

Sail to Asciidoc

Table of Contents

Introduction	1
A RISC-V example	2
Documentation bundles	3
Potential workflows	4
JSON format	4
Sail command line flags	5
Syntax highlighting	5
Asciidoc commands	5
Examples	5
Working with multiple Sail files	5
Function body formatting	6
Documenting definitions with multiple clauses	8
Documenting mapping clauses	9
Anchors	10
Spans	11
Wavedrom encoding diagrams	11
Splitting Sail definitions	13

Introduction

This document describes how to incorporate [Sail](#) source code into ISA manuals written in asciidoc, using the [asciidocctor](#) tool. This is primarily aimed at producing documentation for the [sail-riscv](#) specification, but can in principle be used for any ISA description written in Sail.

ISA specifications typically describe the behaviour of instructions with a combination of prose, tables, diagrams, and pseudocode for each instruction. Sail is a language intended to support precise definition of real-world ISA semantics, enabling what was formally pseudocode to become executable and testable. It supports emulation for architectural compliance tests, and can generate definitions for theorem provers (such as [Coq](#) and [Isabelle](#)) to support mechanised reasoning over an ISA specification. It is a first-order imperative language, with a syntax inspired the existing pseudocode used in various architecture manuals, and with influences modern programming languages like Go, Swift, and Rust. The intent is it will therefore be accessible to engineers familiar with these languages or existing ISA pseudocode.

Until now, we have had some ad-hoc methods to incorporate Sail source into LaTeX documents, but here we present a more robust and extensible system for incorporating Sail definitions into Asciidoc manuals. There are two main components to this. First, there is an extension to Sail which produces *documentation bundles* — files indexing the contents of a Sail source file. Second, there is an [asciidocctor](#) plugin, which uses these bundles to include Sail source code in asciidoc manuals


without duplicating the code, which would invariably lead to it getting out of sync over time.

A RISC-V example

Here we show how we can take a RISC-V instruction, and generate an encoding diagram matching those used in the existing AsciiDoc port of the unpriv specification. We can also reference and typeset the Sail source from the authoritative model, without requiring that it is simply copy-pasted.

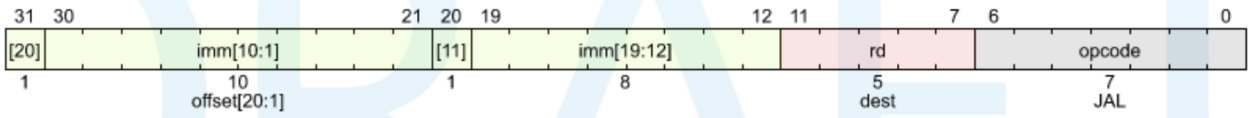
Consider the JAL instruction from the currently hand-written RISC-V unpriv specification.

convention uses 'x1' as the return address register and 'x5' as an alternate link register.



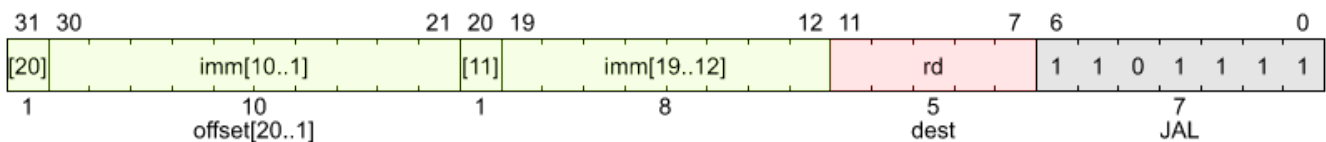
*The alternate link register supports calling millicode routines (e.g., those to save and restore registers in compressed code) while preserving the regular return address register. The register **x5** was chosen as the alternate link register as it maps to a temporary in the standard calling convention, and has an encoding that is only one bit different than the regular link register.*

Plain unconditional jumps (assembler pseudoinstruction `J`) are encoded as a JAL with `rd=x0`.



The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit `imm` to the register `rs1`, then setting the

We can generate the same diagram from the Sail source code that defines the JAL instruction:



Other than the opcode containing the actual bits, the automatically generated image is almost identical. To generate this, we had to enter:

```
include::sailwavedrom:encdec[left-clause="RISCV_JAL(_ , _)",right,type=mapping]
```

Which accesses the encoding information contained in the following Sail:

```
$[wavedrom _ offset[20..1] _ _ dest JAL]  
mapping clause encdec =  
    RISCV_JAL(imm : bits(20) @ 0b0, rd)  
    <->  
    imm[20] @ imm[10..1] @ imm[11] @ imm[19..12] @ rd @ 0b1101111
```

We can use an attribute in the Sail source to include information like the labels that would otherwise not be included.

We can also include the (body of the) Sail function that defines the JAL instruction using

```
sail::execute[clause="RISCV_JAL(_, _)",part=body,unindent]
```

This fetches the source code for the JAL instruction from the Sail model, and typesets it in our Asciidoc document as follows:

```
let t : xlenbits = PC + EXTS(imm);
/* Extensions get the first checks on the prospective target address. */
match ext_control_check_pc(t) {
  Ext_ControlAddr_Error(e) => {
    ext_handle_control_check_error(e);
    RETIRE_FAIL
  },
  Ext_ControlAddr_OK(target) => {
    /* Perform standard alignment check */
    if bit_to_bool(target[1]) & not(haveRVC())
    then {
      handle_mem_exception(target, E_Fetch_Addr_Align());
      RETIRE_FAIL
    } else {
      X(rd) = get_next_pc();
      set_next_pc(target);
      RETIRE_SUCCESS
    }
  }
}
```

Documentation bundles

Documentation bundles are produced by Sail using the `-doc` flag. This works like other Sail target flags, such as `-c` to generate C code, `-coq` for Coq definitions, and so on. For example, if we have three Sail files `a.sail`, `b.sail`, and `c.sail`, we could produce documentation for them using:

```
sail -doc a.sail b.sail c.sail
```

to produce only documentation for `c.sail` (which may still depend on the other files), we can use:

```
sail -doc a.sail b.sail c.sail -doc_file c.sail
```

The documentation bundle is a JSON file and will be placed in `sail_doc/doc.json`. The output directory can be changed using the `-o` option, so:

```
sail -doc a.sail b.sail c.sail -doc_file c.sail -o my_doc
```

will produce a file `my_doc/doc.json`. See [Sail command line flags](#) for other flags.

Potential workflows

The tooling has been carefully designed to not impose any particular workflow. You can:

- Generate the documentation bundle in advance and check it in. This means documentation contributors will not need Sail to build the documentation. This may lead the bundle becoming out of date, but there are various options here also:
 - You can check if the bundle is up-to-date each time you build the document
 - You can check if the bundle is up-to-date using some kind of continuous integration system
- Have a Makefile (or other build tool) generate the bundle every time the documentation is built.

These are only two possibilities. There are likely many others.

JSON format

The first few keys in the JSON file give information about the documented Sail files, and the state of the repository they are in. These keys can be accessed by custom build tooling to check if the bundle is up to date. As an example:

```
{
  "version": 1,
  "git": {
    "commit": "773a19d17432a1c60cc95a87a587c8255bef9e75",
    "dirty": true
  },
  "embedding": "plain",
  "hashes": { "doc.sail": { "md5": "45ee16b971a5fc084bea556d83f27aff" } },
}
```

- The `version` key exists so the bundle format can be updated in the future. It is currently always set to 1.
- The `git` key contains the commit hash of the repository within which the `sail` command that produced the documentation bundle was produced. It also contains a flag that is true if the working tree has uncommitted changes. See https://mirrors.edge.kernel.org/pub/software/scm/git/docs/gitglossary.html#def_dirty for details.
- The `embedding` field tells us how other subsequent fields are encoded. See the `-doc_embed` option below for details.
- The `hashes` field contains a checksum for each Sail file included in the documentation bundle. Like the git commit hash it can be used by tools to check whether the bundle is up-to-date.

The rest of the file contains information for all the documented Sail definitions.

Sail command line flags

- `-doc` — Tells Sail to generate documentation.
- `-doc_file <file>` — Include Sail definitions in `file` in the generated documentation bundle. This option can be passed multiple times. If `-doc_file` is not passed then all files provided to Sail will have documentation generated for them.
- `-doc_embed <plain|base64>` — This option embeds the source code directly within the documentation bundle rather than referencing it from an external file. If `-doc_embed plain` is used then the source and comments is included as-is (with appropriate escaping to be included in a JSON file). With `-doc_embed base64` the source and comments are stored in the JSON as base64 encoded strings. `-doc_embed` is useful if you documentation is separate from the Sail source you are documenting.
- `-doc_compact` — By default the JSON output is pretty-printed with indentation. When this option is used the JSON documentation bundle is printed in a compact form, omitting all unnecessary spaces.
- `-doc_format <format>` — This option controls the format for the output. Currently supported options are `adoc` and `asciidoc` (the default) which are the same and both output suitable for the Sail to Asciidoc plugin. Eventually `latex` will also be allowed once the older Sail to Latex documentation generation has been ported to the new documentation system written for this plugin.
- `-doc_bundle <file>` The name of the generated documentation bundle file. By default this is `doc.json`. It will be placed in the folder specified by the output `-o` option, which defaults to a folder named `sail_doc`.

Syntax highlighting

The Sail to Asciidoc plugin provides a lexer for rouge, an extensible Ruby syntax highlighter supported by asciidoctor and asciidoctor-pdf. If another highlighter with a fixed set of supported languages is used, everything will still work, but without highlighting.

Asciidoc commands

The Sail to Asciidoc repository contains several examples demonstrating the various commands in the `examples` subdirectory. These examples are included below.

Examples

Working with multiple Sail files

The documentation bundle is specified using an asciidoc attribute, for example:

```
:sail-doc: sail_doc/doc.json
```

Each command takes a `from` parameter which specifies which bundle to pull the source from. This defaults to `from=sail-doc` if left unspecified. In the following examples this is used for each separate example, so you will see `from=<file>` which corresponds to a documentation bundle generated in `examples/<file>.json`.

Function body formatting

In this example, we show how to include the source code from a Sail function.

Sail source

```
default Order dec
$include <prelude.sail>

/*! We can choose to display this entire function, or we can just
display its body. There are various options to control the body
formatting. */
function main() -> unit = {
  let str = "Hello, World!";
  print_endline(str);
  print_endline("Goodbye!")
}
```

Result

We can choose to display this entire function, or we can just display its body. There are various options to control the body formatting.

For example, to format the entire function:

```
sail::main[from=function-body]
```

```
function main() -> unit = {
  let str = "Hello, World!";
  print_endline(str);
  print_endline("Goodbye!")
}
```

If we want to reference an included function, we can use an AsciiDoc cross-reference. The `sail` macro will generate an id using the name of the Sail function and the `from` attribute. In this example, we can use `<<function-body-main>>` to reference the function which results in `[function-body-main]`. It can also be given a name in the standard AsciiDoc way using `<<function-body-main,main>>`, which results in `main`.

We can format just the body of the function using `part=body`:

```
sail::main[from=function-body,part=body]
```

```
let str = "Hello, World!";  
print_endline(str);  
print_endline("Goodbye!")
```

The body can be unindented the block using either `unindent` or `dedent`:

```
sail::main[from=function-body,part=body,dedent]
```

```
let str = "Hello, World!";  
print_endline(str);  
print_endline("Goodbye!")
```

We can also trim leading and trailing whitespace using `trim` or `strip` (although this is not useful here):

```
sail::main[from=function-body,part=body,trim]
```

```
let str = "Hello, World!";  
print_endline(str);  
print_endline("Goodbye!")
```

Rather than using the `sail` block macro directly we can also use an include macro, like so:

```
[source,sail]  
----  
include::sail:main[from=function-body]  
----
```

```
function main() -> unit = {  
  let str = "Hello, World!";  
  print_endline(str);  
  print_endline("Goodbye!")  
}
```

The advantage of using the include macro is it lets us include multiple definitions within the same asciidoc source block, at the expense of being more verbose. It can also be used in places where the asciidoc processor doesn't like seeing a block macro, but would allow a block.

A disadvantage of using the include block is that it doesn't automatically generate an anchor for us

to cross reference. An anchor would instead have to be manually added to the block.

Documenting definitions with multiple clauses

In this example we show how to document *scattered functions*, a Sail feature that allows us to split apart the various cases of the function into multiple *clauses*. It may seem hard to document these, as the function clauses share the same name, and are only distinguished by their *pattern*.

Sail source

```
default Order dec
$include <prelude.sail>

/* Pretend we have accessors for reading and writing registers */
val rX : bits(5) -> bits(32)

val wX : (bits(5), bits(32)) -> unit

overload X = {rX, wX}

scattered union Instr

val execute : Instr -> unit

union clause Instr = Add : (bits(5), bits(5), bits(5))

function clause execute Add(rd, rx, ry) = {
  X(rd) = add_bits(X(rx), X(ry))
}

union clause Instr = Sub : (bits(5), bits(5), bits(5))

function clause execute Sub(0b000000, rx, ry) = {
  ()
}

function clause execute Sub(rd, rx, ry) = {
  X(rd) = sub_bits(X(rx), X(ry))
}
```

Result

To include just the **Add** clause, we can use the following command:

```
sail::execute[from=clauses,clause="Add(_, _, _)"]
```

which produces:


```
function clause execute Add(rd, rx, ry) = {
  X(rd) = add_bits(X(rx), X(ry))
}
```

The `clause` attribute allows us to match on the pattern, using syntax similar to that found in Sail. The underscore is the *wildcard* pattern, that allows us to match anything.

The `Sub` instruction has two function clauses. For the first one where the destination register is `0b000000` we can include it using:

```
sail::execute[from=clauses,clause="Sub(0b000000, _, _)"]
```

which produces:

```
function clause execute Sub(0b000000, rx, ry) = {
  ()
}
```

The next clause we can include similarly, like so:

```
sail::execute[from=clauses,clause="Sub(rd, _, _)"]
```

which produces:

```
function clause execute Sub(rd, rx, ry) = {
  X(rd) = sub_bits(X(rx), X(ry))
}
```

Documenting mapping clauses

Sail allows us to specify bi-directional functions, called *mappings*. These can be broken into multiple scattered functions in the same way that functions can (see the `clauses.adoc` example for details).

Sail source

```
default Order dec
$include <prelude.sail>

scattered union Instr

val encdec : Instr <-> bits(32)
```

```
union clause Instr = Add : (bits(5), bits(5), bits(5))

mapping clause encdec =
  Add(rd, rx, ry) <-> 0xFFFF @ rd : bits(5) @ 0b1 @ rx : bits(5) @ ry : bits(5)
```

Result

However in this case, we can select the clause we want to document by matching on either the left or the right pattern of the mapping.

```
sail::encdec[from=mapping-clauses,left-clause="Add(_, _, _)",type=mapping]
```

which produces:

```
mapping clause encdec =
  Add(rd, rx, ry) <-> 0xFFFF @ rd : bits(5) @ 0b1 @ rx : bits(5) @ ry : bits(5)
```

NOTE

The matching language in the `left-clause` and `right-clause` attributes is a subset of the Sail pattern language that includes constructor patterns, identifiers, wildcards, and binary literals.

Here we see the usage of the `type` attribute. By default the `sail` macro will include functions (i.e. `type=function`), but here we want to reference a mapping. The argument of `type` is the toplevel Sail keyword for the type of definition we want to include, so it can be `function`, `mapping`, `val`, `type`, `register`, `let`. We use the `type=type` for all top-level Sail type definitions, such as unions, enums, and structs.

Anchors

An *anchor* is a special Sail attribute `$anchor` that we can attach documentation comments to. This allows us to include comments that are not otherwise associated with any toplevel Sail definition. The anchor directive provides the name we use to reference the comment in the asciidoc source.

Sail source

```
default Order dec
$include <string.sail>

/*! This demonstrates the use of an _anchor_ directive to provide a
name to a free standing documentation comment without a toplevel
definition. */
$anchor my_comment

/*! Documentation comment for `main`. */
function main() -> unit = {
```

```
    print_endline("Hello, World!")
}
```

Result

```
include::sailcomment:my_comment[from=anchor,type=anchor]

include::sailcomment:main[from=anchor]

sail::main[from=anchor]
```

This demonstrates the use of an *anchor* directive to provide a name to a free standing documentation comment without a toplevel definition.

Documentation comment for `main`.

```
function main() -> unit = {
    print_endline("Hello, World!")
}
```

Spans

What if we want to include some arbitrary span of Sail source rather than select definitions? This can be achieved using `$span` directives in Sail.

Sail source

```
include::span.sail
```

Result

```
sail::PREAMBLE[from=span,type=span]
```

```
default Order dec

$include <prelude.sail>
```

Wavedrom encoding diagrams

Sail source

```
default Order dec
#include <prelude.sail>

scattered union Instr

val encdec : Instr <-> bits(32)

union clause Instr = Add : (bits(5), bits(5), bits(5))

$[wavedrom REG3 dest ADD input input]
mapping clause encdec =
  Add(rd, rx, ry) <-> 0xFFFF @ rd : bits(5) @ 0b1 @ rx : bits(5) @ ry : bits(5)

union clause Instr = Sub : (bits(5), bits(5), bits(5))

$[wavedrom REG3 _ SUB _ _]
mapping clause encdec =
  Sub(rd, rx, ry) <-> 0xFFFF @ rd : bits(5) @ 0b1 @ rx : bits(5) @ ry : bits(5)

union clause Instr = Xor : (bits(5), bits(5), bits(5))

mapping clause encdec =
  Xor(rd, rx, ry) <-> 0xFFFE @ 0b1 @ rd : bits(5) @ rx : bits(5) @ ry : bits(5)
```

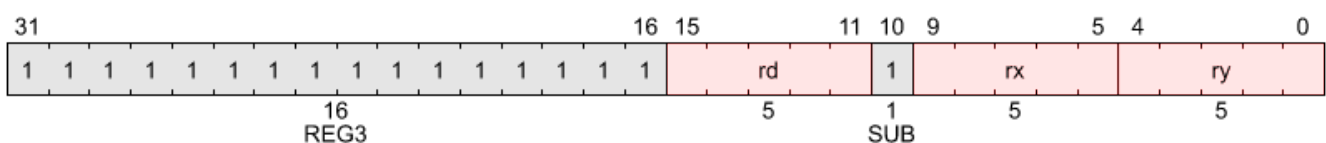
Result

The diagram for the Add clause:

```
include::sailwavedrom:encdec[from=wavedrom,left-clause="Add(_ , _ ,
_)",type=mapping,right]
```

The diagram for the Sub clause. Note how we can use underscores to skip labels:

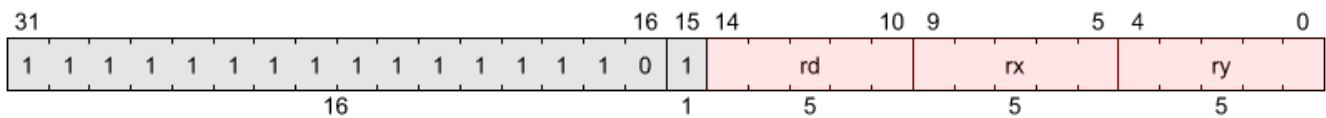
```
include::sailwavedrom:encdec[from=wavedrom,left-clause="Sub(_ , _ ,
_)",type=mapping,right]
```



The `$[wavedrom labels]` attribute can be omitted, as with `Xor`:

```
include::sailwavedrom:encdec[from=wavedrom,left-clause="Xor(_ , _ ,
```

```
_)",type=mapping,right]
```



Splitting Sail definitions

Sometimes we have a Sail function that corresponds to multiple functions we want to document. Here we can split the function by applying *constant propagation* with the `split` attribute in Sail. This works when the function has an enumeration as an argument.

WARNING

This feature is somewhat experimental as it relies on calling Sail's constant propagation pass and pretty printer during document bundle preparation, neither of which were really intended for this use case, so while it works for simple functions you might run into places where it fails for more complex inputs. Notice also that this happens after overloads have been resolved, so we see `rX` and `wX` in the below examples, rather than the overload `X`.

Sail source

```
default Order dec
$include <prelude.sail>

val rX : bits(5) -> bits(32)

val wX : (bits(5), bits(32)) -> unit

overload X = {rX, wX}

enum Op = ADD | SUB

$[split op]
function instr(rd: bits(5), rs1: bits(5), rs2: bits(5), op: Op) -> unit = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result: bits(32) = match op {
    ADD => add_bits(rs1_val, rs2_val),
    SUB => sub_bits(rs1_val, rs2_val),
  };

  X(rd) = result
}
```

Result

As an example, here we can take the above `instr` which implements both `ADD` and `SUB` and generate just the `ADD` case:

```
sail::instr[from=split,split=ADD]
```

which produces:

```
let rs1_val = rX(rs1);
let rs2_val = rX(rs2);
let result : bits(32) = add_bits(rs1_val, rs2_val);
wX(rd, result)
```

Alternatively, for the `SUB` case:

```
sail::instr[from=split,split=SUB]
```

produces

```
let rs1_val = rX(rs1);
let rs2_val = rX(rs2);
let result : bits(32) = sub_bits(rs1_val, rs2_val);
wX(rd, result)
```