Integrated development environment for teaching and research on RISC-V processors (PDC2023-145832-I00)

Integrated development environment for teaching and research on RISC-V processors

CREATOR

D 1.1
Report on the design of the new instruction definition model and compiler

Universidad Carlos III de Madrid

January, 2026

CONTENTS

1. NEW COMPILER

1.1. New Compiler based on Rust

The compiler developed in Rust consists of four main components:

- Architecture manager: responsible for validating and loading the ISA to be used during compilation.
- Error renderer: responsible for transforming information related to an error into a format that can be displayed to a user.
- Syntactic analyzer: responsible for performing syntactic analysis of the assembly code to enable its analysis and translation.
- Semantic analyzer: responsible for performing semantic analysis of the assembly code and translating it into a format that can be used by CREATOR.

Figure 4.1 shows the UML component model of the compiler, including the relationships between the components.

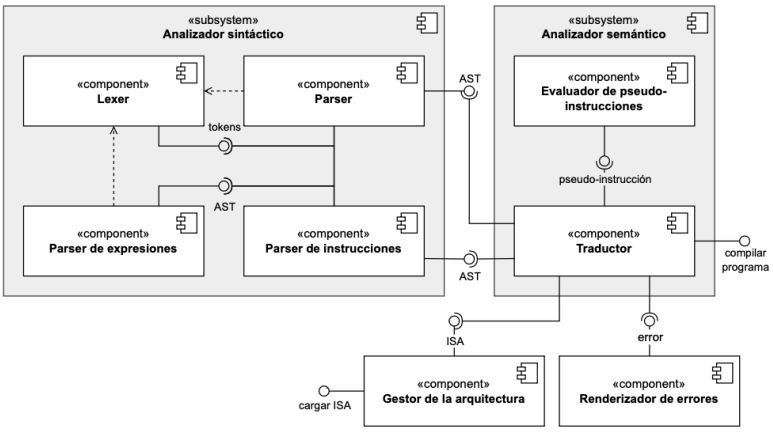

Figure 1.1.1: Compiler Structure

Next figures show example os errors analyzed by the new compiler.

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 1.s -o min --color
[E11] Error: Label repeated is already defined
     ┌─[ assembly:10:1 ]
     │
   3 │ repeated: li t0, 0
     │ ────┬─────
     │     └──────── Note: Label also defined here
     │
  10 │ repeated: add t2, t0, t1
     │ ────┬─────
     │     └──────── Duplicate label
     │
     │ Help: Consider renaming either of the labels
─────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

Figure 1.1.2: Error example 1

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 3.s -o min --color
[E22] Error: Value 65536 is outside of the valid range of the field
    ┌─[ assembly:2:7 ]
    │
  2 │ .half 0x10000
    │       ──┬──
    │         └──── This expression has value 65536
    │
    │ Note: Allowed range is [-32768, 65535]
────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

Figure 1.1.3: Error example 2

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 5.s -o min --color
[E02] Error: Instruction addo isn't defined
    ┌─[ assembly:7:7 ]
    │
  7 │       addo t0, t0, 1
    │       ──┬─
    │         └──── Unknown instruction
    │
    │ Help: Did you mean add or addi?
────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

Figure 1.1.4: Error example 3

2. NEW MODEL BASED ON SAIL SPECIFICATION

This section will address the development of a new RISC-V simulator based on the Sail instruction semantics specification language. This work is also published in an article presented at the SARTECO 2025 Conference [1].

2.1. Sail model language to simulator implementation.

Traditionally, the design and implementation of architectures had been described in natural-language documents, but these mechanisms led to ambiguities and problems during verification. This situation evolved and used high-level programming languages like C or Python, which allowed us to prove the architecture's design. However, these languages still do not have a formal verification process to check semantic correctness, instruction behavior, security against weaknesses, etc.

Given this need to formally verify the design, a language was developed that enables such verification: Sail. RISC-V adopted this language to design and verify the instruction set architecture that they were developing. Verification helps avoid errors such as Intel's $FDIV$, which cost 500$ million in 1995 [2].

Sail is described as a specification language of instruction set architectures. Its behavior is similar to a framework. It also gives developers the ability to implement instruction structure, their behavior, encoding and decoding, the hardware resources involved, and the processor architecture. The main feature of this language is that it is not compilable; in other words, it does not allow the generation of a simulator that can be executed. In Sail, the syntax, behavior, and data types are checked to ensure they do not introduce ambiguity or inconsistency during execution.

To generate the simulator, Sail allows exporting the specification to a high-level language and building it to simulate (in C or OCaml). It also allows the implementation to build mathematical models to prove in tools. For example, *Isabelle* [3], *HOL4* [4], or *Coq* [5]. These tools enable formal verification through mathematical and logical demonstrations of the behavior of emulated software and hardware, and of the security against weaknesses that could be exploited by attackers.

This language has been used to specify the RISC-V instruction set architecture based on the specification implemented by the organization[1]. This definition has been used as a basis for developing the new simulator. The repository contains the implementation of the whole RISC-V instruction set, and some of them meet the project requirements. Highlighting system calls that, while maintaining the standard definition, have been updated for integration into the simulator web, including a simple, intuitive debugging mode to facilitate this task.

The main reason for choosing this solution is that JavaScript is not a suitable language for running applications in web environments, whereas WebAssembly is, as it improves execution performance by an average of 30% and better manages browser resources [6]. This tool is presented as a complement to JavaScript, not a replacement [7], as JavaScript is required to execute its modules. However, in terms of the execution performance of native applications in web environments, WebAssembly offers better performance, in some situations achieving

[1] **Available in** `https://github.com/riscv/sail-riscv`

performance close to that of a native application [8].

Regarding the simulator and the language used to specify the architecture and the instruction set of RISC-V, the specification was made with Sail of a basis RISC-V instruction (*add rd, rs1, rs2*) extracted from the repository that has been used as the basis for the specification of the architecture and instruction set is attached.

First, it is necessary to describe how an instruction is encoded. An instruction contains the code of operation that shows how it should behave when it is executed and the elements affected by that instruction.

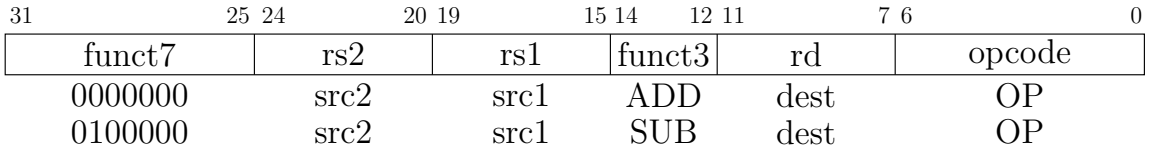

Figure 2.1.1: Encoding structure of an instruction.

The operation code generally consists of 2 or 3 fields as shown in Figure 2.1.1. These fields specify the type of instruction operation (operations with registers, branch operations, operations with immediates, etc.) and the exact operation to be performed once the type of instruction has been specified (add, subtract, bit shift with registers, etc.). Two fields are primarily needed to specify the type and behavior of an instruction. However, if more fields are required to represent and identify more instructions of a type due to size limitations, an additional field is added to the coding to compensate for the lack of values. It should be noted that each instruction has a unique operation code to differentiate it from the rest and avoid ambiguities during execution. The rest of the fields encoded in the instruction are related to the resources needed to execute the instruction. These fields can be registers, immediate values, and absolute or relative memory addresses.

The behavior specification of an instruction with Sail syntax is shown in Figure 2.1.2.

```
enum rop = {RISCV_ADD, RISCV_SUB, RISCV_SLL,
            RISCV_SLT, RISCV_SLTU, RISCV_XOR,
            RISCV_SRL, RISCV_SRA, RISCV_OR,
            RISCV_AND}

union clause ast = RTYPE : (regidx, regidx, regidx, rop)

mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)  <->
    0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SUB)  <->
    0b0100000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011
```

Figure 2.1.2: Definition of type and encoding structure of an instruction.

This fragment of code first defines the operation that is inside of a type of instruction (*rop*), and with that it can define the structure that will have this type of instruction (*RTYPE*). Specifically, *RTYPE* are instructions with registers, so in order to execute them, it is necessary to identify three registers (*regidx*), two as operands and one to store the result of the operation, and the operation code *rop*, which will indicate how the instruction should behave. For simplicity, this can be likened to an abstract syntax tree (AST).

To differentiate instructions of the same type and make them identifiable to the simulator, a mapping is established between the decoded and coded functions, making it easier for the simulator to execute the instruction that arrives in binary.

```
function clause execute (RTYPE(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ADD  => rs1_val + rs2_val,
    RISCV_SUB  => rs1_val - rs2_val
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

Figure 2.1.3: Definition of instruction behavior.

Once the simulator can decode the instruction and identify which values correspond, it can implement the behavior of different *RTYPE* instructions in function of the value that contains *rop*. Therefore, first the values to be operated are extracted from the registers (*rs1, rs2*), the operation is identified (*match op*), it is executed and the result is stored in the destination register (*rd*), as can be seen in Figure 2.1.3.

Finally, the relationships between dissembled instructions (*add, sub*, etc.) and assembled instructions are defined for greater completeness and verification of the instruction specification, as shown in Figure 2.1.4. This process is repeated for the rest of the RISC-V instruction set and for the definition of the simulated hardware resources of the processor architecture.

```
mapping rtype_mnemonic : rop <-> string = {
  RISCV_ADD  <-> "add",
  RISCV_SUB  <-> "sub"
}
mapping clause assembly = RTYPE(rs2, rs1, rd, op)
  <-> rtype_mnemonic(op) ^ spc() ^ reg_name(rd) ^ sep() ^ reg_name(rs1)
      ^ sep() ^ reg_name(rs2)
```

Figure 2.1.4: Assembling and disassembling the instruction.

Once the architecture has been implemented and verified, it is exported to Cm, which will build the web simulator to be integrated into CREATOR. During development, a minimal kernel layer was implemented. The main objective is to provide development support to make the simulator easier to use. However, it is possible to implement a custom kernel for personalized management and adaptation to the developer's needs directly within the CREATOR interface using assembly language.

This simulator is generated from the Sail-to-C implementation exported from Sail, which supports generating executable applications in web environments with the same performance as a native application and without the need for major modifications to the implementation, as mentioned above [8].

2.2. Vector instructions and 64-bit support.

This specification language has enabled expanding the set of instructions to be simulated in CREATOR, as well as the variants of the RISC-V architecture to be simulated (currently specified in the standard as 32-bit and 64-bit). CREATOR had a reduced instruction set compared to the standard. Now, the simulator contains the complete RISC-V instruction set specification.

This set of instructions features the integration of vector, compressed, and privileged instruction simulation. This also increases the architecture's simulation capacity, including the vector file register and the control and status file register, which can be interacted with via the assembly language instructions associated with these components. It should be noted that although support is provided for executing the complete set of RISC-V instructions, the extension for executing atomic instructions can be used even though the architecture is not currently designed to apply parallelism in program execution, since it is designed for a single core. However, atomic instructions allow execution even without applying parallelism.

On the other hand, the integration of the 64-bit variant of the RISC-V architecture stands out, enabling the extension of simulation capacity in both components and the instruction set. This new architecture allows working with longer values (64 bits) and greater memory space due to the increased word size. However, in both 32-bit and 64-bit architectures, instruction encoding remains 32 bits.

BIBLIOGRAPHY

[1] J. C. Cano Resa, F. Garcia Carballeira, D. Camarmas Alonso, and A. Calderon Mateos, "Simulador web para risc-v basado en la especificación sail," in *Avances en Arquitectura y Tecnología de Computadores. Actas de las Jornadas SARTECO*, (Sevilla, Spain), Zenodo, Jun. 2025, pp. 367–376. DOI: 10.5281/zenodo.15773218. [Online]. Available: https://doi.org/10.5281/zenodo.15773218.

[2] J. Harrison, "Verification: Industrial applications notes to accompany lectures at 2003 marktoberdorf summer school," in *Conference: First International IMEKO TC6 Conference on Metrology and Digital Transformation*, 2003.

[3] Isabelle, *Isabelle/hol overview*, https://isabelle.in.tum.de/overview.html. Accessed on 7-03-2025. [Online], 2025. Accessed: Mar. 7, 2025. [Online]. Available: https://isabelle.in.tum.de/overview.html.

[4] C. U. o. T. Australian National University, *HOL4*, https://hol-theorem-prover.org/. Accessed on 7-03-2025. [Online], 2025. Accessed: Mar. 7, 2025. [Online]. Available: https://hol-theorem-prover.org/.

[5] C. P.-M. Thierry Coquand Gérard Huet, *Coq*, https://coq.inria.fr/. Accessed on 7-03-2025. [Online], 2025. Accessed: Mar. 7, 2025. [Online]. Available: https://coq.inria.fr/.

[6] J. De Macedo, R. Abreu, R. Pereira, and J. Saraiva, "Webassembly versus javascript: Energy and runtime performance," in *2022 International Conference on ICT for Sustainability (ICT4S)*, IEEE, 2022, pp. 24–34.

[7] W. W. W. Consortium, *Webassembly*, https://webassembly.org/docs/faq/. Accessed on 7-03-2025. [Online], 2025. Accessed: Mar. 7, 2025. [Online]. Available: https://webassembly.org/docs/faq/.

[8] B. Spies and M. Mock, "An evaluation of webAssembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10. DOI: 10.1109/CLEI53233.2021.9640153.