RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems

Willian Analdo Nunes, Angelo Elias Dal Zotto, Caroline da Silva Borges and Fernando Gehm Moraes
PUCRS – School of Technology - Av. Ipiranga 6681, Porto Alegre, Brazil
{willian.nunes, angelo.dalzotto, caroline.s}@edu.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—This paper presents RS5, a hardware and software ecosystem designed to provide efficient RISC-V processing in embedded systems. While there are many RISC-V implementations available in the literature, RS5 distinguishes itself by offering a modular architecture with optional extensions, such as M (multiplication and division) and Xosvm (memory management), tailoring the core to the specific design needs. RS5 includes optional performance counters for application and core behavior monitoring in real-time, and multitasking support through an OS with a custom memory management mechanism. The Memory Management Unit (MMU) is implemented through a custom register set, offering an alternative to the RISC-V Sv32 extension that may not be well-suited for low-area footprint circuits. The main contribution of this work is the RS5 core, which implements the RISC-V RV32I(M) ISA with OS support to manage external peripherals. Results evaluate the performance of RS5 using a set of benchmarks and compare it to other implementations. Results show that RS5 outperforms other implementations regarding area and performance. RS5 is publicly-available at https://github.com/gaph-pucrs/RS5.**

***Index Terms*—RISC-V, performance evaluation, embedded memory management, FPGA prototyping.**

I. Introduction

RISC-V is an Instruction Set Architecture (ISA) governed by a permissive open-source license, allowing its utilization in academic and commercial applications without licensing fees. One of the primary reasons for its rapid popularization is its open standard, which ensures the availability of comprehensive public documentation that details its implementation.

RISC-V is architecturally partitioned into “base” and “extension” modules [1]. A “base module” is a complete ISA, while “extension modules” are specialized sets of instructions categorized based on their functional attributes. Furthermore, the base RISC-V ISA modules exhibit variations, such as instruction length and register bank configurations, offering a range of options tailored to specific application requirements.

The RV**32I** is an integer 32-bit ISA that defines instructions for most general software operations such as arithmetic and logic computations, control flow operations, and memory access. The main ratified extensions are: (*i*) multiplication and division (M); (*ii*) single-precision floating-point (F), double-precision (D) and quad-precision (Q); and (*iii*) atomic-instructions (A) to support coherent memory access. When joined, the RV32IMAFD can be called RV32**G** for general-purpose systems.

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES (Finance Code 001), CNPq (grants 309605/2020-2, 130845/2023-0), and FAPERGS (grant 21/2551-0002047-4).

Zicsr is an extension that provides instructions to execute atomic operations on Control and Status Registers (CSRs), enabling software executing on RISC-V to configure the system behavior. RISC-V also incorporates a privileged architecture definition [2], standardizing the CSR set and defining three main privilege levels at which a RISC-V core can operate, the User- (**U-Mode**), Supervisor- (**S-Mode**), and Machine-mode (**M-Mode**). These privilege modes introduce the capacity for differential trust levels in code execution, thereby enabling the separation of user applications and operating system (OS). The CSR set defined in the privileged architecture is also used in interrupt and exception handling.

Despite the plethora of RISC-V implementations available in the literature, with some examples presented in Section II, the *objective* of this work is to introduce a core that distinguishes itself by providing an integrated ecosystem of both hardware and software components, named **RS5**. The **RS5** core has a modular architecture that includes optional extensions, offering the ability to tailor the core to the specific needs of the target applications. The core also includes optional performance counters to enable real-time application and core behavior monitoring.

The main *contribution* of this work is the **RS5** core, which implements the RISC-V RV32I(M) ISA with OS support, designed to manage external peripherals. This support includes the standardized capability to handle interrupts and exceptions, defined in the RISC-V privileged architecture specification [2], and a memory management extension called **Xosvm**.

II. Related Work

Several open-source designs proposing RISC-V cores exist, and many efforts are underway.

Ibex, formerly known as “Zero-riscy”, was initially developed as an integral component of the Parallel Ultra-Low-Power (PULP) platform, an open-source SoC targeting IoT applications. Ibex is currently maintained by lowRISC, a non-profit organization affiliated with the University of Cambridge Computer Laboratory (https://lowrisc.org/) [3]. The core is a 32-bit implementation that supports the Integer (RV32I) or Embedded (RV32E) ISAs. Additionally, Ibex supports extensions, including M, C (compressed instructions), and B (bit manipulation). The core adheres to the RISC-V privileged architecture specifications with M-Mode and U-Mode. Architecturally, Ibex is characterized by a 2-stage, in-order, single-issue pipeline, specifically designed to minimize power dissipation, with a small silicon area.

The OpenHW Group CORE-V **CV32E40P** (https://github.com/openhwgroup/CV32E40P), previously identified as

"RI5CY" is a 32-bit, in-order processor core featuring a 4-stage pipeline [4, 5]. It supports the M and C extensions, with an optional inclusion of the F extension. The core originates from the OR10N OpenRISC core [6]. Like Ibex, it is also a component of the PULP platform. CV32E40P can include custom extensions, such as hardware loops, Single Instruction Multiple Data (SIMD) operations, and fixed-point arithmetic.

The **SCR1** core [7] is maintained by the Syntacore company from Cambridge (UK). SCR1 is the simplest of seven cores offered by Syntacore (from SCR1 to SCR7) and is the only open-source. It implements the RV32I ISA and the Zicsr extension to support the RISC-V privileged architecture. It also offers the M and C extensions, with the possibility to reduce the register bank, resulting in the RV32E ISA. SCR1 can be configured to have 2 to 4 pipeline stages and offers an optional custom interrupt controller. It is an industry-grade silicon-proven core.

Steel [8] is an academic RV32I_Zicsr core with M-Mode support. It is a 3-stage in-order pipeline designed to allow reusability in RISC-V designs, from embedded projects to complex SoCs. Its differential is a native AXI4-Lite support.

III. Processor Organization

RS5 implements the RV32I_Zicsr_Zicntr instructions set, alongside the M- and U-Mode. Figure 1 presents its organization, where green rectangles separate the four pipeline stages (Section III-A): instruction fetch (**IF**), instruction decode (**ID**), execution unit (**XU**), and retire unit (**WB**). Section III-B details the performance counters (Zicntr and Zihpm). Sections III-C and Section III-D describe the optional extensions, Zmmul/M and Xosvm. Sections III-E and III-F detail the interrupt controller and the Real-Time Clock (RTC).

A. Pipeline Stages

The **IF** stage controls the reception of instructions from the instruction memory (I-MEM). It selects the Program Counter (pc) according to the core reset, trap occurrence, trap return, jumps, and sequential operation. This stage works with virtual memory addresses and is optionally connected to an I-MMU to translate these addresses to physical memory addresses.

The **ID** stage extracts the operation (op) requested by the fetched instruction and the destination register (rd). As the source registers (rs) are already decoded at this stage, it fetches the operands (ops) from the register bank (GPR - general purpose registers) and performs data hazard detection. This stage can identify if an rs of the current instruction is the same as the rd being written by the retirement stage to perform a 1-stage forwarding ($fwd.\ ctl.$), avoiding to wait for the register bank write, thus bypassing the data directly to ops. The data hazard detection emits no-operations ($haz.$), inserting bubbles into the pipeline for the number of cycles necessary to resolve a conflict.

The **XU** stage is responsible for the computation based on op and ops. This stage can have up to five different execution units: (*i*) Arithmetic Logic Unit (ALU): performs arithmetic and logic operations. Its operation result ($ALU\ res.$) can be used to compose the data memory (D-MEM) access $address$ and the $branch\ target$; (*ii*) BRANCH: manages conditional branching ($branch\ taken$); (*iii*) LOAD/STORE: manages D-MEM access ($write\ enable$ and $write\ data$); (*iv*) CSR: executes CSR atomic operations, enforces privileges, and handles traps; and (*v*) Multiplication (MUL) and Division (DIV): the MUL unit can be included independently, while the inclusion of the Division unit implies the inclusion of the Multiplication unit, resulting in the M extension.

Note that memory writing is completed at this stage. The XU also controls the $hold$ signal when an operation result is unavailable as consequence of a multiplication or division operation ($mul./div.\ res.$). A multiplication takes 3 cycles to complete, while a division takes up to 32.

An operation is completed only when it has a valid tag. Both conditional and unconditional jumps are executed at the XU stage. Whenever a branch is taken, the tag is incremented in the IF and XU. This tag also accompanies the instruction as it moves from the IF to the XU stage. If an op arrives at the XU stage with a tag that differs from the one present at this stage, the operation is discarded because it originated from a branch that was not taken.

The **WB** stage executes the write-back of op results. It is responsible for writing data obtained from D-MEM ($mem.\ data$) after a data read was queued from the LOAD/STORE unit. It also controls the $stall$ signal when a memory or peripheral operation is pending. Note that an optional D-MMU can virtualize D-MEM addresses.

B. Performance counters (Zicntr and Zihpm)

RISC-V standardizes performance counters to monitor and measure program execution and system behavior. By analyzing the data collected from these counters, developers can make decisions to improve code efficiency, identify bottlenecks, and enhance overall system performance.

RS5 implements the three counters of the Zicntr extension: (*i*) **cycle**: a counter that is increased at every clock cycle and keeps track of how many cycles have passed since the core boot; (*ii*) **time**: a counter that reports an arbitrary precision time counter that is not affected by clock gating, which is connected to the RTC; and (*iii*) **instret**: a counter that tracks how many instructions were retired by the core. Additionally, RS5 offers **optional** debugging counters accessible through the standardized Zihpm extension. These counters track the number of bubbles issued, context switches, and instructions executed by type, such as branches, arithmetic, or logic.

C. Multiplication and Division Instructions (Zmmul/M)

A hardware-driven approach to execute multiplication and division operations reduces computation time, making it ideal for time-critical tasks such as real-time signal processing, However, it is important to consider trade-offs. Hardware implementations for multiplication and division increase the chip area, design complexity, and costs. In contrast, software-based approaches offer adaptability through modifiable algorithms.

RS5 gives three parameterization options: (*i*) **no hardware for multiplication and division**: any application requiring multiplication or division must include software-based routines; (*ii*) **Zmmul extension**: only integer multiplication operations are implemented with hardware acceleration; (*iii*) **M Extension**: Zmmul extension and integer division operations.

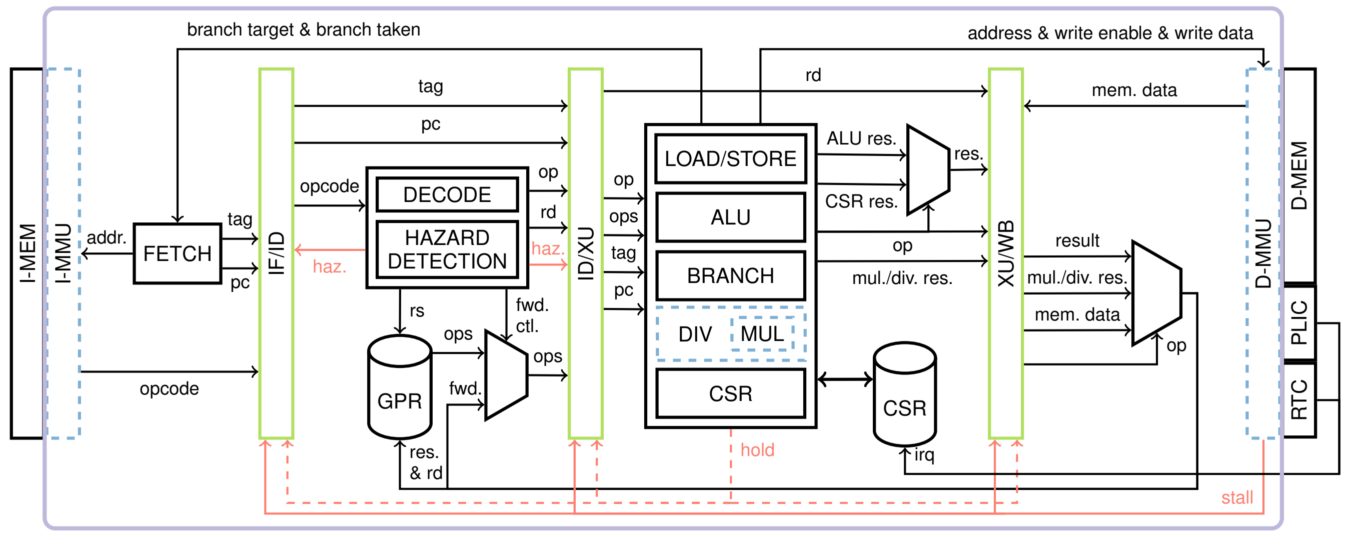

Fig. 1. RS5 Organization. The purple line delimits the core. Components in dashed blue lines are optional. Green components are temporal barriers. Signals that control the pipeline stall are marked in red, and the hold signal is dashed due to only occurring when MUL/DIV is present.

D. MMU (Xosvm)

RS5 includes an optional compact MMU through the **Xosvm** extension ("X": custom extensions, "osvm": *offset and size virtual memory*). This extension enables paged memory organization by dividing the memory into contiguous, statically-sized pages for each process. It was introduced to address two significant issues: firstly, the standard RISC-V Sv32 extension for 32-bit virtual memory systems requires the Supervisor mode (S-Mode), alongside M- and U-Mode, leading to increased hardware overhead with the inclusion of extra CSRs and a Translation Look-aside Buffer (TLB); secondly, while RISC-V may offer a Physical Memory Protection (PMP) extension for security-centric embedded applications, it lacks virtual memory support and incurs area overhead [9]. Thus, the Xosvm extension aims to provide paged memory functionality while minimizing hardware complexity.

The Xosvm extension requires five CSRs that the OS modifies on context switching:

- `mvmio` and `mvmdo`, machine virtual memory instruction/data offset registers. The offset is restricted to power of two values. Consider, for example, all page sizes equal to 4096 bytes (`0x1000`). The offset for the first four user pages are `0x1000`, `0x2000`, `0x3000`, `0x4000`.
- `mvmis` and `mvmds`, machine virtual memory instruction/data size **mask** registers. These registers contain the size mask, i.e., the page size minus one. In the above example, for a page size equal to `0x1000`, `mvmis` and `mvmds` are equal to `0xFFF`.
- `mvmctl`, machine virtual memory control register. Only the least significant bit is writable. This bit enables or disables the Xosvm extension when the core operates in U-Mode. The Xosvm is disabled when the core is in M-Mode. When disabled, either by clearing `mvmctl` or by executing the core in M-Mode, the MMU is bypassed.

Equation $pma = vma \vee offset$ describes how to obtain the Physical Memory Address (pma) from the Virtual Memory Address (vma), i.e., the address generated by a code running in the processor. The *or* operation is equivalent to adding the page offset to the vma, but with a smaller hardware cost.

In the context of the Xosvm extension, memory isolation between pages is ensured, thus safeguarding the contents stored on each page. This is achieved through the equation $exception = (vma \wedge \neg size) \neq 0$. If the result is different from zero, an exception is raised to the processor core.

E. Interrupt control (PLIC)

RS5 adopts an interrupt treatment approach based on a proposal by SiFive [10]. Two types of interrupt controllers are discussed: the Core Local Interrupt Controller (CLIC) and the Platform Level Interrupt Controller (PLIC). CLIC is primarily designed to manage interrupts for multiple hardware threads within the same core, multiplexing timer interrupts, and managing software interrupts. Given that RS5 is a single-thread core, CLIC does not apply to this work. PLIC manages global or external interrupts generated by peripheral devices. Within PLIC, each interrupt can be assigned a configurable priority level ranging from 1 to 7 (highest priority), and 0 disables the interrupt.

RS5 supports timer interrupts through RTC and external interrupts managed by PLIC (irq). PLIC is configurable and provides an interface to Memory-Mapped Registers (MMR).

F. Real-Time Clock (RTC)

RISC-V standardizes RTC access through MMRs. RS5 implements a 64-bit cycle counter that is accessible through MMR and also routed to the **time** counter in Zicntr. The OS uses the RTC to generate timer interrupts to control the scheduling time slice.

IV. Software Stack

RS5 supports a complete software stack with an embedded OS that supports dynamic application loading, called **MAestro** [11]. MAestro supports multitasking through Xosvm and preemptive scheduling through timer interrupts provided by the RTC. The built-in schedulers support best-effort tasks through round-robin, and soft real-time tasks through Least Slack Time

(LST). The kernel is built on top of newlib-nano [12], and provides 6 POSIX system calls to handle standard output, dynamic memory allocation, and process control.

Additionally, MAestro is ready with drivers and a protocol stack to support communication through Network-on-Chip (NoC). If targeting many-core systems, MAestro can also include user-level applications to manage the system. Any user-level application can be leveraged by the OS support library built on top of newlib-nano.

V. Experimental Setup and Results

Experiments operate in two environments: (*i*) simulation, which uses a testbench to emulate peripherals and memory, offering debugging and fast validation, and (*ii*) FPGA prototyping, which validates the core in a real-world scenario.

In the FPGA prototyping, the setup employs a Digilent NEXYS A7 board with a Xilinx xc7a100tcsg324-1 FPGA. The main memory uses Block RAM (BRAM), a high-density memory module in Xilinx FPGAs. The GPR uses Lookup Table RAM (LUTRAM), a Xilinx FPGA primitive device that is both area and power-efficient. However, due to the granularity and atomicity limitations of LUTRAMs, the CSR bank is implemented with flip-flops.

The core uses a byte-addressed true dual-port memory, supporting synchronous reads and writes. The initialization of this memory comes from a binary file with the software. For communication, the setup incorporates a UART module, and the board's built-in buttons serve to emulate external interrupts.

Compliance evaluation with the RISC-V ISA occurs through unit tests rv32ui and rv32um [13] and performance assessment through the EEMBC Coremark benchmark [14]. Results were obtained with GCC 12, optimization flags `-O3` and `-DPERFORMANCE_RUN=1`.

A. Results

Table I presents results related to RS5 and cores cited in Section II. Columns 2-4 present the usage of FPGA resources (LUTs, FFs, and DSPs). The 5th to 7th columns present the supported extensions (all cores have the Zicsr extension), the operation modes, and the maximum achieved frequency by the synthesis tool in the target FPGA. The last column of the Table lists the CoreMark scores [15], corresponding to the performance of the core on the same FPGA, with each core running at the obtained frequency. The upper part of the Table presents the cores using LUTRAM for the register bank, and the lower part of the Table presents the cores using FFs to implement the register bank.

Compared to the baseline RS5, the area overhead (LUTs/FFs) due to the extensions are: (*i*) **Zihpm** (D^*): 11%/53%, mainly due to additional CSRs; (*ii*) **Xosvm** (X^*): 19%/8%, due to new CSRs and MMU logic. (*iii*) **Zmmul**: 4%/-12%, with 12 DSPs modules for the integer multiplication; (*iv*) **M**: 31%/16%, due to integer multiplication and division operations. Including all extensions, the area overhead w.r.t baseline core is 67%/84%.

The Zmmul and M extensions yield comparable performance (212.3) in applications that rely heavily on multiplication. Both extensions execute 18,992 multiplication instructions. The performance similarities between Zmmul and M extensions are attributed to the relatively few division operations (2,210) compared to multiplication operations in the CoreMark benchmark.

TABLE I
RS5 Evaluation and Comparison with Similar Cores
(Z^*:Zicntr, D^*: Zihpm, X^*: Xosvm).

Core	LUTs	FFs	DSPs	Extensions	Modes	Freq. MHz	CoreMark
RS5 Baseline	**2141**	**957**	-	Z^*	M, U	100	86.3
RS5	2380	1466	-	Z^*, D^*	M, U	100	86.3
	2553	1032	-	Z^*, X^*	M, U	100	86.3
	2222	842	12	Z^*, Zmmul	M, U	100	212.3
	2814	1113	12	Z^*, M	M, U	100	212.3
	3574	**1757**	12	Z^*, M, X^*, D^*	M, U	100	212.3
Ibex	2184	1247	-	Z^*, C	M, U	50	46.8
	2688	1329	1	Z^*, M, C	M, U	50	111.6
Steel	2140	1434	-	Z^*	M	50	68.0
RS5 - no LUTRAM	2721	1949	-	Z^*	M, U	100	86.3
	3395	2105	12	Z^*, M	M, U	100	212.3
SCR1	2938	1617	-	-	M	66	70.4
	3518	1747	4	M	M	50	114.3
CV32E40P	5111	2015	5	Z^*, M, C	M, U	70	186.8

The baseline Ibex uses 2%/30% more LUTs/FFs than the baseline RS5, but Ibex implements the C extension. In terms of performance, Ibex achieves 54.2% (baseline) and 52.6% (M) of RS5 CoreMark scores due to its half-operating frequency. Ibex nearly doubles its area when implementing 16 PMP regions [9] (not in the Table), whereas Xosvm overhead is considerably lower ($< 20\%$).

Despite showing similar LUT utilization, Steel uses 50% more FFs w.r.t. RS5 baseline. Note that Steel does not support the U-mode. RS5 performs 27% faster than Steel but operates at 100 MHz instead of 50 MHz.

The absence of the LUTRAM register bank significantly increases resource utilization. RS5 without LUTRAM consumes 27%/104% more LUTs/FFs than its counterpart.

The baseline SCR1 uses 8% more LUTS and 17% less FFs than the baseline RS5 without LUTRAMs, with a CoreMark 18% smaller. The lower usage of FFs in SCR1 is explained mainly by the absence of the Zicntr extension and U-Mode. Enabling the M extension decreases the SCR1 frequency, while resource usage is 4%/-17% in LUTs/FFs than RS5 (no LUTRAM). In terms of performance, SCR1 achieves 80.4% (baseline - 66 MHz) and 50% (M extension - 50 MHz) of the RS5 CoreMark scores.

CV32E40P, despite its higher resource utilization, operates at 70MHz, with a performance only 12% smaller than RS5(M). This result points to a design optimized for performance, since its original design targets ASICs.

VI. Conclusion and Future Work

RS5 offers a modular architecture with optional extensions, performance counters, and OS support. We evaluated RS5's performance against similar RISC-V implementations with RS5 presenting the highest CoreMark performance. The custom Xosvm extension is a low-cost alternative for multitasking with memory protection.

As future work, we plan to explore additional extensions, such as C and bit manipulation, to enhance its performance and versatility. Another target for improvement is the hardware acceleration of the division operation.

References

[1] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," Berkeley, Tech. Rep. 20191213, Dec. 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[2] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture," Berkeley, Tech. Rep. 20211203, Dec. 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[3] LowRISC C.I.C., "Ibex: An Embedded 32-bit RISC-V CPU core," May 2021. [Online]. Available: https://ibex-core.readthedocs.io

[4] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *IEEE International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[5] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[6] M. Gautschi, M. Muehlberghuber, A. Traber, S. Stucki, M. Baer, R. Andri, L. Benini, B. Muheim, and H. Kaeslin, "SIR10US: A tightly coupled elliptic-curve cryptography co-processor for the OpenRISC," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2014, pp. 25–29.

[7] Syntacore, "SCR1 Microcontroller Core," Apr. 2022. [Online]. Available: https://syntacore.com/page/products/processor-ip/scr1

[8] R. da Silva, V. dos Santos, F. Petkowicz, R. Calçada, and R. Reis, "Synthesis of Steel-ASIC, a RISC-V Core," *Journal of Integrated Circuits and Systems*, vol. 17, no. 2, pp. 1–8, 2022.

[9] lowRISC, "Ibex RISC-V Core," 2017. [Online]. Available: https://github.com/lowRISC/ibex

[10] SiFive, Inc, *SiFive Interrupt Cookbook, Version 1.2*, Feb. 2020. [Online]. Available: https://www.starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf

[11] A. E. Dalzotto, M. Ruaro, L. V. Erthal, and F. G. Moraes, "Management Application-a New Approach to Control Many-Core Systems," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2021, pp. 1–6.

[12] Red Hat, "newlib," 2021. [Online]. Available: https://sourceware.org/newlib/

[13] RISC-V Foundation, "Unit Tests for RISC-V Processors," 2015. [Online]. Available: https://github.com/riscv/riscv-tests

[14] EDN Embedded Microprocessor Benchmark Consortium, "CoreMark - An EEMBC Benchmark," 2023. [Online]. Available: https://www.eembc.org/coremark/index.php

[15] ——, "CoreMark - Scores," 2023. [Online]. Available: https://www.eembc.org/coremark/scores.php