JOHANNES KEPLER
UNIVERSITY LINZ

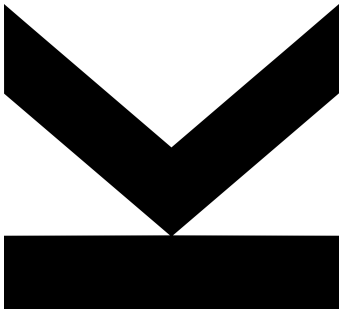
Author
Tobias Kathan

Submission
Institute of Complex Systems

Thesis Supervisor
Univ.-Prof. Dr. Daniel Große

September 2025

A WEB-BASED INSTRUCTION DECODER WITH SEMANTIC EXPLANATIONS FOR RISC-V



Bachelor's Thesis
to confer the academic degree of
Bachelor of Science
in the Bachelor's program
Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

This thesis investigates how binary instructions of the RISC-V instruction set architecture can be made more accessible by combining decoding with human-readable explanations. Building on an existing configuration-driven decoder, this work enriches its output with semantic descriptions sourced from external repositories and integrates the results into a browser-based frontend. The extended system translates raw instruction words into both assembly code and explanatory text, presented interactively via a lightweight web interface. A Rust-based implementation compiled to WebAssembly ensures efficiency and portability across platforms. The resulting tool demonstrates that semantic enrichment improves usability for both expert analysis and teaching scenarios, showing how technical details of instruction sets can be conveyed in a structured and comprehensible way.

Contents

1 Introduction 3
2 Background 4
2.1 RISC-V 4
2.1.1 Base Integer Instruction Set 4
2.2 TOML 7
2.3 Rust Programming Language 7
2.4 Web Assembly 8
2.5 Web Technologies 8
3 Architecture 10
3.1 General Approach 10
3.2 Configuration File Structure 11
3.3 Decoding Process 15
3.4 Evaluation 17
4 Instruction Explanations and Implementation 20
4.1 Configuration Files Adaptation 20
4.2 Instruction Decoder Extension 21
4.3 WebAssembly Bridge 24
4.4 Frontend 25
4.4.1 Integration and Flow 25
4.4.2 Logic and interaction 26
4.4.3 User interface 27
5 Conclusion 31
Bibliography 32

1 Introduction

Instruction decoding is a fundamental task in computer architecture. It transforms raw binary instruction words into meaningful components such as opcodes, registers and immediate values. This process enables both processors and analysis tools to interpret the intended operation of a program. Decoding is therefore essential in a variety of contexts, including debugging, performance monitoring, security analysis and sandboxing [1].

Conventional decoders, however, often provide only a minimal output in the form of an assembly string, for example `addi t1, t2, 10`. While precise, such representations places the burden on users to consult separate documentation to understand the semantics of the instruction. This reduces the efficiency of instruction-level analysis and makes it less approachable for students and practitioners who are new to the architecture.

The present thesis addresses this limitation by extending an existing instruction decoder developed at the Institute of Complex Systems, Johannes Kepler University Linz[1], which is also used as part of the open-source waveform viewer *Surfer* [2]. The baseline decoder follows a configuration-driven design. Instruction set definitions are stored in external files, which makes the tool adaptable to custom ISAs without changes to the core implementation. Building on this foundation, the work introduces an enriched decoding output that supplements the assembly string with a concise description. The result is a structured object that not only encodes the operation in its symbolic form but also conveys its meaning.

To demonstrate the practical value of this approach, the extended decoder is integrated into a browser-based frontend. Users can enter binary, decimal, or hexadecimal instructions and immediately obtain both the assembly representation and an explanatory description. This integration highlights how semantic information can be embedded directly in the decoding pipeline and made accessible through an interactive interface.

[1] https://github.com/ics-jku/instruction-decoder

2 Background

This chapter provides an overview of the RISC-V Instruction Set Architecture, the Rust programming language and the TOML file format. Additionally, it covers WebAssembly (WASM) and the web interface technologies used in this project. These technologies form the foundation for understanding the design and implementation of the instruction decoder and its integration into the web-based application.

2.1 RISC-V

RISC-V is an open *Instruction Set Architecture* (ISA) that was originally developed at the University of California, Berkeley, with the explicit goal of providing a simple, modular and extensible foundation for both academic research and industrial use [3, 4]. In contrast to proprietary ISAs, which are typically restricted by patents and licensing, RISC-V is available under open licenses, making it freely accessible for implementation and extension. This openness has significantly contributed to its adoption across domains ranging from embedded to high-performance systems.

A defining characteristic of RISC-V is its modularity. The architecture specifies a small mandatory base instruction set (`RV32I` or `RV64I`), which provides integer arithmetic, memory access and control flow. The number, `32` or `64`, describes the width of the integer registers and the corresponding size of the address space. Additional features are provided through standardized extensions such as multiplication and division (`M`), atomic operations (`A`) or single-precision floating point (`F`) [5, 6]. System architects can include only the extensions relevant to an application, allowing RISC-V to scale well.

The clean encoding of RISC-V instructions is particularly relevant for instruction decoders. Every instruction is defined by a fixed set of fields, including opcode, registers and immediates. From these fields one can derive a mask and match value that uniquely identifies an instruction. These two numbers are sufficient to check whether a given binary word belongs to a specific instruction, a concept that forms the basis for the configuration-driven decoder described in later chapters.

2.1.1 Base Integer Instruction Set

The base integer instruction set defines the minimal core of RISC-V. It consists of several instruction formats, each of which describes how the instruction word is divided into fields [5]. Two common formats are the R-type and I-type, illustrated in Table 1.

31-25	24-20	19-15	14-12	11-7	6-0	Type
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type

Table 1: Encoding of R-type and I-type instructions in RISC-V

The `opcode` field specifies the general instruction class, while the `funct3` and `funct7` fields disambiguate individual operations within that class. The `rd`, `rs1` and `rs2` fields identify registers. For I-type instructions, the `funct7` and second source register `rs2` are replaced by an immediate value, enabling operations such as adding a constant value to a register.

To illustrate how this structure relates to decoding, consider the binary sequence of an instruction `0b00000000101000111000001100010011` as shown in Figure 1 below. Decoding these fields yield the instruction `addi t1, t2, 10`.

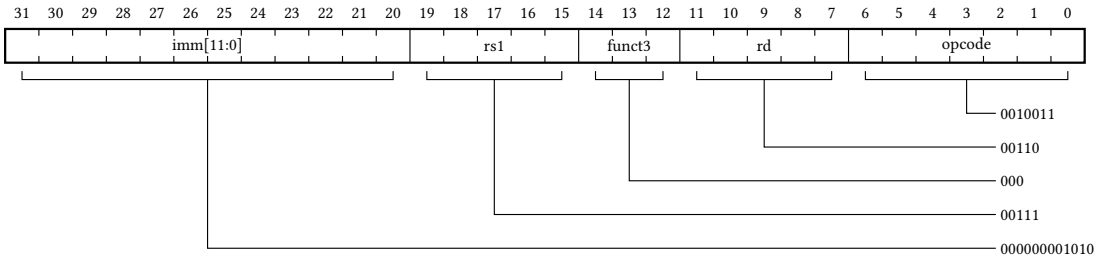


Figure 1: Encoding for `ADDI`

The instruction can be broken down into the following fields. The opcode `0010011`, together with the `funct3` field `000`, identifies the instruction as `ADDI`, an I-type arithmetic operation. The immediate value is represented by the bits `000000001010`, corresponding to the decimal number 10. The source register `rs1` is encoded as `00111`, referring to register `t2` and the destination register `rd` is encoded as `00110`, corresponding to register `t1`.

This layout can be used to derive a mask-match pair from the official RISC-V repository[2]. The mask specifies which bits of the instruction must be compared and the match encodes the expected pattern. For the `ADDI` instruction, the decisive fields are the `opcode` and `funct3` and all other fields are irrelevant and therefore set to zero in the mask. The bit pattern `0b00000000000000111000001111111` shown in Figure 2 corresponds to the hexadecimal representation `0x707f`.

[2]https://github.com/riscv/riscv-opcodes

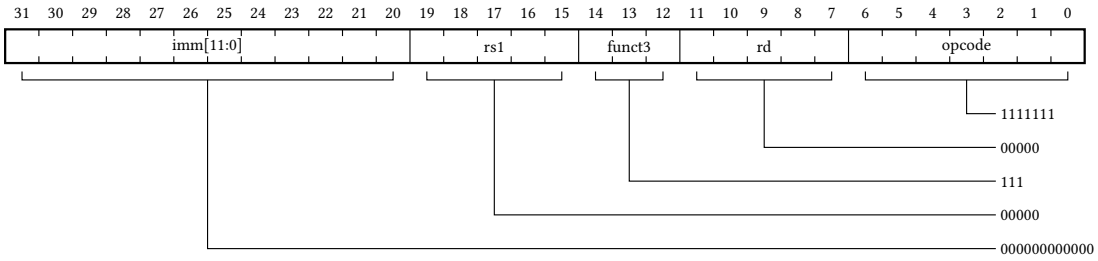


Figure 2: Mask for `ADDI`

While the mask marks the relevant bit positions, the match specifices the *exact pattern* these bits must form to represent a specific instruction. For the `ADDI` instruction, the `opcode` has the value `0010011` and the `funct3` has the value `000`. All other fields are treated as variable and set to zero in the match. The resulting match pattern `0b00000000000000000000000000010011`, shown in Figure 3, represents the hexadecimal value `0x13`.

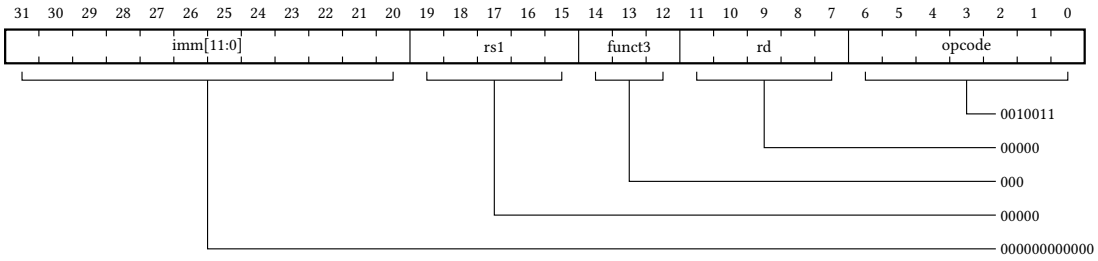


Figure 3: Match for `ADDI`

With this pair, any instruction word can be tested for being an `ADDI` instruction. The decoder applies the mask to the instruction using a bitwise AND operation, which zeros out all irrelevant bits. The resulting value is then compared to the match value. If they are identical, the instruction is identified as an `ADDI`. The operation can be expressed as follows in pseudo-code, as shown in Listing 1.

```
1 m = instruction & 0x707f          // apply mask
2 if m == 0x13                      // compare to match
3   // instruction detected as ADDI
4 else
5   // instruction is not ADDI
```

Listing 1: Example of mask and match for instruction decoding of `ADDI`

This example demonstrates how the structure of RISC-V instructions naturally supports a configuration-driven decoding process. By storing mask and match values in configu-

ration files, together with templates for reconstructing assembly, the decoder can remain generic and reusable while still supporting a wide variety of instructions.

2.2 TOML

TOML [7] (Tom's Obvious, Minimal Language) serves as a human-readable configuration file format that maps unambiguously to hash tables and data structures. The format was designed with the principle that configuration files should be easy for humans to read and write while remaining simple for machines to parse. TOML achieves this balance through a minimal syntax that supports essential data types including strings, integers, floating-point numbers, booleans, dates, arrays and nested tables.

Listing 2 demonstrates a typical TOML configuration structure, showcasing how different data types and organizational patterns can be expressed clearly and concisely.

TOML

```toml
title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
enabled = true
ports = [ 8000, 8001, 8002 ]
data = [ ["delta", "phi"], [3.14] ]
temp_targets = { cpu = 79.5, case = 72.0 }
```

Listing 2: Example TOML configuration

Within the context of this instruction decoder project, TOML serves as the primary configuration mechanism for defining instruction sets, encoding mappings and decoder parameters. This approach provides users with the flexibility to customize and extend the decoder's functionality without requiring modifications to the underlying source code.

2.3 Rust Programming Language

Rust [8] is a modern systems programming language that combines low-level control with strong guarantees of memory and thread safety. Its central innovation is the ownership model, which enforces correct use of references at compile time and prevents errors such as null pointer dereferences or data races. This approach avoids the need for a garbage collector and results in predictable performance characteristics, which is a decisive factor in systems-level development [9].

The language further distinguishes itself through abstractions that do not incur additional runtime cost. Constructs such as iterators or pattern matching can be expressed in concise, high-level code while still compiling to efficient machine instructions. This combination of safety, expressiveness and performance is particularly well suited for tasks such as instruction decoding, which require precise control over binary data while maintaining efficiency [8].

In this thesis, Rust serves as the foundation for the configuration-driven decoder. It ensures that the parsing of configuration files and the representation of decoded instructions remain consistent and reliable. At the same time, the compilation toolchain for WebAssembly allows the same implementation to be deployed unchanged in both command-line and browser environments, bridging systems programming with web-based interaction.

2.4 Web Assembly

WebAssembly [10, 11] (WASM) represents a significant advancement in web technology, providing a portable, low-level bytecode format designed for efficient execution across diverse computing environments. As an open standard, WebAssembly serves as a compilation target for multiple programming languages, including Rust, C and C++, enabling developers to achieve near-native performance in web applications.

The design of WebAssembly prioritizes security through sandboxed execution while maintaining high performance through efficient binary encoding and stack-based virtual machine architecture [12]. The format achieves hardware, language and platform independence, making it suitable for deployment across various operating systems and processor architectures. WebAssembly modules execute within a memory-safe environment that prevents unauthorized access to system resources while providing predictable performance characteristics.

For this thesis, compiling the Rust-based instruction decoder to WebAssembly enables seamless integration with web-based interfaces while maintaining the full functionality and performance of the native implementation. The WebAssembly compilation target facilitates the development of interactive educational tools and debugging interfaces that can run directly in web browsers.

2.5 Web Technologies

The web interface for the instruction decoder is built using modern web technologies, including HTML, CSS and JavaScript (Svelte). It leverages WebAssembly to run the

Rust-based decoder directly in the browser [13], providing users with a responsive and interactive experience. The interface allows users to input binary instructions (in binary, decimal or hexadecimal format), view the decoded assembly output and explore the semantic descriptions generated by the decoder. This integration enhances usability by providing immediate feedback and context for each instruction, making it easier to understand complex assembly code.

3 Architecture

The following sections present the design of the instruction decoder in a structured way. We begin with the general approach, which introduces the guiding principle of separating data from logic and explains why this matters for extensibility and clarity. After that, the structure of the configuration files is described in detail, showing how formats, types, mappings and instruction groups together capture the entire knowledge about the instruction set. Once the fundamental groundwork has been laid, the decoding process itself is explained step by step, from receiving a binary instruction word to producing an assembly string. Finally, the design is evaluated in terms of its advantages and its current limitations.

3.1 General Approach

The design of the decoder is guided by the principle of separating data from logic. Instead of hard-coding knowledge about instructions into the program, all information about encodings, fields and assembly representations is placed in external configuration files. The program itself only provides a stable and generic mechanism for interpreting this information. This makes the design highly modular and mirrors the modular philosophy of RISC-V itself.

At the highest level, the decoding process can be seen as a pipeline with four fixed stages as shown in Figure 4. First, an instruction is compared against known patterns to determine which entry it belongs to. Second, the instruction word is cut into fields according to a predefined type. Third, these raw fields are transformed into meaningful values, such as sign-extended immediates or register names. Finally, the values are substituted into a textual template to produce an assembly instruction string. This pipeline never changes, meaning every instruction, whether it belongs to the base set or to an extension, goes through the same four steps.

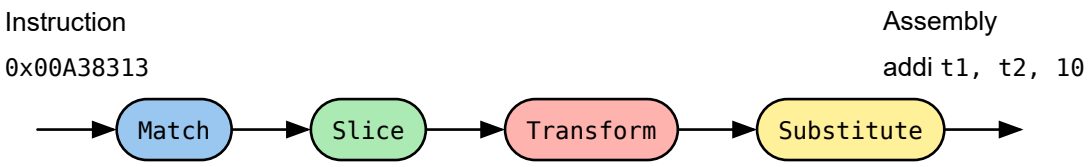


Figure 4: Decoding stages

The consequence of this approach is that the knowledge about instructions becomes transparent. Anyone can open the configuration files and immediately see how an instruction is defined, which fields it uses and how it is displayed. The decoding logic itself does

not need to change when new instructions are added. Instead, new entries are simply written into the configuration and the existing pipeline applies to them automatically. This keeps the core implementation compact and stable while allowing the instruction set to evolve freely.

Another important aspect is extensibility. RISC-V is structured around a small base set and multiple optional extensions. A data-driven design reflects this modularity naturally. Each extension can be described in its own configuration file and the decoder only needs to load the relevant files to support it. This means the tool can grow along with the instruction set, without ever requiring a redesign of its core.

Transparency, extensibility and stability are the three central goals of the general approach. By delegating all instruction-specific knowledge to data files and restricting the program to a uniform decoding pipeline, the design achieves these goals in a straightforward way.

3.2 Configuration File Structure

The external TOML files form the structural basis of the decoder. Each file corresponds to a particular instruction set slice and contains all information required for recognition and representation. To keep the discussion concrete, this section illustrates the concepts using the `RV32I` configuration file as an example.

At the top of every file, two fundamental entries, the name of the set and the instruction width in bits, appear. These values provide context for the decoder. The set name identifies the scope of the definitions, while the width ensures that only instructions of the correct size are matched. For `RV32I`, the file begins with the simple declaration of `set = "RV32I"` and `width = 32`.

The next major component is the section named `formats`. It introduces the different output formats used throughout the file and defines the building blocks required to describe instruction fields. Each building block is called a part. A part specifies a name, the number of bits it covers and a type that determines how these bits should be interpreted. Optionally, a format hint can be added to indicate whether the value should be displayed in decimal, hexadecimal, or another base. In practice, most parts refer to registers or immediates. An excerpt from the `RV32I` file is shown in Listing 3, where integer registers, immediate values and their hexadecimal variant are introduced as reusable parts.

```toml
[formats]
names = ["format_1-0","format_2-0","format_3-0"]
parts = [
  ["rd_Register_int",  5, "Register_int"],
  ["rs1_Register_int", 5, "Register_int"],
  ["imm",  32, "VInt"],
  ["himm", 32, "VInt", "hex"],
]
```

Listing 3: Excerpt from the `RV32I formats` section.

While parts describe possible building blocks, types define how an instruction word is actually cut into them. Each type lists the fields that make up an instruction together with the bit ranges they occupy using top for the most significant bit and bot for the least significant bit of the field. The order of these entries follows the natural orientation from the most significant to the least significant bit. In simple cases, a field corresponds to a contiguous slice, such as the five bits of a destination register. More complex layouts, such as branch immediates, are reconstructed from multiple non-contiguous slices. Figure 5 and Listing 4 illustrate this with an excerpt where the immediate value is spread across several positions but still treated as a single logical field.

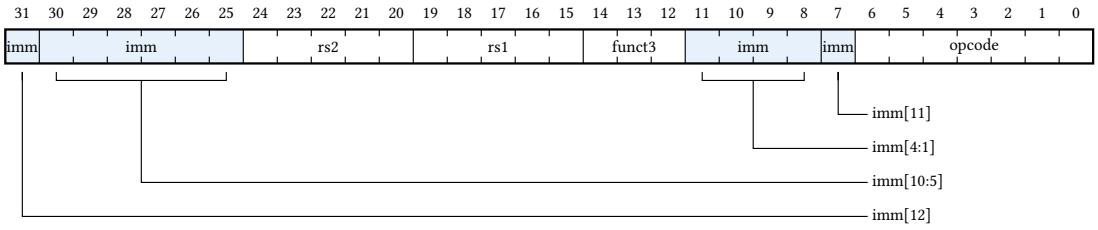


Figure 5: Branch-type layout

```toml
[[types.type_branch]]
name = "imm" top = 12 bot = 12
[[types.type_branch]]
name = "imm" top = 10 bot = 5
[[types.type_branch]]
name = "rs1_Register_int" top = 19 bot = 15
[[types.type_branch]]
name = "rs2_Register_int" top = 24 bot = 20
[[types.type_branch]]
name = "imm" top = 4 bot = 1
[[types.type_branch]]
name = "imm" top = 11 bot = 11
```

Listing 4: Branch-type definition in `RV32I`

The third structural element is the `mappings` section. Mappings associate raw numbers with human-readable names. They are especially important for registers, where numeric indices would otherwise dominate the output. In `RV32I`, integer registers are mapped to their symbolic names, beginning with zero mapped to `zero`, one to `ra` and so forth. Other mappings exist for floating point registers and for the qualifiers used in fence instructions. Listing 5 shows part of this definition, which enables the decoder to output familiar names instead of numbers.

```toml
[mappings]
names = ["Register_int", "Register_float", "Mapping_fence"]

Register_int = ["zero","ra","sp","gp","tp","t0","t1","t2", ...]

Mapping_fence = ["unknown","w","r","rw","o","ow","or","orw", ...]
```

Listing 5: Register and fence mappings in RV32I

Finally, the top-level format definitions connect all of these pieces. For every name listed earlier in `formats.names`, there is a corresponding section that ties together a type, a set of instruction recognition rules and one or more output templates. Recognition is performed by a mask and match pair. If the instruction word, when masked, equals the given match value, the instruction is identified. The output is then constructed using a template that substitutes the mnemonic and the field values in the appropriate order. Listing 6 demonstrates this with a format that handles the `addi` and `lw` instructions. Both rely on the same type, but their textual representation differs slightly, with load instructions shown in the familiar base plus offset form.

```toml
[format_2-0]
type = "type_i"

[format_2-0.instructions.addi]
mask = 0x0000707f
match = 0x00000013

[format_2-0.instructions.lw]
mask = 0x0000707f
match = 0x00002003

[format_2-0.repr]
default = "$name$ %rd_Register_int%, %rs1_Register_int%, %imm%"
lw      = "$name$ %rd_Register_int%, %imm%(%rs1_Register_int%)"
```

Listing 6: Format definition for `RV32I`

In the representation templates, placeholders enclosed in `%` symbols, such as `%rd_Register_int%`, act as variables that are dynamically replaced during decoding. Each placeholder directly corresponds to a field name defined in the parts section of the configuration file. These parts specify how many bits are extracted and how they should be interpreted. The third parameter of a part links it to a mapping, which translates raw numeric values into human-readable names.

For example, the following configuration excerpt in Listing 7 shows how the field `rd_Register_int` is connected to the `Register_int` mapping.

```toml
[formats]
parts = [
    ["rd_Register_int", 5, "Register_int"],
]

[mappings]
names = ["Register_int"]

Register_int = [
    "zero", "ra", "sp", "gp", "tp", "t0", "t1", "t2",
    "s0", "s1", "a0", "a1", "a2", "a3", "a4", "a5",
    ...
]
```

Listing 7: Format definition for `RV32I`

When decoding an instruction, the type definition first identifies which bits belong to the destination register field. If, for example, the bits in positions 11-7 are `00110`, this binary value corresponds to the decimal index 6. Because the part `rd_Register_int` is linked to the `Register_int` mapping, the decoder interprets this numeric index by looking it up in the mapping table, where index 6 is associated with the symbolic register name `t1`. As a result, the placeholder `%rd_Register_int%` is ultimately replaced with `t1` when the template is expanded. The special placeholder `$name$` works in the same way but is resolved to the mnemonic of the recognized instruction, such as `addi` or `lw`.

Taken together, these elements form a coherent hierarchy. The header specifies the context, formats introduce reusable parts, types assemble these parts into full instruction layouts, mappings provide symbolic names and the final format definitions link everything into concrete instructions with recognisable output. The `RV32I` file illustrates this logic, but the structure is generic and applies to all instruction set extensions in exactly the same way.

3.3 Decoding Process

The actual decoding of an instruction is performed in a strictly uniform way. No matter whether the input is a 16-bit compressed word or a 32-bit standard instruction, the decoder applies the same pipeline. The four stages were already introduced in the General Approach. Figure 6 recalls them here and prepares the ground for a more detailed discussion.

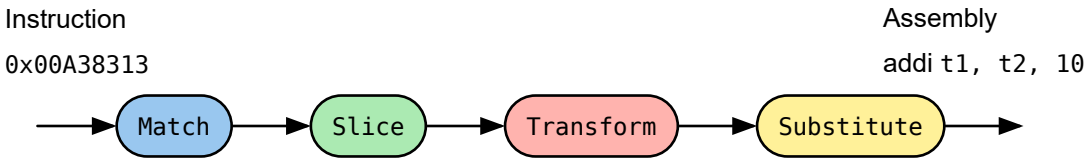


Figure 6: Decoding stages

The pipeline begins with the matching step in Listing 8. Only those instruction sets whose header declares the correct width are considered. Each instruction entry then applies a simple rule: the bitwise conjunction of the word and a mask must equal the given match value. When this condition holds, the instruction has been recognized. This test is the entire decision logic, without any further nested conditions or look-ups in the Rust code.

```rust
fn matches(mask: u128, match: u128, word: u128) -> bool {
    (word & mask) == match
}
```

Listing 8: Instruction recognition

Once recognition succeeds, the selected format points to a type that describes how the instruction word must be cut into slices. Each slice extracts a range of bits and assigns them to a named field. If several slices share the same name, they are joined into a single logical value. This mechanism allows scattered immediate fields, such as those in branch or jump instructions, to be reassembled correctly. The type can also request that the most significant bit be extended before joining, which ensures that sign information propagates properly. After slicing, the decoder has a collection of raw fields, but they are not yet ready for display.

Transformation interprets these fields. Numeric parts are cast to their natural size, mapping-backed parts such as registers are replaced by symbolic names and wide “virtual integers” are prepared either as signed or unsigned values. The choice of signedness is not embedded in the type but comes from the matching instruction entry, which carries a flag. This separation keeps types reusable and makes instruction-level variations explicit.

Finally, substitution produces the assembly line, shown in Listing 9. Templates are plain strings that contain `$name$` for the mnemonic and `%field%` markers for operands. Placeholders are validated when the configuration file is loaded, which means runtime expansion is only a matter of replacement.

```Rust
fn expand(fmt: &str, name: &str, fields: &std::collections::HashMap<String, String>) -> String {
    let mut out = fmt.replace("$name$", name);
    while let Some(b) = out.find('%') {
        let e = b + 1 + out[b + 1..].find('%').unwrap();
        let key = &out[b + 1..e];
        let val = fields.get(key).map(String::as_str).unwrap_or("");
        out = out.replace(&out[b..=e], val);
    }
    out
}
```

Listing 9: Substitution step

Two implementation details are worth to mention. First, the decoder collects all matches and returns the last one. This deliberate policy allows configuration authors to define aliases or let a more specific pattern override a general one without complicating the code. Second, almost all potential errors are caught at load time. Missing keys, type mismatches, unknown mappings, or template placeholders that do not correspond to any slice are reported before decoding begins. This ensures that the runtime path remains compact and fast while error messages stay precise and actionable.

To make the interaction between configuration and code clearer, Figure 7 shows the four stages of the decoding process side by side. The left column lists the information defined in the TOML files, while the right column illustrates how the decoder applies it at runtime. This perspective highlights the fundamental design choice, that the code performs only generic operations and all instruction-specific knowledge resides in the configuration files.

The decoder may find several matches but always returns the last one and the signedness of virtual integers is not determined by the type definition but by the instruction entry that matched.

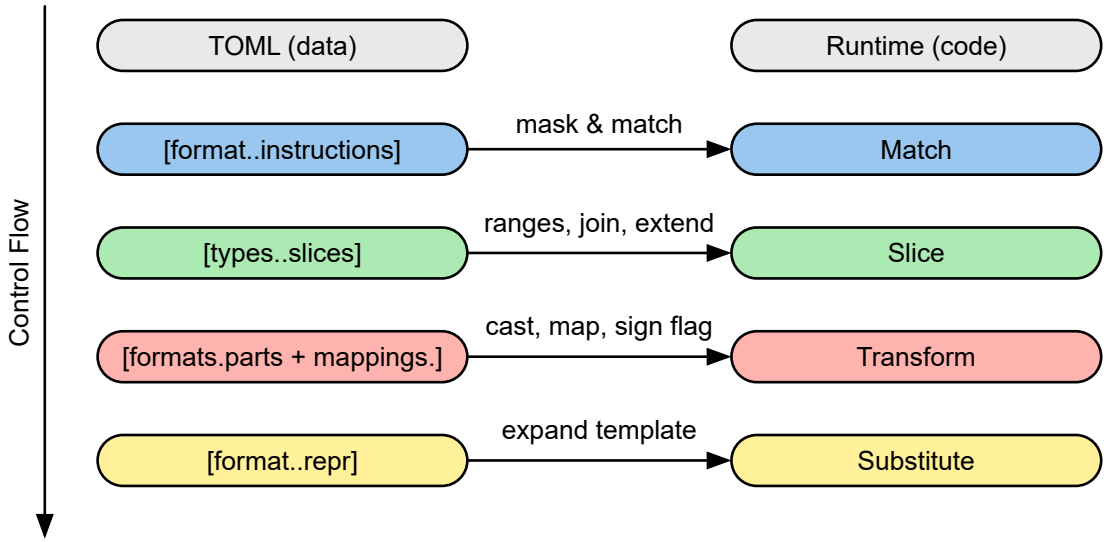


Figure 7: Relationship between configuration data and runtime steps in the decoding process

To close the description, it is useful to see a worked example. Consider the 32-bit word `0x00A38313`. The matching stage finds that the mask and match values under the entry `addi` succeed. The referenced type cuts the instruction into three fields, which are destination register, source register and immediate. Transformation replaces the destination `x1` with `t1`, keeps the source `x0` as `t2` and interprets the immediate as a signed decimal. Substitution applies the template `$name$ %rd_Register_int%, %rs1_Register_int%, %imm%`. The result is the final assembly string `addi t1, t2, 10`.

The same four steps apply to every instruction, regardless of its width or category. The strength of the design is that the Rust implementation never hard-codes this knowledge. It only enforces the sequence of match, slice, transform and substitute. All instruction semantics remain external in the TOML data, which makes the decoder small, auditable and naturally extensible.

3.4 Evaluation

The design described in the previous sections has demonstrated that a decoder for a complex instruction set can be realized with a remarkably compact core, if all instruction knowledge is externalized. The fundamental principle, namely the separation of logic and data, has proven effective in practice. The decoder itself only enforces a uniform sequence of operations, while the details of every instruction are defined in TOML configuration files. This architecture provides three immediate advantages: it is extensible, it is transparent and it reduces the likelihood of errors.

Extensibility becomes clear when considering how new instructions are introduced. In conventional decoders, adding even a single operation requires changes in the program

code, recompilation and a careful review of side effects. Here the situation is entirely different. The core implementation does not need to be touched. A configuration file can be adapted, extended, or replaced and the new instruction becomes available immediately. This approach mirrors the modular structure of RISC-V itself, where the base set can be complemented by optional extensions. Each extension can be placed in its own configuration file and loaded only when relevant. In this way the decoder scales naturally with the growth of the instruction set.

Transparency is another important outcome. All information about encodings, formats and mappings is written in plain text. It can be inspected, reviewed and validated independently of the Rust code. Anyone familiar with the instruction set can understand how a specific operation is defined without reading a single line of the implementation. This makes the system not only easier to verify but also more open to reuse. The same configuration files can be used by external tools, teaching platforms, or visualization environments. A student, for example, could explore instructions interactively in a browser-based tool that simply loads the TOML definitions without re-implementing the decoding logic.

The uniform pipeline of four stages also contributes to reliability. Since every instruction passes through the same sequence of match, slice, transform and substitute, there is no need for large tables or nested conditions in the code. The risk of hidden inconsistencies is reduced because the program applies the same generic operations to all inputs. Configuration files are validated when the decoder is constructed. Missing keys, type mismatches, unresolved mappings, invalid slice ranges and template placeholders that do not correspond to available fields are reported immediately. If any of these checks fail, the decoder is not created. As a result, the runtime decoding path is compact and predictable. At runtime only two outcomes remain. Either the word is successfully decoded into an assembly string or, if no rule matches, the system returns a controlled “`Unknown Instruction`” message.

The evaluation must also consider the limitations of the current version. The most evident restriction is that decoding produces only a single assembly string. The output does not provide a structured list of operands, nor does it attach any semantic information. For tools that wish to analyze operands directly, this is a significant drawback because they must parse the assembly string again. From an educational perspective, the lack of descriptive metadata is equally problematic. The decoder does not explain the effect of an instruction or place it into a broader category. Beginners who encounter the output

receive only a textual representation and must still consult an external reference or documentation to understand the meaning.

These limitations are not flaws in the strict sense but rather boundaries of the chosen design. They define the point at which the current approach reaches its natural end. On one hand, the evaluation confirms that a minimal core can successfully decode a complete base set such as `RV32I`. On the other hand, it makes clear that the approach alone cannot provide structured data, semantic enrichment, or contextual analysis. For future work, the challenge will be to preserve the compactness and transparency of the current design while extending it in these directions.

4 Instruction Explanations and Implementation

This chapter describes the practical realization of the work. It is structured along four components that build on each other. First, the existing configuration files were adapted and enriched with descriptions, so that instructions carry not only their assembly string, but also explanatory context. Second, the decoder program in Rust was extended to make use of this additional information and to return structured results that combine assembly and description. Third, the implementation was compiled to WebAssembly and connected to JavaScript through a minimal binding layer, which enables the decoder to be reused in a browser environment. Finally, a browser-based interface was developed that presents the output interactively, allowing users to enter instruction words and immediately receive both the assembly form and the semantic explanation. Together, these steps form a complete path from raw configuration data to a usable web application.

4.1 Configuration Files Adaptation

The first step in the implementation was the adaptation of the configuration files. The original decoder already relied on TOML files to describe instruction formats, bit fields and mappings. These files contained everything necessary to transform a binary word into an assembly representation, but they did not provide any human-readable explanation of what the decoded instruction actually meant. In practice, the decoder could only produce strings like `addi t1, t2, 10`. While formally correct, such output forces users to consult external references to understand the semantics of the operation. To address this limitation, the configuration was extended with descriptive information for each instruction.

The descriptions were sourced from the *riscv-unified-db*[3] repository, an officially maintained project of *RISC-V International* that provides a machine-readable database of the specification. Its YAML files contain instruction definitions in a structured format together with explanatory text, written in AsciiDoc. The task was to combine this semantic information with the TOML structures already used by the decoder.

A custom Python script performed the enrichment. It parsed all YAML and TOML files, matched instruction names between them and added a new key called `description` to each TOML entry where a counterpart with documentation was found. The added text was first converted from AsciiDoc into HTML, so that it could later be rendered directly in a browser. Instructions without a description in the upstream repository were left unchanged, making the enrichment process strictly additive.

[3] https://github.com/riscv-software-src/riscv-unified-db

A concrete example in Listing 10 shows the outcome. In the original configuration, an instruction entry only specified `mask` and `match`. After enrichment, it also contains a `description` field with an HTML fragment.

Before enrichment

```toml
[format_2-0.instructions.addi]
mask = 0x707f
match = 0x13
```

After enrichment

```toml
[format_2-0.instructions.addi]
mask = 0x707f
match = 0x13
description = """
<p>Adds an immediate value to xs1, and stores the result in xd.</p>
"""
```

Listing 10: Instruction entry before and after enrichment

To avoid inconsistencies, the script enriches only when names match exactly between TOML and YAML. Any instruction missing in either source is left untouched, ensuring that the resulting configuration remains valid. At the end of this step, the TOML files combined both the structural information required for decoding and, when available, a concise human-readable explanation of the instruction semantics. This prepared the ground for the later modifications of the Rust decoder, which could then return the enriched data as part of its output.

4.2 Instruction Decoder Extension

The enriched configuration files prepared the foundation for the second part of the implementation, which is extending the instruction decoder itself. The baseline version of the program already provided a robust pipeline that could take a binary word, match it against instruction formats, extract slices and substitute them into assembly templates. However, its output was limited to a single string representing the instruction in assembly syntax. To support richer use cases, the decoder was modified to return a structured result that combines the assembly representation with an optional description taken from the TOML files.

The extension was intentionally kept small so that it would integrate smoothly with the existing design. Instead of replacing the output type everywhere, the new functionality was added through an additional data structure. This ensured backward compatibility.

Older TOML files without descriptions continued to work, while enriched ones could expose additional information. From the perspective of the decoding pipeline, none of the established stages changed. Matching, slicing and transformation proceed exactly as before. The description field only becomes relevant in the final stage, where the decoded information is assembled into the result.

At the centre of this modification is the new struct `InstructionInfo`, which holds the output of the decoder in two fields, namely the formatted assembly string and the optional description as demonstrated in Listing 11.

```rust
#[derive(Clone, Debug)]
pub struct InstructionInfo {
    pub assembly: String,
    pub description: Option<String>,
}
```

Listing 11: Structured return type of the instruction decoder

The choice of `Option<String>` for the description ensures that the field is strictly optional. When no description is present in the TOML entry, the decoder simply returns `None`. This avoids breaking existing files and allows gradual adoption of the extended schema. The assembly string remains mandatory and continues to be provided in all cases.

The next change was to read the new field from the TOML file, as illustrated in Listing 12. The constructor of the `Instruction` struct was extended by one short block that checks for the presence of a `description` key and, if found, retrieves its string value.

```rust
let description = if table.contains_key("description") {
    handle_err_get(
        table,
        error_stack,
        "description",
        table_prefix,
        Value::String("".to_string()),
    )
    .as_str()
    .map(|s| s.to_string())
} else {
    None
};
```

Listing 12: Optional description field from TOML

Finally, the decode functions were updated to construct the new return type. Whenever an instruction matches, the formatted assembly string is computed as before and the

description field is attached to the result if present. The following excerpt from the method `decode_all` illustrates the integration in Listing 13.

```rust
finds.push(InstructionInfo {
    assembly,
    description: inst.description.clone(),
});
```

Listing 13: Returning assembly and description

The pipeline itself remains unchanged. It still passes through the stages of matching, slicing, transformation and substitution. The only modification is that the final output is now a richer struct rather than a single string. Figure 8 illustrates this integration. The four familiar stages of the decoder are preserved, but the final arrow leads not only to the assembly representation but also to the optional description.

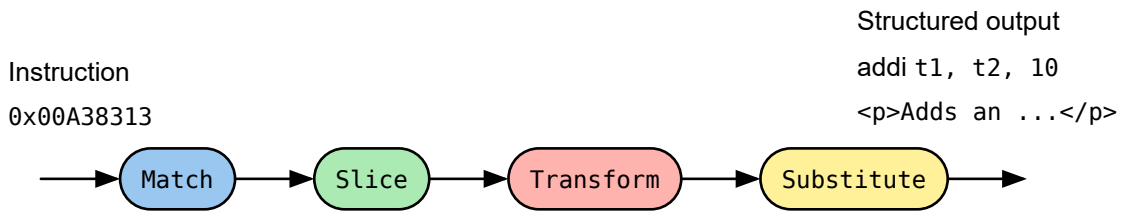


Figure 8: Decoder pipeline extended structured output including description

From the user's point of view, the change is seamless. A call to the decoder now produces an `InstructionInfo` object, which always provides the formatted assembly string and may also provide a description. For example, decoding the binary word `0x00A38313` yields the assembly `addi t1, t2, 10` together with an HTML fragment that explains the semantics of the `addi` instruction. If the TOML entry contains no description, the result still contains the correct assembly but leaves the description field empty.

The benefit of this design lies in its simplicity. The core pipeline of the decoder does not need to know about HTML, AsciiDoc, or any other representation of explanatory text. It merely passes through a string that may or may not be present. This approach keeps the code base small and predictable, avoids introducing runtime dependencies and ensures compatibility with both old and new configuration files. At the same time, it enables downstream applications, such as the web interface introduced later, to display not just raw assembly but also a concise explanation of the instruction's semantics.

4.3 WebAssembly Bridge

The next step was to make the decoder available in a browser environment. For this purpose, the Rust implementation was compiled to WebAssembly and connected to JavaScript via the `wasm-bindgen` crate. This bridge is intentionally minimalistic. It forwards configuration and instruction words from JavaScript into Rust and it returns structured results back. All decoding logic remains in Rust, so the WebAssembly layer introduces no new semantics.

At the heart of the bridge is a wrapper around the existing decoder. The struct is annotated with `#[wasm_bindgen]`, which exposes it as a usable class to JavaScript, is presented in Listing 14.

```rust
#[wasm_bindgen]
pub struct InstructionDecoder {
    decoder: Decoder,
}
```

Listing 14: WebAssembly Wrapper for Instruction Decoder

The wrapper accepts a list of TOML configuration strings when constructed and afterwards these are parsed into Rust data structures. Once initialized, the decoder can be reused for multiple queries without reloading the configuration.

Decoding itself is made available through simple methods. One of them accepts a string that may be written in decimal, hexadecimal, or binary notation. The bridge only converts types and propagates errors. Listing 15 displays the usage of method `decode_from_string` as explained before.

```rust
#[wasm_bindgen]
impl InstructionDecoder {
    #[wasm_bindgen]
    pub fn decode_from_string(
        &self,
        s: &str,
        width: usize
    ) -> Result<InstructionInfo, JsValue> {
        self.decoder
            .decode_from_string(s, width)
            .map(InstructionInfo::from)
            .map_err(|e| JsValue::from_str(&e))
    }
}
```

Listing 15: Exported decode method for string input

The return type is also exported with `#[wasm_bindgen]`, as shown in Listing 16. It mirrors the internal `InstructionInfo` struct but provides explicit getters so that JavaScript code can access the fields as object properties.

```rust
#[wasm_bindgen]
#[derive(Clone)]
pub struct InstructionInfo {
    assembly: String,
    description: Option<String>,
}

#[wasm_bindgen]
impl InstructionInfo {
    #[wasm_bindgen(getter)]
    pub fn assembly(&self) -> String {
        self.assembly.clone()
    }

    #[wasm_bindgen(getter)]
    pub fn description(&self) -> Option<String> {
        self.description.clone()
    }
}
```

Listing 16: Return type accessible as a JavaScript object

4.4 Frontend

The frontend forms the visible part of the project. It demonstrates how enriched instruction data and the WebAssembly bridge can be turned into a usable tool for interactive decoding. Unlike command-line utilities, which primarily address developers or automated pipelines, the frontend provides an accessible interface that runs entirely in the browser. It connects three components introduced earlier. The enriched TOML files as configuration, the Rust decoder compiled to WebAssembly and a lightweight single–page application built with modern web tooling. Together these elements allow users to input instructions and receive immediate explanations without installing additional software.

4.4.1 Integration and Flow

The web application was implemented with Svelte and bundled using Vite. These technologies were chosen for their performance and simplicity. Vite provides fast incremental builds during development, while in production it emits a compact static bundle containing compiled Svelte components, the JavaScript glue produced by `wasm-bindgen`, the `.wasm` binary and the enriched TOML files. This ensures that the tool can be deployed as static assets without server-side processing. Svelte supplies the reactivity model that underpins

user interaction, while TailwindCSS offers consistent design aligned with the institute's visual style.

A central aspect of the build pipeline is the seamless integration of WebAssembly. Vite's plugin system allows `.wasm` modules to be imported like any other asset, while top-level `await` ensures that WebAssembly initialisation happens naturally at start-up. A minimal configuration suffices to combine these elements, shown in Figure 9.

```ts
// vite.config.ts
import { defineConfig } from "vite";
import { svelte } from "@sveltejs/vite-plugin-svelte";
import wasm from "vite-plugin-wasm";
import topLevelAwait from "vite-plugin-top-level-await";

export default defineConfig({
  plugins: [svelte(), topLevelAwait(), wasm()],
});
```

Figure 9: Minimal Vite configuration

At runtime the frontend does not attempt to parse TOML itself. Instead, the enriched TOML strings are imported as static assets and passed directly into the WebAssembly bridge, which forwards them to the Rust decoder. This avoids duplicating parsing logic across environments and guarantees that both the web interface and the command-line tool rely on identical code paths. Once initialised, decoder objects for `RV32` and `RV64` are kept alive in memory, ensuring that subsequent queries can be answered without repeated parsing or validation.

4.4.2 Logic and interaction

The interaction flow is linear and predictable. A user provides an instruction in the input form, the frontend validates and normalises the string and the corresponding decoder instance is called through the WebAssembly bridge. The decoder returns a structured object that contains the formatted assembly string and, if available, the enriched HTML description. This output is then rendered in the interface.

The input logic supports binary, hexadecimal and decimal numbers, with or without prefixes. Automatic detection reduces friction, since users do not need to declare the format explicitly. Bit width is inferred from context, for example by checking the length of a hexadecimal or binary string. Invalid inputs are caught early and reported with specific error messages. This behaviour keeps the interface robust and responsive, while still delegating all domain-specific decoding to the Rust module.

A shortened code excerpt in Figure 10 illustrates the structure of the submit handler. It shows how input is validated and normalised before being passed to the decoder, but omits secondary details such as error messages for brevity.

```ts
const handleSubmit = (input: string) => {
  let value: number, width: 16 | 32;

  if (/^[01]+$/.test(input)) {
    width = input.length === 16 ? 16 : 32;
    value = parseInt(input, 2);
  } else if (input.startsWith("0x")) {
    const h = input.slice(2);
    width = h.length === 4 ? 16 : 32;
    value = parseInt(h, 16);
  } else {
    value = parseInt(input, 10);
    width = value <= 0xffff ? 16 : 32;
  }

  return decoder.decode(value, width);
};
```

Figure 10: Simplified input normalisation before calling the decoder

The logic emphasises separation of concerns. Validation and normalisation remain in the frontend, while the interpretation of instruction semantics is fully handled by the WebAssembly decoder. This architecture reduces redundancy and makes the frontend a thin, reliable layer that translates user actions into decoder queries.

4.4.3 User interface

The user interface was designed to be compact, clear and visually consistent. Blue accents highlight interactive elements, while neutral tones provide structure for labels and borders. The ICS logo in the header ties the tool to its academic context, underlining its role as a teaching and research aid.

The landing view is shown in Figure 11. It presents a header with title and logo, an input form at the centre and whitespace that keeps focus on interaction. Users can begin entering instructions immediately.

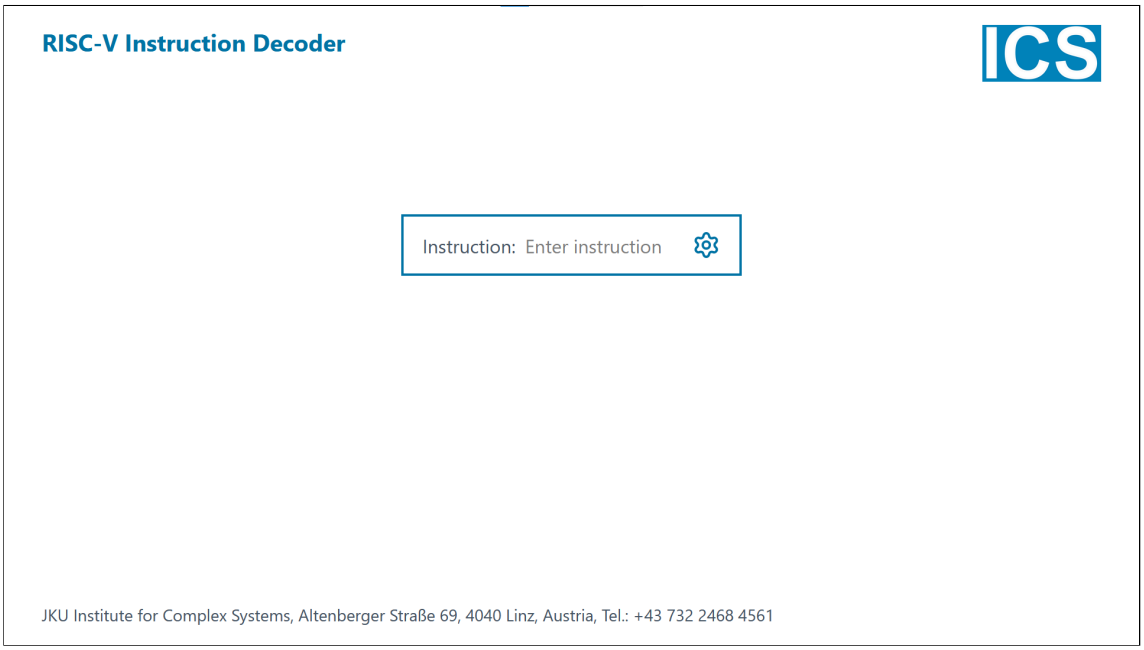


Figure 11: Landing view

After decoding, results appear in a card view shown in Figure 12. The card contains the assembly in the first row and a short teaser of the description in the second. A small icon allows expansion into a modal window. This design balances brevity with depth, offering immediate feedback while still enabling more detailed study.

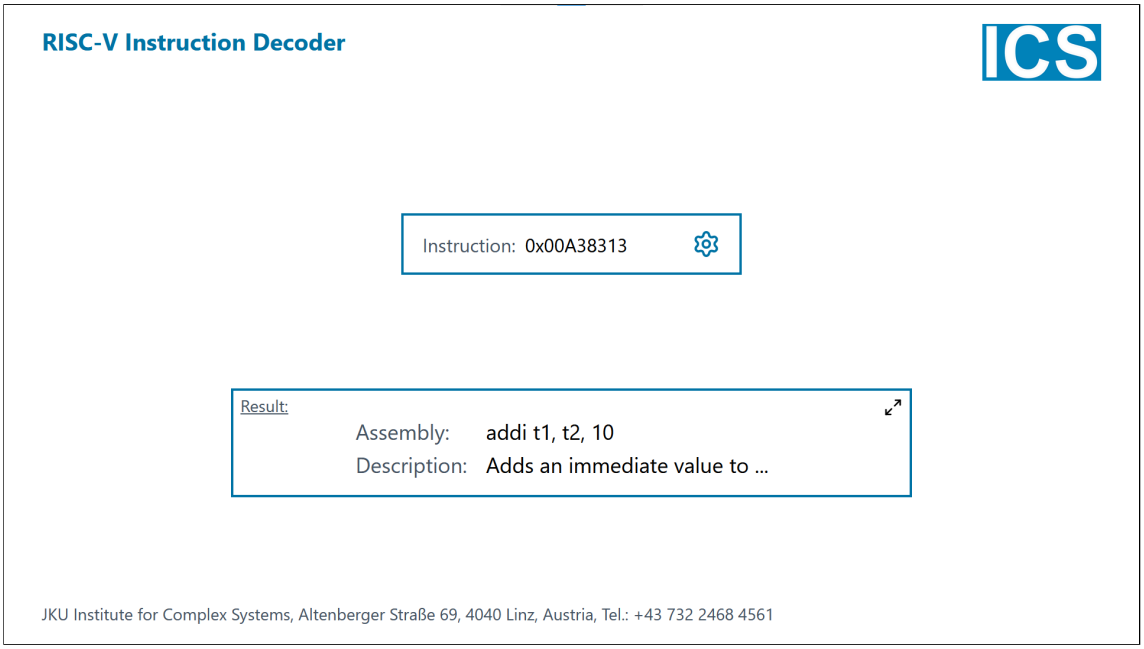


Figure 12: Result card

Longer descriptions are shown in the modal view in Figure 13. Here the full HTML fragment is displayed with consistent typography. Tailwind's typography plugin ensures

that headings, paragraphs and code snippets remain correctly formatted based on their HTML tags.

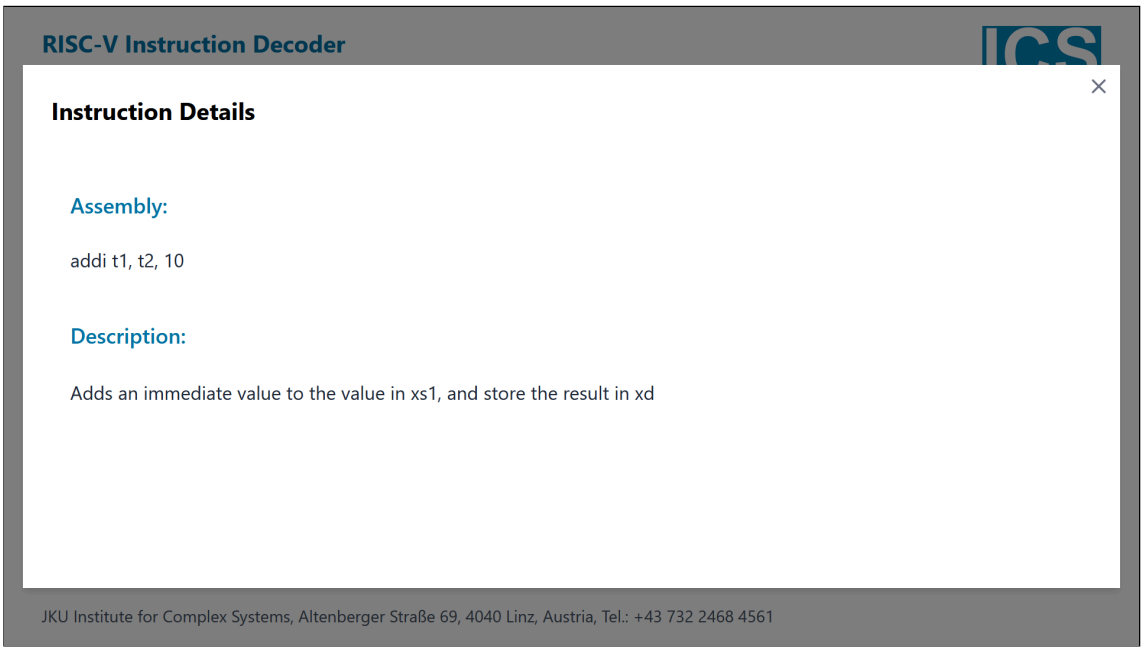


Figure 13: Detailed instruction view

The interface also includes a settings dialog for selecting between `RV32` and `RV64` and for reviewing supported input formats. This reduces reliance on external documentation and ensures that the user understands how instructions should be entered. Figure 14 shows the dialog.

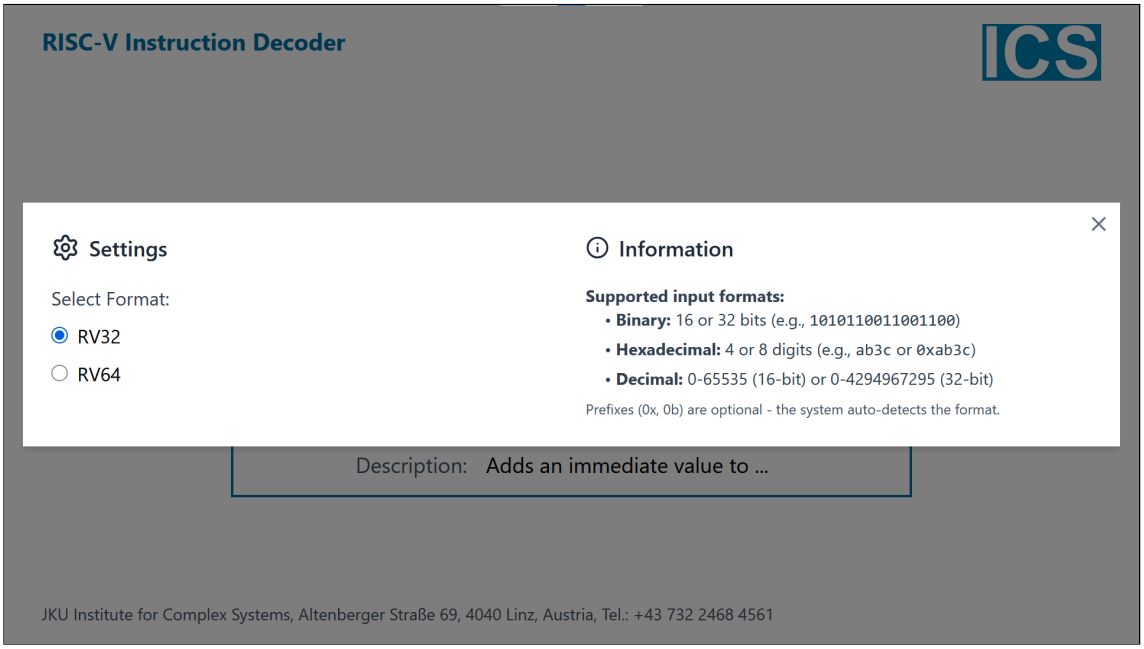


Figure 14: Settings dialog with format selection and input hints

The interface is responsive across devices. On smaller screens, elements stack vertically and remain fully functional. On larger displays, whitespace preserves focus. Accessibility was considered through sufficient text contrast, labelled input fields and dismissible modals. A footer with institutional contact information completes the presentation.

5 Conclusion

This thesis extended an existing instruction decoder with the capability to provide semantic explanations alongside assembly output and demonstrated the result as an interactive web-based tool. The work began by enriching TOML configuration files with explanatory content derived from external RISC-V repositories. The Rust decoder was then adapted to produce structured results containing both symbolic assembly code and optional descriptions. Through compilation to WebAssembly, the extended decoder was integrated into a lightweight Svelte/Vite frontend that allows users to interactively decode instructions in the browser.

The contributions of this thesis are threefold. First, the enrichment of configuration files unified structural and semantic knowledge in a single source of truth. Second, the decoder was extended with minimal but effective modifications that preserved compatibility with existing configurations. Third, the development of a responsive frontend illustrated how the approach can be deployed as an educational and analysis tool accessible without specialized software. Together, these contributions show that a data-driven decoder can evolve into a platform that not only translates binary words into assembly but also conveys their meaning in context.

Several limitations remain. The current implementation supports only a subset of RISC-V and provides explanations in static form. More advanced features such as operand-level metadata, visualization of instruction execution, or support for compressed instructions were beyond the scope of this thesis. Future work may address these directions, broaden ISA support and explore integration with debugging or teaching environments.

Overall, the project demonstrates how combining structural fidelity with semantic enrichment can improve the accessibility of instruction-level analysis. The resulting system is lightweight, extensible and practical, highlighting how instruction decoders can evolve into tools that serve both experts and learners.

Bibliography

[1] N. Jay and B. P. Miller, “Structured random differential testing of instruction decoders,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 84–94.

[2] F. Skarman, L. Klemmer, D. Große, O. Gustafsson, and K. Laeufer, “Surfer – An Extensible Waveform Viewer,” in *International Conference on Computer Aided Verification (CAV)*, 2025. [Online]. Available: https://ics.jku.at/files/2025CAV_Surfer.pdf

[3] A. Waterman *et al.*, “The RISC-V instruction set manual,” *Volume I: User-Level ISA', version*, vol. 2, pp. 1–79, 2014.

[4] H. of RISC-V, Accessed: Aug. 20, 2025. [Online]. Available: https://riscv.org/about/#history

[5] D. Patterson and A. Waterman, *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[6] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, “The risc-v instruction set manual volume 2: Privileged architecture version 1.7,” 2015.

[7] T. Preston-Werner, Accessed: Jul. 28, 2025. [Online]. Available: https://toml.io/en/

[8] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[9] R. Jung, “Understanding and evolving the Rust programming language,” 2020.

[10] A. Rossberg, “Webassembly specification,” *WebAssembly Community Group*, vol. 2, 2021.

[11] B. Sletten, *WebAssembly: The Definitive Guide*. " O'Reilly Media, Inc.", 2021.

[12] World Wide Web Consortium (W3C), “WebAssembly Core Specification 1.0.” Accessed: Aug. 22, 2025. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[13] Mozilla Developer Network, “WebAssembly Concepts.” Accessed: Aug. 24, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly