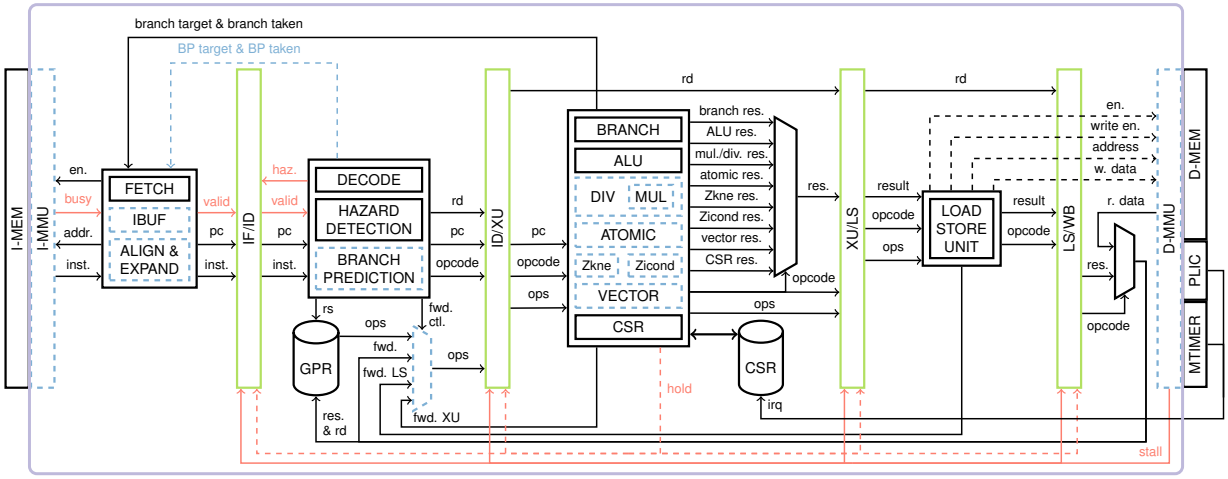


Fig. 2: RS5 Organization. The purple line delimits the core. Components in dashed blue lines are optional. Green components are temporal barriers. Signals that control the pipeline stall or generate bubbles are marked in red, and the hold signal is dashed because it occurs only when *M*, *A*, *V*, or *Zkne* ISEs are present. Adapted from: [19].

generated by peripheral devices. PLIC is configurable and provides an interface to MMRs.

The RS5 supports timer interrupts generated by the MTIMER and external interrupts managed by the PLIC. The MTIMER is a 64-bit counter accessible through MMRs and mapped to the timer registers defined by the Zicntr extension. The Zephyr RTOS (Section II.B) uses the MTIMER to generate timer interrupts and to control scheduler time slices. All peripherals are AXI4-Lite slaves controlled through MMRs. Peripheral events are signaled to the processor via PLIC, and the operating system drivers handle register access and interrupt servicing.

The RS5 core was validated using the RISCOF (RISC-V Architectural Test Framework [23]), which provides a standardized, automated methodology for verifying compliance with the RISC-V ISA. Passing the RISCOF test suite demonstrates that the processor correctly implements the architectural specification, including instruction semantics, privilege behavior, and corner cases defined by the standard. This level of compliance ensures software portability, compatibility with existing toolchains, and correct execution of operating systems and applications. Figure 3 presents the initial part of the RISCOF log, showing that the architectural test passed for the included ISEs.

II.B Software

The software layer includes device drivers, linker scripts, a Hardware Abstraction Layer (HAL), and libraries for accessing hardware resources. The platform supports two software execution models: bare-metal applications and the Zephyr RTOS [24].

Bare-Metal. Bare-metal programming includes the essential components required to program the SoC. A custom library, `libsoc`, provides the infrastructure for bare-metal software development. The `libsoc` library includes drivers for peripherals, as well as support for interrupt handling and for the AES and Secure Hash Algorithm (SHA) cryptographic algorithms, adapted from the TinyCrypt library [25]. The bare-metal environment uses `newlib-nano` as the standard C library, providing support for functions such as IO and memory management.

Report generated on **2026-01-06** 18:09 GMT by riscof v

Environment

Riscof Version	1.25.3
Riscv-arch-test Version/Commit Id	3.9.1
DUT	RS5
Reference	sail c simulator
ISA	RV32IMACUZicond_Zicntr_Zicsr_Zihpm_Zca_Zcb_Zkne
User Spec Version	2.2
Privilege Spec Version	1.10

Yaml

Show all details / Hide all details

Name
./RS5/riscof/extensions_work/extensions_checked.yaml *(show details)*
./RS5/riscof/extensions_work/platform_checked.yaml *(show details)*

Please visit YAML specifications for more information.

Summary

✓ 112Passed, ✓ 0Failed

Fig. 3 Validation of the RS5 with RISCOF, including the ISEs.

Zephyr RTOS. Zephyr [24] is an open-source, portable, and energy-efficient real-time operating system. It provides a comprehensive development environment, including tools for HAL management, debugging, and testing. Zephyr includes drivers compatible with the SoC modules, such as the PLIC and GPIO. The Zephyr build system also supports several RISC-V extensions used in the `RS5V-SoC`.

II.C FPGA Prototyping

Figure 4 shows the debugging environment of the SoC prototyped on an Artix FPGA. The setup includes a Nexys A7 board [26] connected to a host computer and a Raspberry Pi that transmits data to the FPGA, emulating a generic peripheral that acquires sensor samples. Communication with the host is performed using the SoC-CLI tool. The debugging interface displays five concurrent threads, with operating system messages shown in the lower-left corner.

Validation of both hardware and software components on an FPGA prior to ASIC fabrication enables early detection of functional errors, integration issues, and corner cases. In addition, it supports refining the SoC and software under re-

Fig. 4 RS5V-SoC debug environment.

alistic execution conditions, including interactions with external devices. This FPGA-to-ASIC approach is important for reducing technical risks and increasing confidence in the ASIC implementation.

III VXU - Vector Processing Unit

RISC-V entered the domain of SIMD and parallel computing with the introduction of the RVV. The ratified specification, released in 2021, defines a scalable and vector-length-agnostic architecture. This open-source RISC-V extension implements an ISA for data-parallel processing, supporting variable vector lengths, unit-stride, constant-stride, and indexed-stride memory accesses, as well as integer, floating-point, and predicate instructions. Hwacha [27] served as a reference architecture for RVV.

The RS5 RVV targets the $Zve32x$ subset of the RVV specification. Even minimal RVV configurations require an extensive instruction set [18]. In this work, more than 60 vector instructions were implemented, nearly twice the 36 instructions of the RV32I base ISA. This motivates restricting the supported subset, as the complete $Zve32x$ specification exceeds 100 instructions.

The implemented *Zve32x* subset is designed for integer embedded workloads: it supports vector element widths (SEW - Selected Element Width) of 8/16/32 bits, vector configuration, and all vector loads/stores. The design omits widening, reduction, fixed-point, and mask instructions, and supports only a subset of permutation operations, excluding multi-element slides, register gather, and compress, to reduce implementation and verification complexity. The implemented subset was validated using representative machine-learning benchmarks compiled with GCC 14 and GCC 15 auto-vectorization; no unimplemented instructions were generated, indicating that these omissions do not affect the reported speedups.

RS5V-SoC extends the RVV architecture presented in [18] by parameterizing both the memory interface width and the number of vector lanes. These modifications significantly modify the original design. The Vector Unit (VXU) is implemented as a **tightly** coupled accelerator and operates as a multi-cycle execution unit integrated into the 3rd pipeline stage. Vector instructions are decoded by the processor decode stage and executed by the VXU.

The parameters explored in this work are: (i) *VLEN*, the physical size of each vector register in bits; (ii) the number of *lanes*, corresponding to the processing elements that sequentially operate on a vector register; and (iii) the *bus width*, which ranges from 32 bits to VLEN and determines the number of bits transferred per unit-stride memory access. Increasing the bus width reduces the latency of load and store operations and can therefore improve performance. Concerning the number of lanes, consider, for example, a 128-bit VLEN, a 32-bit SEW, and four lanes: a single vector instruction processes four elements in parallel. With fewer lanes, additional instructions are required to complete the same vector operation.

To evaluate the RS5 VXU performance, we used eight benchmarks. Some benchmarks were adapted from the ARA repository [28, 29, 30] to support only integer data types and ensure compatibility across different VLEN configurations. The selected benchmarks are *Dot-Product*, *Conv3D*, *Dropout*, *GEMV*, *JacobiD*, *Matmul*, *Pathfinder*, and *SPMV*. All benchmarks were manually vectorized to eliminate compiler-induced variability. Scalar versions were also implemented to measure speedup. Simulations were performed post-synthesis and annotated with the worst-case corner Standard Delay Format (SDF).

We synthesized 29 VXU configurations to identify the one that provides the best performance–area trade-off. These configurations vary the VLEN (64, 128, and 256 bits), the bus width (32 to 256 bits), and the number of lanes (1 to 8). The simulation setup includes only the processor and memory, not the SoC, to avoid interference from the bus and peripherals with the results.

Table I summarizes the design space exploration. The first three columns describe the VXU configuration, followed by the geometric mean speedup across all benchmarks, the area normalized to the scalar core, and the resulting Figure of Merit (FoM).

Speedup is primarily correlated with the number of lanes and, secondarily, with the VLEN size. The bus width has a comparatively minor impact. Configurations that have larger VLENs and higher lane counts achieve the highest speedups. The **area** is dominated by the VLEN size, mainly due to the vector register file, followed by the number of lanes, which determines the degree of parallel execution.

Equation 1 defines the Figure of Merit (FoM), which captures the trade-off between performance and area. A higher FoM indicates an efficient solution that achieves higher speedup without an excessive increase in area. Conversely, a low FoM indicates an inefficient solution that consumes significant area while delivering limited performance gains.

$$FoM = \frac{Speedup}{NormalizedArea} \tag{1}$$

The four highest FoM values in Table I are highlighted in gold. The best overall area–performance trade-off is achieved with VLEN = 128 bits, a 32-bit bus width, and four lanes (128-32-4 configuration). While increasing the bus width in configurations 128-64-4 and 128-128-4 improves

Table I.: Speedup–area trade-off for the 29 VXU configurations synthesized in a 65-nm technology. Speedup is measured relative to the scalar baseline, based on the number of clock cycles required to execute the benchmark suite.

VLEN	Bus width	Lanes	Speedup (Geom. Mean)	Relative Area	FoM
Scalar Core - RS5 without VXU			1.00	1.00	1.00
64	32	1	1.59	1.52	1.05
		2	2.08	1.60	1.30
	64	1	1.65	1.95	0.84
		2	2.18	2.02	1.08
128	32	1	2.06	2.41	0.86
		2	2.91	2.51	1.16
		4	3.72	2.69	1.38
	64	1	2.20	2.76	0.80
		2	3.18	2.94	1.08
		4	4.16	3.13	1.33
	128	1	2.25	3.69	0.61
		2	3.28	3.80	0.86
		4	4.34	4.00	1.08
256	32	1	2.43	4.17	0.58
		2	3.64	4.33	0.84
		4	4.95	4.68	1.06
		8	6.15	5.34	1.15
	64	1	2.62	4.62	0.57
		2	4.06	4.78	0.85
		4	5.71	5.13	1.11
		8	7.31	5.80	1.26
	128	1	2.74	5.47	0.50
		2	4.33	5.67	0.76
		4	6.23	6.01	1.04
		8	8.15	6.61	1.23
	256	1	2.77	7.80	0.36
		2	4.42	7.43	0.60
		4	6.43	7.84	0.82
		8	8.49	8.43	1.01

speedup, it incurs a higher area cost. The largest configuration (256-bit VLEN) achieves an average speedup of 8.49× in the most complex hardware configuration, at the expense of an area increase of 8.43× relative to the scalar core.

The average speedup presented in Table I was obtained with a 32-bit SEW. For an 8-bit SEW, the speedup is multiplied by ≈4, since each vector instruction now processes 16 elements instead of 4. For the *Conv3D* benchmark, the speedup increased from 7.8 to 32.2.

Based on these results, we selected the VXU configuration 128-32-4 for the `RS5V-SoC`. Choosing a 32-bit bus width is particularly advantageous, as it avoids modifications to the existing memory interface.

ASIC RVV implementations such as ARA (0.44 mm^2 in 28 nm with 4 lanes and VLEN=4096) and SPEED (1.10 mm^2 in 28 nm with 4 lanes and VLEN=4096) target higher-throughput workloads. Nevertheless, RS5V-SoC is positioned as a compact RVV-enabled SoC for resource-constrained embedded systems, supporting vector lengths up to 256. A logical synthesis of the RS5V-SoC with its largest configuration (8 lanes with VLEN=256) in TSMC 28 nm at 500 MHz (the same target frequency reported for ARA and SPEED) occupies 0.17 mm^2. Therefore, emphasizing a different design point in the RVV SoC space: reduced area cost with scalable vector support for embedded workloads.

IV RS5V-SoC ASIC DESIGN

This section details the design of the `RS5V-SoC`, presenting each step from Register-Transfer Level (RTL) to the Graphic Design System II (GDSII) layout, with the resulting design ready for tape-out:

- Section IV.A - describes the method adopted to verify the SoC after each design step;
- Section IV.B - floorplanning of the SoC, considering the die area, PADs, macros (memories) and standard cells;
- Section IV.C - physically aware logic synthesis, guided by the floorplan;
- Section IV.D - formal verification between the RTL and the synthesized netlist;
- Section IV.E - physical synthesis and signoff steps;
- Section IV.F - Layout Versus Schematic (LVS), comparing the GDSII against the netlist;
- Section IV.G - Design Rule Check (DRC) on the GDSII (rules, antenna, wirebond);
- Section IV.H - metal fill, final step of the flow prior to tape-out.

The target technology is TSMC 65 nm (*tcbn65lp_220a*). The tape-out is in progress and is funded by SBMicro through the APCI 2025 call [31]. The flow presented in this section is general and may be applied to other technologies.

IV.A Functional Verification

This paper does not aim to describe the verification methodology. The design employs a regression-testing framework in which testbenches exercise each system module by executing software on the RS5 core. Regression tests are first applied at the RTL and subsequently repeated after logical synthesis (Section IV.C) and physical synthesis (Section IV.E), corresponding to gate-level simulations with timing annotation.

This procedure enables the functional verification of the SoC after each design step. The most critical step is the first annotated simulation performed after logical synthesis because the RTL description may contain elements that do not translate into correct or fully functional hardware. For this reason, comprehensive regression coverage that can exercise all SoC subsystems is essential. The second annotated simulation, executed after physical synthesis, further verifies whether the final timing constraints are satisfied across the analyzed corners.

Regarding back-annotated simulations, it is important to verify: (1) whether combinational delays are fully and correctly back-annotated, as indicated by the number of *Pathdelays* (Figure 5); and (2) whether combined setup/hold timing checks are fully annotated, as reflected by the *$setuphold* entry. The absence of annotation for individual *$setup* and *$hold* checks, together with 100% annotation of *$setuphold*, is expected in modern SDF flows and does not indicate missing timing verification. Figure 5 presents the simulation log associated with this verification. Verifying delay annotation is essential, as it ensures that the back-annotated simulation effectively accounts for delays introduced by physical interconnections and cells present in the circuit.

Note that the final signoff (Figure 11) is always performed with STA (static timing analysis) tools, not simulation. SDF-based simulation is a sanity check, not a replacement for STA.

```
SDF statistics:
 No. of Pathdelays = 1030808     No. of Disabled Pathdelays = 0    Annotated = 100.00%} (1030783/1030808)
 No. of Tchecks    = 109995      No. of Disabled Tchecks    = 0    Annotated = 76.11% (83712/109995)
                       Total(T)       Disabled(D)      Annotated(A)      Percentage(A/(T-D))
 Path Delays            1030808                 0             1030783                   100.00
      $setup                 10                 0                   0                     0.00
       $hold              13132                 0                   0                     0.00
     $period                 15                 0                   9                    60.00
      $width              40109                 0               40093                    99.96
   $recovery              13119                 0                   0                     0.00
     $recrem                  4                 0                   4                   100.00
  $setuphold              43606                 0               43606                   100.00
```

Fig. 5 SDF log for the RS5V-SoC.

IV.B Floorplanning

After RTL validation, the first step in generating the design is the floorplanning phase. We used Cadence Innovus 22.1. Design flows frequently start with logic synthesis. However, in designs with macros, such as memories and pads, the floorplan guides the logic synthesis to achieve timing closure.

Before running the floorplan script, it is necessary to prepare the macros: memories, IO cells, and bonding pads.

The design contains three memory modules: the main memory (32 KB) and two buffers for the QSPI module (1 KB and 0.5 KB). A memory generator tool from TSMC creates these blocks, providing LEF, GDS, and Verilog views, among others. In the context of floorplanning, the LEF files provide the physical dimensions of these blocks, which guide the floorplanning process.

For this project, a 44-pin package is used, of which 36 pins are occupied:

- 4 pins for core power supply (core_vdd1, core_vdd2, core_vss1, core_vss2);
- 4 pins for pad-ring power supply (ring_vdd1, ring_vdd2, ring_vss1, ring_vss2);
- 2 pins for the UART interface (pad_rx, pad_tx);
- 2 synchronization pins (pad_clk, pad_rst);
- 15 bidirectional GPIO pins, providing flexible external interfacing and software-controlled configuration as inputs or outputs;
- 9 pins for the QSPI interface, used to connect the SoC to an external flash memory.

The positions of these 36 pins are planned before floorplanning begins, using an io file. This file specifies the pad-ring configuration and the pad placement. It defines global IO parameters, margin rules, and assigns signal, power, ground, and corner pads to fixed positions around the die.

Floorplanning **starts** by configuring the physical environment and the technology context. The floorplanning script loads path definitions, sets the top module name, and defines the power and ground nets (VDD, VDDPST, VSS, VSSPST, POC (Power-On Control)). Multi-Mode Multi-Corner (MMMC) views are then loaded to establish timing and operating conditions. After that, the script reads all LEF files, including standard-cells, memories, and IO pad libraries. This step defines the physical geometry and routing rules for each cell in the layout. Finally, a *dummy* Verilog netlist of the SoC is imported, and the design is initialized, which creates the internal database representation required for physical implementation. The *dummy* Verilog instantiates the top module (empty) and all IO cells.

Next, the script defines technology and routing constraints. The process node is defined as 65 nm, and the allowable routing stack is constrained from M2 to M8.

The die, IO, and core regions are then defined using geometric constraints. The SBMicro APCI call [31] defined the die area as 1000×1000 μm (1 mm^2). An IO-to-core margin of 110 μm is reserved for the pad ring, power rings, and routing, resulting in a core area of **0.608** mm^2.

After defining these regions, the floorplan is created using a 7-track standard cell library (core7T). At this step, global power and ground nets are logically connected to all matching instance pins. This ensures that all standard cells and macros will later receive power via a global distribution network.

The next step is to place the IO cells by reading the io file. The script uses the -no_die_size_adjust option to ensure that these placements do not modify the pre-defined die dimensions. After pad placement, IO filler cells are inserted to maintain pad-ring continuity, which is essential for both manufacturing integrity and electrostatic discharge (ESD) protection. It is important to mention that this SoC has only one power domain (digital), so there is no need to use IO cut cells. However, if the chip contains more than one power domain and mixes analog and digital modules, in addition to the IO filler cells, IO cut cells should also be placed.

In this design, the IO region uses a Circuit Under Pad (CUP) technique, in which the IO cell is placed below a top-level bonding pad. The bonding pad provides the mechanical and electrical interface for wire bonding, while the underlying IO cell implements the active circuitry required for signal buffering, level shifting, and ESD protection. This vertical integration improves area efficiency by allowing core circuitry to be placed beneath the bonding pad.

Following pad placement, the flow proceeds to place macros, the main RAM, and the two QSPI First-In, First-Out (FIFO) structures. Their locations are computed relative to the core boundaries to achieve controlled placement near the core edges, thereby sharing the power supply with the core power ring. Each macro is instantiated with a defined position and orientation, and it is marked as fixed to prevent it from moving.

To protect these macros, the script applies placement and routing halos. Placement halos reserve whitespace around macros to prevent standard cells from being placed too close, while routing halos prevent signal routing from crossing sensitive macro areas. These protections reduce coupling noise, improve routability, and prevent DRC violations.

Figure 6 presents a graphical view of the floorplan after

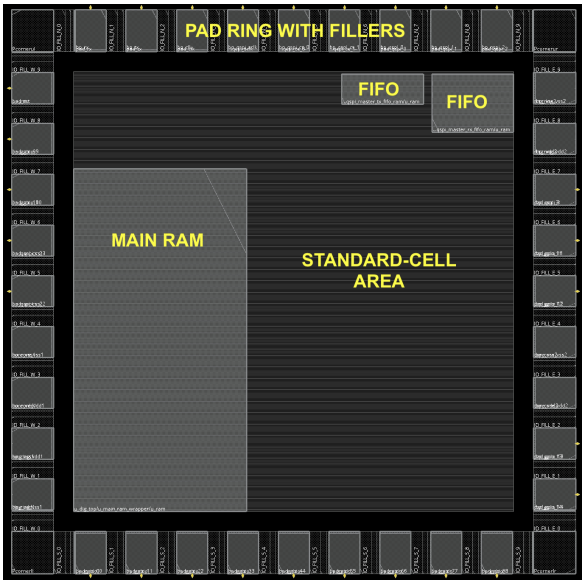


Fig. 6: Initial floorplanning, with IO cells (pad and superposed bonding pad), memory blocks, and standard-cell regions defined.

these initial steps. Placing the main RAM in the bottom-left corner optimizes power planning by sharing the power ring and aligns its pins with the internal interconnect (right side). This placement also avoids fragmenting the standard-cell region, which is important for cell placement, clock-tree synthesis, and routing. The FIFO macros, which are associated with peripheral communication, were placed close to the upper pad ring and away from the central processor logic. This decision keeps the peripheral data paths close to the I/O boundary, reduces routing congestion in the processor region, and leaves a contiguous standard-cell area for implementing the core logic. Therefore, we selected the placement of the macro blocks to reduce routing complexity, preserve placement regularity, and simplify timing and physical-design closure.

The next phase places `endcaps` and `well taps`. Endcaps are inserted at every standard-cell row boundary to ensure well and diffusion termination (Figure 7a), which is necessary for manufacturability. Well taps are then inserted at a controlled spacing to bias the substrate and wells, preventing latch-up and ensuring stable transistor operation. A verification step checks that the maximum allowed distance between taps is respected.

Power planning is performed through a hierarchical infrastructure of rings and stripes. First, 7 μm power rings are created around the entire core (horizontal: M7, vertical:M8). The peak current supported by these external rings is 64.02 mA@100^{o}C. These rings provide a low-resistance global path for power and ground. Next, dedicated block-level rings are added around each memory macro (Figure 7b) using intermediate metal layers (M4 and M5). These local rings isolate macro-level current demand and improve local voltage stability, both of which are essential for reliable SRAM operation.

To strengthen the global power network, it is necessary to insert vertical and horizontal power stripes across the core. Vertical stripes on M8 and horizontal stripes on M7 distribute power uniformly and reduce IR drop. Figure 7c shows the inserted stripes. Note that stripes do not traverse the QSPI FIFOs due to design rules associated with these blocks.

Once rings and stripes are in place, power routing is executed. This step connects power nets between core supply pads, block pins, and core pins across all allowed layers from M1 to M8. This ensures that power is not only geometrically present but also electrically continuous throughout the design.

Next, the script executes pin assignment, during which all IO pins are aligned and legalized based on the pad placement. The script performs checks to ensure that all pins comply with spacing, alignment, and accessibility rules. This step ensures that the logical ports defined in the *dummy* Verilog are physically accessible through the correct pads. Figure 7d presents the final floorplan layout.

The resulting floorplan is written as Design Exchange Format (DEF) and Library Exchange Format (LEF) files, along with a database snapshot. This floorplan serves as input to the logic synthesis.

IV.C Logic Synthesis

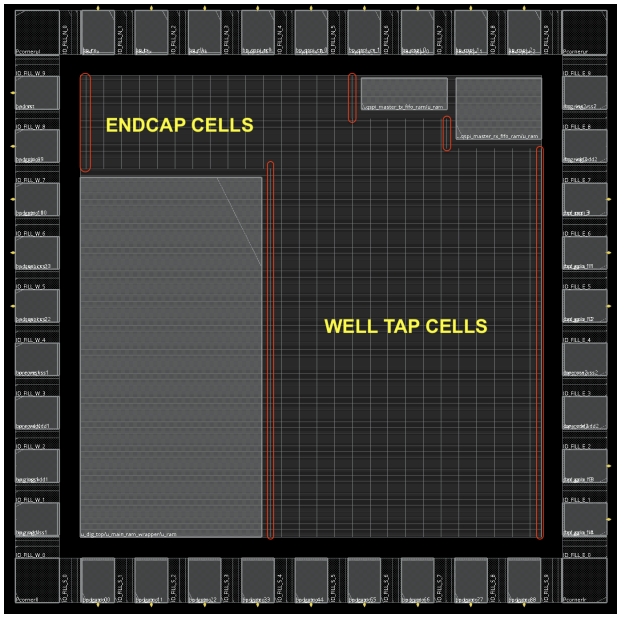
The logical synthesis, executed with Cadence `Genus` 22.17, uses a **physically aware** configuration, with the generated floorplan (DEF) as the physical reference. The interconnect modeling mode is set to Physical Layout Estimator (PLE), which allows `Genus` to extract wire parasitics from the physical layout rather than abstract wire-load models. This step ensures that timing optimization during synthesis reflects interconnect delays derived from the floorplan.

Several global optimization and power-awareness settings are enabled. High effort is selected for both synthesis and power optimization, ensuring logic restructuring and cell selection. Latch inference is disabled to prevent unintended level-sensitive elements. Retiming is also enabled with high effort, allowing `Genus` to move registers across combinational logic for improved timing, even in the presence of asynchronous resets. A dedicated verification flow with `Conformal` is activated to verify the correctness of the retimed logic formally.

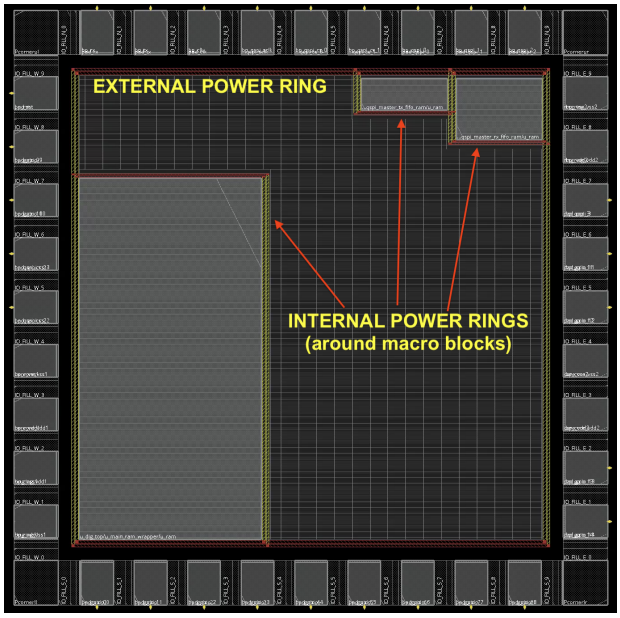
After these configurations, the MMMC timing environment is loaded. This introduces the set of Process–Voltage–Temperature (PVT) corners, RC corners, delay corners, and analysis views that will be used throughout synthesis. In parallel, all physical libraries are read via LEF files, including standard cells, SRAM blocks, and the IO pad library. This provides physical dimensions, pin locations, and routing blockages for all cells and macros. Next, the complete RTL is then read.

The IO pad cells and power-related cells are explicitly declared as *don't-touch* objects. This prevents `Genus` from optimizing, resizing, or restructuring any part of the pad ring, corner cells, or power pads, which is mandatory because these elements are fixed by the physical design rules and by the floorplanning step.

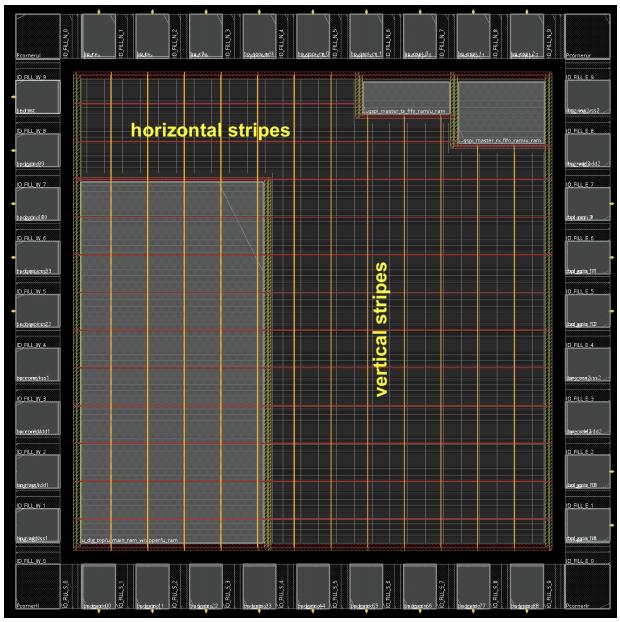
The design is then elaborated to create the hierarchical internal representation of `RS5V-SoC`. At this point, the previously generated floorplan DEF is loaded into `Genus`, with the options to preserve all instances and routed nets. This operation tightly couples synthesis with the physical layout by



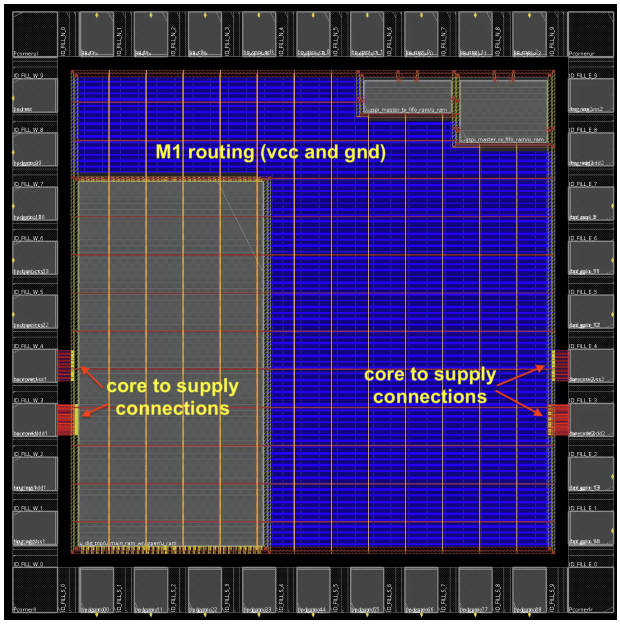
(a) Placement of the `endcaps` and `well taps` cells.



(b) External and internal power rings. Internal power rings supply macro (memories) blocks.



(c) Floorplanning with vertical and horizontal power stripes.



(d) Final floorplan layout.

Fig. 7 Floorplanning of the `RS5V-SoC`.

importing macro placement, core boundaries, routing blockages, and power structures. The design is then initialized with explicit top and bottom routing layer limits, ensuring consistency with the routing stack defined during floorplanning.

A set of integrity and timing checks is performed after elaboration. These include structural consistency checks, verification of timing constraints, clock reporting, clock-group validation, and unconstrained-path detection. This early analysis stage ensures that the combination of RTL, floorplan, MMMC environment, and SDC constraints forms a coherent space before any mapping or optimization begins.

The core of the flow is the physically guided synthesis, with 3 main steps: (*i*) generic synthesis with physical awareness (`syn_generic -physical`); (*ii*) technology mapping under floorplan constraints (`syn_map -physical`); and (*iii*) spatial optimization (`syn_opt -spatial`). During this step, timing, area, and power are optimized with explicit knowledge of instance placement, macro blockages, routing congestion, and estimated interconnect delays.

After physical optimization, the netlist is uniquified, ensuring that all hierarchical instances are uniquely named. This is an important step for equivalence checking.

The script then generates a set of reports, including clock-gating statistics, PLE-based interconnect estimation, gate count, area breakdown, and sequential optimization summaries. Timing and power reports are generated for each MMMC analysis view under worst-case, nominal, and best-case operating conditions. For each of these views, a SDF file is produced, enabling post-synthesis simulation.

Finally, the synthesis deliverables are written. These include the mapped gate-level Verilog netlist, a version prepared for logical equivalence checking, and an automatically generated `Conformal` script for formal verification against the RTL. In addition, a complete `Genus` design database is exported in a format compatible with `Innovus`.

Points to note at the end of the logic synthesis:

1. Generated SDFs for the three corners, in addition to the mapped netlist;
2. Generated the `Conformal` script;
3. Area report, summarized in Table II. The **cell area** (0.461 mm^2) must be lower than the floorplanning area (0.608 mm^2). If the estimated cell area exceeds the initial floorplan area, it is necessary to reexecute it with a larger area. It is the cell area that is considered, not the total area, because the nets (Net Area) are routed over the cells;

Table II.: Area breakdown of main SoC instances extracted from the Genus report. Area in μm^2.

Instance	Cell Count	Cell Area	Net Area	Total Area
cpu	84,184	228,962	178,280	407,242
u_main_ram_wrapper	44	186,481	388	186,869
u_qspi_master	1,744	27,245	4,428	31,673
random_gen_inst	1,703	5,211	2,280	7,490
gpio_ctrl	830	3,423	2,980	6,403
u_mem_loader	777	2,675	551	3,227
rtc1	674	2,288	577	2,866
uart_rx_tx	444	1,527	508	2,035
systimer_inst	339	1,215	423	1,638
u_bus	255	480	969	1,449
plic1	193	750	285	1,035
UART_RX_32	91	631	67	698
plic_to_axi	37	88	8	96
RAM_to_AXI	26	64	14	77
rtc_to_axi	14	39	3	42
Others	368	801	10,291	11,092
Total	**91,723**	**461,881**	**202,052**	**663,933**

4. Timing report should present positive slack for all three corners. Figure 8 shows partial reports. For this design, the clock was set to 8 ns. We accepted a small violation in the slowest case to be fixed in the signoff step.

```
Path 1: VIOLATED (-2 ps) Setup Check with Pin
↪  u_dig_top/cpu/fetch1/instruction_o_reg[15]/CP->D
          View: analysis_view_1p08v_125c_capwst_slowest
         Group: p_clk

Path 1: MET (1739 ps) Setup Check with Pin
↪  u_dig_top/u_qspi_master/data_to_spi_reg[1]/CPN->D
          View: analysis_view_1p20v_25c_captyp_nominal
         Group: clk_qspi_div_1

Path 1: MET (2234 ps) Setup Check with Pin
↪  u_dig_top/u_qspi_master/u_spi_clkdiv/negedge_shadow_ff_q_reg/CPN->D
          View: analysis_view_1p32v_m40c_capbst_fastest
         Group: p_clk
```

Fig. 8 Timing reports for the `RS5V-SoC`.

IV.D Formal Equivalence Verification

Formal equivalence verification, executed with Cadence `Conformal` 23.2, ensures that the synthesized netlist preserves the functional behavior of the original RTL description. Unlike simulation-based approaches, which rely on input stimulus and therefore cannot guarantee full coverage, formal verification analyzes all possible input combinations and state transitions. This comparison is independent of test vectors, enabling the detection of functional mismatches that might arise during logic synthesis, retiming, resource sharing, finite-state machine re-encoding, or other structure-changing optimizations.

`Conformal` operates by elaborating both the golden model (the RTL description) and the synthesized netlist, building canonical logic representations, and comparing corresponding points in the designs. It identifies equivalence classes, unreachable logic, sequential correspondences, black-box boundaries, datapath transformations, and any discrepancies in control or datapath behavior. Special handling is required for memories, macros, and clock-gated or retimed structures, all of which are incorporated into `Conformal`'s mapping and proof engines.

In this design, `Conformal` identified non-equivalences originating from the vector processing unit. These findings revealed design inconsistencies that were not observed during simulation, including: (i) the absence of explicit reset behavior in several registers, which could lead to undefined behavior after initialization; (ii) the presence of free-mode registers (sequential elements whose next-state logic is not fully defined for all input combinations or execution paths), which could produce unpredictable values under corner-case input sequences; and (iii) the existence of unintended combinational loops, which could compromise timing analysis and result in silicon-level functional failures.

The correction of such conditions required modifications to the RTL before synthesis to ensure deterministic initialization, complete specification of next-state logic, and removal of feedback paths. By resolving these issues at the RTL level, the final synthesized design was confirmed equivalent by Conformal, showing that netlist matches the RTL netlist architectural behavior, as presented in Figure 9.

```
// Run Hier_compare Command      : run_hier_compare
↪  ../../../build/tsmc_65/work/hier_tmp2.lec.do -dynamic_hierarchy
↪  -verbose (line: 140888)
// Hierarchical Compare Result   : Equivalent
// Smart instance selection      : Disable

// Hierarchical Comparison Statistics
================================================================
     Modules       Total         Max          Min          Avg
----------------------------------------------------------------
         119        1149          82            0           10
================================================================
  Modules: Total number of modules.
  Total: Sum of all module runtimes(sec).
  Max: Runtime of module with longest runtime.
  Min: Runtime of module with shortest runtime.
  Avg: Average runtime of all modules.
```

Fig. 9 Formal equivalence report for the `RS5V-SoC`.

IV.E Physical Synthesis and Signoff

The physical synthesis begins by importing the `Genus`-exported design database into `Innovus`. Power and ground nets are declared, and power/ground pins are instantiated on metal layers to identify the supply nets. The same MMMC timing environments used during logical synthesis are reapplied to achieve timing closure. Global effort levels for timing, congestion, and power optimization are set to extreme, and `Innovus` is configured to perform signal integrity (SI) timing analysis. Routing is constrained from M2 to M7.

Note that power stripes use M8, but for standard-cell routing, this layer was avoided because we observed DRC antenna errors (Section IV.G) when using it.

The placement begins with a pre-placement timing analysis that characterizes the design before the cells are moved. Routing blockages are applied over memory macros to prevent illegal wiring in these regions and to minimize DRC and coupling risks. The `place_opt_design` command performs global and detailed placement with timing- and congestion-driven optimization. Tie-high and tie-low cells are then inserted with controlled fanout and distance to ensure proper handling of constant logic levels. After placement legality checks, `Innovus` proceeds to Clock Tree Synthesis (CTS).

CTS constructs balanced and physically feasible clock networks. The `clock_opt_design` command builds the clock distribution network to minimize skew and insertion delay. Post-CTS optimization resolves violations introduced by the clock network, focusing first on hold violations, then on setup violations, and finally on electrical constraints. Timing analysis after CTS verifies that the design is converging toward closure across all corners.

Figure 10 illustrates the synthesized clock distribution tree, showing the hierarchical propagation of the clock signal from the IO pad to the sequential elements. The vertical axis represents the clock arrival time in nanoseconds, while the horizontal axis reflects the structural branching introduced by CTS. Most clock sinks exhibit clustered arrival times, indicating a balanced distribution. A visible spread among leaf nodes indicates clock skew. The specified skew was set to 5% of the clock period (8 ns), corresponding to 0.2 ns. The figure shows that the vertical distance between leaves respects this constraint (between 0.642 and 0.762 ns).

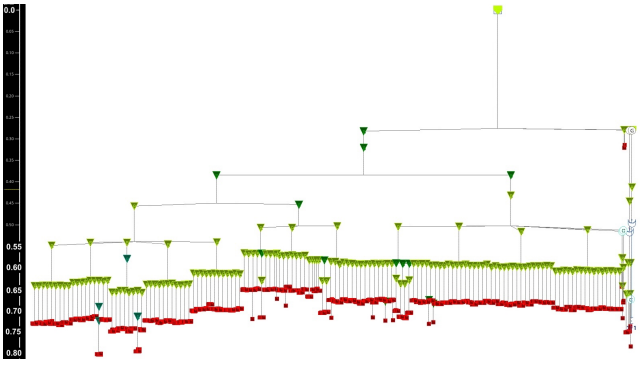


Fig. 10 Clock distribution tree.

Routing is then performed with `route_opt_design`, which integrates congestion reduction, timing improvement, and SI mitigation. After routing, `Innovus` switches to post-route RC extraction and performs an extraction to obtain interconnect parasitics. Post-route optimization focuses on eliminating remaining setup and hold violations using extracted delays.

The flow then starts the **signoff** step, which includes wire spreading, multi-cut via insertion, and reliability improvements. Additional routing refinement commands further improve the detailed routing. Figure 11 presents the signoff report.

time_design_signoff Summary

Setup mode	all	default	reg2cgate	reg2reg
WNS (ns):	0.001	6.166	1.149	0.001
TNS (ns):	0.000	0.000	0.000	0.000
Violating Paths:	0	0	0	0
analysis_view_lp08v_125c_capwst_slowest				
	0.001	6.166	1.149	0.001
	0.000	0.000	0.000	0.000
	0	0	0	0
analysis_view_lp20v_25c_captyp_nominal				
	1.528	6.764	3.961	1.528
	0.000	0.000	0.000	0.000
	0	0	0	0
analysis_view_lp32v_m40c_capbst_fastest				
	2.289	7.099	5.357	2.289
	0.000	0.000	0.000	0.000
	0	0	0	0

DRVs	Signal nets		Clock nets	
	Nr nets(terms)	Worst Vio	Nr nets(terms)	Worst Vio
max_cap	0 (0)	0	0 (0)	0
max_tran	0 (0)	0	0 (0)	0
max_fanout	1 (1)	-33	0 (0)	0

Hold mode	all	default	reg2cgate	reg2reg
WNS (ns):	0.100	N/A	0.551	0.100
TNS (ns):	0.000	N/A	0.000	0.000
Violating Paths:	0	0	0	0
analysis_view_lp08v_125c_capwst_slowest				
	0.293	N/A	1.667	0.293
	0.000	N/A	0.000	0.000
	0	0	0	0
analysis_view_lp20v_25c_captyp_nominal				
	0.166	N/A	0.930	0.166
	0.000	N/A	0.000	0.000
	0	0	0	0
analysis_view_lp32v_m40c_capbst_fastest				
	0.100	N/A	0.551	0.100
	0.000	N/A	0.000	0.000
	0	0	0	0

Fig. 11 Signoff report for the `RS5V-SoC` @ 125 MHz.

The timing signoff report, Figure 11, shows that the circuit satisfies all setup and hold timing constraints across the analyzed corners. Setup analysis shows no violations, with zero Total Negative Slack (TNS) and a marginally positive Worst Negative Slack (WNS) in the slowest operating condition (1.08 V, 125 °C), indicating tight but valid timing convergence. In nominal and fast corners, the timing margins increase, confirming the design robustness. Hold analysis likewise reports no violations, with positive margins in all scenarios, including the fastest corner. Electrical rule checking reveals no capacitance or transition violations and only a single isolated fanout violation on a signal net. Overall, the report indicates that the design is timing-clean and suitable for tape-out.

DRC is executed with Siemens `Calibre` v2022.3, its markers imported, and signoff DRC fixes applied. This extraction–timing–DRC loop is repeated until both electrical and manufacturing constraints meet signoff criteria. Filler cells are inserted to ensure well and diffusion continuity across the layout.

The post-physical synthesis power estimation was obtained using `Innovus` (`IQuantus` Extraction and `Voltus` Power Estimation) with default activity assumptions. Table III summarizes the power breakdown by group, executed for the typical corner, VDD=1.2V, f=125MHz. The total power at the low-VDD (worst-case timing) and high-VDD (best-case timing) corners is 55.27 mW and 85.41 mW, respectively.

Table III.: Post-physical synthesis power breakdown by group, typical corner. Dynamic power is estimated by `Innovus` using default activity assumptions. Unit: **mW**.

Group	Internal	Switching	Leakage	Total	(%)
Sequential	28.33	1.43	0.00	29.76	43.45
Macro	4.47	0.02	0.02	4.50	6.57
IO	13.84	1.12	0.00	14.96	21.84
Combinational	5.70	9.64	0.01	15.35	22.40
Clock	0.63	3.31	0.00	3.93	5.74
Total	**52.96**	**15.50**	**0.03**	**68.49**	**100.00**

Although this power estimation is conservative (Table III), it can be interpreted as an upper bound on power as it assumes high switching activity across all system modules. To obtain workload-representative values, we performed activity-based power analysis by extracting switching activity from post-layout SDF-annotated simulations. Even compressed switching activity formats (e.g., `shm`) can produce large files (over 20 GB), resulting in long simulation and power estimation execution times. Therefore, the resulting power figures presented in Table IV reflect the processor–bus–memory subsystem, excluding peripherals and IO cells. Overall, the activity-annotated results provide a more accurate power estimate for the evaluated workloads, while the default estimate serves mainly as a reference for SoC power dissipation.

Table IV.: Power obtained from the switching activity - row "`Innovus` estimation " for comparison purposes. Unit: **mW**.

Benchmark	Worst-Case	Typical	Best-Case
`Innovus` estimation	55.27	68.49	85.41
fibonacci	28.24	32.88	41.24
genv (vect)	30.47	35.34	44.10
conv3d-8b (vect)	30.84	35.43	44.41
dotproduct-8b (vect)	34.70	38.41	48.78
matmul-8b (vect)	36.48	39.77	50.86

Table V reports the area breakdown of the main blocks. As discussed in Section IV.B, the core floorplan area is 0.608 mm^2, of which 0.462185 mm^2 is occupied, corresponding to an effective utilization of 76%. Note that the standard-cell area is dominated by the CPU, which accounts for approximately 90%. The memory subsystem is implemented using SRAM macros in the `u_main_ram_wrapper` and `u_qspi_master` blocks.

Compare Table V to Table II (`Genus` area estimation): 0.461880 mm^2 versus 0.462185 mm^2, a small difference due to the insertion of physical cells, such as clock buffers.

The final deliverables include a GDSII file, Figure 12, simulation and LVS netlists, SDF files, and an `Innovus` design database.

IV.F LVS – Layout Versus Schematic

LVS verification is a step that ensures the physical layout of a circuit implements the logical design described by the schematic or gate-level netlist. As mentioned above, `Conformal` ensures the formal equivalence between the RTL and the netlist. LVS compares the devices and connections extracted from the GDSII layout against the netlist to guarantee that both representations describe the same electrical circuit. Issues such as shorted nets, missing connections, incorrect transistor dimensions, swapped pins, or unintended device creations cannot be detected through timing or functional simulation. LVS is therefore a requirement before tape-out.

Table V.: Area breakdown of the main blocks under `RS5V-SoC`. Percentages are normalized to total area (462,185.49 μm^2).

Block	Std-cell (μm^2)	RAM (μm^2)	Total (μm^2)	(%)
`cpu`	229197	0	229197	49.59
`u_main_ram_wrapper`	57	186414	186472	40.35
`u_qspi_master`	4642	22615	27258	5.90
`random_gen_inst`	5211	0	5211	1.13
`gpio_ctrl`	3438	0	3438	0.74
`u_mem_loader`	2667	0	2667	0.58
Others (peripherals/glue)	7943	0	7943	1.72
Total	**253156**	**209030**	**462185**	**100.00**

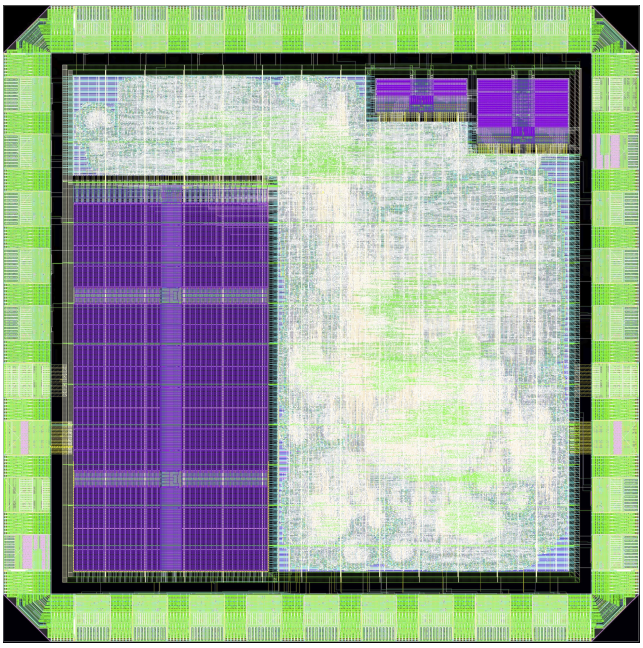


Fig. 12 Final layout of `RS5V-SoC`.

The LVS flow begins by translating the netlist produced by `Innovus` into a SPICE-format netlist using `Calibre v2lvs` (Verilog-to-LVS). This conversion produces the "schematic" representation that `Calibre` uses for comparison, with definitions of supply nets and include files for IO and technology models.

`Calibre` LVS is then executed in hierarchical mode, using the foundry rule deck and `.hcell` descriptions for memory blocks to accelerate the comparison. `Calibre` extracts devices and nets from the GDSII layout and performs a structural and electrical comparison. The tool reports any mismatches, including missing or extra devices, connectivity differences, pin swaps, and parameter deviations.

Finally, LVS and Electrical Rule Check (ERC) reports are generated. The ERC report highlights electrical issues such as floating nets or missing supplies, while the LVS report indicates equivalence status and any mismatches requiring correction. A successful LVS confirms that the final mask layout is electrically equivalent to the synthesized and implemented design.

Figure 13 presents a snippet of the LVS report, showing the equivalence between the "schematic" and the layout. The LVS log also shows that the SoC has **2,756,627** transistors, 146 resistors, and 429 diodes.

```
****************************************************************
          OVERALL COMPARISON RESULTS
****************************************************************

                    #       ###################       _   _
                   #        #                 #       *   *
                 #   #      #     CORRECT     #         |
                  # #       #                 #       \___/
                   #        ###################

  Warning:  Unbalanced smashed mosfets were matched.
  Warning:  Ambiguity points were found and resolved arbitrarily.

NUMBERS OF OBJECTS AFTER TRANSFORMATION
----------------------------------------

                Layout    Source         Component Type
                ------    ------         --------------
 Ports:             33        33

 Nets:         1103843   1103843

 Instances:    1658204   1658204         MN (4 pins)
               1098423   1098423         MP (4 pins)
                   146       146         rppolywo (2 pins)
                   429       429         D (2 pins)
                ------    ------
 Total Inst:   2757202   2757202
```

Fig. 13 LVS check for the RS5V-SoC.

IV.G DRC – Design Rule Checking

DRC is a step in physical verification that ensures the layout complies with the geometric and topological manufacturing constraints defined by the foundry. Even if a design is functionally correct and passes LVS, it may still be impossible to fabricate reliably if it violates DRC constraints. The DRC verification requires three complementary steps:

1. Full foundry **DRC** deck. This check validates widths, spacings, enclosures, metal density, and layer interactions across the chip. Its purpose is to guarantee compliance with the baseline manufacturing rules and to identify any geometric violations introduced during placement, routing, or post-route optimization.
2. **Antenna** rules, which protect MOS transistors from plasma-induced charging during fabrication [32]. Long metal segments connected only to gate terminals can accumulate charge during plasma etching, potentially damaging the gate oxide. Antenna DRC checks the ratio between metal area and gate area, identifies vulnerable nets, and ensures that necessary protection structures, such as antenna diodes, are present.
3. **Wire bonding** rules. Wire bonding is a packaging technique in which thin metal wires connect the die pads to the package pins. Validating the padframe with a wirebond DRC ensures that the layout is compatible with the target packaging technology and prevents failures such as incomplete bonds, pad cratering, or wire lift-off during assembly.

Together, these three DRC executions provide a comprehensive verification of manufacturability, electrical robustness, and packaging compatibility.

Figure 14 presents a summary of the execution of the three DRC runs. The main DRC deck reported 73 issues: 43 related to layer density, which are solved using metal fill, and 30 related to Electrostatic Discharge (ESD) in the IO cells. The latter are waivable by the foundry, as documented in the IO cells datasheet, acknowledging their occurrence and that can be waived. Antenna and wirebond presented no issues.

```
---  DRC SUMMARY
---
...
RULECHECK OD.DN.xx ..................... TOTAL Result Count = 6
RULECHECK PO.DN.2 ...................... TOTAL Result Count = 9
RULECHECK Mx.DN.1 ...................... TOTAL Result Count = 21
RULECHECK DMx.R.1 ...................... TOTAL Result Count = 7
RULECHECK ESD.22g ...................... TOTAL Result Count = 30
...
TOTAL CPU Time:                  718
TOTAL REAL Time:                 38
TOTAL Original Layer Geometries: 11114549 (86000152)
TOTAL DRC RuleChecks Executed:   1808
TOTAL DRC Results Generated:     73 (883)

---  ANTENNA SUMMARY
---
TOTAL CPU Time:                  56
TOTAL REAL Time:                 6
TOTAL Original Layer Geometries: 8388351 (57794164)
TOTAL DRC RuleChecks Executed:   25
TOTAL DRC Results Generated:     0 (0)


---  WIREBOND SUMMARY
---
TOTAL CPU Time:                  43
TOTAL REAL Time:                 6
TOTAL Original Layer Geometries: 2699486 (46221642)
TOTAL DRC RuleChecks Executed:   205
TOTAL DRC Results Generated:     0 (0)
```

Fig. 14 DRC checks for the RS5V-SoC.

IV.H Metal fill

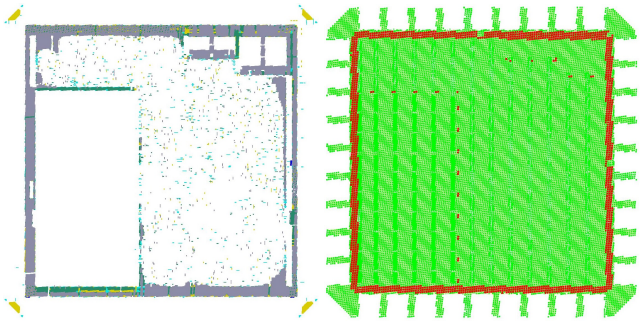
Metal fill is a mandatory tape-out step required to satisfy the foundry metal density rules. CMOS processes require each metal layer to maintain minimum and maximum pattern densities to ensure uniform planarization during Chemical-Mechanical Polishing (CMP). Without sufficient uniformity, topography variations can occur, leading to dishing, erosion, resistance shifts, or even open/short defects after fabrication. Metal fill solves this issue by inserting non-functional "dummy" metal shapes in sparsely routed regions to homogenize the density across the chip.

In this design, the fill procedure is executed through a Calibre rule deck (*Dummy_FEOL_BEOL_Calibre*). Calibre analyzes the layout, identifies low-density regions, and inserts fill polygons while ensuring they do not violate spacing rules or impact electrical behavior. After generating the fill shapes, a layout merge operation produces the final GDSII file that integrates the routed design and the dummy structures. This final filled layout is the last step prior to tape-out. Figure 15 presents two figures with dummy layers before the layout merge and the final DRC check.

V Conclusion

This paper presented the RS5V-SoC, a RISC-V–based SoC that integrates a vector processing unit (VXU) and cryptographic extensions targeting embedded and edge applications. The integration of a RISC-V Vector Extension subset into a microcontroller-oriented platform addresses a gap in the literature, which has focused on vector processing for high-performance systems rather than resource-constrained embedded designs.

This work conducted a design-space exploration of the VXU by varying architectural parameters, including VLEN, the number of lanes, and the memory bus width. The results quantified the trade-off between performance and silicon area across 29 VXU configurations, leading to the selection of a 128-bit VLEN, four-lane configuration with a 32-bit



```
---  DRC SUMMARY
---
TOTAL CPU Time:                  804
TOTAL REAL Time:                 41
TOTAL Original Layer Geometries: 12506789 (87457588)
TOTAL DRC RuleChecks Executed:   1808
TOTAL DRC Results Generated:     30 (840)
```

Fig. 15: Dummy masks. Left: M1-M4. Right: M5-M9, OD, PO. **DRC**: only the waivable issues (ESD.22g).

bus width as the most balanced solution. This configuration achieves speedups over the scalar core while preserving compatibility with the existing memory interface.

In addition, the paper described the ASIC implementation of the RS5V-SoC in 65 nm CMOS technology. The design achieved timing closure across all analyzed corners and was verified for manufacturability, demonstrating that the proposed architecture is suitable for fabrication.

The public repository, https://github.com/gaph-pucrs/RS5, provides the SystemVerilog RTL description of the RS5 processor and its supported extensions. Synthesis and physical implementation scripts are not included because they depend on technology-specific files covered by the foundry NDA.

Future work will focus on experimental validation of the RS5V-SoC through silicon testing after fabrication, including performance measurements and power characterization of the fabricated chip. Further extensions include: (i) integration of a bootloader to enable code loading directly from external flash memory via the QSPI interface; (ii) support for additional RVV features; (iii) expanded benchmark evaluation, including a normalized comparison against academic RVV-enabled designs covering area per lane and workload-based performance per watt; (iv) energy efficiency analysis comparing scalar and vectorized kernel executions under post-layout switching activity; and (v) evaluation of energy efficiency under real application workloads on the fabricated silicon.

Acknowledgments

We thank SBMicro for supporting chip funding through the APCI 2025 call and CADENCE for the academic licenses that enabled the chip design. This work was financed in part by CAPES (Finance Code 001), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), 305621/2024-6; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), 23/2551-0002200-1.

References

[1] K. Chiu, G. Eichler, B. B. Seyoum, and L. P. Carloni, "EigenEdge: Real-Time Software Execution at the Edge with RISC-V and Hardware Accelerators," in *Cyber-Physical Systems and Internet of Things Week*, 2023, pp. 209–214, https://doi.org/10.1145/3576914.3587510.

[2] ESP, "ESP – the open-source SoC platform," 2025, https://www.esp.cs.columbia.edu/.

[3] Z. Azad, G. Yang, R. Agrawal, D. Petrisko, M. B. Taylor, and A. Joshi, "RACE: RISC-V SoC for En/decryption Acceleration on the Edge for Homomorphic Computation," in *ISLPED*, 2022, pp. 1–6, https://doi.org/10.1145/3531437.3539725.

[4] B. Sá, F. Marques, M. Rodriguez, J. Martins, and S. Pinto, "Holistic RISC-V Virtualization: CVA6-based SoC," in *ACM International Conference on Computing Frontiers*, 2023, pp. 389–390, https://doi.org/10.1145/3587135.3591436.

[5] S. Machetti, P. D. Schiavone, T. C. Müller, M. P. Quirós, and D. Atienza, "X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators," *CoRR*, pp. 1–21, 2024, https://arxiv.org/abs/2401.05548.

[6] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "FANN-on-MCU: An Open-Source Toolkit for Energy-Efficient Neural Network Inference at the Edge of the Internet of Things," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4403–4417, 2020, https://doi.org/10.1109/JIOT.2020.2976702.

[7] SiFive, "SiFive Launches Industry's First Open-Source RISC-V SoC," 2016, https://www.sifive.com/press/sifive-launches-industrys-first-open-source-risc-v-soc.

[8] OpenHW, "The CORE-V Family of Open-Source RISC-V Cores," 2024, https://github.com/openhwgroup/core-v-cores.

[9] A. Jadhav, "An Overview of SHAKTI Processor Program," 2020, https://abopen.com/news/an-overview-of-shakti-processor-program.

[10] OpenTitan, "OpenTitan Earl Grey Chip Datasheet," 2024, https://opentitan.org/book/hw/top_earlgrey/doc/specification.html.

[11] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen *et al.*, "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product," in *ISCA*, 2020, pp. 52–64, https://doi.org/10.1109/ISCA45697.2020.00016.

[12] Andes Technology Corporation, "Risc-v: Nx27v," 2021, https://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v/.

[13] R.-V. Foundation, "RISC-V "V" Vector Extension," 2021, https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

[14] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," Berkeley, Tech. Rep., 2024, https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link.

[15] C. Wang, C. Fang, X. Wu, Z. Wang, and J. Lin, "A Scalable RISC-V Vector Processor Enabling Efficient Multi-Precision DNN Inference," in *ISCAS*, 2024, pp. 1–5, https://doi.org/10.1109/ISCAS58744.2024.10558028.

[16] M. Johns and T. J. Kazmierski, "A minimal RISC-V vector processor for embedded systems," in *FSL*, 2020, pp. 1–4, https://doi.org/10.1109/FDL50818.2020.9232940.

[17] R. Faccenda, W. Nunes, A. D. Zotto, C. Gewehr, L. Luza, L. Damo, V. Zanini, E. Bernardon, V. Mibielli, N. Cidal, and F. G. Moraes, "RS5-SoC: A Flexible Open-Source RISC-V Platform for Embedded Systems," in *SBCCI*, 2025, pp. 1–5, https://doi.org/10.1109/SBCCI66862.2025.11218649.

[18] W. A. Nunes and F. G. Moraes, "Accelerating Machine Learning with RISC-V Vector Extension and Auto-Vectorization Techniques," in *ISCAS*, 2025, pp. 1–5, https://doi.org/10.1109/ISCAS56072.2025.11043225.

[19] W. Nunes, A. Dal Zotto, C. Borges, and F. G. Moraes, "RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems," in *LASCAS*, 2024, pp. 1–5, https://doi.org/10.1109/LASCAS60203.2024.10506171.

[20] ARM, "AXI4 and AXI4-Lite interfaces," 2025, https://developer.arm.com/documentation/dui0534/b/Parameter-Descriptions/Interface/AXI4-and-AXI4-Lite-interfaces/.

[21] J. I. M. Bezerra, G. Machado, R. I. Soares, V. V. de Almeida Camargo, and A. Molter, "FPGA implementation of low cost and low power chaotic encryption scheme based on a discrete-space chaotic map," *Multimedia Tools and Applications*, vol. 84, no. 28, pp. 33 331–33 352, 2025, https://doi.org/10.1007/s11042-024-20537-9.

[22] I. SiFive, *SiFive Interrupt Cookbook, Version 1.2*, 2020, https://www.starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf.

[23] RISCOF, "RISC-V Architectural Test Framework," 2025, https://github.com/riscv-software-src/riscof.

[24] Zephyr, "Zephyr Project RTOS," 2025, https://www.zephyrproject.org/.

[25] Intel, "TinyCrypt Cryptographic Library," 2017, https://github.com/intel/tinycrypt.

[26] Digilent, "Nexys A7 AMD Artix™ 7 FPGA Trainer Board," 2025, https://digilent.com/shop/nexys-a7-amd-artix-7-fpga-trainer-board-recommended-for-ece-curriculum.

[27] C. Schmidt, A. Ou, and K. Asanović, "Hwacha V4: Decoupled Data Parallel Custom Extension," in *"Proc. Inaugural RISC-V Summit"*, 2018, pp. 1–40, https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

[28] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multiprecision Floating-Point Support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019, https://doi.org/10.1109/TVLSI.2019.2950087.

[29] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini, "A New Ara for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design," in *ASAP*, 2022, pp. 43–51, https://doi.org/10.1109/ASAP54787.2022.00017.

[30] M. Perotti, M. Cavalcante, R. Andri, L. Cavigelli, and L. Benini, "Ara2: Exploring Single- and Multi-Core Vector Processing With an Efficient RVV 1.0 Compliant Open-Source Processor," *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1822–1836, 2024, https://doi.org/10.1109/TC.2024.3388896.

[31] SBMICRO, "Programa APCI: Programa de Apoio a Projeto de Circuitos Integrados em Universidades," 2025, https://sbmicro.org.br/programas/apci.

[32] Wikipedia, "Antenna effect," 2025, https://en.wikipedia.org/wiki/Antenna_effect.