Manual

Sail RISC-V Coding Guidance

29.05.2025
Jiongjia Lu
Chipsalliance T1 Team

Contents

1. Abstract

This project is a RISC-V ISA model implementation using Sail. We want the flexibility to opt-in a new instruction set, and gain maintainability from simple design, which is not available from sail-riscv and spike now.

This document aims to help developers understand the example Sail model implementation and act as a guidance for how to fix bugs, add new instruction, add new architecture, or create a new model based on this project. For developers who don't understand Sail and Sail toolchains, we also provide some additional document upon official manual.

Currently the example project support `rv64gc`.

2. Introduction

2.1. How to build this document

Typst 0.13.1 is required.

```
typst compile ./docs/dev.typ ./docs/rendered.pdf
```

2.2. Gloassary

Name	Notes
boat	*boat* is the emulator utilizing the user provided *sail_impl* RISC-V models to drive a RISC-V Core
Sail	The rems-project/sail project. This document will ofter use "Sail" to refer to the Sail Project, Sail Programming Language and Sail CLI tools
sail_impl	*sail_impl* reference to this project, which should provide a Sail implemented RISC-V execution model, from instruction description to register status.
sailcodegen	A Scala implemented Sail code generator

2.3. Designs

The This project project employs a distinct methodology compared to standard Sail projects. To enhance maintainability, the Instruction Set Architecture (ISA) model definition is segmented into multiple code snippets. A CLI tool for code generation is then utilized to assemble the complete Sail code from these snippets. This approach allows each instruction to be

defined in an individual file, thereby creating a singular, reviewable unit of code.

All architectural states are consolidated and managed within a separate, dedicated file. We have a guiding principle that each Sail model should exclusively define architectural states. Any interactions with hardware or the underlying system are exposed through an external C Application Programming Interface (API). For code readability, these hardware interactions, such as memory read and write operations, must be declared in a designated Sail file, accompanied by a corresponding C header file.

We advocate for minimizing the amount of C code. The C code should primarily function as an intermediary layer, connecting the Sail ISA model with the emulator. Consequently, developers should make every effort to avoid creating and linking additional C libraries.

The explicitly defined C API enables the use of nearly any programming language that supports a Foreign Function Interface (FFI) to implement hardware behaviors not defined in the ISA specification. For This project, the Rust programming language was selected to develop the emulator. This choice was driven by Rust's design simplicity and the readability of its code. Furthermore, the Rust community provides robust compiler toolchains, which helps to reduce the time spent on configuring and managing the build system.

3. Codegen CLI

4. Model

This section covers detail implementation and coding guidance for a Sail ISA model.

4.1. Instruction behavior

All ISA behavior describe file are placed under `model` directory.

Instruction behavior definition files are separated by instruction name, and grouped by their corresponding instruction sets. The instruction sets group rule follows riscv/riscv-opcodes. For example, load word instruction `lw` should be placed under `rv_i` directory, and load double word instruction

`ld` should be placed under `rv64_i` directory. Developers can check https://github.com/riscv/riscv-opcodes/tree/master/extensions for details.

Each file is treated as function body. The Sail code-gen CLI will traverse the `model/<extension>` directory and generate function signature for all the files. So developers can consider writing instruction file is in a function context, with arguments like global built-in value binding and some prelude function imported. The Sail code-gen CLI generate the function argument using encoding specified in riscv-opcodes.

For example, when developers try to add `addiw` instruction, their should check rv64_i encoding file for the instruction encoding. And as we can see, `addiw` is a instruction in rv64_i instruction sets, so developers should create a `addiw` file under `rv64_i` directory. Then as riscv-opcodes shown, `addiw` has three variable fields: `rd`, `rs1`, and `imm12`, these fields will be generated as Sail function arguments, and available in each file context. At runtime, they will be the bits vector value extracted from instruction decode result.

So developers can have following implementation for `addiw` behavior.

```
let result : XLENBITS = sign_extend(imm12) + read_GPR(rs1);
write_GPR(rd, sign_extend(result[63..0]));
tick_pc();
```

- `XLENBITS`: A prelude bits type with pre-defined length. In the context of rv64,

it is set as `bits(64)`.

- `sign_extend`: A prelude function that do a sign extend to the value.
- `imm12`: function arguments that will be generated at build time.
- `read_GPR`: A prelude function that read general purpose register file by id.
- `rs1`: function arguments that will be generated at build time.
- `write_GPR`: A prelude function that will write general purpose register file with given data.
- `tick_pc`: A prelude function that will increase the `PC`.

After code-gen phase, we will have following Sail function generated for executing `addiw` when decoding match:

```
union clause ast = ADDIW : (bits(5), bits(5), bits(12))
mapping clause encdec = ADDIW(rd, rs1, imm12) <-> imm12 @ rs1 @ 0b000 @
rd @ 0b0011011
function clause execute (ADDIW(rd, rs1, imm12)) = {
  let result : XLENBITS = sign_extend(imm12) + read_GPR(rs1);
  write_GPR(rd, sign_extend(result[63..0]));
```

```
  tick_pc();
}
```

4.2. FFI Interface

The model implementation is required to provide a C header file named "model_prelude.h". All external functions necessary for the model's operation should be defined within this file. This serves as a clear reference for emulator developers, informing them of the specific APIs that the Sail model relies upon and that, consequently, need to be implemented on the emulator side.

4.3. "Global" functions

Each instruction description file is a lonely individual file. Since these instruction description file will finally be assembled into one Sail file, there all have corresponding context.

This section instruct you all the possible context when defining a instruction.

- All the library functions: TODO: jump to library section
- Registers defined at riscv-opcodes TODO: explain possible register name trim

4.4. Provided library functions

4.4.1. Memory Read/Write API

Following are required memory operation that should be implemented at emulator side:

Name	Type	Description
`phy_read_byte`	`bits(64) -> bits(8)`	`[phy_read_byte address]` is the value of the byte at physical `[address]`
`phy_read_half_word`	`bits(64) -> bits(16)`	`[phy_read_half address]` is two bytes value starting at physical `[address]`
`phy_read_word`	`bits(64) -> bits(32)`	`[phy_read_word address]` is word length value starting at physical `[address]`
`phy_read_double_word`	`bits(64) -> bits(64)`	`[phy_read_double_word address]` is the 64-bit

		value starting at physical `[address]`
`phy_write_byte`	`bits(64) -> bits(8) -> unit`	`[phy_write_byte address value]` write byte `[value]` to physical `[address]`
`phy_write_half_word`	`bits(64) -> bits(16) -> unit`	`[phy_write_half address value]` write two bytes value to physical `[address]`
`phy_write_word`	`bits(64) -> bits(32) -> unit`	`[phy_write_word address value]` write word length value to physical `[address]`
`phy_write_double_word`	`bits(64) -> bits(64) -> unit`	`[phy_write_double_word address value]` write 64-bit value to physical `[address]`

5. Emulator

5.1. Development notes

To have rust-analyzer and compiled Sail RISC-V model in environment, users can set the `$EDITOR` environment and run `make dev` in sail-impl/boat directory.

5.2. Test notes

In “sail-impl/boat” directory, run `make` with following targets:

- run a demo with boat emulator: `make boat_demo`
- run a demo with spike emulator: `make spike_demo`
- run a demo with difftest: `make difftest_demo`

5.3. Logging usage

The `sail-ffi` crate is a library crate used for composing a emulator, thus logging should always use `Level::DEBUG` and `Level::TRACE` only. `Level::INFO` and error handling shoule be up lifting to emulator application side.

All architecture states change should be recorded with `TRACE` level event and contains `event_type` and `action` field for other software to easily deserialize to corresponding data type.

Current implementation contains following event type:

5.3.1. physical_memory

For “physical_memory” event type, current implmentation records following fields:

- action: a text value indicate current action to physical memory. Possible value: **“read”**, **“write”**.
- bytes: a integer value indicate the total bytes get operated on physical memory. Possible value: **1,2,4,8**.
- address: a 64-bit integer value indicate the start address of this action to the physical memory.
- data: a debug value in text indicating the value read from or write to the physical value.
- message: optional text value with human readable emulator status attached

5.3.2. arch_state

For “arch_state” event type, current implementation records following fields:

- action: a text value indicate current action to architecture states. Possible value: **“register_update”**.
- pc: a 64-bit integer value indicate the current PC of this action.
- reg_idx: if current action is “register_update”, `reg_idx` is a integer number represent the index of the changed register.
- data: if current action is “register_update”, `data` is a 64-bit integer showing the data ready to be written to register.
- message: optional text value with human readable emulator status attached

5.3.3. reset_vector

This event occurs when Sail model `PC` register get explicitly updated. For **reset_vector** event type, current implementation records following fields:

- new_addr: a 64-bit integer value indicate the new PC.

All event unrelated but useful for knowing `sail-ffi` running status shoule be logging with `Level::DEBUG`.

5.4. Sail FFI

To initialize a Sail model, drive it to process each instruction, and read model statistics, it is necessary to call generated functions and access Sail values at runtime from the emulator.

However, directly accessing C values from Rust can introduce multiple safety issues and complicate side-effect management.

To establish clear referencing and limitations, each emulator implementation must provide a sail_prelude.h C header file. This header file will function as an API-level language bridge between C and the emulator, specifying and restricting the functionalities the emulator side can and should utilize. All exposed functions and values must be kept private within a Rust module. Developers are required to manually write corresponding wrapper functions for all these FFI values and functions, rather than exposing them directly outside the Rust crate.

5.4.1. Required value from Sail

Signature	Type	Notes
`unit`	value type	Unit type defined at Sail side
`march_bits`	value type	Machine word length
`zstep :: unit -> unit`	function type	Run one step for fetch-decode-execute loop
`model_init :: void -> void`	function type	Initialize all registers