# Prototyping custom RISC-V instructions with Seal5 and CoreDSL*

Jan Schlamelcher[1], Thomas Goodfellow[1], Bewoayia Kebianyor[1], Gregor Nitsche[1], Philipp van Kempen[2] and Kim Grüttner[1]

[1]German Aerospace Center - Institute of Systems Engineering for Future Mobility
[2]Electronic Design Automation, Technical University of Munich

**Abstract**

*Seal5 provides the efficient transformation of custom* RISC-V *instructions defined in CoreDSL into the LLVM toolchain and the ETISS instruction set simulator. This paper evaluates this approach by implementing an extension for the ChaCha20 cipher, achieving a substantial performance improvement in the test case without any change to its source code and a tiny increase in core size. The results demonstrate the potential of this approach for rapid exploration of customising a RISC-V-based product through extension instructions.*

## Introduction

Edge computing has increased the importance of quickly adapting architectures for diverse properties such as low-latency, low-power, high-throughput, digital sovereignty, reduced infrastructure costs, and real-time decision-making, in applications ranging from low-cost I.o.T. devices to autonomous systems. Because the RISC-V architecture is explicitly based upon composing the ISA from both standard and custom instruction sets it is perfectly suited for rapid prototyping of edge computing hardware. In contrast, the existing compiler toolchains lack convenient mechanisms for incorporating custom extensions.

This paper focuses on exploiting that RISC-V extensibility by using Seal5 for rapidly exploring the merits of using custom instructions. An extensible compiler toolchain for RISC-V is presented, based upon the already well-accepted LLVM, which is a rather more capable evolution of what was earlier introduced [1]. Instructions to accelerate a cryptographic cipher are presented with the improvements in code size and speed achieved in a test application, which illustrates that this method provides a productive and fruitful development pathway.

## Background

The approach taken in this work uses a high-level description of an extension ISA to automatically modify a compiler toolchain to both provide easy access to those instructions expose and also to automatically exploit them. Because the modification process is quick it allows for easy iteration of the ISA design, enabling cheaper and deeper design exploration.

The extension ISA is described in CoreDSL [2], a domain-specific language for modeling processor cores with 'C'-like syntax. This is transformed by the Seal5 [3] tooling into patches for both the LLVM toolchain [4] (including the clang compiler) and the ETISS [5] instruction set simulator. Having an ISS immediately available for an extension enables both validation of that it functions correctly and provides insight into its actual merits.

A previous implementation of this approach [6] had the limitation that the instructions added by an extension were supported for direct invocation from compiler intrinsic functions or assembly language. The Seal5 framework also generates selection patterns, enabling the compiler to automatically exploit the instructions.

## Methods

The tooling described in the previous section has already been detailed in [1], however the 64-bit MAC used as the sample extension in [6] provided a rather limited demonstration of the approach. The instructions provided did not fit the common RISC-V patterns, potentially increasing the cost of implementation on a core and restricting its use to code specifically written for it. For real applications users not needing a 64-bit MAC on a 32-bit platform would prefer the pre-existing xcorev.mac instruction.

A more motivating example is found with instructions to accelerate the ChaCha20 cipher[1][8]. ChaCha20 generates the blocks of pseudo-random bytes that are used in ciphering operations by repeatedly shuffling the contents of a 64-byte initialization vector (IV) with ADD, XOR, and bitwise-rotation in-

*This work has been developed in the ZuSE project Scale4Edge. Scale4Edge is funded by the German ministry of education and research (BMBF) (reference number 16ME0465).

[1] This use case was inspired by earlier discussions with Imperas, who used it as an sample in [7]

structions . It provides scope for a small extension to have a meaningful impact and moreover is not already directly supported through the RISC-V cryptography extensions.

The central transformation of ChaCha20 is the "quarter round", detailed here in 'C'-like pseudo-code where <<< denotes a leftward bit rotation and the variables are 32-bit words from specific locations in the IV:

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<=  8;
c += d; b ^= c; b <<<=  7;
```

The pairing of XOR and a fixed-length rotation presents the possibility of combining them into custom operations, as seen written in a fragment of CoreDSL for the first such pairing:

```
if (rd != 0) {
  unsigned<32> xor = X[rs1] ^ X[rs2];
  X[rd] = (xor << 16) | (xor >> 16);
}
```

The remaining three pairings were described by similar CoreDSL. The LLVM toolchain and ETISS were then built with support for the "xchacha20" extension.

A straightforward implementation of the ChaCha20 quarter_round() written in 'C' was used to evaluate both the correctness and performance of the resulting code. It was exercised through the complete ChaCha20 transformation of 80 quarter rounds, confirming its correct function by generating a defined test sequence. When compiled for a simple rv32i core the quarter_round() function executed 1847 instructions, whereas when built for the ChaCha20 extension only 887 instructions were executed.

As a comparison quarter_round() were also implemented by explicitly invoking the extension instructions as intrinsic functions. This also executed 887 instructions, demonstrating that the LLVM selection patterns generated by Seal5 enabled clang to optimally exploit the extension.

Although the focus of this work has been software simulation a single experiment of integrating the ChaCha20 extension into a core was performed. This used the Longnail High Level Synthesis tooling [9] to generate a hardware module from the same CoreDSL specification, before integrating that into a 4-stage version of the VexRISCV core. It was observed that the extension only makes up approximately 1% of the whole core area, which gives feedback on the cost/benefit considerations.

For rapid development it is important that the tools are responsive. This work was performed on a modest 32-core server, where, after the initial configuration and build, subsequent rebuilds took 5-10 minutes. The hardware synthesis flow added less than 5 minutes.

# Discussion

The experimental results demonstrate the potential of Seal5 to exploit custom RISC-V extensions with little effort on the software developer's part. In a well-suited case such as the ChaCha20 extension presented here the performance is gained with no source code change; in other cases the presence of intrinsic functions and integrated assembly reduces the developer burden.

In conclusion, this work highlights the importance of RISC-V's extensibility in modern edge computing design spaces, with a focus on hardware-software co-design and compiler-supported extensions. This may be of particular value in edge computing where a multitude of designs optimized for specific applications is expected.

# References


[1] Jan Schlamelcher et al. "Extending Clang/LLVM with Custom Instructions using TableGen – An Experience Report". In: *MBMV 2024; 27. Workshop*. 2024, pp. 204–213.

[2] Minres. *CoreDSL 2 Programmer's Manual*. https://github.com/Minres/CoreDSL/wiki/CoreDSL-2-programmer's-manual. Accessed: February 7, 2025.

[3] Philipp Van Kempen et al. "Seal5: Semi-Automated LLVM Support for RISC-V ISA Extensions Including Autovectorization". In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. 2024, pp. 335–342. DOI: 10.1109/DSD64264.2024.00052.

[4] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[5] Daniel Mueller-Gritschneder et al. "ETISS-ML: A multi-level instruction set simulator with RTL-level fault injection support for the evaluation of cross-layer resiliency techniques". In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. IEEE, 2018, pp. 609–612. ISBN: 978-3-9819263-0-9. DOI: 10.23919/DATE.2018.8342081. URL: https://doi.org/10.23919/DATE.2018.8342081.

[6] DLR Scale4Edge Team. *Extensible Compiler*. https://github.com/DLR-SE/extensible-compiler. Accessed: February 7, 2025.

[7] Imperas Software Limited. *Imperas RISCV Custom Instruction Flow Application Note*. https://www.ovpworld.org/documents/Imperas_RISCV_Custom_Instruction_Flow_Application_Note.pdf. Accessed: February 7, 2025. 2020.

[8] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. June 2018. DOI: 10.17487/RFC8439. URL: https://www.rfc-editor.org/info/rfc8439.

[9] Julian Oppermann et al. “Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS ’24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 591–606. ISBN: 9798400703867. DOI: 10.1145/3620666.3651375. URL: https://doi.org/10.1145/3620666.3651375.